# Table of Contents

## Section I: Getting Started with Vitis

### Chapter 1: Vitis Software Platform Release Notes
- What's New........................................................................................................................... 11
- Supported Platforms........................................................................................................... 11
- Changed Behavior................................................................................................................ 12
- Known Issues........................................................................................................................ 12

### Chapter 2: Installation
- Installation Requirements................................................................................................... 13
- Vitis Software Platform Installation................................................................................... 15

### Chapter 3: Introduction to the Vitis Environment for Acceleration
- Accelerated Flow Application Development Using the Vitis Software Platform........... 19
- Execution Model................................................................................................................... 21
- Data Center Application Acceleration Development Flow.............................................. 22
- Embedded Processor Application Acceleration Development Flow.............................. 24
- Build Targets......................................................................................................................... 27
- Tutorials and Examples....................................................................................................... 27

### Chapter 4: Methodology for Accelerating Applications with the Vitis Software Platform
- Introduction.......................................................................................................................... 29
- Methodology for Architecting a Device Accelerated Application................................... 32
- Methodology for Developing C/C++ Kernels.................................................................... 45

## Section II: Developing Applications

### Chapter 5: Programming Model
- Device Topology.................................................................................................................. 60
Kernel Properties................................................................................................................................. 60

**Chapter 6: Host Application** ............................................................................................................... 64
  Setting Up the Runtime Environment.................................................................................................. 64
  Executing Commands in the FPGA...................................................................................................... 69
  Post-Processing and FPGA Cleanup..................................................................................................... 81
  Summary................................................................................................................................................ 82

**Chapter 7: C/C++ Kernels** .................................................................................................................. 83
  Process Execution Modes...................................................................................................................... 83
  Data Types............................................................................................................................................. 85
  Interfaces................................................................................................................................................ 87
  Loops...................................................................................................................................................... 91
  Dataflow Optimization............................................................................................................................ 97
  Array Configuration............................................................................................................................... 100
  Function Inlining.................................................................................................................................. 104
  Summary................................................................................................................................................ 104

**Chapter 8: RTL Kernels** .................................................................................................................... 106
  Requirements of an RTL Kernel........................................................................................................... 106
  RTL Kernel Development Flow............................................................................................................ 109
  Design Recommendations for RTL Kernels........................................................................................ 115

**Chapter 9: Streaming Data Transfers** ............................................................................................... 119
  Streaming Data Transfers between Kernels (K2K)................................................................................ 119
  Free-Running Kernel............................................................................................................................. 120

**Chapter 10: Best Practices for Acceleration with Vitis** ..................................................................... 123

**Section III: Building and Running the Application** .......................................................................... 125

**Chapter 11: Setting Up the Vitis Environment** ................................................................................ 126

**Chapter 12: Build Targets** ............................................................................................................... 127
  Software Emulation............................................................................................................................... 127
  Hardware Emulation............................................................................................................................. 128
  System Hardware Target...................................................................................................................... 129

**Chapter 13: Building the Host Program** ........................................................................................... 130
Chapter 14: Building the Device Binary

Chapter 15: Packaging the System

Chapter 16: Output Directories of the v++ Command

Chapter 17: Running Emulation

Chapter 18: Running the Application Hardware Build

Section IV: Profiling, Optimizing, and Debugging the Application
Chapter 20: Optimizing the Performance .............................................................. 224
Host Optimization ................................................................................................. 224
Kernel Optimization .............................................................................................. 232
Topological Optimization ...................................................................................... 260

Chapter 21: Debugging Applications and Kernels .............................................. 270
Debugging Flows ................................................................................................... 270
Debugging in Software Emulation ........................................................................ 271
Debugging in Hardware Emulation ....................................................................... 278
Debugging During Hardware Execution ............................................................... 283
Debugging on Embedded Processor Platforms ................................................... 306
Example of Command Line Debugging ............................................................... 311

Section V: Vitis Environment Reference Materials ............................................ 314

Chapter 22: Vitis Compiler Command ................................................................. 316
Vitis Compiler General Options .......................................................................... 316
--advanced Options .............................................................................................. 332
--clock Options ..................................................................................................... 338
--connectivity Options ......................................................................................... 342
--debug Options ................................................................................................... 345
--hls Options ......................................................................................................... 346
--linkhook Options ............................................................................................... 349
--package Options ................................................................................................ 350
--profile Options .................................................................................................. 354
--vivado Options .................................................................................................. 357
Vitis Compiler Configuration File .......................................................................... 360
Using the Message Rule File ................................................................................ 362

Chapter 23: emconfigutil Utility .......................................................................... 364

Chapter 24: kernelinfo Utility ............................................................................... 366
Kernel Definition .................................................................................................... 367
Arguments .............................................................................................................. 367
Ports ........................................................................................................................ 368

Chapter 25: launch_emulator Utility .................................................................. 369
Chapter 26: manage_ipcache Utility .......................................................... 373

Chapter 27: package_xo Command ............................................................ 374

Chapter 28: platforminfo Utility ............................................................... 377
  Basic Platform Information ................................................................. 378
  Hardware Platform Information ......................................................... 379
  Interface Information ...................................................................... 379
  Clock Information ........................................................................... 379
  Valid SLRs ....................................................................................... 380
  Resource Availability ..................................................................... 380
  Memory Information .................................................................... 381
  Feature ROM Information ............................................................. 382
  Software Platform Information .................................................... 382
Platforminfo for xilinx_zcu104_base_202010_1 .................................... 383

Chapter 29: RTL Kernel XML File ............................................................. 385

Chapter 30: xbutil Utility ....................................................................... 388
  clock ........................................................................................... 389
  dmatest ...................................................................................... 391
  dump ......................................................................................... 392
  m2mtest .................................................................................... 397
  mem --read .............................................................................. 399
  mem --write ............................................................................ 400
  p2p ......................................................................................... 402
  program ................................................................................... 403
  query ....................................................................................... 404
  reset ....................................................................................... 411
  scan (xbutil) ............................................................................ 412
  status ..................................................................................... 414
  top ......................................................................................... 417
  validate ................................................................................... 419
  version .................................................................................... 420

Chapter 31: xbmgmt Utility ................................................................. 422
  config ....................................................................................... 423
  flash ......................................................................................... 424
Chapter 32: xclbinutil Utility
xclbin Information.................................................................436
Hardware Platform Information.....................................436
Clocks..................................................................................437
Memory Configuration..........................................................437
Kernel Information.................................................................438
Tool Generation Information........................................439

Chapter 33: xrt.ini File.................................................................441

Chapter 34: HLS Pragmas..........................................................448

Section VI: Using the Vitis Analyzer........................................450

Chapter 35: Working with Reports........................................454

Chapter 36: Vitis Analyzer GUI and Window Manager.................456
Diff Two Text Files.................................................................458
Cross-Probing between Reports..............................................459

Chapter 37: Platform and System Diagrams.................................462

Chapter 38: AI Engine Graphs and Arrays.................................465

Chapter 39: Configuring the Vitis Analyzer.................................467

Chapter 40: Setting Guidance Thresholds..................................469

Chapter 41: Creating an Archive File........................................472

Section VII: Using the Vitis IDE................................................474

Chapter 42: Vitis Command Options........................................475

Chapter 43: Creating a Vitis IDE Project....................................476
Launch a Vitis IDE Workspace................................................476
Create an Application Project.................................................................................................. 477
Understanding the Vitis IDE.................................................................................................. 484
Adding Sources......................................................................................................................... 485
Working in the Project Editor View......................................................................................... 488
Working in the Assistant View................................................................................................. 490
Output Directories from the Vitis IDE..................................................................................... 492

Chapter 44: Building the System.......................................................................................... 497
Vitis IDE Guidance View.......................................................................................................... 498
Working with Vivado Tools from the Vitis IDE....................................................................... 499

Chapter 45: Vitis IDE Debug Flow....................................................................................... 501
Using the Standalone Debug Flow.......................................................................................... 503
vitis -debug Command Line................................................................................................... 505

Chapter 46: Configuring the Vitis IDE................................................................................ 509
Vitis Project Settings............................................................................................................... 509
Vitis Build Configuration Settings......................................................................................... 511
Vitis Hardware Function Settings.......................................................................................... 512
Vitis Binary Container Settings.............................................................................................. 514
Vitis Toolchain Settings.......................................................................................................... 516
Vitis Run and Debug Configuration Settings......................................................................... 521

Chapter 47: Project Export and Import.............................................................................. 526
Export a Vitis Project............................................................................................................... 526
Import a Vitis Project............................................................................................................... 527
Import Projects from Git......................................................................................................... 528

Chapter 48: Getting Started with Examples...................................................................... 531
Installing Examples and Libraries......................................................................................... 531
Using Local Copies................................................................................................................ 533

Chapter 49: RTL Kernel Wizard......................................................................................... 535
Launch the RTL Kernel Wizard............................................................................................... 535
Using the RTL Kernel Wizard............................................................................................... 536
Using the RTL Kernel Project in Vivado IDE........................................................................ 544

Section VIII: Using Vitis Embedded Platforms.................................................................... 553
Getting Started with Vitis

Navigating Content by Design Process

Xilinx® documentation is organized around a set of standard design processes to help you find relevant content for your current development task. All Versal™ ACAP design process Design Hubs can be found on the Xilinx.com website. This document covers the following design processes:

- **Host Software Development**: Developing the application code, accelerator development, including library, XRT, and Graph API use.

This section contains the following chapters:

- Vitis Software Platform Release Notes
- Installation
- Introduction to the Vitis Environment for Acceleration
- Methodology for Accelerating Applications with the Vitis Software Platform
Vitis Software Platform Release Notes

This section contains information regarding the features and updates of the Vitis™ software platform in this release. It also contains information regarding the features and updates of the Vitis software platform for Versal™ AI Engine development.

What's New

For information about what's new in this version of the Vitis™ unified software development platform, see the Vitis What's New Page.

Supported Platforms

Data Center Accelerator Cards


Refer to Alveo Data Center Accelerator Card Platforms User Guide (UG1120) for specifications of each accelerator card and available target platforms. The Getting Started section for each accelerator card has information for deploying your applications on that card.

Refer to Installing Xilinx Runtime and Platforms for more information on setting up XRT and platforms.

Embedded Platforms

Embedded platforms available for use with the Vitis core development kit can be found at the Embedded Platforms download page. Embedded processor platforms such as the Zynq UltraScale + MPSoC ZCU104/ZCU102 base platform as well as Zynq-7000 base platforms can be optionally used for both the Vitis application acceleration development flow, and the Vitis embedded software development flow.
Versal Platform for AI Engine Development

The VCK190 platform is available for use with the Vitis application acceleration development flow, as described in Versal ACAP AI Engine Programming Environment User Guide (UG1076). The platform enables development of designs that include:

- AI Engine graphs and kernels
- Programmable Logic kernels
- Host application targeting the Linux or a bare metal OS running on the Arm processor in the Versal device.

Changed Behavior

The following table specifies differences between this release and prior releases that impact behavior or flow when migrating.

Table 1: Changed Behavior Summary

<table>
<thead>
<tr>
<th>Area</th>
<th>Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vitis IDE</td>
<td>Previously, accelerated application projects had a system project that contained the application project. This application project contained all the source code and build instructions in a single project. There is now a new project hierarchy that creates a top-level system project with separate projects for each element: a host application project, a hardware kernels project, and a hw_link project. Instead of creating an application project, you now need to create a PL kernel project. The top-level system project is automatically created as part of this process. Existing projects from earlier releases must to be updated to the new project hierarchy. When you open the older projects, you will be prompted to upgrade the project to the new structure. The tool will handle this upgrade for you. It is recommended to back up your existing projects before the upgrade.</td>
</tr>
<tr>
<td>Vitis compiler</td>
<td>The --profile_kernel command has become the --profile command.</td>
</tr>
<tr>
<td></td>
<td>The --dk command has become the --debug command.</td>
</tr>
<tr>
<td></td>
<td>The --package command previously did not require renaming the xclbin file, but now either renames the output file to a.xclbin by default, or names it according to the -o option.</td>
</tr>
<tr>
<td>Vitis HLS</td>
<td>The DISAGGREGATE pragma or directive previously could not be applied to structs on the interface. Disaggregating structs on the interface is now supported.</td>
</tr>
<tr>
<td></td>
<td>The ARRAY_PARTITION pragma and directive was not previously supported for arrays defined on the function interface. It is supported in this release.</td>
</tr>
</tbody>
</table>

Known Issues

Known issues for the Vitis software platform are available in AR#73646.
Installation Requirements

The Vitis™ software platform consists of an integrated development environment (IDE) for interactive project development, and command-line tools for scripted or manual application development. The Vitis software platform also includes the Vivado® Design Suite for implementing the kernel on the target device, and for developing custom hardware platforms.

Note: Windows OS support is limited to the Vitis embedded software development flow. The acceleration features in the Vitis software platform are not supported.

Some requirements listed here are only required for software acceleration features, but not for embedded software development features. Xilinx recommends installing all the required packages to have the best experience with the Vitis software platform.

To install and run on a computer, your system must meet the following minimum requirements.

Note: Support for RHEL/CentOS 7.4 and 7.5 will end in the 2021.1 release.

Table 2: Application Acceleration Development Flow Minimum System Requirements

<table>
<thead>
<tr>
<th>Component</th>
<th>Requirement</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Operating System</strong></td>
<td><strong>Development (Build Machine OS)</strong></td>
</tr>
<tr>
<td></td>
<td>Linux, 64-bit:</td>
</tr>
<tr>
<td></td>
<td>• RHEL/CentOS 7.4, 7.5, 7.6, 7.7, 7.8, 8.1, 8.2</td>
</tr>
<tr>
<td></td>
<td>• Ubuntu 16.04.5 LTS, 16.04.6 LTS, 18.04.1 LTS, 18.04.2 LTS, 18.04.3 LTS,</td>
</tr>
<tr>
<td></td>
<td>18.04.4 LTS, 20.04 LTS</td>
</tr>
<tr>
<td></td>
<td>• Amazon Linux 2 AL2 LTS</td>
</tr>
<tr>
<td></td>
<td>For edge acceleration (embedded platforms):</td>
</tr>
<tr>
<td></td>
<td>• PetaLinux 2020.2</td>
</tr>
<tr>
<td><strong>System Memory</strong></td>
<td>For Alveo cards: 64 GB (80 GB is recommended)</td>
</tr>
<tr>
<td></td>
<td>For embedded: 32 GB</td>
</tr>
<tr>
<td><strong>Internet Connection</strong></td>
<td>Required for downloading drivers and utilities.</td>
</tr>
</tbody>
</table>
Table 2: Application Acceleration Development Flow Minimum System Requirements (cont’d)

<table>
<thead>
<tr>
<th>Component</th>
<th>Requirement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hard disk space</td>
<td>100 GB</td>
</tr>
</tbody>
</table>

Table 3: AI Engine Development Flow Minimum System Requirements

<table>
<thead>
<tr>
<th>Component</th>
<th>Requirement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operating System</td>
<td>Linux, 64-bit:</td>
</tr>
<tr>
<td></td>
<td>• RHEL/CentOS 7.4, 7.5, 7.6, 7.7, 7.8, 8.1, 8.2</td>
</tr>
<tr>
<td></td>
<td>• Ubuntu 16.04.5 LTS, 16.04.6 LTS, 18.04.1 LTS, 18.04.2 LTS, 18.04.3 LTS,</td>
</tr>
<tr>
<td></td>
<td>18.04.4 LTS, 20.04 LTS</td>
</tr>
<tr>
<td>System Memory</td>
<td>64 GB (80 GB is recommended)</td>
</tr>
<tr>
<td>Internet Connection</td>
<td>Required for downloading drivers and utilities.</td>
</tr>
<tr>
<td>Hard disk space</td>
<td>100 GB</td>
</tr>
</tbody>
</table>

OpenCL Installable Client Driver Loader

The Vitis™ environment supports the OpenCL Installable Client Driver (ICD) extension (cl_khr_icd). This extension allows multiple implementations of OpenCL to co-exist on the same system. The ICD Loader acts as a supervisor for all installed platforms, and provides a standard handler for all API calls.

Applications can choose an OpenCL platform from the list of installed platforms. Based on the platform ID specified by the application, the ICD dispatches the OpenCL host calls to the right runtime.

TIP: This is an optional package to install if your system has or uses multiple versions of the OpenCL library.

Xilinx does not provide the OpenCL ICD library, so the following should be used to install the library on your system.

Ubuntu

On Ubuntu the ICD library is packaged with the distribution. Install the following packages:

```bash
sudo apt-get install ocl-icd-libopencl1
sudo apt-get install opencl-headers
sudo apt-get install ocl-icd-opencl-dev
```
RHEL/CentOS

For RHEL/CentOS use EPEL to install the following packages:

```bash
sudo yum install ocl-icd
sudo yum install ocl-icd-devel
sudo yum install opencl-headers
```

**Note:** Refer to [https://fedoraproject.org/wiki/EPEL](https://fedoraproject.org/wiki/EPEL) for information on installing EPEL if needed.

---

**Vitis Software Platform Installation**

**Installing the Vitis Software Platform**

Ensure your system meets all requirements described in Installation Requirements.

💡 **TIP:** To reduce installation time, disable anti-virus software and close all open programs that are not needed.

1. Go to the Xilinx Downloads Website.
2. Download the installer for your operating system.
3. Run the installer, which opens the Welcome page of the Xilinx Unified 2020.2 Installer.
4. Click **Next** to open the Select Install Type page of the Installer.
5. Enter your Xilinx user account credentials, and then select **Download and Install Now**.
6. Click **Next** to open the Accept License Agreements page of the Installer.
7. Accept the terms and conditions by clicking each **I Agree** check box.
8. Click **Next** to open the Select Product to Install page of the Installer.
9. Select **Vitis** and click **Next** to open the Vitis Unified Software Platform page of the Installer.
10. Customize your installation by selecting design tools and devices (optional).

🌟 **IMPORTANT!** Do not deselect the following option. It is required for installation.

- Devices → Install devices for Alveo and Xilinx Edge acceleration platforms

**Note:** Both the Vitis tools and Vivado Design Suite are installed as part of the Vitis Unified Software Platform. You do not need to separately install Vivado tools. You can also install System Generator and Model Composer if needed.

11. Click **Next** to open the Select Destination Directory page of the Installer
12. Specify the installation directory, review the location summary, review the disk space required to insure there is enough space, and click **Next** to open the Installation Summary page of the Installer.
13. Click **Install** to begin the installation of the software.

**IMPORTANT!** Do not deselect the following option. It is required for installation.

- Devices → Install devices for Alveo and Xilinx Edge acceleration platforms

**Note:** Both the Vitis tools and Vivado Design Suite are installed as part of the Vitis Unified Software Platform. You do not need to separately install Vivado tools. You can also install System Generator and Model Composer if needed.

After a successful installation of the Vitis software, a confirmation message is displayed, with a prompt to run the `installLibs.sh` script.

1. Locate the script at: `<install_dir>/Vitis/<release>/scripts/installLibs.sh`, where `<install_dir>` is the location of your installation, and `<release>` is the installation version.
2. Run the script using `sudo` privileges as follows:

   ```bash
   sudo installLibs.sh
   ```

The command installs a number of necessary packages for the Vitis tools based on the OS of your system.

**IMPORTANT!** Pay attention to any messages returned by the script. You might need to install any missing packages manually.

### Installing Xilinx Runtime and Platforms

Xilinx Runtime (XRT) is implemented as a combination of user-space and kernel driver components. XRT supports Alveo PCIe-based cards, as well as Versal and Zynq UltraScale+ MPSoC-based embedded system platforms, and provides a software interface to Xilinx programmable logic devices.

You must install XRT for use in the Vitis application acceleration development flow. You do not need to reinstall it for every additional platform you choose to download.

**IMPORTANT!** XRT installation uses standard Linux RPM and Linux DEB distribution files, and root access is required for all software and firmware installations.

- `<rpm-dir>` or `<deb-dir>` is the directory where you downloaded the packages to install.

To download and install the XRT package for your operating system, do the following.

2. From the **Getting Started** page, you can choose to download the XRT package for a specific Alveo Data Center accelerator card, or for Embedded Platforms.
**Note:** Installing XRT is not required when targeting Arm®-based embedded platforms: Vitis compiler has its own copy of `xclbinutil` for hardware generation, and for software compilation, you can use the XRT from the `sysroot`. Look for **Common images for Embedded Vitis platforms** on the downloads page.

After choosing the platform, you will be redirected to a web site with instructions for downloading XRT and the required files for the selected platform.

3. Follow the directions to install XRT and your selected platform.

**TIP:** The instructions for installing the Alveo Data Center accelerator cards are provided on the platform download page. Instructions for installing the Embedded Platforms can be found in the following section.

---

**Installing Embedded Platforms**

Embedded platforms are available to download from the [Vitis Embedded Platforms download page](https://www.xilinx.com) for use in the Vitis unified software platform. For the Vitis embedded software development flow, you can use embedded platforms with Linux, standalone/bare metal, or RTOS domains. To support the Vitis application acceleration development flow, embedded platforms must run Linux, with XRT integrated into the `rootfs`. Some of the supported platforms are listed below. A complete list can be found on the downloads page.

- **xilinx_zcu102_base_202020_1:** Provides the platform definition, XRT drivers, and shared libraries for the ZCU102 Zynq UltraScale+ MPSoC based platform.

- **xilinx_zcu102_base_dfx_202020_1:** Provides the platform definition, XRT drivers, and shared libraries for the ZCU102 Zynq UltraScale+ MPSoC based platform with dynamic function exchange.

- **xilinx_zc702_base_202020_1:** Provides the platform definition, XRT drivers, and shared libraries for the ZC702 Zynq-7000 SoC based platform.

- **xilinx_zc706_base_202020_1:** Provides the platform definition, XRT drivers, and shared libraries for the ZC706 Zynq®-7000 SoC based platform.

- **xilinx_vck190_base_202020_1:** Provides the platform definition, XRT drivers, and shared libraries for the Versal based platform.

To install a platform, download the zip file and extract it into `/opt/xilinx/platforms`, or extract it into a separate location and add that location to the `PLATFORM_REPO_PATHS` environment variable.

Embedded platforms require a `sysroot` to cross-compile the host application for the Vitis application acceleration flow. Look for the **Common images for Embedded Vitis platforms** block on the downloads page, and download and extract the common image for your platform architecture.
Running `sdk.sh` extracts and installs the `sysroot`. This package also provides a pre-compiled kernel image and `rootfs`.

You can add the `sysroot` to a Makefile for your command line project, or add it to your application project in the Vitis IDE. For example, in your Makefile point `<SYSROOT>` to `/<install_path>/aarch64-xilinx-linux`, which is generated when running `sdk.sh`.

**Setting Up the Environment to Run the Vitis Software Platform**

To configure the environment to run the Vitis software platform, run the following scripts in a command shell to set up the tools to run in that shell:

```
#setup XILINX_VITIS and XILINX_VIVADO variables
source <Vitis_install_path>/Vitis/2020.2/settings64.sh
#setup XILINX_XRT
source /opt/xilinx/xrt/setup.sh
```

**TIP:** `.csh` scripts are also provided.

This sets up the tools for the Vitis application acceleration development flow, the Vitis embedded software development flow, and the AI Engine tools for development on Versal AI Engine devices.

To use any platforms you have downloaded as described in Installing Xilinx Runtime and Platforms, set the following environment variable to point to the location of the platforms:

```
export PLATFORM_REPO_PATHS=<path to platforms>
```

This identifies the location of platform files for the tools, and makes them accessible to your design projects.
Introduction to the Vitis Environment for Acceleration

Accelerated Flow Application Development Using the Vitis Software Platform

The Vitis™ unified software platform is a new tool that combines all aspects of Xilinx® software development into one unified environment. The Vitis software platform supports both the Vitis embedded software development flow, for Xilinx Software Development Kit (SDK) users looking to move into the next generation technology, and the Vitis application acceleration development flow, for software developers looking to use the latest in Xilinx FPGA-based software acceleration. This content is primarily concerned with the application acceleration flow, and use of the Vitis core development kit and Xilinx Runtime (XRT).

The Vitis application acceleration development flow provides a framework for developing and delivering FPGA accelerated applications using standard programming languages for both software and hardware components. The software component, or host program, is developed using C/C++ to run on x86 or embedded processors, with OpenCL™ API calls to manage runtime interactions with the accelerator. The hardware component, or kernel, can be developed using C/C++, OpenCL C, or RTL. The Vitis software platform promotes concurrent development and test of the Hardware and Software elements of an heterogeneous application.
As shown in the figure above, the Vitis unified software platform consists of the following features and elements:

- **Vitis technology targets** acceleration hardware platforms, such as the Alveo™ Data Center accelerator cards, and Versal or Zynq® UltraScale+™ MPSoC-based embedded processor platforms.

- **XRT** provides an API and drivers for your host program to connect with the target platform, and handles transactions between your host program and accelerated kernels.

- **Vitis core development kit** provides the software development tool stack, such as compilers and cross-compilers, to build your host program and kernel code, analyzers to let you profile and analyze the performance of your application, and debuggers to help you locate and fix any problems in your application.

- **Vitis accelerated libraries** provide performance-optimized FPGA acceleration with minimal code changes, and without the need to re-implement your algorithms to harness the benefits of Xilinx adaptive computing. Vitis accelerated libraries are available for common functions of math, statistics, linear algebra and DSP, and also for domain specific applications, like vision and image processing, quantitative finance, database, data analytics, and data compression. For more information on Vitis accelerated libraries, refer to [https://xilinx.github.io/Vitis_Libraries/](https://xilinx.github.io/Vitis_Libraries/).
FPGA Acceleration

Xilinx FPGAs offer many advantages over traditional CPU/GPU acceleration, including a custom architecture capable of implementing any function that can run on a processor, resulting in better performance at lower power dissipation. When compared with processor architectures, the structures that comprise the programmable logic (PL) fabric in a Xilinx device enable a high degree of parallelism in application execution.

To realize the advantages of software acceleration on a Xilinx device, you should look to accelerate large compute-intensive portions of your application in hardware. Implementing these functions in custom hardware gives you an ideal balance between performance and power.

For more information on how to architect an application for optimal performance and other recommended design techniques, review the Methodology for Accelerating Applications with the Vitis Software Platform.

---

Execution Model

In the Vitis core development kit, an application program is split between a host application and hardware accelerated kernels with a communication channel between them. The host program, written in C/C++ and using API abstractions like OpenCL, runs on a host processor (such as an x86 server or an Arm processor for embedded platforms), while hardware accelerated kernels run within the programmable logic (PL) region of a Xilinx device.

The API calls, managed by XRT, are used to process transactions between the host program and the hardware accelerators. Communication between the host and the kernel, including control and data transfers, occurs across the PCIe® bus or an AXI bus for embedded platforms. While control information is transferred between specific memory locations in the hardware, global memory is used to transfer data between the host program and the kernels. Global memory is accessible by both the host processor and hardware accelerators, while host memory is only accessible by the host application.

For instance, in a typical application, the host first transfers data to be operated on by the kernel from host memory into global memory. The kernel subsequently operates on the data, storing results back to the global memory. Upon kernel completion, the host transfers the results back into the host memory. Data transfers between the host and global memory introduce latency, which can be costly to the overall application. To achieve acceleration in a real system, the benefits achieved by the hardware acceleration kernels must outweigh the added latency of the data transfers.
The target platform contains the FPGA accelerated kernels, global memory, and the direct memory access (DMA) for memory transfers. Kernels can have one or more global memory interfaces and are programmable. The Vitis core development kit execution model can be broken down into the following steps:

1. The host program writes the data needed by a kernel into the global memory of the attached device through the PCIe interface on an Alveo Data Center accelerator card, or through the AXI bus on an embedded platform.
2. The host program sets up the kernel with its input parameters.
3. The host program triggers the execution of the kernel function on the FPGA.
4. The kernel performs the required computation while reading data from global memory, as necessary.
5. The kernel writes data back to global memory and notifies the host that it has completed its task.
6. The host program reads data back from global memory into the host memory and continues processing as needed.

The FPGA can accommodate multiple kernel instances on the accelerator, both different types of kernels, and multiple instances of the same kernel. XRT transparently orchestrates the interactions between the host program and kernels in the accelerator. XRT architecture documentation is available at https://xilinx.github.io/XRT/.

Data Center Application Acceleration Development Flow

The following diagram describes the steps needed to build and run an application for use on the Alveo Data Center accelerator cards. The steps are summarized below, and the details of each step can be found throughout this documentation.
Figure 2: Application Development Flow for Data Center Accelerator Cards

- **x86 Application Compilation**: Compile the host application to run on the x86 processor using the G++ compiler to create a host executable file. The host program interacts with kernels in the PL region. For more information on writing the host application, refer to Section II: Developing Applications. For more information on compiling the host application, refer to Building the Host Program.
PL Kernel Compilation and Linking:

PL kernels are compiled for implementation in the PL region of the target platform. PL kernels can be compiled into Xilinx object form (XO) file using the Vitis compiler (v++), Vitis HLS for C/C++ kernels, or the package_xo command for RTL kernels. For information on coding kernels, refer to C/C++ Kernels or RTL Kernels.

The Vitis compiler also links the kernel XO files with the hardware platform to create a device executable (.xclbin) for the application. For more information, refer to Building the Device Binary.

Xilinx object (XO) files are linked with the target hardware platform by the v++ --link command to create a device binary file (.xclbin) that is loaded into the Xilinx device on the target platform.

Running the Application: For Alveo Data Center accelerator cards, the .xclbin file is the required build object for running the system. When running the application, you can run software emulation, hardware emulation, or run on the actual physical accelerator platform. For more information, refer to Running the Application Hardware Build.

When the build target is software or hardware emulation, the Vitis compiler generates simulation models of the kernels in the device binary. As described in Build Targets, emulation targets let you build, run, and iterate the design over relatively quick cycles; debugging the application and evaluating performance.

When the build target is the hardware system, Vitis compiler generates the .xclbin using the Vivado Design Suite to run synthesis and implementation, and resolve timing. This process is automated to generate high quality results; however, hardware-savvy developers can fully leverage the Vivado tools in their design process.

Embedded Processor Application Acceleration Development Flow

The following diagram describes the steps needed to build and run an application using Arm® processors and kernels running in programmable logic regions of Versal ACAP, Zynq UltraScale+ MPSoC, and Zynq-7000 SoC devices. The steps are summarized below, and the details of each step can be found throughout this documentation.
Figure 3: Application Development Flow for Versal ACAP and Zynq UltraScale+ MPSoC Devices

**PS Application Compilation**
- main.cpp
- Arm GNU Compiler
- GDB

**PL Kernel Compilation and Linking**
- RTL
- C/C++
- design.cfg
- package.xo
- Vitis HLS
- v++ Compiler
- kernel.xo

**AI Engine Array**
- kern.cc
- adf.cxx
- AI Engine Simulator
- AI Engine Compiler

**System Package**
- sysroot
- rootfs
- Image
- v++ Package
- .xclbin
- host.elf
- libadf.a

**SW/HW Emulation**
- launch_emulator.sh

**Running the Application**
- xrt.ini
- Profile
- Optimize
- Debug
- >host.elf ./kernels.xclbin

**SD Card**
- Boot.Bin

*Optional, only applies to Versal AI Engine Core series only.*
• **PS Application Compilation**: Compile the host application to run on the Cortex®-A72 or Cortex-A53 core processor using the GNU Arm cross-compiler to create an ELF file. The host program interacts with kernels in the PL and AI Engine regions of the device. For more information on writing the host application, refer to Section II: Developing Applications. For more information on compiling the host application, refer to Building the Host Program.

• **AI Engine Array (Optional for Versal AI Engine Core series only)**: Some Versal ACAP devices incorporate an AI Engine array of very-long instruction word (VLIW) processors with single instruction multiple data (SIMD) vector units that are highly optimized for compute-intensive applications such as 5G wireless and artificial intelligence (AI) applications. AI Engine graphs and kernels are built using Vitis tools such as the aiecompiler and aiesimulator, and can be integrated into the embedded processor application acceleration flow as described in Versal ACAP AI Engine Programming Environment User Guide (UG1076).

• **PL Kernel Compilation and Linking**: PL kernels are compiled for implementation in the PL region of the target platform. PL kernels can be compiled into Xilinx object form (XO) file using the Vitis compiler (v++) or Vitis HLS for C/C++ kernels, or the package_xo command for RTL kernels. For more information on coding kernels, refer to C/C++ Kernels or RTL Kernels. The Vitis compiler also links the kernel XO files with the hardware platform to create a device executable (.xclbin) for the application. For more information, refer to Building the Device Binary.

Xilinx object (XO) files are linked with the target hardware platform by the v++ --link command to create a device binary file (.xclbin) that is loaded into the Xilinx device on the target platform.

• **System Package**: Use the v++ --package command to gather the required files to configure and boot the system, to load and run the application, including the host application and PL kernel binaries. This step builds the necessary package to run software or hardware emulation and debug, or to create an SD card to run your application on hardware. For more information, refer to Packaging the System.

• **Running the Application**: When running the application, you can run software emulation, hardware emulation, or run on the actual physical accelerator platform. Running the application on embedded processor platforms is different from running on data center accelerator cards. For more information, refer to Running the Application Hardware Build.

  • When the build target is software or hardware emulation, the Vitis compiler generates simulation models of the kernels in the device binary. As described in Build Targets, emulation targets let you build, run, and iterate the design over relatively quick cycles; debugging the application and evaluating performance.

  • When the build target is the hardware system, Vitis compiler generates the .xclbin using the Vivado Design Suite to run synthesis and implementation, and resolve timing. This process is automated to generate high quality results; however, hardware-savvy developers can fully leverage the Vivado tools in their design process.
Build Targets

The Vitis compiler build process generates the host program executable and the FPGA binary (.xclbin). The nature of the FPGA binary is determined by the build target.

- **When the build target is software or hardware emulation**, the Vitis compiler generates simulation models of the kernels in the FPGA binary. These emulation targets let you build, run, and iterate the design over relatively quick cycles; debugging the application and evaluating performance.

- **When the build target is the hardware system**, Vitis compiler generates the .xclbin for the hardware accelerator, using the Vivado Design Suite to run synthesis and implementation. It uses these tools with predefined settings proven to provide good quality of results. Using the Vitis core development kit does not require knowledge of these tools; however, hardware-savvy developers can fully leverage these tools and use all the available features to implement kernels.

The Vitis compiler provides three different build targets, two emulation targets used for debug and validation purposes, and the default hardware target used to generate the actual FPGA binary:

- **Software Emulation** (sw_emu): Both the host application code and the kernel code are compiled to run on the host processor. This allows iterative algorithm refinement through fast build-and-run loops. This target is useful for identifying syntax errors, performing source-level debugging of the kernel code running together with application, and verifying the behavior of the system.

- **Hardware Emulation** (hw_emu): The kernel code is compiled into a hardware model (RTL), which is run in a dedicated simulator. This build-and-run loop takes longer but provides a detailed, cycle-accurate view of kernel activity. This target is useful for testing the functionality of the logic that will go in the FPGA and getting initial performance estimates.

- **Hardware** (hw): The kernel code is compiled into a hardware model (RTL) and then implemented on the FPGA, resulting in a binary that will run on the actual FPGA.

Tutorials and Examples

To help you quickly get started with the Vitis core development kit, you can find tutorials, example applications, and hardware kernels in the following repositories on https://github.com/xilinx/Vitis-Tutorials.

- **Vitis Application Acceleration Development Flow Tutorials**: Provides a number of tutorials that can be worked through to teach specific concepts regarding the tool flow and application development.
The **Getting Started** pathway tutorials are an excellent place to start as a new user.

- **Vitis Examples**: Hosts many examples to demonstrate good design practices, coding guidelines, design pattern for common applications, and most importantly, optimization techniques to maximize application performance. The on-boarding examples are divided into several main categories. Each category has various key concepts illustrated by individual examples in both OpenCL™ C and C/C++ frameworks, when applicable. All examples include a Makefile to enable building for software emulation, hardware emulation, and running on hardware, and a `README.md` file with a detailed explanation of the example.

Now that you have an idea of the elements of the Vitis core development kit and how to write and build an application for acceleration, review the best approach for your design problem.
Introduction

This content focuses on Data Center applications and PCIe®-based acceleration cards, but the concepts developed here are also generally applicable to embedded applications.

Acceleration: An Industrial Analogy

There are distinct differences between CPUs, GPUs, and programmable devices. Understanding these differences is key to efficiently developing applications for each kind of device and achieving optimal acceleration.

Both CPUs and GPUs have pre-defined architectures, with a fixed number of cores, a fixed-instruction set, and a rigid memory architecture. GPUs scale performance through the number of cores and by employing SIMD/SIMT parallelism. In contrast, programmable devices are fully customizable architectures. The developer creates compute units that are optimized for application needs. Performance is achieved by creating deeply pipelined datapaths, rather than multiplying the number of compute units.

Think of a CPU as a group of workshops, with each one employing a very skilled worker. These workers have access to general purpose tools that let them build almost anything. Each worker crafts one item at a time, successively using different tools to turn raw material into finished goods. This sequential transformation process can require many steps, depending on the nature of the task. The workshops are independent, and the workers can all be doing different tasks without distractions or coordination problems.

A GPU also has workshops and workers, but it has considerably more of them, and the workers are much more specialized. They have access to only specific tools and can do fewer things, but they do them very efficiently. GPU workers function best when they do the same few tasks repeatedly, and when all of them are doing the same thing at the same time. After all, with so many different workers, it is more efficient to give them all the same orders.
Programmable devices take this workshop analogy into the industrial age. If CPUs and GPUs are groups of individual workers taking sequential steps to transform inputs into outputs, programmable devices are factories with assembly lines and conveyer belts. Raw materials are progressively transformed into finished goods by groups of workers dispatched along assembly lines. Each worker performs the same task repeatedly and the partially finished product is transferred from worker to worker on the conveyer belt. This results in a much higher production throughput.

Another major difference with programmable devices is that the factories and assembly lines do not already exist, unlike the workshops and workers in CPUs and GPUs. To refine our analogy, a programmable device would be like a collection of empty lots waiting to be developed. This means that the device developer gets to build factories, assembly lines, and workstations, and then customizes them for the required task instead of using general purpose tools. And just like lot size, device real-estate is not infinite, which limits the number and size of the factories which can be built in the device. Properly architecting and configuring these factories is therefore a critical part of the device programming process.

Traditional software development is about programming functionality on a pre-defined architecture. Programmable device development is about programming an architecture to implement the desired functionality.

**Methodology Overview**

The methodology is comprised of two major phases:

1. Architecting the application
2. Developing the C/C++ kernels

In the first phase, the developer makes key decisions about the application architecture by determining which software functions should be mapped to device kernels, how much parallelism is needed, and how it should be delivered.

In the second phase, the developer implements the kernels. This primarily involves structuring source code and applying the desired compiler pragma to create the desired kernel architecture and meet the performance target.
Performance optimization is an iterative process. The initial version of an accelerated application will likely not produce the best possible results. The methodology described in this guide is a process involving constant performance analysis and repeated changes to all aspects of the implementation.

**Recommendations**

A good understanding of the Vitis™ unified software platform programming and execution model is critical to embarking on a project with this methodology. The following resources provide the necessary knowledge to be productive with the Vitis software platform:

- **Section II: Developing Applications**
- **Vitis Application Acceleration Development Flow Tutorials** on GitHub.

In addition to understanding the key aspects of the Vitis software platform, a good understanding of the following topics will help achieve optimal results with this methodology:

- Application domain
- Software acceleration principles
- Concepts, features and architecture of device
- Features of the targeted device accelerator card and corresponding target platform
- Parallelism in hardware implementations ([http://kastner.ucsd.edu/hlsbook/](http://kastner.ucsd.edu/hlsbook/))
Methodology for Architecting a Device Accelerated Application

Before beginning the development of an accelerated application, it is important to architect it properly. In this phase, the developer makes key decisions about the architecture of the application and determines factors such as what software functions should be mapped to device kernels, how much parallelism is needed, and how it should be delivered.

Figure 5: Methodology for Architecting the Application

1. Baseline performance and establish goals
2. Identify functions to accelerate
3. Identify FPGA device parallelization needs
   - Determine how much parallelism is needed
   - Determine datapath width
   - Determine number of compute units
4. Identify SW application parallelization needs
   - Minimize CPU idle time
   - Keep FPGA accelerators utilized
   - Optimize data transfers to and from the FPGA
5. Refine architectural details
   - Finalize accelerator boundaries
   - Decide accelerator placement and connectivity

This section walks through the various steps involved in this process. Taking an iterative approach through this process helps refine the analysis and leads to better design decisions.

Step 1: Establish a Baseline Application Performance and Establish Goals

Start by measuring the runtime and throughput performance, to identify bottlenecks of the current application running on your existing platform. These performance numbers should be generated for the entire application (end-to-end) as well as for each major function in the application. The most effective way is to run the application with profiling tools, like valgrind, callgrind, and GNU gprof. The profiling data generated by these tools show the call graph with the number of calls to all functions and their execution time. These numbers provide the baseline for most of the subsequent analysis process. The functions that consume the most execution time are good candidates to be offloaded and accelerated onto FPGAs.
**Measure Running Time**

Measuring running time is a standard practice in software development. This can be done using common software profiling tools such as gprof, or by instrumenting the code with timers and performance counters.

The following figure shows an example profiling report generated with gprof. Such reports conveniently show the number of times a function is called and the amount of time spent (runtime).

![Gprof Output Example](image)

**Measure Throughput**

Throughput is the rate at which data is being processed. To compute the throughput of a given function, divide the volume of data the function processed by the running time of the function.

\[
T_{SW} = \max(V_{INPUT}, V_{OUTPUT}) / \text{Running Time}
\]

Some functions process a pre-determined volume of data. In this case, simple code inspection can be used to determine this volume. In some other cases, the volume of data is variable. In this case, it is useful to instrument the application code with counters to dynamically measure the volume.

Measuring throughput is as important as measuring running time. While device kernels can improve overall running time, they have an even greater impact on application throughput. As such, it is important to look at throughput as the main optimization target.

**Determine the Maximum Achievable Throughput**

In most device-accelerated systems, the maximum achievable throughput is limited by the PCIe® bus. PCIe performance is influenced by many different aspects, such as motherboard, drivers, target platform, and transfer sizes. Run DMA tests upfront to measure the effective throughput of PCIe transfers and thereby determine the upper bound of the acceleration potential, such as the xbuutil dma test.
An acceleration goal that exceeds this upper bound throughput cannot be met as the system will be I/O bound. Similarly, when defining kernel performance and I/O requirements, keep this upper bound in mind.

**Establish Overall Acceleration Goals**

Determining acceleration goals early in the development is necessary because the ratio between the acceleration goal and the baseline performance will drive the analysis and decision-making process.

Acceleration goals can be hard or soft. For example, a real-time video application could have the hard requirement to process 60 frames per second. A data science application could have the soft goal to run 10 times faster than an alternative implementation.

Either way, domain expertise is important for setting obtainable and meaningful acceleration goals.

**Step 2: Identify Functions to Accelerate**

After establishing the performance baseline, the next step is to determine which functions should be accelerated in the device.

**TIP:** Minimize changes to the existing code at this point so you can quickly generate a working design on the FPGA and get the baselined performance and resource numbers.

When selecting functions to accelerate in hardware, there are two aspects to consider:

- **Performance bottlenecks:** Which functions are in application hot spots?
- **Acceleration potential:** Do these functions have the potential for acceleration?
**Identify Performance Bottlenecks**

In a purely sequential application, performance bottlenecks can be easily identified by looking at profiling reports. However, most real-life applications are multi-threaded and it is important to take the effects of parallelism in consideration when looking for performance bottlenecks.

The following figure represents the performance profile of an application with two parallel paths. The width of each rectangle is proportional to the performance of each function.

*Figure 8: Application with Two Parallel Paths*

The above performance visualization in the context of parallelism shows that accelerating only one of the two paths will not improve the application's overall performance. Because paths A and B re-converge, they are dependent upon each other to finish. Likewise, accelerating A2, even by 100x, will not have a significant impact on the performance of the upper path. Therefore, the performance bottlenecks in this example are functions A1, B1, B2, and B3.

When looking for acceleration candidates, consider the performance of the entire application, not just of individual functions.

**Identify Acceleration Potential**

A function that is a bottleneck in the software application does not necessarily have the potential to run faster in a device. A detailed analysis is usually required to accurately determine the real acceleration potential of a given function. However, some simple guidelines can be used to assess if a function has potential for hardware acceleration:

- **What is the computational complexity of the function?**

  Computational complexity is the number of basic computing operations required to execute the function. In programmable devices, acceleration is achieved by creating highly parallel and deeply pipelined data paths. These would be the assembly lines in the earlier analogy. The longer the assembly line and the more stations it has, the more efficient it will be compared to a worker taking sequential steps in his workshop.

  Good candidates for acceleration are functions where a deep sequence of operations needs to be performed on each input sample to produce an output sample.

- **What is the computational intensity of the function?**
Computational intensity of a function is the ratio of the total number of operations to the total amount of input and output data. Functions with a high computational intensity are better candidates for acceleration because the overhead of moving data to the accelerator will be comparatively lower.

- **What is the data access locality profile of the function?**

  The concepts of data reuse, spatial locality, and temporal locality are useful to assess how much overhead of moving data to the accelerator can be optimized. Spatial locality reflects the average distance between several consecutive memory access operations. Temporal locality reflects the average number of access operations for an address in memory during program execution. The lower these measures the better, because it makes data more easily cacheable in the accelerator, reducing the need to expensive and potentially redundant accesses to global memory.

- **How does the throughput of the function compare to the maximum achievable in a device?**

  Device-accelerated applications are distributed, multi-process systems. The throughput of the overall application will not exceed the throughput of its slowest function. The nature of this bottleneck is application specific and can come from any aspect of the system: I/O, computation or data movement. The developer can determine the maximum acceleration potential by dividing the throughput of the slowest function by the throughput of the selected function.

  \[
  \text{Maximum Acceleration Potential} = \frac{T_{\text{Min}}}{T_{\text{SW}}}
  \]

  On Alveo Data Center accelerator cards, the PCIe bus imposes a throughput limit on data transfers. While it may not be the actual bottleneck of the application, it constitutes a possible upper bound and can therefore be used for early estimates. For example, considering a PCIe throughput of 10 GB/s and a software throughput of 50 MB/s, the maximum acceleration factor for this function is 200x.

These four criteria are not guarantees of acceleration, but they are reliable tools to identify the right functions to accelerate on a device.

### Step 3: Identify Device Parallelization Needs

After the functions to be accelerated have been identified and the overall acceleration goals have been established, the next step is to determine what level of parallelization is needed to meet the goals.

The factory analogy is again helpful to understand what parallelism is possible within kernels.

As described, the assembly line allows the progressive and simultaneous processing of inputs. In hardware, this kind of parallelism is called pipelining. The number of stations on the assembly line corresponds to the number of stages in the hardware pipeline.
Another dimension of parallelism within kernels is the ability to process multiple samples at the same time. This is like putting not just one, but multiple samples on the conveyor belt at the same time. To accommodate this, the assembly line stations are customized to process multiple samples in parallel. This is effectively defining the width of the datapath within the kernel.

Performance can be further scaled by increasing the number of assembly lines. This can be accomplished by putting multiple assembly lines in a factory, and also by building multiple identical factories with one or more assembly lines in each of them.

The developer will need to determine which combination of parallelization techniques will be most effective at meeting the acceleration goals.

**Estimate Hardware Throughput without Parallelization**

The throughput of the kernel without any parallelization can be approximated as:

\[
T_{HW} = \frac{\text{Frequency}}{\text{Computational Intensity}} = \frac{\text{Frequency} \times \max(V_{INPUT}, V_{OUTPUT})}{V_{OPS}}
\]

**Frequency** is the clock frequency of the kernel. This value is determined by the targeted acceleration platform, or target platform. For instance, the maximum kernel clock on an Alveo U200 Data Center accelerator card is 300 MHz.

As previously mentioned, the Computational Intensity of a function is the ratio of the total number of operations to the total amount of input and output data. The formula above clearly shows that functions with a high volume of operations and a low volume of data are better candidates for acceleration.

**Determine How Much Parallelism is Needed**

After the equation above has been calculated, it is possible to estimate the initial HW/SW performance ratio:

\[
\text{Speed-up} = \frac{T_{HW}}{T_{SW}} = \frac{F_{\text{max}} \times \text{Running Time}}{V_{ops}}
\]

Without any parallelization, the initial speed-up will most likely be less than 1.

Next, calculate how much parallelism is needed to meet the performance goal:

\[
\text{Parallelism Needed} = \frac{T_{Goal}}{T_{HW}} = \frac{T_{Goal} \times V_{ops}}{(F_{\text{max}} \times \max(V_{INPUT}, V_{OUTPUT}))}
\]

This parallelism can be implemented in various ways: by widening the datapath, by using multiple engines, and by using multiple kernel instances. The developer should then determine the best combination given their needs and the characteristics of their application.
**Determine How Many Samples the Datapath Should be Processing in Parallel**

One possibility is to accelerate the computation by creating a wider datapath and processing more samples in parallel. Some algorithms lend themselves well to this approach, whereas others do not. It is important to understand the nature of the algorithm to determine if this approach will work and if so, how many samples should be processed in parallel to meet the performance goal.

Processing more samples in parallel using a wider datapath improves performance by reducing the latency (running time) of the accelerated function.

**Determine How Many Kernels Can and Should be Instantiated in the Device**

If the datapath cannot be parallelized (or not sufficiently), then look at adding more kernel instances, as described in Creating Multiple Instances of a Kernel. This is usually referred to as using multiple compute units (CUs).

Adding more kernel instances improves the performance of the application by allowing the execution of more invocations of the targeted function in parallel as shown below. Multiple data sets are processed concurrently by the different instances. Application performance scales linearly with the number of instances, provided that the host application can keep the kernels busy.
As illustrated in the Using Multiple Compute Units tutorial, the Vitis technology makes it easy to scale performance by adding additional instances.

At this point, the developer should have a good understanding of the amount of parallelism necessary in the hardware to meet performance goals and through a combination of datapath width and kernel instances, how that parallelism will be achieved.

**Step 4: Identify Software Application Parallelization Needs**

While the hardware device and its kernels are designed to offer potential parallelism, the software application must be engineered to take advantage of this potential parallelism.

Parallelism in the software application is the ability for the host application to:

- Minimize idle time and do other tasks while the device kernels are running.
- Keep the device kernels active performing new computations as early and often as possible.
- Optimize data transfers to and from the device.

---

*Figure 9: Improving Performance with Multiple Compute Units*
In the world of factories and assembly lines, the host application would be the headquarters keeping busy and planning the next generation of products while the factories manufacture the current generation.

Similarly, headquarters must orchestrate the transport of goods to and from the factories and send them requests. What is the point of building many factories if the logistics department does not send them raw material or blueprints of what to create?

**Minimize CPU Idle Time While the Device Kernels are Running**

Device-acceleration is about offloading certain computations from the host processor to the kernels in the device. In a purely sequential model, the application would be waiting idly for the results to be ready and resume processing, as shown in the above figure.

Instead, engineer the software application to avoid such idle cycles. Begin by identifying parts of the application that do not depend on the results of the kernel. Then structure the application so that these functions can be executed on the host in parallel to the kernel running in the device.

**Keep the Device Kernels Utilized**

Kernels might be present in the device, but they will only run when the application requests them. To maximize performance, engineer the application so that it will keep the kernels busy.

Conceptually, this is achieved by issuing the next requests before the current ones have completed. This results in pipelined and overlapping execution, leading to kernels being optimally utilized, as shown in the following figure.
Figure 11: Pipelined Execution of Accelerators

In this example, the original application repeatedly calls func1, func2 and func3. Corresponding kernels (K1, K2, K3) have been created for the three functions. A naïve implementation would have the three kernels running sequentially, like the original software application does. However, this means that each kernel is active only a third of the time. A better approach is to structure the software application so that it can issue pipelined requests to the kernels. This allows K1 to start processing a new data set at the same time K2 starts processing the first output of K1. With this approach, the three kernels are constantly running with maximized utilization.

More information on software pipelining can be found in the Vitis Application Acceleration Development Flow Tutorials.

Optimize Data Transfers to and from the Device

In an accelerated application, data must be transferred from the host to the device especially in the case of PCIe-based applications. This introduces latency, which can be very costly to the overall performance of the application.
Data needs to be transferred at the right time, otherwise the application performance is negatively impacted if the kernel must wait for data to be available. It is therefore important to transfer data ahead of when the kernel needs it. This is achieved by overlapping data transfers and kernel execution, as described in *Keep the Device Kernels Utilized*. As shown in the sequence in the previous figure, this technique enables hiding the latency overhead of the data transfers and avoids the kernel having to wait for data to be ready.

Another method of optimizing data transfers is to transfer optimally sized buffers. As shown in the following figure, the effective PCIe throughput varies greatly based on the transferred buffer size. The larger the buffer, the better the throughput, ensuring the accelerators always have data to operate on and are not wasting cycles. It is usually better to make data transfers of 1 MB or more. Running DMA tests upfront can be useful for finding the optimal buffer sizes. Also, when determining optimal buffer sizes, consider the effect of large buffers on resource utilization and transfer latency.

Another method of optimizing data transfers is to transfer optimally sized buffers. The effective data transfer throughput varies greatly based on the size of transferred buffer. The larger the buffer, the better the throughput, ensuring the accelerators always have data to operate on and are not wasting cycles.

As shown in the following figure, on PCIe-based systems it is usually better to make data transfers of 1 MB or more. Running DMA tests in advance using the `xbutil` utility can be useful for finding the optimal buffer sizes. For more information, see `dmatest`. 
Xilinx recommends grouping multiple sets of data in a common buffer to achieve the highest possible throughput.

**Conceptualize the Desired Application Timeline**

The developer should now have a good understanding of what functions need to be accelerated, what parallelism is needed to meet performance goals, and how the application will be delivered.

At this point, it is very useful to summarize this information in the form of an expected application timeline. Application timeline sequences, such as the ones shown in Keep the Device Kernels Utilized, are very effective ways of representing performance and parallelization in action as the application runs. They represent how the potential parallelism built into the architecture is mobilized by the application.
The Vitis software platform generates timeline views from actual application runs. If the developer has a desired timeline in mind, they can compare it to the actual results, identify potential issues, and iterate and converge on the optimal results, as shown in the above figure.

Step 5: Refine Architectural Details

Before proceeding with the development of the application and its kernels, the final step consists of refining and deriving second order architectural details from the top-level decisions made up to this point.

Finalize Kernel Boundaries

As discussed earlier, performance can be improved by creating multiple instances of kernels (compute units). However, adding CUs has a cost in terms of I/O ports, bandwidth, and resources.

In the Vitis software platform flow, kernel ports have a maximum width of 512 bits (64 bytes) and have a fixed cost in terms of device resources. Most importantly, the targeted platform sets a limit on the maximum number of ports which can be used. Be mindful of these constraints and use these ports and their bandwidth optimally.

An alternative to scaling with multiple compute units is to scale by adding multiple engines within a kernel. This approach allows increasing performance in the same way as adding more CUs: multiple data sets are processed concurrently by the different engines within the kernel.

Placing multiple engines in the same kernel takes the fullest advantage of the bandwidth of the kernel’s I/O ports. If the datapath engine does not require the full width of the port, it can be more efficient to add additional engines in the kernel than to create multiple CUs with single engines in them.

Putting multiple engines in a kernel also reduces the number of ports and the number of transactions to global memory that require arbitration, improving the effective bandwidth.
On the other hand, this transformation requires coding explicit I/O multiplexing behavior in the kernel. This is a trade-off the developer needs to make.

**Decide Kernel Placement and Connectivity**

After the kernel boundaries have been finalized, the developer knows exactly how many kernels will be instantiated and therefore how many ports will need to be connected to global memory resources.

At this point, it is important to understand the features of the target platform and what global memory resources are available. For instance, the Alveo™ U200 Data Center accelerator card has 4 x 16 GB banks of DDR4 and 3 x 128 KB banks of PLRAM distributed across three super-logic regions (SLRs). For more information, refer to Vitis Software Platform Release Notes.

If kernels are factories, then global memory banks are the warehouses through which goods transit to and from the factories. The SLRs are like distinct industrial zones where warehouses preexist and factories can be built. While it is possible to transfer goods from a warehouse in one zone to a factory in another zone, this can add delay and complexity.

Using multiple DDRs helps balance the data transfer loads and improves performance. This comes with a cost, however, as each DDR controller consumes device resources. Balance these considerations when deciding how to connect kernel ports to memory banks. As explained in Mapping Kernel Ports to Memory, establishing these connections is done through a simple compiler switch, making it easy to change configurations if necessary.

After refining the architectural details, the developer should have all the information necessary to start implementing the kernels, and ultimately, assembling the entire application.

**Methodology for Developing C/C++ Kernels**

The Vitis software platform supports kernels modeled in either C/C++ or RTL (Verilog, VHDL, System Verilog). This methodology guide applies to C/C++ kernels. For details on developing RTL kernels, see RTL Kernels.

The following key kernel requirements for optimal application performance should have already been identified during the architecture definition phase:

- Throughput goal
- Latency goal
- Datapath width
- Number of engines
- Interface bandwidth
These requirements drive the kernel development and optimization process. Achieving the kernel throughput goal is the primary objective, as overall application performance is predicated on each kernel meeting the specified throughput.

The kernel development methodology therefore follows a throughput-driven approach and works from the outside-in. This approach has two phases, as also described in the following figure:

1. Defining and implementing the macro-architecture of the kernel
2. Coding and optimizing the micro-architecture of the kernel

Before starting the kernel development process, it is essential to understand the difference between functionality, algorithm, and architecture; and how they pertain to the kernel development process.

- Functionality is the mathematical relationship between input parameters and output results.
- Algorithm is a series of steps for performing a specific functionality. A given functionality can be performed using a variety of different algorithms. For instance, a sort function can be implemented using a "quick sort" or a "bubble sort" algorithm.
- Architecture, in this context, refers to the characteristics of the underlying hardware implementation of an algorithm. For instance, a particular sorting algorithm can be implemented with more or less comparators executing in parallel, with RAM or register-based storage, and so on.

You must understand that the Vitis compiler generates optimized hardware architectures from algorithms written in C/C++. However, it does not transform a particular algorithm into another one.

Therefore, because the algorithm directly influences data access locality as well as potential for computational parallelism, your choice of algorithm has a major impact on achievable performance, more so than the compiler's abilities or user specified pragmas.

The following methodology assumes that you have identified a suitable algorithm for the functionality that you want to accelerate.
Figure 14: Kernel Development Methodology

1. Partition code into a load-compute-store pattern
   - Create top level functions with desired interface
   - Code load, compute, store functions
   - Connect functions following dataflow style

2. Partition compute blocks into smaller functions
   - Decompose compute to meet throughput goal
   - Aim for functions with a single loop nest
   - Connect functions following dataflow style

3. Identify loops for optimization
   - Calculate latency target for each loop
   - Generate HLS reports
   - Identify loops exceeding latency target

4. Improve loop latency
   - Unroll loops
   - Partition and reshape arrays

5. Improve loop throughput
   - Eliminate I/O contentions
   - Eliminate loop-carried dependencies

About the High-Level Synthesis Compiler

Before starting the kernel development process, the developer should have familiarity with high-level synthesis (HLS) concepts. The HLS compiler turns C/C++ code into RTL designs which then map onto the device fabric.

The HLS compiler is more restrictive than standard software compilers. For example, there are unsupported constructs including: system function calls, dynamic memory allocation and recursive functions. For more information, see Unsupported C Constructs in the Vitis HLS Flow.

More importantly, always keep in mind that the structure of the C/C++ source code has a strong impact on the performance of the generated hardware implementation. This methodology guide will help you structure the code to meet the application throughput goals. For specific information on programming kernels, see C/C++ Kernels.

Verification Considerations

This methodology described in this guide is iterative in nature and involves successive code modifications. Xilinx® recommends verifying the code after each modification. This can be done using standard software verification methods or with the Vitis integrated design environment (IDE) software or hardware emulation flows. In either case, make sure your testing provides sufficient coverage and verification quality.
Step 1: Partition the Code into a Load-Compute-Store Pattern

A kernel is essentially a custom datapath (optimized for the desired functionality) and an associated data storage and motion network. Also referred to as the memory architecture or memory hierarchy of the kernel, this data storage and motion network is responsible for moving data in and out of the kernel and through the custom datapath as efficiently as possible.

Knowing that kernel accesses to global memory are expensive and that bandwidth is limited, it is very important to carefully plan this aspect of the kernel.

To help with this, the first step of the kernel development methodology requires structuring the kernel code into the load-compute-store pattern.

This means creating a top-level function with:

- Interface parameters matching the desired kernel interface.
- Three sub-functions: load, compute, and store.
- Local arrays or hls::stream variables to pass data between these functions.

Structuring the kernel code this way enables task-level pipelining, also known as HLS dataflow. This compiler optimization results in a design where each function can run simultaneously, creating a pipeline of concurrently running tasks. This is the premise of the assembly line in our factory, and this structure is key to achieving and sustaining the desired throughput. For more information about HLS dataflow, see Dataflow Optimization.

The load function is responsible for moving data external to the kernel (that is, global memory) to the compute function inside the kernel. This function does not perform any data processing but focuses on efficient data transfers, including buffering and caching if necessary.

The compute function, as its name suggests, is where all the processing is done. At this stage of the development flow, the internal structure of the compute function is not important.

The store function mirrors the load function. It is responsible for moving data out of the kernel, taking the results of the compute function and transferring them to global memory outside the kernel.
Creating a load-compute-store structure that meets the performance goals starts by engineering the flow of data within the kernel. Some factors to consider are:

- How does the data flow from outside the kernel into the kernel?
- How fast does the kernel need to process this data?
- How is the processed data written to the output of the kernel?

Understanding and visualizing the data movement as a block diagram will help to partition and structure the different functions within the kernel.

A working example featuring the load-compute-store pattern can be found on the Vitis Examples GitHub repository.

**Create a Top-Level Function with the Desired Interface**

The Vitis technology infers kernel interfaces from the parameters of the top-level function. Therefore, start by writing a kernel top-level function with parameters matching the desired interface.

Input parameters should be passed as scalars. Blocks of input and output data should be passed as pointers. Compiler pragmas should be used to finalize the interface definition. For complete details, see Interfaces.

**Code the Load and Store Functions**

Data transfers between the kernel and global memories have a very big influence on overall system performance. If not properly done, they will throttle the kernel. It is therefore important to optimize the load and store functions to efficiently move data in and out of the kernel and optimally feed the compute function.

The layout of data in global memory matches the layout of data in the software application. This layout must be known when writing the load and store functions. Conversely, if a certain data layout is more favorable for moving data in and out of the kernel, it is possible to adapt buffer layout in the software application. Either way, the kernel developer and application developer need to agree on how data is organized in buffers and global memory.

The following are guidelines for improving the efficiency of data transfers in and out of the kernel.

**Match Port Width to Datapath Width**

In the Vitis software platform, the port of a kernel can be up to 512 bits wide, which means that a kernel can read or write up to 64 bytes per clock cycle per port.
Xilinx recommends matching the width of the kernel ports to width of the datapath in the compute function. For instance, if the datapath needs to process 16 bytes in parallel to meet the desired throughput, then ports should be made 128-bit wide to allow reading and writing 16 bytes in parallel.

In some cases, it might be useful to access the full width bits of the interface even if the datapath does not need them. This can help reduce contention when many kernels are trying to access the same global memory bank. However, this will usually lead to additional buffering and internal memory resources in the kernel.

**Use Burst Transfers**

The first read or write request to global memory is expensive, but subsequent contiguous operations are not. Transferring data in bursts hides the memory access latency and improves bandwidth usage and efficiency of the memory controller.

Atomic accesses to global memory should always be avoided unless absolutely required. The load and store functions should be coded to always infer bursting transaction. This can be done using a `memcpy` operation as shown in the `vadd.cpp` file in the GitHub example, or by creating a tight `for` loop accessing all the required values sequentially, as explained in Interfaces in Section II: Developing Applications.

**Minimize the Number of Data Transfers from Global Memory**

Since accesses to global memory can add significant latency to the application, only make necessary transfers.

The guideline is to only read and write the necessary values, and only do so once. In situations where the same value must be used several times by the compute function, buffer this value locally instead of reading it from global memory again. Coding the proper buffering and caching structure can be key to achieving the throughput goal.

**Code the Compute Functions**

The compute function is where all the actual processing is done. This first step of the methodology is focused on getting the top-level structure right and optimizing data movement. The priority is to have a function with the right interfaces and make sure the functionality is correct. The following sections focus on the internal structure of the compute function.

**Connect the Load, Compute, and Store Functions**

Use standard C/C++ variables and arrays to connect the top-level interfaces and the load, compute and store functions. It can also be useful to use the `hls::stream` class, which models a streaming behavior.
Streaming is a type of data transfer in which data samples are sent in sequential order starting from the first sample. Streaming requires no address management and can be implemented with FIFOs. For more information about the `hls::stream` class, see Using HLS Streams in the Vitis HLS Flow.

When connecting the functions, use the canonical form required by the HLS compiler. See this Dataflow Optimization for more information. This helps the compiler build a high-throughput set of tasks using the dataflow optimization. Key recommendations include:

- Data should be transferred in the forward direction only, avoiding feedback whenever possible.
- Each connection should have a single producer and a single consumer.
- Only the load and store functions should access the primary interface of the kernel.

At this point, the developer has created the top-level function of the kernel, coded the interfaces and the load/store functions with the objective of moving data through the kernel at the desired throughput.

**Step 2: Partition the Compute Blocks into Smaller Functions**

The next step is to refine the main compute function, decomposing it into a sequence of smaller sub-functions, as shown in the following figure.
**Decompose to Identify Throughput Goals**

In a dataflow system like the one created with this approach, the slowest task will be the bottleneck.

\[
\text{Throughput(Kernel)} = \min(\text{Throughput(Task}_1), \text{Throughput(Task}_2), \ldots, \text{Throughput(Task}_N))
\]

Therefore, during the decomposition process, always have the kernel throughput goal in mind and assess whether each sub-function will be able to satisfy this throughput goal.

In the following steps of this methodology, the developer will get actual throughput numbers from running the Vitis HLS compiler. If these results cannot be improved, the developer will have to iterate and further decompose the compute stages.

**Aim for Functions with a Single Loop Nest**

As a general rule, if a function has sequential loops in it, these loops execute sequentially in the hardware implementation generated by the HLS compiler. This is usually not desirable, as sequential execution hampers throughput.
However, if these sequential loops are pushed into sequential functions, then the HLS compiler can apply the dataflow optimization and generate an implementation that allows the pipelined and overlapping execution of each task. For more information on the dataflow optimization, see Exploiting Task Level Parallelism: Dataflow Optimization in the Vitis HLS Flow.

During this partitioning and refining process, put sequential loops into individual functions. Ideally, the lowest-level compute block should only contain a single perfectly-nested loop. For more information on loops, see Loops.

**Connect Compute Functions Using the Dataflow ‘Canonical Form’**

The same rules regarding connectivity within the top-level function apply when decomposing the compute function. Aim for feed-forward connections and having a single producer and consumer for each connecting variable. If a variable must be consumed by more than one function, then it should be explicitly duplicated.

When moving blocks of data from one compute block to another, the developer can choose to use arrays or hls::stream objects.

Using arrays requires fewer code changes and is usually the fastest way to make progress during the decomposition process. However, using hls::stream objects can lead to designs using less memory resources and having shorter latency. It also helps the developer reason about how data moves through the kernel, which is always an important thing to understand when optimizing for throughput.

Using hls::stream objects is usually a good thing to do, but it is up to the developer to determine the most appropriate moment to convert arrays to streams. Some developers will do this very early on while others will do this at the very end, as a final optimization step. This can also be done using the `pragma HLS dataflow`.

At this stage, maintaining a graphical representation of the architecture of the kernel can be very useful to reason through data dependencies, data movement, control flows, and concurrency.

**Step 3: Identify Loops Requiring Optimization**

At this point, the developer has created a dataflow architecture with data motion and processing functions intended to sustain the throughput goal of the kernel. The next step is to make sure that each of the processing functions are implemented in a way that deliver the expected throughput.

As explained before, the throughput of a function is measured by dividing the volume of data processed by the latency, or running time, of the function.

\[ T = \max(V_{\text{INPUT}}, V_{\text{OUTPUT}}) / \text{Latency} \]
Both the target throughput and the volume of data consumed and produced by the function should be known at this stage of the ‘outside-in’ decomposition process described in this methodology. The developer can therefore easily derive the latency target for each function.

The Vitis HLS compiler generates detailed reports on the throughput and latency of functions and loops. Once the target latencies have been determined, use the HLS reports to identify which functions and loops do not meet their latency target and require attention, as described in HLS Report.

The latency of a loop can be calculated as follows:

\[
\text{Latency}_{\text{Loop}} = (\text{Steps} + \text{II} \times (\text{TripCount} - 1)) \times \text{ClockPeriod}
\]

Where:

- **Steps**: Duration of a single loop iteration, measured in number of clock cycles
- **TripCount**: Number of iterations in the loop.
- **II**: Initiation Interval, the number of clock cycles between the start of two consecutive iterations. When a loop is not pipelined, its II is equal to the number of Steps.

Assuming a given clock period, there are three ways to reduce the latency of a loop, and thereby improve the throughput of a function:

- Reduce the number of Steps in the loop (take less time to perform one iteration).
- Reduce the Trip Count, so that the loop performs fewer iterations.
- Reduce the Initiation Interval, so that loop iterations can start more often.

Assuming a trip count much larger than the number of steps, halving either the II or the trip count can be sufficient to double the throughput of the loop.

Understanding this information is key to optimizing loops with latencies exceeding their target. By default, the Vitis HLS compiler will try to generate loop implementations with the lowest possible II. Start by looking at how to improve latency by reducing the trip count or the number of steps. See Loops for more information.

### Step 4: Improve Loop Latencies

After identifying loops latencies that exceed their target, the first optimization to consider is loop unrolling.

**Apply Loop Unrolling**

Loop unrolling unwinds the loop, allowing multiple iterations of the loop to be executed together, reducing the loop's overall trip count.
In the industrial analogy, factories are kernels, assembly lines are dataflow pipelines, and stations are compute functions. Unrolling creates stations which can process multiple objects arriving at the same time on the conveyor belt, which results in higher performance.

**Figure 17: Loop Unrolling**

```
for (int i = 0; i < N; i++)
{
    acc += A[i] + B[i];
}
```

```
for (int i = 0; i < N; i++)
{
    #pragma HLS UNROLL factor=4
    acc += A[i] * B[i];
}
```

Loop unrolling can widen the resulting datapath by the corresponding factor. This usually increases the bandwidth requirements as more samples are processed in parallel. This has two implications:

- The width of the function I/Os must match the width of the datapath and vice versa.
- No additional benefit is gained by loop unrolling and widening the datapath to the point where I/O requirements exceed the maximum size of a kernel port (512 bits / 64 bytes).

The following guidelines will help optimize the use of loop unrolling:

- Start from the innermost loop within a loop nest.
- Assess which unroll factor would eliminate all loop-carried dependencies.
- For more efficient results, unroll loops with fixed trip counts.
- If there are function calls within the unrolled loop, in-lining these functions can improve results through better resource sharing, although at the expense of longer synthesis times. Note also that the interconnect may become increasingly complex and lead to routing problems later on.
- Do not blindly unroll loops. Always unroll loops with a specific outcome in mind.
Apply Array Partitioning

Unrolling loops changes the I/O requirements and data access patterns of the function. If a loop makes array accesses, as is almost always the case, ensure that the resulting datapath can access all the data it needs in parallel.

If unrolling a loop does not result in the expected performance improvement, this is almost always because of memory access bottlenecks.

By default, the Vitis HLS compiler maps large arrays to memory resources with a word width equal to the size of one array element. In most cases, this default mapping needs to be changed when loop unrolling is applied.

As explained in Array Configuration, the HLS compiler supports various pragmas to partition and reshape arrays. Consider using these pragmas when loop unrolling to create a memory structure that allows the desired level of parallel accesses.

Unrolling and partitioning arrays can be sufficient to meet the latency and throughput goals for the targeted loop. If so, shift to the next loop of interest. Otherwise, look at additional optimizations to improve throughput.

Step 5: Improve Loop Throughput

If improving loop latency by reducing the trip count was not sufficient, look at ways to reduce the initiation interval (II).

The loop II is the count of clock cycles between the start of two loop iterations. The Vitis HLS compiler will always try to pipeline loops, minimize the II, and start loop iterations as early as possible, ideally starting a new iteration each clock cycle (II=1).

There are two main factors that can limit the II:

- I/O contentions
- Loop-carried dependencies

The HLS Schedule Viewer automatically highlights loop dependencies limiting the II. It is a very useful visualization tool to use when working to improve the II of a loop.

Eliminate I/O Contentions

I/O contentions appear when a given I/O port of internal memory resources must be accessed more than once per loop iteration. A loop cannot be pipelined with an II lower than the number of times an I/O resource is accessed per loop iteration. If port A must be accessed four times in a loop iteration, then the lowest possible II will be 4 in single-port RAM.
The developer needs to assess whether these I/O accesses are necessary or if they can be eliminated. The most common techniques for reducing I/O contentions are:

- **Creating internal cache structures**
  
  If some of the problematic I/O accesses involve accessing data already accessed in prior loop iterations, then a possibility is to modify the code to make local copies of the values accessed in those earlier iterations. Maintaining a local data cache can help reduce the need for external I/O accesses, thereby improving the potential II of the loop.

  This example on the Vitis Accel Examples GitHub repository illustrates how a shift register can be used locally, cache previously read values, and improve the throughput of a filter.

- **Reconfiguring I/Os and memories**
  
  As explained earlier in the section about improving latency, the HLS compiler maps arrays to memories, and the default memory configuration can not offer sufficient bandwidth for the required throughput. The array partitioning and reshaping pragmas can also be used in this context to create memory structure with higher bandwidth, thereby improving the potential II of the loop.

### Eliminate Loop-Carried Dependencies

The most common case for loop-carried dependencies is when a loop iteration relies on a value computed in a prior iteration. There are differences whether the dependencies are on arrays or on scalar variables. For more information, see **Optimal Loop Unrolling to Improve Pipelining** in the Vitis HLS Flow.

- **Eliminating dependencies on arrays**
  
  The HLS compiler performs index analysis to determine whether array dependencies exist (read-after-write, write-after-read, write-after-write). The tool may not always be able to statically resolve potential dependencies and will in this case report false dependencies.

  Special compiler pragmas can overwrite these dependencies and improve the II of the design. In this situation, be cautious and do not overwrite a valid dependency.

- **Eliminating dependencies on scalars**
  
  In the case of scalar dependencies, there is usually a feedback path with a computation scheduled over multiple clock cycles. Complex arithmetic operations such as multiplications, divisions, or modulus are often found on these feedback paths. The number of cycles in the feedback path directly limits the potential II and should be reduced to improve II and throughput. To do so, analyze the feedback path to determine if and how it can be shortened. This can potentially be done using HLS scheduling constraints or code modifications such as reducing bit widths.
Advanced Techniques

If an II of 1 is usually the best scenario, it is rarely the only sufficient scenario. The goal is to meet the latency and throughput goal. To this extent, various combinations of II and unroll factor are often sufficient.

The optimization methodology and techniques presented in this guide should help meet most goals. The HLS compiler also supports many more optimization options which can be useful under specific circumstances. A complete reference of these optimizations can be found in HLS Pragmas.
Developing Applications

This section contains the following chapters:

- Programming Model
- Host Application
- C/C++ Kernels
- RTL Kernels
- Streaming Data Transfers
- OpenCL Kernel Development
- Best Practices for Acceleration with Vitis
Chapter 5

Programming Model

The Vitis™ core development kit supports heterogeneous computing using the industry standard OpenCL™ framework (https://www.khronos.org/opencl/). The host program executes on the processor (x86 or Arm®) and offloads compute intensive tasks through Xilinx Runtime (XRT) to execute on a hardware kernel running on programmable logic (PL) using the OpenCL programming paradigm.

Device Topology

In the Vitis core development kit, targeted devices can include Xilinx® MPSoCs or UltraScale+™ FPGAs connected to a processor, such as an x86 host through a PCIe bus, or an Arm processor through an AXI4 interface. The FPGA contains a programmable region that implements and executes hardware kernels.

The FPGA platform contains one or more global memory banks. The data transfer from the host machine to kernels and from kernels to the host happens through these global memory banks. The kernels running in the FPGA can have one or more memory interfaces. The connection from the global memory banks to those memory interfaces are configurable, as their features are determined by the kernel compilation options.

Multiple kernels can be implemented in the PL of the Xilinx device, allowing for significant application acceleration. A single kernel can also be instantiated multiple times. The number of instances of a kernel is programmable, and determined by linking options specified when building the FPGA binary. For more information on specifying these options, refer to Linking the Kernels.

Kernel Properties

In the Vitis application acceleration development flow, kernels are the processing elements executing in the PL region of the Xilinx device. The Vitis software platform supports kernels written in C/C++, RTL, or OpenCL C. Regardless of source language, all kernels have the same properties and must adhere to same set of requirements. This is what allows the system compiler linker and Xilinx Runtime (XRT) to seamlessly interact with the kernels.
This topic describes the properties and requirements of kernels in the Vitis application acceleration flow, discussing kernel execution modes, kernel interface requirements, and clock and reset requirements.

Kernel Execution Modes

There are three types of kernel execution modes, as described in the following table. These modes are mutually exclusive; each kernel can only operate in one of these modes. Kernels with different execution modes can be linked together by the Vitis linker to form the FPGA binary.

Table 4: Kernel Execution Modes

<table>
<thead>
<tr>
<th>Sequential Mode</th>
<th>Pipelined Mode</th>
<th>Free-Running Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control protocol: ap_ctrl_hs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• A kernel is started by the host application using an API call.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Once the kernel is done, it notifies the host application.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• The kernel can only restart once current task is completed.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Legacy mode for kernels using memory-based data transfers.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Control protocol: ap_ctrl_chain</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• A kernel is started by the host application using an API call.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Once the kernel is ready for new data, it notifies the host application.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• The kernel can be restarted before its current task is completed.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Improves performance as multiple invocations of kernel can be overlapped.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Default mode for kernels using memory-based data transfers.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Control protocol: ap_ctrl_none</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• A kernel starts as soon as the device is programmed with the xclbin.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• A kernel is running continuously and synchronizes on availability of data.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Free-running mode is not supported for kernels described in OpenCL C.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Kernel Interfaces

Kernel interfaces are used to exchange data with the host application, other kernels, or device I/Os. There are three types of interfaces allowed, each designed for a particular kind of data transfer. It is common for kernels to have multiple interfaces of different types.

Functional Properties

The following table describes the functional properties of kernel interfaces.
Table 5: Functional Properties

<table>
<thead>
<tr>
<th></th>
<th>Register</th>
<th>Memory Mapped</th>
<th>Streaming</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Designed for transferring scalars between the host application and the kernel.</td>
<td>• Designed for bi-directional data transfers with global memory (DDR, PLRAM, HBM).</td>
<td>• Designed for uni-directional data transfers between kernels or between the host application and kernels.</td>
<td></td>
</tr>
<tr>
<td>• Register reads and writes are initiated by the host application.</td>
<td>• Access pattern is usually random.</td>
<td>• Access pattern is sequential.</td>
<td></td>
</tr>
<tr>
<td>• The kernel acts as a slave.</td>
<td>• Introduces additional latency for memory transfers.</td>
<td>• Does not use global memory.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• The kernel acts as a master accessing data stored into global memory.</td>
<td>• Better performance than memory-mapped transfers.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Base address of data is sent via the Register interface.</td>
<td>• Data set is unbounded.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• The host application allocates the buffer for the size of the dataset.</td>
<td>• Sideband signal can be used to indicate the last value in the stream.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Free-running kernels cannot have memory mapped interfaces.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Clock and Reset Requirements

**Table 7: Requirements**

<table>
<thead>
<tr>
<th>C/C++/OpenCL C Kernel</th>
<th>RTL Kernel</th>
</tr>
</thead>
<tbody>
<tr>
<td>C kernel does not require any input from user on clock ports and reset ports. The HLS tool will always generate RTL with clock port <code>ap_clk</code> and reset port <code>ap_rst_n</code>.</td>
<td>• Requires a clock port. Must be named <code>ap_clk</code>.</td>
</tr>
<tr>
<td></td>
<td>• Optional clock port. Must be named <code>ap_clk_2</code>.</td>
</tr>
<tr>
<td></td>
<td>• Optional reset port. Must be named <code>ap_rst_n</code>. This signal is driven by the synchronous reset in the <code>ap_clk</code> clock domain.</td>
</tr>
<tr>
<td></td>
<td>• This reset signal is active-Low.</td>
</tr>
<tr>
<td></td>
<td>• Another optional reset port. Must be named <code>ap_rst_n_2</code>. This signal is driven by synchronous reset in the <code>ap_clk_2</code> clock domain.</td>
</tr>
</tbody>
</table>
Host Application

In the Vitis™ core development kit, host code is written in C or C++ language using the Xilinx® runtime (XRT) API or industry standard OpenCL™ API. The XRT native API is described on the XRT site at https://xilinx.github.io/XRT/2020.2/html/xrt_native_apis.html. The Vitis core development kit supports the OpenCL 1.2 API as described at https://www.khronos.org/registry/OpenCL/specs/opencl-1.2.pdf. XRT extensions to OpenCL are described at https://xilinx.github.io/XRT/2020.2/html/opencl_extension.html.

**TIP:** The code examples shown in this text use the OpenCL C language API.

In general, the structure of the host code can be divided into three sections:

1. Setting up the environment.
2. Core command execution including executing one or more kernels.
3. Post processing and release of resources.

**TIP:** The Vitis core development kit supports the OpenCL Installable Client Driver (ICD) extension (cl_khr_icd). This extension allows multiple implementations of OpenCL to co-exist on the same system. For details and installation instructions, refer to OpenCL Installable Client Driver Loader.

**Note:** For multithreading the host program, exercise caution when calling a fork() system call from a Vitis core development kit application. The fork() does not duplicate all the runtime threads. Hence, the child process cannot run as a complete application in the Vitis core development kit. It is advisable to use the posix_spawn() system call to launch another process from the Vitis software platform application.

Setting Up the Runtime Environment

The host code in the Vitis core development kit follows the OpenCL programming paradigm. To setup the runtime environment properly, the host application needs to initialize the standard OpenCL structures: target platform, devices, context, command queue, and program.

**TIP:** The host code examples and API commands used in this document follow the OpenCL C API. However, XRT also supports the OpenCL C++ wrapper API, and many of the Vitis Examples are written using the C++ API. For more information on this C++ wrapper API, refer to https://www.khronos.org/registry/OpenCL/specs/opencl-cplusplus-1.2.pdf.
Platform

Upon initialization, the host application needs to identify a platform composed of one or more Xilinx devices. The following code fragment shows a common method of identifying a Xilinx platform.

```c
cl_platform_id platform_id;         // platform id
err = clGetPlatformIDs(16, platforms, &platform_count);

// Find Xilinx Platform
for (unsigned int iplat=0; iplat<platform_count; iplat++) {
    err = clGetPlatformInfo(platforms[iplat],
        CL_PLATFORM_VENDOR, 1000,
        (void *)cl_platform_vendor, NULL);

    if (strcmp(cl_platform_vendor, "Xilinx") == 0) {
        // Xilinx Platform found
        platform_id = platforms[iplat];
    }
}
```

The OpenCL API call `clGetPlatformIDs` is used to discover the set of available OpenCL platforms for a given system. Then, `clGetPlatformInfo` is used to identify Xilinx device based platforms by matching `cl_platform_vendor` with the string 'Xilinx'.

**RECOMMENDED:** Though it is not explicitly shown in the preceding code, or in other host code examples used throughout this chapter, it is always a good coding practice to use error checking after each of the OpenCL API calls. This can help debugging and improve productivity when you are debugging the host and kernel code in the emulation flow, or during hardware execution. The following code fragment is an error checking code example for the `clGetPlatformIDs` command.

```c
err = clGetPlatformIDs(16, platforms, &platform_count);
if (err != CL_SUCCESS) {
    printf("Error: Failed to find an OpenCL platform!\n");
    printf("Test failed\n");
    exit(1);
}
```

Devices

After a Xilinx platform is found, the application needs to identify the corresponding Xilinx devices.
The following code demonstrates finding all the Xilinx devices, with an upper limit of 16, by using API `clGetDeviceIDs`.

```c
cl_device_id devices[16]; // compute device id
char cl_device_name[1001];

err = clGetDeviceIDs(platform_id, CL_DEVICE_TYPE_ACCELERATOR, 16, devices, &num_devices);
printf("INFO: Found %d devices\n", num_devices);

// iterate all devices to select the target device.
for (uint i=0; i<num_devices; i++) {
    err = clGetDeviceInfo(devices[i], CL_DEVICE_NAME, 1024, cl_device_name, 0);
    printf("CL_DEVICE_NAME %s\n", cl_device_name);
}
```

**IMPORTANT!** The `clGetDeviceIDs` API is called with the `platform_id` and `CL_DEVICE_TYPE_ACCELERATOR` to receive all the available Xilinx devices.

### Sub-Devices

In the Vitis core development kit, sometimes devices contain multiple kernel instances of a single kernel or of different kernels. While the OpenCL API `clCreateSubDevices` allows the host code to divide a device into multiple sub-devices, the Vitis core development kit supports equally divided sub-devices (using `CL_DEVICE_PARTITION_EQUALLY`), each containing one kernel instance.

The following example shows:

1. Sub-devices created by equal partition to execute one kernel instance per sub-device.
2. Iterating over the sub-device list and using a separate context and command queue to execute the kernel on each of them.
3. The API related to kernel execution (and corresponding buffer related) code is not shown for the sake of simplicity, but would be described inside the function `run_cu`.

```c
cl_uint num_devices = 0;
cl_device_partition_property props[3] = {CL_DEVICE_PARTITION_EQUALLY, 1, 0};

// Get the number of sub-devices
clCreateSubDevices(device, props, 0, nullptr, &num_devices);

// Container to hold the sub-devices
std::vector<cl_device_id> devices(num_devices);

// Second call of clCreateSubDevices
// We get sub-device handles in devices.data()
clCreateSubDevices(device, props, num_devices, devices.data(), nullptr);

// Iterating over sub-devices
std::for_each(devices.begin(), devices.end(), [=](cl_device_id sdev) {
    // Context for sub-device
```
auto context = clCreateContext(0, 1, &sdev, nullptr, nullptr, &err);

// Command-queue for sub-device
auto queue = clCreateCommandQueue(context, sdev,
        CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE, &err);

// Execute the kernel on the sub-device using local context and
queue run_cu(context, queue, kernel); // Function not shown
}

IMPORTANT! As shown in the example, you must create a separate context for each sub-device. Though
OpenCL supports a context that can hold multiple devices and sub-devices, XRT requires each device and
sub-device to have a separate context.

Context

The clCreateContext API is used to create a context that contains a Xilinx device that will
communicate with the host machine.

context = clCreateContext(0, 1, &device_id, NULL, NULL, &err);

In the code example, the clCreateContext API is used to create a context that contains one
Xilinx device. Xilinx recommends creating only one context per device or sub-device. However,
the host program should use multiple contexts if sub-devices are used with one context for each
sub-device.

Command Queues

The clCreateCommandQueue API creates one or more command queues for each device. The
FPGA can contain multiple kernels, which can be either the same or different kernels. When
developing the host application, there are two main programming approaches to execute kernels
on a device:

1. Single out-of-order command queue: Multiple kernel executions can be requested through
the same command queue. XRT dispatches kernels as soon as possible, in any order, allowing
concurrent kernel execution on the FPGA.

2. Multiple in-order command queue: Each kernel execution will be requested from different in-
order command queues. In such cases, XRT dispatches kernels from the different command
queues, improving performance by running them concurrently on the device.

The following is an example of standard API calls to create in-order and out-of-order command
queues.

    // Out-of-order Command queue
    commands = clCreateCommandQueue(context, device_id,
        CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE, &err);

    // In-order Command Queue
    commands = clCreateCommandQueue(context, device_id, 0, &err);
Program

The host and kernel code are compiled separately to create separate executable files: the host program executable and the FPGA binary (.xclbin). When the host application runs, it must load the .xclbin file using the `clCreateProgramWithBinary` API.

The following code example shows how the standard OpenCL API is used to build the program from the .xclbin file.

```c
unsigned char *kernelbinary;
char *xclbin = argv[1];
printf("INFO: loading xclbin %s\n", xclbin);
int size=load_file_to_memory(xclbin, (char **) &kernelbinary);
size_t size_var = size;
cl_program program = clCreateProgramWithBinary(context, 1, &device_id,
    &size_var,(const unsigned char **) &kernelbinary,
    &status, &err);

// Function
int load_file_to_memory(const char *filename, char **result)
{
    uint size = 0;
    FILE *f = fopen(filename, 'rb');
    if (f == NULL) {
        *result = NULL;
        return -1; // -1 means file opening fail
    }
    fseek(f, 0, SEEK_END);
    size = ftell(f);
    fseek(f, 0, SEEK_SET);
    *result = (char *)malloc(size+1);
    if (size != fread(*result, sizeof(char), size, f)) {
        free(*result);
        return -2; // -2 means file reading fail
    }
    fclose(f);
    (*result)[size] = 0;
    return size;
}
```

The example performs the following steps:

1. The kernel binary file, .xclbin, is passed in from the command line argument, `argv[1].`

   **TIP:** Passing the .xclbin through a command line argument is one approach. You can also hardcode the kernel binary file in the host program, define it with an environment variable, read it from a custom initialization file, or another suitable mechanism.

2. The `load_file_to_memory` function is used to load the file contents in the host machine memory space.

3. The `clCreateProgramWithBinary` API is used to complete the program creation process in the specified context and device.
Executing Commands in the FPGA

Once the OpenCL environment is initialized, the host application is ready to issue commands to the device and interact with the kernels. These commands include:

1. Setting up the kernels.
2. Buffer transfer to/from the FPGA.
3. Kernel execution on FPGA.
4. Event synchronization.

Setting Up Kernels

After setting up the runtime environment, such as identifying devices, creating the context, command queue, and program, the host application should identify the kernels that will execute on the device, and set up the kernel arguments.

The OpenCL API `clCreateKernel` should be used to access the kernels contained within the `.xclbin` file (the "program"). The `cl_kernel` object identifies a kernel in the program loaded into the FPGA that can be run by the host application. The following code example identifies two kernels defined in the loaded program.

```c
kernel1 = clCreateKernel(program, '<kernel_name_1>', &err);
kernels2 = clCreateKernel(program, '<kernel_name_2>', &err);  // etc
```

Setting Kernel Arguments

In the Vitis software platform, two types of arguments can be set for kernel objects:

1. Scalar arguments are used for small data transfer, such as constant or configuration type data. These are write-only arguments from the host application perspective, meaning they are inputs to the kernel.

2. Memory buffer arguments are used for large data transfer. The value is a pointer to a memory object created with the context associated with the program and kernel objects. These can be inputs to, or outputs from the kernel.

Kernel arguments can be set using the `clSetKernelArg` command, as shown in the following example for setting kernel arguments for two scalar and two buffer arguments.

```c
// Create memory buffers
cl_mem dev_buf1 = clCreateBuffer(context, CL_MEM_WRITE_ONLY | CL_MEM_USE_HOST_PTR, size, &host_mem_ptr1, NULL);
cl_mem dev_buf2 = clCreateBuffer(context, CL_MEM_READ_ONLY | CL_MEM_USE_HOST_PTR, size, &host_mem_ptr2, NULL);
int err = 0;
```
// Setup scalar arguments
cl_uint scalar_arg_image_width = 3840;
err |= clSetKernelArg(kernel, 0, sizeof(cl_uint), &scalar_arg_image_width);
cl_uint scalar_arg_image_height = 2160;
err |= clSetKernelArg(kernel, 1, sizeof(cl_uint), &scalar_arg_image_height);

// Setup buffer arguments
err |= clSetKernelArg(kernel, 2, sizeof(cl_mem), &dev_buf1);
err |= clSetKernelArg(kernel, 3, sizeof(cl_mem), &dev_buf2);

IMPORTANT! Although OpenCL allows setting kernel arguments any time before enqueuing the kernel, you should set kernel arguments as early as possible. XRT will error out if you try to migrate a buffer before XRT knows where to put it on the device. Therefore, set the kernel arguments before performing any enqueue operation (for example, clEnqueueMigrateMemObjects) on any buffer.

For all kernel buffer arguments you must allocate the buffer on the device global memories. However, sometimes the content of the buffer is not required before the start of the kernel execution. For example, the output buffer content will only be populated during the kernel execution, and hence it is not important prior to kernel execution. In this case, you should specify clEnqueueMigrateMemObject with the CL_MIGRATE_MEM_OBJECT_CONTENT_UNDEFINED flag so that migration of the buffer will not involve the DMA operation between the host and the device, thus improving performance.

### Buffer Allocation on the Device

By default, when kernels are linked to the platform the memory interfaces from all the kernels are connected to a single default global memory bank. As a result, only a single compute unit (CU) can transfer data to and from the global memory bank at one time, limiting the overall performance of the application.

If the device contains only one global memory bank, then this is the only option. However, if the device contains multiple global memory banks, you can customize the global memory bank connections by modifying the memory interface connection for a kernel during linking. The method for performing this is discussed in detail in Mapping Kernel Ports to Memory. Overall performance is improved by using separate memory banks for different kernels or compute units, enabling multiple kernel memory interfaces to concurrently read and write data.

IMPORTANT! XRT must detect the kernel's memory connection to send data from the host program to the correct memory location for the kernel. XRT will automatically find the buffer location from the kernel binary files if clSetKernelArgs is used before any enqueue operation on the buffer, such as clEnqueueMigrateMemObject.

### Buffer Creation and Data Transfer

Interactions between the host program and hardware kernels rely on creating buffers and transferring data to and from the memory in the device. This process makes use of functions like clCreateBuffer and clEnqueueMigrateMemObjects.
There are two methods for allocating memory buffers, and transferring data:

1. **Letting XRT Allocate Buffers**
2. **Using Host Pointer Buffers**

In the case where XRT allocates the buffer, use `enqueueMapBuffer` to capture the buffer handle. In the second case, allocate the buffer directly with `CL_MEM_USE_HOST_PTR`, so you do not need to capture the handle.

There are a number of coding practices you can adopt to maximize performance and fine-grain control. The OpenCL API supports additional commands for reading and writing buffers. For example, you can use `clEnqueueWriteBuffer` and `clEnqueueReadBuffer` commands in place of `clEnqueueMigrateMemObjects`. However, some of these commands have different effects that must be understood when using them. For example, `clEnqueueReadBufferRect` can read a rectangular region of a buffer object to the host application, but it does not transfer the data from the device global memory to the host. You must first use `clEnqueueReadBuffer` to transfer the data from the device global memory, and then use `clEnqueueReadBufferRect` to read the desired rectangular portion into the host application.

**Letting XRT Allocate Buffers**

On data center platforms, it is more efficient to allocate memory aligned on 4k page boundaries. On embedded platforms it is more efficient to perform contiguous memory allocation. In either case, you can let the XRT allocate host memory when creating the buffers. This is done by using the `CL_MEM_ALLOC_HOST_PTR` flag when creating the buffers, and then mapping the allocated memory to user-space pointers using `clEnqueueMapBuffer`. With this approach, it is not necessary to create a host space pointer aligned to the 4K boundary.

The `clEnqueueMapBuffer` API maps the specified buffer and returns a pointer created by XRT to this mapped region. Then, fill the host side pointer with your data, followed by `clEnqueueMigrateMemObject` to transfer the data to and from the device. The following code example uses this style:

```c
// Two cl_mem buffer, for read and write by kernel
cl_mem dev_mem_read_ptr = clCreateBuffer(context, CL_MEM_ALLOC_HOST_PTR | CL_MEM_READ_ONLY,
                                           sizeof(int) * number_of_words, NULL, NULL);

cl_mem dev_mem_write_ptr = clCreateBuffer(context, CL_MEM_ALLOC_HOST_PTR | CL_MEM_WRITE_ONLY,
                                           sizeof(int) * number_of_words, NULL, NULL);

cl::Buffer in1_buf(context, CL_MEM_ALLOC_HOST_PTR | CL_MEM_READ_ONLY,
                    sizeof(int) * DATA_SIZE, NULL, &err);
```
To work with an example using `clEnqueueMapBuffer`, refer to Data Transfer (C) in the Vitis Examples GitHub repository.

### Using Host Pointer Buffers

**IMPORTANT!** Using `CL_MEM_USE_HOST_PTR` is not recommended for embedded platforms. Embedded platforms require contiguous memory allocation and should use the `CL_MEM_ALLOC_HOST_PTR` method, as described in Letting XRT Allocate Buffers.

There are two main parts of a `cl_mem` object: host side pointer and device side pointer. Before the kernel starts its operation, the device side pointer is implicitly allocated on the device side memory (for example, on a specific location inside the device global memory) and the buffer becomes a resident on the device. Using `clEnqueueMigrateMemObjects` this allocation and data transfer occur upfront, much ahead of the kernel execution. This especially helps to enable software pipelining if the host is executing the same kernel multiple times, because data transfer for the next transaction can happen when kernel is still operating on the previous data set, and thus hide the data transfer latency of successive kernel executions.
The OpenCL framework provides a number of APIs for transferring data between the host and the device. Typically, data movement APIs, such as `clEnqueueWriteBuffer` and `clEnqueueReadBuffer`, implicitly migrate memory objects to the device after they are enqueued. They do not guarantee when the data is transferred, and this makes it difficult for the host application to synchronize the movement of memory objects with the computation performed on the data.

Xilinx recommends using `clEnqueueMigrateMemObjects` instead of `clEnqueueWriteBuffer` or `clEnqueueReadBuffer` to improve the performance. Using this API, memory migration can be explicitly performed ahead of the dependent commands. This allows the host application to preemptively change the association of a memory object, through regular command queue scheduling, to prepare for another upcoming command. This also permits an application to overlap the placement of memory objects with other unrelated operations before these memory objects are needed, potentially hiding or reducing data transfer latencies. After the event associated with `clEnqueueMigrateMemObjects` has been marked complete, the host program knows the memory objects have been successfully migrated.

**TIP:** Another advantage of `clEnqueueMigrateMemObjects` is that it can migrate multiple memory objects in a single API call. This reduces the overhead of scheduling and calling functions to transfer data for more than one memory object.

The following code shows the use of `clEnqueueMigrateMemObjects`:

```c
int host_mem_ptr[MAX_LENGTH]; // host memory for input vector

// Fill the memory input
for(int i=0; i<MAX_LENGTH; i++) {
    host_mem_ptr[i] = <... >
}

cl_mem dev_mem_ptr = clCreateBuffer(context,
    CL_MEM_READ_WRITE | CL_MEM_USE_HOST_PTR,
    sizeof(int) * number_of_words, host_mem_ptr, NULL);

clSetKernelArg(kernel, 0, sizeof(cl_mem), &dev_mem_ptr);

err = clEnqueueMigrateMemObjects(commands, 1, dev_mem_ptr, 0, 0,
    NULL, NULL);
```

### Allocating Page-Aligned Host Memory

XRT allocates memory space in 4K boundary for internal memory management. If the host memory pointer is not aligned to a page boundary, XRT performs extra `memcpy` to make it aligned. Hence you should align the host memory pointer with the 4K boundary to save the extra memory copy operation.
The following is an example of how `posix_memalign` is used instead of `malloc` for the host memory space pointer.

```c
int *host_mem_ptr; // = (int*) malloc(MAX_LENGTH*sizeof(int));
// Aligning memory in 4K boundary
posix_memalign(&host_mem_ptr,4096,MAX_LENGTH*sizeof(int));
// Fill the memory input
for(int i=0; i<MAX_LENGTH; i++) {
    host_mem_ptr[i] = <... >
}
```

```c
cl_mem dev_mem_ptr = clCreateBuffer(context,
    CL_MEM_READ_WRITE | CL_MEM_USE_HOST_PTR,
    sizeof(int) * number_of_words, host_mem_ptr, NULL);
err = clEnqueueMigrateMemObjects(commands, 1, dev_mem_ptr, 0, 0,
    NULL, NULL);
```

### Sub-Buffers

Though not very common, using sub-buffers can be very useful in specific situations. The following sections discuss the scenarios where using sub-buffers can be beneficial.

#### Reading a Specific Portion from the Device Buffer

Consider a kernel that produces different amounts of data depending on the input to the kernel. For example, a compression engine where the output size varies depending on the input data pattern and similarity. The host can still read the whole output buffer by using `clEnqueueMigrateMemObjects`, but that is a suboptimal approach as more than the required memory transfer would occur. Ideally the host program should only read the exact amount of data that the kernel has written.

One technique is to have the kernel write the amount of the output data at the start of writing the output data. The host application can use `clEnqueueReadBuffer` two times, first to read the amount of data being returned, and second to read exact amount of data returned by the kernel based on the information from the first read.

```c
clEnqueueReadBuffer(command_queue,device_write_ptr, CL_FALSE, 0,
    sizeof(int) * 1, 
    &kernel_write_size, 0, nullptr, &size_read_event);
clEnqueueReadBuffer(command_queue,device_write_ptr, CL_FALSE,
    DATA_READ_OFFSET, kernel_write_size, host_ptr, 1, &size_read_event,
    &data_read_event);
```

With `clEnqueueMigrateMemObject`, which is recommended over `clEnqueueReadBuffer` or `clEnqueueWriteBuffer`, you can adopt a similar approach by using sub-buffers. This is shown in the following code sample.
TIP: The code sample shows only partial commands to demonstrate the concept.

```c
// Create a small sub-buffer to read the quantity of data
cl_buffer_region buffer_info_1 = {0, 1 * sizeof(int)};
cl_mem size_info = clCreateSubBuffer (device_write_ptr, CL_MEM_WRITE_ONLY,
CL_BUFFER_CREATE_TYPE_REGION, &buffer_info_1, &err);

// Map the sub-buffer into the host space
auto size_info_host_ptr = clEnqueueMapBuffer(queue, size_info, ...);

// Read only the sub-buffer portion
clEnqueueMigrateMemObjects(queue, 1, &size_info,
CL_MIGRATE_MEM_OBJECT_HOST, ...);

// Retrieve size information from the already mapped size_info_host_ptr
kernel_write_size = ............

// Create sub-buffer to read the required amount of data
cl_buffer_region buffer_info_2 = {DATA_READ_OFFSET, kernel_write_size};
cl_mem buffer_seg = clCreateSubBuffer (device_write_ptr,
CL_MEM_WRITE_ONLY,
CL_BUFFER_CREATE_TYPE_REGION, &buffer_info_2, &err);

// Map the sub-buffer into the host space
auto read_mem_host_ptr = clEnqueueMapBuffer(queue, buffer_seg, ...);

// Migrate the subbuffer
clEnqueueMigrateMemObjects(queue, 1, &buffer_seg,
CL_MIGRATE_MEM_OBJECT_HOST, ...);

// Now use the read data from already mapped read_mem_host_ptr
```

Device Buffer Shared by Multiple Memory Ports or Multiple Kernels

Sometimes memory ports of kernels only require small amounts of data. However, managing small sized buffers, transferring small amounts of data, may have potential performance issues for your application. Alternatively, your host program can create a larger size buffer, divided into smaller sub-buffers. Each sub-buffer is assigned as a kernel argument as discussed in Setting Kernel Arguments, for each of those memory ports requiring small amounts of data.

Once sub-buffers are created they are used in the host code similar to regular buffers. This can improve performance as XRT handles a large buffer in a single transaction, instead of several small buffers and multiple transactions.
Kernel Execution

Often the compute intensive task required by the host application can be defined inside a single kernel, and the kernel is executed only once to work on the entire data range. Because there is an overhead associated with multiple kernel executions, invoking a single monolithic kernel can improve performance. Though the kernel is executed only one time, and works on the entire range of the data, the parallelism is achieved on the FPGA inside the kernel hardware. If properly coded, the kernel is capable of achieving parallelism by various techniques such as instruction-level parallelism (loop pipeline) and function-level parallelism (dataflow). These different kernel coding techniques are discussed in C/C++ Kernels.

When the kernel is compiled to a single hardware instance (or CU) on the FPGA, the simplest method of executing the kernel is using `clEnqueueTask` as shown below.

```c
err = clEnqueueTask(commands, kernel, 0, NULL, NULL);
```

XRT schedules the workload, or the data passed through OpenCL buffers from the kernel arguments, and schedules the kernel tasks to run on the accelerator on the Xilinx FPGA.

**IMPORTANT! Though using `clEnqueueNDRangeKernel` is supported (only for OpenCL kernel), Xilinx recommends using `clEnqueueTask`.**

However, sometimes using a single `clEnqueueTask` to run the kernel is not always feasible due to various reasons. For example, the kernel code can become too big and complex to optimize if it attempts to perform all compute intensive tasks in a single execution. Sometimes multiple kernels can be designed performing different tasks on the FPGA in parallel, requiring multiple enqueue commands. Or the host application can be receiving data over time, and not all the data can be processed at one time. Therefore, depending on the situation and application, you may need to break the data and the task of the kernel into multiple `clEnqueueTask` commands. In this case, an out-of-order command queue, or an in-order command queue can determine how the kernel tasks are processed as explained in Command Queues. In addition, multiple kernel tasks can be implemented as blocking events, or non-blocking events as described in Event Synchronization. These can all affect the performance of the design.

The following topics discuss various methods you can use to run a kernel, run multiple kernels, or run multiple instances of the same kernel on the accelerator.

**Task Parallelism Using Different Kernels**

Sometimes the compute intensive task required by the host application can be broken into multiple, different kernels designed to perform different tasks on the FPGA in parallel. By using multiple `clEnqueueTask` commands in an out-of-order command queue, for example, you can have multiple kernels performing different tasks, running in parallel. This enables the task parallelism on the FPGA.
**Spatial Data Parallelism: Increase Number of Compute Units**

Sometimes the compute intensive task required by the host application can process the data across multiple hardware instances of the same kernel, or compute units (CUs) to achieve data parallelism on the FPGA. If a single kernel has been compiled into multiple CUs, the `clEnqueueTask` command can be called multiple times in an out-of-order command queue, to enable data parallelism. Each call of `clEnqueueTask` would schedule a workload of data in different CUs, working in parallel.

**Temporal Data Parallelism: Host-to-Kernel Dataflow**

Sometimes, the data processed by a compute unit passes from one stage of processing in the kernel, to the next stage of processing. In this case, the first stage of the kernel may be free to begin processing a new set of data. In essence, like a factory assembly line, the kernel can accept new data while the original data moves down the line.

To understand this approach, assume a kernel has only one CU on the FPGA, and the host application enqueues the kernel multiple times with different sets of data. As shown in Using Host Pointer Buffers, the host application can migrate data to the device global memory ahead of the kernel execution, thus hiding the data transfer latency by the kernel execution, enabling software pipelining.

However, by default, a kernel can only start processing a new set of data only when it has finished processing the current set of data. Although `clEnqueueMigrateMemObject` hides the data transfer time, multiple kernel executions still remain sequential.

By enabling host-to-kernel dataflow, it is possible to further improve the performance of the accelerator by restarting the kernel with a new set of data while the kernel is still processing the previous set of data. As discussed in Enabling Host-to-Kernel Dataflow, the kernel must implement the `ap_ctrl_chain` interface, and must be written to permit processing data in stages. In this case, XRT restarts the kernel as soon as it is able to accept new data, thus overlapping multiple kernel executions. However, the host program must keep the command queue filled with requests so that the kernel can restart as soon as it is ready to accept new data.

The following is a conceptual diagram for host-to-kernel dataflow.
The longer the kernel takes to process a set of data from start to finish, the greater the opportunity to use host-to-kernel dataflow to improve performance. Rather than waiting until the kernel has finished processing one set of data, simply wait until the kernel is ready to begin processing the next set of data. This allows *temporal parallelism*, where different stages of the same kernel processes a different set of data from multiple `clEnqueueTask` commands, in a pipelined manner.

For advanced designs, you can effectively use both the spatial parallelism with multiple CUs to process data, combined with temporal parallelism using host-to-kernel dataflow, overlapping kernel executions on each compute unit.

**IMPORTANT!** *Embedded processor platforms do not support the host-to-kernel dataflow feature.*

### Enabling Host-to-Kernel Dataflow

If a kernel is capable of accepting more data while it is still operating on data from the previous transactions, XRT can send the next batch of data. The kernel then works on multiple data sets in parallel at different stages of the algorithm, thus improving performance. To support host-to-kernel dataflow, the kernel has to implement the `ap_ctrl_chain` protocol using the `pragma HLS interface` for the function return:

```c
void kernel_name( int *inputs, 
  ... ) // Other input or Output ports
{
  #pragma HLS INTERFACE ..... // Other interface pragmas
  #pragma HLS INTERFACE ap_ctrl_chain port=return bundle=control

  /* Kernel implementation */
}
```

**IMPORTANT!** *To take advantage of the host-to-kernel dataflow, the kernel must also be written to process data in stages, such as pipelined at the loop-level as discussed in Loop Pipelining, or pipelined at the task-level as discussed in Dataflow Optimization.*
Symmetrical and Asymmetrical Compute Units

As discussed in Creating Multiple Instances of a Kernel, multiple compute units (CUs) of a single kernel can be instantiated on the FPGA during the kernel linking process. CUs can be considered symmetrical or asymmetrical with regard to other CUs of the same kernel.

- **Symmetrical**: CUs are considered symmetrical when they have exactly the same connectivity options, and therefore have identical connections to global memory. As a result, the Xilinx Runtime can use them interchangeably. A call to `clEnqueueTask` can result in the invocation of any instance in a group of symmetrical CUs.

- **Asymmetrical**: CUs are considered asymmetrical when they do not have exactly the same connectivity options, and therefore do not have identical connections to global memory. Using the same setup of input and output buffers, it is not possible for XRT to execute asymmetrical CUs interchangeably.

Kernel Handle and Compute Units

The first time `clSetKernelArg` is called for a given kernel object, XRT identifies the group of symmetrical CUs for subsequent executions of the kernel. When `clEnqueueTask` is called for that kernel, any of the symmetrical CUs in that group can be used to process the task.

If all CUs for a given kernel are symmetrical, a single kernel object is sufficient to access any of the CUs. However, if there are asymmetrical CUs, the host application will need to create a unique kernel object for each group of asymmetrical CUs. In this case, the call to `clEnqueueTask` must specify the kernel object to use for the task, and any matching CU for that kernel can be used by XRT.

Creating Kernel Objects for Specific Compute Units

For creating kernels associated with specific compute units, the `clCreateKernel` command supports specifying the CUs at the time the kernel object is created by the host program. The syntax of this command is shown below:

```c
// Create kernel object only for a specific compute unit
cl_kernel kernelA = clCreateKernel(program, "<kernel_name>: {compute_unit_name}", &err);
// Create a kernel object for two specific compute units
cl_kernel kernelB = clCreateKernel(program, "<kernel_name>: {CU1,CU2}", &err);
```

**IMPORTANT!** As discussed in Creating Multiple Instances of a Kernel, the number of CUs is specified by the `connectivity.nk` option in a config file used by the `v++` command during linking. Therefore, whatever is specified in the host program, to create or enqueue kernel objects, must match the options specified by the config file used during linking.
In this case, the Xilinx Runtime identifies the kernel handles \((\text{kernelA}, \text{kernelB})\) for specific CUs, or group of CUs, when the kernel is created. This lets you control which kernel configuration, or specific CU instance is used, when using \texttt{clEnqueueTask} from within the host program. This can be useful in the case of asymmetrical CUs, or to perform load and priority management of CUs.

**Using Compute Unit Name to Get Handle of All Asymmetrical Compute Units**

If a kernel instantiates multiple CUs that are not symmetrical, the \texttt{clCreateKernel} command can be specified with CU names to create different CU groups. In this case, the host program can reference a specific CU group by using the \texttt{cl_kernel} handle returned by \texttt{clCreateKernel}.

In the following example, the kernel \texttt{mykernel} has five CUs: K1, K2, K3, K4, and K5. The K1, K2, and K3 CUs are a symmetrical group, having symmetrical connection on the device. Similarly, CUs K4 and K5 form a second symmetrical CU group. The following code segment shows how to address a specific CU group using \texttt{cl_kernel} handles.

```c
// Kernel handle for Symmetrical compute unit group 1: K1,K2,K3
cl_kernel kernelA = clCreateKernel(program,"mykernel:{K1,K2,K3} ",&err);
for(i=0; i<3; i++) {
    // Creating buffers for the kernel_handle1
    ....
    // Setting kernel arguments for kernel_handle1
    ....
    // Enqueue buffers for the kernel_handle1
    ....
    // Possible candidates of the executions K1,K2 or K3
    clEnqueueTask(commands, kernelA, 0, NULL, NULL);
    //
}

// Kernel handle for Symmetrical compute unit group 1: K4, K5
cl_kernel kernelB = clCreateKernel(program,"mykernel:{K4,K5} ",&err);
for(int i=0; i<2; i++) {
    // Creating buffers for the kernel_handle2
    ....
    // Setting kernel arguments for kernel_handle2
    ....
    // Enqueue buffers for the kernel_handle2
    ....
    // Possible candidates of the executions K4 or K5
    clEnqueueTask(commands, kernelB, 0, NULL, NULL);
}
```

**Event Synchronization**

All OpenCL enqueue-based API calls are asynchronous. These commands will return immediately after the command is enqueued in the command queue. To pause the host program to wait for results, or resolve any dependencies among the commands, an API call such as \texttt{clFinish} or \texttt{clWaitForEvents} can be used to block execution of the host program.
The following code shows examples for `clFinish` and `clWaitForEvents`.

```c
err = clEnqueueTask(command_queue, kernel, 0, NULL, NULL);
// Execution will wait here until all commands in the command queue are finished
clFinish(command_queue);

// Create event, read memory from device, wait for read to complete, verify results
cl_event readevent;
int host_mem_output_ptr[MAX_LENGTH];
clEnqueueReadBuffer(command_queue, dev_mem_ptr, CL_TRUE, 0, sizeof(int) * number_of_words,
                    host_mem_output_ptr, 0, NULL, &readevent);
// Wait for clEnqueueReadBuffer event to finish
clWaitForEvents(1, &readevent);
// After read is complete, verify results ...
```

Note how the commands have been used in the example above:

1. The `clFinish` API has been explicitly used to block the host execution until the kernel execution is finished. This is necessary otherwise the host can attempt to read back from the FPGA buffer too early and may read garbage data.

2. The data transfer from FPGA memory to the local host machine is done through `clEnqueueReadBuffer`. Here the last argument of `clEnqueueReadBuffer` returns an event object that identifies this particular read command, and can be used to query the event, or wait for this particular command to complete. The `clWaitForEvents` command specifies a single event (the `readevent`), and waits to ensure the data transfer is finished before verifying the data.

---

**Post-Processing and FPGA Cleanup**

At the end of the host code, all the allocated resources should be released by using proper release functions. If the resources are not properly released, the Vitis core development kit might not be able to generate a correct performance related profile and analysis report.

```c
clReleaseCommandQueue(Command_Queue);
clReleaseContext(Context);
clReleaseDevice(Target_Device_ID);
clReleaseKernel(Kernel);
clReleaseProgram(Program);
free(Platform_IDs);
free(Device_IDs);
```
Summary

As discussed in earlier topics, the recommended coding style for the host program in the Vitis core development kit includes the following points:

1. Add error checking after each OpenCL API call for debugging purpose, if required.
2. In the Vitis core development kit, one or more kernels are separately compiled/linked to build the XCLBIN file. The API `clCreateProgramWithBinary` is used to build the `cl_program` object from the kernel binary.
3. Use buffer for setting the kernel argument (`clSetKernelArg`) before any enqueue operation on the buffer.
4. Transfer data back and forth from the host code to the kernel by using `clEnqueueMigrateMemObjects`.
5. For data center platforms using `CL_MEM_USE_HOST_PTR`, apply `posix_memalign` to align the host memory pointer at 4K boundary as described in Allocating Page-Aligned Host Memory.
6. Preferably use the out-of-order command queue for concurrent command execution on the FPGA.
7. Execute the whole workload with `clEnqueueTask`, rather than splitting the workload by using `clEnqueueNDRangeKernel`.
8. Use event synchronization commands, `clFinish` and `clWaitForEvents`, to resolve dependencies of the asynchronous OpenCL API calls.
9. Release all OpenCL allocated resources when finished.
Chapter 7

C/C++ Kernels

In the Vitis™ core development kit, the kernel code is generally a compute-intensive part of the algorithm and meant to be accelerated on the FPGA. The Vitis core development kit supports the kernel code written in C/C++, OpenCL™, and also in RTL. This guide mainly focuses on the C kernel coding style.

During the runtime, the C/C++ kernel executable is called through the host code executable.

IMPORTANT! Because the host code and the kernel code are developed and compiled independently, there could be a name mangling issue if one is written in C and the other in C++. To avoid this issue, wrap the kernel function declaration with the \texttt{extern "C"} linkage in the header file, or wrap the whole function in the kernel code.

\begin{verbatim}
extern "C" {
  void kernel_function(int *in, int *out, int size);
}
\end{verbatim}

Process Execution Modes

As discussed in Kernel Execution Modes, there are three types of execution modes. These modes are determined by block protocols assigned to the kernels on the function return. The block protocol can be assigned using \texttt{#pragma HLS INTERFACE}. The modes and block protocol to enable them are listed below:

- **Pipeline**: Enabled by the default block protocol of \texttt{ap_ctrl_chain}
- **Sequential**: Serial access mode enabled by \texttt{ap_ctrl_hs}
- **Free-Running**: Enabled by \texttt{ap_ctrl_none}

For more information on how XRT supports these execution modes, refer to Supported Kernel Execution Models.
Pipeline Mode

If a kernel can accept more data while it is still operating on data from previous transactions, XRT can send the next batch of data as described in Temporal Data Parallelism: Host-to-Kernel Dataflow. Pipeline mode lets the kernel overlap multiple kernel enqueues, which improves the overall throughput.

To support pipeline mode, the kernel has to use the `ap_ctrl_chain` protocol, the default protocol used by HLS. This protocol can also be enabled by assigning the `#pragma HLS INTERFACE` to the function return as shown in the following example.

```c
void kernel_name( int *inputs,
...         )// Other input or Output ports
{
#pragma HLS INTERFACE .....   // Other interface pragmas
#pragma HLS INTERFACE ap_ctrl_chain port=return bundle=control
```

For pipeline mode to be successful, the kernel should have a longer latency for the queue of the kernel, or else there would be insufficient time for the kernel to process each batch of data, and you would not see the benefit of the pipeline.

---

**IMPORTANT!** To take advantage of the host-to-kernel dataflow, the kernel must also be written to process data in stages, such as pipelined at the loop-level, as discussed in Loop Pipelining, or pipelined at the task-level, as discussed in Dataflow Optimization.

---

For legacy reasons, the kernel also supports sequential mode that can be configured using the `ap_ctrl_hs` block protocol for the function return in the `#pragma HLS INTERFACE`. If a pipelined kernel is unable to process data in a pipelined manner, it reverts to sequential mode.

Free-Running Mode

By default Vitis HLS generates a kernel with synchronization controlled by the host application. The host controls and monitors the start and end of the kernel. However, in some cases the kernel does not need to be controlled by the host, such as in a continuously running process or data stream. This is called a free running kernel, as it is free of any control handshake. The Vitis tool supports this using the `ap_ctrl_none` block protocol in the `#pragma HLS INTERFACE` as shown in the following example.

```c
void kernel_top(hls::stream<ap_axiu >& input, hls::stream<ap_axiu >&
output)
{
#pragma HLS interface axis port=input
#pragma HLS interface axis port=output
#pragma HLS interface ap_ctrl_none port=return // Special pragma for Free
running kernel
#pragma HLS DATAFLOW
// The kernel is using DATAFLOW optimization
while(1) { ... }
}```
The kernel will run only when there is data available at the input if configured using the AXI4-Stream interface. Otherwise the kernel stalls and waits for more data. For additional details, refer to Streaming Data Transfers.

Data Types

As it is faster to write and verify the code by using native C data types such as int, float, or double, it is a common practice to use these data types when coding for the first time. However, the code is implemented in hardware and all the operator sizes used in the hardware are dependent on the data types used in the accelerator code. The default native C/C++ data types can result in larger and slower hardware resources that can limit the performance of the kernel. Instead, consider using bit-accurate data types to ensure the code is optimized for implementation in hardware. Using bit-accurate, or arbitrary precision data types, results in hardware operators which are smaller and faster. This allows more logic to be placed into the programmable logic and also allows the logic to execute at higher clock frequencies while using less power.

Consider using bit-accurate data types instead of native C/C++ data types in your code.

RECOMMENDED: Consider using bit-accurate data types instead of native C/C++ data types in your code.

In the following sections, the two most common arbitrary precision data types (arbitrary precision integer type and arbitrary precision fixed-point type) supported by the Vitis compiler are discussed.

Note: These data types should be used for C/C++ kernels only, not for OpenCL kernel (or inside the host code)

Arbitrary Precision Integer Types

Arbitrary precision integer data types are defined by ap_int or ap_uint for signed and unsigned integer respectively inside the header file ap_int.h. To use arbitrary precision integer data type:

- Add header file ap_int.h to the source code.
- Change the bit types to ap_int<N> or ap_uint<N>, where N is a bit-size from 1 to 1024.

The following example shows how the header file is added and the two variables are implemented to use 9-bit integer and 10-bit unsigned integer.

```
#include "ap_int.h"
ap_int<9> var1 // 9 bit signed integer
ap_uint<10> var2 // 10 bit unsigned integer
```


Arbitrary Precision Fixed-Point Data Types

Some existing applications use floating point data types as they are written for other hardware architectures. However, fixed-point data types are a useful replacement for floating point types which require many clock cycles to complete. When choosing to implement floating-point versus fixed-point arithmetic for your application and accelerators, carefully evaluate trade-offs in power, cost, productivity, and precision.

As discussed in Reduce Power and Cost by Converting from Floating Point to Fixed Point (WP491), using fixed-point arithmetic instead of floating point for applications can increase power efficiency, and lower the total power required. Unless the entire range of the floating-point type is required, the same accuracy can often be implemented with a fixed-point type, resulting in the same accuracy with smaller and faster hardware.

Fixed-point data types model the data as an integer and fraction bits. The fixed-point data type requires the `ap_fixed` header, and supports both a signed and unsigned form as follows:

- **Header file:** `ap_fixed.h`
- **Signed fixed point:** `ap_fixed<W,I,Q,O,N>`
- **Unsigned fixed point:** `ap_ufixed<W,I,Q,O,N>`

- **W** = Total width < 1024 bits
- **I** = Integer bit width. The value of I must be less than or equal to the width (W). The number of bits to represent the fractional part is W minus I. Only a constant integer expression can be used to specify the integer width.
- **Q** = Quantization mode. Only predefined enumerated values can be used to specify Q. The accepted values are:
  - `AP_RND`: Rounding to plus infinity.
  - `AP_RND_ZERO`: Rounding to zero.
  - `AP_RND_MIN_INF`: Rounding to minus infinity.
  - `AP_RND_INF`: Rounding to infinity.
  - `AP_RND_CONV`: Convergent rounding.
  - `AP_TRN`: Truncation. This is the default value when Q is not specified.
  - `AP_TRN_ZERO`: Truncation to zero.
- **O** = Overflow mode. Only predefined enumerated values can be used to specify O. The accepted values are:
  - `AP_SAT`: Saturation.
  - `AP_SAT_ZERO`: Saturation to zero.
**Interfaces**

Two types of data transfer occur from the host machine to and from the kernels on the FPGA. Data pointers are transferred between the host CPU and the accelerator through global memory banks. Scalar data is passed directly from the host to the kernel.

The Vitis HLS tool, which is part of the Vitis core development kit, automatically assigns interface ports for the parameters of your C/C++ kernel function. These port assignments are made during the v++ compilation process. The following sections provide additional details of these interface ports, and your ability to manually assign them, or override the default assignments using the INTERFACE pragma. If there are no user-defined INTERFACE pragmas in the code, then the following interface protocols are assigned by the Vitis tool:

- **AXI4 Master interfaces (m_axi)** are assigned to pointer arguments of the C/C++ function.
- **AXI4-Lite interfaces (s_axilite)** are assigned to scalar arguments, control signals for arrays, global variables, and the return value of the software function.
- **Vitis HLS automatically infers burst transactions to aggregate memory accesses to maximize the throughput bandwidth and/or minimize the latency. For more information on burst transfers, refer to Optimizing Burst Transfers in the Vitis HLS Flow.**
• When `hls::stream` is used to define a parameter type, the Vitis HLS tool infers an `axis` streaming interface.

**Memory Mapped Interfaces**

Memory mapped interfaces are inferred from pointer parameters. They allow kernels to read and write data in global memory, which is the memory that is shared between kernels and the host application. Therefore, memory mapped interfaces are a convenient way of sharing data across different elements of the accelerated application, but interfaces are only allowed for sequential and pipelined kernel execution modes as described in **Kernel Execution Modes**.

To customize the default interfaces assigned by the Vitis tools during compilation, you can use the INTERFACE pragma. For optimal performance, Xilinx recommends performing burst transfers, if possible up to the AXI protocol limit of 4 KB per transfer.

**Kernel Interfaces**

```c
void cnn( int *pixel, // Input pixel
          int *weights, // Input Weight Matrix
          int *out, // Output pixel
          ... // Other input or Output ports
```

In the example above, the kernel function has three pointer parameters: `pixel`, `weights`, and `out`. By default the Vitis compiler will map these three parameters to the same AXI4 interface (m_axi).

The default interface mapping inferred by the compiler is equivalent to the following INTERFACE pragmas:

```c
#pragma HLS INTERFACE m_axi port=pixel   offset=slave bundle=gmem
#pragma HLS INTERFACE m_axi port=weights offset=slave bundle=gmem
#pragma HLS INTERFACE m_axi port=out     offset=slave bundle=gmem
```

**TIP:** The inferred pragma is not added to the code by the tool; it is shown here to represent the default settings assigned to the interface port.

The `bundle` keyword on the INTERFACE pragma defines the name of the port. The system compiler will create a port for each unique bundle name, resulting in a compiled kernel object (XO) file that has a single AXI interface, m_axi_gmem. When the same `bundle` name is used for different interfaces, this results in these interfaces being mapped to same port.

**TIP:** The `gmem` name is short for global memory; however, it is not a keyword and is just used for consistency. You can assign your own names for the bundles.
Sharing ports helps save FPGA resources by eliminating AXI interfaces, but it can limit the performance of the kernel because all the memory transfers have to go through a single port. The m_axi port has independent READ and WRITE channels, so with a single m_axi port, you can do reads and writes simultaneously. However, the bandwidth and throughput of the kernel can be increased by creating multiple ports, using different bundle names, to connect to multiple memory banks. There are many options for configuring the INTERFACE, as described in `pragma HLS interface`. Some reasons to manually define an INTERFACE pragma in your code could include:

- Specifying the bundle for the INTERFACE pragma to separate AXI signals into separate bundles.
- Specifying the interface width to deviate from default int = 64 bytes (512-bits).
- Specifying AXI properties for burst transactions.

```c
void cnn( int *pixel, // Input pixel
          int *weights, // Input Weight Matrix
          int *out, // Output pixel
          ... // Other input or Output ports

#pragma HLS INTERFACE m_axi port=pixel   offset=slave bundle=gmem
#pragma HLS INTERFACE m_axi port=weights offset=slave bundle=gmem1
#pragma HLS INTERFACE m_axi port=out     offset=slave bundle=gmem
```

In the example above, two bundle names create two distinct ports: gmem and gmem1. The kernel will access pixel and out data through the gmem port, while weights data will be accessed through the gmem1 port. As a result, the kernel will be able to make parallel accesses to pixel and weights, potentially improving the throughput of the kernel.

**IMPORTANT!** Specify bundle= names using all lowercase characters, so you can assign it to a specific memory bank using the `connectivity.sp` option.

The INTERFACE pragma is used during v++ compilation, resulting in a compiled kernel object (XO) file with two separate AXI interfaces, m_axi_gmem and m_axi_gmem1, that can be connected to global memory as needed. During system compiler linking, the separate interfaces can be mapped to different global memory banks using the connectivity.sp option in a configuration file, as described in Mapping Kernel Ports to Memory.

**Memory Interface Width Considerations**

The maximum data width from the global memory to and from the kernel is 512-bits. To maximize the data transfer rate, it is recommended that you use this full data width. By default in the Vitis kernel flow, the Vitis HLS tool automatically re-sizes the kernel interface ports up to 512-bits to improve burst access. For more information, refer to Automatic Port Width Resizing in the Vitis HLS Flow.

**TIP:** The Synthesis Summary report in Vitis HLS includes information about port widening. However, to review this report you will need to launch the tool.
There are some pros and cons to using the automatic port width resizing feature which you should consider when using this feature:

- Improves the read latency from memory as the tool is reading a big vector, instead of the data type size.
- Adds resources as it needs to buffer the big vector, and shift the data to the data path size.
- Automatic port width resizing supports only standard C data types and does not support non-aggregate types such as `ap_int`, `ap_uint`, `struct`, or `array`.

**TIP:** You can disable automatic port widening, and manually size the kernel port if needed.

### Reading and Writing by Burst

Accessing the global memory bank interface from the kernel has a large latency, so global memory transfer should be done in burst. For more information on burst transfers, refer to [Optimizing Burst Transfers](#) in the Vitis HLS Flow.

**TIP:** The Synthesis Summary report in Vitis HLS includes detailed information about burst transfers in the kernel. However, to review this report you will need to launch the tool.

To infer the burst, the following pipelined loop coding style is recommended.

```c
hls::stream<datatype_t> str;

INPUT_READ: for(int i=0; i<INPUT_SIZE; i++) {
    #pragma HLS PIPELINE
    str.write(inp[i]); // Reading from Input interface
}
```

In the code example, a pipelined `for` loop is used to read data from the input memory interface, and writes to an internal `hls::stream` variable. The above coding style reads from the global memory bank in burst.

It is a recommended coding style to implement the `for` loop operation in the example above inside a separate function, and apply the dataflow optimization, as discussed in [Dataflow Optimization](#). The code example below shows how this would look, letting the compiler establish dataflow between the read, execute, and write functions:

```c
top_function(datatype_t * m_in, // Memory data Input
datatype_t * m_out, // Memory data Output
int inp1,     // Other Input
int inp2) {     // Other Input
#pragma HLS DATAFLOW

hls::stream<datatype_t> in_var1; // Internal stream to transfer
hls::stream<datatype_t> out_var1; // data through the dataflow region

read_function(m_in, inp1, in_var1); // Read function contains pipelined for loop
    // to infer burst
```
execute_function(in_var1, out_var1, inpl, inp2); // Core compute function
write_function(out_var1, m_out); // Write function contains pipelined for loop
    // to infer burst
}

**Scalar Inputs**

Scalar inputs are typically control variables that are directly loaded from the host machine. They can be thought of as programming data or parameters under which the main kernel computation takes place. These kernel inputs are write-only from the host side. In the following function, the scalar parameters are `width` and `height`.

```c
void process_image(int *input, int *output, int width, int height) {
    // The scalar arguments are assigned a default INTERFACE pragma, which is inferred by the tool.
    #pragma HLS INTERFACE s_axilite port=width bundle=control
    #pragma HLS INTERFACE s_axilite port=height bundle=control
    
    // In this example, there are two scalar inputs that specify the image width and height. These data inputs come to the kernel directly from the host machine and not through global memory banks. The pragmas shown are not added to the code by the tool.

    #important! currently, the vitis core development kit supports only one control interface bundle for each kernel. Therefore, the bundle= name should be same for all scalar data inputs and the function return. in the preceding example, bundle=control is used for all scalar inputs.
```

**Streaming Interfaces**

If the data is accessed sequentially, a streaming interface can be used. This interface enables direct streaming of data from the host to kernel, and from the kernel to host without the need to migrate the data through the global memory as an intermediate step. The streaming interface can also be used between two kernels where one kernel is streaming data as a producer to another kernel acting as a consumer. This transfer also occurs directly and without making use of global memory. For more information, refer to **Streaming Data Transfers**.

**Loops**

Loops are an important aspect for a high performance accelerator. Generally, loops are either pipelined or unrolled to take advantage of the highly distributed and parallel FPGA architecture to provide a performance boost compared to running on a CPU.
By default, loops are neither pipelined nor unrolled. Each iteration of the loop takes at least one clock cycle to execute in hardware. Thinking from the hardware perspective, there is an implicit *wait until clock* for the loop body. The next iteration of a loop only starts when the previous iteration is finished.

**Loop Pipelining**

By default, every iteration of a loop only starts when the previous iteration has finished. In the loop example below, a single iteration of the loop adds two variables and stores the result in a third variable. Assume that in hardware this loop takes three cycles to finish one iteration. Also, assume that the loop variable `len` is 20, that is, the `vadd` loop runs for 20 iterations in the kernel. Therefore, it requires a total of 60 clock cycles (20 iterations * 3 cycles) to complete all the operations of this loop.

```c
vadd: for(int i = 0; i < len; i++) {
    c[i] = a[i] + b[i];
}
```

**TIP:** It is good practice to always label a loop as shown in the above code example (*vadd:*). This practice helps with debugging when working in the Vitis core development kit. Note that the labels generate warnings during compilation, which can be safely ignored.

Pipelining the loop executes subsequent iterations in a pipelined manner. This means that subsequent iterations of the loop overlap and run concurrently, executing at different sections of the loop-body. Pipelining a loop can be enabled by the `pragma HLS pipeline`. Note that the pragma is placed inside the body of the loop.

```c
vadd: for(int i = 0; i < len; i++) {
    #pragma HLS PIPELINE
    c[i] = a[i] + b[i];
}
```

In the example above, it is assumed that every iteration of the loop takes three cycles: read, add, and write. Without pipelining, each successive iteration of the loop starts in every third cycle. With pipelining the loop can start subsequent iterations of the loop in fewer than three cycles, such as in every second cycle, or in every cycle.

The number of cycles it takes to start the next iteration of a loop is called the initiation interval (II) of the pipelined loop. So II = 2 means each successive iteration of the loop starts every two cycles. An II = 1 is the ideal case, where each iteration of the loop starts in the very next cycle. When you use `pragma HLS PIPELINE`, the compiler always tries to achieve II = 1 performance.

The following figure illustrates the difference in execution between pipelined and non-pipelined loops. In this figure, (A) shows the default sequential operation where there are three clock cycles between each input read (II = 3), and it requires eight clock cycles before the last output write is performed.
In the pipelined version of the loop shown in (B), a new input sample is read every cycle (II = 1) and the final output is written after only four clock cycles: substantially improving both the II and latency while using the same hardware resources.

IMPORTANT! Pipelining a loop causes any loops nested inside the pipelined loop to get unrolled.

If there are data dependencies inside a loop, as discussed in Loop Dependencies, it might not be possible to achieve II = 1, and a larger initiation interval might be the result.

**Loop Unrolling**

The compiler can also unroll a loop, either partially or completely to perform multiple loop iterations in parallel. This is done using the `#pragma HLS unroll`. Unrolling a loop can lead to a very fast design, with significant parallelism. However, because all the operations of the loop iterations are executed in parallel, a large amount of programmable logic resource are required to implement the hardware. As a result, the compiler can face challenges dealing with such a large number of resources and can face capacity problems that slow down the kernel compilation process. It is a good guideline to unroll loops that have a small loop body, or a small number of iterations.

In the preceding example, you can see `#pragma HLS UNROLL` has been inserted into the body of the loop to instruct the compiler to unroll the loop completely. All 20 iterations of the loop are executed in parallel if that is permitted by any data dependency.
**TIP:** Completely unrolling a loop can consume significant device resources, while partially unrolling the loop provides some performance improvement while using fewer hardware resources.

**Partially Unrolled Loop**

To completely unroll a loop, the loop must have a constant bound (20 in the example above). However, partial unrolling is possible for loops with a variable bound. A partially unrolled loop means that only a certain number of loop iterations can be executed in parallel.

The following code examples illustrates how partially unrolled loops work:

```c
array_sum: for (int i=0; i<4; i++) {
    #pragma HLS UNROLL factor=2
    sum += arr[i];
}
```

In the above example the UNROLL pragma is given a factor of 2. This is the equivalent of manually duplicating the loop body and running the two loops concurrently for half as many iterations. The following code shows how this would be written. This transformation allows two iterations of the above loop to execute in parallel.

```c
array_sum_unrolled: for (int i=0; i<4; i+=2) {
    // Manual unroll by a factor 2
    sum += arr[i];
    sum += arr[i+1];
}
```

Just like data dependencies inside a loop impact the initiation interval of a pipelined loop, an unrolled loop performs operations in parallel only if data dependencies allow it. If operations in one iteration of the loop require the result from a previous iteration, they cannot execute in parallel, but execute as soon as the data from one iteration is available to the next.

**RECOMMENDED:** A good methodology is to PIPELINE loops first, and then UNROLL loops with small loop bodies and limited iterations to improve performance further.

**Loop Dependencies**

Data dependencies in loops can impact the results of loop pipelining or unrolling. These loop dependencies can be within a single iteration of a loop or between different iterations of a loop. The straightforward method to understand loop dependencies is to examine an extreme example. In the following code example, the result of the loop is used as the loop continuation or exit condition. Each iteration of the loop must finish before the next can start.

```c
Minim_Loop: while (a != b) {
    if (a > b) {
        a -= b;
    } else {
        b -= a;
    }
}
```
This loop cannot be pipelined. The next iteration of the loop cannot begin until the previous iteration ends.

Dealing with various types of dependencies with the Vitis compiler is an extensive topic requiring a detailed understanding of the high-level synthesis procedures underlying the compiler. For more information, refer to "Dependencies with Vitis HLS" in Vitis High-Level Synthesis User Guide (UG1399).

**Nested Loops**

Coding with nested loops is a common practice. Understanding how loops are pipelined in a nested loop structure is key to achieving the desired performance.

If the HLS PIPELINE pragma is applied to a loop nested inside another loop, the `v++` compiler attempts to flatten the loops to create a single loop, and apply the PIPELINE pragma to the constructed loop. The loop flattening helps in improving the performance of the kernel.

The compiler is able to flatten the following types of nested loops:

1. **Perfect nested loop:**
   - Only the inner loop has a loop body.
   - There is no logic or operations specified between the loop declarations.
   - All the loop bounds are constant.

2. **Semi-perfect nested loop:**
   - Only the inner loop has a loop body.
   - There is no logic or operations specified between the loop declarations.
   - The inner loop bound must be a constant, but the outer loop bound can be a variable.

The following code example illustrates the structure of a perfect nested loop:

```c
ROW_LOOP: for(int i=0; i< MAX_HEIGHT; i++) {
    COL_LOOP: For(int j=0; j< MAX_WIDTH; j++) {
        #pragma HLS PIPELINE
        // Main computation per pixel
    }
}
```

The above example shows a nested loop structure with two loops that performs some computation on incoming pixel data. In most cases, you want to process a pixel in every cycle, hence, PIPELINE is applied to the nested loop body structure. The compiler is able to flatten the nested loop structure in the example because it is a perfect nested loop.
The nested loop in the preceding example contains no logic between the two loop declarations. No logic is placed between the ROW_LOOP and COL_LOOP; all of the processing logic is inside the COL_LOOP. Also, both the loops have a fixed number of iterations. These two criteria help the v+ compiler flatten the loops and apply the PIPELINE constraint.

**RECOMMENDED:** If the outer loop has a variable boundary, then the compiler can still flatten the loop. You should always try to have a constant boundary for the inner loop.

### Sequential Loops

If there are multiple loops in the design, by default they do not overlap, and execute sequentially. This section introduces the concept of dataflow optimization for sequential loops. Consider the following code example:

```c
void adder(unsigned int *in, unsigned int *out, int inc, int size) {

    unsigned int in_internal[MAX_SIZE];
    unsigned int out_internal[MAX_SIZE];
    mem_rd: for (int i = 0 ; i < size ; i++) {
        #pragma HLS PIPELINE
        // Reading from the input vector 'in' and saving to internal variable
        in_internal[i] = in[i];
    }
    compute: for (int i=0; i<size; i++) {
        #pragma HLS PIPELINE
        out_internal[i] = in_internal[i] + inc;
    }
    mem_wr: for(int i=0; i<size; i++) {
        #pragma HLS PIPELINE
        out[i] = out_internal[i];
    }
}
```

In the previous example, three sequential loops are shown: mem_rd, compute, and mem_wr.

- The `mem_rd` loop reads input vector data from the memory interface and stores it in internal storage.
- The main `compute` loop reads from the internal storage and performs an increment operation and saves the result to another internal storage.
- The `mem_wr` loop writes the data back to memory from the internal storage.

This code example is using two separate loops for reading and writing from/to the memory input/output interfaces to infer burst read/write.
By default, these loops are executed sequentially without any overlap. First, the mem_rd loop finishes reading all the input data before the compute loop starts its operation. Similarly, the compute loop finishes processing the data before the mem_wr loop starts to write the data. However, the execution of these loops can be overlapped, allowing the compute (or mem_wr) loop to start as soon as there is enough data available to feed its operation, before the mem_rd (or compute) loop has finished processing its data.

The loop execution can be overlapped using dataflow optimization as described in Dataflow Optimization.

**Dataflow Optimization**

Dataflow optimization is a powerful technique to improve the kernel performance by enabling task-level pipelining and parallelism inside the kernel. It allows the `v++` compiler to schedule multiple functions of the kernel to run concurrently to achieve higher throughput and lower latency. This is also known as task-level parallelism.

The following figure shows a conceptual view of dataflow pipelining. The default behavior is to execute and complete `func_A`, then `func_B`, and finally `func_C`. With the `pragma HLS dataflow` enabled, the compiler can schedule each function to execute as soon as data is available. In this example, the original `top` function has a latency and interval of eight clock cycles. With the dataflow optimization, the interval is reduced to only three clock cycles.

![Dataflow Optimization Figure](image-url)
Dataflow Coding Example

In the dataflow coding example you should notice the following:

1. The `pragma HLS dataflow` is applied to instruct the compiler to enable dataflow optimization. This is not a data mover, which deals with interfacing between the PS and PL, but instead addresses how the data flows through the accelerator.

2. The `stream` class is used as a data transferring channel between each of the functions in the dataflow region.

   **TIP:** The `stream` class infers a first-in first-out (FIFO) memory circuit in the programmable logic. This memory circuit, which acts as a queue in software programming, provides data-level synchronization between the functions and achieves better performance.

```c
void compute_kernel(ap_int<256> *inx, ap_int<256> *outx, DTYPE alpha) {
    hls::stream<unsigned int>inFifo;
    #pragma HLS STREAM variable=inFifo depth=32
    hls::stream<unsigned int>outFifo;
    #pragma HLS STREAM variable=outFifo depth=32

    #pragma HLS DATAFLOW
    read_data(inx, inFifo);
    // Do computation with the acquired data
    compute(inFifo, outFifo, alpha);
    write_data(outx, outFifo);
    return;
}
```

Canonical Forms of Dataflow Optimization

Xilinx recommends writing the code inside a dataflow region using canonical forms. There are canonical forms for dataflow optimizations for both functions and loops.

- Functions: The canonical form coding guideline for dataflow inside a function specifies:

  1. Use only the following types of variables inside the dataflow region:
     a. Local non-static scalar/array/pointer variables.
     b. Local static `hls::stream` variables.

  2. Function calls transfer data only in the forward direction.

  3. Array or `hls::stream` should have only one producer function and one consumer function.

  4. The function arguments (variables coming from outside the dataflow region) should only be read, or written, not both. If performing both read and write on the same function argument then read should happen before write.

  5. The local variables (those that are transferring data in forward direction) should be written before being read.
The following code example illustrates the canonical form for dataflow within a function. Note that the first function (func1) reads the inputs and the last function (func3) writes the outputs. Also note that one function creates output values that are passed to the next function as input parameters.

```c
void dataflow(Input0, Input1, Output0, Output1) {
    UserDataType C0, C1, C2;
    #pragma HLS DATAFLOW
    func1(read Input0, read Input1, write C0, write C1);
    func2(read C0, read C1, write C2);
    func3(read C2, write Output0, write Output1);
}
```

- **Loop:** The canonical form coding guideline for dataflow inside a loop body includes the coding guidelines for a function defined above, and also specifies the following:
  1. Initial value 0.
  2. The loop condition is formed by a comparison of the loop variable with a numerical constant or variable that does not vary inside the loop body.
  3. Increment by 1.

The following code example illustrates the canonical form for dataflow within a loop.

```c
void dataflow(Input0, Input1, Output0, Output1) {
    UserDataType C0, C1, C2;
    for (int i = 0; i < N; ++i) {
        #pragma HLS DATAFLOW
        func1(read Input0, read Input1, write C0, write C1);
        func2(read C0, read C0, read C1, write C2);
        func3(read C2, write Output0, write Output1);
    }
}
```

**Troubleshooting Dataflow**

The following behaviors can prevent the Vitis compiler from performing dataflow optimizations:

2. Bypassing tasks.
3. Feedback between tasks.
5. Loops with multiple exit conditions or conditions defined within the loop.

If any of the above conditions occur inside the dataflow region, you might need to re-architect your code to successfully achieve dataflow optimization.
Array Configuration

The Vitis compiler maps large arrays to the block RAM memory in the PL region. These block RAM can have a maximum of two access points or ports. This can limit the performance of the application as all the elements of an array cannot be accessed in parallel when implemented in hardware.

Depending on the performance requirements, you might need to access some or all of the elements of an array in the same clock cycle. To achieve this, the `pragma HLS array_partition` can be used to instruct the compiler to split the elements of an array and map it to smaller arrays, or to individual registers. The compiler provides three types of array partitioning, as shown in the following figure. The three types of partitioning are:

- **block**: The original array is split into equally sized blocks of consecutive elements of the original array.
- **cyclic**: The original array is split into equally sized blocks interleaving the elements of the original array.
- **complete**: Split the array into its individual elements. This corresponds to resolving a memory into individual registers. This is the default for the ARRAY_PARTITION pragma.

![Partitioning Arrays](Figure 21)

For block and cyclic partitioning, the `factor` option specifies the number of arrays that are created. In the preceding figure, a factor of 2 is used to split the array into two smaller arrays. If the number of elements in the array is not an integer multiple of the factor, the later arrays will have fewer elements.
When partitioning multi-dimensional arrays, the `dimension` option is used to specify which dimension is partitioned. The following figure shows how the `dimension` option is used to partition the following example code in three different ways:

```c
void foo (...) {
    // my_array[dim=1][dim=2][dim=3]
    // The following three pragma results are shown in the figure below
    // #pragma HLS ARRAY_PARTITION variable=my_array dim=3 <block|cyclic> factor=2
    // #pragma HLS ARRAY_PARTITION variable=my_array dim=1 <block|cyclic> factor=2
    // #pragma HLS ARRAY_PARTITION variable=my_array dim=0 complete
    int  my_array[10][6][4];
    ...
}
```

**Figure 22: Partitioning the Dimensions of an Array**

- **Partition dimension 3**
  - `my_array[10][6][4]` partition dimension 3
  - `my_array_0[10][6]`
  - `my_array_1[10][6]`
  - `my_array_2[10][6]`
  - `my_array_3[10][6]`

- **Partition dimension 1**
  - `my_array[10][6][4]` partition dimension 1
  - `my_array_0[6][4]`
  - `my_array_1[6][4]`
  - `my_array_2[6][4]`
  - `my_array_3[6][4]`
  - `my_array_4[6][4]`
  - `my_array_5[6][4]`
  - `my_array_6[6][4]`
  - `my_array_7[6][4]`
  - `my_array_8[6][4]`
  - `my_array_9[6][4]`

- **Partition dimension 0**
  - `my_array[10][6][4]` partition dimension 0
  - 10x6x4 = 240 elements

The examples in the figure demonstrate how partitioning dimension 3 results in four separate arrays and partitioning dimension 1 results in 10 separate arrays. If 0 is specified as the dimension, all dimensions are partitioned.

**The Importance of Careful Partitioning**

A complete partition of the array maps all the array elements to the individual registers. This helps in improving the kernel performance because all of these registers can be accessed concurrently in a same cycle.

**CAUTION!** Complete partitioning of the large arrays consumes a lot of PL region. It could even cause the compilation process to slow down and face capacity issue. Partition the array only when it is needed. Consider selectively partitioning a particular dimension or performing a block or cycle partitioning.
Choosing a Specific Dimension to Partition

Suppose A and B are two-dimensional arrays representing two matrices. Consider the following Matrix Multiplication algorithm:

```c
int A[64][64];
int B[64][64];

ROW_WISE: for (int i = 0; i < 64; i++) {
    COL_WISE : for (int j = 0; j < 64; j++) {
        #pragma HLS PIPELINE
        int result = 0;
        COMPUTE_LOOP: for (int k = 0; k < 64; k++) {
            result += A[i][k] * B[k][j];
        }
        C[i][j] = result;
    }
}
```

Due to the PIPELINE pragma, the \texttt{ROW\_WISE} and \texttt{COL\_WISE} loop is flattened together and \texttt{COMPUTE\_LOOP} is fully unrolled. To concurrently execute each iteration (k) of the \texttt{COMPUTE\_LOOP}, the code must access each column of matrix A and each row of matrix B in parallel. Therefore, the matrix A should be split in the second dimension, and matrix B should be split in the first dimension.

```c
#pragma HLS ARRAY_PARTITION variable=A dim=2 complete
#pragma HLS ARRAY_PARTITION variable=B dim=1 complete
```

Choosing Between Cyclic and Block Partitions

Here the same matrix multiplication algorithm is used to demonstrate choosing between cyclic and block partitioning and determining the appropriate factor, by understanding the array access pattern of the underlying algorithm.

```c
int A[64 * 64];
int B[64 * 64];
#pragma HLS ARRAY_PARTITION variable=A dim=1 cyclic factor=64
#pragma HLS ARRAY_PARTITION variable=B dim=1 block factor=64

ROW_WISE: for (int i = 0; i < 64; i++) {
    COL_WISE : for (int j = 0; j < 64; j++) {
        #pragma HLS PIPELINE
        int result = 0;
        COMPUTE_LOOP: for (int k = 0; k < 64; k++) {
            result += A[i * 64 + k] * B[k * 64 + j];
        }
        C[i* 64 + j] = result;
    }
}
```
In this version of the code, A and B are now one-dimensional arrays. To access each column of matrix A and each row of matrix B in parallel, cyclic and block partitions are used as shown in the above example. To access each column of matrix A in parallel, cyclic partitioning is applied with the factor specified as the row size, in this case 64. Similarly, to access each row of matrix B in parallel, block partitioning is applied with the factor specified as the column size, or 64.

Minimizing Array Accesses with Caching

As arrays are mapped to block RAM with limited number of access ports, repeated array accesses can limit the performance of the accelerator. You should have a good understanding of the array access pattern of the algorithm, and limit the array accesses by locally caching the data to improve the performance of the kernel.

The following code example shows a case in which accesses to an array can limit performance in the final implementation. In this example, there are three accesses to the array mem[N] to create a summed result.

```c
#include "array_mem_bottleneck.h"
dout_t array_mem_bottleneck(din_t mem[N]) {
    dout_t sum=0;
    int i;
    SUM_LOOP:for(i=2;i<N;++i)
        sum += mem[i] + mem[i-1] + mem[i-2];
    return sum;
}
```

The code in the preceding example can be rewritten as shown in the following example to allow the code to be pipelined with a II = 1. By performing pre-reads and manually pipelining the data accesses, there is only one array read specified inside each iteration of the loop. This ensures that only a single-port block RAM is needed to achieve the performance.

```c
#include "array_mem_perform.h"
dout_t array_mem_perform(din_t mem[N]) {
    din_t tmp0, tmp1, tmp2;
    dout_t sum=0;
    int i;
    tmp0 = mem[0];
    tmp1 = mem[1];
    SUM_LOOP:for (i = 2; i < N; i++) {
        tmp2 = mem[i];
        sum += tmp2 + tmp1 + tmp0;
        tmp0 = tmp1;
        tmp1 = tmp2;
    }
    return sum;
}
```

**RECOMMENDED:** Consider minimizing the array access by caching to local registers to improve the pipelining performance depending on the algorithm.
Function Inlining

C code generally consists of several functions. By default, each function is compiled, and optimized separately by the Vitis compiler. A unique hardware module will be generated for the function body and reused as needed.

From a performance perspective, in general it is better to inline the function, or dissolve the function hierarchy. This helps Vitis compiler to perform optimization more globally across the function boundary. For example, if a function is called inside a pipelined loop, then inlining the function helps the compiler to do more aggressive optimization and results in a better pipeline performance of the loop (lower initiation interval or II number).

The following INLINE pragma placed inside the function body instruct the compiler to inline the function.

```c
foo_sub (p, q) {
  #pragma HLS INLINE
  ...
  ...
}
```

However, if the function body is very big and called several times inside the main kernel function, then inlining the function may cause capacity issues due to consuming too many resources. In cases like that you might not inline such functions, and let the v++ compiler optimize the function separately in its local context.

Summary

As discussed in earlier topics, several important aspects of coding the kernel for FPGA acceleration using C/C++ include the following points:

1. Consider using arbitrary precision data types, `ap_int`, and `ap_fixed`.
2. Understand kernel interfaces to determine scalar and memory interfaces. Use `bundle switch` with different names if separate DDR memory banks will be specified in the linking stage.
3. Use Burst read and write coding style from and to the memory interface.
4. Consider exploiting the full width of DDR banks during the data transfer when selecting width of memory data inputs and outputs.
5. Get the greatest performance boost using pipelining and dataflow.
6. Write perfect or semi-perfect nested loop structure so that the v++ compiler can flatten and apply pipeline effectively.
7. Unroll loops with a small number of iterations and low operation count inside the loop body.
8. Consider understanding the array access pattern and apply complete partition to specific dimensions or apply block or cyclic partitioning instead of a complete partition of the whole array.

9. Minimize the array access by using local cache to improve kernel performance.

10. Consider inlining the function, specifically inside the pipelined region. Functions inside the dataflow should not be inlined.
Chapter 8

RTL Kernels

In the Vitis application acceleration development flow, C++ source code can be compiled into Xilinx® object (XO) files that can be linked with a target platform into an FPGA executable (XCLBIN). RTL IP from the Vivado® Design Suite can also be packaged as XO files that can be linked into an XCLBIN, as long as they adhere to Vivado IP Packaging guidelines, and requirements of the Vitis compiler. Those requirements are described here.

Requirements of an RTL Kernel

An RTL module must meet both interface and software requirements to be used as an RTL kernel within the Vitis tools. For more information on kernel properties, see Kernel Properties.

It might be necessary to revise the RTL module or Vivado IP packaging to meet the kernel requirements outlined in the following sections.

Kernel Interface Requirements

To satisfy the Vitis core development kit execution model, an RTL kernel must adhere to the requirements described in Kernel Properties. The RTL kernel must have at least one clock interface port to supply a clock to the kernel logic. The various interface requirements are summarized in the following table.

IMPORTANT! In some cases, the port names must be written exactly as shown.

Table 8: RTL Kernel Interface and Port Requirements

<table>
<thead>
<tr>
<th>Port or Interface</th>
<th>Description</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>ap_clk</td>
<td>Primary clock input port</td>
<td>• Required port.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Name must be exact.</td>
</tr>
<tr>
<td>ap_clk_2</td>
<td>Secondary optional clock input port</td>
<td>• Optional port.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Name must be exact.</td>
</tr>
</tbody>
</table>
### Table 8: RTL Kernel Interface and Port Requirements (cont’d)

<table>
<thead>
<tr>
<th>Port or Interface</th>
<th>Description</th>
<th>Comment</th>
</tr>
</thead>
</table>
| **ap_rst_n**      | Primary active-Low reset input port | • Optional port.  
                      • Name must be exact.  
                      • This signal should be internally pipelined to improve timing.  
                      • This signal is driven by a synchronous reset in the ap_clk clock domain. |
| **ap_rst_n_2**    | secondary optional active-Low reset input | • Optional port.  
                      • Name must be exact.  
                      • This signal should be internally pipelined to improve timing.  
                      • This signal is driven by a synchronous reset in the ap_clk_2 clock domain. |
| **interrupt**     | Active-High interrupt. | • Optional port.  
                      • Name must be exact. |
| **s_axi_control** | One (and only one) AXI4-Lite slave control interface | • Required port*  
                      • Name must be exact; case sensitive.  
                      **Note:** *The port is generally required, though there are exceptions such as the Free-Running Kernel.* |
| **AXI4_MASTER**   | One or more AXI4 master interfaces for global memory access | • Optional port.  
                      • All AXI4 master interfaces must have 64-bit addresses (32 bits on Zynq-7000 devices).  
                      • The RTL kernel developer is responsible for partitioning global memory spaces. Each partition in the global memory becomes a kernel argument. The memory offset for each partition must be set by a control register programmable via the AXI4-Lite slave interface.  
                      • AXI4 masters must not use Wrap or Fixed burst types and must not use narrow (sub-size) bursts meaning AxSIZE should match the width of the AXI data bus.  
                      • Any user logic or RTL code that does not conform to the requirements above, must be wrapped or bridged to satisfy these requirements. |
| **AXI4_STREAM**   | One or more AXI4-Stream interfaces for one-way data transfers between kernels or between the host application and kernels. | • Optional port.  
                      • Cannot be used with bi-directional ports.  
                      • Use the STREAM interface template in the Vivado Design Suite.  
                      • Refer to AXI4-Stream Interfaces in *Vitis High-Level Synthesis User Guide* (UG1399) for additional information on interface requirements. |
Kernel Controls

The following table outlines the required register map such that a kernel can be used within the Vitis tools and XRT. The control register is required by kernels that specify `ap_ctrl_hs` and `ap_ctrl_chain` execution models, while the interrupt related registers are only required for designs with interrupts. All user-defined registers must begin at location 0x10; locations below this are reserved.

If your RTL design has a different execution model, it must be adapted to ensure that it will operate in this manner.

**TIP: Kernels that specify `ap_ctrl_none` do not require the control registers described below.**

<table>
<thead>
<tr>
<th>Table 9: Register Address Map</th>
</tr>
</thead>
<tbody>
<tr>
<td>Offset</td>
</tr>
<tr>
<td>0x0</td>
</tr>
<tr>
<td>0x4</td>
</tr>
<tr>
<td>0x8</td>
</tr>
<tr>
<td>0xC</td>
</tr>
<tr>
<td>0x10</td>
</tr>
</tbody>
</table>

The following table shows the control signals that are accessed through the control register (offset 0x0). The available signals are used by the different control protocols as explained in Supported Kernel Execution Models in the XRT documentation. For the sequential execution mode `ap_ctrl_hs`, for example, the host typically writes 0x00000001 to the offset 0 control register which sets Bit 0, clears Bits 1 and 2, and polls on reading `ap_done` signal until it is a 1.

**Table 10: Control Register Signals**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>ap_start</td>
<td>Asserted when the kernel can start processing data. Cleared on handshake with <code>ap_done</code> being asserted.</td>
</tr>
<tr>
<td>1</td>
<td>ap_done</td>
<td>Asserted when the kernel has completed operation. Cleared on read.</td>
</tr>
<tr>
<td>2</td>
<td>ap_idle</td>
<td>Asserted when the kernel is idle.</td>
</tr>
<tr>
<td>3</td>
<td>ap_ready</td>
<td>Asserted by the kernel when it is ready to accept the new data</td>
</tr>
<tr>
<td>4</td>
<td>ap_continue</td>
<td>Asserted by the XRT to allow kernel keep running</td>
</tr>
<tr>
<td>31:5</td>
<td>Reserved</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

The control register or its signals are determined by the kernel execution mode (`ap_ctrl_hs` or `ap_ctrl_chain`).
**TIP:** The control register is not required when the kernel execution mode is `ap_ctrl_none`, as described in Free-Running Kernel.

The following interrupt related registers are only required if the kernel has an interrupt.

**Table 11: Global Interrupt Enable (0x4)**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Global Interrupt Enable</td>
<td>When asserted, along with the IP Interrupt Enable bit, the interrupt is enabled.</td>
</tr>
<tr>
<td>31:1</td>
<td>Reserved</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

**Table 12: IP Interrupt Enable (0x8)**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Interrupt Enable</td>
<td>When asserted, along with the Global Interrupt Enable bit, the interrupt is enabled.</td>
</tr>
<tr>
<td>31:1</td>
<td>Reserved</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

**Table 13: IP Interrupt Status (0xC)**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Interrupt Status</td>
<td>Toggle on write.</td>
</tr>
<tr>
<td>31:1</td>
<td>Reserved</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

**Interrupt**

RTL kernels can optionally have an interrupt port containing a single interrupt. The port name must be called `interrupt` and be active-High. It is enabled when both the global interrupt enable (`GIE`) and interrupt enable register (`IER`) bits are asserted in the Control Register block.

By default, the IER uses the internal `ap_done` signal to trigger an interrupt. Further, the interrupt is cleared only when writing a one to bit-0 of the IP Interrupt Status Register.

This logic should be reflected in the Verilog code for the RTL kernel, and also in the associated `component.xml` and `kernel.xml` files. The `kernel.xml` file is stored inside the `kernel.xo` file and is generated automatically when using the `package_xo` command or RTL Kernel Wizard.

---

**RTL Kernel Development Flow**

This section explains the two-step process for creating RTL kernels for the Vitis core development kit, which includes:
1. Package the RTL block as a standard Vivado IP.

2. Package the RTL kernel into a Xilinx Object (XO) file.

The packaged XO file is a container encapsulating the Vivado IP object (including source files) and associated kernel XML file. Using the Vitis compiler, the XO file can be combined with other kernels, and linked with the target platform and built for hardware or hardware emulation flows.

**IMPORTANT!** An RTL kernel will not support software emulation unless you provide a C-model for the kernel as explained in Creating the XO File from the RTL Kernel.

**Package the RTL Code as a Vivado IP**

RTL kernels must be packaged as a Vivado IP that can be used with the IP integrator. For details on IP packaging in the Vivado tool, see the Vivado Design Suite User Guide: Creating and Packaging Custom IP (UG1118).

The following required interfaces for the RTL kernel must be packaged:

- The AXI4-Lite interface name must be packaged as S_AXI_CONTROL, but the underlying AXI ports can be named differently.
- Any memory-mapped AXI4 interfaces must be packaged as AXI4 master endpoints with 64-bit address support.

**RECOMMENDED:** Xilinx strongly recommends that AXI4 interfaces be packaged with AXI meta data HAS_BURST=0 and SUPPORTS_NARROW_BURST=0. These properties can be set in an IP-level bd.tcl file. This indicates wrap and fixed burst type is not used, and narrow (sub-size burst) is not used.

- You can also implement the AXI4-Stream interface.
- `ap_clk` and `ap_clk_2` must be packaged as clock interfaces (`ap_clk_2` is only required when the RTL kernel has two clocks).
- `ap_rst_n` and `ap_rst_n_2` must be packaged as active-Low reset interfaces (when the RTL kernel has a reset).
- `ap_clk` must be associated with all AXI4-Lite, AXI4, AXI4-Stream interfaces, and also any reset signals, `ap_rst_n`, on the kernel.

To package the IP, use the following steps:

1. Create and package a new IP.

   a. From a Vivado project, with your RTL source files added, select **Tools → Create and Package New IP**.

   b. Select **Package your current project**, and click **Next**.

       You can select the default location for your IP, or choose a different location.
c. To open the Package IP window, select Finish.

2. Associate the clock to the AXI interfaces.

   In the Ports and Interfaces section of the Package IP window, you can associate the `ap_clk` with the AXI4 interfaces, and reset signal if needed.

   a. Right-click an interface, and select **Associate Clocks**.

      This opens the Associate Clocks dialog box which lists the `ap_clk`, and perhaps `ap_clk_2`.

   b. Select the `ap_clk` and click **OK** to associate it with the interface.

   c. Make sure to repeat this step to associate `ap_clk` with each of the AXI interfaces, and the reset.

3. Add the control registers and offsets.

   The kernel requires control registers as discussed in **Kernel Controls**. The following table shows a list of the required registers.

   **Table 14: Address Map**

<table>
<thead>
<tr>
<th>Register Name</th>
<th>Description</th>
<th>Address Offset</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>CTRL</td>
<td>Control Signals.</td>
<td>0x000</td>
<td>32</td>
</tr>
<tr>
<td></td>
<td>IMPORTANT! The CTRL register and <code>&lt;kernel_args&gt;</code> are required on all kernels. The interrupt related registers are only required for designs with interrupts.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GIER</td>
<td>Global Interrupt Enable Register. Used to enable interrupt to the host.</td>
<td>0x004</td>
<td>32</td>
</tr>
<tr>
<td>IP_IER</td>
<td>IP Interrupt Enable Register. Used to control which IP generated signal are used to generate an interrupt.</td>
<td>0x008</td>
<td>32</td>
</tr>
<tr>
<td>IP_ISR</td>
<td>IP Interrupt Status Register. Provides interrupt status.</td>
<td>0x00C</td>
<td>32</td>
</tr>
<tr>
<td><code>&lt;kernel_args&gt;</code></td>
<td>This includes a separate entry for each kernel argument as needed on the software function interface. All user-defined registers must begin at location 0x10; locations below this are reserved.</td>
<td>0x010</td>
<td>32/64</td>
</tr>
<tr>
<td></td>
<td>Scalar arguments are 32-bits wide. m_axi and axis interfaces are 64 bits wide.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

   a. To create the address map described in the table, select the **Addressing and Memory** section of the Package IP window. Right-click in the **Address Blocks** and select the **Add Register** command.

      This opens the Add Register dialog box in which you can enter one of the register names from the table above.

   b. Repeat as needed to add all required registers.
This creates a Registers table in the Addressing and Memory section. You can edit the table to add the Description, Address Offset, and Size to each register. The Registers table should look similar to the following example.

<table>
<thead>
<tr>
<th>Name</th>
<th>Display Name</th>
<th>Description</th>
<th>Address Offset</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>CTRL</td>
<td></td>
<td>Control signals</td>
<td>0x00</td>
<td>32</td>
</tr>
<tr>
<td>GIB</td>
<td></td>
<td>Global Interrupt Enable Register</td>
<td>0x04</td>
<td>32</td>
</tr>
<tr>
<td>IP ISP</td>
<td></td>
<td>IP Interrupt Enable Register</td>
<td>0x08</td>
<td>32</td>
</tr>
<tr>
<td>scpaxi</td>
<td></td>
<td>IP Interrupt Status Register</td>
<td>0x0C</td>
<td>32</td>
</tr>
<tr>
<td>A</td>
<td></td>
<td>m00_axi</td>
<td>0x10</td>
<td>32</td>
</tr>
<tr>
<td>B</td>
<td></td>
<td>m01_axi</td>
<td>0x18</td>
<td>32</td>
</tr>
</tbody>
</table>

**c.** Finally, select the register for each of the pointer arguments from your table, right-click and select the **Add Register Parameter** command. Enter the name `ASSOCIATED_BUSIF` into the dialog box that opens, and click **OK**.

This lets you define an association between the register and the AXI4 Interface. In the value field of the added parameter, enter the name of the `m_axi` interface assigned to the specific argument you are defining. In the example above, the argument A uses the `m00_axi` interface, and the argument B uses the `m01_axi` interface.

**4.** Add required properties to the IP:

The IP requires a few standard properties that you can add to your core. The easiest way to do this is by using the following commands from the Vivado Tcl Console.

```tcl
set_property sdx_kernel true [ipx::current_core]
set_property sdx_kernel_type rtl [ipx::current_core]
```

**5.** At this point you should be ready to package your IP.

**a.** Select the **Review and Package** section of the Package IP window, review the Summary and After Packaging sections, and make whatever changes are needed.

**IMPORTANT!** You must enable the generation of an IP archive file. If the After Packaging section indicates *An archive will not be generated*, you must select the **Edit packaging settings** link and enable the **Create archive of IP** setting.
b. When you are ready, click **Package IP**.

The Vivado tool packages your kernel IP and opens a dialog box to inform you of success. You can go on to package the kernel using the `package_xo` command, as described in *Creating the XO File from the RTL Kernel*.

6. To test if the RTL kernel is packaged correctly for the IP integrator, try to instantiate the packaged kernel IP into a block design in the IP integrator. For information on the tool, refer to *Vivado Design Suite User Guide: Designing IP Subsystems Using IP Integrator (UG994)*.

7. The kernel IP should show the various interfaces described above. Examine the IP in the canvas view. The properties of the AXI interface can be viewed by selecting the interface on the canvas. Then in the Block Interface Properties window, select the **Properties** tab and expand the **CONFIG** table entry. If an interface is to be read-only or write-only, the unused AXI channels can be removed and the **READ_WRITE_MODE** is set to read-only or write-only.

8. If the RTL kernel has constraints which refer to constraints in the static area such as clocks, then the RTL kernel constraint file needs to be marked as late processing order to ensure RTL kernel constraints are correctly applied.

There are two methods to mark constraints as late processing order:

a. If the constraints are given in a `.ttcl` file, add `<: setFileProcessingOrder 'late' :>` to the `.ttcl` preamble section of the file as follows:

```ttcl
<: set ComponentName [getComponentNameString] :>
<: setOutputDirectory "./" :>
<: setFileName $ComponentName :>
<: setFileExtension ".xdc" :>
<: setFileProcessingOrder 'late' :>
```

b. If constraints are defined in an `.xdc` file, then add the following four lines starting at `<spirit:define>` in the `component.xml`. The four lines in the `component.xml` need to be next to the area where the `.xdc` file is called. In the following example, `my_ip_constraint.xdc` file is being called with the subsequent late processing order defined.

```xml
<spirit:file>
  <spirit:name>ttcl/my_ip_constraint.xdc</spirit:name>
  <spirit:userFileType>ttcl</spirit:userFileType>
  <spirit:userFileType>USED_IN.implementation</spirit:userFileType>
  <spirit:userFileType>USED_IN.synthesis</spirit:userFileType>
  <spirit:define>
    <spirit:name>processing_order</spirit:name>
    <spirit:value>late</spirit:value>
  </spirit:define>
</spirit:file>
```

**Creating the XO File from the RTL Kernel**

The final step is to package the RTL IP into a Xilinx object (XO) file, so the kernel can be used in the Vitis core development kit. This is done using the `package_xo` Tcl command in the Vivado Design Suite.
After packaging the IP, the `package_xo` command is run from within the Vivado tool. The `package_xo` command uses the `component.xml` file from the IP to create the necessary `kernel.xml` if possible. The Vivado tool runs design rule checks as a pre-processor for `package_xo` to determine that everything is available and either processes the IP to create the XO file, or returns errors indicating any issues that might exist.

The following example packages an RTL kernel IP named `test_sincos`, found in the specified IP directory, into an object file named `test.xo`, creating the required `kernel.xml` file, and using the `ap_ctrl_chain` protocol:

```bash
package_xo -xo_path ./test.xo -kernel_name test_sincos \   -ctrl_protocol ap_ctrl_chain -ip_directory ./ip/
```

The output of the `package_xo` command is the `test.xo` file, that can be added as a source file to the `v++ --link` command as discussed in Section III: Building and Running the Application, or added to an application project as discussed in Section VII: Using the Vitis IDE.

In some cases, you might find it necessary to provide a `kernel.xml` file for your IP, as specified in the requirements described in RTL Kernel XML File. You can use the `-kernel_xml` option to specify the file for the `package_xo` command. In this case, the `package_xo` command uses the `kernel.xml` as specified. The following example shows this command.

```bash
package_xo -xo_path ./test.xo -kernel_name test_sincos \   -kernel_xml ./src/kernel.xml -ip_directory ./ip/
```

To use the RTL kernel during software emulation, you must provide a C-model for the kernel. The C-model must have a function prototype that compiles in hardware to the same interface used in your RTL kernel. However, the C-model does not need to be synthesizeable by the HLS tool.

You can use the `package_xo -kernel_files` option to add a C-model to the packaged RTL kernel:

```bash
package_xo -xo_path ./test.xo -kernel_name test_sincos -kernel_xml ./src/kernel.xml \   -ip_directory ./ip/ -kernel_files ./imports/sincos_cmodel.cpp
```

The `package_xo` command packages the C-model files into `cpu_sources` inside the XO. The following C-model file suffixes are automatically recognized:

- .cl = OpenCL
- .c, .cpp, .cxx = C/C++
Design Recommendations for RTL Kernels

While the RTL Kernel Wizard assists in packaging RTL designs for use within the Vitis core development kit, the underlying RTL kernels should be designed with recommendations from the *UltraFast Design Methodology Guide for Xilinx FPGAs and SoCs* (UG949).

In addition to adhering to the interface and packaging requirements, the kernels should be designed with the following performance goals in mind:

- **Memory Performance Optimizations for AXI4 Interface**
- **Managing Clocks in an RTL Kernel**
- **Quality of Results Considerations**
- **Debug and Verification Considerations**

### Memory Performance Optimizations for AXI4 Interface

The AXI4 interfaces typically connects to DDR memory controllers in the platform.

**RECOMMENDED:** For optimal frequency and resource usage, it is recommended that one interface is used per memory controller.

For best performance from the memory controller, the following is the recommended AXI interface behavior:

- Use an AXI data width that matches the native memory controller AXI data width, typically 512-bits.
- Do not use `WRAP`, `FIXED`, or sub-sized bursts.
- Use burst transfer as large as possible (up to 4k byte AXI4 protocol limit).
- Avoid use of deasserted write strobes. Deasserted write strobes can cause error-correction code (ECC) logic in the DDR memory controller to perform read-modify-write operations.
- Use pipelined AXI transactions.
- Avoid using threads if an AXI interface is only connected to one DDR controller.
- Avoid generating write address commands if the kernel does not have the ability to deliver the full write transaction (non-blocking write requests).
- Avoid generating read address commands if the kernel does not have the capacity to accept all the read data without back pressure (non-blocking read requests).
- If a read-only or write-only interfaces are desired, the ports of the unused channels can be commented out in the top level RTL file before the project is packaged into a kernel.
• Using multiple threads can cause larger resource requirements in the infrastructure IP between the kernel and the memory controllers.

Managing Clocks in an RTL Kernel

An RTL kernel can have up to two external clock interfaces; a primary clock, `ap_clk`, and an optional secondary clock, `ap_clk_2`. Both clocks can be used for clocking internal logic. However, all external RTL kernel interfaces must be clocked on the primary clock. Both primary and secondary clocks support independent automatic frequency scaling.

Note: When targeting an embedded processor platform or an Alveo accelerator card from the 2020.2 release, you can use the techniques described in Managing Clock Frequencies to map any number of kernel clocks to clock frequencies from the hardware platform.

If you require additional clocks within the RTL kernel, a frequency synthesizer such as the Clocking Wizard IP or MMCM/PLL primitive can be instantiated within the RTL kernel. Therefore, your RTL kernel can use just the primary clock, both primary and secondary clock, or primary and secondary clock along with an internal frequency synthesizer. The following shows the advantages and disadvantages of using these three RTL kernel clocking methods:

• Single input clock: `ap_clk`
  - External interfaces and internal kernel logic run at the same frequency.
  - No clock-domain-crossing (CDC) issues.
  - Frequency of `ap_clk` can automatically be scaled to allow kernel to meet timing.

• Two input clocks: `ap_clk` and `ap_clk_2`
  - Kernel logic can run at either clock frequency.
  - Need proper CDC technique in the RTL kernel to move from one frequency to another.
  - Both `ap_clk` and `ap_clk_2` can automatically scale their frequencies independently to allow the kernel to meet timing.

• Using a frequency synthesizer inside the kernel:
  - Additional device resources required to generate clocks.
  - Must have `ap_clk` and optionally `ap_clk_2` interfaces.
  - Generated clocks can have different frequencies for different CUs.
  - Kernel logic can run at any available clock frequency.
  - Need proper CDC technique to move from one frequency to another.

When using a frequency synthesizer in the RTL kernel there are some constraints you should be aware of:
1. RTL external interfaces are clocked at `ap_clk`.

2. The frequency synthesizer can have multiple output clocks that are used as internal clocks to the RTL kernel.

3. You must provide a Tcl script to downgrade DRCs related to clock resource placement in Vivado placement to prevent a DRC error from occurring. Refer to `CLOCK_DEDICATED_ROUTE` in the Vivado Design Suite Properties Reference Guide (UG912) for more information. The following is an example of the needed Tcl command that you will add to your Tcl script:

   ```tcl
   set_property CLOCK_DEDICATED_ROUTE ANY_CMT_COLUMN [get_nets pfm_top_i/static_region/base_clocking/clkwiz_kernel/inst/CLK_CORE_DRP_I/clk_inst/clk_out1]
   ```

   **Note:** This constraint should be edited to reflect the clock structure of your target platform.

4. Specify the Tcl script from step 3 for use by Vivado implementation, after optimization, by using the `v++ --vivado.prop` option as described in `--vivado Options`. The following option specifies a Tcl script for use by Vivado implementation, after completing the optimization step:

   ```bash
   --vivado.prop:run.impl_1.STEPS.OPT_DESIGN.TCL.POST={<PATH>/<Script_Name>.tcl}
   ```

5. Specify the two global clock input frequencies which can be used by the kernels (RTL or HLS-based). Use the `v++ --kernel_frequency` option to ensure the kernel input clock frequency is as expected. For example to specify one clock use:

   ```bash
   v++ --kernel_frequency 250
   ```

   For two clocks, you can specify multiple frequencies based on the clock ID. The primary clock has clock ID 0 and the secondary has clock ID 1.

   ```bash
   v++ --kernel_frequency 0:250|1:500
   ```

   **TIP:** Ensure that the PLL or MMCM output clock is locked before RTL kernel operations. Use the locked signal in the RTL kernel to ensure the clock is operating correctly.

After adding the frequency synthesizer to an RTL kernel, the generated clocks are not automatically scalable. Ensure the RTL kernel passes timing requirements, or `v++` will return an error like the following:

```text
ERROR: [VPL-1] design did not meet timing - Design did not meet timing. One or more unscalable system clocks did not meet their required target frequency. Please try specifying a clock frequency lower than 300 MHz using the `--kernel_frequency` switch for the next compilation. For all system clocks, this design is using 0 nanoseconds as the threshold worst negative slack (WNS) value. List of system clocks with timing failure.
```

In this case you will need to change the internal clock frequency, or optimize the kernel logic to meet timing.
Quality of Results Considerations

The following recommendations help improve results for timing and area:

- Pipeline all reset inputs and internally distribute resets avoiding high fanout nets.
- Reset only essential control logic flip-flops.
- Consider registering input and output signals to the extent possible.
- Understand the size of the kernel relative to the capacity of the target platforms to ensure fit, especially if multiple kernels will be instantiated.
- Recognize platforms that use stacked silicon interconnect (SSI) technology. These devices have multiple die and any logic that must cross between them should be flip-flop to flip-flop timing paths.

Debug and Verification Considerations

- RTL kernels should be verified in their own test bench using advanced verification techniques including verification components, randomization, and protocol checkers. The AXI Verification IP (VIP) is available in the Vivado IP catalog and can help with the verification of AXI interfaces. The RTL kernel example designs contain an AXI VIP-based test bench with sample stimulus files.
- Hardware emulation should be used to test the host code software integration or to view the interaction between multiple kernels.
Chapter 9

Streaming Data Transfers

Streaming Data Transfers between Kernels (K2K)

The Vitis™ core development kit also supports streaming data transfer between two kernels. Consider the situation where one kernel is performing some part of the computation, and the second kernel completes the operation after receiving the output data from the first kernel. With kernel-to-kernel streaming support, data can move directly from one kernel to another without having to transmit back through the global memory. This results in a significant performance improvement.

Host Coding Guidelines

The kernel ports involved in kernel-to-kernel streaming do not require setup using the clSetKernelArg from the host code. All kernel arguments not involved in the streaming connection should be set up using clSetKernelArg as described in Setting Kernel Arguments. However, kernel ports involved in streaming will be defined within the kernel itself, and are not addressed by the host program.

Streaming Kernel Coding Guidelines

In a kernel, the streaming interface directly sending or receiving data to another kernel streaming interface is defined by hls::stream with the ap_axiu<32, 0, 0, 0> data type. The ap_axiu<32, 0, 0, 0> data type requires the use of the ap_axi_sdata.h header file.

IMPORTANT! Host-to-kernel and kernel-to-host streaming requires the use of the qdma_axis data type. Both the ap_axiu and qdma_axis data types are defined inside the ap_axi_sdata.h header file that is distributed with the Vitis software platform installation.

The following example shows the streaming interfaces of the producer and consumer kernels.

```c
// Producer kernel - provides output as a data stream
// The example kernel code does not show any other inputs or outputs.
void kernel1 (.... , hls::stream<ap_axiu<32, 0, 0, 0> >& stream_out) {
```
Because the `hls::stream` data type is defined, the Vitis HLS tool infers `axis` interfaces. The following INTERFACE pragmas are shown as an example, but are not added to the code.

```c
#pragma HLS INTERFACE axis port=stream_out
#pragma HLS INTERFACE axis port=stream_in
```

**TIP:** These example kernels show the definition of the streaming input/output ports in the kernel signature, and the handling of the input/output stream in the kernel code. The connection of `kernel1` to `kernel2` must be defined during the kernel linking process as described in *Specifying Streaming Connections between Compute Units*.

For more information on mapping streaming connections, refer to Section III: Building and Running the Application.

---

**Free-Running Kernel**

The Vitis core development kit provides support for one or more free-running kernels. Free-running kernels have no control signal ports, and cannot be started or stopped. The *no-control signal* feature of the free-running kernel results in the following characteristics:

- The free-running kernel has no memory input or output port, and therefore it interacts with the host or other kernels (other kernels can be regular kernel or another free-running kernel) only through streams.
- When the FPGA is programmed by the binary container (*xclbin*), the free-running kernel starts running on the FPGA, and therefore it does not need the `clEnqueueTask` command from the host code.
- The kernel works on the stream data as soon as it starts receiving from the host or other kernels, and it stalls when the data is not available.
• The free-running kernel needs a special interface pragma `ap_ctrl_none` inside the kernel body.

### Host Coding for Free-Running Kernels

If the free-running kernel interacts with the host, the host code should manage the stream operation by `clCreateStream/clReadStream/clWriteStream` as discussed in Host Coding Guidelines. As the free-running kernel has no other types of inputs or outputs, such as memory ports or control ports, there is no need to specify `clSetKernelArg`. The `clEnqueueTask` is not used because the kernel works on the stream data as soon as it starts receiving from the host or other kernels, and it stalls when the data is not available.

### Coding Guidelines for Free-Running Kernels

As mentioned previously, the free-running kernel only contains `hls::stream` inputs and outputs. The recommended coding guidelines include:

- Use `hls::stream<ap_axiu<D,0,0,0>>` if the port is interacting with another stream port from the kernel.
- Use `hls::stream<qdma_axis<D,0,0,0>>` if the port is interacting with the host.
- Use the `hls::stream` data type for the function parameter causes Vitis HLS to infer an AXI4-Stream port `(axis)` for the interface.
- The free-running kernel must also specify the following special INTERFACE pragma.

```c
#pragma HLS interface ap_ctrl_none port=return
```

**IMPORTANT!** The kernel interface should not have any `#pragma HLS interface s_axilite` or `#pragma HLS interface m_axi` (as there should not be any memory or control port).

The following code example shows a free-running kernel with one input and one output communicating with another kernel. The `while(1)` loop structure contains the substance of the kernel code, which repeats as long as the kernel runs.

```c
#include <stream.h>

void kernel_top(hls::stream<ap_axiu<32,0,0,0>> &input,
                hls::stream<ap_axiu<32,0,0,0>> &output) {
  #pragma HLS interface ap_ctrl_none port=return  // Special pragma for free-running kernel
  #pragma HLS DATAFLOW  // The kernel is using DATAFLOW optimization

  while(1) {
    
  }
}
```
**TIP:** The example shows the definition of the streaming input/output ports in a free-running kernel. However, the streaming connection from the free-running kernel to or from another kernel must be defined during the kernel linking process as described in *Specifying Streaming Connections between Compute Units.*
Chapter 10

Best Practices for Acceleration with Vitis

Below are some specific things to keep in mind when developing your application code and hardware function in the Vitis™ core development kit.

- Review the Methodology for Accelerating Applications with the Vitis Software Platform section for information about acceleration methodology.
- Look to accelerate functions that have a high ratio of compute time to input and output data volume. Compute time can be greatly reduced using FPGA kernels, but data volume adds transfer latency.
- Accelerate functions that have a self-contained control structure and do not require regular synchronization with the host.
- Transfer large blocks of data from host to global device memory. One large transfer is more efficient than several smaller transfers. Run a bandwidth test to find the optimal transfer size.
- Only copy data back to host when necessary. Data written to global memory by a kernel can be directly read by another kernel. Memory resources include PLRAM (small size but fast access with lowest latency), HBM (moderate size and access speed with some latency), and DDR (large size but slow access with high latency).
- Take advantage of the multiple global memory resources to evenly distribute bandwidth across kernels.
- Maximize bandwidth usage between kernel and global memory by performing 512-bit wide bursts.
- Cache data in local memory within the kernels. Accessing local memories is much faster than accessing global memory.
- In the host application, use events and non-blocking transactions to launch multiple requests in a parallel and overlapping manner.
- In the FPGA, use different kernels to take advantage of task-level parallelism and use multiple CUs to take advantage of data-level parallelism to execute multiple tasks in parallel and further increase performance.
- Within the kernels take advantage of tasks-level with dataflow and instruction-level parallelism with loop unrolling and loop pipelining to maximize throughput.
- Some Xilinx FPGAs contain multiple partitions called super logic regions (SLRs). Keep the kernel in the same SLR as the global memory bank that it accesses.
• Use software and hardware emulation to validate your code frequently to make sure it is functionally correct.

• Frequently review the Vitis Guidance report as it provides clear and actionable feedback regarding deficiencies in your project.
Building and Running the Application

After the host program and the kernel code is written, you can build the application, which includes building the host program and platform file (xclbin). The build process follows a standard compilation and linking process for both the host program and the kernel code, followed by packaging the outputs for use. However, the first step in building the application is to identify the build target, indicating if you are building for test or simulation of the application, or building for the target hardware. After building, both the host program and the FPGA binary, you will be ready to run the application.

This section contains the following chapters:

- Setting Up the Vitis Environment
- Build Targets
- Building the Host Program
- Building the Device Binary
- Packaging the System
- Output Directories of the v++ Command
- Running Emulation
- Running the Application Hardware Build
Chapter 11

Setting Up the Vitis Environment

The Vitis™ unified software platform includes three elements that must be installed and configured to work together properly: the Vitis core development kit, the Xilinx® Runtime (XRT), and an accelerator card such as the Alveo™ Data Center accelerator card. The requirements of installation and configuration are described in Installation.

If you have the elements of the Vitis software platform installed, you need to setup the environment to run in a specific command shell by running the following scripts (.csh scripts are also provided):

```
#setup XILINX_VITIS and XILINX_VIVADO variables
source <Vitis_install_path>/settings64.sh
#setup XILINX_XRT
source /opt/xilinx/xrt/setup.sh
```

You can also specify the location of the available platforms for use with your Vitis IDE by setting the following environment variable:

```
export PLATFORM_REPO_PATHS=<path to platforms>
```

**TIP:** The `PLATFORM_REPO_PATHS` environment variable points to directories containing platform files (.xpfm).
Build Targets

The build target of the Vitis™ tool defines the nature and contents of the FPGA binary (.xclbin) created during compilation and linking. There are three different build targets: two emulation targets used for validation and debugging purposes: software emulation and hardware emulation, and the default system hardware target used to generate the FPGA binary (.xclbin) loaded into the Xilinx® device.

Compiling for an emulation target is significantly faster than compiling for the real hardware. The emulation run is performed in a simulation environment, which offers enhanced debug visibility and does not require an actual accelerator card.

Table 15: Comparison of Emulation Flows with Hardware Execution

<table>
<thead>
<tr>
<th>Software Emulation</th>
<th>Hardware Emulation</th>
<th>Hardware Execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Host application runs with a C/C++ or OpenCL™ model of the kernels.</td>
<td>Host application runs with a simulated RTL model of the kernels.</td>
<td>Host application runs with actual hardware implementation of the kernels.</td>
</tr>
<tr>
<td>Used to confirm functional correctness of the system.</td>
<td>Test the host / kernel integration, get performance estimates.</td>
<td>Confirm that the system runs correctly and with desired performance.</td>
</tr>
<tr>
<td>Fastest build time supports quick design iterations.</td>
<td>Best debug capabilities, moderate compilation time with increased visibility of the kernels.</td>
<td>Final FPGA implementation, long build time with accurate (actual) performance results.</td>
</tr>
</tbody>
</table>

Software Emulation

The main goal of software emulation (sw_emu) is to ensure functional correctness of the host program and kernels. Software emulation provides a purely functional execution, without any modeling of timing delays, or latency; it does not give any indication of the accelerator performance.

The kernel code is always compiled and running natively. The application code is either:

- Compiled and running natively on an x86 processor (Data Center platforms)
- Cross-compiler to the Arm® processor and running in an emulator (Embedded platforms)

Thus, software emulation is typically used for algorithm refinement, debugging functional issues, and letting developers iterate quickly through the code to make improvements. The software programming model of fast compilation and run iterations is preserved.
The `v++` compiler does the minimum transformation of the kernel code to create the FPGA binary to run the host program and kernel code together. Software emulation takes the C-based kernel code and compiles it with GCC. It runs each kernel as a separate C-thread. If there are multiple compute units of a single kernel, each CU is run as a separate thread. Therefore, it mimics the parallel execution model of the hardware. However, within each kernel the execution is modeled sequentially although there might be parallelism within a kernel when running on hardware. The software emulation driver implements the XRT API and acts as a bridge between the user application running XRT and the device process modeling the hardware components.

**TIP:** For RTL kernels, software emulation can be supported if a C model is associated with the kernel. The RTL Kernel Development Flow provides an option to associate C model files with the RTL kernel for support of software emulation flows.

The following describes the software emulation limitations:

- There is a global memory limit of 16 GB which should not be exceeded for simulation purposes.
- Software emulation is not supported for AI Engine kernels.
- Software emulation does not support AXI4-Stream Interfaces without Side-Channels (see Vitis High-Level Synthesis User Guide (UG1399)).

As discussed in Vitis Compiler Command, the software emulation target is specified in the `v++` command with the `-t` option:

```
v++ -t sw_emu ...
```

You can use the GDB debugger for both the host application and the kernel code, set break points or use `printf()` to print information and checkpoints. For details on how to debug the host application or the kernel during software emulation, refer to Debugging in Software Emulation.

---

**Hardware Emulation**

Hardware emulation runs an RTL simulation of the programmable logic design, where the PL kernels are integrated with a cycle-approximate model of the hardware platform.

Hardware emulation is especially useful for the following tasks:

- Checking the functional correctness of the RTL code synthesized from the C, C++, or OpenCL kernel code
- Testing the interactions between different kernels or multiple CUs
- Using hardware waveforms to gain detailed visibility into internal activity of the kernels
- Getting initial performance estimates for the application
Each kernel is compiled to a hardware model (RTL). During hardware emulation, kernels are run in the Vivado logic simulator, with a waveform viewer to examine the kernel design. Some third-party simulators are also supported as described in RTL Simulator Support. In addition, hardware emulation provides performance and resource estimates for the hardware implementation.

SystemC models are provided for the key IP used in the hardware platform, like Versal NoC/DDR memory, CIPS, PS block, AI Engine, UltraScale+ MIG DDR memory, and AXI4 SmartConnect. These IP models are used during hardware emulation to improve simulation performance and results.

In hardware emulation, compile and execution times are longer than software emulation, but it provides a detailed, cycle-accurate, view of kernel activity. Xilinx recommends using small data sets for validation during hardware emulation to keep runtimes manageable.

**IMPORTANT!** The DDR memory model and the memory interface generator (MIG) model used in hardware emulation are high-level simulation models. These models provide good simulation performance, but only approximate latency values and are not cycle-accurate like the kernels. Therefore, performance numbers shown in the profile summary report are approximate, and should be used for guidance and for comparing relative performance between different kernel implementations.

As discussed in Vitis Compiler Command, the hardware emulation target is specified in the `v++` command with the `-t` option:

```
v++ -t hw_emu ...
```

## System Hardware Target

When the build target is the hardware, `v++` builds the FPGA binary for the Xilinx device by running Vivado synthesis and implementation on the design. It is normal for this build target to take a longer period of time than generating either the software or hardware emulation targets in the Vitis IDE. However, the final FPGA binary can be loaded into the hardware of the accelerator card, or embedded processor platform, and the application can be run in its actual operating environment.

As discussed in Vitis Compiler Command, the system hardware target is specified in the `v++` command with the `-t` option:

```
v++ -t hw ...
```
Building the Host Program

The host program, written in C/C++ using either the XRT native API or OpenCL™ API calls, is built using the GNU C++ compiler (g++) which is based on GNU compiler collection (GCC). Each source file is compiled to an object file (.o) and linked with the Xilinx® runtime (XRT) shared library to create the executable which runs on the host CPU.

**TIP:** g++ supports many standard GCC options which are not documented here. For information refer to the GCC Option Summary.

Compiling and Linking for x86

**TIP:** Set up the command shell or window as described in Setting Up the Vitis Environment prior to running the tools.

Each source file of the host application is compiled into an object file (.o) using the g++ compiler.

```g++ ... -c <source_file1> <source_file2> ... <source_fileN>```

The generated object files (.o) are linked with the Xilinx Runtime (XRT) shared library to create the executable host program. Linking is performed using the -l option.

```g++ ... -l <object_file1.o> ... <object_fileN.o>```

Compiling and linking for x86 follows the standard g++ flow. The only requirement is to include the XRT header files and link the XRT shared libraries.

When compiling the source code, the following g++ options are required:

- -I$XILINX_XRT/include/: XRT include directory.
- -I$XILINX_VIVADO/include: Vivado tools include directory.
- -std=c++11: Define the C++ language standard.

When linking the executable, the following g++ options are required:

- -L$XILINX_XRT/lib/: Look in XRT library.
-lOpenCL: Search the named library during linking.
-lpthread: Search the named library during linking.
-lrt: Search the named library during linking.
-lstdc++: Search the named library during linking.

Note: In the Vitis Examples you may see the addition of xcl2.cpp source file, and the -I../libs/xcl2 include statement. These additions to the host program and g++ command provide access to helper utilities used by the example code, but are generally not required for your own code.

Building XRT Native API

XRT provides a native XRT API for C, C++, and Python, as described on the XRT site at https://xilinx.github.io/XRT/2020.2/html/xrt_native_apis.html. To use the native XRT API, the host application must link with the xrt_coreutil library. The command line uses a few different settings as shown in the following example, which combines compilation and linking:

```
g++ -g -std=c++14 -I$XILINX_XRT/include -L$XILINX_XRT/lib -lxrt_coreutil -lpthread \
-o host.exe host.cpp
```

IMPORTANT! The XRT API requires the use of -std=c++14.

Compiling and Linking for Arm

TIP: Set up the command shell or window as described in Setting Up the Vitis Environment prior to running the tools.

The host program (host.exe), is cross-compiled for an Arm processor, and linked using the following two step process:

1. Compile the host.cpp into an object file (.o) using the GNU Arm cross-compiler version of g++:

```
Note: aarch64 is used for Zynq® UltraScale+™ (A53) and Versal™ (A72) devices. aarch32 is used for Zynq-7000 SoC (A9) and the tool chain is in a different location.

```

```
$XILINX_VITIS/gnu/aarch64/lin/aarch64-linux/bin/aarch64-linux-gnu-g++ \
-D__USE_XOPEN2K8 -I$SYSROOT/usr/include/xrt -I$XILINX_VIVADO/include \
-I$SYSROOT/usr/include -c -fmessage-length=0 -std=c++14 \
--sysroot=$SYSROOT -o src/host.o ../src/host.cpp
```

2. Link the object file with required libraries to build the executable application.

```
$XILINX_VITIS/gnu/aarch64/lin/aarch64-linux/bin/aarch64-linux-gnu-g++ \
-o host.exe src/host.o -lxilinuxopencl -lpthread -lrt -lstdc++ -lgmp -lxrt_core \
-L$SYSROOT/usr/lib/ --sysroot=$SYSROOT
```
When compiling the application for use with an embedded process, you must specify the *sysroot* for the application. The *sysroot* is part of the platform where the basic system root file structure is defined, and is installed as described in *Installing Embedded Platforms*.

**IMPORTANT!** The above examples rely on the use of `$SYSROOT` environment variable that must be used to specify the location of the *sysroot* for your embedded platform.

The following are key elements to compiling the host code for an edge platform:

- **Compilation:**
  - The cross compiler needed is the `aarch64-linux-gnu-g++` found in the Vitis installation hierarchy.
  - The required include paths are:
    - `$SYSROOT/usr/include`
    - `$SYSROOT/usr/include/xrt`
    - `$XILINX_VIVADO/include`

- **Linking:**
  - `$SYSROOT/usr/lib`: Library paths location.
  - `xilinuxopencl`: XRT required library.
  - `pthread`: XRT required library.
  - `rt`: XRT required library.
  - `stdc++`: XRT required library.
  - `gmp`: XRT required library.
  - `xrt_core`: XRT required library.
Building XRT Native API

XRT provides a native XRT API for C, C++, and Python, as described on the XRT site at https://xilinx.github.io/XRT/2020.2/html/xrt_native_apis.html. To use the native XRT API, the host application must link with the xrt_coreutil library instead of the xlinxopencl library. The command line uses a few different settings as shown in the following example for compilation and linking:

```
$XILINX_VITIS/gnu/aarch64/lin/aarch64-linux/bin/aarch64-linux-gnu-g++ -c \ 
-D__USE_XOPEN2K8 -I$SYSROOT/usr/include/xrt -I$XILINX_VIVADO/include \ 
-I$SYSROOT/usr/include -fmessage-length=0 -std=c++14 --sysroot=$SYSROOT \ 
-o src/host.o ../src/host.cpp
```

```
$XILINX_VITIS/gnu/aarch64/lin/aarch64-linux/bin/aarch64-linux-gnu-g++ -l \ 
-lxrt_coreutil -lpthread -lrt -lstdc++ -lgmp -lxrt_core -L$SYSROOT/usr/lib/ \ 
--sysroot=$SYSROOT -o host.exe src/host.o
```
Building the Device Binary

The kernel code is written in C, C++, OpenCL™ C, or RTL, and is built by compiling the kernel code into a Xilinx® object (XO) file, and linking the XO files into a device binary (XCLBIN) file, as shown in the following figure.

**Figure 23: Device Build Process**

The process, as outlined above, has two steps:

1. Build the Xilinx object files from the kernel source code.
   
   - For C, C++, or OpenCL kernels, the `v++ -c` command compiles the source code into Xilinx object (XO) files. Multiple kernels are compiled into separate XO files.
   
   - For RTL kernels, the `package_xo` command produces the XO file to be used for linking. Refer to RTN Kernels for more information.
You can also create kernel object (XO) files working directly in the Vitis™ HLS tool. Refer to Compiling Kernels with the Vitis HLS for more information.

2. After compilation, the `v++ -l` command links one or multiple kernel objects (XO), together with the hardware platform XSA file, to produce the device binary XCLBIN file.

TIP: The `v++` command can be used from the command line, in scripts, or a build system like `make`, and can also be used through the Vitis IDE as discussed in Section VII: Using the Vitis IDE.

Compiling Kernels with the Vitis Compiler

IMPORTANT! Set up the command shell or window as described in Setting Up the Vitis Environment prior to running the tools.

The first stage in building the xclbin file is to compile the kernel code using the Xilinx Vitis compiler. There are multiple `v++` options that need to be used to correctly compile your kernel. The following is an example command line to compile the `vadd` kernel:

```
v++ -t sw_emu --platform xilinx_u200_xdma_201830_2 -c -k vadd \
-I'./src' -o'vadd.sw_emu.xo' ./src/vadd.cpp
```

The various arguments used are described below. Note that some of the arguments are required.

- `-t <arg>`: Specifies the build target, as discussed in Build Targets. Software emulation (sw_emu) is used as an example. Optional. The default is hw.
- `--platform <arg>`: Specifies the accelerator platform for the build. This is required because runtime features, and the target platform are linked as part of the FPGA binary. To compile a kernel for an embedded processor application, specify an embedded processor platform: `--platform $PLATFORM_REPO_PATHS/zcu102_base/zcu102_base.xpfm`.
- `-c`: Compile the kernel. Required. The kernel must be compiled (-c) and linked (-l) in two separate steps.
- `-k <arg>`: Name of the kernel associated with the source files.
- `-o'<output>.xo'`: Specify the shared object file output by the compiler. Optional.
- `<source_file>`: Specify source files for the kernel. Multiple source files can be specified. Required.

The above list is a sample of the extensive options available. Refer to Vitis Compiler Command for details of the various command line options. Refer to Output Directories of the `v++ Command` to get an understanding of the location of various output files.
Compiling Kernels with the Vitis HLS

The use model described for the Vitis core development kit is a top-down approach, starting with C/C++ or OpenCL code, and working toward compiled kernels. However, you can also directly develop the kernel to produce a Xilinx object (XO) file to be paired for linking using `v++` to produce the `.xclbin`. This approach can be used for C/C++ kernels using the Vitis HLS tool, which is the focus of this section, or RTL kernels using the Vivado Design Suite. Refer to RTL Kernels for more information.

The approach of developing the kernel directly, either in RTL or C/C++, to produce an XO file, is sometimes referred to as the bottom-up flow. This allows you to validate kernel performance and perform optimizations within the Vitis HLS tool, and export the Xilinx object file for use in the Vitis application acceleration development flow. Refer to the Vitis HLS Flow for more information on using that tool.

![Figure 24: Vitis HLS Bottom-Up Flow](X21852-101920)

The benefits of the Vitis HLS bottom-up flow can include:

- Design, validate, and optimize the kernel separately from the main application.
- Enables a team approach to design, with collaboration on host program and kernel development.
- Specific kernel optimizations are preserved in the XO file.
- A collection of XO files can be used and reused like a library.

Creating Kernels in the Vitis HLS

Generating kernels from C/C++ code for use in the Vitis core development kit follows the standard Vitis HLS process. However, because the kernel is required to operate in the Vitis software platform, the standard kernel requirements must be satisfied (see Kernel Properties). Most importantly, the interfaces must be modeled as AXI memory interfaces, except for scalar parameters which are mapped to an AXI4-Lite interface. Vitis HLS automatically defines the interface ports to meet the standard kernel requirements when using the Vitis Bottom Up Flow as described here.
The process for creating and compiling your HLS kernel is outlined briefly below. You should refer to Creating a New Vitis HLS Project in the Vitis HLS Flow documentation for a more complete description of this process.

1. Launch Vitis HLS to open the integrated design environment (IDE), and specify File → New Project.
2. In the New Vitis HLS Project wizard, specify the Project name, define the Location for the project, and click Next.
3. In the Add/Remove Files page, click Add Files to add the kernel source code to the project. Select Top Function to define the kernel function by clicking the Browse button, and click Next when done.
4. You can specify a C-based simulation test bench if you have one available, by clicking Add Files, or skip this by clicking Next.

**TIP:** As discussed in the Vitis HLS documentation, the use of a test bench is strongly recommended.

5. In the Solution Configuration page, you must specify the Clock Period for the kernel.
6. Choose the target platform by clicking the browse button (...) in the Part Selection field to open the Device Selection Dialog box. Select Boards, and select the target platform for your compiled kernel, as shown below. Click OK to select the platform and return to the Solution Configuration page.

7. In the Solution Configuration page, select the Vitis Kernel Flow Target drop-down menu under Flow Target, and click Finish to complete the process and create your HLS kernel project.

⭐️ IMPORTANT! You must select the Vitis Kernel Flow Target to generate the Xilinx object (XO) file from the project.

When the HLS project has been created you can Run C-Synthesis to compile the kernel code. Refer to the Vitis HLS documentation for a complete description of the HLS tool flow.

After synthesis is completed, the kernel can be exported as an XO file for use in the Vitis core development kit. The export command is available through the Solution→Export RTL command from the main menu.

Specify the file location, and the kernel is exported as a Xilinx object XO file.

The (XO) file can be used as an input file during the v++ linking process. Refer to Linking the Kernels for more information. You can also add it to an application project in the Vitis IDE, as discussed in Creating a Vitis IDE Project.

However, keep in mind that HLS kernels, created in the bottom-up flow described here, have certain limitations when used in the Vitis application acceleration development flow. Software emulation is not supported for applications using HLS kernels, because duplicated header file dependencies can create issues. GDB debug is not supported in the hardware emulation flow for HLS kernels, or RTL kernels.

**Vitis HLS Script for Creating Kernels**

If you run HLS synthesis through Tcl scripts, you can edit the following script to create HLS kernels as previously described:

```tcl
# Define variables for your HLS kernel:
set projName <proj_name>
set krnlName <kernel_name>
set krnlFile <kernel_source_code>
set krnlTB <kernel_test_bench>
set krnlPlatform <target_part>
set path <path_to_project>

#Script to create and output HLS kernel
open_project $projName
set_top $krnlName
add_files $krnlFile
add_files -tb $krnlTB
open_solution "solution1"
set_part $krnlPlatform
create_clock -period 10 -name default
```
config_flow -target vitis
csim_design
csynth_design
cosim_design
export_design -flow impl -format xo -output "./hlsKernel/hlsKernel.xo"

Run the HLS kernel script by using the following command after setting up your environment as discussed in Setting Up the Vitis Environment.

vitis_hls -f <hls_kernel_script>.tcl

---

Packaging RTL Kernels with package_xo

Kernels written in RTL are compiled in the Vivado tool using the package_xo command line utility which generates a Xilinx object (XO) file which can subsequently used by the v++ command, during the linking stage. (See package_xo Command.) The process for creating RTL kernels, and using the package_xo command to generate an XO file is described in RTL Kernels.

---

Linking the Kernels

TIP: Set up the command shell or window as described in Setting Up the Vitis Environment prior to running the tools.

The kernel compilation process results in a Xilinx object (XO) file whether the kernel is written in C/C++, OpenCL C, or RTL. During the linking stage, XO files from different kernels are linked with the platform to create the FPGA binary container file (.xclbin) used by the host program.

Similar to compiling, linking requires several options. The following is an example command line to link the vadd kernel binary:

```
v++ -t sw_emu --platform xilinx_u200_xdma_201830_2 --link vadd.sw_emu.xo \
-o 'vadd.sw_emu.xclbin' --config ./connectivity.cfg
```

This command contains the following arguments:

- `-t <arg>`: Specifies the build target. Software emulation (sw_emu) is used as an example. When linking, you must use the same `-t` and `--platform` arguments as specified when the input (XO) file was compiled.
- `--platform <arg>`: Specifies the platform to link the kernels with. To link the kernels for an embedded processor application, you simply specify an embedded processor platform: `--platform $PLATFORM_REPO_PATHS/zcu102_base/zcu102_base.xpfm`
- `--link`: Link the kernels and platform into an FPGA binary file (xclbin).
• `<input>.xo`: Input object file. Multiple object files can be specified to build into the `.xclbin`.

• `-o`'<output>.xclbin'`: Specify the output file name. The output file in the link stage will be an `.xclbin` file. The default output name is `a.xclbin`.

• `--config ./connectivity.cfg`: Specify a configuration file that is used to provide `v++` command options for a variety of uses. Refer to Vitis Compiler Command for more information on the `--config` option.

TIP: Refer to Output Directories of the `v++` Command to get an understanding of the location of various output files.

Beyond simply linking the Xilinx object (XO) files, the linking process is also where important architectural details are determined. In particular, this is where the number of compute unit (CUs) to instantiate into hardware is specified, connections from kernel ports to global memory are assigned, and CUs are assigned to SLRs. The following sections discuss some of these build options.

**Creating Multiple Instances of a Kernel**

By default, the linker builds a single hardware instance from a kernel. If the host program will execute the same kernel multiple times, due to data processing requirements for instance, then it must execute the kernel on the hardware accelerator in a sequential manner. This can impact overall application performance. However, you can customize the kernel linking stage to instantiate multiple hardware compute units (CUs) from a single kernel. This can improve performance as the host program can now make multiple overlapping kernel calls, executing kernels concurrently by running separate compute units.

Multiple CUs of a kernel can be created by using the `connectivity.nk` option in the `v++` config file during linking. Edit a config file to include the needed options, and specify it in the `v++` command line with the `--config` option, as described in Vitis Compiler Command.

For example, for the `vadd` kernel, two hardware instances can be implemented in the config file as follows:

```plaintext
[connectivity]
#nk=<kernel_name>:<number>:<cu_name>.<cu_name>...  
nk=vadd:2
```

Where:

• `<kernel_name>`: Specifies the name of the kernel to instantiate multiple times.

• `<number>`: The number of kernel instances, or CUs, to implement in hardware.

• `<cu_name>.<cu_name>...`: Specifies the instance names for the specified number of instances. This is optional, and the CU name will default to `kernel_1` when it is not specified.
Then the config file is specified on the `v++` command line:

```
v++ --config vadd_config.cfg ...
```

In the `vadd` example above, the result is two instances of the `vadd` kernel, named `vadd_1` and `vadd_2`.

---

**TIP**: You can check the results by using the `xclbinutil` command to examine the contents of the `xclbin` file. Refer to `xclbinutil Utility`.

---

The following example results in three CUs of the `vadd` kernel, named `vadd_X`, `vadd_Y`, and `vadd_Z` in the `xclbin` binary file:

```
[connectivity]
k=vadd:3:vadd_X.vadd_Y.vadd_Z
```

## Mapping Kernel Ports to Memory

The link phase is when the memory ports of the kernels are connected to memory resources which include DDR, HBM, and PLRAM. By default, when the `xclbin` file is produced during the `v++` linking process, all kernel memory interfaces are connected to the same global memory bank (or `gmem`). As a result, only one kernel interface can transfer data to/from the memory bank at one time, limiting the performance of the application due to memory access.

While the Vitis compiler can automatically connect CU to global memory resources, you can also manually specify which global memory bank each kernel port (or interface) is connected to. Proper configuration of kernel to memory connectivity is important to maximize bandwidth, optimize data transfers, and improve overall performance. Even if there is only one compute unit in the device, mapping its input and output ports to different global memory banks can improve performance by enabling simultaneous accesses to input and output data.

---

**IMPORTANT!** Up to 15 kernel ports can be connected to a single global memory bank. Therefore, if there are more than 15 memory interfaces, then you must explicitly perform the memory mapping as described here, using the `--connectivity.sp` option to distribute connections across different memory banks.

---

The following example is based on the `Kernel Interfaces` example code. Start by assigning the kernel arguments to separate bundles to increase the available ports, then assign those ports to separate memory banks:

1. Assign interfaces to separate bundles in the kernel code:

   ```c
   void cnn( int *pixel, // Input pixel
             int *weights, // Input Weight Matrix
             int *out, // Output pixel
             ... // Other input or Output ports
   
   #pragma HLS INTERFACE m_axi port=pixel offset=slave bundle=gmem
   #pragma HLS INTERFACE m_axi port=weights offset=slave bundle=gmem1
   #pragma HLS INTERFACE m_axi port=out offset=slave bundle=gmem
   ```
Note that the memory interface inputs pixel and weights are assigned different bundle names in the example above. This creates two separate ports that can be assigned to separate global memory banks.

**IMPORTANT!** You must specify bundle= names using all lowercase characters to be able to assign it to a specific memory bank using the --connectivity.sp option.

2. Edit a config file to include the --connectivity.sp option, and specify it in the v++ command line with the --config option, as described in Vitis Compiler Command.

For example, for the cnn kernel shown above, the connectivity.sp option in the config file would be as follows:

```
[connectivity]
#sp=<compute_unit_name>.<interface_name>:<bank name>
sp=cnn_1.m_axi_gmem:DDR[0]
sp=cnn_1.m_axi_gmem1:DDR[1]
```

Where:

- `<compute_unit_name>` is an instance name of the CU as determined by the connectivity.nk option, described in Creating Multiple Instances of a Kernel, or is simply `<kernel_name>_1` if multiple CUs are not specified.

- `<interface_name>` is the name of the kernel port as defined by the HLS INTERFACE pragma, including m_axi_ and the bundle name. In the cnn kernel above, the ports would be m_axi_gmem and m_axi_gmem1.

  **TIP:** If the port is not specified as part of a bundle, then the `<interface_name>` is simply the specified port name, without the m_axi_ prefix.

- `<bank_name>` is denoted as DDR[0], DDR[1], DDR[2], and DDR[3] for a platform with four DDR banks. You can also specify the memory as a contiguous range of banks, such as DDR[0:2], in which case XRT will assign the memory bank at run time.

Some platforms also provide support for PLRAM, HBM, HP or MIG memory, in which case you would use PLRAM[0], HBM[0], HP[0] or MIG[0]. You can use the platforminfo utility to get information on the global memory banks available in a specified platform. Refer to platforminfo Utility for more information.

In platforms that include both DDR and HBM memory banks, kernels must use separate AXI interfaces to access the different memories. DDR and PLRAM access can be shared from a single port.

**IMPORTANT!** The customized bank connection needs to be reflected in the host code as well, as described in Assigning DDR Bank in Host Code.
Using PCIe Slave-Bridge to Connect to Host Memory

The PCIe® Slave-Bridge IP is provided on some data center platforms to let kernels access directly to host memory. Configuring the device binary to connect to memory requires changing the link specified by the `--connectivity.sp` command below. It also requires changes to the accelerator card setup and your host application as described at PCIe Slave-Bridge in the XRT documentation.

```plaintext
[connectivity]
## Syntax
##sp=<cu_name>.<interface_name>:HOST[0]
sp=cnn_1.m_axi_gmem:HOST[0]
```

In the command syntax above, the CU name and interface name are the same, but the bank name is hard-coded to HOST[0].

**HBM Configuration and Use**

Some algorithms are memory bound, limited by the 77GB/s bandwidth available on DDR-based Alveo cards. For those applications there are HBM (High Bandwidth Memory) based Alveo cards, providing up to 460 GB/s memory bandwidth. For the Alveo implementation, 2 16-layer HBM (HBM2 specification) stacks are incorporated into the FPGA package and connected into the FPGA fabric with an interposer. A high-level diagram of the two HBM stacks is as follows.

![High-Level Diagram of Two HBM Stacks](image.png)

This implementation provides:

- 8GB HBM memory
- 32 256MB HBM segments, called pseudo channels (PCs)
- An independent AXI channel for communication with the FPGA through a segmented crossbar switch per pseudo channel
• A two-channel memory controller per two PCs
• 14.375 GB/s max theoretical bandwidth per PC
• 460 GB/S (32 * 14.375 GB/s) max theoretical bandwidth for the HBM subsystem

Although each PC has a theoretical max performance of 14.375 GB/s, this is less than the theoretical max of 19.25 GB/s for a DDR channel. To get better than DDR performance, designs must efficiently use multiple AXI masters into the HBM subsystem. The programmable logic has 32 HBM AXI interfaces that can access any memory location in any of the PCs on either of the HBM stacks through a built-in switch providing access to the full 8 GB memory space. For more detailed information on the HBM, refer to AXI High Bandwidth Controller LogiCORE IP Product Guide (PG276).

Note: Because of the complexity and flexibility of the built-in switch, there are many combinations that result in congestion at a particular memory location or in the switch itself. Interleaved read and write transactions cause a drop in efficiency with respect to read-only or write-only due to memory controller timing parameters (bus turnaround). Write transactions that span both HBM stacks will also experience degraded performance, and should be avoided. It is important to plan memory accesses so that kernels access limited memory where possible, and to isolate the memory accesses for different kernels into different HBM PCs.

Connection to the HBM is managed by the HBM Memory Subsystem (HMSS) IP, which enables all HBM PCs, and automatically connects the XDMA to the HBM for host access to global memory. When used with the Vitis compiler, the HMSS is automatically customized to activate only the necessary memory controllers and ports as specified by the --connectivity.sp option to connect both the user kernels and the XDMA to those memory controllers for optimal bandwidth and latency. Refer to the Using HBM Tutorial for additional information and examples.

In the following config file example, the kernel input ports in1 and in2 are connected to HBM PCs 0 and 1 respectively, and writes output buffer out to HBM PCs 3–4. Each HBM PC is 256 MB, giving a total of 1 GB of memory access for this kernel.

```
[connectivity]
sp=krnl.in1:HBM[0]
sp=krnl.in2:HBM[1]
sp=krnl.out:HBM[3:4]
```

Note: In the config file, only the mapping to the HBM pseudo channel is defined, and each AXI interface should only access a contiguous subset of the available 32 HBM PCs. The HMSS chooses the appropriate HBM port to access memory and to maximize bandwidth and minimize latency.

The HBM ports are located in the bottom SLR of the device. The HMSS automatically handles the placement and timing complexities of AXI interfaces crossing super logic regions (SLR) in SSI technology devices. By default, without specifying the --connectivity.sp or --connectivity.slr options on v++, all kernel AXI interfaces access HBM[0] and all kernels are assigned to SLR0. However, you can specify the SLR assignments of kernels using the --connectivity.slr option. Refer to Assigning Compute Units to SLRs for more information.
Random Access and the RAMA IP

HBM performs well in applications where sequential data access is required. However, for applications requiring random data access, performance can vary significantly depending on the application requirements (for example, the ratio of read and write operations, minimum transaction size, and size of the memory space being addressed). In these cases, the addition of the Random Access Memory Attachment (RAMA) IP to the target platform can significantly improve random memory access efficiency in cases where the required memory exceeds the 256 MB limit of a single HBM PC. Refer to RAMA LogiCORE IP Product Guide (PG310) for more information.

TIP: To effectively use the RAMA IP in your application the kernel should access memory from multiple HBM PCs and should use a static single ID on the AXI transaction ID ports (AxID), or slowly changing (pseudo-static) AXI transaction IDs. If these conditions are not met, the thread creation used in the RAMA IP to improve performance has little effect, and consumes programmable logic resources for no purpose.

Add the RAMA IP to the target platform during the system linking process using the following v++ command option to specify a Tcl script to define the ports of interest:

```bash
v++ -l --advanced.param compiler.userPreSysLinkOverlayTcl=<path_to>/user_tcl_file.tcl
```

Within this user-specified Tcl script, an API is provided to let you configure the HMSS resource:

```bash
hbm_memory_subsystem::ra_master_interface <Endpoint AXI master interface>
```

The following example has two AXI master ports (M00_AXI and M01_AXI) for random access:

```bash
hbm_memory_subsystem::ra_master_interface [get_bd_intf_pins dummy/M00_AXI]
hbm_memory_subsystem::ra_master_interface [get_bd_intf_pins dummy/M01_AXI]
validate_bd_design -force
```

It is important to end the Tcl script with the `validate_bd_design` command as shown above to allow the information to be collected correctly by the HBM subsystem, and the block design to be updated.

PLRAM Configuration and Use

Alveo accelerator cards contain HBM DRAM and DDR DRAM memory resources. In some accelerator cards, an additional memory resource available is internal FPGA PLRAM (UltraRAM and BlockRAM). Supporting platforms typically contain instances of PLRAM in each SLR. The size and type of each PLRAM can be configured on the target platform before kernels or Compute Units are linked into the system.
You can use a Tcl script to configure the PLRAM before system linking occurs. The use of the Tcl script can be enabled on the v++ command line as follows:

```bash
v++ -l --advanced.param compiler.userPreSysLinkOverlayTcl=<path_to>/user_tcl_file.tcl
```

Within this user-specified Tcl script, an API is provided to let you configure the PLRAM instance or memory resource:

```tcl
sdx_memory_subsystem::update_plram_specification <memory_subsystem_bdcell> <plram_resource> <plram_specification>
```

The `<plram_specification>` is a Tcl dictionary consisting of the following entries (entries below are the default values for each instance in the platform):

```tcl
{
  SIZE 128K # Up to 4M
  AXI_DATA_WIDTH 512 # Up to 512
  SLR_ASSIGNMENT SLR0 # SLR0 / SLR1 / SLR2
  READ_LATENCY 1 # To optimise timing path
  MEMORY_PRIMITIVE BRAM # BRAM or URAM
}
```

In the example below, PLRAM_MEM00 is changed to be 2 MB in size and composed of UltraRAM; PLRAM_MEM01 is changed to be 4 MB in size and composed of UltraRAM. PLRAM_MEM00 and PLRAM_MEM01 correspond to the --conectivity.sp memory resources PLRAM[0] and PLRAM[1].

```tcl
# Setup PLRAM
sdx_memory_subsystem::update_plram_specification [get_bd_cells /memory_subsystem] PLRAM_MEM00 { SIZE 2M AXI_DATA_WIDTH 512 SLR_ASSIGNMENT SLR0 READ_LATENCY 10 MEMORY_PRIMITIVE URAM}

sdx_memory_subsystem::update_plram_specification [get_bd_cells /memory_subsystem] PLRAM_MEM01 { SIZE 4M AXI_DATA_WIDTH 512 SLR_ASSIGNMENT SLR0 READ_LATENCY 10 MEMORY_PRIMITIVE URAM}

validate_bd_design -force
save_bd_design
```

The `READ_LATENCY` is an important attribute, because it sets the number of pipeline stages between memories cascaded in depth. This varies by design, and affects the timing QoR of the platform and the eventual kernel clock rate. In the example above for PLRAM_MEM01:

- 4 MB of memory are required in total.
- Each UltraRAM is 32 KB (64 bits wide). 4 MB × 32 KB → 128 UltraRAMs in total.
- Each PLRAM instance is 512 bits wide → 8 UltraRAMs are required in width.
- 128 total UltraRAMs with 8 UltraRAMs in width → 16 UltraRAMs in depth.
- A good rule of thumb is to pick a read latency of depth/2 + 2 → in this case, `READ_LATENCY` = 10.
This allows a pipeline on every second UltraRAM, resulting in the following:

- Good timing performance between UltraRAMs.
- Placement flexibility; not all UltraRAMs need to be placed in the same UltraRAM column for cascade.

### Specifying Streaming Connections between Compute Units

The Vitis core development kit supports streaming data transfer between two kernels, allowing data to move directly from one kernel to another without having to transmit back through global memory. However, the process has to be implemented in the kernel code itself, as described in Streaming Data Transfers between Kernels (K2K), and also specified during the kernel build process.

**Note:** This also supports streaming connections to/from free running kernels as described in Free-Running Kernel.

The streaming data ports of kernels can be connected during v++ linking using the `--connectivity.sc` option. This option can be specified at the command line, or from a config file that is specified using the `--config` option, as described in Vitis Compiler Command.

To connect the streaming output port of a producer kernel to the streaming input port of a consumer kernel, setup the connection in the `v++` config file using the `connectivity.stream_connect` option as follows:

```
[connectivity]
#stream_connect=<cu_name>.<output_port>:<cu_name>.<input_port>:
   [<fifo_depth>]
stream_connect=vadd_1.stream_out:vadd_2.stream_in
```

Where:

- `<cu_name>` is an instance name of the CU as determined by the `connectivity.nk` option, described in Creating Multiple Instances of a Kernel.
- `<output_port>` or `<input_port>` is the streaming port defined in the producer or consumer kernel as described in Streaming Kernel Coding Guidelines, or as described in Coding Guidelines for Free-Running Kernels.
- `[:<fifo_depth>]` inserts a FIFO of the specified depth between the two streaming ports to prevent stalls. The value is specified as an integer.
Assigning Compute Units to SLRs

Currently, Xilinx devices on Data Center accelerator cards use stacked silicon consisting of several Super Logic Regions (SLRs) to provide device resources, including global memory. For best performance, when assigning ports to global memory banks, as described in Mapping Kernel Ports to Memory, it is best that the CU instance is assigned to the same SLR as the global memory it is connected to. In this case, you will want to manually assign the kernel instance, or CU into the same SLR as the global memory to ensure the best performance.

A CU can be assigned to an SLR using the `connectivity.slr` option in a config file. The syntax of the `connectivity.slr` option in the config file is as follows:

```plaintext
[connectivity]
#slr=<compute_unit_name>:<slr_ID>
slr=vadd_1:SLR2
slr=vadd_2:SLR3
```

where:

- `<compute_unit_name>` is an instance name of the CU as determined by the `connectivity.nk` option, described in Creating Multiple Instances of a Kernel, or is simply `<kernel_name>_1` if multiple CUs are not specified.
- `<slr_ID>` is the SLR number to which the CU is assigned, in the form SLR0, SLR1,...

The assignment of a CU to an SLR must be specified for each CU separately, but is not required. If an assigned CU is connected to global memory located in another SLR, the tool will automatically insert SLR crossing registers to help with timing closure. In the absence of an SLR assignment, the `v++` linker is free to assign the CU to any SLR.

After editing the config file to include the SLR assignments, you can use it during the `v++` linking process by specifying the config file using the `--config` option:

```
v++ -l --config config_slr.cfg ...
```

Managing Clock Frequencies

**IMPORTANT!** The `--clock` options described here are not supported by legacy platforms of Data Center accelerator cards. On older platforms, kernels operate at the default operating frequency of the platform.

Generally, in embedded processor platforms, and also in newer Data Center accelerator cards, the device binary can connect multiple kernels to the platform with different clock frequencies. Each kernel, or unique instance of the kernel can connect to a specified clock frequency, or multiple clocks, and different kernels can use different clock frequencies generated by the platform.
You can specify the kernel frequency during compilation using the `--hls.clock` command. This lets you compile the kernel targeting the specified frequency, and lets the Vitis HLS tool perform validation of the kernel logic at the specified frequency. This is just an implementation target for compilation, but does provide optimization and feedback.

During the linking process, when the kernels are connected to the platform to build the device binary, you can specify the clock frequency for the kernels using the `--clock` Options of the `v++` command.

1. Compile the HLS code using the Vitis compiler. For RTL kernels, go to step 2.

   ```bash
   v++ -c -k <krnl_name> --hls.clock freqHz:<krnl_name>
   ```

   **Note:** To change the frequency at which Vitis HLS kernels are compiled, use `--hls.clock arg:kernelName`. The `arg` must be in Hz (for example, `250000000Hz` is 250 MHz).

2. During linking, specify the clock frequency or clock ID for each clock signal in a kernel with the following command:

   ```bash
   v++ -l ... --clock.freqHz <freqHz>:kernelName.ap_clk
   ```

   When specifying the clock ID, the kernel frequency is specified by the clock frequency of the specified clock on the platform. When specifying the clock frequency, the platform attempts to create the specified frequency by scaling the platform clocks. In some cases, the clock frequency can only be achieved in some approximation, and you can specify the `--clock.tolerance` or `--clock.default_tolerance` to indicate an acceptable range. If the specified clock frequency cannot be scaled within the acceptable tolerance, a warning is issued and the kernel is connected to the default clock.

**Managing Vivado Synthesis and Implementation Results**

---

**TIP:** This topic requires an understanding of the Vivado Design Suite tools and design methodology as described in UltraFast Design Methodology Guide for Xilinx FPGAs and SoCs (UG949).

In most cases, the Vitis environment completely abstracts away the underlying process of synthesis and implementation of the programmable logic region, as the CUs are linked with the hardware platform and the FPGA binary (`xclbin`) is generated. This removes the application developer from the typical hardware development process, and the need to manage constraints such as logic placement and routing delays. The Vitis tool automates much of the FPGA implementation process.
However, in some cases you might want to exercise some control over the synthesis and implementation processes deployed by the Vitis compiler, especially when large designs are being implemented. Towards this end, the Vitis tool offers some control through specific options that can be specified in a `v++` configuration file, or from the command line. The following are some of the methods you can interact with and control the Vivado synthesis and implementation results.

- Using the `--vivado` options to manage the Vivado tool.
- Using multiple implementation strategies to achieve timing closure on challenging designs.
- Using the `-to_step` and `-from_step` options to run the compilation or linking process to a specific step, perform some manual intervention on the design, and resume from that step.
- Interactively editing the Vivado project, and using the results for generating the FPGA binary.

### Using the -vivado and -advanced Options

Using the `--vivado` option, as described in `--vivado Options`, and the `--advanced` option as described in `--advanced Options`, you can perform a number of interventions on the standard Vivado synthesis or implementation.

1. Pass Tcl scripts with custom design constraints or scripted operations.

   You can create Tcl scripts to assign XDC design constraints to objects in the design, and pass these Tcl scripts to the Vivado tools using the PRE and POST Tcl script properties of the synthesis and implementation steps. For more information on Tcl scripting, refer to the Vivado Design Suite User Guide: Using Tcl Scripting (UG894). While there is only one synthesis step, there are a number of implementation steps as described in the Vivado Design Suite User Guide: Implementation (UG904). You can assign Tcl scripts for the Vivado tool to run before the step (PRE), or after the step (POST). The specific steps you can assign Tcl scripts to include the following: `SYNTH_DESIGN`, `INIT_DESIGN`, `OPT_DESIGN`, `PLACE_DESIGN`, `ROUTE_DESIGN`, `WRITE_BITSTREAM`.

   **TIP:** There are also some optional steps that can be enabled using the `--vivado.prop run.impl_1.steps.phys_opt_design.is_enabled=1` option. When enabled, these steps can also have Tcl PRE and POST scripts.

   An example of the Tcl PRE and POST script assignments follow:

   ```
   --vivado.prop run.impl_1.STEPS.PLACE_DESIGN.TCL.PRE=/.../xxx.tcl
   ```

   In the preceding example a script has been assigned to run before the `PLACE_DESIGN` step. The command line is broken down as follows:

   - `--vivado` is the `v++` command-line option to specify directives for the Vivado tools.
   - `prop` keyword to indicate you are passing a property setting.
   - `run.` keyword to indicate that you are passing a run property.
2. Setting properties on run, file, and fileset design objects.

This is very similar to passing Tcl scripts as described above, but in this case you are passing values to different properties on multiple design objects. For example, to use a specific implementation strategy such as Performance_Explore and disable global buffer insertion during placement, you can define the properties as shown below:

```
[vivado]
prop=run.impl_1.STEPS.OPT_DESIGN.ARGS.DIRECTIVE=Explore
prop=run.impl_1.STEPS.PLACE_DESIGN.ARGS.DIRECTIVE=Explore
prop=run.impl_1.{STEPS.PLACE_DESIGN.ARGS.MORE OPTIONS}={-no_bufg_opt}
prop=run.impl_1.STEPS.PHYS_OPT_DESIGN.IS_ENABLED=true
prop=run.impl_1.STEPS.PHYS_OPT_DESIGN.ARGS.DIRECTIVE=Explore
prop=run.impl_1.STEPS.ROUTE_DESIGN.ARGS.DIRECTIVE=Explore
```

In the example above, the Explore value is assigned to the STEPS.XXX.DIRECTIVE property of various steps of the implementation run. Note the syntax for defining these properties is:

```
<object>.<instance>.property=<value>
```

Where:

- `<object>` can be a design run, a file, or a fileset object.
- `<instance>` indicates a specific instance of the object.
- `<property>` specifies the property to assign.
- `<value>` defines the value of the property.

In this example the object is a run, the instance is the default implementation run, impl_1, and the property is an argument of the different step names, In this case the DIRECTIVE, IS_ENABLED, and {MORE OPTIONS}. Refer to `--vivado Options` for more information on the command syntax.

3. Enabling optional steps in the Vivado implementation process.

The build process runs Vivado synthesis and implementation to generate the device binary. Some of the implementation steps are enabled and run as part of the default build process, and some of the implementation steps can be optionally enabled at your discretion.
Optional steps can be listed using the `--list_steps` command, and include:

- `vpl.impl.power_opt_design`
- `vpl.impl.post_place_power_opt_design`
- `vpl.impl.phys_opt_design`
- `vpl.impl.post_route_phys_opt_design`

An optional step can be enabled using the `--vivado.prop` option. For example, to enable `PHYS_OPT_DESIGN` step, use the following config file content:

```bash
[vivado]
prop=run.impl_1.steps.phys_opt_design.is_enabled=1
```

When an optional step is enabled as shown above, the step can be specified as part of the `-from_step/-to_step` command as described below in Running `--to_step` or `--from_step`, or enable a Tcl script to run before or after the step as described in `--linkhook Options`.

4. Passing parameters to the tool to control processing.

The `--vivado` option also allows you to pass parameters to the Vivado tools. The parameters are used to configure the tool features or behavior prior to launching the tool. The syntax for specifying a parameter uses the following form:

```
--vivado.param <object><parameter>=<value>
```

The keyword param indicates that you are passing a parameter for the Vivado tools, rather than a property for a design object. You must also define the <object> it applies to, the <parameter> that you are specifying, and the <value> to assign it.

The following example project indicates the current Vivado project, `writeIntermediateCheckpoints`, is the parameter being passed and the value is 1, which enables this boolean parameter.

```
--vivado.param project.writeIntermediateCheckpoints=1
```

5. Managing the reports generated during synthesis and implementation.

**IMPORTANT! You must also specify `--save-temps` on the v++ command line when customizing the reports generated by the Vivado tool to preserve the temporary files created during synthesis and implementation, including any generated reports.**

You might also want to generate or save more than the standard reports provided by the Vivado tools when run as part of the Vitis tools build process. You can customize the reports generated using the `--advanced.misc` option as follows:

```bash
[advanced]
misc=report=type report_utilization name synth_report_utilization_summary steps [synth_design] runs [__KERNEL__] options []
misc=report=type report_timing_summary name impl_report_timing_summary_init_design_summary steps [init_design] runs [impl_1] options [-max_paths 10]
misc=report=type report_utilization name impl_report_utilization_init_design_summary steps [init_design] runs [impl_1] options []
misc=report=type report_control_sets name impl_report_control_sets_place_design_summary steps [place_design] runs
```
The syntax of the command line is explained using the following example:

```
misc=report=type report_bus_skew name
impl_report_bus_skew_route_design_summary steps {route_design} runs {impl_1} options {} [-warn_on_violation]
```

- **misc=report**: Specifies the --advanced.misc option as described in --advanced Options, and defines the report configuration for the Vivado tool. The rest of the command line is specified in name/value pairs, reflecting the options of the create_report_config Tcl command as described in Vivado Design Suite Tcl Command Reference Guide (UG835).

- **type report_bus_skew**: Relates to the -report_type argument, and specifies the type of the report as the report_bus_skew. Most of the report_* Tcl commands can be specified as the report type.

- **name impl_report_bus_skew_route_design_summary**: Relates to the -report_name argument, and specifies the name of the report. Note this is not the file name of the report, and generally this option can be skipped as the report names will be auto-generated by the tool.

- **steps {route_design}**: Relates to the -steps option, and specifies the synthesis and implementation steps that the report applies to. The report can be specified for use with multiple steps to have the report regenerated at each step, in which case the name of the report will be automatically defined.

- **runs {impl_1}**: Relates to the -runs option, and specifies the name of the design runs to apply the report to.

- **options {-warn_on_violation}**: Specifies various options of the report_* Tcl command to be used when generating the report. In this example, the -warn_on_violation option is a feature of the report_bus_skew command.

**IMPORTANT!** There is no error checking to ensure the specified options are correct and applicable to the report type specified. If you indicate options that are incorrect the report will return an error when it is run.
Running Multiple Implementation Strategies for Timing Closure

For challenging designs, it can take multiple iterations of Vivado implementation using multiple different strategies to achieve timing closure. This topic shows you how to launch multiple implementation strategies at the same time in the hardware build (--hw), and how to identify and use successful runs to generate the device binary and complete the build.

As explained in --vivado Options the --vivado.impl.strategies command enables you to specify multiple strategies to run in a single build pass. The command line would look as follows:

```
v++ --link -s -g -t hw --platform xilinx_zcu102_base_202010_1 -I . \   
--vivado.impl.strategies "Performance_Explore,Area_Explore" -o \ 
kernel.xclbin hello.xo
```

In the example above, the Performance_Explore and Area_Explore strategies are run simultaneously in the Vivado build to see which returns the best results. You can specify the ALL to have all available strategies run within the tool.

You can also determine this option in a configuration file in the following form:

```
#Vivado Implementation Strategies
[vivado]
impl.strategies=Performance_Explore,Area_Explore
```

The Vitis compiler automatically picks the first completed run results that meets timing to proceed with the build process and generate the device binary. However, you can also direct the tool to wait for all runs to complete and pick the best results from the completed runs before proceeding. This would require the use of the --advanced.compiler directive as shown:

```
[advanced]
param=compiler.multiStrategiesWaitOnAllRuns=1
```

You can also manually review the results of all implementation strategies after they have completed. Then, use the results of any of the implementation runs by using the --reuse_impl option as described in Using -to_step and Launching Vivado Interactively.

Using -to_step and Launching Vivado Interactively

The Vitis compiler lets you stop the build process after completing a specified step (--to_step), manually intervene in the design or files in some way, and then continue the build by specifying a step the build should resume from (--from_step). The --from_step directs the Vitis compiler to resume compilation from the step where --to_step left off, or some earlier step in the process. The --to_step and --from_step are described in Vitis Compiler General Options.
IMPORTANT! The --to_step and --from_step options are sequential build options that require you to use the same project directory when launching v++ --link --from_step as you specified when using v++ --link --to_step.

The Vitis compiler also provides a --list_steps option to list the available steps for the compilation or linking processes of a specific build target. For example, the list of steps for the link process of the hardware build can be found by:

```
v++ --list_steps --target hw --link
```

This command returns a number of steps, both default steps and optional steps that the Vitis compiler goes through during the linking process of the hardware build. Some of the default steps include: system_link, vpl, vpl.create_project, vpl.create_bd, vpl.generate_target, vpl.synth, vpl.impl.opt_design, vpl.impl.place_design, vpl.impl.route_design, and vpl.impl.write_bitstream.

Optional steps include: vpl.impl.power_opt_design, vpl.impl.post_place_power_opt_design, vpl.impl.phys_opt_design, and vpl.impl.post_route_phys_opt_design.

TIP: An optional step must be enabled before specifying it with --from_step or --to_step as previously described in Using the -vivado and -advanced Options.

Launching the Vivado IDE for Interactive Design

For example, with the --to_step command, you can launch the build process to Vivado synthesis and then start the Vivado IDE on the project to manually place and route the design. To perform this you would use the following command syntax:

```
v++ --target hw --link --to_step vpl.synth --save-temps --platform <PLATFORM_NAME> <XO_FILES>
```

TIP: As shown in the example above, you must also specify --save-temps when using --to_step to preserve any temporary files created by the build process.

This command specifies the link process of the hardware build, runs the build through the synthesis step, and saves the temporary files produced by the build process.

You can launch the Vivado tool directly on the project built by the Vitis compiler using the --interactive command. This opens the Vivado project found at <temp_dir>/link/vivado/vpl/prj in your build directory, letting you interactively edit the design:

```
v++ --target hw --link --interactive --save-temps --platform <PLATFORM_NAME> <XO_FILES>
```
When invoking the Vivado IDE in this mode, you can open the synthesis or implementation runs to manage and modify the project. You can change the run details as needed to close timing and try different approaches to implementation. You can save the results to a design checkpoint (DCP), or generate the project bitstream (.bit) to use in the Vitis environment to generate the device binary.

After saving the DCP from within the Vivado IDE, close the tool and return to the Vitis environment. Use the --reuse_impl option to apply a previously implemented DCP file in the v++ command line to generate the xclbin.

**IMPORTANT!** The --reuse_impl option is an incremental build option that requires you to apply the same project directory when resuming the Vitis compiler with --reuse_impl that you specified when using --to_step to start the build.

The following command completes the linking process by using the specified DCP file from the Vivado tool to create the project.xclbin from the input files.

```bash
v++ --link --platform <PLATFORM_NAME> -o'project.xclbin' project.xo --reuse_impl './_x/link/vivado/routed.dcp'
```

You can also use a bitstream file generated by the Vivado tool to create the project.xclbin:

```bash
v++ --link --platform <PLATFORM_NAME> -o'project.xclbin' project.xo --reuse_bit './_x/link/vivado/project.bit'
```

**Additional Vivado Options**

Some additional switches that can be used in the v++ command line or config file include the following:

- `--export_script`/`--custom_script` edit and use Tcl scripts to modify the compilation or linking process.
- `--remote_ip_cache` specify a remote IP cache directory for Vivado synthesis.
- `--no_ip_cache` turn off the IP cache for Vivado synthesis. This causes all IP to be re-synthesized as part of the build process, scrubbing out cached data.

**Controlling Report Generation**

The `v++ -R` option (or `--report_level`) controls the level of information to report during compilation or linking for hardware emulation and system targets. Builds that generate fewer reports will typically run more quickly.
The command line option is as follows:

```
$ v++ -R <report_level>
```

Where `<report_level>` is one of the following options:

- **-R0**: Minimal reports and no intermediate design checkpoints (DCP).
- **-R1**: Includes R0 reports plus:
  - Identifies design characteristics to review for each kernel (`report_failfast`).
  - Identifies design characteristics to review for the full post-optimization design.
  - Saves post-optimization design checkpoint (DCP) file for later examination or use in the Vivado Design Suite.

  **TIP:** `report_failfast` is a utility that highlights potential device usage challenges, clock constraint problems, and potential unreachable target frequency (MHz).

- **-R2**: Includes R1 reports plus:
  - Includes all standard reports from the Vivado tools, including saved DCPs after each implementation step.
  - Design characteristics to review for each SLR after placement.
- **-Restimate**: Forces Vitis HLS to generate a System Estimate report, as described in System Estimate Report.

  **TIP:** This option is useful for the software emulation build (`-t sw_emu`).
Chapter 15

Packaging the System

After compiling and linking your kernel code to build the .xclbin, you need to package the device binary, along with any required supporting files, to build a package that can be run for software or hardware emulation, or can be booted and run on the hardware device. The v++ --package step, or -p, packages the final product at the end of the v++ compile and link process. This is a required step for all Versal™ platforms, including AI Engine platforms, and embedded processor platforms.

As described in --package Options, this command lets you package your design and define various files required for booting and configuring the Xilinx® device for use during emulation or in production systems. It collects the various elements to create an SD card, or other means to program the device, to define the operating system, and to load the application and kernel code.

Packaging for Embedded Platforms

For embedded platforms, the --package command supports a variety of tool flows and platforms, including Versal, AI Engine, and Zynq® devices. The command line is shown below:

```bash
v++ --package -t [sw_emu | hw_emu | hw] --platform <platform> input.xclbin [ -o output.xclbin ]
```

*Note*: If the output option (-o) is not specified, the tool creates an output file with the default name of a.xclbin.

In the case of AI Engine platforms, the package process also takes the libadf.a file produced by the aiecompiler command and integrates it into the output device binary. For more information, refer to the Versal ACAP AI Engine Programming Environment User Guide (UG1076).

The --package command has a range of options for use with the different platforms and build targets supported by the Vitis tools. In the Vitis IDE, the package process is automated and the tool creates the required files as needed. However, in the command line flow, you must specify the v++ --package command or add the [package] tag in the config file with the right options for the job. The following is an example command for hardware emulation that runs the package process for a ZCU104 based application:

```bash
v++ --package -t hw_emu --platform xilinx_zcu104_base_202010_1 --save-temps ./input.xclbin ./output.xclbin --config package.cfg
```
Where, the `--config package.cfg` option specifies a configuration file for the Vitis compiler with the various options specified for the package process. The following is the content of an example configuration file:

```ini
[package]
out_dir=sd_card
boot_mode=sd
image_format=ext4
rootfs=/tmp/platforms/sw/zynqmp/xilinx-zynqmp-common-v2020.1/rootfs.ext4
sd_file=/tmp/platforms/sw/zynqmp/xilinx-zynqmp-common-v2020.1/image
sd_file=host.elf
sd_file=output.xclbin
sd_file=xrt.ini
sd_file=launch_app.sh
```

For software and hardware emulation, the command takes the `.xclbin` file as input, produces a script to launch emulation (`launch_sw_emu.sh` or `launch_hw_emu.sh`), and writes needed support files to a specified output folder, `--package.out_dir`.

Additional files required for running the application, such as data files needed as input or to validate the application, or the `xrt.ini` file for profiling and debug, must be included in the output files, and can be transferred individually using the `sd_file` option, or transferred as a directory using the `sd_dir` option as explained in `--package Options`.

For hardware builds, the `--package` command creates an `sd_card` folder, or the `QSPI.img` depending on the boot mode specified with the `--package.boot_mode` option.

**TIP:** For bare metal ELF files running on PS cores, you should also add the following option to the command line:

```
--package.ps_elf <elf>,<core>
```

The package command creates an output folder called `sd_card`, that contains all of the files needed to run hardware emulation for the application, modeling the boot process of an `sd_card`. For hardware builds, it contains the files required for creating an SD card to boot the device.

After creating the `sd_card` folder, copy the contents to an SD card to create the boot image.

**Note:** On Windows OS you must use a third-party tool, such as Etcher, to write on the SD card for use in booting the Xilinx device.

After the package process completes you can use the Vitis analyzer tool to visualize and navigate the relevant reports or log files by running the following command:

```
vitis_analyzer ./<output>.package_summary
```
Packaging for Data Center Platforms

**TIP:** The `--package` command is not required for Data Center accelerator cards.

The `--package` command essentially copies the device binary (.xclbin) to produce the output .xclbin. The command line is shown below:

```
v++ --package -t [sw_emu | hw_emu | hw] --platform <platform> input.xclbin [-o output.xclbin ]
```

If the output option (`-o`) is not specified, the tool creates an output file with the default name of `a.xclbin`.

After the package process completes, you can use the Vitis analyzer tool to visualize and navigate the relevant reports or log files by running the following command:

```
vitis_analyzer ./<output>.package_summary
```
Chapter 16

Output Directories of the v++ Command

The directory structure generated by the command-line flow has been organized to let you easily find and access files from the project. By navigating the various compile, link, logs, and reports directories, you can easily find generated files. Similarly, each kernel will also have a directory structure created.

When using v++ on the command line, by default it creates a directory structure during compile and link. The XO and XCLBIN files are always generated in the current working directory. All the intermediate files are created under the directory specified by the --temp_dir option, which defaults to _x when --temp_dir is not specified. The link, logs, and reports directories default to inside of the temp_dir, and contain the respective information on the builds.

You can optionally change the directory structure using the following v++ options:

```
--temp_dir <dir_name>
--log_dir <dir_name>
--report_dir <dir_name>
```

The example application uses the following command lines:

```
## Kernel Compilation command:
v++ -t hw_emu --config design.cfg -c -k mmult -I'../src' \
-o'mmult.hw_emu.xilinx_u200_xdma_201830_3.xo' '../src/mmult.cpp'

## Device Binary Linking Command:
v++ -t hw_emu --config design.cfg -l \
-o'mmult.hw_emu.xilinx_u200_xdma_201830_3.xclbin'
mmult.hw_emu.xilinx_u200_xdma_201830_3.xo
```

The design.cfg file includes the following:

```
platform=xilinx_u200_xdma_201830_3
debug=1
save-temps=1
temp_dir=temp_dir

[connectivity]
k=mmult:1:mmult_1
```
The output directory structure follows, where $cwd$ is the current working directory from which the commands are launched:

```bash
### V++ Command Line Directory Structure
$cwd$
  >design.cfg
  >emconfig.json -- emulation platform generated by emconfigutil
  >emulation_debug.log
  >host.exe -- host executable
  >mmult.hw_emu.xilinx_u200_xdma_201830_3.xclbin -- device binary
generated by v++ --link
  >mmult.hw_emu.xilinx_u200_xdma_201830_3.xclbin.info -- device binary
  information file generated by kernelinfo utility
  >mmult.hw_emu.xilinx_u200_xdma_201830_3.xclbin.link_summary -- summary
  report of the v++ link command viewable in Vitis analyzer
  >mmult.hw_emu.xilinx_u200_xdma_201830_3.xclbin.run_summary -- summary
  report of the hardware emulation run viewable in Vitis analyzer
  >mmult.hw_emu.xilinx_u200_xdma_201830_3.xo -- compiled kernel object
  file generated by v++ --compile
  >mmult.hw_emu.xilinx_u200_xdma_201830_3.xo.compile_summary -- summary
  report of the v++ compile command viewable in Vitis analyzer
  >profile_summary.csv -- profile summary produced by XRT during the
  host/kernel run
  >temp_dir -- The build directory specified by the --temp_dir option.
  This defaults to _x when not specified.
  >link -- the output files of the link process
  >activetask.json
  >int
  >address_map.xml
  >appendSection.rtd
  >behav_waveform
  >xsim -- the Vivado simulator contents with results
  from hardware emulation
  >behav.xse
  >cf2sw_full.rtd
  >cf2sw.rtd
  >consolidated.cf
  >dr.bd.tcl
  >kernel_info.dat
  >_kernel_inst_paths.dat
  >kernel_service.json
  >mmult
  >mmult.hw_emu.xilinx_u200_xdma_201830_3_build.rtd
  >mmult.hw_emu.xilinx_u200_xdma_201830_3.gpp_so.log
  >mmult.hw_emu.xilinx_u200_xdma_201830_3.rtd
  >mmult.hw_emu.xilinx_u200_xdma_201830_3.so
  >mmult.hw_emu.xilinx_u200_xdma_201830_3.xml
  >mmult.hw_emu.xilinx_u200_xdma_201830_3_xml.rtd
  >_new_clk_freq
  >sds1.dat
```
Section III: Building and Running the Application

Chapter 16: Output Directories of the v++ Command

>syslinkConfig.ini
>systemDiagramModel.json
>systemDiagramModelSlrBaseAddress.json
>vplConfig.ini
>vplsettings.json
>xclbin_orig.1.xml
>xclbin_orig.xml
>xclbin_orig.xml.tmp
>xo -- the temporary kernel files created by the v++ --
compile command
    >ip_repo
    >mmult
>link.spr
>link.steps.log
>run_link
    >gen_run.xml
    >htr.txt
    >vpl.pb
>sys_link -- files related to the platform used during linking
    >bd
    >cfgraph
    >dr.xml
    >emu
    >iprepo
    >sdsl.dat
    >_sysl
    >xilinx_u200_xdma_201830_3.hpfm
>vivado -- the Vivado Design Suite files for synthesis, implementation, and bitstream generation
    >vivado.spr
    >vpl
    >gen_run.xml
    >htr.txt
    >ipirun.tcl
    >ISEWrap.js
    >ISEWrap.sh
    >openprj.tcl
    >output
        >emu_ooc_copy.xdc
        >insert_debug_profiling.tcl
        >_post_sys_link_gen_constrs.xdc
        >resource.json
    >prj -- the Vivado Design Suite project files
        >prj.cache
        >prj.gen
        >prj.hw
        >prj.ip_user_files
        >prj.sim
        >prj.srcs
        >prj.xpr
    >rundef.js
    >runme.bat
    >runme.log
    >runme.sh
    >scripts
        >_vivado_params.tcl
    >vivado_config_hw_emu.tcl
    >vivado.jou
    >vivado.log
    >vivado.pb
    >vpl.tcl
>logs
Section III: Building and Running the Application

Chapter 16: Output Directories of the v++ Command

>link -- Logs from the linking process
>link.steps.log
>v++.log
>mmult.hw_emu.xilinx_u200_xdma_201830_3 -- Logs from the compilation process
>mmult.hw_emu.xilinx_u200_xdma_201830_3.steps.log
>mmult.vitis.hls.log
>v++.log
>optraceViewer.html
>mmult.hw_emu.xilinx_u200_xdma_201830_3
>mmult
>htr.txt
>ISEWrap.js
>ISEWrap.sh
>mmult -- Vits HLS files used and generated during compilation
>hls.app
>ip
>kernel.xml
>kernel.xml.orig
>mmult.design.xml
>solution
>mmult.tcl
>runme.js
>runme.bat
>runme.log
>runme.sh
>vitis_hls.log
>vitis_hls.pb
>mmult.hw_emu.xilinx_u200_xdma_201830_3.spr
>mmult.hw_emu.xilinx_u200_xdma_201830_3.steps.log

>reports
>link -- Reports generated during the linking process
>system_estimate_mmult.hw_emu.xilinx_u200_xdma_201830_3.xtxt
>v+
+_link_mmult.hw_emu.xilinx_u200_xdma_201830_3.guidance.html
>mmult.hw_emu.xilinx_u200_xdma_201830_3 -- Reports generated during the compilation process
>hls.reports
>system_estimate_mmult.hw_emu.xilinx_u200_xdma_201830_3.xtxt
>v+
+_compile_mmult.hw_emu.xilinx_u200_xdma_201830_3.guidance.html
>vv++_compile_mmult.hw_emu.xilinx_u200_xdma_201830_3_guidance.json
-- Design guidance generated during compilation
>vv++_compile_mmult.hw_emu.xilinx_u200_xdma_201830_3_guidance.pb
>vv++_link_mmult.hw_emu.xilinx_u200_xdma_201830_3_guidance.json -- Design guidance generated during linking
>vv++_link_mmult.hw_emu.xilinx_u200_xdma_201830_3_guidance.pb
>timeline_trace.csv -- Timeline of events generated by XRT during runtime.
>vv++_mmult.hw_emu.xilinx_u200_xdma_201830_3.log -- Log file generated from the v++ commands
>xcd.log
>xclbin.run_summary -- secondary run_summary report
>xilinx_u200_xdma_201830_3-0-
mult.hw_emu.xilinx_u200_xdma_201830_3_simulate.log

>xilinx_u200_xdma_201830_3-0-
mult.hw_emu.xilinx_u200_xdma_201830_3_xsc_report.log

>xrc.log

>xrt.ini
Running Emulation

Development of a user application and hardware kernels targeting an FPGA requires a phased development approach. Because FPGA, Versal™ ACAP, and Zynq UltraScale+ MPSoC are programmable devices, building the device binary for hardware takes some time. To enable quicker iterations without having to go through the full hardware compilation flow, the Vitis™ tool provides emulation targets on which the application and kernels can be run. Compiling for emulation targets is significantly faster than compiling for the actual hardware. Additionally, emulation targets provide full visibility into the application or accelerator, thus making it easier to perform debugging. Once your design passes in emulation, then in the late stages of development you can compile and run the application on the hardware platform.

The Vitis tool provides two emulation targets:

- **Software emulation (sw_emu)**: The software emulation build compiles and links quickly, and the host program runs either natively on an x86 processor or in the QEMU emulation environment. The PL kernels are natively compiled and running on the host machine. This build target lets you quickly iterate on both the host code and kernel logic.

- **Hardware emulation (hw_emu)**: The host program runs in sw_emu, natively on x86 or in the QEMU, but the kernel code is compiled into an RTL behavioral model which is run in the Vivado® simulator or other supported third-party simulators. This build and run loop takes longer but provides a cycle-accurate view of kernel logic.

Compiling for either of the emulation targets is seamlessly integrated into the Vitis command line and IDE flows. You can compile your host and kernel source code for either emulation target, without making any change to the source code. For your host code, you do not need to compile differently for emulation as the same host executable or PS application ELF binary can be used in emulation. Emulation targets support most of the features including XRT APIs, buffer transfer, platform memory SP tags, kernel-to-kernel connections, etc.
Running Emulation Targets

The emulation targets have their own target specific drivers which are loaded by XRT. Thus, the same CPU binary can be run as-is without recompiling, by just changing the target mode during runtime. Based on the value of the `XCL_EMULATION_MODE` environment variable, XRT loads the target specific driver and makes the application interface with an emulation model of the hardware. The allowed values of `XCL_EMULATION_MODE` are `sw_emu` and `hw_emu`. If `XCL_EMULATION_MODE` is not set, then XRT will load the hardware driver.

**IMPORTANT!** It is required to set `XCL_EMULATION_MODE` when running emulation.

![Figure 26: XRT Drivers](X24709-101320)

You can also use the `xrt.ini` file to configure various options applicable to emulation. There is an `[Emulation]` specific section in `xrt.ini`, as described in `xrt.ini File`.

Data Center vs. Embedded Platforms

Emulation is supported for both data center and embedded platforms. For data center platforms, the host application is compiled for x86 server, while the device is modeled as separate x86 process emulating the hardware. The user host code and the device model process communicate using RPC calls. For embedded platforms, where the CPU code is running on the embedded Arm processor, emulation flows use QEMU (Quick Emulator) to mimic the Arm-based PS-subsystem. In QEMU, you can boot embedded Linux and run Arm binaries on the emulation targets.

For running software emulation (`sw_emu`) and hardware emulation (`hw_emu`) of a data center application, you must compile an emulation model of the accelerator card using the `emconfigutil` command and set the `XCL_EMULATION_MODE` environment variable prior to launching your application. The steps are detailed in Running Emulation on Data Center Accelerator Cards.
For running `sw_emu` or `hw_emu` of an embedded application, you must launch the QEMU emulation environment on the x86 processor to model the execution environment of the Arm processor. This requires the use of the `launch_emulator` command, or shell scripts generated during the build process. The details of this flow are explained in Running Emulation on an Embedded Processor Platform.

## QEMU

QEMU stands for Quick Emulator. It is a generic and open source machine emulator. Xilinx provides a customized QEMU model that mimics the Arm based processing system present on Versal ACAP, Zynq® UltraScale+™ MPSoC, and Zynq-7000 SoC devices. The QEMU model provides the ability to execute CPU instructions at almost real time without the need for real hardware. For more information, refer to the Xilinx Quick Emulator User Guide: QEMU.

For hardware emulation, the Vitis emulation targets use QEMU and co-simulate it with an RTL and SystemC-based model for rest of the design to provide a complete execution model of the entire platform. You can boot an embedded Linux kernel on it and run the XRT-based accelerator application. Because QEMU can execute the Arm instructions, you can take the Arm binaries and run them in emulation flows as-is without the need to recompile. QEMU also allows you to debug your application using GDB and TCF-based target connections from Xilinx System Debugger (XSDB).

The Vitis emulation flow also uses QEMU to emulate the MicroBlaze™ processor to model the platform management modules (PLM and PMU) of the devices. On Versal devices, the PLM firmware is used to load the PDI to program sections of the PS and AI Engine model.

To ensure that the QEMU configuration matches the platform, there are additional files that must be provided as part of `sw` directory of Vitis platforms. Two common files, `qemu_args.txt` and `pmc_args.txt`, contain the command line arguments to be used when launching QEMU. When you create a custom platform, these two files are automatically added to your platform with default contents. You can review the files and edit as needed to model your custom platform. Refer to a Xilinx embedded platform for an example.

Because QEMU is a generic model, it uses a Linux device tree style DTB formatted file to enable and configure various hardware modules. A default QEMU hardware DTB file is shipped with the Vitis tools in the `<vitis_installation>/data/emulation/dtbs` folder. However, if your platform requires a different QEMU DTB, you can package it as part of your platform.

*TIP: The QEMU DTB represents the hardware configuration for QEMU, and is different from the DTB used by the Linux kernel.*
Running Emulation on Data Center Accelerator Cards

**TIP:** Set up the command shell or window as described in *Setting Up the Vitis Environment* prior to running the builds.

1. Set the desired runtime settings in the `xrt.ini` file. This step is optional.

   As described in *xrt.ini File*, the file specifies various parameters to control debugging, profiling, and message logging in XRT when running the host application and kernel execution. This enables the runtime to capture debugging and profile data as the application is running. The Emulation group in the `xrt.ini` provides features that affect your emulation run.

   **TIP:** Be sure to use the `v++ -g` option when compiling your kernel code for emulation mode.

2. Create an `emconfig.json` file from the target platform as described in *emconfigutil Utility*. This is required for running hardware or software emulation.

   The emulation configuration file, `emconfig.json`, is generated from the specified platform using the `emconfigutil` command, and provides information used by the XRT library during emulation. The following example creates the `emconfig.json` file for the specified target platform:

   ```
   emconfigutil --platform xilinx_u200_xdma_201830_2
   ```

   In emulation mode, the runtime looks for the `emconfig.json` file in the same directory as the host executable, and reads in the target configuration for the emulation runs.

   **TIP:** It is mandatory to have an up-to-date JSON file for running emulation on your target platform.

3. Set the `XCL_EMULATION_MODE` environment variable to `sw_emu` (software emulation) or `hw_emu` (hardware emulation) as appropriate. This changes the application execution to emulation mode.

   Use the following syntax to set the environment variable for C shell (csh):

   ```
   setenv XCL_EMULATION_MODE sw_emu
   ```

   Bash shell:

   ```
   export XCL_EMULATION_MODE=sw_emu
   ```

   **IMPORTANT!** The emulation targets will not run if the `XCL_EMULATION_MODE` environment variable is not properly set.

4. Run the application.
With the runtime initialization file (xrt.ini), emulation configuration file (emconfig.json), and the XCL_EMULATION_MODE environment set, run the host executable with the desired command line argument.

**IMPORTANT!** The INI and JSON files must be in the same directory as the executable.

For example:

```
./host.exe kernel.xclbin
```

**TIP:** This command line assumes that the host program is written to take the name of the xclbin file as an argument, as most Vitis examples and tutorials do. However, your application may have the name of the xclbin file hard-coded into the host program, or may require a different approach to running the application.

---

### Running Emulation on an Embedded Processor Platform

**RECOMMENDED:** The file size limit on your machine should either be set to unlimited or a higher value (over 16 GB) because embedded HW Emulation can create files with larger file size for memory.

**TIP:** Set up the command shell or window as described in Setting Up the Vitis Environment prior to running the builds.

1. Set the desired runtime settings in the xrt.ini file.

   As described in xrt.ini File, the file specifies various parameters to control debugging, profiling, and message logging in XRT when running the host application and kernel execution. As described in Enabling Profiling in Your Application this enables the runtime to capture debugging and profile data as your application is running.

   The xrt.ini file, as well as any additional files required for running the application, must be included in the output files as explained in Packaging for Embedded Platforms.

   **TIP:** Be sure to use the `v++ -g` option when compiling your kernel code for emulation mode.

2. Launch the QEMU emulation environment by running the launch_sw_emu.sh script or launch_hw_emu.sh script.

   `launch_sw_emu.sh -forward-port 1440 22`
The script is created in the emulation directory during the packaging process, and uses the `launch_emulator` command to setup and launch QEMU. When launching the emulation script you can also specify options for the `launch_emulator` command. Such as the `-forward-port` option to forward the QEMU port to an open port on the local system. This will be needed when trying to copy files from QEMU as discussed in Step 5 below. Refer to `launch_emulator Utility` for details of the command.

Another example would be to specify `launch_hw_emu.sh -enable-debug` to configure additional XTERMs to be opened for QEMU and PL processes to observe live transcripts of command execution to aid in debugging the application. This is not enabled by default, but can be useful when needed for debug.

3. Mount and configure the QEMU shell with the required settings.

The Xilinx embedded base platforms have rootfs on a separate EXT4 partition on the SD card. After booting Linux, this partition needs to be mounted. If you are running emulation manually, you need to run the following commands from the QEMU shell:

```bash
mount /dev/mmcblk0p1 /mnt
cd /mnt
export LD_LIBRARY_PATH=/mnt:/tmp:$LD_LIBRARY_PATH
export XCL_EMULATION_MODE=hw_emu
export XILINX_XRT=/usr
export XILINX_VITIS=/mnt
```

**TIP:** You can set the `XCL_EMULATION_MODE` environment variable to `sw_emu` for software emulation, or `hw_emu` for hardware emulation. This configures the host application to run in emulation mode.

4. Run the application from within the QEMU shell.

With the runtime initialization (xrt.ini), the `XCL_EMULATION_MODE` environment set, run the host executable with the command line as required by the host application. For example:

```bash
./host.elf kernel.xclbin
```

**TIP:** This command line assumes that the host program is written to take the name of the xclbin file as an argument, as most Vitis examples and tutorials do. However, your application can have the name of the xclbin file hard-coded into the host program, or can require a different approach to running the application.

5. After the application run has completed, you might have some files that were produced by the runtime, such as `profile_summary.csv`, `timeline.csv`, and `xclbin.run_summary`. These files can be found in the /mnt folder inside the QEMU environment. However, to view these files you must copy them from the QEMU Linux system back to your local system. The files can be copied using the `scp` command as follows:

```bash
scp -P 1440 root@<host-ip-address>:/mnt/<file> <dest_path>
```

Where:

- **1440** is the QEMU port to connect to.
• `root@<host-ip-address>` is the root login for the PetaLinux running under QEMU on the specified IP address. The default root password is "root".

• `/mnt/<file>` is the path and file name of the file you want to copy from the QEMU environment.

• `<dest_path>` specifies the path and file name to copy the file to on the local system.

For example:

```bash
scp -P 1440 root@172.55.12.26:/mnt/xclbin.run_summary
```

6. When your application has completed emulation and you have copied any needed files, click `Ctrl + a + x` keys to terminate the QEMU shell and return to the Linux shell.

**Note:** If you have trouble terminating the QEMU environment, you can kill the processes it launches to run the environment. The tool reports the process IDs (pids) at the start of the transcript, or you can specify the `-pid-file` option to capture the pids when launching emulation.

---

**Speed and Accuracy of Hardware Emulation**

Hardware emulation uses a mix of SystemC and RTL co-simulation to provide a balance between accuracy and speed of simulation. The SystemC models are a mix of purely functional models and performance approximate models. Hardware emulation does not mimic hardware accuracy 100%, therefore you should expect some differences in behavior between running emulation and executing your application on hardware. This can lead to significant differences in application performance, and sometimes differences in functionality can also be observed.

Functional differences with hardware typically point to a race condition or some unpredictable behavior in your design. So, an issue seen in hardware might not always be reproducible in hardware emulation, though most behavior related to interactions between the host and the accelerator, or the accelerator and the memory are reproducible in hardware emulation. This makes hardware emulation an excellent tool to debug issues with your accelerator prior to running on hardware.

The following table lists models that are used to mimic the hardware platform and their accuracy levels.

<table>
<thead>
<tr>
<th>Hardware Functionality</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Host to Card PCIe® Connection and DMA</td>
<td>For data center platforms, the connection to the x86 host server over PCIe is done as a purely functional model and does not have any performance modeling. Thus, any issues related to PCIe bandwidth cannot be reflected in hardware emulation runs.</td>
</tr>
<tr>
<td>(XDMA, SlaveBridge)</td>
<td></td>
</tr>
<tr>
<td>Hardware Functionality</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>UltraScale™ DDR Memory, SmartConnect</td>
<td>The SystemC models for the DDR memory controller, AXI SmartConnect, and other data path IPs are usually throughput approximate. They typically do not model the exact latency of the hardware IP. The model can be used to gauge a relative performance trend as you modify your application or the accelerator kernel.</td>
</tr>
<tr>
<td>AI Engine</td>
<td>The AI Engine SystemC model is cycle approximate, though it is not intended to be 100% cycle accurate. A common model is used between AI Engine Simulator and hardware emulation, thus enabling a reasonable comparison between the two stages.</td>
</tr>
<tr>
<td>Versal NoC and DDR Models</td>
<td>The Versal NoC and DDR SystemC models are cycle approximate.</td>
</tr>
<tr>
<td>Arm Processing Subsystem (PS, CIPS)</td>
<td>The Arm PS is modeled using QEMU, which is a purely functional execution model. For more information, see QEMU.</td>
</tr>
<tr>
<td>User Kernel (accelerator)</td>
<td>Hardware emulation uses RTL for the user accelerator. As follows, the accelerator behavior by itself is 100% accurate. However, the accelerator is surrounded by other approximate models.</td>
</tr>
<tr>
<td>Other I/O Models</td>
<td>For hardware emulation, there is generic Python or C-based traffic generator which can be interfaced with the emulation process. You can generate abstract traffic at AXI protocol level which mimics the I/O in your design. Because these models are abstract, any issues observed on the specific hardware board will not be shown in hardware emulation.</td>
</tr>
</tbody>
</table>

Because hardware emulation uses RTL co-simulation as its execution model, the speed of execution is orders of magnitude slower as compared to real hardware. Xilinx recommends using small data buffers. For example, if you have a configurable vector addition and in hardware you are performing a 1024 element \texttt{vadd}, in emulation you might restrict it to 16 elements. This will enable you to test your application with the accelerator, while still completing execution in reasonable time.
Working with RTL Simulators in Hardware Emulation

RTL Simulator Support

The Vitis tool uses the Vivado logic simulator (xsim) as the default simulator for all platforms, including Alveo Data Center accelerator cards, and Versal and Zynq UltraScale+ MPSoC embedded platforms. However, for Versal embedded platforms, like xilinx_vck190_base or custom platforms similar to it, the Vitis tool also supports the use of third-party simulators for hardware emulation: Mentor Graphics Questa Advanced Simulator, Xcelium, and VCS. The specific versions of the supported simulators are the same as the versions supported by Vivado Design Suite.

TIP: For data center platforms, hardware emulation supports the U250_XDMA platform with Questa Advanced Simulator. This support does not include features like peer-to-peer (P2P), SlaveBridge, or other features unless explicitly mentioned.

Enabling a third-party simulator requires some additional configuration options to be implemented during generation of the device binary (.xclbin) and supporting Tcl scripts. The specific requirements for each simulator is discussed below.

- **Questa**: Add the following advanced parameters and Vivado properties to a configuration file for use during linking:

  ```
  ## Final set of additional options required for running simulation using Questa Simulator
  [advanced]
  param=hw_emu.simulator=QUESTA
  [vivado]
  prop=project.__CURRENT__.simulator.questa_install_dir=/tools/gensys/questa/2020.2/bin/
  prop=project.__CURRENT__.compxlib.questa_compiled_library_dir=<install_dir>/clibs/questa/2020.2/lin64/lib/
  prop=fileset.sim_1.questa.compile.sccom.cores={4}
  ```

  After generating the configuration file you can use it in the v++ command line as follows:

  ```
  v++ -link --config questa_sim.cfg
  ```
• **Xcelium**: Add the following advanced parameters and Vivado properties to a configuration file for use during linking:

```bash
## Final set of additional options required for running simulation using
Questa Simulator
[advanced]
param=hw_emu.simulator=XCELIUM
[vivado]
prop=project.__CURRENT__.compxlib.xcelium_compiled_library_dir=/proj/
    xbuilds/2020.2_daily_latest/clibs/xcelium/20.03.005/lin64/lib/
prop=fileset.sim_1.xcelium.elaborate.xmelab.more_options={-timescale 1ns/
    1ps}
```

After generating the configuration file you can use it in the `v++` command line as follows:

```bash
v++ -link --config xcelium.cfg
```

• **VCS**: Add the following advanced parameters and Vivado properties to a configuration file for use during linking:

```bash
## Final set of additional options required for running simulation using
Questa Simulator
[advanced]
param=hw_emu.simulator=VCS
[vivado]
prop=project.__CURRENT__.simulator.vcs_install_dir=/tools/gensys/vcs/
    Q-2020.03/bin/
prop=project.__CURRENT__.compxlib.vcs_compiled_library_dir=/proj/xbuilds/
    2020.2_daily_latest/clibs/vcs/Q-2020.03/lin64/lib/
prop=project.__CURRENT__.simulator.vcs_gcc_install_dir=/tools/installs/
    synopsys/vg_gnu/2019.06/amd64/gcc-6.2.0_64/bin
```

After generating the configuration file you can use it in the `v++` command line as follows:

```bash
v++ -link --config vcs_sim.cfg
```

You can use the `-user-pre-sim-script` and `-user-post-sim-script` options from the `launch_emulator` command to specify Tcl scripts to run before the start of simulation, or after simulation completes. As an example, in these scripts, you can use the `$cwd` command to get the run directory of the simulator and copy any files needed prior to simulation, or copy any output files generated at the end of simulation.

To enable hardware emulation, you must setup the environment for simulation in the Vivado Design Suite. A key step for setup is pre-compiling the RTL and SystemC models for use with the simulator. To do this, you must run the `compile_sim_lib` command in the Vivado tool. For more information on pre-compilation of simulation models, refer to the *Vivado Design Suite User Guide: Logic Simulation (UG900).*
When creating your Versal platform ready for simulation, the Vivado tool generates a simulation wrapper which must be instantiated in your simulation test bench. So, if the top most design module is `<top>`, then when calling `launch_simulation` in the Vivado tool, it will generate a `<top>_sim_wrapper` module, and also generates `xlnoc.bd`. These files are generated as simulation-only sources and will be overwritten anytime `launch_simulation` is called in the Vivado tool. Platform developers need to instantiate this module in the test bench and not their own `<top>` module.

**Using the Simulator Waveform Viewer**

Hardware emulation uses RTL and SystemC models for execution. A regular application and HLS-based kernel developer does not need to be aware of the hardware level details. The Vitis analyzer provides sufficient details of the hardware execution model. However, for advanced users who are familiar with HW signal and protocols, they can launch hardware emulation with the simulator waveform running, as described in *Waveform View and Live Waveform Viewer*.

By default, when running `v++ --link -t hw_emu`, the tool compiles the simulation models in optimized mode. However, when you also specify the `-g` switch, you enable hardware emulation models to be compiled in debug mode. During the application runtime, use the `-g` switch with the `launch_hw_emu.sh` command to run the simulator interactively in GUI mode with waveforms displayed. By default, the hardware emulation flow adds common signals of interest to the waveform window. However, you can pause the simulator to add signals of interest and resume simulation.

**AXI Transactions Display in XSIM Waveform**

Many models in hardware emulation use SystemC transaction-level modeling (TLM). In these cases, interactions between the models cannot be viewed as RTL waveforms. However, Vivado simulator (xsim) provides a transaction level viewer. For standard platforms, these interface objects can be added to the waveform view, similar to how RTL signals are added. As an example, to add an AXI interface to the waveform, use the following Tcl command in `xsim`:

```
add_wave <HDL_objects>
```

Using the `add_wave` command, you can specify full or relative paths to HDL objects. For additional details on how to interpret the TLM waveform and how to add interfaces in the GUI, see the *Vivado Design Suite User Guide: Logic Simulation (UG900)*.
Using I/O Traffic Generators

Introduction

Some user applications such as video streaming and Ethernet-based applications make use of I/O ports on the platform to stream data into and out of the platform. For these applications, performing hardware emulation of the design requires a mechanism to mimic the hardware behavior of the I/O port, and to simulate data traffic running through the ports. I/O traffic generators let you model traffic through the I/O ports during hardware emulation in the Vitis application acceleration development flow, or during logic simulation in the Vivado Design Suite.

Adding Traffic Generators to your Design

Xilinx devices have rich I/O interfaces. The Alveo accelerator cards primarily have PCIe and DDR memory interfaces which have their own specific model. However, your platforms could also have other I/Os, for example GT-kernel based generic I/O, Video Streams, and Sensor data. I/O Traffic Generator kernels provide a method for platforms and applications to inject traffic onto the I/O during simulation.

This solution requires both the inclusion of streaming I/O kernels (XO) or IP in your design, and the use of a Python or C++ API provided by Xilinx to inject traffic or to capture output data from the emulation process. The Xilinx provided Python or C++ library can be used to integrate traffic generator code into your application, run it as a separate process, and have it interface with the emulation process. Currently, Xilinx provides a library that enables interfacing at AXI4-Stream level to mimic any Streaming I/O.

The streaming I/O model can be used to emulate streaming traffic on the platform, and also support delay modeling. You can add streaming I/O to your application, or add them to your custom platform design as described below:

• Streaming I/O kernels can be added to the device binary (XCLBIN) file like any other compiled kernel object (XO) file, using the v++ --link command. The Vitis installation provides kernels for AXI4-Stream interfaces of various data widths. These can be found in the software installation at $XILINX_VITIS/data/emulation/XO.

Add these to your designs using the following example command:

```
v++ -t hw_emu --link $XILINX_VITIS/data/emulation/XO/  
sim_ipc_axis_master_32.xo $XILINX_VITIS/data/emulation/XO/  
sim_ipc_axis_slave_32.xo  
```

In the example above, the sim_ipc_axis_master_32.xo and sim_ipc_axis_slave_32.xo provide 32-bit master and slave kernels that can be linked with the target platform and other kernels in your design to create the .xclbin file for the hardware emulation build.
• IPC modules can also be added to a platform block design using the Vivado IP integrator feature for Versal and Zynq UltraScale+ MPSoC custom platforms. The tool provides `sim_ipc_axis_master_v1_0` and `sim_ipc_axis_slave_v1_0` IP to add to your platform design. These can be found in the software installation at `$XILINX_VIVADO/data/emulation/hw_em/ip_repo`.

The following is an example Tcl script used to add IPC IP to your platform design, which will enable you to inject data traffic into your simulation from an external process written in Python or C++:

```tcl
# Update IP Repository path if required
set_property -name ip_repo_paths $XILINX_VIVADO/data/emulation/hw_em/ip_repo [current_project]
## Add AXIS Master
create_bd_cell -type ip -vlnv xilinx.com:ip:sim_ipc_axis_master:1.0 sim_ipc_axis_master_0
# Change Model Property if required
set_property -dict [list CONFIG.C_M00_AXIS_TDATA_WIDTH {64}]
[get_bd_cells sim_ipc_axis_master_0]
## Add AXIS Slave
create_bd_cell -type ip -vlnv xilinx.com:ip:sim_ipc_axis_slave:1.0 sim_ipc_axis_slave_0
# Change Model Property if required
set_property -dict [list CONFIG.C_S00_AXIS_TDATA_WIDTH {64}]
[get_bd_cells sim_ipc_axis_slave_0]
```

Writing Traffic Generators in Python

You must also include a traffic generator process while simulating your application to generate data traffic on the I/O traffic generators, or to capture output data from the emulation process. The Xilinx provided Python or C++ library can be used to create the traffic generator code as described below.

• For Python, set `$PYTHONPATH` on the command terminal:

```bash
setenv PYTHONPATH $XILINX_VIVADO/data/emulation/hw_em/lib/python:
$XILINX_VIVADO/data/python/xtlm_ipc/xtlm_ipc_v1_0
```

• Sample Python code to connect with the `gt_master` instance would look like the following:

```python
from xilinx_xtlm import ipc_axis_master_util
from xilinx_xtlm import xtlm_ipc
import struct
import binascii

# Instantiating AXI Master Utilities
master_util = ipc_axis_master_util('gt_master')

# Create payload
payload = xtlm_ipc.axi_stream_packet()
payload.data = "BINARY_DATA"  # One way of getting "BINARY_DATA" from integer can be like payload.data = bytes(bytearray(struct.pack('i', int_number))) More info @ https://docs.python.org/3/library/struct.html
payload.tlast = True  # AXI Stream Fields
```
#Optional AXI Stream Parameters
payload.tuser = "OPTIONAL_BINARY_DATA"
payload.tkeep = "OPTIONAL_BINARY_DATA"

#Send Transaction
master_util.b_transport(payload)
master_util.disconnect() #Disconnect connection between Python & Emulation

• Sample Python code to connect with the gt_slave instance would look like the following:

Blocking Receive
from xilinx_xtlm import ipc_axis_slave_util
from xilinx_xtlm import xtlm_ipc

#Instantiating AXI Slave Utilities
slave_util = ipc_axis_slave_util("gt_slave")

#Sample payload (Blocking Call)
payload = slave_util.sample_transaction()
slave_util.disconnect() #Disconnect connection between Python & Emulation

Writing Traffic Generators in C++

• For C++ the API is available at $XILINX_VIVADO/data/cpp/xtlm_ipc/xtlm_ipc_v1_0/src/. The C++ API provides both blocking and non-blocking function support. The following snippets show the usage.

  TIP: A sample Makefile is also available to generate the executable.

• Blocking send:

#include "xtlm_ipc.h" //Include file

void send_packets()
{
    // Instantiate IPC socket with name matching in IPI diagram...
    xtlm_ipc::axis_initiator_socket_util<xtlm_ipc::BLOCKING>
    socket_util("gt_master");

    const unsigned int NUM_TRANSACTIONS = 8;
    xtlm_ipc::axi_stream_packet packet;

    std::cout << "Sending " << NUM_TRANSACTIONS << " Packets..."
    //generate_data() is your custom code to generate traffic
    std::vector<char> data = generate_data();
    // Set packet attributes...
    packet.set_data(data.data(), data.size());
    packet.set_data_length(data.size());
    packet.set_tlast(1);
    //Additional AXIS attributes can be set if required
    socket_util.transport(packet); //Blocking transport API to send
    the transaction
}

### Blocking receive:

```c
#include "xtlm_ipc.h"

void receive_packets()
{
    // Instantiate IPC socket with name matching in IPI diagram...
    xtlm_ipc::axis_target_socket_util<xtlm_ipc::BLOCKING>
    socket_util("gt_slave");

    const unsigned int NUM_TRANSACTIONS = 8;
    unsigned int num_received = 0;
    xtlm_ipc::axi_stream_packet packet;

    std::cout << "Receiving " << NUM_TRANSACTIONS << " packets...
    while(num_received < NUM_TRANSACTIONS) {
        socket_util.sample_transaction(packet); //API to sample the transaction
        //Process the packet as per requirement.
        num_received += 1;
    }
}
```

### Non-Blocking send:

```c
#include <algorithm>    // std::generate
#include "xtlm_ipc.h"

xtlm_ipc::axi_stream_packet generate_packet()
{
    xtlm_ipc::axi_stream_packet packet;
    // generate_data() is your custom code to generate traffic
    std::vector<char> data = generate_data();

    // Set packet attributes...
    packet.set_data(data.data(), data.size());
    packet.set_data_length(data.size());
    packet.set_tlast(1);
    //packet.set_tlast(std::rand()%2);
    // Option to set tuser tkeep optional attributes...
    return packet;
}

void send_packets()
{
    // Instantiate IPC socket with name matching in IPI diagram...
    xtlm_ipc::axis_initiator_socket_util<xtlm_ipc::NON_BLOCKING>
    socket_util("gt_master");
    // Instantiate Non Blocking specialization

    const unsigned int NUM_TRANSACTIONS = 8;
    xtlm_ipc::axi_stream_packet packet;

    std::cout << "Sending " << NUM_TRANSACTIONS << " Packets..."
    for(int i = 0; i < NUM_TRANSACTIONS; i++) {
```
packet = generate_packet(); // Or user's test pattern / live data etc.

socket_util.transport(packet);
}

• Non-Blocking receive:

```c
#include <unistd.h>
#include "xtlm_ipc.h"

void receive_packets()
{
    // Instantiate IPC socket with name matching in IPI diagram...
    xtlm_ipc::axis_target_socket_util<xtlm_ipc::NON_BLOCKING>
    socket_util("gt_slave");

    const unsigned int NUM_TRANSACTIONS = 8;
    unsigned int num_received = 0, num_outstanding = 0;
    xtlm_ipc::axi_stream_packet packet;

    std::cout << "Receiving " << NUM_TRANSACTIONS << ' packets...'
    << std::endl;
    while(num_received < NUM_TRANSACTIONS) {
        num_outstanding = socket_util.get_num_transactions();
        num_received += num_outstanding;

        if(num_outstanding != 0) {
            std::cout << 'Outstanding packets = ' << num_outstanding
            << std::endl;
            for(int i = 0; i < num_outstanding; i++) {
                socket_util.sample_transaction(packet);
                print(packet);
            }
        }
        usleep(100000); // As transaction is non-blocking we would like to
give some delay between consecutive samplings
    }
}
```

• The following is an example Makefile for the blocking receive above:

```make
GCC=/usr/bin/g++
IPC_XTLM=$(XILINX_VIVADO)/data/cpp/xtlm_ipc/xtlm_ipc_v1_0/src/
PROTO_PATH=$(XILINX_VIVADO)/data/simmodels/xsim/2020.2/lnx64/6.2.0/ext/protobuf/
BOOST=$(XILINX_VIVADO)/tps/boost_1_64_0/

SRC_FILE=b_receive.cpp
.PHONY: run all
default: all
test:
all : b_receive
b_receive: $(SRC_FILE)
    $(GCC) $(SRC_FILE) $(IPC_XTLM)xtlm_ipc.pb.cc -I$(IPC_XTLM)/ -I$(PROTO_PATH)/include/ -L$(PROTO_PATH) -lprotobuf -o $@ -lpthread -I$(BOOST)/
```
Running Traffic Generators

After generating an external process binary as shown above using the library found in $XILINX_VIVADO/data/cpp/xtlm_ipc/xtlm_ipc_v1_0/src/, you can run the emulation using the following steps:

1. Launch the Vitis hardware emulation or Vivado simulation using the standard process and wait for the simulation to start.
2. From another terminal(s), launch the external process such as Python or C++.
Chapter 18

Running the Application Hardware Build

Running the application hardware build allows you to see your application running on an accelerator card, such as the Alveo™ Data Center accelerator card, or an embedded processor platform targeting a Versal™ ACAP or Zynq® devices. The performance data and results captured here are the actual performance of your accelerated application. Yet the profiling data from this run might still reveal opportunities to optimize your design.

**TIP:** To use the accelerator card, you must have it installed as described in Getting Started with Alveo Data Center Accelerator Cards (UG1301).

1. Edit the `xrt.ini` file as described in [xrt.ini File](#).

   This is optional, but recommended when running on hardware for evaluation purposes. You can configure XRT with the `xrt.ini` file to capture debugging and profile data as the application is running. To capture event trace data when running the hardware, refer to Enabling Profiling in Your Application. To debug the running hardware, refer to Debugging During Hardware Execution.

   **TIP:** Ensure to use the `v++ -g` option when compiling your kernel code for debugging.

2. Unset the `XCL_EMULATION_MODE` environment variable.

   **IMPORTANT!** The hardware build will not run if the `XCL_EMULATION_MODE` environment variable is set to an emulation target.

3. For embedded platforms, boot the SD card.

   **TIP:** This step is only required for platforms using Xilinx embedded devices such as Versal ACAP or Zynq UltraScale+ MPSoC.

   For an embedded processor platform, copy the contents of the `.sd_card` folder produced by the `v++ --package` command to an SD card as the boot device for your system. Boot your system from the SD card.

4. Run your application.
The specific command line to run the application will depend on your host code. A common implementation used in Xilinx tutorials and examples is as follows:

```
./host.exe kernel.xclbin
```

**TIP:** This command line assumes that the host program is written to take the name of the `xclbin` file as an argument, as most Vitis examples and tutorials do. However, your application can have the name of the `xclbin` file hard-coded into the host program, or can require a different approach to running the application.
Profiling, Optimizing, and Debugging the Application

Running the system, either in emulation or on the system hardware, presents a series of potential challenges and opportunities. Running the system for the first time, you can profile the application to identify bottlenecks, or performance issues that offer opportunities to optimize the design, as discussed in the sections below. Of course, running the application can also reveal coding errors, or design errors that need to be debugged to get the system running as expected.

This section contains the following chapters:

- Profiling the Application
- Optimizing the Performance
- Debugging Applications and Kernels
Profiling the Application

The Vitis™ core development kit generates various system and kernel resource performance reports during compilation. These reports help you establish a baseline of performance for your application, identify bottlenecks, and help to identify target functions that can be accelerated in hardware kernels as discussed in Methodology for Architecting a Device Accelerated Application. The Xilinx® Runtime (XRT) collects profiling data during application execution in both emulation and hardware builds. Examples of profiling and event data that can be reported includes:

- Host and device timeline events
- OpenCL™ or XRT native API call sequences
- Kernel execution sequence
- Kernel start and stop signals
- FPGA trace data including AXI transactions
- Power profile data for the accelerator card
- AI Engine profiling and event trace
- User event and range profiling

Profiling reports and data can be used to isolate performance bottlenecks in the application, identify problems in the system, and optimize the design to improve performance. Optimizing an application requires optimizing both the application host code and any hardware accelerated kernels. The host code must be optimized to facilitate data transfers and kernel execution, while the kernel should be optimized for performance and resource usage.

There are four distinct areas to be considered when performing algorithm optimization in Vitis: System resource usage and performance, kernel optimization, host optimization, and data transfer optimization. The following Vitis reports and graphical tools support your efforts to profile and optimize these areas:

- Guidance
- System Estimate Report
- HLS Report
- Profile Summary Report
- Application Timeline
- Waveform View and Live Waveform Viewer
When properly enabled as described in Enabling Profiling in Your Application, reports are automatically generated while running the active build, either from the command line as described in Section III: Building and Running the Application, or from the Vitis integrated design environment (IDE). Separate reports are generated for the different build targets and can be found in the respective report directories. Refer to Output Directories of the v++ Command or Output Directories from the Vitis IDE for more information on locating these reports.

Reports can be viewed in Vitis analyzer, or in some cases from the Vitis IDE. To access these reports from Vitis analyzer, open the run_summary report as explained in Section VI: Using the Vitis Analyzer.

Baselining Functionality and Performance

Methodology for Accelerating Applications with the Vitis Software Platform provides an overview of designing an application beginning with profiling the application to identify functions to accelerate, leading into recommended ways of developing C/C++ accelerators. As discussed in the this guide, it is very important to understand the architecture and performance of your application before you start any optimization effort. This is achieved by establishing a baseline for the application in terms of functions and performance.
**Figure 27: Baselining Functions and Performance Flow**

1. **Run application on processor**
   - Profile application to identify bottlenecks and select functions to be accelerated.
   - Convert host code to use OpenCL APIs
   - Convert target functions to kernels
   - Run Software Emulation Verify Function Correctness
2. **Run Hardware Emulation**
   - Analyze Kernel Compilation Reports, Profile Summary, Timeline Trace, Device HW Transactions
3. **Build and Run application on FPGA acceleration card**
   - Analyze Profile Summary
   - Analyze Timeline Trace
4. **Function/Performance baselined**

**Identify Bottlenecks**

The first step is to identify the bottlenecks of the your application running on your target platform. The most effective way is to run the application with profiling tools like the user profiling features described in Custom Profiling of the Host Application, or valgrind, callgrind, and GNU gprof. The profiling data generated by these tools show the call graph with the number of calls to all functions and their execution time.
Run Software and Hardware Emulation

Run software and hardware emulation on the accelerated application as described in Running Emulation, to verify functional correctness, and to generate profiling data on the host code and the kernels. Use Vitis analyzer to review the kernel compilation reports, profile summary, timeline trace, and device hardware transactions to understand the baseline performance estimate for timing interval, latency, and resource utilization, such as DSP and block RAM.

Build and Run the Application

The last step in baselining is building and running the application on an FPGA acceleration card, like one of the Alveo™ Data Center accelerator cards, as described in Running the Application Hardware Build. Analyze the reports from the system compilation, and the profiling data from application execution to see the actual performance and resource utilization on hardware.

**TIP:** Save all the reports during the baseline process, so that you can refer back to them and compare results during optimization.

Enabling Profiling in Your Application

To enable profiling and capturing event trace data during the execution of your application, you must instrument your application for this task. You must enable additional logic, and consume additional device resources to track the host and kernel execution steps, and capture event data. This process requires optionally modifying your host application to capture custom data, modifying your kernel XO during compilation and the XCLBIN during linking to capture different types of profile data from the device side activity, and configuring the Xilinx runtime (XRT) as described in the xrt.ini File to capture data during the application runtime.

**TIP:** While capturing profile data is a critical part of the profiling and optimization process for building your accelerated application, it does consume additional resources and impacts performance. You should be sure to clean these elements out of your final production build.

There are many different types of profiling for your applications, depending on which elements your system includes, and what type of data you want to capture. The following table shows some of the levels of profiling that can be enabled, and discusses which are complimentary and which are not.

**Table 17: Profiling Host and Kernels**

<table>
<thead>
<tr>
<th>Profile/Trace</th>
<th>Description</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Host ApplicationOpenCL API and some limited device side (kernel) profiling.</td>
<td>Specified by the use of the <code>profile</code> and <code>timeline_trace</code> options in the <code>xrt.ini</code> file.</td>
<td>Generates the <code>profile_summary.csv</code> and <code>timeline_trace.csv</code> files.</td>
</tr>
</tbody>
</table>
### Table 17: Profiling Host and Kernels (cont'd)

<table>
<thead>
<tr>
<th>Profile/Trace</th>
<th>Description</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Host Application XRT Native API</td>
<td>Specified by the use of the <code>xrt_profile</code> option in the <code>xrt.ini</code> file.</td>
<td>Generates trace events for the XRT API.</td>
</tr>
<tr>
<td>Host Application User-Event Profiling</td>
<td>Requires additional code in the host application as described in Custom Profiling of the Host Application.</td>
<td>Generates user range data and user events for the host application.</td>
</tr>
<tr>
<td>Low Overhead Profiling</td>
<td>Specified by the use of the <code>lop_trace</code> option in the <code>xrt.ini</code> file.</td>
<td>Generates the <code>lop_trace.csv</code> file as described in Enabling Low Overhead Profiling. Is disabled by <code>profile=true</code> in the <code>xrt.ini</code> file.</td>
</tr>
<tr>
<td>Device Side Profiling</td>
<td>Enabled by the use of <code>--profile</code> options during <code>v++</code> compilation and linking, as described in <code>--profile Options</code>.</td>
<td>Enables capturing data traffic between the host and kernel, kernel stalls, the execution times of kernels and compute units (CUs), as well as monitoring activity in Versal AI Engines.</td>
</tr>
<tr>
<td>AI Engine Graph and Kernels</td>
<td>Specified by the use of the <code>aie_profile</code> and <code>aie_trace</code> options in the <code>xrt.ini</code> file. These options can be specified together or separately.</td>
<td>Generates the <code>aie_profile_&lt;device&gt;.csv</code> and <code>aie_trace_##_&lt;stream id&gt;.txt</code> reports. Cannot be used with <code>profile=true</code> in the <code>xrt.ini</code> file. Is also disabled by the presence of user event profiling in the host application.</td>
</tr>
<tr>
<td>Power Profile</td>
<td>Specified by the use of the <code>xrt_profile</code> option in the <code>xrt.ini</code> file.</td>
<td>Generates the <code>power_profile_&lt;device&gt;.csv</code> report.</td>
</tr>
<tr>
<td>Vitis AI Profiling</td>
<td>Specified by the use of the <code>vitis_ai_profile</code> option in the <code>xrt.ini</code> file.</td>
<td>Enables counter profiling of DPUs to generate the <code>profile_summary.csv</code> file. Is disabled by <code>profile=true</code> in the <code>xrt.ini</code> file.</td>
</tr>
</tbody>
</table>

The device binary (`xclbin`) file is configured for capturing limited device-side profiling data by default. However, using the `--profile` option during the Vitis compiler linking process instruments the device binary by adding Acceleration Monitors and AXI Performance Monitors to the system. This option has multiple instrumentation options: `--profile.data`, `--profile.stall`, and `--profile.exec`, as described in the `--profile Options`. As an example, add `--profile.data` to the `v++` linking command line:

\`v++ -g -l --profile.data all:all:all ...\`

**TIP:** Be sure to also use the `v++ -g` option when compiling your kernel code for debugging with software or hardware emulation.
After your application is enabled for profiling during the `v++` compile and link process, data gathering during application runtime must also be enabled in XRT by editing the `xrt.ini` file as discussed above. For example, the following `xrt.ini` file will enable OpenCL profiling, power profiling, and event and stall trace capture when the application is run:

```
[Debug]
profile=true
power_profile=true
timeline_trace=true
data_transfer_trace=coarse
stall_trace=all
```

To enable the profiling of Kernel Internals data, you must also add the `debug_mode` tag in the `[Emulation]` section of the `xrt.ini`:

```
[Emulation]
debbug_mode=batch
```

If you are collecting a large amount of trace data, you can increase the amount of available memory for capturing data by specifying the `--trace_memory` option during `v++` linking, and add the `trace_buffer_size` keyword in the `xrt.ini`.

- `--trace_memory`: Indicates what type of memory to use for capturing trace data, as described in Vitis Compiler General Options.

- `trace_buffer_size`: Specifies the amount of memory to use for capturing the trace data during the application runtime.

Finally, you can enable continuous trace capture to continuously offload device trace data while the application is running, so in the event of a application or system crash, some trace data is available to help debug the application. To enable, add the `continuous_trace` keyword in the `xrt.ini` file.

**Custom Profiling of the Host Application**

All XRT related actions from the host application are automatically tracked for profiling, through either the OpenCL API calls, or the XRT API calls. However, you can also profile the host application beyond the XRT related events, capturing event data based on user-specified actions or events. This feature provides two types of custom profiling:

- **User range**: Profiles the specified start/end times across a range of code. This captures the span of time within which an action occurs in the host application.

- **User events**: Marks an event in the timeline. The user event is added to the timeline waveform at whatever point in time it occurs.

**TIP:** You can use these features early in the design process as described in Baselining Functionality and Performance, even prior to separating functions to run in the Xilinx device hardware.
Using custom profiling requires a few changes in your host application source code and build process. You must make use of C or C++ API in your code, as described below, and you must include the `xrt_coreutil` library when linking your host application.

- The C/C++ API are described below, but can also be found at the following URL: https://github.com/Xilinx/XRT/blob/master/src/runtime_src/core/include/experimental/xrt_profile.h
- For both C and C++ you must add the following:
  ```
  #include xrt/core/include/experimental/xrt_profile.h
  ```
- When linking host code, add `-lxrt_coreutil` to the compiler command line.

**Profiling of C++ Code**

For C++ code the provided objects are:

- `user_range`: This object captures the start time and end time of a measured range of activity with the specified ID. The object constructor is:
  ```
  user_range(const std::string& label, const std::string& tooltip);
  ```
- `user_event`: This object marks an event occurring at single point in time, adding the specified label onto the timeline trace. The object constructor is:
  ```
  user_event();
  ```

Use the `user_range` to construct an object and start keeping track of time immediately upon construction. Usage details of the `user_range` objects:

- If a `user_range` is instantiated using the default constructor, no time is marked until the user calls `user_range.start()` with the label and tooltip.
- You can instantiate a `user_range` object passing the label and tooltip strings. This starts monitoring the range immediately.
- You must call `user_range.start()` and `user_range.stop()` to capture ranges of time you are interested in.
- If `user_range.stop()` is not called, then any range being tracked lasts until the `user_range` object is destructed.
- The `user_range` object can be reused any number of times, by calling `user_range.start() / user_range.stop()` pairs in the host code.
- Sequential calls to `user_range.start()` ignore all but the first call.
- Sequential calls to `user_range.stop()` ignore all but the first call.

Usage of the `user_event` objects:

- A `user_event` object must be instantiated using the default constructor.
• Calls to `user_event.mark()` will create a user marker on the timeline trace at that particular time.

• `user_event.mark()` takes an optional `const char*` argument which will appear as a label on the timeline trace.

The debug_profile example of the Vitis_Accel_Examples demonstrates user event profiling in a host application. With your host application properly instrumented, XRT can capture profile data from these user-defined ranges and events, as well as the standard XRT API-based events. You must enable profiling in the `xrt.ini` file as explained previously.

**IMPORTANT! Currently, user events are displayed on a separate timeline than the Application Timeline used for OpenCL profiling. When viewing the Run Summary in Vitis analyzer, you will see two separate timelines.**

### Profiling of C Code

For C code the provided functions are:

• `xrtURStart()`: This function establishes the start time of a measured range of activity with the specified ID. The function signature is:

```c
void xrtURStart(unsigned int id, const char* label, const char* tooltip)
```

• `xrtUREnd()`: This function marks the end time of a measured range with the specified ID. The function signature is:

```c
void xrtUREnd(unsigned int id)
```

• `xrtUEMark()`: This function marks an event occurring at single point in time, adding the specified label onto the timeline trace. The function signature is:

```c
void xrtUEMark(const char* label)
```

Use the `xrtURStart()` and `xrtUREnd()` functions to start keeping track of time immediately, and specify an ID to pair the start/end calls and define the user range. Usage details of the `user_range` functions:

• Start/End ranges of one ID can be nested inside other Start/End ranges of a different ID.

• It is your responsibility to make sure the IDs match for the Start/End range you are profiling.

**IMPORTANT! Multiple calls to `xrtURStart` and `xrtUREnd` with the same ID can cause unexpected behavior.**

• The user range can have a label that is added to the timeline, and a tooltip that is displayed when you place the cursor over the user range.

A call to `xrtUEMark()` will create a user marker on the timeline trace at the point of the event.
xrtUEMark() lets you specify a label for the event. The label will appear on the timeline with the mark.

- You can use NULL for the label to add an unlabeled mark.

The following is example code:

```c
int main(int argc, char* argv[]) {
  xrtURStart(0, "Software execution", "Whole program execution");
...
  //TARGET_DEVICE macro needs to be passed from gcc command line
  if(argc != 2) {
    std::cout << 'Usage: " << argv[0] << '" << xclbin' << std::endl;
    return EXIT_FAILURE;
  }
...
  q.enqueueTask(krnl_vector_add);
  
  // The result of the previous kernel execution will need to be retrieved in
  // order to view the results. This call will transfer the data from
  // FPGA to
  // source_results vector
  q.enqueueMigrateMemObjects({buffer_result}, CL_MIGRATE_MEM_OBJECT_HOST);
  ....
  q.finish();
  xrtUEMark("Starting verification");
}
```

### Enabling Low Overhead Profiling

The Vitis software platform supports low overhead profiling that provides minimal information with little effect on execution time. Using this option during runtime, the timeline trace is still available but with a reduced amount of information. Low overhead profiling captures minimal information on OpenCL events and dumps a CSV file called lop_trace.csv at the end of execution. Low overhead profiling can be run in all three flows (hardware, hardware emulation, and software emulation).

To enable low overhead profiling, there is a new flag in the "Debug" section of the xrt.ini File called lop_trace. By default, lop_trace is FALSE and must be enabled by setting the ini parameter to TRUE.

```ini
xrt.ini file
[Debug]
lop_trace=true
```

**TIP:** The lop_trace parameter can be enabled alongside other profiling parameters, but doing so eliminates any benefit of low overhead profiling by capturing all profiling data as well.
When `lop_trace=true` is enabled, the runtime will generate `lop_trace.csv` which can be viewed in the Run Summary within Vitis analyzer.

```
vitis_analyzer <project>.run_summary
```

To obtain the lowest possible overhead, information collected in normal OpenCL profiling is omitted. Specifically, the following information is expected to not be available in the low overhead profiling trace:

- Device events, such as compute unit executions or kernel memory transfers
- Information about memory reads or writes, such as destination address or size
- Information about kernel enqueues, such as kernel name or NDRRange sizes
- Dependencies between buffer transfers and kernel enqueue

---

**Guidance**

The Vitis core development kit has a comprehensive design guidance tool that provides immediate, actionable guidance to the software developer for issues detected in their designs. These issues might be related to the source code, or due to missed tool optimizations. Also, the rules are generic rules based on an extensive set of reference designs. Therefore, these rules might not be applicable for your specific design. It is up to you to understand the specific guidance rules and take appropriate action based on your specific algorithm and requirements.

Guidance is generated from the Vitis HLS, Vitis profiler, and Vivado Design Suite when invoked by the `v++` compiler. The generated design guidance can have several severity levels; warning messages, informational messages and design rule checks are provided during software emulation, hardware emulation, and system builds. The profile design guidance helps you interpret the profiling results which allows you to focus on improving performance.

Guidance includes message text for reported violations, a brief suggested resolution, and a detailed resolution provided as a web link. You can determine your next course of action based on the suggested resolution. This helps improve productivity by quickly highlighting issues and directing you to additional information in using the Vitis technology.

Design guidance is automatically generated after building or running an application from the command line or Vitis IDE.

You can open the Guidance report as discussed in Section VI: Using the Vitis Analyzer. To access the Guidance report, open the Compile Summary, the Link Summary, or the Run Summary, and open the Guidance report.
Kernel Guidance is generated by the Vitis HLS tool after kernel is built using `v++` compile command. This can be viewed in the Vitis analyzer by opening the Compile Summary report. Kernel guidance as well as Compile Summary files are generated for each kernel compiled. Kernel guidance includes recommendations on using Dataflow; and possible reasons why the expected throughout could not be achieved.

System Guidance is generated after kernel is built using the `v++` link command. This can be viewed in the Vitis analyzer by opening the Link Summary report. System guidance includes all Kernel Guidance checks, and provides comprehensive review before running your application.

Run Guidance is generated when your generated `.xclbin` is run, and is a feature of the XRT. This can be viewed by opening the Run Summary in the Vitis analyzer. Run Guidance includes checks like if Kernel Stall is above 50%, recommendations if PLRAM can be used instead of DDR, etc.

With the Guidance report open, the Guidance view displays the messages along with resolution columns. The resolutions also have extended weblink help available.

The following image shows an example of the Guidance report displayed in the Vitis analyzer. For example, clicking a link in the Name column opens a description of the rule check. Links in the Details column can open source code, select a design object such as a kernel, or navigate to another report.

**Figure 28: Design Guidance Example**

![Design Guidance Example](image)

**TIP:** As described in Setting Guidance Thresholds, you can manually edit the values in the Threshold column of the Run Guidance report to customize the report.

There is one HTML guidance report for each run of the `v++` command, including compile and link. The report files are located in the `--report_dir` under the specific output name. For example:

- `v++_compile_<output>_guidance.html` for `v++` compilation
• v++_link_<output>_guidance.html for v++ linking

You can click the web link in the Resolution column to get additional details about the resolution. The Guidance Messaging web page lists all of the current messages for your review.

Figure 29: Guidance Messaging Web Page

Kernel and Compute Unit objects, as well as profile reported data values, can also be cross-probed to other views like the System Diagram or Profile Report. Refer to Working with Reports for more information.

Opening the Guidance Report

When kernels are compiled and when the FPGA binary is linked, guidance reports are generated automatically by the v++ command. You can view these reports in the Vitis analyzer by opening the <output_filename>.compile_summary or the <output_filename>.link_summary for the application project. The <output_filename> is the output of the v++ command.

As an example, launch the Vitis analyzer and open the report using this command:

vitis_analyzer <output_filename>.link_summary

When the Vitis analyzer opens, it displays the link summary report, as well as the compile summary, and a collection of reports generated during the compile and link processes. Both the compile and link steps generate Guidance reports to view by clicking the Build heading on the left-hand side. Refer to Section VI: Using the Vitis Analyzer for more information.
Interpreting Guidance Data

The Guidance view places each entry in a separate row. Each row might contain the name of the guidance rule, threshold value, actual value, and a brief but specific description of the rule. The last field provides a link to reference material intended to assist in understanding and resolving any of the rule violations.

In the GUI Guidance view, guidance rules are grouped by categories and unique IDs in the Name column and annotated with symbols representing the severity. These are listed individually in the HTML report. In addition, as the HTML report does not show tooltips, a full Name column is included in the HTML report as well.

The following list describes all fields and their purpose as included in the HTML guidance reports.

- **Id**: Each guidance rule is assigned a unique ID. Use this id to uniquely identify a specific message from the guidance report.

- **Name**: The Name column displays a mnemonic name uniquely identifying the guidance rule. These names are designed to assist in memorizing specific guidance rules in the view.

- **Severity**: The Severity column allows the easy identification of the importance of a guidance rule.

- **Full Name**: The Full Name provides a less cryptic name compared to the mnemonic name in the Name column.

- **Categories**: Most messages are grouped within different categories. This allows the GUI to display groups of messages within logical categories under common tree nodes in the Guidance view.

- **Threshold**: The Threshold column displays an expected threshold value, which determines whether or not a rule is met. The threshold values are determined from many applications that follow good design and coding practices.

- **Actual**: The Actual column displays the values actually encountered on the specific design. This value is compared against the expected value to see if the rule is met.

- **Details**: The Details column provides a brief message describing the specifics of the current rule.

- **Resolution**: The Resolution column provides a pointer to common ways the model source code or tool transformations can be modified to meet the current rule. Clicking the link brings up a popup window or the documentation with tips and code snippets that you can apply to the specific issue.
System Estimate Report

The process step with the longest execution time includes building the hardware system and the FPGA binary to run on Xilinx devices. Build time is also affected by the target device and the number of compute units instantiated onto the FPGA fabric. Therefore, it is useful to estimate the performance of an application without needing to build it for the system hardware.

The System Estimate report provides estimates of FPGA resource usage and the estimated frequency at which the hardware accelerated kernels can operate. The report is automatically generated for hardware emulation and system hardware builds. The report contains high-level details of the user kernels, including resource usage and estimated frequency. This report can be used to guide design optimization.

You can also force the generation of the System Estimate report with the following option:

```
vv++ .. --report_level estimate
```

An example report is shown in the figure:

**Figure 30: System Estimate**

<table>
<thead>
<tr>
<th>Resource Usage</th>
<th>Estimated Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>System Estimate</td>
<td>Result for Vitis Analyz...</td>
</tr>
<tr>
<td>Design Name</td>
<td>svit_ref acceleration_000000_2</td>
</tr>
<tr>
<td>Target Device</td>
<td>svit_ref_acceleration_000000_2</td>
</tr>
<tr>
<td>Target Clock</td>
<td>300.000000MHz</td>
</tr>
<tr>
<td>Total Number of Kernels</td>
<td>1</td>
</tr>
</tbody>
</table>

Kernel Summary:

<table>
<thead>
<tr>
<th>Kernel Name</th>
<th>Type</th>
<th>Target</th>
<th>OpenCL Library</th>
<th>Compute Units</th>
</tr>
</thead>
<tbody>
<tr>
<td>main</td>
<td>c</td>
<td>sgpp_kernels_000000_2</td>
<td>main</td>
<td>svit_ref_acceleration_000000_2</td>
</tr>
</tbody>
</table>

Timing Information (MHz):

- Compute Unit Kernel Name: svit_ref_acceleration_000000_2
- Frequency: 300.000000 MHz
- Estimated Frequency: 300.000000 MHz
Opening the System Estimate Report

The System Estimate report can be opened in the Vitis analyzer tool, intended for viewing reports from the Vitis compiler when the application is built, and the XRT library when the application is run. You can launch the Vitis analyzer and open the report using the following command:

```
vitis_analyzer <output_filename>.link_summary
```

The `<output_filename>` is the output of the v++ command. This opens the Link Summary for the application project in the Vitis analyzer tool. Then, select the System Estimate report. Refer to Section VI: Using the Vitis Analyzer for more information.

Interpreting the System Estimate Report

The System Estimate report generated by the v++ command provides information on every binary container in the application, as well as every compute unit in the design. The report is structured as follows:

- Target device information
- Summary of every kernel in the application
- Detailed information on every binary container in the solution

The following example report file represents the information generated for the estimate report:

```
--- Design Name:             mmult.hw_emu.xilinx_u200_xdma_201830_2
Target Device:           xilinx:u200:xdma:201830.2
Target Clock:            300.000000MHz
Total number of kernels: 1
---

Kernel Summary

<table>
<thead>
<tr>
<th>Kernel Name</th>
<th>Type</th>
<th>Target</th>
<th>OpenCL Library</th>
<th>Compute Units</th>
</tr>
</thead>
<tbody>
<tr>
<td>mmult</td>
<td>c</td>
<td>fpga0:OCL_REGION_0</td>
<td>mmult.hw_emu.xilinx_u200_xdma_201830_2</td>
<td>1</td>
</tr>
</tbody>
</table>

---

OpenCL Binary:     mmult.hw_emu.xilinx_u200_xdma_201830_2
Kernels mapped to: clc_region

Timing Information (MHz)

<table>
<thead>
<tr>
<th>Compute Unit</th>
<th>Kernel Name</th>
<th>Module Name</th>
<th>Target Frequency</th>
<th>Estimated Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>mmult_1</td>
<td>mmult</td>
<td>mmult</td>
<td>300.300293</td>
<td>411.015198</td>
</tr>
</tbody>
</table>

Latency Information (clock cycles)

<table>
<thead>
<tr>
<th>Compute Unit</th>
<th>Kernel Name</th>
<th>Module Name</th>
<th>Start Interval</th>
<th>Best Case</th>
<th>Avg Case</th>
<th>Worst Case</th>
</tr>
</thead>
<tbody>
<tr>
<td>mmult_1</td>
<td>mmult</td>
<td>mmult</td>
<td>826 - 829</td>
<td>825</td>
<td>827</td>
<td>828</td>
</tr>
</tbody>
</table>
```
Design and Target Device Summary

All design estimate reports begin with an application summary and information about the target device. The device information is provided in the following section of the report:

Design Name: mmult.hw_emu.xilinx_u200_xdma_201830_2
Target Device: xilinx:u200:xdma:201830.2
Target Clock: 300.000000MHz
Total number of kernels: 1

For the design summary, the information provided includes the following:

- **Target Device**: Name of the Xilinx device on the target platform that runs the FPGA binary built by the Vitis compiler.

- **Target Clock**: Specifies the target operating frequency for the compute units (CUs) mapped to the FPGA fabric.

Kernel Summary

This section lists all of the kernels defined for the application project. The following example shows the kernel summary:

<table>
<thead>
<tr>
<th>Kernel Name</th>
<th>Type</th>
<th>Target</th>
<th>OpenCL Library</th>
<th>Compute Units</th>
</tr>
</thead>
<tbody>
<tr>
<td>mmult</td>
<td>c</td>
<td>fpga0:OCL_REGION_0</td>
<td>mmult.hw_emu.xilinx_u200_xdma_201830_2</td>
<td>1</td>
</tr>
</tbody>
</table>

In addition to the kernel name, the summary also provides the execution target and type of the input source. Because there is a difference in compilation and optimization methodology for OpenCL™, C, and C/C++ source files, the type of kernel source file is specified.

The Kernel Summary section is the last summary information in the report. From here, detailed information on each compute unit binary container is presented.
Timing Information

For each binary container, the detail section begins with the execution target of all compute units (CUs). It also provides timing information for every CU. As a general rule, if the estimated frequency for the FPGA binary is higher than the target frequency, the CU will be able to run in the device. If the estimated frequency is below the target frequency, the kernel code for the CU needs to be further optimized to run correctly on the FPGA fabric. This information is shown in the following example:

OpenCL Binary: mmult.hw_emu.xilinx_u200_xdma_201830_2
Kernels mapped to: clc_region

<table>
<thead>
<tr>
<th>Compute Unit</th>
<th>Kernel Name</th>
<th>Module Name</th>
<th>Target Frequency</th>
<th>Estimated Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>mmult_1</td>
<td>mmult</td>
<td>mmult</td>
<td>300.300293</td>
<td>411.015198</td>
</tr>
</tbody>
</table>

It is important to understand the difference between the target and estimated frequencies. CUs are not placed in isolation into the FPGA fabric. CUs are placed as part of a valid FPGA design that can include other components defined by the device developer to support a class of applications.

Because the CU custom logic is generated one kernel at a time, an estimated frequency that is higher than the target frequency indicates that the CU can run at the higher estimated frequency. Therefore, CU should meet timing at the target frequency during implementation of the FPGA binary.

Latency Information

The latency information presents the execution profile of each CU in the binary container. When analyzing this data, it is important to recognize that all values are measured from the CU boundary through the custom logic. In-system latencies associated with data transfers to global memory are not reported as part of these values. Also, the latency numbers reported are only for CUs targeted at the FPGA fabric. The following is an example of the latency report:

<table>
<thead>
<tr>
<th>Compute Unit</th>
<th>Kernel Name</th>
<th>Module Name</th>
<th>Start Interval</th>
<th>Best Case</th>
<th>Avg Case</th>
<th>Worst Case</th>
</tr>
</thead>
<tbody>
<tr>
<td>mmult_1</td>
<td>mmult</td>
<td>mmult</td>
<td>826 - 829</td>
<td>825</td>
<td>827</td>
<td>828</td>
</tr>
</tbody>
</table>

The latency report is divided into the following fields:

- Start interval
- Best case latency
- Average case latency
- Worst case latency

The start interval defines the amount of time that has to pass between invocations of a CU for a given kernel.
The best, average, and worst case latency numbers refer to how much time it takes the CU to generate the results of one ND Range data tile for the kernel. For cases where the kernel does not have data dependent computation loops, the latency values will be the same. Data dependent execution of loops introduces data specific latency variation that is captured by the latency report.

The interval or latency numbers will be reported as "undef" for kernels with one or more conditions listed below:

- OpenCL kernels that do not have explicit `reqd_work_group_size(x,y,z)
- Kernels that have loops with variable bounds

*Note:* The latency information reflects estimates based on the analysis of the loop transformations and exploited parallelism of the model. These advanced transformations such as pipelining and data flow can heavily change the actual throughput numbers. Therefore, latency can only be used as relative guides between different runs.

**Area Information**

Although the FPGA can be thought of as a blank computational canvas, there are a limited number of fundamental building blocks available in each FPGA. These fundamental blocks (FF, LUT, DSP, block RAM) are used by the Vitis compiler to generate the custom logic for each CU in the design. The quantity of fundamental resources needed to implement the custom logic for a single CU determines how many CUs can be simultaneously loaded into the FPGA fabric. The following example shows the area information reported for a single CU:

<table>
<thead>
<tr>
<th>Area Information</th>
<th>Compute Unit</th>
<th>Kernel Name</th>
<th>Module Name</th>
<th>FF</th>
<th>LUT</th>
<th>DSP</th>
<th>BRAM</th>
<th>URAM</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>mmult_1</td>
<td>mmult</td>
<td>mmult</td>
<td>81378</td>
<td>35257</td>
<td>1036</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>

**HLS Report**

The HLS report provides details about the high-level synthesis (HLS) process of a user kernel and is generated during the compilation process for hardware emulation and system builds. This process translates the C/C++ and OpenCL kernel into the hardware description language used for implementing the kernel logic on the FPGA. The report provides estimated FPGA resource usage, operating frequency, latency, and interface signals of the custom-generated hardware logic. These details provide many insights to guide kernel optimization.

When running from the Vitis IDE, this report can be found in the following directory: 

```
_x/<kernel_name>.<target>.<platform>/<kernel_name>/<kernel_name>/
solution/syn/report
```
The HLS report can be opened from the Vitis analyzer by opening the Compile Summary, or the Link Summary as described in Section VI: Using the Vitis Analyzer. An example of the HLS report is shown.

**Figure 31: HLS Report**

### Generating and Opening the HLS Report

**IMPORTANT!** You must specify the `--save-temps` option during the build process to preserve the intermediate files produced by Vitis HLS, including the reports. The HLS report and HLS guidance are only generated for hardware emulation and system builds for C and OpenCL kernels. They are not generated for software emulation or RTL kernels.

The HLS report can be viewed through the Vitis analyzer by opening the `<output_filename>.compile_summary` or the `<output_filename>.link_summary` for the application project. The `<output_filename>` is the output of the `v++` command.

You can launch the Vitis analyzer and open the report using the following command:

```
vitis_analyzer <output_filename>.compile_summary
```

When the Vitis analyzer opens, it displays the Compile Summary and a collection of reports generated during the compile process. Refer to Section VI: Using the Vitis Analyzer for more information.

### Interpreting the HLS Report

The HLS Synthesis report is a spreadsheet listing the module hierarchy in the left column. Each module and loop generated by the HLS run is represented in this hierarchy. The HLS Synthesis report contains the following columns:

- Violation Type
- Latency in clock cycles
- Latency in absolute time (µs)
- Iteration latency
- Iteration Interval
- Loop Tripcount
- Pipelined
- Utilization Estimates of BRAM, DSP, FF, and LUT
- Negative Slack

If this information is part of a hierarchical block, it will sum up the information of the blocks contained in the hierarchy. Therefore, the hierarchy can also be navigated from within the report when it is clear which instance contributes to the overall design.

**CAUTION!** The absolute counts of cycles and latency numbers are based on estimates identified during HLS synthesis, especially with advanced transformations, such as pipelining and dataflow. Therefore, these numbers might not accurately reflect the final results. If you encounter question marks in the report, this might be due to variable bound loops, and you are encouraged to set trip counts for such loops to have some relative estimates presented in this report.

---

**Profile Summary Report**

When properly configured, the Xilinx Runtime (XRT) collects profiling data on host applications and kernels. XRT automatically captures profiling data for the host application as it makes calls to the runtime either through OpenCL or XRT API calls. You can add user calls to your host application to capture additional profiling information, as explained in Custom Profiling of the Host Application. To capture details of the kernel operations you must instrument kernels using the `--profile` options as explained in the next section.

After the application finishes execution, the Profile Summary report is saved as a `.csv` file in the directory where the compiled host code is executed. The Profile Summary provides annotated details regarding the overall application performance. All data generated during the execution of the application is grouped into categories. The Profile Summary lets you examine the kernel execution and data transfer statistics.

**TIP:** The Profile Summary report can be generated for all build configurations. However, with the software emulation build, the report will not include any data transfer details under kernel execution efficiency and data transfer efficiency. This information is only generated in hardware emulation or system builds.

An example of the Profile Summary report is shown below.
Generating and Opening the Profile Summary Report

Capturing the data required for the Profile Summary requires a few steps prior to actually running the application.

1. The FPGA binary (xclbin) file is configured for capturing profiling data by default. However, using the `v++ --profile` option during the linking process enables a greater level of detail in the profiling data captured. For more information, see the --profile Options.

2. The runtime requires the presence of an xrt.ini file, as described in xrt.ini File, that includes the keyword for capturing profiling data:

   ```
   [Debug]
   profile = true
   ```

3. To enable the profiling of Kernel Internals data, you must also add the `debug_mode` tag in the [Emulation] section of the xrt.ini:

   ```
   [Emulation]
   debug_mode = batch
   ```

With profiling enabled in the device binary and in the xrt.ini file, the runtime creates the `profile_summary.csv` report file when running the application, and also creates the `profile_kernels.csv` and `timeline_kernels.csv` files when Kernel Internals is enabled. These files are linked to the Profile Summary report which can be viewed in the Vitis analyzer tool through the Run Summary. Open the Run Summary using the following command:

```
vitis_analyzer <project>.run_summary
```

Related Information

Running the Application Hardware Build  
Section VI: Using the Vitis Analyzer
Interpreting the Profile Summary

The profile summary includes a number of useful statistics for your host application and kernels. The report provides a general idea of the functional bottlenecks in your application. The following tables show the profile summary descriptions.

Settings

This displays the report and XRT configuration settings.

Summary

This displays summary statistics including device execution time and device power.

Kernels & Compute Units

The following table displays the profile summary data for all kernel functions scheduled and executed.

Table 18: Kernel Execution

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kernel</td>
<td>Name of kernel</td>
</tr>
<tr>
<td>Enqueues</td>
<td>Number of times kernel is enqueued. When the kernel is enqueued only once, the following stats will all be the same.</td>
</tr>
<tr>
<td>Total Time</td>
<td>Sum of runtimes of all enqueues (measured from START to END in OpenCL execution model) (in ms)</td>
</tr>
<tr>
<td>Minimum Time</td>
<td>Minimum runtime of all enqueues</td>
</tr>
<tr>
<td>Average Time</td>
<td>Average kernel runtime (in ms)</td>
</tr>
<tr>
<td></td>
<td>(Total time) / (Number of enqueues)</td>
</tr>
<tr>
<td>Maximum Time</td>
<td>Maximum runtime of all enqueues (in ms)</td>
</tr>
</tbody>
</table>

The following table displays the profile summary data for top kernel functions.

Table 19: Top Kernel Execution

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kernel</td>
<td>Name of kernel</td>
</tr>
<tr>
<td>Kernel Instance Address</td>
<td>Host address of kernel instance (in hex)</td>
</tr>
<tr>
<td>Context ID</td>
<td>Context ID on host</td>
</tr>
<tr>
<td>Command Queue ID</td>
<td>Command queue ID on host</td>
</tr>
<tr>
<td>Device</td>
<td>Name of device where kernel was executed (format: &lt;device&gt;-&lt;ID&gt;)</td>
</tr>
<tr>
<td>Start Time</td>
<td>Start time of execution (in ms)</td>
</tr>
<tr>
<td>Duration</td>
<td>Duration of execution (in ms)</td>
</tr>
</tbody>
</table>
This following table displays the profile summary data for all compute units on the device.

**Table 20: Compute Unit Utilization**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compute Unit</td>
<td>Name of compute unit</td>
</tr>
<tr>
<td>Kernel</td>
<td>Kernel this compute unit is associated with</td>
</tr>
<tr>
<td>Device</td>
<td>Name of the device (format: <code>&lt;device&gt;_&lt;ID&gt;</code>)</td>
</tr>
<tr>
<td>Calls</td>
<td>Number of times the compute unit is called</td>
</tr>
<tr>
<td>Dataflow Execution</td>
<td>Specifies whether the CU is executed with dataflow</td>
</tr>
<tr>
<td>Max Parallel Executions</td>
<td>Number of executions in the dataflow region</td>
</tr>
<tr>
<td>Dataflow Acceleration</td>
<td>Shows the performance improvement due to dataflow execution</td>
</tr>
<tr>
<td>CU Utilization (%)</td>
<td>Shows the percent of the total kernel runtime that is consumed by the CU</td>
</tr>
<tr>
<td>Total Time</td>
<td>Sum of the runtimes of all calls (in ms)</td>
</tr>
<tr>
<td>Minimum Time</td>
<td>Minimum runtime of all calls (in ms)</td>
</tr>
<tr>
<td>Minimum runtime of all calls</td>
<td>(Total time) / (Number of work groups)</td>
</tr>
<tr>
<td>Maximum Time</td>
<td>Maximum runtime of all calls (in ms)</td>
</tr>
<tr>
<td>Clock Frequency</td>
<td>Clock frequency used for a given accelerator (in MHz)</td>
</tr>
</tbody>
</table>

This following table displays the profile summary data for running times and stalls for compute units on the device.

**Table 21: Compute Unit Running Times & Stalls**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compute Unit</td>
<td>Name of compute unit</td>
</tr>
<tr>
<td>Running Time</td>
<td>Total time compute unit was running (in µs)</td>
</tr>
<tr>
<td>Intra-Kernel Dataflow Stalls (%)</td>
<td>Percent time the compute unit was stalling from intra-kernel streams</td>
</tr>
<tr>
<td>External Memory Stalls (%)</td>
<td>Percent time the compute unit was stalling from external memory accesses</td>
</tr>
<tr>
<td>Inter-Kernel Pipe Stalls (%)</td>
<td>Percent time the compute unit was stalling from inter-kernel pipe accesses</td>
</tr>
</tbody>
</table>

**Kernel Data Transfers**

This following table displays the data transfer for kernels to the global memory.

**Table 22: Data Transfer**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compute Unit Port</td>
<td>Name of compute unit/port</td>
</tr>
<tr>
<td>Kernel Arguments</td>
<td>List of kernel arguments attached to this port</td>
</tr>
<tr>
<td>Device</td>
<td>Name of device (format: <code>&lt;device&gt;_&lt;ID&gt;</code>)</td>
</tr>
<tr>
<td>Memory Resources</td>
<td>Memory resource accessed by this port</td>
</tr>
<tr>
<td>Transfer Type</td>
<td>Type of kernel data transfers</td>
</tr>
</tbody>
</table>
Table 22: Data Transfer (cont’d)

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Transfers</td>
<td>Number of kernel data transfers (in AXI transactions)</td>
</tr>
<tr>
<td>Note: This might contain printf transfers.</td>
<td></td>
</tr>
<tr>
<td>Transfer Rate</td>
<td>Rate of kernel data transfers (in MB/s):</td>
</tr>
<tr>
<td>Transfer Rate = (Total Bytes) / (Total CU Execution Time)</td>
<td>Where total CU execution time is the total time the CU was active</td>
</tr>
<tr>
<td>Avg Bandwidth Utilization (%)</td>
<td>Average bandwidth of kernel data transfers:</td>
</tr>
<tr>
<td>Bandwidth Utilization (%) = (100 * Transfer Rate) / (0.6 * Max. Theoretical Rate)</td>
<td></td>
</tr>
<tr>
<td>Avg Size</td>
<td>Average size of kernel data transfers (in KB):</td>
</tr>
<tr>
<td>Avg Size = (Total KB) / (Number of Transfers)</td>
<td></td>
</tr>
<tr>
<td>Avg Latency</td>
<td>Average latency of kernel data transfers (in ns)</td>
</tr>
</tbody>
</table>

This following table displays the top data transfer for kernels to the global memory.

Table 23: Top Data Transfer

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compute Unit</td>
<td>Name of compute unit</td>
</tr>
<tr>
<td>Device</td>
<td>Name of device</td>
</tr>
<tr>
<td>Number of Transfers</td>
<td>Number of write and read data transfers</td>
</tr>
<tr>
<td>Avg Bytes per Transfer</td>
<td>Average bytes of kernel data transfers:</td>
</tr>
<tr>
<td>Avg Bytes = (Total Bytes) / (Number of Transfers)</td>
<td></td>
</tr>
<tr>
<td>Transfer Efficiency (%)</td>
<td>Efficiency of kernel data transfers:</td>
</tr>
<tr>
<td>Efficiency = (Average Bytes) / min((Memory Byte Width * 256), 4096)</td>
<td></td>
</tr>
<tr>
<td>Total Data Transfer</td>
<td>Total data transferred by kernels (in MB):</td>
</tr>
<tr>
<td>Total Data = (Total Write) + (Total Read)</td>
<td></td>
</tr>
<tr>
<td>Total Write</td>
<td>Total data written by kernels (in MB)</td>
</tr>
<tr>
<td>Total Read</td>
<td>Total data read by kernels (in MB)</td>
</tr>
<tr>
<td>Total Transfer Rate</td>
<td>Average total data transfer rate (in MB/s):</td>
</tr>
<tr>
<td>Total Transfer Rate = (Total Data Transfer) / (Total CU Execution Time)</td>
<td></td>
</tr>
<tr>
<td>Where total CU execution time is the total time the CU was active</td>
<td></td>
</tr>
</tbody>
</table>

This following table displays the data transfer streams.

Note: This table is only shown if there is stream data

Table 24: Data Transfer Streams

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Master Port</td>
<td>Name of master compute unit and port</td>
</tr>
<tr>
<td>Master Kernel Arguments</td>
<td>List of kernel arguments attached to this port</td>
</tr>
<tr>
<td>Slave Port</td>
<td>Name of slave compute unit and port</td>
</tr>
<tr>
<td>Slave Kernel Arguments</td>
<td>List of kernel arguments attached to this port</td>
</tr>
</tbody>
</table>
Table 24: Data Transfer Streams (cont’d)

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Device</td>
<td>Name of device (format: &lt;device&gt;-&lt;ID&gt;)</td>
</tr>
<tr>
<td>Number of Transfers</td>
<td>Number of stream data packets</td>
</tr>
<tr>
<td>Transfer Rate</td>
<td>Rate of stream data transfers (in MB/s):</td>
</tr>
<tr>
<td></td>
<td>Transfer Rate = (Total Bytes) / (Total CU Execution Time)</td>
</tr>
<tr>
<td></td>
<td>Where total CU execution time is the total time the CU was active</td>
</tr>
<tr>
<td>Avg Size</td>
<td>Average size of kernel data transfers (in KB):</td>
</tr>
<tr>
<td></td>
<td>Average Size = (Total KB) / (Number of Transfers)</td>
</tr>
<tr>
<td>Link Utilization (%)</td>
<td>Link utilization (%):</td>
</tr>
<tr>
<td></td>
<td>Link Utilization = 100 * (Link Busy Cycles - Link Stall Cycles - Link Starve Cycles) / (Link Busy Cycles)</td>
</tr>
<tr>
<td>Link Starve (%)</td>
<td>Link starve (%):</td>
</tr>
<tr>
<td></td>
<td>Link Starve = 100 * (Link Starve Cycles) / (Link Busy Cycles)</td>
</tr>
<tr>
<td>Link Stall (%)</td>
<td>Link stall (%):</td>
</tr>
<tr>
<td></td>
<td>Link Stall = 100 * (Link Stall Cycles) / (Link Busy Cycles)</td>
</tr>
</tbody>
</table>

Host Data Transfers

This following table displays profile data for all write transfers between the host and device memory through PCI Express® link.

Table 25: Top Memory Writes

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buffer Address</td>
<td>Specifies the address location for the buffer</td>
</tr>
<tr>
<td>Context ID</td>
<td>OpenCL Context ID on host</td>
</tr>
<tr>
<td>Command Queue ID</td>
<td>OpenCL Command queue ID on host</td>
</tr>
<tr>
<td>Start Time</td>
<td>Start time of write operation (in ms)</td>
</tr>
<tr>
<td>Duration</td>
<td>Duration of write operation (in ms)</td>
</tr>
<tr>
<td>Buffer Size</td>
<td>Amount of data being transferred (in KB)</td>
</tr>
<tr>
<td>Writing Rate</td>
<td>Data transfer rate (in MB/s):</td>
</tr>
<tr>
<td></td>
<td>(Buffer Size)/(Duration)</td>
</tr>
</tbody>
</table>

This following table displays profile data for all read transfers between the host and device memory through PCI Express® link.

Table 26: Top Memory Reads

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buffer Address</td>
<td>Specifies the address location for the buffer</td>
</tr>
<tr>
<td>Context ID</td>
<td>Context ID on host</td>
</tr>
<tr>
<td>Command Queue ID</td>
<td>Command queue ID on host</td>
</tr>
<tr>
<td>Start Time</td>
<td>Start time of read operation (in ms)</td>
</tr>
</tbody>
</table>
Table 26: Top Memory Reads (cont’d)

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Duration</td>
<td>Duration of read operation (in ms)</td>
</tr>
<tr>
<td>Buffer Size</td>
<td>Amount of data being transferred (in KB)</td>
</tr>
<tr>
<td>Reading Rate</td>
<td>Data transfer rate (in MB/s): (Buffer Size) / (Duration)</td>
</tr>
</tbody>
</table>

This following table displays the data transfer for host to the global memory.

Table 27: Data Transfer

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Context: Number of Devices</td>
<td>Context ID and number of devices in context</td>
</tr>
<tr>
<td>Transfer Type</td>
<td>Type of kernel host transfers</td>
</tr>
<tr>
<td>Number of Buffer Transfers</td>
<td>Number of host buffer transfers</td>
</tr>
<tr>
<td></td>
<td>Note: This might contain printf transfers.</td>
</tr>
<tr>
<td>Transfer Rate</td>
<td>Rate of host buffer transfers (in MB/s): Transfer Rate = (Total Bytes) / (Total Time in µs)</td>
</tr>
<tr>
<td>Avg Bandwidth Utilization (%)</td>
<td>Average bandwidth of host buffer transfers: Bandwidth Utilization (%) = (100 * Transfer Rate) / (Max. Theoretical Rate)</td>
</tr>
<tr>
<td>Avg Size</td>
<td>Average size of host buffer transfers (in KB): Average Size = (Total KB) / (Number of Transfers)</td>
</tr>
<tr>
<td>Total Time</td>
<td>Sum of host buffer transfer durations (in ms)</td>
</tr>
<tr>
<td>Avg Time</td>
<td>Average of host buffer transfer durations (in ms)</td>
</tr>
</tbody>
</table>

API Calls

This following table displays the profile data for all OpenCL host API function calls executed in the host application. The top displays a bar graph of the API call time as a percent of total time.

Table 28: API Calls

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>API Name</td>
<td>Name of the API function (for example, clCreateProgramWithBinary, clEnqueueNDRangeKernel)</td>
</tr>
<tr>
<td>Calls</td>
<td>Number of calls to this API made by the host application</td>
</tr>
<tr>
<td>Total Time</td>
<td>Sum of runtimes of all calls (in ms)</td>
</tr>
<tr>
<td>Minimum Time</td>
<td>Minimum runtime of all calls (in ms)</td>
</tr>
<tr>
<td>Average Time</td>
<td>Average Time (in ms) (Total time) / (Number of calls)</td>
</tr>
<tr>
<td>Maximum Time</td>
<td>Maximum runtime of all calls (in ms)</td>
</tr>
</tbody>
</table>
Device Power

This following table displays the profile data for device power.

**Table 29: Device Power**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Power Used By Platform</td>
<td>Shows a line graph of the three power rails on a Data Center acceleration card:</td>
</tr>
<tr>
<td></td>
<td>• 12V Auxiliary</td>
</tr>
<tr>
<td></td>
<td>• 12V PCIe</td>
</tr>
<tr>
<td></td>
<td>• Internal power</td>
</tr>
<tr>
<td></td>
<td>These show the power (W) usage of the card over time.</td>
</tr>
<tr>
<td>Temperature</td>
<td>One chart will be created for each device that has non-zero temperature readings. Displays</td>
</tr>
<tr>
<td></td>
<td>one line for each temperature sensor with readouts in (°C).</td>
</tr>
<tr>
<td>Fan Speed</td>
<td>One chart will be created for each device that has non-zero fan speed readings. The fan</td>
</tr>
<tr>
<td></td>
<td>speed is measure in RPM.</td>
</tr>
</tbody>
</table>

Kernel Internals

This following table displays the running time for compute units in microseconds (µs) and reports stall time as a percent of the running time.

**TIP:** The Kernel Internals tab reports time in µs, while the rest of the Profile Summary reports time in milliseconds (ms).

**Table 30: CU Runtime and Stalls**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compute Unit</td>
<td>Indicates the compute unit instance name</td>
</tr>
<tr>
<td>Running Time</td>
<td>Reports the total running time for the CU (in µs)</td>
</tr>
<tr>
<td>Intra-Kernel Dataflow Stalls (%)</td>
<td>Reports the percentage of running time consumed in stalls when streaming data</td>
</tr>
<tr>
<td></td>
<td>between kernels</td>
</tr>
<tr>
<td>External Memory Stalls (%)</td>
<td>Reports the percentage of running time consumed in stalls for memory transfers</td>
</tr>
<tr>
<td></td>
<td>outside the CU</td>
</tr>
<tr>
<td>Inter-Kernel Pipe Stalls (%)</td>
<td>Reports the percentage of running time consumed in stalls when streaming data</td>
</tr>
<tr>
<td></td>
<td>to or from outside the CU</td>
</tr>
</tbody>
</table>

This following table displays the data transfer for specific ports on the compute unit.

**Table 31: CU Port Data Transfers**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Port</td>
<td>Indicates the port name on the compute unit</td>
</tr>
<tr>
<td>Compute Unit</td>
<td>Indicates the compute unit instance name</td>
</tr>
<tr>
<td>Write Time</td>
<td>Specifies the total data write time on the port (in µs)</td>
</tr>
<tr>
<td>Outstanding Write (%)</td>
<td>Specifies the percentage of the runtime consumed in the write process</td>
</tr>
<tr>
<td>Read Time</td>
<td>Specifies the total data read time on the port (in µs)</td>
</tr>
</tbody>
</table>
Table 31: CU Port Data Transfers (cont’d)

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Outstanding Read (%)</td>
<td>Specifies the percentage of the runtime consumed in the read process</td>
</tr>
</tbody>
</table>

This following table displays the functional port data transfers on the compute unit.

Table 32: Functional Port Data Transfers

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Port</td>
<td>Name of port</td>
</tr>
<tr>
<td>Function</td>
<td>Name of function</td>
</tr>
<tr>
<td>Compute Unit</td>
<td>Name of compute unit</td>
</tr>
<tr>
<td>Write Time</td>
<td>Total time the port had an outstanding write (in µs)</td>
</tr>
<tr>
<td>Outstanding Write (%)</td>
<td>Percent time the port had an outstanding write</td>
</tr>
<tr>
<td>Read Time</td>
<td>Total time the port had an outstanding read (in µs)</td>
</tr>
<tr>
<td>Outstanding Read (%)</td>
<td>Percent time the port had an outstanding read</td>
</tr>
</tbody>
</table>

This following table displays the running time and stalls on the compute unit.

Table 33: Functions

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compute Unit</td>
<td>Name of compute unit</td>
</tr>
<tr>
<td>Function</td>
<td>Name of function</td>
</tr>
<tr>
<td>Running Time</td>
<td>Total time function was running (in ms)</td>
</tr>
<tr>
<td>Intra-Kernel Dataflow Stalls</td>
<td>Percent time the function was stalling from intra-kernel streams (in ms)</td>
</tr>
<tr>
<td>External Memory Stalls</td>
<td>Percent time the function was stalling from external memory accesses (in ms)</td>
</tr>
<tr>
<td>Inter-Kernel Pipe Stalls</td>
<td>Percent time the function was stalling from inter-kernel pipe accesses (in ms)</td>
</tr>
</tbody>
</table>

Shell Data Transfers

This following table displays the DMA data transfers.

Table 34: DMA Data Transfer

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Device</td>
<td>Name of device (format: &lt;device&gt;-&lt;ID&gt;)</td>
</tr>
<tr>
<td>Transfer Type</td>
<td>Type of data transfers</td>
</tr>
<tr>
<td>Number of Transfers</td>
<td>Number of data transfers (in AXI transactions)</td>
</tr>
<tr>
<td>Transfer Rate</td>
<td>Rate of data transfers (in MB/s): Transfer Rate = (Total Bytes) / (Total Time in µs)</td>
</tr>
<tr>
<td>Total Data Transfer</td>
<td>Total amount of data transferred (in MB)</td>
</tr>
<tr>
<td>Total Time</td>
<td>Total duration of data transfers (in ms)</td>
</tr>
</tbody>
</table>
Table 34: DMA Data Transfer (cont'd)

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Avg Size</td>
<td>Average size of data transfers (in KB): Average Size = (Total KB) / (Number of Transfers)</td>
</tr>
<tr>
<td>Avg Latency</td>
<td>Average latency of data transfers (in ns)</td>
</tr>
</tbody>
</table>

For DMA bypass and Global Memory to Global Memory data transfers, see the DMA Data Transfer table above.

NoC Counters

NoC Counters display the NoC Counters Read and NoC Counters Write. These sections are only displayed if there is a non-zero NoC counter data.

Each section has a table containing summary data with line graphs for transfer rate and latency. The graphs can have multiple NoC counters, so you will be able to toggle the counters ON/OFF through check boxes in the Chart column of the table.

Depending on the design, it can be possible to correlate NoC counters to CU ports. In this case, the CU port will appear in the table, and selecting it will cross-probe to the system diagram, profile summary, and any other views that include CU ports as selectable objects.

Table 35: NoC Counters Read or Write

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Name of NoC port</td>
</tr>
<tr>
<td>Traffic Class</td>
<td>Traffic class type</td>
</tr>
<tr>
<td>Requested QoS</td>
<td>QoS (MB/s): Requested quality of service (in MB/s)</td>
</tr>
<tr>
<td>Min Transfer Rate</td>
<td>Rate of minimum data transfers (in MB/s)</td>
</tr>
<tr>
<td>Avg Transfer Rate</td>
<td>Rate of average data transfers (in MB/s)</td>
</tr>
<tr>
<td>Max Transfer Rate</td>
<td>Rate of maximum data transfers (in MB/s)</td>
</tr>
<tr>
<td>Avg Size</td>
<td>Average size of data transfers (in KB): Average Size = (Total KB) / (Number of Transfers)</td>
</tr>
<tr>
<td>Min Latency</td>
<td>Minimum latency of data transfers (in ns)</td>
</tr>
<tr>
<td>Avg Latency</td>
<td>Average latency of data transfers (in ns)</td>
</tr>
<tr>
<td>Max Latency</td>
<td>Maximum latency of data transfers (in ns)</td>
</tr>
</tbody>
</table>

AI Engine Counters

AI Engine counters display if there is a non-zero AI Engine counter data. If there is an incompatible configuration of the AI Engine counters, this section will display a message stating that the configuration does not support performance profiling.

This section has a table containing summary data with line graphs for active time and usage. The usage chart is only available if stall profiling is enabled.
The graphs can have multiple AI Engine counters, so you will be able to toggle the counters ON/OFF through check boxes in the Chart column of the table.

It will be possible to cross-probe tiles to the AI Engine array and graph views.

**Table 36: AI Engine Counters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tile</td>
<td>AI Engine Tile [Column, Row]</td>
</tr>
<tr>
<td>Active Time (ms)</td>
<td>Amount of time (in ms) this tile was active</td>
</tr>
<tr>
<td>Stall Time (ms)</td>
<td>Amount of time (in ms) this tile was active but stalled</td>
</tr>
<tr>
<td>Stall Time (%)</td>
<td>Percent of time this tile was active but stalled</td>
</tr>
<tr>
<td>Active Utilization (ms)</td>
<td>Amount of time (in ms) this tile was active and not stalled</td>
</tr>
<tr>
<td>Active Utilization (%)</td>
<td>Percent of time this tile was active and not stalled</td>
</tr>
<tr>
<td>Clock Frequency (MHz)</td>
<td>Frequency (in MHz) of clock used for AI Engine tiles</td>
</tr>
</tbody>
</table>

**Application Timeline**

The Application Timeline collects and displays host and kernel events on a common timeline to help you understand and visualize the overall health and performance of your systems. The graphical representation lets you see issues regarding kernel synchronization and efficient concurrent execution. The displayed events include:

- OpenCL API calls from the host code.
- Device trace data including compute units, AXI transaction start/stop.
- Host events and kernel start/stops.

While this is useful for debugging and profiling the application, the timeline and device trace data are not collected by default, which can affect performance by adding time to the application execution. However, the trace data is collected with dedicated resources in the kernel, and does not affect kernel functionality. The data is offloaded only at the end of the run (v++ --trace_memory option).

The following is a snapshot of the Application Timeline window which displays host and device events on a common timeline. Host activity is displayed at the top of the image and kernel activity is shown on the bottom of the image. Host activities include creating the program, running the kernel and data transfers between global memory and the host. The kernel activities include read/write accesses and transfers between global memory and the kernel(s). This information helps you understand details of application execution and identify potential areas for improvements.
Timeline data can be enabled and collected through the command line flow. However, viewing must be done in the Vitis analyzer as described in Section VI: Using the Vitis Analyzer.

**Generating and Opening the Application Timeline**

To generate the Application Timeline report, you must complete the following steps to enable timeline and device trace data collection in the command line flow:

1. Instrument the FPGA binary during linking, by adding Acceleration Monitors and AXI Performance Monitors to kernels using the `v++ --profile` option as described in **--profile Options**. As an example, add `--profile.data` to the `v++` linking command line:

   ```
   v++ -g -l --profile.data all:all:all ...
   ```

2. After the kernels are instrumented during the build process, data gathering must also be enabled during the runtime execution of the application by editing the `xrt.ini` file. Refer to **xrt.ini File** for more information.

   The following `xrt.ini` file will enable maximum information gathering when the application is run:

   ```
   [Debug]
   profile=true
timeline_trace=true
data_transfer_trace=coarse
stall_trace=all
   ```
After running the application, the Application Timeline data is captured in a CSV file called `timeline_trace.csv`.

3. The CSV report can be viewed in the Vitis analyzer tool by opening the Run Summary produced during the application execution. You can launch the Vitis analyzer and open the Run Summary using the following command:

```
vitis_analyzer <project>.run_summary
```

### Interpreting the Application Timeline

The Application Timeline window displays host and device events on a common timeline. This information helps you understand details of application execution and identify potential areas for improvements. The Application Timeline report has two main sections: Host and Device. The Host section shows the trace of all the activity originating from the host side. The Device section shows the activity of the CUs on the FPGA.

The report has the following structure:

- **Host**
  - **OpenCL API Calls:** All OpenCL API calls are traced here. The activity time is measured from the host perspective.
    - **General:** All general OpenCL API calls such as `clCreateProgramWithBinary`, `clCreateContext`, and `clCreateCommandQueue`, are traced here.
    - **Queue:** OpenCL API calls that are associated with a specific command queue are traced here. This includes commands such as `clEnqueueMigrateMemObjects`, and `clEnqueueNDRangeKernel`. If the user application creates multiple command queues, then this section shows all the queues and activities.
  - **Data Transfer:** In this section the DMA transfers from the host to the device memory are traced. There are multiple DMA threads implemented in the OpenCL runtime and there is typically an equal number of DMA channels. The DMA transfer is initiated by the user application by calling OpenCL APIs such as `clEnqueueMigrateMemObjects`. These DMA requests are forwarded to the runtime which delegates to one of the threads. The data transfer from the host to the device appear under **Write** as they are written by the host, and the transfers from device to host appear under **Read**.
• **Kernel Enqueues:** The kernels enqueued by the host program are shown here. The kernels here should not be confused with the kernels/CUs on the device. Here kernel refers to the NDRangeKernels and tasks created by the OpenCL commands `clEnqueueNDRangeKernels` and `clEnqueueTask`. These are plotted against the time measured from the host's perspective. Multiple kernels can be scheduled to be executed at the same time, and they are traced from the point they are scheduled to run until the end of the kernel execution. This is the reason for multiple entries. The number of rows depend on the number of overlapping kernel executions.

  *Note:* Overlapping of the kernels should not be mistaken for actual parallel execution on the device as the process might not be ready to execute right away.

• **Device "name"**

• **Binary Container "name":** Binary container name.

• **Accelerator "name":** Name of the compute unit (a.k.a., Accelerator) on the FPGA.

  • **User Functions:** In the case of the Vitis HLS tool kernels, functions that are implemented as data flow processes are traced here. The trace for these functions show the number of active instances of these functions that are currently executing in parallel. These names are generated in hardware emulation when waveform is enabled.

    *Note:* Function level activity is only possible in hardware emulation.

    • Function: "name a"
    • Function: "name b"

• **Read:** A CU reads from the DDR over AXI-MM ports. The trace of a data read by a CU is shown here. The activity is shown as transaction and the tool-tip for each transaction shows more details of the AXI transaction. These names are generated when `--profile.data` is for the CU.

• **Write:** A CU writes to the DDR over AXI-MM ports. The trace of data written by a CU is shown here. The activity is shown as transactions and the tool-tip for each transaction shows more details of the AXI transaction. This is generated when `--profile.data` is specified for the CU.

---

**Waveform View and Live Waveform Viewer**

The Vitis core development kit can generate a Waveform view when running hardware emulation. It displays in-depth details at the system-level, CU level, and at the function level. The details include data transfers between the kernel and global memory and data flow through inter-kernel pipes. These details provide many insights into performance bottlenecks from the system-level down to individual function calls to help optimize your application.
The Live Waveform Viewer is similar to the Waveform view, however, it provides even lower-level details with some degree of interactivity. The Live Waveform Viewer can also be opened using the Vivado logic simulator, \texttt{xsim}.

\textbf{Note:} The Waveform view allows you to examine the device transactions from within the Vitis analyzer, as described in Section VI: Using the Vitis Analyzer. In contrast, the Live Waveform Viewer opens the Vivado simulation waveform viewer to examine the hardware transactions in addition to any user selected signals.

Waveform data is not collected by default because it requires the runtime to generate simulation waveforms during hardware emulation, which consumes more time and disk space. Refer to Generating and Opening the Waveform Reports for instructions on enabling these features.

\textbf{Figure 34: Waveform View}

You can also open the waveform database (\texttt{.wdb}) file with the Vivado logic simulator through the Linux command line:

\texttt{xsim -gui <filename.wdb> &}

\textbf{TIP:} The \texttt{.wdb} file is written to the directory where the compiled host code is executed.

\section*{Generating and Opening the Waveform Reports}

Follow these instructions to enable waveform data collection from the command line during hardware emulation and open the viewer:
1. Enable debug code generation during compilation and linking using the `-g` option.

   ```
v++ -c -g -t hw_emu ...
   ```

2. Create an `xrt.ini` file in the same directory as the host executable with the following contents (see `xrt.ini` File for more information):

   ```
   [Debug]
   profile=true
   timeline_trace=true
   
   [Emulation]
   debug_mode=batch
   ```

   The `debug_mode=batch` enables the capture of waveform data (`.wdb`) by running simulation in batch mode. You can also enable the Live Waveform Viewer to launch simulation in interactive mode using the following setting in the `xrt.ini`:

   ```
   [Emulation]
   debug_mode=gui
   ```

   **TIP:** If Live Waveform Viewer is enabled, the simulation waveform opens during the hardware emulation run.

3. Run the hardware emulation build of the application as described in Running the Application Hardware Build. The hardware transaction data is collected in the waveform database file, `<hardware_platform>-<device_id>-<xclbin_name>.wdb`. Refer to Output Directories of the `v++` Command or Output Directories from the Vitis IDE for more information on locating these reports.

4. Open the Waveform view in the Vitis analyzer by opening the Run Summary, and opening the Waveform report:

   ```
   vitis_analyzer <project>.run_summary
   ```

### Interpreting Data in the Waveform Views

The following image shows the Waveform view:
The Waveform and Live Waveform views are organized hierarchically for easy navigation.

- The Waveform view is based on the actual waveforms generated during hardware emulation (Kernel Trace). This allows the viewer to descend all the way down to the individual signals responsible for the abstracted data. However, because the Waveform view is generated from the post-processed data, no additional signals can be added to the report, and some of the runtime analysis cannot be visualized, such as DATAFLOW transactions.

- The Live Waveform viewer is displaying the Vivado logic simulator (xsim) run, so you can add extra signals and internals of the register transfer (RTL) design to the live view. Refer to the Vivado Design Suite User Guide: Logic Simulation (UG900) for information on working with the Waveform viewer.

The hierarchy of the Waveform and Live Waveform views include the following:
• **Device "name"**: Target device name.

• **Binary Container "name"**: Binary container name.

  • **Memory Data Transfers**: For each DDR Bank, this shows the trace of all the read and write request transactions arriving at the bank from the host.

  • **Kernel "name" 1:1:1**: For each kernel and for each compute unit of that kernel, this section breaks down the activities originating from the compute unit.

• **Compute Unit: "name"**: Compute unit name.

• **CU Stalls (%):** Stall signals are provided by the Vitis HLS tool to inform you when a portion of the circuit is stalling because of external memory accesses, internal streams (that is, dataflow), or external streams (that is, OpenCL pipes). The stall bus shown in detailed kernel trace compiles all of the lowest level stall signals and reports the percentage that are stalling at any point in time. This provides a factor of how much of the kernel is stalling at any point in the simulation.

  For example, if there are 100 lowest level stall signals and 10 are active on a given clock cycle, then the CU Stall percentage is 10%. If one goes inactive, then it is 9%.

• **Data Transfers**: This shows the read/write data transfer accesses originating from each Master AXI port of the compute unit to the DDR.

• **User Functions**: This information is available for the HLS kernels and shows the user functions.

  • **Function: "name"**: Function name.

    • **Dataflow/Pipeline Activity**: This shows the number of parallel executions of the function if the function is implemented as a dataflow process.

    • **Active Iterations**: This shows the currently active iterations of the dataflow. The number of rows is dynamically incremented to accommodate the visualization of any concurrent execution.

    • **StallNoContinue**: This is a stall signal that tells if there were any output stalls experienced by the dataflow processes (function is done, but it has not received a continue from the adjacent dataflow process).

    • **RTL Signals**: These are the underlying RTL control signals that were used to interpret the above transaction view of the dataflow process.

  • **Function Stalls**: Shows the different types of stalls experienced by the process.

    • **External Memory**: Stalls experienced while accessing the DDR memory.

    • **Internal-Kernel Pipe**: If the compute units communicated between each other through pipes, then this will show the related stalls.

  • **Intra-Kernel Dataflow**: FIFO activity internal to the kernel.
- **Function I/O**: Actual interface signals.
- **Function**: "name": Function name.
- **Function**: "name": Function name.
Host Optimization

This section focuses on optimization of the host program, which uses the OpenCL™ API to schedule the individual compute unit executions, and data transfers to and from the FPGA board. For optimizing data transfer and compute calls, you need to think about concurrent execution of tasks through the OpenCL command queue(s). This section discusses common pitfalls, and how to recognize and address them.

Reducing Overhead of Kernel Enqueing

The OpenCL-based execution model supports data parallel and task parallel programming models. An OpenCL host generally needs to call different kernels multiple times. These calls are enqueued in a command queue, either in a certain sequence, or in an out-of-order command queue. Then depending on the availability of compute resources and task data they get scheduled for execution on the device.

Kernel calls can be enqueued for execution on a command queue using `clEnqueueTask`. The dispatching process is executed on the host processor. The dispatcher invokes kernel execution after transferring the kernel arguments to the accelerator running on the device. The dispatcher uses a low-level Xilinx® Runtime (XRT) library for transferring kernel arguments and issuing trigger commands for starting the compute. The overhead of dispatching the commands and arguments to the accelerator can be between 30 µs and 60 µs, depending on the number of arguments set for the kernel. You can reduce the impact of this overhead by minimizing the number of times the kernel needs to be executed, and minimizing calls to `clEnqueueTask`. Ideally, you should finish all the compute in a single call to `clEnqueueTask`.

You can minimize the calls to `clEnqueueTask` by batching your data and invoking the kernel one time, with a loop wrapped around the original implementation to avoid the overhead of multiple enqueue calls. It can also improve data transfer performance between the host and accelerator, by transferring fewer large data packets rather than many small data packets. For more information on reducing overhead on kernel execution, see Kernel Execution.
The following example shows a simple kernel with given work or data size to process.

```c
#define SIZE 256
extern "C" {
    void add(int *a , int *b, int inc){
        int buff_a[SIZE];
        for(int i=0;i<size;i++)
        {
            buff_a[i] = a[i];
        }
        for(int i=0;i<size;i++)
        {
            b[i] = a[i]+inc;
        }
    }
}
```

The following example shows the same simple kernel optimized to process batched data. Depending on the `num_batches` argument the kernel can process multiple inputs of size 256 in a single call and avoid the overhead of multiple `clEnqueueTask` calls. The host application changes to allocate data and buffers in chunks of `SIZE * num_batches`, essentially batching the memory allocation and transfer of data between the host global and device memory.

```c
#define SIZE 256
extern "C" {
    void add(int *a , int *b, int inc, int num_batches){
        int buff_a[SIZE];
        for(int j=0;j<num_batches;j++)
        {
            for(int i=0;i<size;i++)
            {
                buff_a[i] = a[i];
            }
            for(int i=0;i<size;i++)
            {
                b[i] = a[i]+inc;
            }
        }
    }
}
```
In the OpenCL execution model, all data is transferred from the host main memory to the global device memory first, and then from the global device memory to the kernel for computation. The computation results are written back from the kernel to the global device memory, and lastly from the global memory to the host main memory. A key factor in determining strategies for kernel data movement optimization is understanding how data can be efficiently moved around between different level of memories maximizing the efficient use of bandwidth on all the memory interfaces.
**RECOMMENDED:** Optimize the data movement in the application before optimizing computation.

During data movement optimization, it is important to isolate data transfer code from computation code because inefficiency in computation might cause stalls in data movement. You should focus on modifying the data transfer logic in the host and kernel code during this optimization step. The goal is to maximize the system level data throughput by maximizing data transfer bandwidth and device global memory bandwidth usage. It usually takes multiple iterations of running software emulation, hardware emulation, as well as execution in hardware to achieve optimum performance.

**Overlapping Data Transfers with Kernel Computation**

Applications, such as database analytics, have a much larger data set than can be stored in the available global device memory on the acceleration device. They require the complete data to be transferred and processed in blocks. Techniques that overlap the data transfers with the computation are critical to achieve high performance for these applications.

An example can be found in the `vadd` kernel from the `overlap` example in the `host` category of Vitis Accelerated Examples on GitHub. This examples demonstrates techniques to overlap Host (CPU) and FPGA computation in the application. In this example, the kernel processes two arrays by adding them together and writing to output. From the host perspective, there are four tasks to perform in this example:

1. Write buffer a (`W_a`)
2. Write buffer b (`W_b`)
3. Execute `vadd` kernel
4. Read buffer c (`R_c`)

Using a simple in-order command queue without data transfer optimization, the overall execution timeline trace should look similar to the one shown below:

*Figure 37: Host View of Tasks*
Using an out-of-order command queue, data transfer and kernel execution can overlap as illustrated in the figure below. In the host code for this example, double buffering is used for all buffers so that the kernel can process one set of buffers while the host can operate on the other set of buffers.

The OpenCL event object provides an easy method to set up complex operation dependencies and synchronize host threads and device operations. Events are OpenCL objects that track the status of operations. Event objects are created by kernel execution commands, read, write, and copy commands on memory objects, or user events created using `clCreateUserEvent`.

You can ensure an operation has completed by querying the events returned by these commands. The arrows in the figure below show how event triggering can be set up to achieve optimal performance.

**Figure 38: Event Triggering Setup**

In the example, the host code (`host.cpp`) enqueues the four tasks in a loop to process the complete data set. It also sets up event synchronization between different tasks to ensure that data dependencies are met for each task. The double buffering is set up by passing different memory objects values to `clEnqueueMigrateMemObjects` API. The event synchronization is achieved by having each API call wait for other event as well as trigger its own event when the API completes.

The Application Timeline view below clearly shows that the data transfer time is completely hidden, while the compute unit `vadd_1` is running constantly.

**Figure 39: Data Transfer Time Hidden in Application Timeline View**
**Buffer Memory Segmentation**

Allocation and deallocation of memory buffers can lead to memory segmentation in the DDR controllers. This might result in sub-optimal performance of compute units, even if they could theoretically execute in parallel.

This issue occurs most often when multiple pthreads for different compute units are used and the threads allocate and release many device buffers with different sizes every time they enqueue the kernels. In this case, the timeline trace will exhibit gaps between kernel executions and it might seem the processes are sleeping.

Each buffer allocated by runtime should be continuous in hardware. For large memory, it might take some time to wait for that space to be freed, when many buffers are allocated and deallocated. This can be resolved by allocating device buffer and reusing it between different enqueues of a kernel.

**Compute Unit Scheduling**

Scheduling kernel operations is key to overall system performance. This becomes even more important when implementing multiple compute units (of the same kernel or of different kernels). This section examines the different command queues responsible for scheduling the kernels.

**Multiple In-Order Command Queues**

The following figure shows an example with two in-order command queues, CQ0 and CQ1. The scheduler dispatches commands from each queue in order, but commands from CQ0 and CQ1 can be pulled out by the scheduler in any order. You must manage synchronization between CQ0 and CQ1 if required.

*Figure 40: Example with Two In-Order Command Queues*
The following is code extracted from host.cpp of the concurrent_kernel_execution_c example that sets up multiple in-order command queues and enqueues commands into each queue:

```c
OCL_CHECK(err,  
    cl::CommandQueue ooo_queue(context, 
        device,  
        CL_QUEUE_PROFILING_ENABLE |  
        CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE,  
        &err));
...

    printf('[OOO Queue]: Enqueueing scale kernel\n');
    OCL_CHECK(err,  
        err = ooo_queue.enqueueTask(
            kernel_mscale, nullptr, &ooo_events[0]));
    set_callback(ooo_events[0], "scale");
...

    // This is an out of order queue, events can be executed in any order.  
    // this call depends on the results of the previous call we must pass  
    // the event object from the previous call to this kernel's event wait list.  
    printf('[OOO Queue]: Enqueueing addition kernel (Depends on scale)\n');
    kernel_wait_events.resize(0);
    kernel_wait_events.push_back(ooo_events[0]);
    OCL_CHECK(err,  
        err = ooo_queue.enqueueTask(
            kernel_madd,  
            &kernel_wait_events, // Event from previous call  
            &ooo_events[1]));
    set_callback(ooo_events[1], "addition");
...

    // This call does not depend on previous calls so we are passing nullptr  
    // into the event wait list. The runtime should schedule this kernel in  
    // parallel to the previous calls.  
    printf('[OOO Queue]: Enqueueing matrix multiplication kernel\n');
    OCL_CHECK(err,  
        err = ooo_queue.enqueueTask(
            kernel_mmult,  
            nullptr,  
            &ooo_events[2]));
    set_callback(ooo_events[2], "matrix multiplication");
```

**Single Out-of-Order Command Queue**

The following figure shows an example with a single out-of-order command queue. The scheduler can dispatch commands from the queue in any order. You must manually define event dependencies and synchronizations as required.
The following is code extracted from `host.cpp` of the `concurrent_kernel_execution_c` example that sets up a single out-of-order command queue and enqueues commands as needed:

```cpp
OCL_CHECK(err,
    cl::CommandQueue ooo_queue(context,
                               device,
                               CL_QUEUE_PROFILING_ENABLE |
                               CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE,
                               &err));
...
    printf('[OOO Queue]: Enqueueing scale kernel\n');
    OCL_CHECK(err,
        err = ooo_queue.enqueueTask(
            kernel_mscale, nullptr, &ooo_events[0]));
    set_callback(ooo_events[0], "scale");
...
    // This is an out of order queue, events can be executed in any order.
    // Since this call depends on the results of the previous call we must pass the
    // event object from the previous call to this kernel's event wait list.
    printf('[OOO Queue]: Enqueueing addition kernel (Depends on scale)\n');
    kernel_wait_events.resize(0);
    kernel_wait_events.push_back(ooo_events[0]);
    OCL_CHECK(err,
        err = ooo_queue.enqueueTask(
            kernel_madd,
            &kernel_wait_events, // Event from previous call
            &ooo_events[1]));
    set_callback(ooo_events[1], "addition");
    // This call does not depend on previous calls so we are passing nullptr
    // into the event wait list. The runtime should schedule this kernel in
    // parallel to the previous calls.
    printf('[OOO Queue]: Enqueueing matrix multiplication kernel\n');
    OCL_CHECK(err,
        err = ooo_queue.enqueueTask(
            kernel_mmult,
            nullptr,
            &ooo_events[2]));
    set_callback(ooo_events[2], "matrix multiplication");
```
The Application Timeline view shows that the compute unit \texttt{mmult\_1} is running in parallel with the compute units \texttt{mscale\_1} and \texttt{madd\_1}, using both multiple in-order queues and single out-of-order queue methods.

\textit{Figure 42: Application Timeline View Showing mult\_1 Running with mscale\_1 and madd\_1}

---

**Kernel Optimization**

One of the key advantages of an FPGA is its flexibility and capacity to create customized designs specifically for your algorithm. This enables various implementation choices to trade off algorithm throughput versus power consumption. The following guidelines help manage the design complexity and achieve the desired design goals.
The goal of kernel optimization is to create processing logic that can consume all the data as soon as it arrives at the kernel interfaces. The key metric is the initiation interval (II), or the number of clock cycles before the kernel can accept new input data. Optimizing the II is generally achieved by expanding the processing code to match the data path with techniques such as function pipelining, loop unrolling, array partitioning, data flowing, etc.
Interface Attributes (Detailed Kernel Trace)

The detailed kernel trace provides easy access to the AXI transactions and their properties. The AXI transactions are presented for the global memory, as well as the Kernel side (Kernel "pass" 1:1:1) of the AXI interconnect. The following figure illustrates a typical kernel trace of a newly accelerated algorithm.

![Accelerated Algorithm Kernel Trace](image)

Most interesting with respect to performance are the fields:

- **Burst Length**: Describes how many packages are sent within one transaction.
- **Burst Size**: Describes the number of bytes being transferred as part of one package.

Given a burst length of 1 and just 4 bytes per package, it will require many individual AXI transactions to transfer any reasonable amount of data.

**Note**: The Vitis core development kit never creates burst sizes less than 4 bytes, even if smaller data is transmitted. In this case, if consecutive items are accessed without AXI bursts enabled, it is possible to observe multiple AXI reads to the same address.

Small burst lengths, as well as burst sizes, considerably less than 512 bits are therefore good opportunities to optimize interface performance.
Using Burst Data Transfers

Transferring data in bursts hides the memory access latency and improves bandwidth usage and efficiency of the memory controller. Also, check the HLS report for bursting information.

**RECOMMENDED:** Infer burst transfers from successive requests of data from consecutive address locations. Refer to *Reading and Writing by Burst* for more details.

If burst data transfers occur, the detailed kernel trace will reflect the higher burst rate as a larger burst length number:

*Figure 45: Burst Data Transfer with Detailed Kernel Trace*
In the previous figure, it is also possible to observe that the memory data transfers following the AXI interconnect are actually implemented rather differently (shorter transaction time). Hover over these transactions, you would see that the AXI interconnect has packed the 16 x 4 byte transaction into a single package transaction of 1 x 64 bytes. This effectively uses the AXI4 bandwidth which is even more favorable. The next section focuses on this optimization technique in more detail.

Burst inference is heavily dependent on coding style and access pattern. However, you can ease burst detection and improve performance by isolating data transfer and computation, as shown in the following code snippet:

```c
void kernel(T in[1024], T out[1024]) {
    T tmpIn[1024];
    T tmpOut[1024];
    read(in, tmpIn);
    process(tmpIn, tmpOut);
    write(tmpOut, out);
}
```

In short, the function `read` is responsible for reading from the AXI input to an internal variable (tmpIn). The computation is implemented by the function `process` working on the internal variables `tmpIn` and `tmpOut`. The function `write` takes the produced output and writes to the AXI output. For more information on burst, see the *Vitis High-Level Synthesis User Guide* (UG1399).

The isolation of the read and write function from the computation results in:

- Simple control structures (loops) in the read/write function which makes burst detection simpler.
- The isolation of the computational function away from the AXI interfaces, simplifies potential kernel optimization. See *Kernel Optimization* for more information.
- The internal variables are mapped to on-chip memory, which allow faster access compared to AXI transactions. Acceleration platforms supported in the Vitis core development kit can have as much as 10 MB on-chip memories that can be used as pipes, local memories, and private memories. Using these resources effectively can greatly improve the efficiency and performance of your applications.

**Using Full AXI Data Width**

The user data width between the kernel and the memory controller can be configured by the Vitis compiler based on the data types of the kernel arguments. To maximize the data throughput, Xilinx recommends that you choose data types map to the full data width on the memory controller. The memory controller in all supported acceleration cards supports 512-bit user interface, which can be mapped to C/C++ arbitrary precision data type `ap_int<512>` or OpenCL vector data types such as `int16`. 
As described in Memory Interface Width Considerations, the default is for Vitis HLS to automatically re-size the kernel interface ports up to 512-bits to improve burst access. As shown on the following figure, you can observe burst AXI transactions (Burst Length 16) and a 512-bit package size (Burst Size 64 bytes).

**Figure 46: Burst AXI Transactions**

This example shows good interface configuration as it maximizes AXI data width as well as actual burst transactions.

Complex structs or classes, used to declare interfaces, can lead to very complex hardware interfaces due to memory layout and data packing differences. This can introduce potential issues that are very difficult to debug in a complex system.

**RECOMMENDED:** Use simple structs for kernel arguments that can be packed to 32-bit boundary. Refer to the Custom Data Type Example in kernel_to_gmem category at Xilinx Getting Started Example on GitHub for the recommended method to use structs.
Optimizing Kernel to Kernel Communication

Support for hardware accelerator pipelines that communicate through streams is one of the major advantages of FPGAs and FPGA-based SoCs and have been used in DSP and image processing applications, as well as in communication systems. As described in Streaming Data Transfers between Kernels (K2K), AXI4-Stream interfaces can be used to stream data from one kernel to another without having to use the external memory, which greatly improves the overall system latency.

Kernel ports involved in streaming are defined within the kernel, and are not addressed by the host program. There is no need to send data back to global memory before it is forwarded for processing. The connections between the kernels are directly defined during the v++ linking process as described in Specifying Streaming Connections between Compute Units.

Optimizing Memory Architecture

Memory architecture is a key aspect of implementation. Due to the limited access bandwidth, it can heavily impact the overall performance, as shown in the following example:

```c
void run (ap_uint<16> in[256][4],
         ap_uint<16> out[256]) {
    ...
    ap_uint<16> inMem[256][4];
    ap_uint<16> outMem[256];
    ...
    // Preprocess input to local memory
    for( int j=0; j<256; j++) {
        #pragma HLS PIPELINE OFF
        ap_uint<16> sum = 0;
        for( int i = 0; i<4; i++) {
            sum += inMem[j][i];
        }
        outMem[j] = sum;
    }
    ...
    // Postprocess write local memory to output
}
```

This code adds the four values associated with the inner dimension of the two dimensional input array. If implemented without any additional modifications, it results in the following estimates:
Figure 47: Performance Estimates

<table>
<thead>
<tr>
<th>Timing (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Summary</strong></td>
</tr>
<tr>
<td>Clock</td>
</tr>
<tr>
<td>ap_clk</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Latency (clock cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Summary</strong></td>
</tr>
<tr>
<td>Latency</td>
</tr>
<tr>
<td>min</td>
</tr>
<tr>
<td>5908</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Detail</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Instance</strong></td>
</tr>
<tr>
<td><strong>Loop</strong></td>
</tr>
<tr>
<td>Loop Name</td>
</tr>
<tr>
<td>Loop 1</td>
</tr>
<tr>
<td>Loop 2</td>
</tr>
<tr>
<td>Loop 2.1</td>
</tr>
<tr>
<td>Loop 3</td>
</tr>
</tbody>
</table>

The overall latency of 4608 (Loop 2) is due to 256 iterations of 18 cycles (16 cycles spent in the inner loop, plus the reset of sum, plus the output being written). This is observed in the Schedule Viewer in the HLS Project. The estimates become considerably better when unrolling the inner loop.
However, this improvement is largely because of the process using both ports of a dual port memory. This can be seen from the Schedule Viewer in the HLS Project:

**Figure 48: Performance Estimates**

<table>
<thead>
<tr>
<th>Timing (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Summary</strong></td>
</tr>
<tr>
<td>Clock</td>
</tr>
<tr>
<td>ap_clk</td>
</tr>
</tbody>
</table>

**Latency (clock cycles)**

<table>
<thead>
<tr>
<th><strong>Summary</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Latency</td>
</tr>
<tr>
<td>min</td>
</tr>
<tr>
<td>2580</td>
</tr>
</tbody>
</table>

**Detail**

**Instance**

**Loop**

<table>
<thead>
<tr>
<th>Loop Name</th>
<th>Latency</th>
<th>Initiation Interval</th>
<th>Iteration Latency</th>
<th>Target</th>
<th>Trip Count</th>
<th>Pipelined</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Loop 1</td>
<td>1034</td>
<td>12</td>
<td>1034</td>
<td>-</td>
<td>-</td>
<td>yes</td>
</tr>
<tr>
<td>- Loop 2</td>
<td>1280</td>
<td>5</td>
<td>1280</td>
<td>-</td>
<td>-</td>
<td>no</td>
</tr>
<tr>
<td>- Loop 3</td>
<td>257</td>
<td>3</td>
<td>257</td>
<td>1</td>
<td>1</td>
<td>yes</td>
</tr>
</tbody>
</table>

**Figure 49: Schedule Viewer**

**Operation\Control Step**

<table>
<thead>
<tr>
<th>Loop 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>j5(phi_mux)</td>
</tr>
<tr>
<td>exitcond2(cmp)</td>
</tr>
<tr>
<td>j3(+)</td>
</tr>
<tr>
<td>tmp_s(+)</td>
</tr>
<tr>
<td>inMem_V_load(read)</td>
</tr>
<tr>
<td>inMem_V_load_1(read)</td>
</tr>
<tr>
<td>tmp_6(+)</td>
</tr>
<tr>
<td>tmp_11(+)</td>
</tr>
<tr>
<td>inMem_V_load_2(read)</td>
</tr>
<tr>
<td>inMem_V_load_3(read)</td>
</tr>
<tr>
<td>tmp1(+)</td>
</tr>
<tr>
<td>tmp2(+)</td>
</tr>
<tr>
<td>sum_V_3(+)</td>
</tr>
<tr>
<td>node_81(write)</td>
</tr>
</tbody>
</table>
Two read operations are performed per cycle to access all the values from the memory to calculate the sum. This is often an undesired result as this completely blocks the access to the memory. To further improve the results, the memory can be split into four smaller memories along the second dimension:

```
#pragma HLS ARRAY_PARTITION variable=inMem complete dim=2
```

For more information, see `pragma HLS array_partition`.

This results in four array reads, all executed on different memories using a single port:

**Figure 50: Executed Four Arrays Results**

Using a total of $256 \times 4$ cycles $= 1024$ cycles for loop 2.
Alternatively, the memory can be reshaped into a single memory with four words in parallel. This is performed through the pragma:

```
#pragma HLS array_reshape variable=inMem complete dim=2
```

For more information, see `pragma HLS array_reshape`.

This results in the same latency as when the array partitioning, but with a single memory using a single port:
Although, either solution creates comparable results with respect to overall latency and utilization, reshaping the array results in cleaner interfaces and less routing congestion making this the preferred solution.

Note: This completes array optimization, in a real design the latency could be further improved by employing loop parallelism (see Loop Parallelism).

```c
void run (ap_uint<16> in[256][4],
         ap_uint<16> out[256]) {
    ...
    ap_uint<16> inMem[256][4];
    ap_uint<16> outMem[256];
    #pragma HLS array_reshape variable=inMem complete dim=2
    ... Preprocess input to local memory
    for( int j=0; j<256; j++) {
        #pragma HLS PIPELINE OFF
        ap_uint<16> sum = 0;
        for( int i = 0; i<4; i++) {
            #pragma HLS UNROLL
            sum += inMem[j][i];
        }
        outMem[j] = sum;
    }
    ... Postprocess write local memory to output
}
```

**Optimizing Computational Parallelism**

By default, C/C++ does not model computational parallelism, as it always executes any algorithm sequentially. However, fully configurable computational engines like FPGAs allow more freedom to exploit computational parallelism.

**Coding Data Parallelism**

To leverage computational parallelism during the implementation of an algorithm on the FPGA, it should be mentioned that the synthesis tool will need to be able to recognize computational parallelism from the source code first. Loops and functions are prime candidates for reflecting computational parallelism and compute units in the source description. However, even in this case, it is key to verify that the implementation takes advantage of the computational parallelism as in some cases the Vitis technology might not be able to apply the desired transformation due to the structure of the source code.
It is quite common, that some computational parallelism might not be reflected in the source code to begin with. In this case, it will need to be added. A typical example is a kernel that might be described to operate on a single input value, while the FPGA implementation might execute computations more efficiently in parallel on multiple values. This kind of parallel modeling is described in Task Parallelism.

A 512-bit interface can be created using OpenCL vector data types such as int16 or C/C++ arbitrary precision data type ap_int<512>. These vector types can also be used as a powerful way to model data parallelism within a kernel, with up to 16 data paths operating in parallel in case of int16. Refer to the Median Filter Example in the vision category at Xilinx Getting Started Example on GitHub for the recommended method to use vectors.

**Loop Parallelism**

Loops are the basic C/C++/OpenCL API method of representing repetitive algorithmic code. The following example illustrates various implementation aspects of a loop structure:

```c
for(int i = 0; i<255; i++) {
    out[i] = in[i]+in[i+1];
}
out[255] = in[255];
```

This code iterates over an array of values and adds consecutive values, except the last value. If this loop is implemented as written, each loop iteration requires two cycles for implementation, which results in a total of 510 cycles for implementation. This can be analyzed in detail through the Schedule Viewer in the HLS Project.
This can also be analyzed in terms of total numbers and latency through the Vivado synthesis results:
The key numbers here are the latency numbers and total LUT usage. For example, depending on the configuration, you could get latency of 511 and total LUT usage of 47. As a result, these values can vary based on the implementation choices. While this implementation will require very little area, it results in significant latency.

**Unrolling Loops**

Unrolling a loop enables the full parallelism of the model to be used. To perform this, mark a loop to be unrolled and the tool will create the implementation with the most parallelism possible. To mark a loop to unroll, an OpenCL loop can be marked with the UNROLL attribute:

```
__attribute__((opencl_unroll_hint))
```
Or a C/C++ loop can use the unroll pragma:

```c
#pragma HLS UNROLL
```

For more information, see Loop Unrolling.

When applied to this specific example, the Schedule Viewer in the HLS Project will be:

*Figure 55: Schedule Viewer*

The following figure shows the estimated performance:
Therefore, the total latency was considerably improved to be 127 cycles and as expected the computational hardware was increased to 4845 LUTs, to perform the same computation in parallel.

However, if you analyze the for-loop, you might ask why this algorithm cannot be implemented in a single cycle, as each addition is completely independent of the previous loop iteration. The reason is the memory interface is used for the variable `out`. The Vitis core development kit uses dual port memory by default for an array. However, this implies that at most two values can be written to the memory per cycle. Thus to see a fully parallel implementation, you must specify that the variable `out` should be kept in registers as in this example:

```c
#pragma HLS array_partition variable= out complete dim= 0
```

For more information, see `pragma HLS array_partition`.
The results of this transformation can be observed in the following Schedule Viewer:

**Figure 57: Transformation Results in Schedule Viewer**

<table>
<thead>
<tr>
<th>Current Module</th>
<th>Operation Control Step</th>
<th>Step</th>
</tr>
</thead>
<tbody>
<tr>
<td>run</td>
<td>in_0_v_read(read)</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>in_1_v_read(read)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>tmp_3(+)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>node_1029(write)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>in_2_v_read(read)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>tmp_3_1(+)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>node_1032(write)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>in_3_v_read(read)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>tmp_3_2(+)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>node_1035(write)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>in_4_v_read(read)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>tmp_3_3(+)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>node_1038(write)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>in_5_v_read(read)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>tmp_3_4(+)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>node_1041(write)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>in_6_v_read(read)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>tmp_3_5(+)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>node_1044(write)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>in_7_v_read(read)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>tmp_3_6(+)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>node_1047(write)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>in_8_v_read(read)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>tmp_3_7(+)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>node_1050(write)</td>
<td></td>
</tr>
</tbody>
</table>

The associated estimates are:
Accordingly, this code can be implemented as a combinatorial function requiring only a fraction of the cycle to complete.

**Pipelining Loops**

Pipelining loops allow you to overlap iterations of a loop in time, as discussed in Loop Pipelining. Allowing loop iterations to operate concurrently is often a good approach, as resources can be shared between iterations (less resource utilization), while requiring less execution time compared to loops that are not unrolled.

Pipelining is enabled in C/C++ through the `pragma HLS pipeline`:

```
#pragma HLS PIPELINE
```
While the OpenCL API uses the `xcl_pipeline_loop` attribute:

```c
__attribute__((xcl_pipeline_loop))
```

**Note:** The OpenCL API has an additional method of specifying loop pipelining, see `xcl_pipeline_workitems`. The reason is the work item loops are not explicitly stated and pipelining these loops require this attribute:

```c
__attribute__((xcl_pipeline_workitems))
```

In this example, the Schedule Viewer in the HLS Project produces the following information:

**Figure 59: Pipelining Loops in Schedule Viewer**

![Schedule Viewer screenshot]

With the overall estimates being:
Because each iteration of a loop consumes only two cycles of latency, there can only be a single iteration overlap. This enables the total latency to be cut into half compared to the original, resulting in 257 cycles of total latency. However, this reduction in latency was achieved using fewer resources when compared to unrolling.

In most cases, loop pipelining by itself can improve overall performance. Yet, the effectiveness of the pipelining depends on the structure of the loop. Some common limitations are:

- Resources with limited availability such as memory ports or process channels can limit the overlap of the iterations (Initiation Interval).
- Loop-carry dependencies, such as those created by variable conditions computed in one iteration affecting the next, might increase the II of the pipeline.
These are reported by the tool during high-level synthesis and can be observed and examined in the Schedule Viewer. For the best possible performance, the code might have to be modified to remove these limiting factors, or the tool needs to be instructed to eliminate some dependency by restructuring the memory implementation of an array, or breaking the dependencies all together.

**Task Parallelism**

Task parallelism allows you to take advantage of dataflow parallelism. In contrast to loop parallelism, when task parallelism is deployed, full execution units (tasks) are allowed to operate in parallel taking advantage of extra buffering introduced between the tasks.

See the following example:

```c
void run (ap_uint<16> in[1024],
    ap_uint<16> out[1024]) {
    ap_uint<16> tmp[128];
    for(int i = 0; i<8; i++) {
        processA(&(in[i*128]), tmp);
        processB(tmp, &(out[i*128]));
    }
}
```

When this code is executed, the function `processA` and `processB` are executed sequentially 128 times in a row. Given the combined latency for `processA` and `processB`, the loop is set to 278 and the total latency can be estimated as:

*Figure 61: Performance Estimates*

<table>
<thead>
<tr>
<th>Performance Estimates</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Timing (ns)</strong></td>
</tr>
<tr>
<td><strong>Summary</strong></td>
</tr>
<tr>
<td>Clock</td>
</tr>
<tr>
<td>ap_clk</td>
</tr>
<tr>
<td><strong>Latency (clock cycles)</strong></td>
</tr>
<tr>
<td><strong>Summary</strong></td>
</tr>
<tr>
<td>Latency</td>
</tr>
<tr>
<td>min</td>
</tr>
<tr>
<td>35585</td>
</tr>
<tr>
<td><strong>Detail</strong></td>
</tr>
<tr>
<td><strong>Instance</strong></td>
</tr>
<tr>
<td><strong>Loop</strong></td>
</tr>
</tbody>
</table>

The extra cycle is due to loop setup and can be observed in the Schedule Viewer.
For C/C++ code, task parallelism is performed by adding the DATAFLOW pragma into the for-loop:

```c
#pragma HLS DATAFLOW
```

For OpenCL API code, add the attribute before the for-loop:

```c
__attribute__((xcl_dataflow))
```

Refer to Dataflow Optimization, HLS Pragmas, and OpenCL Attributes for more details on this topic.

As illustrated by the estimates in the HLS report, applying the transformation will considerably improve the overall performance effectively using a double (ping-pong) buffer scheme between the tasks:

![Figure 62: Performances Estimates](image)

The overall latency of the design has almost halved in this case due to concurrent execution of the different tasks of the different iterations. Given the 139 cycles per processing function and the full overlap of the 128 iterations, this allows the total latency to be:

```
(1x only processA + 127x both processes + 1x only processB) * 139 cycles = 17931 cycles
```

Using task parallelism is a powerful method to improve performance when it comes to implementation. However, the effectiveness of applying the DATAFLOW pragma to a specific and arbitrary piece of code might vary vastly. It is often necessary to look at the execution pattern of the individual tasks to understand the final implementation of the DATAFLOW pragma. Finally, the Vitis core development kit provides the Detailed Kernel Trace, which illustrates concurrent execution.
For this Detailed Kernel Trace, the tool displays the start of the dataflow loop, as shown in the previous figure. It illustrates how processA is starting up right away with the beginning of the loop, while processB waits until the completion of the processA before it can start up its first iteration. However, while processB completes the first iteration of the loop, processA begins operating on the second iteration, etc.

A more abstract representation of this information is presented in Application Timeline for the host and device activity.
Optimizing Device Resources

Data Width

One, if not the most important aspect for performance is the data width required for the implementation. The tool propagates port widths throughout the algorithm. In some cases, especially when starting out with an algorithmic description, the C/C++/OpenCL code might only use large data types such as integers even at the ports of the design. However, as the algorithm is mapped to a fully configurable implementation, smaller data types such as 10-/12-bit might often suffice. It is beneficial to check the size of basic operations in the HLS Synthesis report during optimization.

In general, when the Vitis core development kit maps an algorithm onto the FPGA, more processing is required to comprehend the C/C++/OpenCL API structure and extract operational dependencies. Therefore, to perform this mapping the Vitis core development kit generally partitions the source code into operational units which are then mapped onto the FPGA. Several aspects influence the number and size of these operational units (ops) as seen by the tool.

In the following figure, the basic operations and their bit-width are reported.
Figure 64: Operations Utilization Estimates

<table>
<thead>
<tr>
<th>Summary</th>
<th>BRAM_18K</th>
<th>DSP48E</th>
<th>FF</th>
<th>LUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>DSP</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Expression</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>102</td>
</tr>
<tr>
<td>FIFO</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Instance</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Memory</td>
<td>0</td>
<td>-</td>
<td>24</td>
<td>12</td>
</tr>
<tr>
<td>Multiplexer</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>80</td>
</tr>
<tr>
<td>Register</td>
<td>-</td>
<td>-</td>
<td>51</td>
<td>-</td>
</tr>
<tr>
<td>Total</td>
<td>0</td>
<td>0</td>
<td>75</td>
<td>194</td>
</tr>
<tr>
<td>Available</td>
<td>4320</td>
<td>5520</td>
<td>1326720</td>
<td>663360</td>
</tr>
<tr>
<td>Available SLR</td>
<td>2160</td>
<td>2760</td>
<td>663360</td>
<td>331680</td>
</tr>
<tr>
<td>Utilization (%)</td>
<td>0</td>
<td>0</td>
<td>~0</td>
<td>~0</td>
</tr>
<tr>
<td>Utilization SLR (%)</td>
<td>0</td>
<td>0</td>
<td>~0</td>
<td>~0</td>
</tr>
</tbody>
</table>

**Detail**

**Instance**

**DSP48**

**Memory**

**FIFO**

**Expression**

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Operation</th>
<th>DSP48E</th>
<th>FF</th>
<th>LUT</th>
<th>Bitwidth P0</th>
<th>Bitwidth P1</th>
</tr>
</thead>
<tbody>
<tr>
<td>i_1_fu_124_p2</td>
<td>+</td>
<td>0</td>
<td>0</td>
<td>11</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>i_2_fu_148_p2</td>
<td>+</td>
<td>0</td>
<td>0</td>
<td>15</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>i_3_fu_179_p2</td>
<td>+</td>
<td>0</td>
<td>0</td>
<td>15</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>sum_i0_fu_194_p2</td>
<td>+</td>
<td>0</td>
<td>0</td>
<td>15</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>sum_i1_fu_158_p2</td>
<td>+</td>
<td>0</td>
<td>0</td>
<td>15</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>exitcond_fu_118_p2</td>
<td>icmp</td>
<td>0</td>
<td>0</td>
<td>9</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>exitcond_i6_fu_173_p2</td>
<td>icmp</td>
<td>0</td>
<td>0</td>
<td>11</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>exitcond_i8_fu_142_p2</td>
<td>icmp</td>
<td>0</td>
<td>0</td>
<td>11</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>Total</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>102</td>
<td>50</td>
<td>39</td>
</tr>
</tbody>
</table>

**Multiplexer**

**Register**

Look for bit widths of 16, 32, and 64 bits commonly used in algorithmic descriptions and verify that the associated operation from the C/C++/OpenCL API source actually requires the bit width to be this large. This can considerably improve the implementation of the algorithm, as smaller operations require less computation time.

**Fixed Point Arithmetic**

Some applications use floating point computation only because they are optimized for other hardware architecture. Using fixed point arithmetic for applications like deep learning can save the power efficiency and area significantly while keeping the same level of accuracy.
RECOMMENDED: Xilinx recommends exploring fixed point arithmetic for your application before committing to using floating point operations.

Macro Operations

It is sometimes advantageous to think about larger computational elements. The tool will operate on the source code independently of the remaining source code, effectively mapping the algorithm without consideration of surrounding operations onto the FPGA. When applied, the Vitis technology keeps operational boundaries, effectively creating macro operations for specific code. This uses the following principles:

- Operational locality to the mapping process
- Reduction in complexity for the heuristics

This might create vastly different results when applied. In C/C++, macro operations are created with the help of `#pragma HLS inline off`. While in the OpenCL API, the same kind of macro operation can be generated by not specifying the following attribute when defining a function:

```c
_attribute__((always_inline))
```

For more information, see `pragma HLS inline`.

Using Optimized Libraries

The OpenCL specification provides many math built-in functions. All math built-in functions with the `native_` prefix are mapped to one or more native device instructions and will typically have better performance compared to the corresponding functions (without the `native_` prefix). The accuracy and in some cases the input ranges of these functions is implementation-defined. In the Vitis technology, these `native_` built-in functions use the equivalent functions in the Vitis HLS tool Math library, which are already optimized for Xilinx FPGAs in terms of area and performance.

RECOMMENDED: Xilinx recommends using `native_` built-in functions or the HLS tool Math library if the accuracy meets the application requirement.

Exploring Kernel Optimizations Using Vitis HLS

All kernel optimizations using OpenCL or C/C++ can be performed from within the Vitis core development kit. The primary performance optimizations, such as those discussed in this section (pipelining function and loops, applying dataflow to enable greater concurrency between functions and loops, unrolling loops, etc.), are performed by the Vitis HLS tool.
The Vitis core development kit automatically calls the HLS tool. However, to use the GUI analysis capabilities, you must launch the HLS tool directly from within the Vitis technology. Using the HLS tool in standalone mode, as discussed in Compiling Kernels with the Vitis HLS, enables the following enhancements to the optimization methodology:

- The ability to focus solely on the kernel optimization because there is no requirement to execute emulation.
- The skill to create multiple solutions, compare their results, and explore the solution space to find the most optimum design.
- The competence to use the interactive Analysis Perspective to analyze the design performance.

**IMPORTANT!** Only the kernel source code is incorporated back into the Vitis core development kit. After exploring the optimization space, ensure that all optimizations are applied to the kernel source code as OpenCL attributes or C/C++ pragmas.

To open the HLS tool in standalone mode, from the Assistant window, right-click the hardware function object, and select Open HLS Project, as shown in the following figure.

*Figure 65: Open HLS Project*
Topological Optimization

This section focuses on the topological optimization. It looks at the attributes related to the rough layout and implementation of multiple compute units and their impact on performance.

Multiple Compute Units

Depending on available resources on the target device, multiple compute units of the same kernel (or different kernels) can be created to run in parallel, which improves the system processing time and throughput. For more details, see Creating Multiple Instances of a Kernel.

Using Multiple DDR Banks

Acceleration cards supported in Vitis technology provide one, two, or four DDR banks, and up to 80 GB/s raw DDR bandwidth. For kernels moving large amount of data between the FPGA and the DDR, Xilinx® recommends that you direct the Vitis compiler and runtime library to use multiple DDR banks.

In addition to DDR banks, the host application can access PLRAM to transfer data directly to a kernel. This feature is enabled using the connectivity.sp option in a configuration file specified with the v++ --config option. Refer to Mapping Kernel Ports to Memory for more information on implementing this optimization and Memory Mapped Interfaces on data transfer to the global memory banks.

To take advantage of multiple DDR banks, you need to assign CL memory buffers to different banks in the host code as well as configure the xclbin file to match the bank assignment in v++ command line.

The following block diagram shows the Global Memory Two Banks (C) example in Vitis Examples on GitHub. This example connects the input pointer interface of the kernel to DDR bank 0, and the output pointer interface to DDR bank 1.

Figure 66: Global Memory Two Banks Example
Assigning DDR Bank in Host Code

IMPORTANT! This is optional and only needed in specific cases as described below.

During the Vitis tool flow, the kernel port to memory bank connectivity can be established using the --connectivity.sp switch as described in Mapping Kernel Ports to Memory. The xclbin generated by v++ contains the information about the kernel port to memory connectivity so that XRT can allocate buffers appropriately. When a buffer is created in the host code, XRT automatically assigns the buffer to memory from the kernel xclbin, and manages the buffers internally. If a single kernel port is connected to multiple memory banks, XRT always starts from the lower numbered bank.

In most cases, this approach is sufficient. However, in some specific cases you may need to manually assign the buffer location (or special property) in the host code. For this purpose, the Xilinx OpenCL vendor extension provides a buffer extension called CL_MEM_XRT_PTR_XILINX to specifically manage bank assignment in the host code. The following code example shows the required header file and code for assigning input and output buffers to DDR bank 0 and bank 1:

```c
#include <CL/cl_ext.h>
...
int main(int argc, char** argv)
{
    ...
    cl_mem_ext_ptr_t inExt, outExt;  // Declaring two extensions for both buffers
    inExt.flags = 0|XCL_MEM_TOPOLOGY; // Specify Bank0 Memory for input memory
    outExt.flags = 1|XCL_MEM_TOPOLOGY; // Specify Bank1 Memory for output Memory
    inExt.obj = 0  ; outExt.obj = 0; // Setting Obj and Param to Zero
    inExt.param = 0 ; outExt.param = 0;

    int err;
    //Allocate Buffer in Bank0 of Global Memory for Input Image using Xilinx Extension
    cl_mem buffer_inImage = clCreateBuffer(world.context, CL_MEM_READ_ONLY
                   | CL_MEM_EXT_PTR_XILINX,
                   image_size_bytes, &inExt, &err);
    if (err != CL_SUCCESS){
        std::cout << "Error: Failed to allocate device Memory" << std::endl;
        return EXIT_FAILURE;
    }
    //Allocate Buffer in Bank1 of Global Memory for Input Image using Xilinx Extension
    cl_mem buffer_outImage = clCreateBuffer(world.context,
                  CL_MEM_WRITE_ONLY | CL_MEM_EXT_PTR_XILINX,
                  image_size_bytes, &outExt, NULL);
    if (err != CL_SUCCESS){
        std::cout << "Error: Failed to allocate device Memory" << std::endl;
        return EXIT_FAILURE;
    }
    ...
}
```
The extension pointer `cl_mem_ext_ptr_t` is a `struct` as defined below:

```c
typedef struct{
    unsigned flags;
    void *obj;
    void *param;
} cl_mem_ext_ptr_t;
```

- **Valid values for `flags` are:**
  - `XCL_MEM_DDR_BANK0`
  - `XCL_MEM_DDR_BANK1`
  - `XCL_MEM_DDR_BANK2`
  - `XCL_MEM_DDR_BANK3`
  - `<id> | XCL_MEM_TOPOLOGY`

  **Note:** The `<id>` is determined by looking at the Memory Configuration section in the `xxx.xclbin.info` file generated next to the `xxx.xclbin` file. In the `xxx.xclbin.info` file, the global memory (DDR, HBM, PLRAM, etc.) is listed with an index representing the `<id>`.

- `obj` is the pointer to the associated host memory allocated for the CL memory buffer only if `CL_MEM_USE_HOST_PTR` flag is passed to `clCreateBuffer` API, otherwise set it to NULL.

- `param` is reserved for future use. Always assign it to 0 or NULL.

Here are some specific cases where you might want to use the extension pointer:

- **P2P Buffer:** For an explanation and example, refer to [https://xilinx.github.io/XRT/master/html/p2p.html](https://xilinx.github.io/XRT/master/html/p2p.html)

- **Host-Memory Buffer:** For an explanation and example, refer to [https://xilinx.github.io/XRT/master/html/sb.html](https://xilinx.github.io/XRT/master/html/sb.html)

- **Allocating the host buffer to a specific bank when the kernel port is connected to multiple banks:** For example, DDR[0:1]. This use case is described in detail in the Using Multiple DDR Banks lab of the Vitis Optimizing Accelerated FPGA Applications: Bloom Filter Example tutorial.

### Example of Allocating the Host Buffer to A Specific Bank

An example of the third case listed above, where you might need to use `cl_mem_ext_ptr_t`, is when the host and kernel are both accessing the DDR bank simultaneously, and you would like to split the data so that kernel and host access memory banks in a ping pong fashion. When the host is writing/reading to a specific memory bank, the kernel is writing/reading from another bank so that these host/kernel accesses don’t compete and impact performance. For this scenario, you must manage the buffer allocation yourself.
The kernel ports in the xclbin are connected to DDR bank1 and bank2, and reading the data from these banks alternatively. The connectivity is established during linking by the Vitis compiler using the --connectivity.sp switch:

```
[connectivity]
sp=runOnfpga_1.input_words:DDR[1:2]
```

From the host code, you can send the `input_words` data to DDR banks 1 and 2 alternatively. Two Xilinx extension pointer (cl_mem_ext_ptr_t) objects are created as shown in the example code below. The object flags will determine which DDR bank each buffer will be assigned to for the kernel to access. The kernel argument can be set to `input_words[0]` and `input_words[1]` for consecutive kernel enqueues.

```
#include <CL/cl_ext.h>
...
int main(int argc, char** argv)
{
    cl_mem_ext_ptr_t buffer_words_ext[2];

    buffer_words_ext[0].flags = 1 | XCL_MEM_TOPOLOGY; // DDR[1]
    buffer_words_ext[0].param = 0;
    buffer_words_ext[0].obj   = input_doc_words;

    buffer_words_ext[1].flags = 2 | XCL_MEM_TOPOLOGY; // DDR[2]
    buffer_words_ext[1].param = 0;
    buffer_words_ext[1].obj   = input_doc_words;
...
```

### Assigning Global Memory for Kernel Code

#### Creating Multiple AXI Interfaces

OpenCL kernels, C/C++ kernels, and RTL kernels have different methods for assigning function parameters to AXI interfaces.

- For OpenCL kernels, the `--max_memory_ports` option is required to generate one AXI4 interface for each global pointer on the kernel argument. The AXI4 interface name is based on the order of the global pointers on the argument list.

The following code is taken from the example gmem_2banks_ocl in the ocl_kernels category from the Vitis Accel Examples on GitHub:

```
__kernel __attribute__((reqd_work_group_size(1, 1, 1)))
void apply_watermark(__global const TYPE * __restrict input,
    __global TYPE * __restrict output, int width, int height) {
    ...
}
```

In this example, the first global pointer `input` is assigned an AXI4 name M_AXI_GMEM0, and the second global pointer `output` is assigned a name M_AXI_GMEM1.
• For C/C++ kernels, multiple AXI4 interfaces are generated by specifying different “bundle” names in the HLS INTERFACE pragma for different global pointers. Refer to Kernel Interfaces for more information.

The following is a code snippet from the `gmem_2banks` example that assigns the input pointer to the bundle `gmem0` and the output pointer to the bundle `gmem1`. The bundle name can be any valid C string, and the AXI4 interface name generated will be `M_AXI_<bundle_name>`. For this example, the input pointer will have AXI4 interface name as `M_AXI_gmem0`, and the output pointer will have `M_AXI_gmem1`. Refer to `pragma HLS interface` for more information.

```c
#pragma HLS INTERFACE m_axi port=input  offset=slave bundle=gmem0
#pragma HLS INTERFACE m_axi port=output offset=slave bundle=gmem1
```

• For RTL kernels, the port names are generated during the import process by the RTL kernel wizard. The default names proposed by the RTL kernel wizard are `m00_axi` and `m01_axi`. If not changed, these names have to be used when assigning a DDR bank through the `connectivity.sp` option in the configuration file. Refer to Mapping Kernel Ports to Memory for more information.

### Assigning AXI Interfaces to DDR Banks

**IMPORTANT!** When using more than one DDR interface, Xilinx requires you to specify the DDR memory bank for each kernel/CU, and specify the SLR to place the kernel into. For more information, see Mapping Kernel Ports to Memory and Assigning Compute Units to SLRs.

The following is an example configuration file that specifies the `connectivity.sp` option, and the `v++` command line that connects the input pointer `(M_AXI_GMEM0)` to DDR bank 0 and the output pointer `(M_AXI_GMEM1)` to DDR bank 1:

**The `config_sp.cfg` file:**

```ini
[connectivity]
sp=apply_watermark_1.m_axi_gmem0:DDR[0]
sp=apply_watermark_1.m_axi_gmem1:DDR[1]
```

**The `v++` command line:**

```
v++ apply_watermark --config config_sp.cfg
```

You can use the Device Hardware Transaction view to observe the actual DDR Bank communication, and to analyze DDR usage.
Assigning AXI Interfaces to PLRAM

Some platforms support PLRAMs. In these cases, use the same --connectivity.sp option as described in Assigning AXI Interfaces to DDR Banks, but use the name, PLRAM[id]. Valid names supported by specific platforms can be found in the Memory Configuration section of the xclibin.info file generated alongside xclbin.

Kernel SLR and DDR Memory Assignments

Kernel compute unit (CU) instance and DDR memory resource floorplanning are keys to meeting quality of results of your design in terms of frequency and resources. Floorplanning involves explicitly allocating CUs (a kernel instance) to SLRs and mapping CUs to DDR memory resources. When floorplanning, both CU resource usage and DDR memory bandwidth requirements need to be considered.

The largest Xilinx FPGAs are made up of multiple stacked silicon dies. Each stack is referred to as a super logic region (SLR) and has a fixed amount of resources and memory including DDR interfaces. Available device SLR resources which can be used for custom logic can be found in the Vitis Software Platform Release Notes, or can be displayed using the platforminfo utility described in platforminfo Utility.

You can use the actual kernel resource utilization values to help distribute CUs across SLRs to reduce congestion in any one SLR. The system estimate report lists the number of resources (LUTs, Flip-Flops, BRAMs, etc.) used by the kernels early in the design cycle. The report can be generated during hardware emulation and system compilation through the command line or GUI and is described in System Estimate Report.
Use this information along with the available SLR resources to help assign CUs to SLRs such that no one SLR is over-utilized. The less congestion in an SLR, the better the tools can map the design to the FPGA resources and meet your performance target. For mapping memory resources and CUs, see Mapping Kernel Ports to Memory and Assigning Compute Units to SLRs.

**Note:** While compute units can be connected to any available DDR memory resource, it is also necessary to account for the bandwidth requirements of the kernels when assigning to SLRs.

After allocating your CUs to SLRs, map any CU master AXI port(s) to DDR memory resources. Xilinx recommends connecting to a DDR memory resource in the same SLR as the CU. This reduces competition for the limited SLR-crossing connection resources. In addition, connections between SLRs use super long line (SLL) routing resources, which incurs a greater delay than a standard intra-SLR routing.

It might be necessary to cross an SLR region to connect to a DDR resource in a different SLR. However, if both the `connectivity.sp` and the `connectivity.slr` directives are explicitly defined, the tools automatically add additional crossing logic to minimize the effect of the SLL delay, and facilitates better timing closure.

**Guidelines for Kernels that Access Multiple Memory Banks**

The DDR memory resources are distributed across the super logic regions (SLRs) of the platform. Because the number of connections available for crossing between SLRs is limited, the general guidance is to place a kernel in the same SLR as the DDR memory resource with which it has the most connections. This reduces competition for SLR-crossing connections and avoids consuming extra logic resources associated with SLR crossing.
Figure 68: Kernel and Memory in Same SLR

Note: The image on the left shows a single AXI interface mapped to a single memory bank. The image on the right shows multiple AXI interfaces mapped to the same memory bank.

As shown in the previous figure, when a kernel has a single AXI interface that maps only a single memory bank, the platforminfo utility described in platforminfo Utility lists the SLR that is associated with the memory bank of the kernel; therefore, the SLR where the kernel would be best placed. In this scenario, the design tools might automatically place the kernel in that SLR without need for extra input; however, you might need to provide an explicit SLR assignment for some of the kernels under the following conditions:

- If the design contains a large number of kernels accessing the same memory bank.
- A kernel requires some specialized logic resources that are not available in the SLR of the memory bank.

When a kernel has multiple AXI interfaces and all of the interfaces of the kernel access the same memory bank, it can be treated in a very similar way to the kernel with a single AXI interface, and the kernel should reside in the same SLR as the memory bank that its AXI interfaces are mapping.
**Figure 69: Memory Bank in Adjoining SLR**

*Note:* The image on the left shows one SLR crossing is required when the kernel is placed in SLR0. The image on the right shows two SLR crossings are required for kernel to access memory banks.

When a kernel has multiple AXI interfaces to multiple memory banks in different SLRs, the recommendation is to place the kernel in the SLR that has the majority of the memory banks accessed by the kernel (shown it the figure above). This minimizes the number of SLR crossings required by this kernel which leaves more SLR crossing resources available for other kernels in your design to reach your memory banks.

When the kernel is mapping memory banks from different SLRs, explicitly specify the SLR assignment as described in *Kernel SLR and DDR Memory Assignments.*
Figure 70: Memory Banks Two SLRs Away

Note: The image on the left shows two SLR crossings are required to access all of the mapped memory banks. The image on the right shows three SLR crossings are required to access all of the mapped memory banks.

As shown in the previous figure, when a platform contains more than two SLRs, it is possible that the kernel might map a memory bank that is not in the immediately adjacent SLR to its most commonly mapped memory bank. When this scenario arises, memory accesses to the distant memory bank must cross more than one SLR boundary and incur additional SLR-crossing resource costs. To avoid such costs it might be better to place the kernel in an intermediate SLR where it only requires less expensive crossings into the adjacent SLRs.
Chapter 21

Debugging Applications and Kernels

The Vitis™ unified software platform provides application-level debug features and techniques that allow the host code, kernel code, and the interactions between them to be debugged. These features and techniques are split between software debugging and hardware debugging flows.

For software debugging, the host and kernel code can be debugged using the Vitis IDE, or using GDB from the command line as a standard debug tool.

For hardware debugging, kernels running on hardware can be debugged using Xilinx® virtual cable (XVC) running over the PCIe® bus, for Alveo™ Data Center accelerator cards, and debugged using USB-JTAG cables for both Alveo cards and embedded processor platforms.

Debugging Flows

The Vitis unified software platform provides application-level debug features which allow the host code, the kernel code, and the interactions between them to be efficiently debugged in either the Vitis IDE, or from the command line. The recommended debugging flow consists of three levels of debugging:

- **Debugging in Software Emulation** to confirm the algorithm functionality of the application as represented in both your host program and kernel code.
- **Debugging in Hardware Emulation** to compile the kernel into RTL, confirm the behavior of the generated logic, and evaluate the simulated performance of the hardware.
- **Debugging During Hardware Execution** to implement the FPGA binary and debug the application running in hardware.

This three-tiered approach allows debugging the host and kernel code, and the interactions between them at different levels of abstraction. Each provides specific insights into the design and makes debugging easier. All flows are supported through an integrated GUI flow as well as through a batch flow using basic compile time and runtime setup options.

In the case of applications running on embedded processor platforms, some additional setup is required as described in *Debugging on Embedded Processor Platforms*. 
Debugging in Software Emulation

**IMPORTANT!** The following steps describe debugging from the command line. However, the Vitis IDE offers a standalone debug environment for use with the Vitis application acceleration projects created from the command line. Refer to Using the Standalone Debug Flow for more information.

The Vitis unified software platform supports typical software debugging for the host code at all times, the kernel code when running in software emulation mode, and at points during hardware emulation mode. This is a standard software debug flow using breakpoints, stepping through code, analyzing variables, and forcing the code into specific states.

The following figure shows the debug flow during software emulation for the host and kernel code (written in C/C++ or OpenCL™) using the GNU debugging (GDB) tool. Notice the two instances of GDB to separately debug the host and kernel processes, and the use of the debug server (`xrt_server`).

![Software Emulation Diagram](image)

**Figure 71: Software Emulation**

Xilinx recommends iterating the design as much as possible in Software Emulation, which takes little compile time and executes quickly. For more detailed information on software emulation, see Software Emulation.
GDB-Based Debugging

**IMPORTANT!** Both the host and kernel code must be compiled for debugging using the `-g` option.

For the GNU debugging (GDB), you can debug the kernel or host code, adding breakpoints, and inspecting variables. This familiar software debug flow allows quick design, compile, and debug to validate the functionality of your application. The Vitis debugger also provides extensions to GDB to let you examine the content of the Xilinx Runtime (XRT) library from the host program. These extensions can be used to debug protocol synchronization issues between the host and the kernel.

The Vitis core development kit supports GDB host program debugging in all flows, but kernel debugging is limited to software and hardware emulation modes. Debugging features need to be enabled in your host and kernel code by using the `-g` option during compilation and linking.

This section shows how host and kernel debugging can be performed with the help of GDB. Because this flow should be familiar to most software developers, this section focuses on the extensions of host code debugging capabilities for the XRT library and the requirements of kernel debug.

**Xilinx Runtime Library GDB Extensions**

The Vitis debugger (**xgdb**) enables new GDB commands that give you visibility from the host application into the XRT library.

**Note:** If you launch GDB outside of the Vitis debugger, the command extensions need to be enabled using the `appdebug.py` script as described in Launching Host and Kernel Debug.

There are two kinds of commands which can be called from the `gdb` command line:

1. **xprint** commands that give visibility into XRT library data structures (**cl_command_queue**, **cl_event**, and **cl_mem**). These commands are explained below.
2. **xstatus** commands that give visibility into IP running on the Vitis target platform when debugging during hardware execution.

You can get more information about the `xprint` and `xstatus` commands by using the `help <command>` from the `gdb` command prompt.

A typical application for these commands is when you see the host application hang. In this case, the host application could be waiting for the command queue to finish, or waiting on an event list. Printing the command queue using the `xprint queue` command can tell you what events are unfinished, allowing you to analyze dependencies between events.
The output of both of these commands is automatically tracked when debugging with the Vitis IDE. In this case, three tabs are provided next to the common tabs for Variables, Breakpoints, and Registers in the upper left corner of the debug perspective. These are labeled Command Queue, Memory Buffers, and Platform Debug, showing the output of `xprint queue`, `xprint mem`, and `xstatus`, respectively.

### xprint Commands

The arguments to `xprint queue` and `xprint mem` are optional. The application debug environment keeps track of all the XRT library objects and automatically prints all valid queues and `cl_mem` objects if the argument is not specified. In addition, the commands do a proper validation of supplied command queue, event, and `cl_mem` arguments.

```plaintext
xprint queue [<cl_command_queue>]
xprint event <cl_event>
xprint mem [<cl_mem>]
xprint kernel
xprint all
```

### xstatus Commands

This functionality is only available in the system flow (hardware execution) and not in any of the emulation flows.

```plaintext
xstatus all
xstatus --<ipname>
```

### GDB Kernel-Based Debugging

GDB kernel debugging is supported for the software emulation and hardware emulation flows. When the GDB executable is connected to the kernel in the IDE or command line flows, you can set breakpoints and query the content of variables in the kernel, similar to normal host code debugging. This is fully supported in the software emulation flow because the kernel GDB processes attach to the spawned software processes.

However, during hardware emulation, the kernel source code is transformed into RTL, created by Vitis HLS, and executed. As the RTL model is simulated, all transformations for performance optimization and concurrent hardware execution are applied. For that reason, not all C/C++/OpenCL lines can be uniquely mapped to the RTL code, and only limited breakpoints are supported and at only specific variables can be queried. Today, the GDB tool therefore breaks on the next possible line based on requested breakpoint statements and clearly states if variables can not be queried based on the RTL transformations.

### Command Line Debug Flow

TIP: Set up the command shell or window as described in Setting Up the Vitis Environment prior to running the tools.
The following describes the steps required to run the debug flow in software emulation from the command line. Refer to Section VII: Using the Vitis IDE for information on debugging in the IDE. Debugging in the Vitis core development kit uses the following steps:

1. Compiling and linking the host code for debugging by adding the -g option to the g++ command line as described in Building the Host Program.

2. Compiling and linking the kernel code for debugging by adding the -g option to the v++ command line as described in Building the Device Binary.
   
   Note: When debugging OpenCL kernels, there are additional steps that you can take during compiling and linking as described in Debugging OpenCL Kernels.

3. Launching GDB to debug the application. This process involves three command target platforms as described in Launching Host and Kernel Debug.

**Debugging OpenCL Kernels**

For OpenCL kernels, additional runtime checks can be performed during software emulation. These additional checks include:

- Checking whether an OpenCL kernel makes out-of-bounds accesses to the interface buffers (fsanitize=address).
- Checking whether the kernel makes accesses to uninitialized local memory (fsanitize=memory).

These are Vitis compiler options that are enabled through the --advanced compiler option as described in --advanced Options, using the following command syntax:

```
--advanced.param compiler.fsanitize=address,memory
```

When applied, the emulation run produces a debug log with emulation diagnostic messages that are written to `<project_dir>/Emulation-SW/<proj_name>-Default>/emulation_debug.log`.

The fsanitize directive can also be specified in a config file, as follows:

```ini
[advanced]
#param=<param_type>:<param_name>.<value>
param=compiler.fsanitize=address,memory
```

Then the config file is specified on the v++ command line:

```
v++ -l -t sw_emu --config ./advanced.cfg -o bin_kernel.xclbin
```

Refer to the Vitis Compiler Configuration File for more information on the --config option.
Launching Host and Kernel Debug

In software emulation, to better model the hardware accelerator, the execution of the FPGA binary is spawned as a separate process. If you are using GDB to debug the host code, breakpoints set in kernel code are not encountered because the kernel code is not run within the host code process. To support the concurrent debugging of the host and kernel code, the Vitis debugger provides a system to attach to spawned kernels through the use of the debug server (xrt_server). To connect the host and kernel code to the debug server, you must open three terminal windows using the following process.

**TIP:** This flow should also work while using a graphical front-end for GDB, such as the data display debugger (DDD) available from GNU. The following steps are the instructions for launching GDB.

1. Open three terminal windows, and set up each window as described in Setting Up the Vitis Environment. The three windows are for:
   - Running xrt_server
   - Running GDB (xgdb) on the Host Code
   - Running GDB (xgdb) on the Kernel Code

2. In the first terminal, after setting up the terminal environment, start the Vitis debug server using the following command:

   ```
xrt_server --sdx-url
   ```

   The debug server listens for debug commands from the host and kernel, connecting the two processes to create a single debug environment. The `xrt_server` returns a listener port `<num>` on standard out. Keep track of the listener port number returned as this port is used by GDB to debug the kernel process. To control this process, you must start new GDB instances and connect to the xrt_server. You will do this in the next steps.

   **IMPORTANT!** With the xrt_server running, all spawned GDB processes wait for control from you. If no GDB ever attaches to the xrt_server, or provides commands, the kernel code appears to hang.

3. In a second terminal, after setting up the terminal environment, launch GDB for the host code as described in the following steps:
   a. Set the `ENABLE_KERNEL_DEBUG` environment variable. For example, in a C-shell use the following:

   ```
   setenv ENABLE_KERNEL_DEBUG true
   ```

   b. Set the `XCL_EMULATION_MODE` environment variable to `sw_emu` mode as described in Running the Application Hardware Build. For example, in a C-shell use the following:

   ```
   setenv XCL_EMULATION_MODE sw_emu
   ```
c. The runtime debug feature must be enabled using an entry in the xrt.ini file, as described in xrt.ini File. Create an xrt.ini file in the same directory as your host executable, and include the following lines:

```ini
[Debug]
app_debug=true
```

This informs the runtime library that the kernel has been compiled for debug, and that XRT library should enable debug features.

d. Start gdb through the Xilinx wrapper:

```bash
xgdb --args <host> <xclbin>
```

Where `<host>` is the name of your host executable, and `<xclbin>` is the name of the FPGA binary. For example:

```bash
xgdb --args host.exe vadd.xclbin
```

Launching GDB from the xgdb wrapper performs the following setup steps for the Vitis debugger:

- Loads GDB with the specified host program.
- Sources the Python script from the GDB command prompt to enable the Vitis debugger extensions:

```bash
gdb> source ${XILINX_XRT}/share/appdebug/appdebug.py
```

4. In a third terminal, after setting up the terminal environment, launch the xgdb command, and run the following commands from the (gdb) prompt:

- For software emulation:

  ```bash
  file <Vitis_path>/data/emulation/unified/cpu_em/generic_pcie/model/genericpciemodel
  ```

  Where `<Vitis_path>` is the installation path of the Vitis core development kit. Using the `$XILINX_VITIS` environment variable will not work inside GDB.

- For hardware emulation:

  1. Locate the xrt_server temporary directory: `/tmp/sdx/$uid`.
  2. Find the xrt_server process id (PID) containing the DWARF file of this debug session.
  3. At the gdb command line, run: `file /tmp/sdx/$uid/$pid/NUM.DWARF`.

- In either case, connect to the kernel process:

  ```bash
  target remote :<num>
  ```

  Where `<num>` is the listener port number returned by the xrt_server.
**TIP:** When debugging software/hardware emulation kernels in the Vitis IDE, these steps are handled automatically and the kernel process is automatically attached, providing multiple contexts to debug both the host code and kernel code simultaneously.

With the three terminal windows running the `xrt_server`, GDB for the host, and GDB for the kernels, you can set breakpoints on your host or kernels as needed, run the `continue` command, and debug your application. When the all kernel invocations have finished, the host code continues and the `xrt_server` connection drops.

**IMPORTANT!** For both software and hardware emulation flows, there are restrictions with respect to the accelerated kernel code debug interactions. Because this code is preprocessed in the software emulation flow, and translated to RTL in the hardware emulation flow, it is not always possible to set breakpoints at all locations. Only a limited number of breakpoints such as preserved loops and functions are supported, especially for hardware emulation. Nevertheless, this setup is useful for debugging the interface of the host code with the kernels.

### Using `printf()` or `cout` to Debug Kernels

The basic approach to debugging algorithms is to verify key code steps and key data values throughout the execution of the program. For application developers, printing checkpoint statements, and outputting current values in the code is a simple and effective method of identifying issues within the execution of a program. This can be done using the `printf()` function, or `cout` for standard output.

#### C/C++ Kernel

For C/C++ kernel models, `printf()` is only supported during software emulation and should be excluded from the Vitis HLS synthesis step. In this case, any `printf()` statement should be surrounded by the following compiler macros:

```c
#ifdef __SYNTHESIS__
    printf("Checkpoint 1 reached");
#endif
```

For C++ kernels, you can also use `cout` in your code to add checkpoints or messages used for debugging the code. For example, you might add the following:

```cpp
std::cout << "TEST " << (match ? 'PASSED' : 'FAILED') << std::endl;
```

#### OpenCL Kernel

The Xilinx Runtime (XRT) library supports the OpenCL™ `printf()` built-in function within kernels in all build configurations: software emulation, hardware emulation, and during hardware execution.
TIP: The printf() function is only supported in all build configurations for OpenCL kernels. For C/C++ kernels, printf() is only supported in software emulation.

The following is an example of using printf() in the kernel, and the output when the kernel is executed with global size of 8:

```c
__kernel __attribute__((reqd_work_group_size(1, 1, 1)))
void hello_world(__global int *a)
{
    int idx = get_global_id(0);
    printf("Hello world from work item %d\n", idx);
    a[idx] = idx;
}
```

The output is as follows:

```
Hello world from work item 0
Hello world from work item 1
Hello world from work item 2
Hello world from work item 3
Hello world from work item 4
Hello world from work item 5
Hello world from work item 6
Hello world from work item 7
```

IMPORTANT! printf() messages are buffered in the global memory and unloaded when kernel execution is completed. If printf() is used in multiple kernels, the order of the messages from each kernel display on the host terminal is not certain. Note, especially when running in hardware emulation and hardware, the hardware buffer size might limit printf output capturing.

Debugging in Hardware Emulation

IMPORTANT! The following steps describe debugging from the command line. However, the Vitis IDE offers a standalone debug environment for use with the Vitis application acceleration projects created from the command line. Refer to Using the Standalone Debug Flow for more information.

During hardware emulation, kernel code is compiled into RTL code so that you can evaluate the RTL logic of kernels prior to implementation into the Xilinx device. The host code can be executed concurrently with a behavioral simulation of the RTL model of the kernel, directly imported, or created through Vitis HLS from the C/C++/OpenCL kernel code. For more information, see Hardware Emulation.

The following figure shows the hardware emulation flow diagram which can be used in the Vitis debugger to validate the host code, profile host and kernel performance, give estimated FPGA resource usage, and verify the kernel using an accurate model of the hardware (RTL). GDB can also be used for more traditional software-style debugging of the host and kernel code.
Verify the host code and the kernel hardware implementation is correct by running hardware emulation on a data set. The hardware emulation flow invokes the Vivado logic simulator in the Vitis core development kit to test the kernel logic that is to be executed on the FPGA fabric. The interface between the models is represented by a transaction-level model (TLM) to limit impact of interface model on the overall execution time. The execution time for hardware emulation is longer than software emulation.

**TIP:** Xilinx recommends that you use small data sets for debug and validation.

During hardware emulation, you can optionally modify the kernel code to improve performance. Iterate your host and kernel code design in hardware emulation until the functionality is correct, and the estimated kernel performance is satisfactory.

## GDB-Based Debugging in Hardware Emulation

Debugging using a software-based GDB flow is fully supported during hardware emulation. Because the Vitis debugger maps the RTL code back to the original C/C++ source code, there is no difference in using GDB for debugging in hardware emulation. However, mapping the code limits the breakpoints and observability of the variables in some cases, because during the RTL generation by Vitis HLS, some variables and loops from the kernel source might have been dissolved, or optimized away.
Using GDB for debugging the host and kernel code in hardware emulation uses the same three-terminal process described for software emulation. Refer to the instructions in Command Line Debug Flow for details of running this flow.

Waveform-Based Kernel Debugging

Because the C/C++ and OpenCL kernel code is synthesized into RTL code using Vitis HLS in the hardware emulation build configuration, you can also use RTL behavioral simulation to analyze the kernel logic. Hardware designers are likely to be familiar with this approach. This waveform-based HDL debugging is supported by the Vitis core development kit using both the command line flow, or through the IDE flow during hardware emulation.

**TIP:** Waveform-based debugging is considered an advanced feature. In most cases, the RTL Logic does not need to be analyzed.

Enable Waveform Debugging with the Vitis Compiler Command

The waveform debugging process can be enabled through the `v++` command using the following steps:

1. Enable debug features in the kernel code during compilation and linking, as described in Building the Device Binary.

   ```
   v++ -g ...
   ```

2. Create an `xrt.ini` file in the same directory as the host executable, as described in `xrt.ini` File, with the following contents:

   ```
   [Emulation]
   debug_mode=batch

   [Debug]
   profile=true
   timeline_trace=true
   data_transfer_trace=fine
   ```

3. Run the application, host and kernel, in hardware emulation mode. The waveform database, reflecting the hardware transaction data, is collected in a file named `<hardware_platform>-<device_id>-<xclbin_name>.wdb`. This file can directly be opened in the Vitis analyzer as described in Section VI: Using the Vitis Analyzer.

   **TIP:** If `debug_mode=gui` in the `xrt.ini`, a live waveform viewer is launched when the application is run, as described in Waveform View and Live Waveform Viewer. This is especially useful when debugging a `hw_emu` hang issue, because you can interrupt the simulation process in the simulator and observe the waveform up to that time.
Run the Waveform-Based Kernel Debugging Flow

The Vitis IDE provides waveform-based HDL debugging in the hardware emulation mode. The waveform is opened in the Vivado waveform viewer which should be familiar to Vivado logic simulation users. The Vitis IDE lets you display kernel interfaces, internal signals, and includes debug controls such as restart, HDL breakpoints, as well as HDL code lookup and waveform markers. In addition, it provides top-level DDR data transfers (per bank) along with kernel-specific details including compute unit stalls, loop pipeline activity, and data transfers.

For details, see Waveform View and Live Waveform Viewer.

If the live waveform viewer is activated, the waveform viewer automatically opens when running the executable. By default, the waveform viewer shows all interface signals and the following debug hierarchy:

Figure 73: Waveform Viewer

- **Memory Data Transfers**: Shows data transfers from all compute units funnel through these interfaces.
TIP: These interfaces could be a different bit width from the compute units. If so, then the burst lengths would be different. For example, a burst of sixteen 32-bit words at a compute unit would be a burst of one 512-bit word at the OCL master.

• **Kernel <kernel name><workgroup size> Compute Unit<CU name>:** Kernel name, workgroup size, and compute unit name.

• **CU Stalls (%):** This shows a summary of stalls for the entire CU. A bus of all lowest-level stall signals is created, and the bus is represented in the waveform as a percentage (%) of those signals that are active at any point in time.

• **Data Transfers:** This shows the data transfers for all AXI masters on the CU.

• **User Functions:** This lists all of the functions within the hierarchy of the CU.

• **Function: <function name>:** This is the function name.

• **Dataflow/Pipeline Activity:** This shows the function-level loop dataflow/pipeline signals for a CU.

• **Function Stalls:** This lists the three stall signals within this function.

• **Function I/O:** This lists the I/O for the function. These I/O are of protocol -m_axi, ap_fifo, ap_memory, or ap_none.

TIP: As with any waveform debugger, additional debug data of internal signals can be added by selecting the instance of interest from the scope menu and the signals of interest from the object menu. Similarly, debug controls such as HDL breakpoints, as well as HDL code lookup and waveform markers are supported. Refer to the Vivado Design Suite User Guide: Logic Simulation (UG900) for more information on working with the waveform viewer.

### Debug Techniques for Hardware Emulation

Due to the approximate models used in hardware emulation, the behavior of an emulated system might not match the hardware. The following list provides some common issues to examine if your application does not give expected results during hardware emulation:

1. Review the host application to ensure that the event dependency between different kernel runs is correctly captured. Such issues can lead to unpredictable behavior. It is also possible that the application can pass in hardware, but there could be a logical bug in your application which can be triggered on hardware under slightly different conditions.

2. If you have an RTL kernel, run the application in debug mode and ensure that no "X" (undriven values) in simulation in the kernel. This indicates incorrect code which can work in hardware but will fail in simulation with unpredictable behavior. If it is an HLS-generated kernel, confirm that all the variables are initialized to appropriate values.
3. Ensure that the amount of data being processed by kernels in hardware emulation is small so that emulation can finish in a reasonable time. Otherwise, it can appear that the application is running forever or has "hung". In this case, when running the application in hardware emulation look for INFO: [Vitis-EM 22] messages in the host application console. Check that the amount of data being read/written to or from global memory is increasing:
   a. If the RD/WR data is increasing, this indicates that application and hardware execution is progressing. The application is not hung, but is taking a really long time to complete. This could be due to large data size or due to kernels performing memory read/write in an inefficient manner. The application and kernel needs to be optimized.
   b. If the RD/WR data is not increasing in successive messages, this indicates that simulation is running but there is a deadlock in the hardware somewhere — either in the kernel or rest of the platform. Review the AXI transactions at the boundary of kernel, interconnect (for example, sdx_memss), and other places to check if there is an incomplete transaction or whether any transaction is being generated by the kernel.

4. Run hardware emulation in waveform mode and also review at the timeline trace. Check whether the kernel is getting "started" and "done" by observing the traffic on its AXI4-Lite interface, or by observing the output interrupt from the kernel.

5. Review the [Emulation] section of the xrt.ini File to enable applicable settings that can help to narrow down the issue in your application or kernel.

---

### Debugging During Hardware Execution

**IMPORTANT!** The following steps describe debugging from the command line. However, the Vitis IDE offers a standalone debug environment for use with the Vitis application acceleration projects created from the command line. Refer to Using the Standalone Debug Flow for more information.

During hardware execution, the actual hardware platform is used to execute the kernels, and you can evaluate the performance of the host program and accelerated kernels just by running the application. However, debugging the hardware build requires additional logic to be incorporated into the application. This will impact both the FPGA resources consumed by the kernel and the performance of the kernel running in hardware. The debug configuration of the hardware build includes special ChipScope debug cores, such as Integrated Logic Analyzer (ILA) and Virtual Input/Output (VIO) cores, and AXI performance monitors for debug purposes.

**TIP:** The additional logic required for debugging the hardware should be removed from the final production build.

The following figure shows the debug process for the hardware build, including debugging the host code using GDB, and using the Vivado hardware manager, with waveform analysis, kernel activity reports, and memory access analysis to identify and localize hardware issues.
With the system hardware build configured for debugging, the host program running on the CPU and the Vitis accelerated kernels running on the Xilinx device can be confirmed to be executing correctly on the actual hardware of the target platform. Some of the conditions that can be identified and analyzed include the following:

- **System hangs caused by protocol violations:**
  - These violations can take down the entire system.
  - These violations can cause the kernel to get invalid data or to hang.
  - It is hard to determine where or when these violations originated.
  - To debug this condition, you should use an ILA triggered off of the AXI protocol checker, which needs to be configured on the Vitis target platform.

- **Problems with the hardware kernel:**
  - Problems sometimes caused by the implementation: timing issues, race conditions, and bad design constraints.
  - Functional bugs that hardware emulation does not reveal.

- **Performance issues:**
  - For example, the frames per second processing is not what you expect.
You can examine data beats and pipelining.
Using an ILA with trigger sequencer, you can examine the burst size, pipelining, and data width to locate the bottleneck.

**Enabling Kernels for Debugging with Chipscope**

**System ILA**

The key to hardware debugging lies in instrumenting the kernels with the required debug logic. The following topic discusses the `v++` linker options that can be used to list the available kernel ports, enable the System Integrated Logic Analyzer (ILA) core on selected ports, and enable the AXI Protocol Checker debug core for checking for protocol violations.

The ILA core provides transaction-level visibility into an instance of a compute unit (CU) running on hardware. AXI traffic of interest can also be captured and viewed using the ILA core. The ILA provides custom event triggering on one or more signals to allow waveform capture at system speeds. The waveforms can be analyzed in a viewer and used to debug hardware, finding protocol violations, or performance issues. It can also be crucial for debugging difficult situation like application hangs.

Captured data can be accessed through the Xilinx virtual cable (XVC) using the Vivado tools. See the *Vivado Design Suite User Guide: Programming and Debugging (UG908)* for complete details.

The ILA core can be added to an existing RTL kernel to enable debugging features within that design, or it can be inserted automatically by the `v++` compiler during the linking stage. The `v++` command provides the `--debug` option as described in --debug Options to attach System ILA cores at the interfaces to the kernels for debugging and performance monitoring purposes.

**Note:** ILA debug cores require system resources, including logic and local memory to capture and store the signal data. Therefore they provide excellent visibility into your kernel, but they can affect both performance and resource utilization.

The `--debug` option to enable ILA IP core insertion has the following syntax:

```
--debug.chipscope <cu_name>[::<interface_name>]
```

In general, the `<interface_name>` is optional. If not specified, all ports are expected to be analyzed.

**AXI Protocol Checker**

The AXI Protocol Checker core monitors AXI interfaces. When attached to an interface, it actively checks for protocol violations and provides an indication of which violation occurred. You can assign it for all CUs in the design, or for specific CUs and ports.
The `--debug` option to enable AXI Protocol Checker insertion has the following syntax:

```
--debug.protocol all
```

The protocol checker can be specified with the keyword `all`, or the `<cu_name>:<interface_name>`.

**Note:** The `--debug.list_ports` option can be specified to return the actual names of ports on the kernel to use with `protocol` or `chipscope`.

An example flow you could use for adding ILA or protocol checkers to your design is outlined below:

1. Compile the kernel source files into an XO file, using the `-g` option to instrument the kernel for debug features:

```
v++ -c -g -k <kernel_name> --platform <platform> -o <kernel_xo_file>.xo <kernel_source_files>
```

2. After the kernel has been compiled into an XO file, use `--debug.list_ports` to cause the `v++` compiler to print the list of valid compute units and port combinations for the kernel:

```
v++ -l -g --platform <platform> --connectivity.nk <kernel_name>:<compute_units>:<kernel_nameN> --debug.list_ports <kernel_xo_file>.xo
```

3. Add the ILA or AXI debug cores on the desired ports by replacing `list_ports` with the appropriate `--debug.chipscope` or `--debug.protocol` command syntax:

```
v++ -l -g --platform <platform> --connectivity.nk <kernel_name>:<compute_units>:<kernel_nameN> --debug.chipscope <compute_unit_name>:<interface_name> <kernel_xo_file>.xo
```

**TIP:** The `--debug` option can be specified multiple times in a single `v++` command line, or configuration file to specify multiple CUs and interfaces.

When the design is built, you can debug the design using the Vivado hardware manager as described in Debugging with ChipScope.

### Adding Debug IP to RTL Kernels

**IMPORTANT! This debug technique requires familiarity with the Vivado Design Suite, and RTL design.**

You can also enable debugging in RTL kernels by manually adding ChipScope debug cores like the ILA and VIO in your RTL kernel code before packaging it for use in the Vitis development flow. From within the Vivado Design Suite, edit the RTL kernel code to manually instantiate an ILA debug core, or VIO IP from the Xilinx IP catalog, similar to using any other IP in Vivado IDE. Refer to the HDL Instantiation flow in the *Vivado Design Suite User Guide: Programming and Debugging (UG908)* to learn more about adding debug cores to your design.
The best time to add debug cores to your RTL kernel is when you create it. However, debug cores consume device resources and can affect performance, so it is good practice to make one kernel for debug and a second kernel for production use. The `rtl_vadd_hw_debug` of the RTL Kernels examples on GitHub shows an ILA debug core instantiated into the RTL kernel source file. The ILA monitors the output of the combinatorial adder as specified in the `src/hdl/krnl_vadd_rtl_int.sv` file.

```verilog
// ILA monitoring combinatorial adder
ila_0 ila_0 ( // input wire clk
    .clk(ap_clk),
    .probe0(areset), // input wire [0:0] probe0
    .probe1(rd_fifo_tvalid_n), // input wire [0:0] probe1
    .probe2(rd_fifo_tready), // input wire [0:0] probe2
    .probe3(rd_fifo_tdata), // input wire [63:0] probe3
    .probe4(adder_tvalid), // input wire [0:0] probe4
    .probe5(adder_tready_n), // input wire [0:0] probe5
    .probe6(adder_tdata) // input wire [31:0] probe6
);```

You can also add the ILA debug core using a Tcl script from within an open Vivado project, using the Netlist Insertion flow described in Vivado Design Suite User Guide: Programming and Debugging (UG908), as shown in the following Tcl script example:

```tcl
create_ip -name ila -vendor xilinx.com -library ip -version 6.2 -module_name ila_0
set_property -dict [list CONFIG.C_PROBE6_WIDTH {32} CONFIG.C_PROBE3_WIDTH {64} CONFIG.C_NUM_OF_PROBES {7} CONFIG.C_EN_STRG_QUAL {1} CONFIG.C_INPUT_PIPE_STAGES {2} CONFIG.C_ADV_TRIGGER {true} CONFIG.ALL_PROBE_SAME_MU_CNT {4} CONFIG.C_PROBE6_MU_CNT {4} CONFIG.C_PROBE5_MU_CNT {4} CONFIG.C_PROBE4_MU_CNT {4} CONFIG.C_PROBE3_MU_CNT {4} CONFIG.C_PROBE2_MU_CNT {4} CONFIG.C_PROBE1_MU_CNT {4} CONFIG.C_PROBE0_MU_CNT {4}] [get_ips ila_0]
```

After the RTL kernel has been instrumented for debug with the appropriate debug cores, you can analyze the hardware in the Vivado hardware manager as described in Debugging with ChipScope.

### Enabling ILA Triggers for Hardware Debug

To perform hardware debug of both the host program and the kernel code running on the target platform, the application host code must be modified to let you set up the ILA trigger conditions after the kernel has been programmed into the device, but before starting the kernel.
Adding ILA Triggers Before Starting Kernels

Pausing the host program can be accomplished through the use of a pause, or wait step in the code, such as the \texttt{wait} for \texttt{enter} function used in the RTL Kernel example on GitHub. The function is defined in the \texttt{src/host.cpp} code as follows:

\begin{Verbatim}
void wait_for_enter(const std::string &msg) {
    std::cout << msg << std::endl;
    std::cin.ignore(std::numeric_limits<std::streamsize>::max(), 'n');
}
\end{Verbatim}

The \texttt{wait} for \texttt{enter} function is used in the \texttt{main} function as follows:

\begin{Verbatim}
....
    std::string binaryFile = xcl::find_binary_file(device_name,'vadd');
    cl::Program::Binaries bins = xcl::import_binary_file(binaryFile);
    devices.resize(1);
    cl::Program program(context, devices, bins);
    cl::Kernel krnl_vadd(program,"krnl_vadd_rtl");

    wait_for_enter("\nPress ENTER to continue after setting up ILA trigger...");

    //Allocate Buffer in Global Memory
    std::vector<cl::Memory> inBufVec, outBufVec;
    cl::Buffer buffer_r1(context,CL_MEM_USE_HOST_PTR | CL_MEM_READ_ONLY,
        vector_size_bytes, source_input1.data());
    ...

    //Copy input data to device global memory
    q.enqueueMigrateMemObjects(inBufVec,0/* 0 means from host*/);

    //Set the Kernel Arguments
    ...

    //Launch the Kernel
    q.enqueueTask(krnl_vadd);
\end{Verbatim}

The use of the \texttt{wait} for \texttt{enter} function pauses the host program to give you time to set up the required ILA triggers and prepare to capture data from the kernel. After the Vivado hardware manager is set up and configured, press \texttt{Enter} to continue running the application.

- For C++ host code, add a pause after the creation of the \texttt{cl::Kernel} object, as shown in the example above.
- For C-language host code, add a pause after the \texttt{clCreateKernel()} function call:
Pausing the Host Application Using GDB

If you are running GDB to debug the host program at the same time as performing hardware debug on the kernels, you can also pause the host program as needed by inserting a breakpoint at the appropriate line of code. Instead of making changes to the host program to pause the application as needed, you can set a breakpoint prior to the kernel execution in the host code. When the breakpoint is reached, you can set up the debug ILA triggers in Vivado hardware manager, arm the trigger, and then resume the host program in GDB.
Debugging with ChipScope

You can use the ChipScope debugging environment and the Vivado hardware manager to help you debug your host application and kernels quickly and more effectively. These tools enable a wide range of capabilities from logic to system-level debug while your kernel is running in hardware. To achieve this, at least one of the following must be true:

- Your Vitis application project has been designed with debug cores, using the --debug.xxx compiler switch, as described in Enabling Kernels for Debugging with Chipscope.
- The RTL kernels used in your project must have been instantiated with debug cores (as described in Adding Debug IP to RTL Kernels).

Checking the FPGA Board for Hardware Debug Support

Supporting hardware debugging requires the platform to support several IP components, most notably the Debug Bridge. Talk to your platform designer to determine if these components are included in the target platform. If a Xilinx platform is used, debug availability can be verified using the platforminfo utility to query the platform. Debug capabilities are listed under the chipscope_debug objects.

For example, to query the a platform for hardware debug support, the following platforminfo command can be used:

```
$ platforminfo --json="hardwarePlatform.extensions.chipscope_debug"
xilinx_u200_xdma_201830_2
{
  "debug_networks": {
    "user": {
      'name': "User Debug Network",
      'pcie_pf': '1',
      'bar_number': '0',
      'axi_baseaddr': '0x000C0000',
      'supports_jtag_fallback': 'false',
      'supports_microblaze_debug': "true",
      'is_user_visible': "true"
    },
    "mgmt": {
      'name': "Management Debug Network",
      'pcie_pf': '0',
      'bar_number': '0',
      'axi_baseaddr': '0x001C0000',
      'supports_jtag_fallback': 'true',
      'supports_microblaze_debug': "true",
      'is_user_visible': "false"
    }
  }
}
```

The response shows that the target platform contains user and mgmt debug networks, supports debugging a MicroBlaze™ processor, and also supports JTAG fallback for the Management Debug Network.
Running XVC and HW Servers

The following steps are required to run the Xilinx virtual cable (XVC) and HW servers, host applications, and also trigger and arm the debug cores in the Vivado hardware manager.

1. Add debug IP to the kernel as discussed in Enabling Kernels for Debugging with Chipscope.
2. Modify the host program to pause at the appropriate point as described in Enabling ILA Triggers for Hardware Debug.
3. Set up the environment for hardware debug, using an automated script described in Automated Setup for Hardware Debug, or manually as described in Manual Setup for Hardware Debug.
4. Run the hardware debug flow using the following process:
   a. Launch the required XVC and the \texttt{hw\_server} of the Vivado hardware manager.
   b. Run the host program and pause at the appropriate point to enable setup of the ILA triggers.
   c. Open the Vivado hardware manager and connect to the XVC server.
   d. Set up ILA trigger conditions for the design.
   e. Continue execution of the host program.
   f. Inspect kernel activity in the Vivado hardware manager.
   g. Rerun iteratively from step b (above) as required.

Automated Setup for Hardware Debug

1. Set up your Vitis core development kit as described in Setting Up the Vitis Environment.
2. Use the \texttt{debug\_hw} script to launch the \texttt{xvc\_pcie} and \texttt{hw\_server} apps as follows:

   \begin{verbatim}
   debug_hw --xvc\_pcie /dev/xvc\_pub.<driver\_id> --hw\_server
   \end{verbatim}

   The \texttt{debug\_hw} script returns the following:

   \begin{verbatim}
   launching xvc\_pcie...
   xvc\_pcie -d /dev/xvc\_pub.<driver\_id> -s TCP::10200
   launching hw\_server...
   hw\_server -sTCP::3121
   \end{verbatim}

   \textbf{TIP:} The \texttt{/dev/xvc\_pub.<driver\_id>} driver character path is defined on your machine, and can be found by examining the \texttt{/dev} folder.

3. Modify the host code to include a pause statement \textit{after} the kernel has been created/downloaded and \textit{before} the kernel execution is started, as described in Enabling ILA Triggers for Hardware Debug.
4. Run your modified host program.
5. Launch Vivado Design Suite using the `debug_hw` script:

```
debug_hw --vivado --host <host_name> --ltx_file ./_x/link/vivado/vpl/prj/prj.runs/impl_1/debug_nets.ltx
```

**TIP:** The `<host_name>` is the name of your system.

As an example, the command window displays the following results:

```
launching vivado... ['vivado', '-source', 'vitis_hw_debug.tcl', '-tclargs', '/tmp/project_1/project_1.xpr', 'workspace/vadd_test/System/pfm_top_wrapper.ltx', 'host_name', '10200', '3121']
```

*** Vivado v2019.2 (64-bit)
*** SW Build 2245749 on Date Time
*** IP Build 2245576 on Date Time
** Copyright 1986-2019 Xilinx, Inc. All Rights Reserved.

start_gui

6. In Vivado Design Suite, run the ILA trigger.

7. Press **Enter** to continue running the host program.

8. In the Vivado hardware manager, see the interface transactions on the kernel compute unit slave control interface in the Waveform view.
Manual Setup for Hardware Debug

**TIP:** The following steps can be used when setting up Nimbix and other cloud platforms.

There are a few steps required to start the debug servers prior to debugging the design in the Vivado hardware manager.

1. Set up your Vitis core development kit as described in Setting Up the Vitis Environment.
2. Launch the `xvc_pcie` server. The file name passed to `xvc_pcie` must match the character driver file installed with the kernel device driver, where `<driver_id>` can be found by examining the `/dev` folder.

   ```
   >xvc_pcie -d /dev/xvc_pub.<device_id>
   
   **TIP:** The `xvc_pcie` server has many useful command line options. You can issue `xvc_pcie -help` to obtain the full list of available options.
   
   3. Start the `hw_server` on port 3121, and connect to the XVC server on port 10201 using the following command:

   ```
   >hw_server -e 'set auto-open-servers xilinx-xvc:localhost:10201' -e 'set always-open-jtag 1'
   
   4. Launch Vivado Design Suite and open the hardware manager:

   ```
   vivado
   
Starting Debug Servers on an Amazon F1 Instance

Instructions to start the debug servers on an Amazon F1 instance can be found here: [https://github.com/aws/aws-fpga/blob/master/hdk/docs/Virtual_JTAG_XVC.md](https://github.com/aws/aws-fpga/blob/master/hdk/docs/Virtual_JTAG_XVC.md).
Debugging Designs Using Vivado Hardware Manager

Traditionally, a physical JTAG connection is used to perform hardware debug for Xilinx devices with the Vivado hardware manager. The Vitis unified software platforms also makes use of the Xilinx virtual cable (XVC) for hardware debugging on remote accelerator cards. To take advantage of this capability, the Vitis debugger uses the XVC server, an implementation of the XVC protocol that allows the Vivado hardware manager to connect to a local or remote target device for debug, using the standard Xilinx debug cores like the ILA or the VIO IP.

The Vivado hardware manager, from the Vivado Design Suite or Vivado debug feature, can be running on the target instance or it can be running remotely on a different host. The TCP port on which the XVC server is listening must be accessible to the host running Vivado hardware manager. To connect the Vivado hardware manager to XVC server on the target, the following steps should be followed on the machine hosting the Vivado tools:

1. Launch the Vivado debug feature, or the full Vivado Design Suite.
2. Select Open Hardware Manager from the Tasks menu, as shown in the following figure.
3. Connect to the Vivado tools `hw_server`, specifying a local or remote connection, and the **Host name** and **Port**, as shown below.
4. Connect to the target instance Virtual JTAG XVC server.
5. Select the `debug_bridge` instance from the Hardware window in the Vivado hardware manager.

Specify the probes file (.ltx) for your design adding it to the Probes → File entry in the Hardware Device Properties window. Adding the probes file refreshes the hardware device, and Hardware window should now show the debug cores in your design.

**TIP:** If the kernel has debug cores as specified in Enabling Kernels for Debugging with Chipscope, the probes file (.ltx) is written out during the implementation of the kernel by the Vivado tool.

6. The Vivado hardware manager can now be used to debug the kernels running on the Vitis software platform. Arm the ILA cores in your kernels and run your host application.
JTAG Fallback for Private Debug Network

Hardware debug for the Alveo Data Center accelerator cards typically uses the XVC-over-PCIe connection due to the inaccessibility of the physical card, and the JTAG connector on the card. While XVC-over-PCIe allows you to remotely debug your application running on the target platform, certain conditions such as AXI interconnect system hangs can prevent you from accessing the hardware debug functionality that depends on these PCIe/AXI features. Being able to debug these kinds of conditions is especially important for platform designers.

The JTAG Fallback feature is designed to provide access to debug networks that were previously only accessible through XVC-over-PCIe. The JTAG Fallback feature can be enabled without having to change the XVC-over-PCIe-based debug network in the platform design.

On the host side, when the Vivado hardware manager user connects through the `hw_server` to a JTAG cable that is connected to the physical JTAG pins of the accelerator card, or device under test (DUT), the `hw_server` disables the XVC-over-PCIe pathway to the hardware. This lets you use the XVC-over-PCIe cable as your primary debug path, but enable debug over the JTAG cable directly when it is required in certain situations. When you disconnect from the JTAG cable, the `hw_server` re-enables the XVC-over-PCIe pathway to the hardware.
JTAG Fallback Steps

Here are the steps required to enable JTAG Fallback:

1. Enable the JTAG Fallback feature of the Debug Bridge (AXI-to-BSCAN mode) master of the debug network to which you want to provide JTAG access. This step enables a BSCAN slave interface on this Debug Bridge instance.

2. Instantiate another Debug Bridge (BSCAN Primitive mode) in the static logic partition of the platform design.

3. Connect the BSCAN master port of the Debug Bridge (BSCAN Primitive mode) from step 2 to the BSCAN slave interface of the Debug Bridge (AXI-to-BSCAN mode) from step 1.

Utilities for Hardware Debugging

In some cases, the normal Vitis IDE and command line debug features are limited in their ability to isolate an issue. This is especially true when the software or hardware appears not to make any progress (hangs). These kinds of system issues are best analyzed with the help of the utilities mentioned in this section.

Using the Linux dmesg Utility

Well-designed kernels and modules report issues through the kernel ring buffer. This is also true for Vitis technology modules that allow you to debug the interaction with the accelerator board on the lowest Linux level.

The `dmesg` utility is a Linux tool that lets you read the kernel ring buffer. The kernel ring buffer holds kernel information messages in a circular buffer. A circular buffer of fixed size is used to limit the resource requirements by overwriting the oldest entry with the next incoming message.

**TIP:** In most cases, it is sufficient to work with the less verbose `xbutil` feature to localize an issue. Refer to Using the Xilinx `xbutil` Utility for more information on using this tool for debug.

In the Vitis technology, the `xocl` module and `xclmgmt` driver modules write informational messages to the ring buffer. Thus, for an application hang, crash, or any unexpected behavior (like being unable to program the bitstream, etc.), the `dmesg` tool should be used to check the ring buffer.

The following image shows the layers of the software platform associated with the target platform.
To review messages from the Linux tool, you should first clear the ring buffer:

```bash
sudo dmesg -c
```

This flushes all messages from the ring buffer and makes it easier to spot messages from the `xocl` and `xclmgmt`. After that, start your application and run `dmesg` in another terminal.

```bash
sudo dmesg
```

The `dmesg` utility prints a record shown in the following example:

```plaintext
In the example shown above, the AXI Firewall 2 has tripped, which is better examined using the `xbutil` utility.

**Using the Xilinx `xbutil` Utility**

The Xilinx board utility (`xbutil`) is a powerful standalone command line utility that can be used to debug lower level hardware/software interaction issues. A full description of this utility can be found in `xbutil Utility`.

With respect to debugging, the following `xbutil` options are of special interest:
• **query**: Provides an overall status of a card including information on the kernels in card memory.

• **program**: Downloads a binary (xclbin) to the programmable region of the Xilinx device.

• **status**: Extracts the status of the Performance Monitors (spm) and the Lightweight AXI Protocol Checkers (lapc).

### Techniques for Debugging Application Hangs

This section discusses debugging issues related to the interaction of the host code and the accelerated kernels. Problems with these interactions manifest as issues such as machine hangs or application hangs. Although the GDB debug environment might help with isolating the errors in some cases (xprint), such as hangs associated with specific kernels, these issues are best debugged using the `dmesg` and `xbutil` commands as shown here.

If the process of hardware debugging does not resolve the problem, it is necessary to perform hardware debugging using the ChipScope feature.

### AXI Firewall Trips

The AXI firewall should prevent host hangs. This is why the AXI Protocol Firewall IP is included in all production Vitis platforms. When the firewall trips, one of the first checks to perform is confirming if the host code and kernels are set up to use the same memory banks. The following steps detail how to perform this check.

1. **Use `xbutil` to program the FPGA:**

   ```bash
   xbutil program -p <xclbin>
   ```

   **TIP:** Refer to `xbutil Utility` for more information on `xbutil`.

2. **Run the `xbutil` query option to check memory topology:**

   ```bash
   xbutil query
   ```

   In the following example, there are no kernels associated with memory banks:
3. If the host code expects any DDR banks/PLRAMs to be used, this report should indicate an issue. In this case, it is necessary to check kernel and host code expectations. If the host code is using the Xilinx OpenCL extensions, it is necessary to check which DDR banks should be used by the kernel. These should match the `connectivity.sp` options specified as discussed in Mapping Kernel Ports to Memory.

**Kernel Hangs Due to AXI Violations**

It is possible for the kernels to hang due to bad AXI transactions between the kernels and the memory controller. To debug these issues, it is required to instrument the kernels.

1. The Vitis core development kit provides two options for instrumentation to be applied during `v++` linking (`--link`). Both of these options add hardware to your implementation, and based on resource utilization it might be necessary to limit instrumentation.

   a. Add Lightweight AXI Protocol Checkers (`lapc`). These protocol checkers are added using the `--debug.protocol` option, as explained in `--debug Options`. The following syntax is used:

   ```
   --debug.protocol <compute_unit_name>:<interface_name>
   ```

   In general, the `<interface_name>` is optional. If not specified, all ports on the CU are expected to be analyzed. The `--debug.protocol` option is used to define the protocol checkers to be inserted. This option can accept a special keyword, `all`, for `<compute_unit_name>` and/or `<interface_name>`.

   **Note:** Multiple `--debug.xxx` options can be specified in a single command line, or configuration file.
b. Adding Performance Monitors (am, aim, asm) enables the listing of detailed communication statistics (counters). Although this is most useful for performance analysis, it provides insight during debugging on pending port activities. The Performance Monitors are added using the --profile option as described in --profile Options. The basic syntax for the --profile option is:

```
--profile.data <krnl_name>|all:<cu_name>|all:<intrfc_name>|all:<counters>|all
```

Three fields are required to determine the specific interface to attach the performance monitor to. However, if resource consumption is not an issue, the keyword all lets you apply the monitoring to all existing kernels, compute units, and interfaces with a single option. Otherwise, you can specify the kernel_name, cu_name, and interface_name explicitly to limit instrumentation.

The last option, <counters>|all, allows you to restrict the information gathering to just counters for large designs, while all (default) includes the collection of actual trace information.

**Note:** Multiple --profile options can be specified in a single command line, or configuration file.

```
[profile]
datakernel1:cu1:m_axi_gmem0
datakernel1:cu1:m_axi_gmem1
datakernel2:cu2:m_axi_gmem
```

2. When the application is rebuilt, rerun the host application using the xclbin with the added AIM IP and LAPC IP.

3. When the application hangs, you can use xbuutil status to check for any errors or anomalies.

4. Check the AIM output:
   - Run xbuutil status --aim a couple of times to check if any counters are moving. If they are moving then the kernels are active.

   **TIP:** Testing AIM output is also supported through GDB debugging using the command extension xstatus spm.

   - If the counters are stagnant, the outstanding counts greater than zero might mean some AXI transactions are hung.

5. Check the LAPC output:
   - Run xbuutil status --lapc to check if there are any AXI violations.

   **TIP:** Testing LACP output is also supported through GDB debugging using the command extension xstatus lapc.

   - If there are any AXI violations, it implies that there are issues in the kernel implementation.
Host Application Hangs When Accessing Memory

Application hangs can also be caused by incomplete DMA transfers initiated from the host code. This does not necessarily mean that the host code is wrong; it might also be that the kernels have issued illegal transactions and locked up the AXI.

1. If the platform has an AXI firewall, such as in the Vitis target platforms, it is likely to trip. The driver issues a SIGBUS error, kills the application, and resets the device. You can check this by running xbutil query. The following figure shows such an error in the firewall status:

```
Firewall Last Error Status:
  0: 0x0 (GOOD)
  1: 0x0 (GOOD)
  2: 0x80000 (RECS_WRITE_TO_BVALID_MAX_WAIT).
  Error occurred on Tue 2017-12-19 11:39:13 PST
Xclbin ID: 0x5a39da87
```

**TIP:** If the firewall has not tripped, the Linux tool, dmesg, can provide additional insight.

2. When you know that the firewall has tripped, it is important to determine the cause of the DMA timeout. The issue could be an illegal DMA transfer, or kernel misbehavior. However, a side effect of the AXI firewall tripping is that the health check functionality in the driver resets the board after killing the application; any information on the device that might help with debugging the root cause is lost. To debug this issue, disable the health check thread in the xclmgmt kernel module to capture the error. This uses common Unix kernel tools in the following sequence:

   a. `sudo modinfo xclmgmt`: This command lists the current configuration of the module and indicates if the health_check parameter is ON or OFF. It also returns the path to the xclmgmt module.

   b. `sudo rmmod xclmgmt`: This removes and disables the xclmgmt kernel module.

   c. `sudo insmod <path to module>/xclmgmt.ko health_check=0`: This re-installs the xclmgmt kernel module with the health check disabled.

**TIP:** The path to this module is reported in the output of the call to modinfo.

3. With the health check disabled, rerun the application. You can use the kernel instrumentation to isolate this issue as previously described.

**Typical Errors Leading to Application Hangs**

The user errors that typically create application hangs are listed below:

- Read-before-write in 5.0+ target platforms causes a Memory Interface Generator error correction code (MIG ECC) error. This is typically a user error. For example, this error might occur when a kernel is expected to write 4 KB of data in DDR, but it produces only 1 KB of data, and then try to transfer the full 4 KB of data to the host. It can also happen if you supply a 1 KB buffer to a kernel, but the kernel tries to read 4 KB of data.
An ECC read-before-write error also occurs if no data has been written to a memory location as the last bitstream download which results in MIG initialization, but a read request is made for that same memory location. ECC errors stall the affected MIG because kernels are usually not able to handle this error. This can manifest in two different ways:

1. The CU might hang or stall because it cannot handle this error while reading or writing to or from the affected MIG. The xbutil query shows that the CU is stuck in a BUSY state and is not making progress.

2. The AXI Firewall might trip if a PCIe® DMA request is made to the affected MIG, because the DMA engine is unable to complete the request. AXI Firewall trips result in the Linux kernel driver killing all processes which have opened the device node with the SIGBUS signal. The xbutil query shows if an AXI Firewall has indeed tripped and includes a timestamp.

If the above hang does not occur, the host code might not read back the correct data. This incorrect data is typically 0s and is located in the last part of the data. It is important to review the host code carefully. One common example is compression, where the size of the compressed data is not known up front, and an application might try to migrate more data to the host than was produced by the kernel.

**Defensive Programming**

The Vitis compiler is capable of creating very efficient implementations. In some cases, however, implementation issues can occur. One such case is if a write request is emitted before there is enough data available in the process to complete the write transaction. This can cause deadlock conditions when multiple concurrent kernels are affected by this issue and the write request of a kernel depends on the input read being completed.

To avoid these situations, a conservative mode is available on the adapter. In principle, it delays the write request until it has all of the data necessary to complete the write. This mode is enabled during compilation by applying the following `--advanced.param` option to the v++ compiler:

```
--advanced.param:compiler.axiDeadLockFree=yes
```

Because enabling this mode can impact performance, you might prefer to use this as a defensive programming technique where this option is inserted during development and testing and then removed during optimization. You might also want to add this option when the accelerator hangs repeatedly.
Debugging on Embedded Processor Platforms

Debugging on embedded processor platforms, such as the `xilinx_zcu104_base_202010_1` platform, requires the use of the QEMU emulation environment to model the Arm processor and operating system for the device. As described in the next sections, running or debugging the application requires the additional step of launching the emulator, or connecting to the hardware platform through a TCF agent.

Emulation Debug for Embedded Processors

From within the Vitis IDE, launching debug for the software and hardware emulation builds include the following steps:

1. In the Assistant view, right-click the `Emulation-SW` or `Emulation-HW` build and select `Set Active` to make the build active.

2. From the Assistant view menu, select the `Debug` ( screenshot ) command, and select the `Launch on Emulator` command to launch the debug environment.

   This will open the Launch on Emulator dialog box as shown in the following figure. This prompts you to confirm launching the emulation environment and connecting to it using a Linux TCF agent. Select `Start Emulator and Debug` to continue.

   ![Launch Emulator Dialog](image)

   This launches the emulation environment (QEMU), and loads the application in preparation for debugging. The application is paused as it enters the `main()` function. The Debug perspective is opened in the Vitis IDE, and you are ready to begin debugging your application.
Hardware Debug for Embedded Processors

For hardware builds the setup involves the following steps:

1. Copy the contents of the `<project>/Hardware/sd_card/sd_card` folder to a physical SD card. This creates a bootable medium for your target platform.

2. Insert the SD card into the card reader of your embedded processor platform.

3. Change the boot-mode settings of the platform to SD boot mode, and power up the board.

4. After the device is booted, enter the `mount` command at the command prompt to get a list of mount points. As shown in the following figure, the `mount` command displays mounting information for the system.

   TIP: Be sure to capture the proper path for the `cd` command in the next step, and subsequent commands, based on the results of the `mount` command.
5. Execute the following commands, for example:

```
cd /run/media/mmcblk0p1
source init.sh
cat /etc/xocl.txt
```

The `cat` command will display the platform name `xilinx_vck190_base_202010_1` to let you confirm it is the same as your specified platform and that your setup is correct.

6. Run `ifconfig` to get the IP address of the target card. You will use the IP address to set up a TCF agent connection in Vitis IDE to connect to the assigned IP address of the embedded processor platform.

7. Create a target connection to the remote accelerator card. Use the `Window → Show view → Xilinx → Target connections` command to open the Target Connections view.

8. In the Target Connections view, right-click on the Linux TCF Agent and select the New Target command to open the New Target Connection dialog box.

9. Specify the Target Name, enable the Set as default target check box, and specify the Host IP address of the accelerator card that you obtained in an earlier step.
10. Click OK to close the dialog box and continue.

11. In the Assistant view, right-click on the Hardware build and select Set Active to make it the active build.

12. From the Assistant view menu, select the Debug ((fmt) command, and select the Debug Configurations command. This opens the Debug Configurations dialog box to let you configure debug for the Hardware build on your specific platform.

Set the following fields on the Main tab of the dialog box:
• **Name:** Specifies a name for your Hardware debug configuration.

• **Linux TCF Agent:** Selects the new agent you built with the specified IP address for the accelerator card.

• **Configuration:** Ensure you have selected the Hardware configuration.

• **Enable Profiling:** If you want to capture trace data from events.

Select the **Application** tab in the Debug Configuration dialog box to see the following fields:

Set the following fields on the Application tab:

• **Local File Path:** Specifies where the files created on the target platform will be written back into your local disk.

• **Remote File Path:** Specifies the remote mount location from the accelerator card as determined in an earlier step.

• **Working directory:** Specifies the location to write files created on the target platform.

13. Select **Apply** to save your changes, and **Debug** to start the process.

This opens the Debug perspective in the Vitis IDE, and connects to the PS application on your hardware platform. The application automatically breaks at the `main()` function to let you set up and configure the debug environment.
Example of Command Line Debugging

To help you get familiar with debugging using the command line flow, this example walks you through building and debugging the IDCT example available from the Xilinx GitHub.

1. In a terminal, set up your environment as described in Setting Up the Vitis Environment.
2. If you have not already done it, clone the Vitis Examples GitHub repository to acquire all of the Vitis examples:

   ```
   git clone https://github.com/Xilinx/Vitis_Accel_Examples.git
   ```

   This creates a Vitis_Examples directory which includes the IDCT example.
3. CD to the IDCT example directory:

   ```
   cd Vitis_Examples/vision/idct/
   ```

   The host code is fully contained in src/idct.cpp and the kernel code is part of src/krnl_idct.cpp.
4. Build the kernel software for software emulation as discussed in Building the Device Binary.
   a. Compile the kernel object file for debugging using the v++ compiler, where -g indicates that the code is compiled for debugging:

   ```
   v++ -t sw_emu --platform <DEVICE> -g -c -k krnl_idct \
   -o krnl_idct.xo src/krnl_idct.cpp
   ```
   
b. Link the kernel object file, also specifying -g:

   ```
   v++ -g -l -t sw_emu --platform <DEVICE> -config config.cfg \
   -o krnl_idct.xclbin krnl_idct.xo
   ```

   The --config option specifies the configuration file, config.cfg, that contains the directives for the build process as described in the Vitis Compiler Configuration File. The contents of the configuration file are as follows:

   ```
   kernel_frequency=250
   [connectivity]
   nk=krnl_idct_1:krnl_idct_1
   sp=krnl_idct_1.m_axi_gmem0:DDR[0]
   sp=krnl_idct_1.m_axi_gmem1:DDR[0]
   sp=krnl_idct_1.m_axi_gmem2:DDR[1]
   [advanced]
   prop=solution.hls_pre_tcl='src/hls_config.tcl'
   ```
5. Compile and link the host code for debugging using the GNU compiler chain, g++ as described in Building the Host Program:

   **Note:** For embedded processor target platforms you will use the GNU Arm cross-compiler as described in Compiling and Linking for Arm.
a. Compile host code C++ files for debugging using the `-g` option:

```bash
$ g++ -c -I${XILINX_XRT}/include -g -o idct.o src/idct.cpp
```

b. Link the object files for debugging using `-g`:

```bash
$ g++ -g -lOpenCL -lpthread -lrt -lstdc++ -L${XILINX_XRT}/lib/ -o idct idct.o
```

6. As described in `emconfigutil Utility`, prepare the emulation environment using the following command:

```bash
$ emconfigutil --platform <device>
```

The actual emulation mode (`sw_emu` or `hw_emu`) then needs to be set through the `XCL_EMULATION_MODE` environment variable. In C-shell this would be as follows:

```bash
$ setenv XCL_EMULATION_MODE sw_emu
```

7. As described in `xrt.ini File`, you must setup the runtime for debug. In the same directory as the compiled host application, create an `xrt.ini` file with the following content:

```
[Debug]
app_debug=true
```

8. Run GDB on the host and kernel code. The following steps guide you through the command line debug process which requires three separate command terminals, setup as described in `Setting Up the Vitis Environment`.

a. In the first terminal, start the XRT debug server, which handles the transactions between the host and kernel code:

```bash
$ ${XILINX_VITIS}/bin/xrt_server --sdx-url
```

b. In a second terminal, set the emulation mode:

```bash
$ setenv XCL_EMULATION_MODE sw_emu
```

Run GDB by executing the following:

```bash
$ xgdb --args idct krnl_idct.xclbin
```

Enter the following on the `gdb` prompt:

```
run
```

c. In the third terminal, attach the software emulation or hardware emulation model to GDB to allow stepping through the design. Here, there is a difference between running software emulation and hardware emulation. In either flow, start up another `xgdb`:

```bash
$ xgdb
```
For debugging in software emulation:

- Type the following on the `gdb` prompt:

```
file <XILINX_VITIS>/data/emulation/unified/cpu_em/generic_pcie/model/genericpciemodel
```

**Note**: Because GDB does not expand the environment variable, you must specify the path to the Vitis software platform installation as represented by `<XILINX_VITIS>`.

For debugging in hardware emulation:

1. Locate the `xrt_server` temporary directory: `/tmp/sdx/$uid`.
2. Find the `xrt_server` process ID (PID) containing the DWARF file of this debug session.
3. At the `gdb` prompt, run:

```
file /tmp/sdx/$uid/$pid/NUM.DWARF
```

In either case, connect to the kernel process:

```
target remote :NUM
```

Where `NUM` is the number returned by the `xrt_server` as the GDB listener port.

At this point, debugging the host and kernel code can be done as usual with GDB, with the host code and the kernel code running in two different GDB sessions. This is common when dealing with different processes.

**IMPORTANT!** Be aware that the application might hit a breakpoint in one process before the next breakpoint in the other process is hit. In these cases, the debugging session in one terminal appears to hang, while the second terminal is waiting for input.
Vitis Environment Reference Materials

The reference materials contained here include the following:

- **Vitis Compiler Command**: A description of the compiler options (-c), the linking options (-l), options common to both compile and linking, and a discussion of the --config options.

- The *xrt.ini* file is used to initialize XRT to produce reports, debug, and profiling data as it transacts business between the host and kernels. This file is used when the application is run, for emulation or hardware builds, and must be created manually when the build process is run from the command line.

- Various Xilinx utilities are provided for the Vitis tools and Xilinx® Runtime (XRT) to provide detailed information about the platform resources, including SLR and memory resource availability, to help you construct the v++ command line, and manage the build and run process.
  - **platforminfo Utility**: The platforminfo utility queries the platforms for which Vitis™ installation to use.
  - **kernelinfo Utility**: The kernelinfo utility prints the function definitions in the given Xilinx object file (O) file.
  - **emconfigutil Utility**: The emulation configuration utility (emconfigutil) is used to automate the creation of the emulation configuration file.
  - **xclbinutil Utility**: The xclbinutil utility operates on a xclbin produced by the Vitis Compiler.
  - **xbutil Utility**: The Xilinx Board Utility (xbutil) is a command line tool used to perform various board installation, administration, and debug tasks.
  - **xbmgmt Utility**: The Xilinx® Board Management (xbmgmt) utility is a standalone command line tool that is included with the Xilinx Runtime (XRT) installation package. It supports both Alveo Data Center accelerator cards and embedded processor-based platforms.

- **TIP**: The Xilinx® Runtime (XRT) Architecture reference material is available on the Xilinx Runtime GitHub repository.

- **package_xo Command**: The Tcl command used in the Vivado Design Suite to package an RTL IP into an XO file, as described in RTL Kernels.
• **HLS Pragmas:** A description of pragmas used by the Vitis HLS tool in synthesizing C/C++ kernels.

• **OpenCL Attributes:** Descriptions of `__attribute__` that can be added to OpenCL™ kernels to direct the results of the kernel build process.
Chapter 22

Vitis Compiler Command

This section describes the Vitis compiler command, v++, and the various options it supports for both compiling and linking FPGA binary.

The Vitis compiler is a standalone command line utility for both compiling kernel accelerator functions into Xilinx object (XO) files, and linking them with other XO files and supported platforms to build an FPGA binary.

For additional information about the use of the v++ command options for compile, link, packaging, and general processes, see these additional sections:

- Compiling Kernels with the Vitis Compiler
- Linking the Kernels
- Packaging the System

Vitis Compiler General Options

The Vitis compiler supports many options for both the compilation process and the linking process. These options provide a range of features, and some apply specifically to compile or link, while others can be used, or are required for both compile and link.

TIP: All Vitis compiler options can be specified in a configuration file for use with the --config option, as discussed in the Vitis Compiler Configuration File. For example, the --platform option can be specified in a configuration file without a section head using the following syntax:

platform=xilinx_u200_xdma_201830_2

--advanced

- Applies to: Compile and link

Specify parameters and properties for use by the v++ command. See --advanced Options for more information.
--board_connection

- Applies to: Compile and link

**--board_connection**

Specifies a dual in-line memory module (DIMM) board file for each DIMM connector slot. The board is specified using the Vendor:Board:Name:Version (vbnv) attribute of the DIMM card as it appears in the board repository.

For example:

```
<DIMM_connector>:<vbnv_of_DIMM_board>
```

-c | --compile

- Applies to: Compile

**--compile**

Required for compilation, but mutually exclusive with --link and --package. Run `v++ -c` to generate XO files from kernel source files.

--clock

- Applies to: Link

Provide a method for assigning clocks to kernels during the linking process. See **--clock Options** for more information.

--config

- Applies to: Compile, link, and package

**--config <config_file> ...**

Specifies a configuration file containing `v++` command options. The configuration file can be used to capture compilation, linking, or packaging strategies, that can be easily reused by referring to the config file on the `v++` command line. In addition, the config file allows the `v++` command line to be shortened to include only the options that are not specified in the config file. Refer to the **Vitis Compiler Configuration File** for more information.

**TIP:** Multiple configuration files can be specified on the `v++` command line. A separate **--config** switch is required for each file used. For example:

```
v++ -l --config cfg_connectivity.cfg --config cfg_vivado.cfg ...
```
--connectivity

- Applies to: Link

Used to specify important architectural details of the device binary during the linking process. See --connectivity Options for more information.

--custom_script

- Applies to: Compile and link

```
--custom_script <kernel_name>:<file_name>
```

This option lets you specify custom Tcl scripts to be used in the build process during compilation or linking. Use with the --export_script option to create, edit, and run the scripts to customize the build process.

When used with the v++ --compile command, this option lets you specify a custom HLS script to be used when compiling the specified kernel. The script lets you modify or customize the Vitis HLS tool. Use the --export_script option to extract a Tcl script Vitis HLS uses to compile the kernel, modify the script as needed, and resubmit using the --custom_script option to better manage the kernel build process.

The argument lets you specify the kernel name, and path to the Tcl script to apply to that kernel. For example:

```
v++ -c -k kernel1 -export_script ...
*** Modify the exported script to customize in some way, then resubmit. ***
v++ -c --custom_script kernel1:./kernel1.tcl ...
```

When used with the v++ --link command for the hardware build target (-t hw), this option lets you specify the absolute path to an edited run_script_map.dat file. This file contains a list of steps in the build process, and Tcl scripts that are run by the Vitis and Vivado tools during those steps. You can edit run_script_map.dat to specify custom Tcl scripts to run at those steps in the build process. You must use the following steps to customize the Tcl scripts:

1. Run the build process specifying the --export_script option as follows:

```
v++ -t hw -l -k kernel1 -export_script ...
```

2. Copy the Tcl scripts referenced in the run_script_map.dat file for any of the steps you want to customize. For example, copy the Tcl file specified for the synthesis run, or the implementation run. You must copy the file to a separate location, outside of the project build structure.

3. Edit the Tcl script to add or modify any of the existing commands to create a new custom Tcl script.

4. Edit the run_script_map.dat file to point a specific implementation step to the new custom script.
5. Relaunch the build process using the `--custom_script` option, specifying the absolute path to the `run_script_map.dat` file as shown below:

```
v++ -t hw -l -k kernel1 -custom_script /path/to/run_script_map.dat
```

**IMPORTANT!** When editing a custom synthesis run script, you must either comment out the lines related to the `dont_touch.xdc` file, or edit the lines to point to a new user-specified `dont_touch.xdc` file. The specific lines to comment or edit are shown below:

```
read_xdc dont_touch.xdc
set_property used_in_implementation false [get_files dont_touch.xdc]
```

*The synthesis run will return an error related to a missing `dont_touch.xdc` file if this is not done.*

**--debug**

- **Applies to:** Link

Specify debug IP core insertion in the device binary (.xclbin). See **--debug Options** for more information.

**-D | --define**

- **Applies to:** Compile and link

```
--define <arg>
```

Valid macro name and definition pair: `<name>`=`<definition>`.

Predefine name as a macro with definition. This option is passed to the `v++` pre-processor.

**--export_script**

- **Applies to:** Compile and link

```
--export_script
```

This option runs the build process up to the point of exporting a script file, or list of script files, and then stops execution. The build process must be completed using the `--custom_script` option. This lets you edit the exported script, or list of scripts, and then rerun the build using your custom scripts.

When used with the `v++ --compile` command, this option exports a Tcl script for the specified kernel, `<kernel_name>.tcl`, that can be used to execute Vitis HLS, but stops the build process before actually launching the HLS tool. This lets you interrupt the build process to edit the generated Tcl script, and then restart the build process using the `--custom_script` option, as shown in the following example:

```
v++ -c -k kernel1 -export_script ...
```
When used with the `v++ --link` command for the hardware build target (`-t hw`), this option exports a `run_script_map.dat` file in the current directory. This file contains a list of steps in the build process, and Tcl scripts that are run by the Vitis and Vivado tools during those steps. You can edit the specified Tcl scripts, customizing the build process in those scripts, and relaunch the build using the `--custom_script` option. Export the `run_script_map.dat` file using the following command:

```bash
v++ -t hw -l -k kernel1 -export_script ...
```

### --from_step

- **Applies to:** Compile and link

```bash
--from_step <arg>
```

Specifies a step name for the Vitis compiler build process, to start the build process from that step. If intermediate results are available, the link process will fast forward and begin execution at the named step if possible. This allows you to run the build through a `--to_step`, and then resume the build process at the `--from_step`, after interacting with your project in some method. You can use the `--list_step` option to determine the list of valid steps.

**IMPORTANT!** The `--from_step` and `--to_step` options are sequential build options that require you to use the same project directory when launching the Vitis compiler using `--from_step` to resume the build as you specified when using `--to_step` to start the build.

For example:

```bash
v++ --link --from_step vpl.update_bd
```

### -g

- **Applies to:** Compile and link

```bash
-g
```

Generates code for debugging the kernel. Using this option adds features to facilitate debugging the kernel as it is compiled and the FPGA binary is built.

For example:

```bash
v++ -g ...
```

### -h | --help

```bash
-h
```
Prints the help contents for the `v++` command. For example:

```
v++ -h
```

**--hls**

- **Applies to:** Compile

Specify options for the Vitis HLS synthesis process during kernel compilation. See **--hls Options** for more information.

**-I | --include**

- **Applies to:** Compile and link

```
--include <arg>
```

Add the specified directory to the list of directories to be searched for header files. This option is passed to the Vitis compiler pre-processor.

**<input_file>**

- **Applies to:** Compile and link

```
<input_file1> <input_file2> ...
```

Specifies an OpenCL or C/C++ kernel source file for `v++` compilation, or Xilinx object (XO) files for `v++` linking.

For example:

```
v++ -l kernel1.xo kernelRTL.xo ...
```

**--interactive**

- **Applies to:** Compile and link

```
--interactive [ impl ]
```

`v++` configures the needed environment and launches the Vivado tool with the implementation project.

Because you are interactively launching the Vivado tool, the linking process is stopped after the `vpl` step, which is the equivalent of using the **--to_step vpl** option in your `v++` command.

When you are done interactively working with the Vivado tool, and you save the design checkpoint (DCP), you can resume the Vitis compiler linking process using the `v++ --from_step rtdgen`, or use the `--reuse_impl` or `--reuse_bit` options to read in the implemented DCP file or bitstream.
For example:

```
v++ --interactive impl
## Interactively use the Vivado tool
v++ --from_step rtdgen
```

- **-k | --kernel**
  - **Applies to:** Compile
  ```
  --kernel <arg>
  ```

Compile only the specified kernel from the input file. Only one `-k` option is allowed per `v++` command. Valid values include the name of the kernel to be compiled from the input `.cl` or `.c/.cpp` kernel source code.

This is required for C/C++ kernels, but is optional for OpenCL kernels. OpenCL uses the `kernel` keyword to identify a kernel. For C/C++ kernels, you must identify the kernel by `-k` or `--kernel`.

When an OpenCL source file is compiled without the `-k` option, all the kernels in the file are compiled. Use `-k` to target a specific kernel.

For example:

```
v++ -c --kernel vadd
```

---

**--kernel_frequency**

- **Applies to:** Compile and link
  ```
  --kernel_frequency <freq> | <clockID>:<freq>[<clockID>:<freq>]
  ```

Specifies a user-defined clock frequency (in MHz) for the kernel, overriding the default clock frequency defined on the hardware platform. The `<freq>` specifies a single frequency for kernels with only a single clock, or can be used to specify the `<clockID>` and the `<freq>` for kernels that support two clocks.

The syntax for overriding the clock on a platform with only one kernel clock, is to simply specify the frequency in MHz:

```
v++ --kernel_frequency 300
```

To override a specific clock on a platform with two clocks, specify the clock ID and frequency:

```
v++ --kernel_frequency 0:300
```
To override both clocks on a multi-clock platform, specify each clock ID and the corresponding frequency. For example:

```
v++ --kernel_frequency 0:300|1:500
```

```
-l | --link
```

This is a required option for the linking process, which follows compilation, but is mutually exclusive with `--compile` or `--package`. Run `v++` in link mode to link XO input files and generate an XCLBIN output file.

```
--linkhook
```

- **Applies to:** Link

Lets you customize the build process for the device binary by specifying Tcl scripts to be run at specific steps in the implementation flow. See `--linkhook Options` for more information.

```
--list_steps
```

- **Applies to:** Compile and link

```
--list_steps
```

List valid run steps for a given target. This option returns a list of steps that can be used in the `--from_step` or `--to_step` options. The command must be specified with the following options:

- `-t | --target [sw_emu | hw_emu | hw ]:
- `[ --compile | --link ]: Specifies the list of steps from either the compile or link process for the specified build target.

For example:

```
v++ -t hw_emu --link --list_steps
```

```
--log_dir
```

- **Applies to:** Compile and link

```
--log_dir <dir_name>
```

Specifies a directory to store log files into. If `--log_dir` is not specified, the tool saves the log files to `./_x/logs`. Refer to Output Directories of the `v++` Command for more information.
For example:

```shell
v++ --log_dir /tmp/myProj_logs ...
```

--message_rules

- **Applies to:** Compile and link

```shell
v++ --message_rules <file_name>
```

Specifies a message rule file with rules for controlling messages. Refer to Using the Message Rule File for more information.

For example:

```shell
v++ --message_rules ./minimum_out.mrf ...
```

--no_ip_cache

- **Applies to:** Compile and link

```shell
v++ --no_ip_cache ...
```

Disables the IP cache for out-of-context (OOC) synthesis for Vivado Synthesis. Disabling the IP cache repository requires the tool to regenerate the IP synthesis results for every build, and can increase the build time. However, it also results in a clean build, eliminating earlier results for IP in the design.

For example:

```shell
v++ --no_ip_cache ...
```

-O | --optimize

- **Applies to:** Compile and link

```shell
v++ --optimize <arg>
```

This option specifies the optimization level of the Vivado implementation results. Valid optimization values include the following:

- 0: Default optimization. Reduces compilation time.

- 1: Optimizes to reduce power consumption by running Vivado implementation strategy `Power_DefaultOpt`. *This takes more time to build the design.*

- 2: Optimizes to increase kernel speed. This option increases build time, but also improves the performance of the generated kernel by adding the `PHYS_OPT_DESIGN` step to implementation.
• 3: This optimization provides the highest level performance in the generated code, but compilation time can increase considerably. This option specifies retiming during synthesis, and enables both PHYS_OPT_DESIGN and POST_ROUTE_PHYS_OPT_DESIGN during implementation.

• s: Optimizes the design for size. This reduces the logic resources of the device used by the kernel by running the Area_Explore implementation strategy.

• quick: Reduces Vivado implementation time, but can reduce kernel performance, and increases the resources used by the kernel. This enables the Flow_RuntimeOptimized strategy for both synthesis and implementation.

For example:

```
v++ --link --optimize 2 -o | --output
```

- `--output`  
  - Applies to: Compile and link  

```
-o <output_name>
```

Specifies the name of the output file generated by the v++ command. The compilation (-c) process output name must end with the XO file suffix, for Xilinx object file. The linking (-l) process output file must end with the XCLBIN file suffix, for Xilinx executable binary.

For example:

```
v++ -o krnl_vadd.xo
```

If --o or --output are not specified, the output file names will default to the following:

- a.o for compilation.  
- a.xclbin for linking.

- `--package`  
  - Applies to: Package  

Specify options for the Vitis compiler to package your design for either running emulation or running on hardware. See --package Options for more information.

- `--platform`  
  - Applies to: Compile and link  

```
--platform <platform_name>
```
Specifies the name of a supported acceleration platform as specified by the $PLATFORM_REPO_PATHS environment variable, or the full path to the platform .xpfm file. For a list of supported platforms for the release, see the Vitis Software Platform Release Notes.

This is a required option for both compilation and linking, to define the target Xilinx platform of the build process. The --platform option accepts either a platform name, or the path to a platform file xpfm, using the full or relative path.

---

**IMPORTANT!** The specified platform and build targets for compiling and linking must match. The --platform and -t options specified when the XO file is generated by compilation, must be the --platform and -t used during linking. For more information, see platforminfo Utility.

For example:

```
v++ --platform xilinx_u200_xdma_201830_2 ...
```

---

**TIP:** All Vitis compiler options can be specified in a configuration file for use with the --config option. For example, the platform option can be specified in a configuration file without a section head using the following syntax:

```
platform=xilinx_u200_xdma_201830_2
```

---

**--profile**

- **Applies to:** Compile and link

Specify options to configure the Xilinx runtime environment to capture application performance information. See --profile Options for more information.

---

**--remote_ip_cache**

- **Applies to:** Compile and link

```
--remote_ip_cache <dir_name>
```

Specifies the location of the remote IP cache directory for Vivado Synthesis to use during out-of-context (OOC) synthesis of IP. OOC synthesis lets the Vivado synthesis tool reuse synthesis results for IP that have not been changed in iterations of a design. This can reduce the time required to build your .xclbin files, due to reusing synthesis results.

When the --remote_ip_cache option is not specified the IP cache is written to the current working directory from which v++ was launched. You can use this option to provide a different cache location, used across multiple projects for instance.

For example:

```
v++ --remote_ip_cache /tmp/IP_cache_dir ...
```
--report_dir

- Applies to: Compile and link

```bash
--report_dir <dir_name>
```

Specifies a directory to store report files into. If `--report_dir` is not specified, the tool saves the report files to `.\x\reports`. Refer to Output Directories of the v++ Command for more information.

For example:

```bash
v++ --report_dir /tmp/myProj_reports ...
```

-R | --report_level

- Applies to: Compile and link

```bash
--report_level <arg>
```

Valid report levels: 0, 1, 2, estimate.

These report levels have mappings kept in the `optMap.xml` file. You can override the installed `optMap.xml` to define custom report levels.

- The `-R0` specification turns off all intermediate design checkpoint (DCP) generation during Vivado implementation. Turns on post-route timing report generation.

- The `-R1` specification includes everything from `-R0`, plus `report_failfast pre-opt_design`, `report_failfast post-opt_design`, and enables all intermediate DCP generation.

- The `-R2` specification includes everything from `-R1`, plus `report_failfast post-route_design`.

- The `-Restimate` specification forces Vitis HLS to generate a design.xml file if it does not exist and then generates a System Estimate report, as described in System Estimate Report.

  **TIP:** This option is useful for the software emulation build (`-t sw_emu`), when `design.xml` is not generated by default.

For example:

```bash
v++ -R2 ...
```

--reuse_bit

```bash
--reuse_bit <arg>
```

- Applies to: Link
Specifies the path and file name of generated bitstream file (.bit) to use when generating the device binary (xclbin) file. As described in Using -to_step and Launching Vivado Interactively, you can specify the --to_step option to interrupt the Vitis build process and manually place and route a synthesized design to generate the bitstream.

**IMPORTANT!** The --reuse_bit option is a sequential build option that requires you to use the same project directory when resuming the Vitis compiler with --reuse_bit that you specified when using --to_step to start the build.

For example:

```
v++ --link --reuse_bit ./project.bit
```

**--reuse_impl**

--reuse_impl <arg>

- **Applies to:** Link

Specifies the path and file name of an implemented design checkpoint (DCP) file to use when generating the device binary (XCLBIN) file. The link process uses the specified implemented DCP to extract the FPGA bitstream and generates the xclbin. You can manually edit the Vivado project created by a previously completed Vitis build, or specify the --to_step option to interrupt the Vitis build process and manually place and route a synthesized design, for instance. This allows you to work interactively with Vivado Design Suite to change the design and use DCP in the build process.

**IMPORTANT!** The --reuse_impl option is an incremental build option that requires you to use the same project directory when resuming the Vitis compiler with --reuse_impl that you specified when using --to_step to start the build.

For example:

```
v++ --link --reuse_impl ./manual_design.dcp
```

**-s | --save-temps**

- **Applies to:** Compile and link

--save-temps

Directs the v++ command to save intermediate files/directories created during the compilation and link process. Use the --temp_dir option to specify a location to write the intermediate files to.

**TIP:** This option is useful for debugging when you encounter issues in the build process.
For example:

```bash
v++ --save_temps ...
```

- **t | --target**

- Applies to: Compile and link

```
-t [ sw_emu | hw_emu | hw ]
```

Specifies the build target, as described in Build Targets. The build target determines the results of the compilation and linking processes. You can choose to build an emulation model for debug and test, or build the actual system to run in hardware. The build target defaults to hw if -t is not specified.

**IMPORTANT!** The specified platform and build targets for compiling and linking must match. The --platform and -t options specified when the XO file is generated by compilation must be the --platform and -t used during linking.

The valid values are:

- **sw_emu**: Software emulation
- **hw_emu**: Hardware emulation
- **hw**: Hardware

For example:

```bash
v++ --link -t hw_emu
```

**--temp_dir**

- Applies to: Compile and link

```
--temp_dir <dir_name>
```

This allows you to manage the location where the tool writes temporary files created during the build process. The temporary results are written by the v++ compiler, and then removed, unless the --save-temps option is also specified.

If --temp_dir is not specified, the tool saves the temporary files to ./_x/temp. Refer to Output Directories of the v++ Command for more information.

For example:

```bash
v++ --temp_dir /tmp/myProj_temp ...
```
--to_step

- Applies to: Compile and link

--to_step <arg>

Specifies a step name, for either the compile or link process, to run the build process through that step. You can use the --list_step option to determine the list of valid compile or link steps.

The build process will terminate after completing the named step. At this time, you can interact with the build results. For example, manually accessing the HLS project or the Vivado Design Suite project to perform specific tasks before returning to the build flow, launch the v++ command with the --from_step option.

IMPORTANT! The --to_step and --from_step options are incremental build options that require you to use the same project directory when launching the Vitis compiler using --from_step to resume the build as you specified when using --to_step to start the build.

You must also specify --save-temps when using --to_step to preserve the temporary files required by the Vivado tools. For example:

v++ --link --save-temps --to_step vpl.update_bd

--trace_memory

- Applies to: Compile and link

--trace_memory <arg>

Use with the --profile.xxx option as described in --profile Options when linking with hardware target, to specify the type and amount of memory to use for capturing trace data.

<FIFO>:<size>|<MEMORY>[<n>] specifies trace buffer memory type for profiling.

- FIFO:<size>: Specified in KB. Default is FIFO:8K. The maximum is 4G.
- Memory[<N>]: Specifies the type and number of memory resource on the platform. Memory resources for the target platform can be identified with the platforminfo command. Supported memory types include HBM, DDR, PLRAM, HP, ACP, MIG, and MC_NOC. For example, DDR[1].

IMPORTANT! When using --trace_memory during the linking step, you should also use the [Debug] trace_buffer_size in the xrt.ini file as described in xrt.ini File.

-v | --version

-v
Prints the version and build information for the `v++` command. For example:

```
$ v++ -v
```

---vivado

- **Applies to:** Link

Specify properties and parameters to configure the Vivado synthesis and implementation environment prior to building the device binary. See `-vivado Options` for more information.

---user_board_repo_paths

- **Applies to:** Compile and link

```
--user_board_repo_paths
```

Specifies an existing user board repository for DIMM board files. This value will be pre-pended to the `board_part_repo_paths` property of the Vivado project.

---user_ip_repo_paths

- **Applies to:** Compile and link

```
--user_ip_repo_paths <repo_dir>
```

Specifies the directory location of one or more user IP repository paths to be searched first for IP used in the kernel design. This value is appended to the start of the `ip_repo_paths` used by the Vivado tool to locate IP cores. IP definitions from these specified paths are used ahead of IP repositories from the hardware platform (.xsa) or from the Xilinx IP catalog.

**TIP:** Multiple `--user_ip_repo_paths` can be specified on the `v++` command line.

The following lists show the priority order in which IP definitions are found during the build process, from high to low. Note that all of these entries can possibly include multiple directories in them.

- **For the system hardware build (`-t hw`):**
  1. IP definitions from `--user_ip_repo_paths`.
  2. Kernel IP definitions (`vpl --iprepo switch value`).
  3. IP definitions from the IP repository associated with the platform.
  4. IP cache from the installation area (for example, `<Install_Dir>/Vitis/2019.2/data/cache/`).
  5. Xilinx IP catalog from the installation area (for example, `<Install_Dir>/Vitis/2019.2/data/ip/`.)
• For the hardware emulation build (-t hw_emu):
  1. IP definitions and User emulation IP repository from --user_ip_repo_paths.
  2. Kernel IP definitions (vpl --iprepo switch value).
  3. IP definitions from the IP repository associated with the platform.
  4. IP cache from the installation area (for example, <Install_Dir>/Vitis/2019.2/data/cache/).
  5. $::env(XILINX_VITIS)/data/emulation/hw_em/ip_repo
  6. $::env(XILINX_VIVADO)/data/emulation/hw_em/ip_repo
  7. Xilinx IP catalog from the installation area (for example, <Install_Dir>/Vitis/2019.2/data/ip/)

For example:

v++ --user_ip_repo_paths ./myIP_repo ...

--advanced Options

The --advanced.param and --advanced.prop options specify parameters and properties for use by the v++ command. When compiling or linking, these options offer fine-grain control over the hardware generated by the Vitis core development kit, and the hardware emulation process.

The arguments for the --advanced.xxx options are specified as <param_name>=<param_value>. For example:

v++ --link --advanced.param compiler.enableXSAIntegrityCheck=true
--advanced.prop kernel.foo.kernel_flags='-std=c++0x'

TIP: All Vitis compiler options can be specified in a configuration file for use with the --config option, as discussed in Vitis Compiler Configuration File. For example, the --platform option can be specified in a configuration file without a section head using the following syntax:

platform=xilinx_u200_xdma_201830_2

--advanced.param

--advanced.param <param_name>=<param_value>

Specifies advanced parameters as described in the table below.
Table 37: Param Options

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Valid Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>compiler.acceleratorBinaryContent</td>
<td>Type: String</td>
<td>Content to insert in xclbin. Valid options are <code>bitstream</code> and <code>dcp</code>. Applies to:</td>
</tr>
<tr>
<td></td>
<td>Default Value: &lt;empty&gt;</td>
<td>• v++ --link</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• vpl.impl</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• xclbinutil</td>
</tr>
<tr>
<td>compiler.addOutputTypes</td>
<td>Type: String</td>
<td>Additional output types produced by the Vitis compiler. Valid values include <code>xclbin</code> and <code>hw_export</code>. Use <code>hw_export</code> to create a fixed XSA from dynamic hardware platforms for use in the Embedded Software Development Flow.</td>
</tr>
<tr>
<td></td>
<td>Default Value: &lt;empty&gt;</td>
<td>Applies to:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• v++ --link</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• vpl.impl</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• XSA generation</td>
</tr>
<tr>
<td>compiler.deadlockDetection</td>
<td>Type: Boolean</td>
<td>Enables detection of kernel deadlocks during the simulation run as part of hardware emulation.</td>
</tr>
<tr>
<td></td>
<td>Default Value: TRUE</td>
<td>The tool posts an Error message to the console and the log file when the application is deadlocked:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>// ERROR!!! DEADLOCK DETECTED at 42979000 ns! SIMULATION WILL BE STOPPED! //</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The message is repeated until the deadlock is terminated. You must manually terminate the application to end the deadlock condition.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>TIP:</strong> When deadlocks are encountered during simulation, you can open the kernel code in Vitis HLS as described in <em>Compiling Kernels with the Vitis HLS</em> for additional deadlock detection and debug capability.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Applies to:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• v++ --compile</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Vitis HLS</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• config_export</td>
</tr>
<tr>
<td>compiler.enableIncrHwEmu</td>
<td>Type: Boolean</td>
<td>Use to enable incremental compilation of the hardware emulation <code>xclbin</code> when there are minor changes made to the platform. This enables a quick rebuild of the device binary for hardware emulation when the platform has been updated.</td>
</tr>
<tr>
<td></td>
<td>Default Value: FALSE</td>
<td>Applies to:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• v++ --link</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• vpl.impl</td>
</tr>
<tr>
<td>Parameter Name</td>
<td>Valid Values</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------------</td>
<td>--------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>compiler.errorOnHoldViolation</td>
<td>Type: Boolean Default Value: TRUE</td>
<td>After the last step of Vivado implementation, during timing analysis check, and clock scaling if needed. If hold violations are found, v++ quits and returns an error by default, and does not generate an xclbin. This parameter lets you over ride the default behavior. Applies to: • v++ --link • vpl.impl</td>
</tr>
<tr>
<td>compiler.fsanitize</td>
<td>Type: String Default Value: &lt;empty&gt;</td>
<td>Enables additional memory access checks for OpenCL kernels as described in Debugging OpenCL Kernels. Valid values include: address, memory. Applies to Software Emulation and Debug.</td>
</tr>
<tr>
<td>compiler.interfaceRdBurstLen</td>
<td>Type: Int Range Default Value: 0</td>
<td>Specifies the expected length of AXI read bursts on the kernel AXI interface. This is used with option compiler.interfaceRdOutstanding to determine the hardware buffer sizes. Values are 1 through 256. Applies to: • v++ --compile • Vitis HLS • config_interface</td>
</tr>
<tr>
<td>compiler.interfaceWrBurstLen</td>
<td>Type: Int Range Default Value: 0</td>
<td>Specifies the expected length of AXI write bursts on the kernel AXI interface. This is used with option compiler.interfaceWrOutstanding to determine the hardware buffer sizes. Values are 1 through 256. Applies to: • v++ --compile • Vitis HLS • config_interface</td>
</tr>
<tr>
<td>compiler.interfaceRdOutstanding</td>
<td>Type: Int Range Default Value: 0</td>
<td>Specifies how many outstanding reads to buffer are on the kernel AXI interface. Values are 1 through 256. Applies to: • v++ --compile • Vitis HLS • config_interface</td>
</tr>
<tr>
<td>compiler.interfaceWrOutstanding</td>
<td>Type: Int Range Default Value: 0</td>
<td>Specifies how many outstanding writes to buffer are on the kernel AXI interface. Values are 1 through 256. Applies to: • v++ --compile • Vitis HLS • config_interface</td>
</tr>
<tr>
<td>Parameter Name</td>
<td>Valid Values</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------------------------</td>
<td>--------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>compiler.maxComputeUnits</td>
<td>Type: Int</td>
<td>Maximum compute units allowed in the system. The default is 60 compute units, or is specified in the hardware platform (.xsa) with the numComputeUnits property. The specified value will override the default value or the hardware platform. The default value of -1 preserves the default. Applies to v++ --link.</td>
</tr>
<tr>
<td>compiler.skipTimingCheckAndFrequencyScaling</td>
<td>Type: Boolean Default Value: FALSE</td>
<td>This parameter causes the Vivado tool to skip the timing check and optional clock frequency scaling that occurs after the last step of implementation process, which is either route_design or post-route phys_opt_design. Applies to: • v++ --link • vpl.impl</td>
</tr>
<tr>
<td>compiler.userPreCreateProjectTcl</td>
<td>Type: String Default Value: &lt;empty&gt;</td>
<td>Specifies a Tcl script to run before creating the Vivado project in the Vitis build process. Applies to: • v++ --link • vpl.create_project</td>
</tr>
<tr>
<td>compiler.userPreSysLinkOverlayTcl</td>
<td>Type: String Default Value: &lt;empty&gt;</td>
<td>Specifies a Tcl script to run after opening the Vivado IP Integrator block design, before running the compiler-generated dr.bd.tcl script in the Vitis build process. Applies to: • v++ --link • vpl.create_bd</td>
</tr>
<tr>
<td>compiler.userPostSysLinkOverlayTcl</td>
<td>Type: String Default Value: &lt;empty&gt;</td>
<td>Specifies a Tcl script to run after running the compiler-generated dr.bd.tcl script. Applies to: • v++ --link • vpl.update_bd</td>
</tr>
<tr>
<td>compiler.userPostDebugProfileOverlayTcl</td>
<td>Type: String Default Value: &lt;empty&gt;</td>
<td>Specifies a Tcl script to run after debug profile overlay insertion in Vivado IP integrator block design in the vpl.update_bd step. Applies to: • v++ --link • vpl.updated_bd</td>
</tr>
<tr>
<td>Parameter Name</td>
<td>Valid Values</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------------------</td>
<td>--------------------------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>compiler.worstNegativeSlack</td>
<td>Type: Float</td>
<td>During timing analysis check, this specifies the worst acceptable negative slack for the design, specified in nanoseconds (ns). When negative slack exceeds the specified value, the tool might try to scale the clock frequency to achieve timing results. This specifies an acceptable negative slack value instead of zero slack. Applies to: • v++ --link • vpl.impl</td>
</tr>
<tr>
<td>compiler.xclDataflowFifoDepth</td>
<td>Type: Int</td>
<td>Specifies the depth of FIFOs used in kernel data flow region. Applies to: • v++ --compile • Vitis HLS • config_dateflow</td>
</tr>
<tr>
<td>hw_emu.compiledLibs</td>
<td>Type: String</td>
<td>Uses mentioned clibs for the specified simulator. Applies to Hardware Emulation and Debug.</td>
</tr>
<tr>
<td>hw_emu.debugMode</td>
<td>gdb</td>
<td>wdb</td>
</tr>
<tr>
<td>hw_emu.enableProtocolChecker</td>
<td>Type: Boolean</td>
<td>Enables the lightweight AXI protocol checker (lapc) during HW emulation. This is used to confirm the accuracy of any AXI interfaces in the design. Applies to Hardware Emulation and Debug.</td>
</tr>
<tr>
<td>hw_emu.platformPath</td>
<td>Type: String</td>
<td>Specifies the path to the custom platform directory. The &lt;platformPath&gt; directory should meet the following requirements to be used in platform creation: • The directory should contain a subdirectory called ip_repo. • The directory should contain a subdirectory called scripts and this scripts directory should contain a hw_em_util.tcl file. The hw_em_util.tcl file should have the following two procedures defined in it: ◦ hw_em_util::add_base_platform ◦ hw_em_util::generate_simulation_scripts_and_compile Applies to Hardware Emulation and Debug.</td>
</tr>
</tbody>
</table>
Table 37: Param Options (cont’d)

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Valid Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hw_emu.post_sim_settings</td>
<td>Type: String</td>
<td>Specifies the path to a Tcl script that is used to configure the settings of the Vivado simulator prior to running hardware emulation. This script is run after the default configuration of the tool, but prior to launching simulation. You can use the Tcl script to override specific settings, or to custom configure the simulator as needed. Applies to Hardware Emulation and Debug.</td>
</tr>
</tbody>
</table>
| hw_emu.scDebugLevel                | none | waveform | log | waveform_and_log Default Value: waveform_and_log | Sets the TLM transaction debug level of the Vivado logic simulator (xsim).  
• NONE to disable TLM debug  
• LOG to dump TLM transaction log info into report file  
• WAVEFORM for enabling the TLM transaction waveform view  
• WAVEFORM_AND_LOG for both the Log Messages and Waveform view  
Applies to Hardware Emulation and Debug. |
| hw_emu.simulator                   | XSIM | QUESTA Default Value: XSIM | Uses the specified simulator for the hardware emulation run.  
Applies to Hardware Emulation and Debug. |

For example:

```
--advanced.param compiler.addOutputTypes="hw_export"
```

**TIP:** This option can be specified in a configuration file under the {advanced} section head using the following format:

```
[advanced]
param=compiler.addOutputTypes="hw_export"
```

**--advanced.prop**

```
--advanced.prop <arg>
```

Specifies advanced kernel or solution properties for kernel compilation where <arg> is one of the values described in the table below.

Table 38: Prop Options

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Valid Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>kernel.&lt;kernel_name&gt;.kernel_flags</td>
<td>Type: String Default Value: &lt;empty&gt;</td>
<td>Sets specific compile flags on the kernel &lt;kernel_name&gt;.</td>
</tr>
</tbody>
</table>
Table 38: Prop Options (cont'd)

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Valid Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>solution.device_repo_path</td>
<td>Type: String</td>
<td>Specifies the path to a repository of hardware platforms. The --platform option with full path to the .xpfm platform file should be used instead.</td>
</tr>
<tr>
<td></td>
<td>Default Value: &lt;empty&gt;</td>
<td></td>
</tr>
<tr>
<td>solution.kernel_compiler_margin</td>
<td>Type: Float</td>
<td>The clock margin (in ns) for the kernel. This value is subtracted from the kernel clock period prior to synthesis to provide some margin for place and route delays.</td>
</tr>
<tr>
<td></td>
<td>Default Value: 12.5% of the kernel clock period.</td>
<td></td>
</tr>
</tbody>
</table>

--advanced.misc

--advanced.misc <arg>

Specifies advanced tool directives for kernel compilation.

--clock Options

IMPORTANT! The --clock options described here are supported on embedded processor platforms and newer platforms for Data Center accelerator cards. On older legacy platforms, clock frequency for kernels can be specified using the --kernel_frequency option.

The --clock.XXX options provide a method for assigning clocks to kernels from the v++ command line and locating the required kernel clock frequency source during the linking process. There are a number of options that can be used with increasing specificity. The order of precedence is determined by how specific a clock option is. The rules are listed in order from general to specific, where the specific rules take precedence over the general rules:

- When no --clock.XX option is specified, the platform default clock will be applied to each compute unit (CU). For kernels with two clocks, clock ID 0 from the platform will be assigned to ap_clk, and clock ID 1 will be assigned to ap_clk_2.
- Specifying --clock.defaultId=<id> defines a specific clock ID for all kernels, overriding the platform default clock assignments.
- Specifying --clock.defaultFreqHz=<Hz> defines a specific clock frequency for all kernels that overrides a user specified default clock ID, and the platform default clock.
- Specifying --clock.id=<id>:<cu_0>[.<clk_pin_0>][.<cu_n>[.<clk_pin_n>]] assigns a clock ID to a list of associated CUs, and optionally the clock pin for the CU.
- Specifying --clock.freqHz=<Hz>:<cu_0>[.<clk_pin_0>] [<cu_n>[.<clk_pin_n>]] assigns the specified clock frequency to the specified clock pin on the specified CU.
**--clock.defaultFreqHz**

```shell
--clock.defaultFreqHz <arg>
```

Specifies a default clock frequency in Hz to use for all kernels. This lets you override the default platform clock, and assign the clock with the specified clock frequency as the default. Where `<arg>` is specified as the clock frequency in Hz.

For example:

```shell
v++ --link --clock.defaultFreqHz 300000000
```

**TIP:** This option can be specified in a configuration file under the `clock` section head using the following format:

```
[clock]
defaultFreqHz=300000000
```

**--clock.defaultId**

```shell
--clock.defaultId <arg>
```

Specifying `--clock.defaultId=<id>` defines a specific clock ID for all kernels, overriding the platform default clock. Where `<arg>` is specified as the clock ID from one of the clocks defined on the target platform, other than the default clock ID.

**TIP:** You can determine the available clock IDs for a target platform using the `platforminfo` utility as described in platforminfo Utility.

For example:

```shell
v++ --link --clock.defaultId 1
```

**TIP:** This option can be specified in a configuration file under the `clock` section head using the following format:

```
[clock]
defaultId=1
```

**--clock.defaultTolerance**

```shell
--clock.defaultTolerance <arg>
```

Specifies a default clock tolerance as a value, or as a percentage of the default clock frequency. When specifying `clock.defaultFreqHz`, you can also specify the tolerance with either a value or percentage. This will update timing constraints to reflect the accepted tolerance.
The tolerance value, \(<\text{arg}>\), can be specified as a whole number, indicating the \(\text{clock.defaultFreqHz} \pm \text{the specified tolerance}\); or as a percentage of the default clock frequency specified as a decimal value.

**IMPORTANT!** The default clock tolerance is 5% when this option is not specified.

For example:

```bash
v++ --link --clock.defaultFreqHz 300000000 --clock.defaultTolerance 0.10
```

**TIP:** This option can be specified in a configuration file under the `clock` section head using the following format:

```ini
[clock]
defaultTolerance=0.10
```

---

**--clock.freqHz**

```
--clock.freqHz <arg>
```

Specifies a clock frequency in Hz and assigns it to a list of associated compute units (CUs) and optionally specific clock pins on the CU. Where \(<\text{arg}>\) is specified as

```
<frequency_in_Hz>:<cu_0>[.<clk_pin_0>],[<cu_n>[.<clk_pin_n>]]:
```

- \(<\text{frequency_in_Hz}>\): Defines the clock frequency specified in Hz.
- \(<\text{cu_0}>[.<clk_pin_0>][,<\text{cu_n}>[.<clk_pin_n>]]\): Applies the defined frequency to the specified CUs, and optionally to the specified clock pin on the CU.

For example:

```bash
v++ --link --clock.freqHz 300000000:vadd_1,vadd_3
```

**TIP:** This option can be specified in a configuration file under the `clock` section head using the following format:

```ini
[clock]
freqHz=300000000:vadd_1,vadd_3
```

---

**--clock.id**

```
--clock.id <arg>
```

Specifies an available clock ID from the target platform and assigns it to a list of associated compute units (CUs) and optionally specific clock pins on the CU. Where \(<\text{arg}>\) is specified as

```
<reference_ID>:<cu_0>[.<clk_pin_0>],[<cu_n>[.<clk_pin_n>]]:
```

- \(<\text{reference_ID}>\): Defines the clock ID to use from the target platform.
You can determine the available clock IDs for a target platform using the platforminfo utility as described in platforminfo Utility.

- `<cu_0>[.<clk_pin_0>],[<cu_n>[.<clk_pin_n>]]`: Applies the defined frequency to the specified CUs and optionally to the specified clock pin on the CU.

For example:

```
v++ --link --clock.id 1:vadd_1,vadd_3
```

**TIP:** This option can be specified in a configuration file under the `[clock]` section head using the following format:

```
[clock]
id=1:vadd_1,vadd_3
```

---

**--clock.tolerance**

`--clock.tolerance <arg>`

Specifies a clock tolerance as a value, or as a percentage of the clock frequency. When specifying `--clock.freqHz`, you can also specify the tolerance with either a value or percentage. This will update timing constraints to reflect the accepted tolerance. Where `<arg>` is specified as `<tolerance>::<cu_0>[.<clk_pin_0>],[<cu_n>[.<clk_pin_n>]]`

- `<tolerance>`: Can be specified either as a whole number, indicating the `clock.freqHz ±` the specified tolerance value; or as a percentage of the clock frequency specified as a decimal value.
- `<cu_0>[.<clk_pin_0>],[<cu_n>[.<clk_pin_n>]]`: Applies the defined clock tolerance to the specified CUs, and optionally to the specified clock pin on the CU.

**IMPORTANT!** The default clock tolerance is 5% of the `clock.FreqHz` when this option is not specified.

For example:

```
v++ --link --clock.tolerance 0.10:vadd_1,vadd_3
```

**TIP:** This option can be specified in a configuration file under the `[clock]` section head using the following format:

```
[clock]
tolerance=0.10:vadd_1,vadd_3
```
--connectivity Options

As discussed in Linking the Kernels, there are a number of --connectivity.XXX options that let you define the topology of the FPGA binary, specifying the number of CUs, assigning them to SLRs, connecting kernel ports to global memory, and establishing streaming port connections. These commands are an integral part of the build process, critical to the definition and construction of the application.

--connectivity.nk

--connectivity.nk <arg>

Where <arg> is specified as
<kernel_name>:#:<cu_name1>.<cu_name2>...<cu_name#>.

This instantiates the specified number of CU (#) for the specified kernel (kernel_name) in the generated FPGA binary (.xclbin) file during the linking process. The cu_name is optional. If the cu_name is not specified, the instances of the kernel are simply numbered: kernel_name_1, kernel_name_2, and so forth. By default, the Vitis compiler instantiates one compute unit for each kernel.

For example:

v++ --link --connectivity.nk vadd:3:vadd_A.vadd_B.vadd_C

TIP: This option can be specified in a configuration file under the [connectivity] section head using the following format:

[connectivity]
nk=vadd:3:vadd_A.vadd_B.vadd_C

--connectivity.slr

--connectivity.slr <arg>

Use this option to assign a CU to a specific SLR on the device. The option must be repeated for each kernel or CU being assigned to an SLR.

IMPORTANT! If you use --connectivity.slr to assign the kernel placement, then you must also use --connectivity.sp to assign memory access for the kernel.

Valid values include:

<cu_name>:<SLR_NUM>

Where:
• <cu_name> is the name of the compute unit as specified in the --connectivity.nk option. Generally this will be <kernel_name>_1 unless a different name was specified.

• <SLR_NUM> is the SLR number to assign the CU to. For example, SLR0, SLR1.

For example, to assign CU vadd_2 to SLR2, and CU fft_1 to SLR1, use the following:

```
v++ --link --connectivity.slr vadd_2:SLR2 --connectivity.slr fft_1:SLR1
```

**TIP:** This option can be specified in a configuration file under the [connectivity] section head using the following format:

```
[connectivity]
slr=vadd_2:SLR2
slr=fft_1:SLR1
```

---

**--connectivity.sp**

```
--connectivity.sp <arg>
```

Use this option to specify the assignment of kernel interfaces to system ports within the platform. A primary use case for this option is to connect kernel ports to specific memory resources. A separate --connectivity.sp option is required to map each interface of a kernel to a particular memory resource. Any kernel interface not explicitly mapped to a memory resource through the --connectivity.sp option will be automatically connected to an available memory resource during the build process.

Valid values include:

```
<cu_name>.<kernel_interface_name>:<sptag[min:max]>
```

Where:

• <cu_name> is the name of the compute unit as specified in the --connectivity.nk option. Generally this will be <kernel_name>_1 unless a different name was specified.

• <kernel_interface_name> is the name of the function argument for the kernel, or compute unit port.

• <sptag> represents a system port tag, such as for memory controller interface names from the target platform. Valid <sptag> names include DDR, PLRAM, and HBM.

• [min:max] enables the use of a range of memory, such as DDR[0:2]. A single index is also supported: DDR[2].

**TIP:** The supported <sptag> and range of memory resources for a target platform can be obtained using the platforminfo command. Refer to platforminfo Utility for more information.
The following example maps the input argument (A) for the specified CU of the VADD kernel to DDR[0:3], input argument (B) to HBM[0:31], and writes the output argument (C) to PLRAM[2]:

```
```

**TIP:** This option can be specified in a configuration file under the `[connectivity]` section head using the following format:

```
[connectivity]
sp=vadd_1.A:DDR[0:3]
sp=vadd_1.B:HBM[0:31]
sp=vadd_1.C:PLRAM[2]
```

---

**--connectivity.sc**

```
--connectivity.sc <arg>
```

Create a streaming connection between two compute units through their AXI4-Stream interfaces. Use a separate `--connectivity.sc` option for each streaming interface connection. The order of connection must be from a streaming output port of the first kernel to a streaming input port of the second kernel. Valid values include:

```
<cu_name>.<streaming_output_port>:<cu_name>.<streaming_input_port>[[:<fifo_depth>]]
```

Where:

- `<cu_name>` is the compute unit name specified in the `--connectivity.nk` option. Generally this will be `<kernel_name>_1` unless a different name was specified.
- `<streaming_output_port>/<streaming_input_port>` is the function argument for the compute unit port that is declared as an AXI4-Stream.
- `[:<fifo_depth>]` inserts a FIFO of the specified depth between the two streaming ports to prevent stalls. The value is specified as an integer.

For example, to connect the AXI4-Stream port `s_out` of the compute unit `mem_read_1` to AXI4-Stream port `s_in` of the compute unit `increment_1`, use the following:

```
--connectivity.sc mem_read_1.s_out:increment_1.s_in
```

**TIP:** This option can be specified in a configuration file under the `[connectivity]` section head using the following format:

```
[connectivity]
sc=mem_read_1.s_out:increment_1.s_in
```
The inclusion of the optional `<fifo_depth>` value lets the v++ linker add a FIFO between the two kernels to help prevent stalls. This will use BRAM resources from the device when specified, but eliminates the need to update the HLS kernel to contain FIFOs. The tool will also instantiate a Clock Converter (CDC) or Datawidth Converter (DWC) IP if the connections have different clocks, or different bus widths.

--debug Options

This option enables debug IP core insertion in the device binary (.xclbin) for hardware debugging. This option lets you specify the type of debug core to add, and which compute unit and interfaces to monitor with ChipScope™. The --debug.xxx options lets you attach AXI protocol checkers and System ILA cores at the interfaces to kernels or specific compute units (CUs) for debugging and performance monitoring purposes:

- The System Integrated Logic Analyzer (ILA) provides transaction level visibility into an accelerated kernel or function running on hardware. AXI traffic of interest can also be captured and viewed using the System ILA core.
- The AXI Protocol Checker debug core is designed to monitor AXI interfaces on the accelerated kernel. When attached to an interface of a CU, it actively checks for protocol violations and provides an indication of which violation occurred.

The --debug.xxx commands can be specified in a configuration file under the [debug] section head using the following format as an example:

```
[debug]
protocol=all:all           # Protocol analyzers on all CUs
protocol=cu2:port3         # Protocol analyzer on port3 of cu2
chipscope=cu2              # ILA on cu2
```

The various options of --debug include the following:

--debug.chipscope

```
--debug.chipscope <cu_name>[:<interface_name>]
```

Adds the System Integrated Logic Analyzer debug core to the specified CUs in the design.

**IMPORTANT!** The --debug.chipscope option requires the `<cu_name>` to be specified and does not accept the keyword `all`. You can optionally specify an `<interface_name>`.

For example, the following command adds an ILA core to the vadd_1 CU:

```
v++ --link --debug.chipscope vadd_1
```
--debug.list_ports

Shows a list of valid compute units and port combinations in the current design. This is informational to help you with crafting a command line or config file for the --debug command.

This option needs to be specified during linking, but does not run the linking process. The required elements of the command line are shown in the following example, which returns the available ports when linking the specified kernels with the listed platform:

v++ --platform <platform> --link --debug.list_ports <kernel.xo>

--debug.protocol

--debug.protocol all|<cu_name>|[:<interface_name>]

Adds the AXI Protocol Checker debug core to the design. This can be specified with the keyword all, or the <cu_name> and optional <interface_name> to add the protocol checker to the specified CU and interface.

For example:

v++ --link --debug.protocol all

--hls Options

The --hls.XXX options described below are used to specify options for the Vitis HLS synthesis process invoked during kernel compilation.

--hls.clock

--hls.clock <arg>

Specifies a frequency in Hz at which the listed kernel(s) should be compiled by Vitis HLS.

Where <arg> is specified as:

<frequency_in_Hz>:<cu_name1>,<cu_name2>,...,<cu_nameN>

- <frequency_in_Hz>: Defines the kernel frequency specified in Hz.
- <cu_name1>,<cu_name2>,...: Defines a list of kernels or kernel instances (CUs) to be compiled at the specified target frequency.

For example:

v++ -c --hls.clock 300000000:mmult,mmadd --hls.clock 100000000:fifo_1
**TIP:** This option can be specified in a configuration file under the `[hls]` section head using the following format:

```
[hls]
clock=300000000:mmult,mmadd
clock=100000000:fifo_1
```

---

**--hls.export_project**

```
--hls.export_project
```

Specifies a directory where the HLS project setup script is exported.

For example:

```
v++ --hls.export_project ./hls_export
```

**TIP:** This option can be specified in a configuration file under the `[hls]` section head using the following format:

```
[hls]
export_project=./hls_export
```

---

**--hls.jobs**

```
--hls.jobs <arg>
```

Specifies the number of jobs for launching HLS runs.

This option specifies the number of parallel jobs Vitis HLS uses to synthesize the RTL kernel code. Increasing the number of jobs allows the tool to spawn more parallel processes and complete faster.

For example:

```
v++ --hls.jobs 4
```

**TIP:** This option can be specified in a configuration file under the `[hls]` section head using the following format:

```
[hls]
jobs=4
```

---

**--hls.lsf**

```
--hls.lsf <arg>
```

Specifies a `bsub` command to submit a job to LSF for HLS runs.
Specifies the `bsub` command line as a string to pass to an LSF cluster. This option is required to use the IBM Platform Load Sharing Facility (LSF) for Vitis HLS synthesis.

For example:

```
v++ --compile --hls.lsf '{bsub -R "select[type=X86_64]\" -N -q medium}''
```

**TIP:** This option can be specified in a configuration file under the `[hls]` section head using the following format:

```
[hls]
lsf='[bsub...
```

`--hls.post_tcl`

`--hls.post_tcl <arg>`

Specifies a Tcl file containing Tcl commands for `vitis_hls` to source after `csynth_design`.

For example:

```
v++ --hls.post_tcl ./runPost.tcl
```

**TIP:** This option can be specified in a configuration file under the `[hls]` section head using the following format:

```
[hls]
post_tcl=./runPost.tcl
```

`--hls.pre_tcl`

`--hls.pre_tcl <arg>`

Specifies a Tcl file containing Tcl commands for `vitis_hls` to source before running `csynth_design`.

For example:

```
v++ --hls.pre_tcl ./runPre.tcl
```

Where `runPre.tcl` contains the following commands to configure `m_axi` interfaces in Vitis HLS:

```
config_interface -m_axi_auto_max_ports=1
config_interface -m_axi_max_bitwidth 512
```
--linkhook Options

The --linkhook.XXX options described below are used to specify Tcl scripts to run at specific steps during the Vitis linking process. Valid steps can be determined using the --linkhook.list_steps command as described below.

--linkhook.custom

--linkhook.custom <step name, path to script file>

Specifies a Tcl script to execute at a predefined point in the build process. The path to specify the script can be an absolute path, or partial path relative to the build directory.

For example, the following command runs the specified Tcl script before the SysLink step in the build:

```
v++ -l --linkhook.custom preSysLink,./runScript.tcl
```

--linkhook.do_first

--linkhook.do_first <step name, path to script file>

Specifies a Tcl script to execute before the given step name. The path to specify the script can be an absolute path, or partial path relative to the build directory.

For example, the following command runs the specified Tcl script before the place_design step in the build:

```
v++ -l --linkhook.do_first vpl.impl.place_design,runScript.tcl
```

--linkhook.do_last

--linkhook.do_last <step name, path to script file>

Specifies a Tcl script to execute immediately after the given step completes. The path to specify the script can be an absolute path, or partial path relative to the build directory.
For example, the following command runs the specified Tcl script after the `place_design` step in the build:

\[ v++ -l --linkhook.do_last vpl.impl.place_design,runScript.tcl \]

`--linkhook.list_steps`

List default and optional build steps that support script hooks for a specified target. This command requires the `--target` to be specified as well as the `--link` option.

For example:

\[ v++ --target hw -l --linkhook.list_steps \]

The command returns both default steps that are always enabled during the build process, and optional steps that you can enable if needed. Refer to Managing Vivado Synthesis and Implementation Results for directions on enabling optional steps.

--package Options

**Introduction**

The `v++ --package`, or `-p` step, packages the final product at the end of the `v++` compile and link build process. This is a required step for all embedded platforms, including Versal devices, AI Engine, and Zynq devices.

The various options of `--package` include the following:

**--package.aie_debug_port**

`--package.aie_debug_port <arg>`

Where `<arg>` specifies a TCP port where emulator will listen for incoming connections from the debugger to debug Versal AI Engine cores.

For example:

\[ v++ -l --package.aie_debug_port 1440 \]

**--package.bl31_elf**

`--package.bl31_elf <arg>`
Where `<arg>` specifies the absolute or relative path to Arm trusted FW ELF that will execute on A72 #0 core.

For example:

```
v++ -l --package.bl31_elf ./arm_trusted.elf
```

`--package.boot_mode`

```
--package.boot_mode <arg>
```

Where `<arg>` specifies `<ospi | qspi | sd>` Boot mode used for running the application in emulation or on hardware.

For example:

```
v++ -l --package.boot_mode sd
```

`--package.defer_aie_run`

```
--package.defer_aie_run
```

Where this option specifies that the Versal AI Engine cores will be enabled by an embedded processor (PS) application. When not specified, the tool will generate CDO commands to enable the AI Engine cores during PDI load instead.

For example:

```
v++ -l --package.defer_aie_run
```

`--package.domain`

```
--package.domain <arg>
```

Where `<arg>` specifies a domain name.

For example:

```
v++ -l --package.domain xrt
```

`--package.dtb`

```
--package.dtb <arg>
```

Where `<arg>` specifies the absolute or relative path to device tree binary (DTB) used for loading Linux on the APU.
For example:

```
v++ -l --package.dtb ./device_tree.image
```

**--package.enable_aie_debug**

```
--package.enable_aie_debug
```

When enabled, the tool generates CDO commands to stop the AI Engine cores during PDI load.

For example:

```
v++ -l --package.enable_aie_debug
```

**--package.image_format**

```
--package.image_format <arg>
```

Where `<arg>` specifies `<ext4 | fat32>` output image file format.

- `ext4`: Linux file system
- `fat32`: Windows file system

---

**IMPORTANT!** `EXT4` format is not supported on Windows.

For example:

```
v++ -l --package.image_format fat32
```

**--package.kernel_image**

```
--package.kernel_image <arg>
```

Where `<arg>` specifies the absolute or relative path to a Linux kernel image file. Overrides the existing image available in the platform. The platform image file is available for download from xilinx.com. Refer to the Vitis Software Platform Installation for more information.

For example:

```
v++ -l --package.kernel_image ./kernel_image
```

**--package.no_image**

```
--package.no_image
```

Bypass SD card image creation. Valid for **--package.boot_mode sd.**
--package.out_dir

--package.out_dir <arg>

Where <arg> specifies the absolute or relative path to the output directory of the --package command.

For example:

v++ -l --package.out_dir ./out_dir

--package.ps_debug_port

--package.ps_debug_port <arg>

Where <arg> specifies the TCP port where emulator will listen for incoming connections from the debugger to debug PS cores.

For example:

v++ -l --package.debug_port 3200

--package.ps_elf

--package.ps_elf <arg>

Where <arg> specifies <ps.elf,core>.

- ps.elf: Specifies the ELF file for the PS core.
- core: Indicates the PS core it should run on.

For example:

v++ -l --package.ps_elf a72_0.elf,a72-0

--package.rootfs

--package.rootfs <arg>

Where <arg> specifies the absolute or relative path to a processed Linux root file system file. The platform RootFS file is available for download from Xilinx.com. Refer to the Vitis Software Platform Installation for more information.

For example:

v++ -l --package.rootfs ./rootfs.ext4
--package.sd_dir

```bash
--package.sd_dir <arg>
```

Where `<arg>` specifies a folder to package into the `sd_card` directory/image. The contents of the directory are copied to a sub-folder of the `sd_card` folder.

For example:

```bash
v++ -l --package.sd_dir ./test_data
```

--package.sd_file

```bash
--package.sd_file <arg>
```

Where `<arg>` specifies an ELF or other data file to package into the `sd_card` directory/image. This option can be used repeatedly to specify multiple files to add to the `sd_card`.

For example:

```bash
v++ -l --package.sd_file ./arm_trusted.elf
```

--package.uboot

```bash
--package.uboot <arg>
```

Where `<arg>` specifies a path to U-boot ELF file which overrides a platform U-boot.

For example:

```bash
v++ -l --package.uboot ./uboot.elf
```

--profile Options

As discussed in [Enabling Profiling in Your Application](#), there are a number of `--profile` options that let you enable profiling of the application and kernel events during runtime execution. This option enables capturing profile data for data traffic between the kernel and host, kernel stalls, the execution times of kernels and compute units (CUs), as well as monitoring activity in Versal AI Engines.

**IMPORTANT!** Using the `--profile` option in `v++` also requires the addition of the `profile=true` statement to the `xrt.ini` file. Refer to [xrt.ini File](#).
The --profile commands can be specified in a configuration file under the [profile] section head using the following format, for example:

```
[profile]
data=all:all:all           # Monitor data on all kernels and CUs
data=k1:all:all             # Monitor data on all instances of kernel k1
data=k1:cu2:port3           # Specific CU master
data=k1:cu2:port3:counters  # Specific CU master (counters only, no trace)
stall=all:all               # Monitor stalls for all CUs of all kernels
stall=k1:cu2                # Stalls only for cu2
exec=all:all                # Monitor execution times for all CUs
exec=k1:cu2                 # Execution times only for cu2
aie=all                     # Monitor all AIE streams
aie=DataIn1                 # Monitor the specific input stream in the SDF
graph
aie=M02_AXIS                # Monitor specific stream interface
```

The various options of the command are described below:

**--profile.aie <arg>**

Enables profiling of AI Engine streams in adaptive data flow (ADF) applications, where `<arg>` is:

```
<ADF_graph_argument|pin name|all>
```

- `<ADF_graph_argument>`: Specifies an argument name from the ADF graph application.
- `<pin_name>`: Indicates a port on an AI Engine kernel.
- `<all>`: Indicates monitoring all stream connections in the ADF application.

For example, to monitor the DataIn1 input stream use the following command:

```
v++ --link --profile.aie:DataIn1
```

**--profile.data <arg>**

Enables monitoring of data ports through the monitor IPs. This option needs to be specified during linking.

Where `<arg>` is:

```
[[<kernel_name>|all]:[<cu_name>|all]:[<interface_name>|all]:[counters|all]]
```

- `[<kernel_name>|all]` defines either a specific kernel to apply the command to. However, you can also specify the keyword all to apply the monitoring to all existing kernels, compute units, and interfaces with a single option.
- `[<cu_name>|all]` when `<kernel_name>` has been specified, you can also define a specific CU to apply the command to, or indicate that it should be applied to all CUs for the kernel.
• `<interface_name>|all>` defines the specific interface on the kernel or CU to monitor for data activity, or monitor all interfaces.

• `<counters|all>` is an optional argument, as it defaults to `all` when not specified. It allows you to restrict the information gathering to just counters for larger designs, while `all` will include the collection of actual trace information.

For example, to assign the data profile to all CUs and interfaces of kernel `k1` use the following command:

```
v++ --link --profile.data:k1:all:all
```

**--profile.exec <arg>**

This option records the execution times of the kernel and provides minimum port data collection during the system run. This option needs to be specified during linking.

*TIP: The execution time of a kernel is collected by default when `--profile.data` or `--profile.stall` is specified. You can specify `--profile.exec` for any CUs not covered by data or stall.*

The syntax for `exec` profiling is:

```
<kernl_name>|all|:|<cu_name>|all|):|<counters|all>
```

For example, to profile to execution of `cu2` for kernel `k1` use the following command:

```
v++ --link --profile.exec:k1:cu2
```

**--profile.stall**

Adds stall monitoring logic to the device binary (.xclbin) which requires the addition of stall ports on the kernel interface. To facilitate this, the `stall` option must be specified during both compilation and linking.

The syntax for `stall` profiling is:

```
<kernl_name>|all|:|<cu_name>|all|):|<counters|all>
```

For example, to monitor stalls of `cu2` for kernel `k1` use the following command:

```
v++ --compile -k k1 --profile.stall ...
v++ --link --profile.stall:k1:cu2 ...
```
--profile.trace_memory

When building the hardware target (-t=hw), use this option to specify the type and amount of memory to use for capturing trace data. You can specify the argument as follows:

```
<FIFO>:<size>|<MEMORY>[<n>]
```

This argument specifies trace buffer memory type for profiling.

- **FIFO:<size>**: Specified in KB. Default is FIFO:8K. The maximum is 4G.
- **Memory[<N>]**: Specifies the type and number of memory resource on the platform. Memory resources for the target platform can be identified with the `platforminfo` command. Supported memory types include HBM, DDR, PLRAM, HP, ACP, MIG, and MC_NOC. For example, DDR[1].

**IMPORTANT!** Use with `[Debug] trace_buffer_size` in the xrt.ini file as described in `xrt.ini File`.

---

--vivado Options

The `--vivado.XXX` options are used to configure the Vivado tools for synthesis and implementation of your device binary (.xclbin). For instance, you can specify the number of jobs to spawn, LSF commands to use for implementation runs, or the specific implementation strategies to use. You can also configure optimization, placement, timing, or specify which reports to output.

**IMPORTANT!** Familiarity with the Vivado Design Suite is required to make the best use of these options. See the Vivado Design Suite User Guide: Implementation (UG904) for more information.

--vivado.impl.jobs

```
--vivado.impl.jobs <arg>
```

Specifies the number of parallel jobs the Vivado Design Suite uses to implement the device binary. Increasing the number of jobs allows the Vivado implementation step to spawn more parallel processes and complete faster jobs.

For example:

```
v++ --link --vivado.impl.jobs 4
```

--vivado.impl.lsf

```
--vivado.impl.lsf <arg>
```

Specifies the `bsub` command line as a string to pass to an LSF cluster. This option is required to use the IBM Platform Load Sharing Facility (LSF) for Vivado implementation.

For example:

```bash
v++ --link --vivado.impl.lsf '{bsub -R \'select[type=X86_64]\' -N -q medium}'
```

**--vivado.impl.strategies**

```bash
--vivado.impl.strategies <arg>
```

Specifies a comma-separated list of strategy names for Vivado implementation runs. Use `ALL` to run all available implementation strategies. This lets you run a variety of implementation strategies at the same time during the build process and allows you to more quickly resolve the timing and routing issues of the design.

**--vivado.param**

```bash
--vivado.param <arg>
```

Specifies parameters for the Vivado Design Suite to be used during synthesis and implementation of the FPGA binary (`xclbin`).

**--vivado.prop**

```bash
--vivado.prop <arg>
```

Specifies properties for the Vivado Design Suite to be used during synthesis and implementation of the FPGA binary (`xclbin`).
### Table 39: Prop Options

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Valid Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>vivado.prop</code> &lt;object_type&gt;.&lt;object_name&gt;.&lt;prop_name&gt;`</td>
<td>Type: Various</td>
<td>This allows you to specify any property used in the Vivado hardware compilation flow. &lt;object_type&gt; is run</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>vivado.prop run.impl_1.STEPS.PHYS_OPT_DESIGN.IS_ENABLED=true</code> <code>vivado.prop run.impl_1.STEPS.PHYS_OPT_DESIGN.ARGS.DIRECTIVE=Explore</code> <code>vivado.prop run.impl_1.STEPS.PLACE_DESIGN.TCL.PRE=/…/xxx.tcl</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>vivado.prop fileset.current.top=foo</code> If &lt;object_type&gt; is set to file, current is not supported. If &lt;object_type&gt; is set to run, the special value of <strong>KERNEL</strong> can be used to specify run optimization settings for ALL kernels, instead of the need to specify them one by one.</td>
</tr>
</tbody>
</table>

For example, from the command line:

```
v++ --link --vivado.prop run.impl_1.STEPS.PHYS_OPT_DESIGN.IS_ENABLED=true --vivado.prop run.impl_1.STEPS.PHYS_OPT_DESIGN.ARGS.DIRECTIVE=Explore --vivado.prop run.impl_1.STEPS.PLACE_DESIGN.TCL.PRE=/…/xxx.tcl
```

The example above enables the optional PHYS_OPT_DESIGN step as part of the Vivado implementation process, sets the Explore directive for that step, and specifies a Tcl script to run before the PLACE_DESIGN step.

**TIP:** As described in Managing Vivado Synthesis and Implementation Results, each step in the Vivado synthesis and implementation process can have a Tcl prescript to run before the step, and a Tcl postscript to run after the step. This lets you customize the build process by inserting pre-processing or post-processing Tcl commands around the different steps. These scripts can be specified as shown in the example above.

These options can also be specified in a configuration file under the `[vivado]` section head using the following format:

```
[vivado]
prop=run.impl_1.STEPS.PHYS_OPT_DESIGN.IS_ENABLED=true
prop=run.impl_1.STEPS.PHYS_OPT_DESIGN.ARGS.DIRECTIVE=Explore
prop=run.impl_1.STEPS.PLACE_DESIGN.TCL.PRE=/…/xxx.tcl
```


**IMPORTANT!** Some Vivado properties have spaces in their name, such as MORE OPTIONS and Tcl syntax requires these properties to be enclosed in braces, {}. However, the placement of the braces in the --vivado options is important. You must surround the complete property name with braces, rather than just a portion of it. For instance, the correct placement would be:

```
--vivado_prop run.impl_1.{STEPS.PLACE_DESIGN.ARGS.MORE OPTIONS}={-no_bufg_opt}
```

While the following would result in an error during the build process:

```
--vivado_prop run.impl_1.STEPS.PLACE_DESIGN.ARGS.{MORE OPTIONS}={-no_bufg_opt}
```

---

**--vivado.synth.jobs**

```
--vivado.synth.jobs <arg>
```

Specifies the number of parallel jobs the Vivado Design Suite uses to synthesize the device binary. Increasing the number of jobs allows the Vivado synthesis to spawn more parallel processes and complete faster jobs.

For example:

```
v++ --link --vivado.synth.jobs 4
```

**--vivado.synth.lsf**

```
--vivado.synth.lsf <arg>
```

Specifies the bsub command line as a string to pass to an LSF cluster. This option is required to use the IBM Platform Load Sharing Facility (LSF) for Vivado synthesis.

For example:

```
v++ --link --vivado.synth.lsf '{bsub -R \"select[type=X86_64]\" -N -q medium}'}
```

---

**Vitis Compiler Configuration File**

A configuration file can also be used to specify the Vitis compiler options. A configuration file provides an organized way of passing options to the compiler by grouping similar switches together, and minimizing the length of the v++ command line. Some of the features that can be controlled through config file entries include:

- HLS options to configure kernel compilation
• Connectivity directives for system linking such as the number of kernels to instantiate or the assignment of kernel ports to global memory
• Directives for the Vivado Design Suite to manage hardware synthesis and implementation.

In general, any `v++` command option can be specified in a configuration file. However, the configuration file supports defining sections containing groups of related commands to help manage build options and strategies. The following table lists the defined sections.

<table>
<thead>
<tr>
<th>Table 40: Section Tags of the Configuration File</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Section Name</strong></td>
</tr>
</tbody>
</table>
| [hls] | compiler | HLS directives --hls Options:  
  • clock  
  • export_project  
  • export_mode  
  • max_memory_ports  
  • memory_port_data_width |
| [clock] | compiler | Clock commands --clock Options:  
  • defaultFreqHz  
  • defaultID  
  • freqHz  
  • id |
| [connectivity] | linker | --connectivity Options:  
  • nk  
  • sp  
  • stream_connect  
  • slr  
  • connect |
| [vivado] | linker | --vivado Options:  
  • param  
  • prop |
| [advanced] | either | --advanced Options:  
  • param  
  • prop  
  • misc |

**TIP:** Comments can be added to the configuration file by starting the line with a "#". The end of a section is specified by an empty line at the end of the section.

Because the `v++` command supports multiple config files on a single `v++` command line, you can partition your configuration files into related options that define compilation and linking strategies or Vivado implementation strategies, and apply multiple config files during the build process.
Configuration files are optional. There are no naming restrictions on the files and the number of configuration files can be zero or more. All `v++` options can be put in a single configuration file if desired. However, grouping related switches into separate files can help you organize your build strategy. For example, group [connectivity] related switches in one file, and [Vivado] options into a separate file.

The configuration file is specified through the use of the `v++ --config` option as discussed in the Vitis Compiler General Options. An example of the `--config` option follows:

```
v++ --config ../src/connectivity.cfg
```

Switches are read in the order they are encountered. If the same switch is repeated with conflicting information, the first switch read is used. The order of precedence for switches is as follows, where item one takes highest precedence:

1. Command line switches.
2. Config files (on command line) from left-to-right.
3. Within a config file, precedence is from top-to-bottom.

### Using the Message Rule File

The `v++` command executes various Xilinx tools during kernel compilation and linking. These tools generate many messages that provide build status to you. These messages might or might not be relevant to you depending on your focus and design phase. The Message Rule file (.mrf) can be used to better manage these messages. It provides commands to promote important messages to the terminal or suppress unimportant ones. This helps you better understand the kernel build result and explore methods to optimize the kernel.

The Message Rule file is a text file consisting of comments and supported commands. Only one command is allowed on each line.

#### Comment

Any line with "#" as the first non-white space character is a comment.

#### Supported Commands

By default, `v++` recursively scans the entire working directory and promotes all error messages to the `v++` output. The `promote` and `suppress` commands below provide more control on the `v++` output.

- **promote**: This command indicates that matching messages should be promoted to the `v++` output.
- **supress:** This command indicates that matching messages should be suppressed or filtered from the \texttt{v++} output. Note that errors cannot be suppressed.

Enter only one command per line.

**Command Options**

The Message Rule file can have multiple \texttt{promote} and \texttt{suppress} commands. Each command can have one and only one of the options below. The options are case-sensitive.

- \texttt{-id [<message_id>]}: All messages matching the specified message ID are promoted or suppressed. The message ID is in format of nnn-mmm. As an example, the following is a warning message from HLS. The message ID in this case is 204-68.

```
WARNING: [V++ 204-68] Unable to enforce a carried dependence constraint (II = 1, distance = 1, offset = 1) between bus request on port 'gmem' (/matrix_multiply_cl_kernel/mmult1.cl:57) and bus request on port 'gmem'-severity [severity_level]
```

For example, to suppress messages with message ID 204-68, specify the following:

\texttt{suppress -id 204-68}.

- \texttt{-severity [<severity_level>]}: The following are valid values for the severity level. All messages matching the specified severity level will be promoted or suppressed.

  - info
  - warning
  - critical_warning

  For example, to promote messages with severity of 'critical-warning', specify the following:

  \texttt{promote -severity critical_warning}.

**Precedence of Message Rules**

The \texttt{suppress} rules take precedence over \texttt{promote} rules. If the same message ID or severity level is passed to both \texttt{promote} and \texttt{suppress} commands in the Message Rule file, the matching messages are suppressed and not displayed.

**Example of Message Rule File**

The following is an example of a valid Message Rule file:

```
# promote all warning, critical warning
promote -severity warning
promote -severity critical_warning
# suppress the critical warning message with id 19-2342
suppress -id 19-2342
```
Chapter 23

emconfigutil Utility

When running software or hardware emulation in the command line flow, it is necessary to create an emulation configuration file, `emconfig.json`, used by the runtime library during emulation. The emulation configuration file defines the device type and quantity of devices to emulate for the specified platform. A single `emconfig.json` file can be used for both software and hardware emulation.

**Note:** When running on real hardware, the runtime and drivers query the installed hardware to determine the device type and quantity installed, along with the device characteristics.

To use the `emconfigutil` utility to automate the creation of the emulation file, specify the target platform and additional options in the `emconfigutil` command line:

```
emconfigutil --platform <platform_name> [options]
```

At a minimum, you must specify the target platform through the `-f` or `--platform` option to generate the required `emconfig.json` file. The specified platform must be the same as specified during the host and hardware builds.

The `emconfigutil` options are provided in the following table.

**Table 41: emconfigutil Options**

<table>
<thead>
<tr>
<th>Option</th>
<th>Valid Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-f</code> or <code>--platform</code></td>
<td>Target device</td>
<td>Required. Defines the target device from the specified platform. For a list of supported devices, refer to Supported Platforms.</td>
</tr>
<tr>
<td><code>--nd</code></td>
<td>Any positive integer</td>
<td>Optional. Specifies number of devices. The default is 1.</td>
</tr>
<tr>
<td><code>--od</code></td>
<td>Valid directory</td>
<td>Optional. Specifies the output directory. When running emulation, the <code>emconfig.json</code> file must be in the same directory as the host executable. The default is to write the output in the current directory.</td>
</tr>
<tr>
<td><code>-s</code> or <code>--save-temps</code></td>
<td>N/A</td>
<td>Optional. Specifies that intermediate files are not deleted and remain after the command is executed. The default is to remove temporary files.</td>
</tr>
<tr>
<td><code>--xp</code></td>
<td>Valid Xilinx parameters and properties.</td>
<td>Optional. Specifies additional parameters and properties. For example:</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>--xp prop:solution.platform_repo_paths=&lt;xsa_path&gt;</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td>This example sets the search path for the target platforms.</td>
</tr>
<tr>
<td><code>-h</code> or <code>--help</code></td>
<td>N/A</td>
<td>Prints command help.</td>
</tr>
</tbody>
</table>
The `emconfigutil` command generates the `emconfig.json` configuration file in the output directory or the current working directory.

**TIP:** When running emulation, the `emconfig.json` file must be in the same directory as the host executable.

The following example creates a configuration file targeting two `xilinx_u200_qdma_201910_1` devices.

```
$emconfigutil --xilinx_u200_qdma_201910_1 --nd 2
```
kernelinfo Utility

The kernelinfo utility extracts and displays information from Xilinx object (XO) files which can be used during host code development. This information includes hardware function names, arguments, offsets, and port data.

The following command options are available:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-h [ --help ]</td>
<td>Print help message.</td>
</tr>
<tr>
<td>-x [ --xo_path ] &lt;arg&gt;</td>
<td>Absolute path to file including file name and .xo extension.</td>
</tr>
<tr>
<td>-l [ --log ] &lt;arg&gt;</td>
<td>By default, information is displayed on the screen. Otherwise, you can use the --log option to output the information as a file.</td>
</tr>
<tr>
<td>-j [ --json ]</td>
<td>Output the file in JSON format.</td>
</tr>
<tr>
<td>[input_file]</td>
<td>XO file. Specify the XO file positionally or use the --xo_path option.</td>
</tr>
<tr>
<td>[output_file]</td>
<td>Output from Xilinx OpenCL Compiler. Specify the output file positionally, or use the --log option.</td>
</tr>
</tbody>
</table>

To run the kernelinfo utility, enter the following in a Linux terminal:

```
kernelinfo <filename.o>
```

The output is divided into three sections:

- Kernel Definitions
- Arguments
- Ports

The report generated by the following command is reviewed to help better understand the report content. Note that the report is broken down into specific sections for better understandability.

```
kernelinfo krnl_vadd.xo
```

Where krnl_vadd.xo is a compiled kernel.
Kernel Definition

Reports high-level kernel definition information. Importantly, for the host code development, the kernel name is given in the name field. In this example, the kernel name is krnl_vadd.

```plaintext
### Kernel Definition
name: krnl_vadd
language: c
vlnv: xilinx.com:hls:krnl_vadd:1.0
preferredWorkGroupSizeMultiple: 1
workGroupSize: 1
debug: true
containsDebugDir: 1
sourceFile: krnl_vadd/cpu_sources/krnl_vadd.cpp
```

Arguments

Reports kernel function arguments.

In the following example, there are four arguments: a, b, c, and n_elements.

```plaintext
### Arg
name: a
addressQualifier: 1
id: 0
port: M_AXI_GMEM
size: 0x8
offset: 0x10
hostOffset: 0x0
hostSize: 0x8
type: int*

### Arg
name: b
addressQualifier: 1
id: 1
port: M_AXI_GMEM
size: 0x8
offset: 0x1C
hostOffset: 0x0
hostSize: 0x8
type: int*

### Arg
name: c
addressQualifier: 1
id: 2
port: M_AXI_GMEM1
size: 0x8
offset: 0x28
hostOffset: 0x0
hostSize: 0x8
type: int*
```
Ports

Reports the memory and control ports used by the kernel.

--- Arg ---
name: n_elements
addressQualifier: 0
id: 3
port: S_AXI_CONTROL
size: 0x4
offset: 0x34
hostOffset: 0x0
hostSize: 0x4
type: int const

--- Port ---
name: M_AXI_GMEM
mode: master
range: 0xFFFFFFFF
dataWidth: 32
portType: addressable
base: 0x0

--- Port ---
name: M_AXI_GMEM1
mode: master
range: 0xFFFFFFFF
dataWidth: 32
portType: addressable
base: 0x0

--- Port ---
name: S_AXI_CONTROL
mode: slave
range: 0x1000
dataWidth: 32
portType: addressable
base: 0x0
launch_emulator Utility

For embedded platforms that have an Arm sub-system, the Vitis tool uses QEMU to emulate the PS subsystem. The QEMU processes must be run along with the RTL simulator process to emulate the entire system in hardware emulation. The launch_emulator is a utility which launches QEMU and manages the synchronization of the PL simulator processes. It launches QEMU and the simulation process with provided arguments. The Vitis IDE also calls this command when starting and stopping the emulator.

**TIP:** For help inside QEMU, press Ctrl + a h while in the emulator shell. To terminate the QEMU command, press Ctrl + a x while in the emulator shell.

For embedded platforms, the --package Options command generates scripts, launch_hw_emu.sh, or launch_sw_emu.sh to call the launch_emulator command with the required arguments based on the platform and the target application.

You can pass additional arguments to the launch_emulator utility from the command line when using the launch_hw_emu.sh or launch_sw_emu.sh wrapper scripts. Simply append the option to the command line when running the script. This allows you to customize the launch_emulator utility as needed to support your specific platform or application.

The following table shows the list of available options.

<table>
<thead>
<tr>
<th>Option</th>
<th>Accepted Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-aie-device-file-path</td>
<td>N/A</td>
<td>Pointer to the AI Engine array size information.</td>
</tr>
<tr>
<td>-aie-shim-sol-path</td>
<td>N/A</td>
<td>Points to the AI Engine shim solution file that provides the AI Engine constraints required for the Kernel.</td>
</tr>
<tr>
<td>-aie-sim-config</td>
<td>N/A</td>
<td>Points to the AI Engine sim config file that provides various AI Engine files that are required for the SystemC Model of AI Engine. This is auto passed by the v++ package. Required for AI Engine designs.</td>
</tr>
</tbody>
</table>
### Table 43: launch_emulator Utility Options (cont'd)

<table>
<thead>
<tr>
<th>Option</th>
<th>Accepted Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-aie-sim-options</td>
<td>N/A</td>
<td>Points to an AI Engine sim options file that has various AI Engine debug flags that are required for debugging the AI Engine SystemC module. The options file should be specified with a relative path with respect to package.hw_emu/sim/behav_waveform/xsim/. <strong>TIP:</strong> This is optional and only applies to AI Engine designs.</td>
</tr>
<tr>
<td>-config-file</td>
<td>Configuration file (ini format)</td>
<td>Configuration file that specifies options to use for the command.</td>
</tr>
<tr>
<td>-device-family</td>
<td>7Series</td>
<td>UltraScale</td>
</tr>
<tr>
<td>-emu-data</td>
<td>N/A</td>
<td>&lt;Additional emulation Data files&gt;: AI Engine Kernel to run QEMU requires shim solution file. Required for AI Engine designs.</td>
</tr>
<tr>
<td>-enable-debug</td>
<td>N/A</td>
<td>Debug mode opening two different XTERMs for QEMU and PL.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>IMPORTANT!</strong> This is very useful for the batch mode users to understand the flow and handshake between the QEMU and PL process.</td>
</tr>
<tr>
<td>-forward-port</td>
<td>&lt;target&gt; &lt;host&gt;</td>
<td>Forwards TCP port from target to host.</td>
</tr>
<tr>
<td>-g</td>
<td>-graphic-xsim</td>
<td>-sim-gui</td>
</tr>
<tr>
<td>-gdb-port</td>
<td>Port number</td>
<td>QEMU waits for GDB connection on &lt;port&gt;.</td>
</tr>
<tr>
<td>-help</td>
<td>N/A</td>
<td>Prints help message.</td>
</tr>
<tr>
<td>-kill</td>
<td>&lt;pid&gt;</td>
<td>Kills a specified emulator process.</td>
</tr>
<tr>
<td>-no-reboot</td>
<td>N/A</td>
<td>Exit QEMU instead of rebooting. Used to exit gracefully from QEMU by executing command reboot -e at the embedded Linux prompt.</td>
</tr>
<tr>
<td>-noc-memory-config</td>
<td>N/A</td>
<td>By default, v++ package creates the NoC memory configuration based on the design configuration, and you can see this file parallel to simulation binaries. You can override this file by replacing the file specified in the simulation binary folder. Use the -user-pre-sim-script option to copy your noc_memory_config.txt file to the simulation binary area and to get the configuration applied.</td>
</tr>
<tr>
<td>&lt;path/to/noc_memeory_config.txt&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-pid-file</td>
<td>File name</td>
<td>Write process ID to &lt;file&gt; for later use with -kill. Used by the Vitis software platform to kill once emulation is successful.</td>
</tr>
<tr>
<td>-pl-sim-args</td>
<td>Arguments to simulator</td>
<td>These arguments gets appended to simulator command line. Alternative to pm-sim-args-file.</td>
</tr>
<tr>
<td>-pl-sim-args-file</td>
<td>Simulation arguments file name</td>
<td>Any options to simulator tool can be given in this file.</td>
</tr>
</tbody>
</table>
### Table 43: launch_emulator Utility Options (cont’d)

<table>
<thead>
<tr>
<th>Option</th>
<th>Accepted Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-pl-sim-dir</code></td>
<td>Simulation directory</td>
<td>Start the Programmable Logic Simulator by launching the scripts from this directory. This is auto passed in the <code>v++</code> package generated script. The tool expects a file called <code>simulate.sh</code> in the specified directory and will execute it to launch the PL simulator (for example, XSIM).</td>
</tr>
<tr>
<td><code>-pl-sim-script</code></td>
<td>Simulation script location</td>
<td>Advanced users can have one direct script to launch simulation (for example, Vivado users). When this option is given, run the script, other options are of no value.</td>
</tr>
<tr>
<td><code>-pmc-args</code></td>
<td>Arguments to PMC</td>
<td>The MicroBlaze QEMU is run to emulate the PMC. Instead of writing into a file called <code>pmc_args.txt</code>, you can directly provide all the arguments that need to be appended to the PMC command line. This is an alternative to <code>-pmc-args-file</code>.</td>
</tr>
<tr>
<td><code>-pmc-args-file</code></td>
<td>PMC QEMU arguments file name</td>
<td>Any options to be passed to PMU/PMC can be given in this file. The specific format is determined by the base file on your chosen platform. This is auto passed in the <code>v++</code> package generated script.</td>
</tr>
<tr>
<td><code>-ps-only</code></td>
<td>N/A</td>
<td>PS only emulation. No PL emulation.</td>
</tr>
<tr>
<td><code>-qemu-args</code></td>
<td>Arguments to QEMU</td>
<td>Instead of writing into a file called <code>qemu_args.txt</code>, you can directly provide all the arguments that need to be appended to the QEMU command line. This is an alternative to <code>qemu-args-file</code>.</td>
</tr>
<tr>
<td><code>-qemu-args-file</code></td>
<td>PS QEMU Arguments file name</td>
<td>Any options to be passed to QEMU can be given in this file. This is specific format where you fetch the base file from the platform chosen. This is auto passed in the <code>v++</code> package generated script.</td>
</tr>
<tr>
<td><code>-qspi-high-image</code></td>
<td>Specify QSPI high image file</td>
<td>The image file which will be passed as a QEMU argument in the form of boot mode. This is auto passed in the <code>v++</code> package generated script. Required only when DUAL QSPI mode is used.</td>
</tr>
<tr>
<td><code>-qspi-image</code></td>
<td>Specify <code>qspi.bin</code></td>
<td>The image file is passed as a QEMU argument in the form of boot mode. This is auto passed in the <code>v++</code> package generated script. Required only when you opt for QSPI mode.</td>
</tr>
<tr>
<td><code>-qspi-low-image</code></td>
<td>Specify QSPI low image file</td>
<td>The image file is passed as a QEMU argument in the form of boot mode. This is auto passed in the <code>v++</code> package generated script. Required only when DUAL QSPI mode is used.</td>
</tr>
<tr>
<td><code>-run-sim-in-gdb</code></td>
<td>N/A</td>
<td>Run Simulator in GDB.</td>
</tr>
<tr>
<td><code>-runtime</code></td>
<td><code>c++/ocl</code></td>
<td>Specify the runtime flow for C++ or OCL.</td>
</tr>
<tr>
<td><code>-sd-card-image</code></td>
<td>Specify <code>sd_card.img</code></td>
<td>The image file is passed as a QEMU argument in the form of boot mode. This is auto passed in the <code>v++</code> package generated script. Required only when SD mode is used.</td>
</tr>
</tbody>
</table>
### Table 43: launch_emulator Utility Options (cont’d)

<table>
<thead>
<tr>
<th>Option</th>
<th>Accepted Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-t</code></td>
<td><code>-target</code></td>
<td>swemu or hwemu</td>
</tr>
<tr>
<td><code>-timeout</code></td>
<td>&lt;n&gt;</td>
<td>Terminates emulation after &lt;n&gt; seconds.</td>
</tr>
<tr>
<td><code>-user-post-sim-script</code></td>
<td>Path to Tcl script</td>
<td>Creates Tcl for any post operations into a Tcl file and pass the Tcl script to this switch.</td>
</tr>
<tr>
<td><code>-user-pre-sim-script</code></td>
<td>Path to Tcl script</td>
<td>For first run, launch_emulator in GUI mode and add the signals that you want to observe. Copies the commands from the Tcl console and save into a Tcl script. From the next run, pass the Tcl script in batch mode, launch_emulator -user-pre-sim-script &lt;path_to_saved_tcl_script&gt;. Only supports the Vivado simulator (xsim).</td>
</tr>
<tr>
<td><code>-vivado</code></td>
<td>$XILINX_VIVADO</td>
<td>Sets the VIVADO_LOC and is used by simulate.sh to load simulation/c-model libraries</td>
</tr>
<tr>
<td><code>-wcfg-file-path</code></td>
<td>N/A</td>
<td>Specify the wcfg file created by the XSIM to open during GUI simulation</td>
</tr>
<tr>
<td><code>-xtlm-aximm-log</code></td>
<td>N/A</td>
<td>This switch will generate xTLM AXI4 transaction logs for interface connection between two SystemC models (with information like address/data/size, etc.). While running the emulation log is available at (directory structure can vary based on v++ options and simulator used): package.hw_emu/sim/behav_waveform/xsim/xsc_report.log</td>
</tr>
<tr>
<td><code>-xtlm-axis-log</code></td>
<td>N/A</td>
<td>This switch will generate xTLM AXI4-Stream transaction logs for interface connection between two SystemC models. While running emulation log is available at (directory structure can vary based on v++ options and simulator used): package.hw_emu/sim/behav_waveform/xsim/xsc_report.log</td>
</tr>
</tbody>
</table>
manage_ipcache Utility

To provide better performance during synthesis of kernels in your application designs, the Vitis compiler uses an IP cache to store and reuse synthesis results. This lets the build process for the .xclbin file avoid having to repeat synthesis for kernels and CUs that have not changed. The IP cache stores the synthesis results and applies them for unchanged kernels in the design.

By default, the IP cache is stored inside the Vitis IDE workspace for a project, or at the level of your builds when running v++ from the command line. You can customize the location for the IP cache using --remote_ip_cache to specify a new location, or disable the use of the IP cache using --no_ip_cache. See Vitis Compiler General Options for information on these options.

The manage_ipcache utility is a standalone utility to help you manage the contents of your IP cache repository. It lets you report statistics on the IP cache repository and remove entries based on a variety of criteria.

Table 44: manage_ipcache Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-c</td>
<td>--cache</td>
</tr>
<tr>
<td>-d</td>
<td>--disk_space &lt;size&gt;</td>
</tr>
<tr>
<td>-h</td>
<td>--help</td>
</tr>
<tr>
<td>-k</td>
<td>--keep_top &lt;N&gt;</td>
</tr>
<tr>
<td>-o</td>
<td>--outfile &lt;file&gt;</td>
</tr>
<tr>
<td>-p</td>
<td>--purge</td>
</tr>
<tr>
<td>-r</td>
<td>--report</td>
</tr>
<tr>
<td>-u</td>
<td>--unused</td>
</tr>
</tbody>
</table>

The following example reports on the entries of the specified IP cache:

```
manage_ipcache --cache ./ip_cache --report
```

The manage_ipcache command returns 0 if successful, or returns -1 if an error occurs.
package_xo Command

Syntax

```
package_xo -kernel_name <arg> [-force] [-kernel_xml <arg>]
[-output_kernel_xml <arg>] [-design_xml <arg>]
[-ip_directory <arg>] [-parent_ip_directory <arg>]
[-kernel_files <args>] [-kernel_xml_args <args>]
[-kernel_xml_pipes <args>] [-kernel_xml_connections <args>]
[ctrl_protocol <arg>] -xo_path <arg> [-quiet] [-verbose]
```

Description

The `package_xo` command is a TCL command within the Vivado Design Suite. Kernels written in RTL are compiled in the Vivado tool using the `package_xo` command line utility which generates a Xilinx object (XO) file which can subsequently used by the `v++` command, during the linking stage.

Table 45: Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-kernel_name &lt;arg&gt;</td>
<td>(Required) Specify the name of the RTL kernel.</td>
</tr>
<tr>
<td>-force</td>
<td>(Optional) Overwrite an existing XO file if one exists.</td>
</tr>
<tr>
<td>-kernel_xml &lt;arg&gt;</td>
<td>(Optional) Specify the path to an existing kernel XML file. The Vivado tool</td>
</tr>
<tr>
<td></td>
<td>will create a kernel.xml file for the XO file if one is not specified.</td>
</tr>
<tr>
<td>-output_kernel_xml</td>
<td>(Optional) Specify the path to write the kernel XML file. The Vivado tool</td>
</tr>
<tr>
<td></td>
<td>will create a kernel.xml file to include in the XO file, and also write it</td>
</tr>
<tr>
<td></td>
<td>to the specified output file.</td>
</tr>
<tr>
<td>-design_xml &lt;arg&gt;</td>
<td>(Optional) Specify the path to an existing design XML file</td>
</tr>
<tr>
<td>-ip_directory &lt;arg&gt;</td>
<td>(Optional) Specify the path to the packaged IP directory.</td>
</tr>
<tr>
<td>-parent_ip_directory</td>
<td>(Optional) If the kernel IP directory specified contains multiple IPs,</td>
</tr>
<tr>
<td></td>
<td>specify a directory path to the parent IP where its component.xml file is</td>
</tr>
<tr>
<td></td>
<td>located directly below.</td>
</tr>
<tr>
<td>-kernel_files</td>
<td>(Optional) Kernel file name(s).</td>
</tr>
</tbody>
</table>

TIP: You can use this option to generate a kernel.xml file which you can edit and use as an input in the `package_xo` command.
Table 45: Arguments (cont’d)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-kernel_xml_args &lt;args&gt;</td>
<td>(Optional) Generate the kernel.xml with the specified function arguments. Each argument value should use the following format: {name:addressQualifier:id:port:size:offset:type:memSize}</td>
</tr>
<tr>
<td></td>
<td>Note: memSize is optional.</td>
</tr>
<tr>
<td>-kernel_xml_pipes &lt;args&gt;</td>
<td>(Optional) Generate the kernel.xml with the specified pipe(s). Each pipe value use the following format: {name:width:depth}</td>
</tr>
<tr>
<td>-kernel_xml_connections &lt;args&gt;</td>
<td>(Optional) Generate the kernel.xml file with the specified connections. Each connection value should use the following format: {srcInst:srcPort:dstInst:dstPort}</td>
</tr>
<tr>
<td>-ctrl_protocol</td>
<td>Kernel control protocol as described in Kernel Execution Modes. Valid values: ap_ctrl_hs, ap_ctrl_chain, ap_ctrl_none, user_managed.</td>
</tr>
<tr>
<td></td>
<td>TIP: The default ap_ctrl_hs is written to the kernel.xml file when -ctrl_protocol is not specified.</td>
</tr>
<tr>
<td>-xo_path &lt;arg&gt;</td>
<td>(Required) Specify the path and file name of the compiled object (XO) file.</td>
</tr>
<tr>
<td>-quiet</td>
<td>(Optional) Execute the command quietly, returning no messages from the command. The command also returns TCL_OK regardless of any errors encountered during execution.</td>
</tr>
<tr>
<td></td>
<td>Note: Any errors encountered on the command-line, while launching the command, will be returned. Only errors occurring inside the command will be trapped.</td>
</tr>
<tr>
<td>-verbose</td>
<td>(Optional) Temporarily override any message limits and return all messages from this command.</td>
</tr>
<tr>
<td></td>
<td>Note: Message limits can be defined with the set_msg_config command.</td>
</tr>
</tbody>
</table>

Examples

The following example creates the specified XO file containing an RTL kernel of the specified name using the ap_ctrl_chain control protocol, and creates the kernel.xml file because one has not been specified:

```
package_xo -xo_path Vadd_A_B.xo -kernel_name Vadd_A_B -ctrl_protocol ap_ctrl_chain -ip_directory ./ip
```
The following example creates the XO file using the specified `kernel.xml` file:

```bash
package_xo -xo_path Vadd_A_B.xo -kernel_name Vadd_A_B -kernel_xml kernel.xml -ip_directory ./ip
```

**TIP:** The control protocol will be defined in the specified `kernel.xml` file.
platforminfo Utility

The platforminfo command line utility reports platform meta-data including information on interface, clock, valid SLRs and allocated resources, and memory in a structured format. This information can be referenced when allocating kernels to SLRs or memory resources for instance.

The following command options are available to use with platforminfo:

**Table 46: platforminfo Commands**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-f [ --force ]</td>
<td>Overwrite an existing output file.</td>
</tr>
<tr>
<td>-h [ --help ]</td>
<td>Print help message and exit.</td>
</tr>
<tr>
<td>-k [ --keys ]</td>
<td>Get keys for a given platform. Returns a list of JSON paths.</td>
</tr>
<tr>
<td>-l [ --list ]</td>
<td>List platforms. Searches the user repo paths $PLATFORM_REPO_PATHS and then the install locations to find .xpfm files.</td>
</tr>
<tr>
<td>-e [ --extended ]</td>
<td>List platforms with extended information. Use with '--list'.</td>
</tr>
<tr>
<td>-d [ --hw ] &lt;arg&gt;</td>
<td>Hardware platform definition (*.dsa) on which to operate. The value must be a full path, including file name and .dsa extension.</td>
</tr>
<tr>
<td>-s [ --sw ] &lt;arg&gt;</td>
<td>Software platform definition (*.spfm) on which to operate. The value must be a full path, including file name and .spfm extension.</td>
</tr>
<tr>
<td>-p [ --platform ] &lt;arg&gt;</td>
<td>Xilinx® platform definition (*.xpfm) on which to operate. The value for --platform can be a full path including file name and .xpfm extension, as shown in example 1 below. If supplying a file name and .xpfm extension without a path, this utility will search only the current working directory. You can also specify just the base name for the platform. When the value is a base name, this utility will search the $PLATFORM_REPO_PATHS, and the install locations, to find a corresponding .xpfm file, as shown in example 2 below.</td>
</tr>
<tr>
<td>-o [ --output ] &lt;arg&gt;</td>
<td>Specify an output file to write the results to. By default the output is returned to the terminal (stdout).</td>
</tr>
</tbody>
</table>

Example 1: --platform /opt/xilinx/platforms/xilinx_u50_gen3x16_iommu_201902_3.xpfm

Example 2: --platform xilinx_u200_iommu_201830_1
### Table 46: platforminfo Commands (cont’d)

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
</table>
| `-j [ --json ] <arg>` | Specify JSON format for the generated output. When used with no value, the `platforminfo` utility prints the entire platform in JSON format. This option also accepts an argument that specifies a JSON path, as returned by the `-k` option. The JSON path, when valid, is used to fetch a JSON subtree, list, or value.  
  
  **Example 1:**
  `platforminfo --json='hardwarePlatform' --platform <platform base name>`
  
  **Example 2:** Specify the index when referring to an item in a list.
  `platforminfo --json='hardwarePlatform.devices[0].name' --platform <platform base name>`
  
  **Example 3:** When using the short option form (`-j`), the value must follow immediately.
  `platforminfo -j"hardwarePlatform.systemClocks[]" -p <platform base name>` |

| `-v [ --verbose ]` | Specify more detailed information output. The default behavior is to produce a human-readable report containing the most important characteristics of the specified platform. |

**Note:** Except when using the `--help` or `--list` options, a platform must be specified. You can specify the platform using the `--platform` option, or using either `--hw`, `--sw`. You can also simply insert the platform name or full path into the command line positionally.

To understand the generated report, condensed output logs, based on the following command are reviewed. Note that the report is broken down into specific sections for better understandability.

```bash
platforminfo -p $PLATFORM_REPO_PATHS/xilinx_u200_xdma_201830_1.xpfm
```

**TIP:** See Platforminfo for xilinx_zcu104_base_202010_1 for an example of embedded processor platforms.

## Basic Platform Information

Platform information and high-level description are reported.

<table>
<thead>
<tr>
<th>Platform:</th>
<th>xdma</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description:</td>
<td></td>
</tr>
</tbody>
</table>
Hardware Platform Information

General information on the hardware platform is reported. For the Software Emulation and Hardware Emulation field, a "1" indicates this platform is suitable for these configurations.

<table>
<thead>
<tr>
<th>Vendor:</th>
<th>xilinx</th>
</tr>
</thead>
<tbody>
<tr>
<td>Board: U200 (xdma)</td>
<td></td>
</tr>
<tr>
<td>Name: xdma</td>
<td></td>
</tr>
<tr>
<td>Version: 201830.3</td>
<td></td>
</tr>
<tr>
<td>Generated Version:</td>
<td>2018.3</td>
</tr>
<tr>
<td>Hardware: 1</td>
<td></td>
</tr>
<tr>
<td>Software Emulation:</td>
<td>1</td>
</tr>
<tr>
<td>Hardware Emulation:</td>
<td>1</td>
</tr>
<tr>
<td>Hardware Emulation Platform: 0</td>
<td></td>
</tr>
<tr>
<td>FPGA Family:</td>
<td>virtexuplus</td>
</tr>
<tr>
<td>FPGA Device:</td>
<td>xcu200</td>
</tr>
<tr>
<td>Board Vendor:</td>
<td>xilinx.com</td>
</tr>
<tr>
<td>Board Name:</td>
<td>xilinx.com:au200:1.0</td>
</tr>
<tr>
<td>Board Part:</td>
<td>xcu200-fsgd2104-2-e</td>
</tr>
</tbody>
</table>

Interface Information

The following shows the reported PCIe interface information.

<table>
<thead>
<tr>
<th>Interface Name: PCIe</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interface Type: gen3x16</td>
</tr>
<tr>
<td>PCIe Vendor Id:</td>
</tr>
<tr>
<td>PCIe Device Id:</td>
</tr>
<tr>
<td>PCIe Subsystem Id:</td>
</tr>
</tbody>
</table>

Clock Information

Reports the maximum kernel clock frequencies available. The Clock Index is the reference used in the `--kernel_frequency v++` directive when overriding the default value.

<table>
<thead>
<tr>
<th>Default Clock Index: 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clock Index: 1</td>
</tr>
<tr>
<td>Frequency: 500.000000</td>
</tr>
<tr>
<td>Clock Index: 0</td>
</tr>
<tr>
<td>Frequency: 300.000000</td>
</tr>
</tbody>
</table>
### Valid SLRs

Reports the valid SLRs in the platform.

<table>
<thead>
<tr>
<th>SLR</th>
<th>LUTs</th>
<th>FFs</th>
<th>BRAMs</th>
<th>DSPs</th>
</tr>
</thead>
<tbody>
<tr>
<td>SLR0</td>
<td>354690</td>
<td>723308</td>
<td>638</td>
<td>2265</td>
</tr>
<tr>
<td>SLR1</td>
<td>159739</td>
<td>331654</td>
<td>326</td>
<td>1317</td>
</tr>
<tr>
<td>SLR2</td>
<td>354839</td>
<td>723294</td>
<td>638</td>
<td>2265</td>
</tr>
</tbody>
</table>

### Resource Availability

The total available resources and resources available per SLR are reported. This information can be used to assess applicability of the platform for the design and help guide allocation of compute unit to available SLRs.

<table>
<thead>
<tr>
<th>Resource</th>
<th>Total</th>
<th>Per SLR</th>
</tr>
</thead>
<tbody>
<tr>
<td>LUTs</td>
<td>1047139</td>
<td>SLR0:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LUTs: 354690</td>
</tr>
<tr>
<td></td>
<td></td>
<td>FFs: 723308</td>
</tr>
<tr>
<td></td>
<td></td>
<td>BRAMs: 638</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DSPs: 2265</td>
</tr>
<tr>
<td>FFs</td>
<td>2186064</td>
<td>SLR1:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LUTs: 159739</td>
</tr>
<tr>
<td></td>
<td></td>
<td>FFs: 331654</td>
</tr>
<tr>
<td></td>
<td></td>
<td>BRAMs: 326</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DSPs: 1317</td>
</tr>
<tr>
<td>BRAMs</td>
<td>1896</td>
<td>SLR2:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LUTs: 354839</td>
</tr>
<tr>
<td></td>
<td></td>
<td>FFs: 723294</td>
</tr>
<tr>
<td></td>
<td></td>
<td>BRAMs: 638</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DSPs: 2265</td>
</tr>
</tbody>
</table>
Memory Information

Reports the available DDR and PLRAM memory connections per SLR as shown in the example output below.

Type: ddr4
Bus SP Tag: DDR
  Segment Index: 0
  Consumption: automatic
  SP Tag: bank0
  SLR: SLR0
  Max Masters: 15
  Segment Index: 1
  Consumption: default
  SP Tag: bank1
  SLR: SLR1
  Max Masters: 15
  Segment Index: 2
  Consumption: automatic
  SP Tag: bank2
  SLR: SLR1
  Max Masters: 15
  Segment Index: 3
  Consumption: automatic
  SP Tag: bank3
  SLR: SLR2
  Max Masters: 15
Bus SP Tag: PLRAM
  Segment Index: 0
  Consumption: explicit
  SLR: SLR0
  Max Masters: 15
  Segment Index: 1
  Consumption: explicit
  SLR: SLR1
  Max Masters: 15
  Segment Index: 2
  Consumption: explicit
  SLR: SLR2
  Max Masters: 15

The **Bus SP Tag** heading can be DDR or PLRAM and gives associated information below.

The **Segment Index** field is used in association with the **SP Tag** to generate the associated memory resource index as shown below.

**Bus SP Tag[Segment Index]**

For example, if **Segment Index** is 0, then the associated DDR resource index would be DDR[0].

This memory index is used when specifying memory resources in the `v++` command as shown below:

```
v++ … --connectivity.sp vadd.m_axi_gmem:DDR[3]
```
There can be more than one Segment Index associated with an SLR. For instance, in the output above, SLR1 has both Segment Index 1 and 2.

The Consumption field indicates how a memory resource is used when building the design:

- **default**: If the `--connectivity.sp` directive is not specified, it uses this memory resource by default during `v++` build. For example in the report below, DDR with Segment Index 1 is used by default.

- **automatic**: When the maximum number of memory interfaces have been used under Consumption: default have been fully applied, then the interfaces under automatic is used. The maximum number of interfaces per memory resource are given in the Max Masters field.

- **explicit**: For PLRAM, consumption is set to explicit which indicates this memory resource is only used when explicitly indicated through the `v++` command line.

---

### Feature ROM Information

The feature ROM information provides build related information on ROM platform and can be requested by Xilinx Support when debugging system issues.

<table>
<thead>
<tr>
<th>Feature ROM Information</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROM Major Version</td>
<td>10</td>
</tr>
<tr>
<td>ROM Minor Version</td>
<td>1</td>
</tr>
<tr>
<td>ROM Vivado Build ID</td>
<td>2388429</td>
</tr>
<tr>
<td>ROM DDR Channel Count</td>
<td>5</td>
</tr>
<tr>
<td>ROM DDR Channel Size</td>
<td>16</td>
</tr>
<tr>
<td>ROM Feature Bit Map</td>
<td>655885</td>
</tr>
<tr>
<td>ROM UUID</td>
<td>00194bb3-707b-49c4-911e-a66899000b6b</td>
</tr>
<tr>
<td>ROM CDMA Base Address 0</td>
<td>620756992</td>
</tr>
<tr>
<td>ROM CDMA Base Address 1</td>
<td>0</td>
</tr>
<tr>
<td>ROM CDMA Base Address 2</td>
<td>0</td>
</tr>
<tr>
<td>ROM CDMA Base Address 3</td>
<td>0</td>
</tr>
</tbody>
</table>

### Software Platform Information

Although software platform information is reported, it is only useful for users that have an OS running on the device, and not applicable to users that use a host machine.

<table>
<thead>
<tr>
<th>Software Platform Information</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Runtimes</td>
<td>1</td>
</tr>
<tr>
<td>Default System Configuration</td>
<td>config0_0</td>
</tr>
<tr>
<td>System Config Name</td>
<td>config0_0</td>
</tr>
<tr>
<td>System Config Description</td>
<td>config0_0 Linux OS on x86_0</td>
</tr>
<tr>
<td>System Config Default Processor Group</td>
<td>x86_0</td>
</tr>
<tr>
<td>System Config Default Boot Image</td>
<td>config0_0</td>
</tr>
<tr>
<td>System Config Is QEMU Supported</td>
<td>0</td>
</tr>
</tbody>
</table>
Platforminfo for xilinx_zcu104_base_202010_1

Use the following command to return the platforminfo for the xilinx_zcu104_base_202010_1 platform:

```
platforminfo -p xilinx_zcu104_base_202010_1
```

The results returned are as follows:

```
============================
Basic Platform Information
============================
Platform: xilinx_zcu104_base_202010_1
File: /platforms/xilinx_zcu104_base_202010_1/
xilinx_zcu104_base_202010_1.xpfm
Description:
A basic static platform targeting the ZCU104 evaluation board, which includes 2GB DDR4, GEM, USB, SDIO interface and UART of the Processing System. It reserves most of the PL resources for user to add acceleration kernels

=======================================
Hardware Platform (Shell) Information
=======================================
Vendor: xilinx
Board: zcu104_base
Name: zcu104_base
Version: 1.0
Generated Version: 2020.1
Software Emulation: 1
Hardware Emulation: 0
FPGA Family: zynqulplus
FPGA Device: xczu7ev
Board Vendor: xilinx.com
Board Name: xilinx.com:zcu104:1.1
Board Part: xczu7ev-ffvc1156-2-e
Maximum Number of Compute Units: 60

=================
Clock Information
=================
Default Clock Index: 0
Clock Index: 0
Frequency: 150.000000
Clock Index: 1
```
| Frequency: | 300.000000 |
| Clock Index: | 2 |
| Frequency: | 75.000000 |
| Clock Index: | 3 |
| Frequency: | 100.000000 |
| Clock Index: | 4 |
| Frequency: | 200.000000 |
| Clock Index: | 5 |
| Frequency: | 400.000000 |
| Clock Index: | 6 |
| Frequency: | 600.000000 |

==================
Memory Information
==================
- Bus SP Tag: HP0
- Bus SP Tag: HP1
- Bus SP Tag: HP2
- Bus SP Tag: HP3
- Bus SP Tag: HPC0
- Bus SP Tag: HPC1

=======================
Feature ROM Information
=======================

============================= Software Platform Information ==============================
Number of Runtimes: 1
Default System Configuration: xilinx_zcu104_base_202010_1
System Configurations:
- System Config Name: xilinx_zcu104_base_202010_1
- System Config Description: xilinx_zcu104_base_202010_1
- System Config Default Processor Group: xrt
- System Config Default Boot Image: standard
- System Config Is QEMU Supported: 1
- System Config Processor Groups:
  - Processor Group Name: xrt
  - Processor Group CPU Type: cortex-a53
  - Processor Group OS Name: linux
- System Config Boot Images:
  - Boot Image Name: standard
  - Boot Image Type:
  - Boot Image BIF: xilinx_zcu104_base_202010_1/boot/linux.bif
  - Boot Image Data: xilinx_zcu104_base_202010_1/xrt/image
  - Boot Image Boot Mode: sd
  - Boot Image RootFileSystem: /mnt
  - Boot Image Read Me: xilinx_zcu104_base_202010_1/boot/generic.readme
  - Boot Image QEMU Args: xilinx_zcu104_base_202010_1/qemu/pmu_args.txt:xilinx_zcu104_base_202010_1/qemu/qemu_args.txt
  - Boot Image QEMU Boot:
  - Boot Image QEMU Dev Tree:

Supported Runtimes:
- Runtime: OpenCL
An XML kernel description file, called kernel.xml, must be created for each RTL kernel, so that it can be used in the Vitis application acceleration development flow. The kernel.xml file specifies kernel attributes like the register map and ports needed by the runtime and Vitis tool flows. The following code shows an example of a kernel.xml file.

```xml
<?xml version="1.0" encoding='UTF-8'?>
<root versionMajor="1" versionMinor="6">
  <kernel name="vitis_kernel_wizard_0" language="ip_c"
    vlnv="mycompany.com:kernel:vitis_kernel_wizard_0:1.0"
    attributes="" preferredWorkGroupSizeMultiple="0" workGroupSize="1"
    interrupt="true">
    <ports>
      <port name='s_axi_control' mode='slave' range='0x1000' dataWidth='32'
        portType='addressable' base='0x0'/>
      <port name='m00_axi' mode='master' range='0xFFFFFFFFFFFFFFFF'
        dataWidth='512' portType='addressable'
        base='0x0'/>
    </ports>
    <args>
      <arg name='axi00_ptr0' addressQualifier='1' id='0' port='m00_axi'
        size='0x8' offset='0x010' type='int*' hostOffset='0x0' hostSize='0x8'/>
    </args>
  </kernel>
</root>
```

Note: The kernel.xml file can be created automatically using the RTL Kernel Wizard to specify the interface specification of your RTL kernel. For more information, refer to RTL Kernel Wizard.

The following table describes the format of the kernel.xml in detail:

**Table 47: Kernel XML File Content**

<table>
<thead>
<tr>
<th>Tag</th>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;root&gt;</td>
<td>versionMajor</td>
<td>For the current release of Vitis software platform, set to 1.</td>
</tr>
<tr>
<td></td>
<td>versionMinor</td>
<td>For the current release of Vitis software platform, set to 6.</td>
</tr>
</tbody>
</table>
Table 47: Kernel XML File Content (cont’d)

<table>
<thead>
<tr>
<th>Tag</th>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td></td>
<td>Kernel name</td>
</tr>
<tr>
<td>language</td>
<td></td>
<td>Always set to ip_c for RTL kernels.</td>
</tr>
<tr>
<td>vlnv</td>
<td></td>
<td>Must match the vendor, library, name, and version attributes in the component.xml of an IP. For example, if component.xml has the following tags:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The vlnv attribute in kernel XML must be set to: xilinx.com:hls:test_sincos:1.0</td>
</tr>
<tr>
<td>attributes</td>
<td></td>
<td>Reserved. Set it to empty string:**</td>
</tr>
<tr>
<td>preferredWorkGroupSizeMultiple</td>
<td></td>
<td>Reserved. Set it to 0.</td>
</tr>
<tr>
<td>workGroupSize</td>
<td></td>
<td>Reserved. Set it to 1.</td>
</tr>
<tr>
<td>interrupt</td>
<td></td>
<td>Set to &quot;true&quot; (interrupt=&quot;true&quot;) if the RTL kernel has an interrupt, otherwise omit.</td>
</tr>
<tr>
<td>hwControlProtocol</td>
<td></td>
<td>Specifies the control protocol for the RTL kernel.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• ap_ctrl_hs: Default control protocol for RTL kernels.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• ap_ctrl_chain: Control protocol for chained kernels that support dataflow. Adds ap_continue to the control registers to enable ap_done/ap_continue completion acknowledgment.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• ap_ctrl_none: Control protocol (none) applied for continuously operating kernels that have no need for start or done. For details, refer to Free-Running Kernel.</td>
</tr>
<tr>
<td>name</td>
<td></td>
<td>Specifies the port name.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>IMPORTANT! The AXI4-Lite interface must be named S_AXI_CONTROL.</td>
</tr>
<tr>
<td>mode</td>
<td></td>
<td>At least one AXI4 master port and one AXI4-Lite slave control port are required. AXI4-Stream ports can be specified to stream data between kernels.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• For AXI4 master port, set to &quot;master.&quot;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• For AXI4 slave port, set to &quot;slave.&quot;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• For AXI4-Stream master port, set to &quot;write_only.&quot;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• For AXI4-Stream slave port, set it &quot;read_only.&quot;</td>
</tr>
<tr>
<td>range</td>
<td></td>
<td>The range of the address space for the port.</td>
</tr>
<tr>
<td>dataWidth</td>
<td></td>
<td>The width of the data that goes through the port, default is 32-bits.</td>
</tr>
<tr>
<td>portType</td>
<td></td>
<td>Indicate whether or not the port is addressable or streaming.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• For AXI4 master and slave ports, set it to &quot;addressable.&quot;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• For AXI4-Stream ports, set it to &quot;stream.&quot;</td>
</tr>
<tr>
<td>base</td>
<td></td>
<td>For AXI4 master and slave ports, set to 0x0. This tag is not applicable to AXI4-Stream ports.</td>
</tr>
<tr>
<td>Tag</td>
<td>Attribute</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td></td>
<td>name</td>
<td>Specifies the kernel software argument name.</td>
</tr>
<tr>
<td></td>
<td>addressQualifier</td>
<td>Valid values: 0: Scalar kernel input argument</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1: global memory</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2: local memory</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3: constant memory</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4: pipe</td>
</tr>
<tr>
<td></td>
<td>id</td>
<td>Only applicable for AXI4 master and slave ports. The ID needs to be sequential. It is used to determine the order of kernel arguments. Not applicable for AXI4-Stream ports.</td>
</tr>
<tr>
<td></td>
<td>port</td>
<td>Specifies the &lt;port&gt; name to which the arg is connected.</td>
</tr>
<tr>
<td></td>
<td>size</td>
<td>Size of the argument in bytes. The default is 4 bytes.</td>
</tr>
<tr>
<td></td>
<td>offset</td>
<td>Indicates the register memory address.</td>
</tr>
<tr>
<td></td>
<td>type</td>
<td>The C data type of the argument. For example, uint*, int*, or float*.</td>
</tr>
<tr>
<td></td>
<td>hostOffset</td>
<td>Reserved. Set to 0x0.</td>
</tr>
<tr>
<td></td>
<td>hostSize</td>
<td>Size of the argument. The default is 4 bytes.</td>
</tr>
<tr>
<td></td>
<td>memSize</td>
<td>For AXI4-Stream ports, memSize sets the depth of the created FIFO.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TIP: Not applicable to AXI4 ports.</td>
</tr>
</tbody>
</table>

The following tags specify additional tags for AXI4-Stream ports. They do not apply to AXI4 ports.

<table>
<thead>
<tr>
<th>&lt;connection&gt;</th>
<th>The connection tag describes the actual connection in hardware, either from the kernel to the FIFO inserted for the PIPE, or from the FIFO to the kernel.</th>
</tr>
</thead>
<tbody>
<tr>
<td>srcInst</td>
<td>Specifies the source instance of the connection.</td>
</tr>
<tr>
<td>srcPort</td>
<td>Specifies the port on the source instance for the connection.</td>
</tr>
<tr>
<td>dstInst</td>
<td>Specifies the destination instance of the connection.</td>
</tr>
<tr>
<td>dstPort</td>
<td>Specifies the port on the destination instance of the connection.</td>
</tr>
</tbody>
</table>
xbutil Utility

**TIP: The next generation of the xbutil command-line tool is in preview mode for the 2020.2 release of XRT. This version will replace the current xbutil in a future release of XRT. Refer to [https://xilinx.github.io/XRT/2020.2/html/xbutil2.html](https://xilinx.github.io/XRT/2020.2/html/xbutil2.html) for information about the new utility.**

The Xilinx® Board Utility (xbutil) is a standalone command line utility that is included with the Xilinx Runtime (XRT) installation package. The xbutil command only supports platforms on Alveo Data Center accelerator cards, and embedded processor-based platforms.

Accelerator cards are partitioned into a user function and a management function to provide different levels of card access. The user function allows end users to load and run their applications, while the management function is for system administrators to manage the card. The xbutil utility interacts with the user function. The xbmgmt utility, which requires root privilege, is for interacting with the management function. The reason for splitting the function access between the two utilities is to provide some security for the management features of the tool.

XRT needs to be installed and identified on the card. For customized Alveo card setups in a Vivado flow, xbutil is not used. It includes multiple commands to validate and identify the installed accelerator card(s) along with additional card details including on card memory, host interface, target platform name, and system information. This information can be used for both card administration and application debugging.

**IMPORTANT! Although xbutil supports embedded processor platforms, only the following commands are available for use with those platforms: dump, help, program (for DFX platforms only), query, scan, and status.**

The xbutil command line format is as follows:

```
xbutil <command> [options]
```

Where the available commands and options are given below.

- `clock`
- `dmatest`
- `dump`
- `m2mtest`
- `mem --read`
TIP: You can use the `help` command to list the available `xbutil` commands and options:

```
xbutil help
```

---

## clock

**IMPORTANT! This option cannot be used with embedded processor platforms.**

The `clock` command allows you to change the clock frequencies of the clock(s) to the Compute Units (CUs) inside a `xclbin`. It has the following command line format:

```
xbutil clock [-d card] [-r region] [-f clock1_freq_MHz] [-g clock2_freq_MHz] [-h clock3_freq_MHz]
```

The clock frequency specified with the `-f` switch is applied to all CUs. Clock frequency for individual CUs cannot be changed independently. In addition, the `xclbin` must be programmed and be capable or running at the specified clock frequency. CUs generated with the Vitis tools only have clock1. RTL-based kernels can have clock2 and clock3 connected.

The following table lists the available options.
**Table 48: xbutil clock Command Options**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
<th>Required</th>
</tr>
</thead>
<tbody>
<tr>
<td>-d &lt;card&gt;</td>
<td>Specifies the target card. &lt;card&gt; can be specified as either the card_id or Bus:Device:Function (BDF). Defaults to card_id = 0 if not specified.</td>
<td>N</td>
</tr>
<tr>
<td></td>
<td><strong>Note:</strong> Use the <code>xbutil scan</code> command to display both the card_id and BDF for installed cards.</td>
<td></td>
</tr>
<tr>
<td>-r &lt;region&gt;</td>
<td>Deprecated and has no effect. Option will be removed in subsequent release.</td>
<td>N</td>
</tr>
<tr>
<td>-f &lt;clock1_freq_MHz&gt;</td>
<td>Specifies clock frequency (in MHz) for the first clock. All platforms have this clock.</td>
<td>Y</td>
</tr>
<tr>
<td>-g &lt;clock2_freq_MHz&gt;</td>
<td>Specifies clock frequency (in MHz) for the second clock. Some platforms might not have this clock.</td>
<td>N</td>
</tr>
<tr>
<td>-h &lt;clock3_freq_MHz&gt;</td>
<td>Specifies clock frequency (in MHz) for the third clock. Some platforms might not have this clock.</td>
<td>N</td>
</tr>
</tbody>
</table>

Use the `xclbinutil` Utility tool to list the available `xclbin` clocks.

It is necessary to program the `xclbin` prior to changing the clock frequency. See `program` to program the `xclbin`. Once the `xclbin` is programmed, the clock frequency can be changed.

For example, to change clock1 in card_ID = 0 to 100 MHz, run the following command:

```
xbutil clock -d 0 -f 100
```

Similarly, to change two clocks in card_ID = 0, such that clock1 is set to 200 MHz and clock2 is set to 250 MHz, run this command:

```
xbutil clock -d 0 -f 200 -g 250
```

The following example is an output after running this command successfully:

```
INFO: Found total 1 card(s). 1 are usable
INFO: xbutil clock succeeded.
```

If no `xclbin` is programmed, a message similar to the following will be displayed. Program the `xclbin` before running the `clock` command.

```
INFO: Found total 1 card(s). 1 are usable
WARNING: 'uuid' invalid, unable to find uuid.
Has the bitstream been loaded? See 'xbutil program'.
ERROR: xbutil clock failed.
```
**dmatest**

**IMPORTANT! This option cannot be used with embedded processor platforms.**

The `dmatest` command is used to validate the card memory throughput by performing data transfer tests between the host machine and global memory on a specified card. The `dmatest` is run as part of the `validate` command.

It has the following command line format:

```bash
xbutil dmatest [-d card] [-b [0x]block_size_KB]
```

The following table lists the available options.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
<th>Required</th>
</tr>
</thead>
<tbody>
<tr>
<td>-d <code>&lt;card&gt;</code></td>
<td>Specifies the target card. <code>&lt;card&gt;</code> can be specified as either the card_id or Bus:Device:Function (BDF). Defaults to card_id = 0 if not specified. Note: Use the <code>xbutil scan</code> command to display both the card_id and BDF for installed cards.</td>
<td>N</td>
</tr>
<tr>
<td>-b <code>blocksize</code></td>
<td>Specifies the test block size (in KB). Block size defaults to 65536 (KB) if <code>-b</code> option is not specified. Block size must be a power of 2. The block size can be specified in both decimal or hexadecimal formats. For example, both <code>-b 1024</code> and <code>-b 0x400</code> set the block size to 1024 KB.</td>
<td>N</td>
</tr>
</tbody>
</table>

It is necessary to program the `xclbin` prior to running `dmatest`. See program to program the `xclbin`.

The `dmatest` command only performs throughput tests on those DDR banks or HBM pseudo-channels (PCs) accessed by the `xclbin` programmed to the card.

An example of the command output on a U200 with an `xclbin` using DDR banks 0, 1, 2, and 3 is shown below:

```
INFO: Found total 1 card(s), 1 are usable
Total DDR size: 65536 MB
Reporting from mem_topology:
Data Validity & DMA Test on bank0
Host -> PCIe -> FPGA write bandwidth = 11341.5 MB/s
Host <-> PCIe <-> FPGA read bandwidth = 11097.3 MB/s
Data Validity & DMA Test on bank1
Host -> PCIe -> FPGA write bandwidth = 11414.6 MB/s
Host <-> PCIe <-> FPGA read bandwidth = 10981.7 MB/s
Data Validity & DMA Test on bank2
Host -> PCIe -> FPGA write bandwidth = 11345.1 MB/s
```
Host <- PCIe <- FPGA read bandwidth = 11189.2 MB/s
Data Validity & DMA Test on bank3
Host -> PCIe -> FPGA write bandwidth = 11121.7 MB/s
Host <- PCIe <- FPGA read bandwidth = 11375.7 MB/s
INFO: xbutil dmatest succeeded.

Similarly, an example of the command output on a U50 with an xclbin using HBM port 0, 1, 2, and 3 is shown below:

INFO: Found total 1 card(s), 1 are usable
Total DDR size: 0 MB
Reporting from mem_topology:
Data Validity & DMA Test on HBM[0]
Buffer Size: 256 MB
Host -> PCIe -> FPGA write bandwidth = 11950.9 MB/s
Host <- PCIe <- FPGA read bandwidth = 11940.3 MB/s
Data Validity & DMA Test on HBM[1]
Buffer Size: 256 MB
Host -> PCIe -> FPGA write bandwidth = 11947 MB/s
Host <- PCIe <- FPGA read bandwidth = 11958.1 MB/s
Data Validity & DMA Test on HBM[2]
Buffer Size: 256 MB
Host -> PCIe -> FPGA write bandwidth = 12077.2 MB/s
Host <- PCIe <- FPGA read bandwidth = 12064.1 MB/s
Data Validity & DMA Test on HBM[3]
Buffer Size: 256 MB
Host -> PCIe -> FPGA write bandwidth = 11989.5 MB/s
Host <- PCIe <- FPGA read bandwidth = 11976 MB/s
INFO: xbutil dmatest succeeded.

If no xclbin is programmed, a message similar to the following will be displayed.

INFO: Found total 1 card(s), 1 are usable
'uuid' invalid, please re-program xclbin.

dump

The dump command returns extensive card and system information including OS, XRT, board, electrical, thermal and xclbin in JSON format to allow for scripted flows. The output format and content are committed and will be backward compatible when changes are made to this command.

It has the following command line format:

```bash
xbutil dump [-d card]
```

The following table lists the available option.
### Table 50: `xbutil dump` Command Option

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
<th>Required</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-d &lt;card&gt;</code></td>
<td>Specifies the target card. <code>&lt;card&gt;</code> can be specified as either the card_id or Bus:Device:Function (BDF). Defaults to card_id = 0 if not specified.</td>
<td>N</td>
</tr>
</tbody>
</table>

**Note:** Use the `xbutil scan` command to display both the card_id and BDF for installed cards.

An example of the command output is shown below:

```json
{
  "version": "1.1.0",
  "system": {
    "sysname": "Linux",
    "release": "4.15.0-74-generic",
    "version": "#84-Ubuntu SMP Thu Dec 19 08:06:28 UTC 2019",
    "machine": "x86_64",
    "glibc": "2.27",
    "linux": "Ubuntu 18.04.3 LTS",
    "cores": "48",
    "memory": "31812",
    "model": "Precision 7920 Tower",
    "now": "Mon Jan 13 15:57:59 2020"
  },
  "runtime": {
    "build": {
      "version": "2.4.19",
      "hash": "be6279809c82b3b6abfd6a6baed6343bd4bda232",
      "date": "2020-01-09 10:57:59",
      "branch": "2019.2_PU1",
      "xocl": "2.4.19,be6279809c82b3b6abfd6a6baed6343bd4bda232",
      "xclmgmt": "2.4.19,be6279809c82b3b6abfd6a6baed6343bd4bda232"
    }
  },
  "board": {
    "info": {
      "dsa_name": "xilinx_u250_xdma_201830_2",
      "vendor": "0x10ee",
      "device": "0x5005",
      "subdevice": "0x000e",
      "subvendor": "0x10ee",
      "xmcversion": "2019107",
      "xmc_oem_id": "0x0",
      "serial_number": "21320493802N",
      "max_power": "225W",
      "sc_version": "4.2.0",
      "ddr_size": "68719476736",
      "ddr_count": "4",
      "clock0": "250",
      "clock1": "500",
      "clock2": "0",
      "pcie_speed": "3",
      "pcie_width": "16",
      "dma_threads": "2",
      "mig_calibrated": "true",
      "idcode": "0x4b57093",
      "fpga_name": "xcu250-fgdd2104-2L-e",
      "dna": ""
    }
  }
}
```
'p2p_enabled': '0',
},
'physical': {
  'thermal': {
    'pcb': {
      'top_front': '51',
      'top_rear': '41',
      'btm_front': '50'
    },
    'fpga_temp': '53',
    'tcrit_temp': '51',
    'fan_presence': 'A',
    'fan_speed': '1262',
    'cage': {
      'temp0': '0',
      'temp1': '0',
      'temp2': '0',
      'temp3': '0'
    }
  },
  'electrical': {
    '12v_pex': {
      'voltage': '12100',
      'current': '1480'
    },
    '12v_aux': {
      'voltage': '12121',
      'current': '1369'
    },
    '3v3_pex': {
      'voltage': '3349',
      'current': '0'
    },
    '3v3_aux': {
      'voltage': '3316'
    },
    'ddr_vpp_bottom': {
      'voltage': '2500'
    },
    'ddr_vpp_top': {
      'voltage': '2500'
    },
    'sys_5v5': {
      'voltage': '5492'
    },
    '1v2_top': {
      'voltage': '1207'
    },
    '1v2_btm': {
      'voltage': '1199'
    },
    '1v8': {
      'voltage': '1824'
    },
    '0v85': {
      'voltage': '856',
      'current': '0'
    },
    'mgt_0v9': {
      'voltage': '908'
    },
    '12v_sw': {
      'voltage': '12038'
    }
  }
}
<table>
<thead>
<tr>
<th>Bank</th>
<th>Type</th>
<th>Temp</th>
<th>Tag</th>
<th>Enabled</th>
<th>Size</th>
<th>Mem Usage</th>
<th>Bo Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>MEM_DDR4</td>
<td>41</td>
<td>bank1</td>
<td>true</td>
<td>16 GB</td>
<td>0 Byte</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>MEM_DDR4</td>
<td>54</td>
<td>bank2</td>
<td>true</td>
<td>16 GB</td>
<td>0 Byte</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>MEM_DDR4</td>
<td>48</td>
<td>bank3</td>
<td>true</td>
<td>16 GB</td>
<td>0 Byte</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>UNUSED</td>
<td>0</td>
<td>PLRAM[0]</td>
<td>false</td>
<td>128 KB</td>
<td>0 Byte</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>UNUSED</td>
<td>0</td>
<td>PLRAM[1]</td>
<td>false</td>
<td>128 KB</td>
<td>0 Byte</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>UNUSED</td>
<td>0</td>
<td>PLRAM[2]</td>
<td>false</td>
<td>128 KB</td>
<td>0 Byte</td>
<td>0</td>
</tr>
</tbody>
</table>
If an invalid card index is supplied, the following message will be displayed:

ERROR: Card index 1 is out of range

**m2mtest**

**IMPORTANT! This option cannot be used with embedded processor platforms.**
The `m2mtest` command performs throughput data transfer tests between two device memory banks on a specified card. Note, only platforms supporting M2M feature, see *Alveo Data Center Accelerator Card Platforms User Guide* (UG1120), can run this command. In addition, it is necessary to download an `xclbin` on the card which uses at least two memory banks prior to running `m2mtest`, else running this command returns an error. The `m2mtest` command only performs throughput tests on those memory banks accessed by the `xclbin` downloaded to the card.

It has the following command line format:

```
xbutil m2mtest [-d card]
```

The following table lists the available option.

**Table 51: xbutil m2mtest Command Option**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
<th>Required</th>
</tr>
</thead>
<tbody>
<tr>
<td>-d <code>&lt;card&gt;</code></td>
<td>Specifies the target card. <code>&lt;card&gt;</code> can be specified as either the card_id or Bus:Device:Function (BDF). Defaults to card_id = 0 if not specified.</td>
<td>N</td>
</tr>
</tbody>
</table>

*Note:* Use the `xbutil scan` command to display both the card_id and BDF for installed cards.

An example of the command output with an `xclbin` using DDR banks 0, 1, 2, and 3 is shown below:

```
INFO: Found total 2 card(s), 2 are usable
bank0 -> bank1 M2M bandwidth: 12050.5 MB/s
bank0 -> bank2 M2M bandwidth: 12074.3 MB/s
bank0 -> bank3 M2M bandwidth: 12082.9 MB/s
bank1 -> bank2 M2M bandwidth: 12061.8 MB/s
bank1 -> bank3 M2M bandwidth: 12105.2 MB/s
bank2 -> bank3 M2M bandwidth: 12065.8 MB/s
INFO: xbutil m2mtest succeeded.
```

If no `xclbin` has been loaded, the following error message will be displayed:

```
'uuid' invalid, please re-program xclbin.
```

If the command is run on platforms not supporting M2M feature, the following error will be displayed:

```
M2M is not available. Skipping validation
ERROR: xbutil m2mtest failed.
```
mem --read

**IMPORTANT! This option cannot be used with embedded processor platforms.**

The `mem --read` command reads the indicated number of bytes starting at a specified memory address and writes the contents into an output file.

It has the following command line format:

```
xbutil mem --read [-d card] [-a [0x]start_addr] [-i size_bytes] [-o output filename]
```

The following table lists the available options.

**Table 52: xbutil mem --read Command Options**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
<th>Required</th>
</tr>
</thead>
<tbody>
<tr>
<td>-d &lt;card&gt;</td>
<td>Specifies the target card. <code>&lt;card&gt;</code> can be specified as either the card_id or Bus:Device:Function (BDF). Defaults to card_id = 0 if not specified. Note: Use the <code>xbutil scan</code> command to display both the card_id and BDF for installed cards.</td>
<td>N</td>
</tr>
<tr>
<td>-a &lt;start_addr&gt;</td>
<td>Specifies a valid starting address in either hex or decimal format. Hex format requires leading <code>0x</code> that is, <code>0x100</code>. Default address is <code>0x0</code>. Valid addresses can be obtained using the Linux <code>dmesg</code> command as outlined below.</td>
<td>N</td>
</tr>
<tr>
<td>-i &lt;size_bytes&gt;</td>
<td>Specifies the memory transfer size in bytes in either hex or decimal format. Hex format requires leading <code>0x</code> that is, <code>0x100</code>. Default size is <code>0x20000</code>.</td>
<td>N</td>
</tr>
<tr>
<td>-o &lt;output_file_name&gt;</td>
<td>Output file name. Default output file is <code>memread.out</code> if output file name not specified.</td>
<td>N</td>
</tr>
</tbody>
</table>

An example of the output using the following command with `xclbin` using DDR banks 0, 1, 2, and 3 shown below:

```
xbutil mem --read -a 0x0 -d2 -i 0x10
```

```
INFO: Found total 3 card(s), 3 are usable
INFO: Reading from single bank, 256 bytes from DDR/HBM/PLRAM address 0x4000000000
INFO: Read size 0x100 B. Total Read so far 0x100
INFO: Read data saved in file: memread.out: Num of bytes: 256 bytes
INFO: xbutil mem succeeded.
```
An example of the content of the file generated with the above command is given below. The Linux hex dump command `xxd` was used to display the file.

```
00000000: 3d3d 3d3d 5354 4152 5420 6f66 2044 4452  ====START of DDR
00000010: 2044 6174 613d 3d3d 3d3d 3d3d 3d3d 0a00   Data========..
00000020: 4141 4141 4141 4141 4141 4141 4141 4141   AAAAAAAAAAAAAA
00000030: 0a3d 3d3d 3d3d 454e 4420 6f66 2044 4452  .=====END of DDR
00000040: 2044 6174 613d 3d3d 3d3d 3d3d 3d3d 0a00   Data========..
```

The following error is returned if an invalid starting address is used. The starting address must be within the address space of the device. In this example 0x400 is an invalid starting address:

```
ERROR: Start address 0x400 is not valid
Available memory banks:
ERROR: xbutil mem failed.
```

**TIP:** Use `grep` to display the available address spaces. For instance, the following command displays the DDR memory base addresses:

```
dmesg | grep -A 10 -i ddr
```

The Linux `dmesg` output will give the base address for the various DDR memories. A sample of the output for DDR[1] is shown below:

```
[23174.283512] xocl 0000:a6:00.1: xocl_init_mem: Memory Bank: DDR[1]
[23174.283514] xocl 0000:a6:00.1: xocl_init_mem: Base Address:0x8000000000
[23174.283515] xocl 0000:a6:00.1: xocl_init_mem: Size:0x400000000
```

Replace the `-i ddr` search term above with the `-i hbm` to look up the base address for HBM memories.

To write a known byte pattern, see `mem --write`.

---

**mem --write**

**IMPORTANT! This option cannot be used with embedded processor platforms.**

The `mem --write` command writes a defined pattern to a specified set of memory locations.

It has the following command line format:

```
xbutil mem --write [-d card] [-a [0x]start_addr] [-i size_bytes] [-e pattern_byte]
```

The following table lists the available options.
Table 53: xbutil mem --write Command Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
<th>Required</th>
</tr>
</thead>
<tbody>
<tr>
<td>-d &lt;card&gt;</td>
<td>Specifies the target card. &lt;card&gt; can be specified as either the card_id or Bus:Device:Function (BDF). Defaults to card_id = 0 if not specified.</td>
<td>N</td>
</tr>
<tr>
<td></td>
<td><strong>Note:</strong> Use the xbutil scan command to display both the card_id and BDF for installed cards.</td>
<td></td>
</tr>
<tr>
<td>-a &lt;start_addr&gt;</td>
<td>Specifies a valid starting address in either hex or decimal format. Hex format requires leading 0x that is, 0x100. Default address is 0x0. Valid addresses can be obtained using the Linux dmesg command as outlined below.</td>
<td>N</td>
</tr>
<tr>
<td>-i &lt;size_bytes&gt;</td>
<td>Specifies the memory transfer size in bytes in either hex or decimal format. Hex format requires leading 0x that is, 0x100.</td>
<td>N</td>
</tr>
<tr>
<td>-e &lt;pattern&gt;</td>
<td>Specifies the byte pattern written to all defined byte locations in either hex or decimal format. Hex format requires leading 0x that is, 0xEF.</td>
<td>N</td>
</tr>
</tbody>
</table>

An example of the output using the following command with xclbin using DDR banks 0, 1, 2, and 3 shown below:

```
xbutil mem --write -a 0x0 -d2 -i 0x10 -e 0xef
```

INFO: Found total 1 card(s), 1 are usable
INFO: Writing to single bank, 16 bytes from DDR/HBM/PLRAM address 0x0
INFO: Writing DDR/HBM/PLRAM with 16 bytes of pattern: 0xef from address 0x0
INFO: xbutil mem succeeded.

The following error is returned if an invalid starting address is used. The starting address must be within the address space of the device. In this example 0x400 is an invalid starting address.

```
ERROR: Start address 0x400 is not valid
Available memory banks:
ERROR: xbutil mem failed.
```

TIP: Use grep to display the available address spaces. For instance, the following command displays the DDR memory base addresses:

```
dmesg | grep -A 10 -i ddr
```

The Linux dmesg output will give the base address for the various DDR memories. A sample of the output for DDR[1] is shown below:

```
[23174.283512] xocl 0000:a6:00.1: xocl_init_mem: Memory Bank: DDR[1] [23174.283514] xocl 0000:a6:00.1: xocl_init_mem: Base Address:0x8000000000
[23174.283515] xocl 0000:a6:00.1: xocl_init_mem: Size:0x400000000
```

Replace the -i ddr search term above with the -i hbm to look up the base address for HBM memories.
To read memory addresses, see `mem --read`.

---

## p2p

**IMPORTANT!** This option cannot be used with embedded processor platforms.

The `p2p` command is used to enable/disable (PCIe Peer-to-Peer Support) feature and check current configuration.

**IMPORTANT!** Only platforms supporting P2P feature, see Alveo Data Center Accelerator Card Platforms User Guide (UG1120), can run this command.

P2P configuration is persistent across warm reboot.

**Note:** Enabling or disabling P2P requires root privilege.

It has the following command line format:

```
xbutil p2p [-d card] --[enable | disable | validate]
```

The following table lists the available options.

**Table 54: `xbutil p2p` Command Options**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
<th>Required</th>
</tr>
</thead>
<tbody>
<tr>
<td>-d &lt;card&gt;</td>
<td>Specifies the target card. <code>&lt;card&gt;</code> can be specified as either the <code>card_id</code> or <code>Bus:Device:Function (BDF)</code>. Defaults to <code>card_id = 0</code> if not specified. &lt;br&gt;<strong>Note:</strong> Use the <code>xbutil scan</code> command to display both the <code>card_id</code> and <code>BDF</code> for installed cards.</td>
<td>N</td>
</tr>
<tr>
<td>--enable</td>
<td>Enables p2p. Option <code>--enable</code>, <code>--disable</code>, and <code>--validate</code> are mutually exclusive. Only one can be in the command. A warm reboot is required if the returned <code>xbutil query</code> status for P2P is <code>no_iomem</code>.</td>
<td>--enable, --disable, --validate mutually exclusive</td>
</tr>
<tr>
<td>--disable</td>
<td>Enables p2p. Option <code>--enable</code>, <code>--disable</code>, and <code>--validate</code> are mutually exclusive. Only one can be in the command. This might require warm reboot to fully disable.</td>
<td>--enable, --disable, --validate mutually exclusive</td>
</tr>
<tr>
<td>--validate</td>
<td>Validates p2p feature. Option <code>--enable</code>, <code>--disable</code>, and <code>--validate</code> are mutually exclusive. Only one can be in the command. Run after a warm reboot if it is needed.</td>
<td>--enable, --disable, --validate mutually exclusive</td>
</tr>
</tbody>
</table>
Use `xbutil query` to display the current status of P2P. The following is a partial output of the `xbutil query` command showing the current status under the heading **P2P Enabled**.

<table>
<thead>
<tr>
<th>PCIe</th>
<th>DMA chan(bidir)</th>
<th>MIG Calibrated</th>
<th>P2P Enabled</th>
</tr>
</thead>
<tbody>
<tr>
<td>GEN 3x16</td>
<td>2</td>
<td>true</td>
<td>false</td>
</tr>
</tbody>
</table>

**Table 55: P2P Enabled Returned Value Definition**

<table>
<thead>
<tr>
<th>Value</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>P2P is enabled.</td>
</tr>
<tr>
<td>false</td>
<td>P2P is disabled.</td>
</tr>
<tr>
<td>no_iomem</td>
<td>P2P is enabled in device but system could not allocate I/O memory, warm reboot is needed.</td>
</tr>
</tbody>
</table>

If P2P is not enabled when trying to validate, validation will be skipped and the following message will be displayed:

*P2P BAR is not enabled. Skipping validation*

If the card platform does not support P2P, the following message will be displayed:

*ERROR: P2P is not supported on this platform*

If no `xclbin` is programmed, the following message will be displayed:

*'uuid' invalid, please re-program xclbin.*

---

**program**

**IMPORTANT!** This option is supported for use with embedded processor DFX platforms.

The `program` command downloads a specified `xclbin` binary to the programmable region on the card.

It has the following command line format:

```
xbutil program [-d card] [-r region] -p <xclbin_filename>
```

The following table lists the available options.
Table 56: xbutil program Command Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
<th>Required</th>
</tr>
</thead>
<tbody>
<tr>
<td>-d &lt;card&gt;</td>
<td>Specifies the target card. &lt;card&gt; can be specified as either the card_id or Bus:Device:Function (BDF). Defaults to card_id = 0 if not specified. Note: Use the xbutil scan command to display both the card_id and BDF for installed cards.</td>
<td>N</td>
</tr>
<tr>
<td>-r &lt;region&gt;</td>
<td>Deprecated and has no effect. Option will be removed in subsequent release.</td>
<td>N</td>
</tr>
<tr>
<td>-p &lt;xclbin_filename&gt;</td>
<td>Specifies the file name of the xclbin binary file to download to the card.</td>
<td>Y</td>
</tr>
</tbody>
</table>

When an xclbin is successfully downloaded to the card, the following message is displayed:

INFO: Found total 1 card(s), 1 are usable
INFO: xbutil program succeeded.

If the subsequent use of the xbutil program uses the same xbutil_filename, the xbutil will not be downloaded as it already exists on the card, but the above message will be identical.

If the specified xclbin file does not exist, the following message will be displayed:

ERROR: Cannot open <my_xclbin>.xclbin. Check that it exists and is readable.
ERROR: xbutil program failed.

query

IMPORTANT! This option cannot be used with embedded processor platforms.

The query command returns detailed card status information in human readable format. See dump for output in JSON format.

It has the following command line format:

xbutil query [-d card [-r region]]

The following table lists the available options.
### Table 57: `xbutil query` Command Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
<th>Required</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-d &lt;card&gt;</code></td>
<td>Specifies the target card. <code>&lt;card&gt;</code> can be specified as either the card_id or Bus:Device:Function (BDF). Defaults to card_id = 0 if not specified. <strong>Note:</strong> Use the <code>xbutil scan</code> command to display both the card_id and BDF for installed cards.</td>
<td>N</td>
</tr>
<tr>
<td><code>-r &lt;region&gt;</code></td>
<td>Deprecated and has no effect. Option will be removed in subsequent release.</td>
<td>N</td>
</tr>
</tbody>
</table>

There is a significant amount of information returned. An example of the output is given below. The output has been divided into separate sections to better describe the content.

### System Configuration

### Table 58: System Configuration Field Definition

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OS Name</td>
<td>Name of the OS running on the machine</td>
</tr>
<tr>
<td>Release</td>
<td>OS release number</td>
</tr>
<tr>
<td>Version</td>
<td>OS Version</td>
</tr>
<tr>
<td>Machine</td>
<td>CPU-based architecture</td>
</tr>
<tr>
<td>Glibc</td>
<td>GLIBC version installed</td>
</tr>
<tr>
<td>Distribution</td>
<td>Distribution</td>
</tr>
<tr>
<td>Now</td>
<td>Current date and time</td>
</tr>
</tbody>
</table>

An example of the system configuration is shown below:

```
System Configuration
OS name:    Linux
Release:    4.15.0-74-generic
Machine:    x86_64
Glibc:      2.23
Distribution:   Ubuntu 16.04.6 LTS
Now:        Wed Jan 22 15:30:36 2020
```

### XRT Information

### Table 59: XRT Field Definition

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Version</td>
<td>XRT version</td>
</tr>
<tr>
<td>Git Hash</td>
<td>Associated GIT hash</td>
</tr>
<tr>
<td>Git Branch</td>
<td>Associated GIT branch</td>
</tr>
<tr>
<td>Build Date</td>
<td>XRT build date</td>
</tr>
</tbody>
</table>
**Table 59: XRT Field Definition (cont’d)**

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XOCL</td>
<td>XOCL version</td>
</tr>
<tr>
<td>XCLMGMT</td>
<td>XCLMGMT version</td>
</tr>
</tbody>
</table>

**XRT Information**

Version: 2.3.1301  
Git Hash: 192e706aea53163a04c574f9b3fe9ed76b6ca471  
Git Branch: 2019.2  
Build Date: 2019-10-24 20:04:29  
XOCL: 2.3.1301.192e706aea53163a04c574f9b3fe9ed76b6ca471  
XCLMGMT: 2.3.1301.192e706aea53163a04c574f9b3fe9ed76b6ca471

**Card Platform (Shell) Information**

**Table 60: Card Platform (Shell) Field Definition**

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shell</td>
<td>Platform installed on the card</td>
</tr>
<tr>
<td>FPGA</td>
<td>FPGA name</td>
</tr>
<tr>
<td>IDCode</td>
<td>ID code of platform</td>
</tr>
<tr>
<td>Vendor</td>
<td>Vendor ID</td>
</tr>
<tr>
<td>Device</td>
<td>Device ID</td>
</tr>
<tr>
<td>SubDevice</td>
<td>SubDevice ID</td>
</tr>
<tr>
<td>SubVendor</td>
<td>SubVendor ID</td>
</tr>
<tr>
<td>SerNum</td>
<td>Unique card serial number</td>
</tr>
<tr>
<td>DDR Size</td>
<td>Total amount of DDR RAM available on the card in GB</td>
</tr>
<tr>
<td>DDR Count</td>
<td>Total number of DDR DIMMs installed on the card</td>
</tr>
<tr>
<td>Clock0</td>
<td>Clock0 frequency in MHz</td>
</tr>
<tr>
<td>Clock1</td>
<td>Clock1 frequency in MHz</td>
</tr>
<tr>
<td>Clock2</td>
<td>Clock2 frequency in MHz</td>
</tr>
<tr>
<td>PCIe</td>
<td>Trained PCIe link status</td>
</tr>
<tr>
<td>DMA chan(bidir)</td>
<td>Number of DMA channels on the card</td>
</tr>
<tr>
<td>MIG Calibrated</td>
<td>When TRUE MIG has been calibrated, FALSE indicates the MIG has not been calibrated.</td>
</tr>
</tbody>
</table>
| P2P Enabled      | Returns status of P2P. Status will be one of the following:  
|                  | • true: P2P is enabled  
|                  | • false: P2P is disabled  
|                  | • no_iomem: P2P is enabled in device but system could not allocate I/O memory, warm reboot is needed |
| OEM ID           | ID used by OEMs                                       |
| Interface UUID   | A unique identifier which can be used to determine whether partial bitstreams containing the various partitions of the platform are logically and physically compatible with each other. |
Table 60: Card Platform (Shell) Field Definition (cont’d)

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logic UUID</td>
<td>A unique identifier which can be used to determine whether partial bitstreams containing the various partitions of the platform are logically and physically compatible with each other.</td>
</tr>
</tbody>
</table>

This is an example output of the card information:

Shell xilinx_u50_gen3x16_xdma_201920_3 IDCode 0x14b77093
Vendor 0x10ee Device 0x5021 SubDevice 0x000e SubVendor 0x10ee SerNum 00501201A030
DDR size 0 Byte DDR count Clock0 Clock1 Clock2
0 Byte 0 250 500 450
PCIe DMA chan(bidir) MIG Calibrated P2P Enabled OEM ID
GEN 3x16 2 true false 0x0
Interface UUID 862c7020a250293e32036f19956669e5
Logic UUID f465b0a3ae8c64f619bc150384ace69b
DNA

Temperature

Card power and thermal information are given next. Temperatures are reported in Celsius.

Table 61: Temperature Field Definition

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PCB Top Front</td>
<td>Temperature at the top front of the PCB in Celsius</td>
</tr>
<tr>
<td>PCB Top Rear</td>
<td>Temperature at the top rear of the PCB in Celsius</td>
</tr>
<tr>
<td>PCB BTM Front</td>
<td>Temperature at the bottom front of the PCB in Celsius</td>
</tr>
<tr>
<td>FPGA Temp</td>
<td>FPGA core temperature in Celsius</td>
</tr>
<tr>
<td>TCRIT Temp</td>
<td>Critical temperature in Celsius of the fan controller. Present for both active and passive cards.</td>
</tr>
<tr>
<td>Fan Presence</td>
<td>Represents the presence of a fan on the card.</td>
</tr>
<tr>
<td>Fan Speed</td>
<td>Fan speed (RPM). Returned as N/A for passive cards.</td>
</tr>
<tr>
<td>QSFP 0,1,2,3</td>
<td>Temperature in Celsius of the QSFP module</td>
</tr>
</tbody>
</table>
An example of the temperature output is given below:

<table>
<thead>
<tr>
<th></th>
<th>PCB TOP FRONT</th>
<th>PCB TOP REAR</th>
<th>PCB BTM FRONT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Temperature(C)</td>
<td>42</td>
<td>37</td>
<td>42</td>
</tr>
<tr>
<td>FPGA TEMP</td>
<td>44</td>
<td>42</td>
<td>A</td>
</tr>
<tr>
<td>TCRIT Temp</td>
<td>42</td>
<td>A</td>
<td>42</td>
</tr>
<tr>
<td>FAN Presence</td>
<td>A</td>
<td>1108</td>
<td></td>
</tr>
<tr>
<td>FAN Speed(RPM)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>QSFP 0</td>
<td>0</td>
<td>0</td>
<td>QSFP 1</td>
</tr>
<tr>
<td>QSFP 1</td>
<td>0</td>
<td>0</td>
<td>QSFP 2</td>
</tr>
<tr>
<td>QSFP 2</td>
<td>0</td>
<td>0</td>
<td>QSFP 3</td>
</tr>
</tbody>
</table>

**Electrical**

This provides various voltage (mV) and current (mA) readings.

*Table 62: Electrical Field Definition*

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>12V PEX</td>
<td>Voltage measurement (12V nominal) from the 12V Power source from the PCIe connector.</td>
</tr>
<tr>
<td>12V AUX</td>
<td>Voltage measurement (12V nominal) from the 12V, 6 or 8-pin PCIe AUX power cable.</td>
</tr>
<tr>
<td>12V PEX Current</td>
<td>Current measurement of the power drawn by the PCIe connector.</td>
</tr>
<tr>
<td>12V AUX Current</td>
<td>Current measurement of the power drawn by the 6 or 8-pin PCIe AUX power cable.</td>
</tr>
<tr>
<td>3V3 PEX</td>
<td>Voltage measurement (3.3V nominal) of the 3.3V Power source from the PCIe connector.</td>
</tr>
<tr>
<td>3V3 AUX</td>
<td>Voltage measurement (3.3V nominal) of the 3.3V AUX Power sourced from the PCIe connector.</td>
</tr>
<tr>
<td>DDR VPP BOTTOM</td>
<td>Voltage measurement (2.5V nominal) for powering the onboard VPP for the DDR4 parts on the lower portion of the card.</td>
</tr>
<tr>
<td>DDR VPP TOP</td>
<td>Voltage measurement (2.5V nominal) for powering the onboard VPP for the DDR4 parts on the upper portion of the card.</td>
</tr>
<tr>
<td>SYS 5V5</td>
<td>Voltage measurement (5.5V nominal) used for powering the onboard VCC_INT regulators. Only on U2XX cards.</td>
</tr>
<tr>
<td>1V2 TOP</td>
<td>Voltage measurement (1.2V nominal) for powering the onboard VDD for the DDR4 parts on the upper portion of the card.</td>
</tr>
<tr>
<td>1V8 TOP</td>
<td>Voltage measurement (1.8V nominal) for powering the onboard VCCAUX, VCCAUXIO, and MGTAVVCAUX regulator used by the FPGA.</td>
</tr>
<tr>
<td>0V85</td>
<td>Voltage measurement (0.85V nominal) of the onboard VCCINTIO and VCCBRAM regulator used by the FPGA.</td>
</tr>
<tr>
<td>MGT 0V9</td>
<td>Voltage measurement (0.9V nominal) of the onboard MGTAVCC regulator for the GTys used by the FPGA.</td>
</tr>
<tr>
<td>12V SW</td>
<td>Voltage measurement (12V nominal) of the 12V, 6 or 8-pin PCIe AUX power cable.</td>
</tr>
<tr>
<td>MGT VTT</td>
<td>Voltage measurement (1.2V nominal) of the onboard MGTAUVT regulator for the GTys used by the FPGA.</td>
</tr>
<tr>
<td>1V2 BTM</td>
<td>Voltage measurement (1.2V nominal) for powering the onboard VDD regulator for the DDR4 parts on the lower portion of the card.</td>
</tr>
<tr>
<td>VCCINT VOL</td>
<td>Voltage measurement (0.72-0.85V nominal) of the onboard VCCINT regulator for the FPGA.</td>
</tr>
<tr>
<td>VCCINT CURR</td>
<td>Current measurement of the VCCINT supply drawn by the card.</td>
</tr>
<tr>
<td>VCCINT BRAM VOL</td>
<td>Voltage measurement (0.85V nominal) of the onboard VCCINT, VCCINTIO, and VCCBRAM regulator used by the FPGA.</td>
</tr>
</tbody>
</table>
Table 62: Electrical Field Definition (cont’d)

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VCC3V3 VOL</td>
<td>Voltage measurement (3.3V nominal) of the onboard 3.3V regulator used by the QSFPs, and other circuits.</td>
</tr>
<tr>
<td>3V3 PEX CURR</td>
<td>Current measurement of the 3.3V motherboard PCIe rail drawn by the card.</td>
</tr>
<tr>
<td>VCC0V85 CURR</td>
<td>Current measurement of the VCCINTIO and VCCBRAM supply drawn by the card.</td>
</tr>
<tr>
<td>HBM1V2 VOL</td>
<td>Voltage of 1.2V for powering the onboard VDD on the DDR4 HBM parts on the card.</td>
</tr>
<tr>
<td>VPP2V5 VOL</td>
<td>Voltage of 2.5V for powering the onboard VPP on the DDR4 parts of the card.</td>
</tr>
</tbody>
</table>

An example of the output is given below:

| Electrical (mV|mA) |  |  |  |  |
|-------|-------|-------|-------|-------|
| 12V PEX | 12V AUX | 12V PEX Current | 12V AUX Current |
| 12101 | 12202 | 1505 | 1268 |
| 3V3 PEX | 3V3 AUX | DDR VPP BOTTOM | DDR VPP TOP |
| 3357 | 3262 | 2500 | 2500 |
| SYS 5V5 | 1V2 TOP | 1V8 TOP | 0V85 |
| 5515 | 1204 | 1836 | 855 |
| MGT 0V9 | 12V SW | MGT VTT | 1V2 BTM |
| 910 | 12064 | 1207 | 1209 |
| VCCINT VOL | VCCINT CURR | VCCINT BRAM VOL | VCC3V3 VOL |
| 851 | 15894 | 0 | 0 |
| 3V3 PEX CURR | VCC0V85 CURR | HBM1V2 VOL | VPP2V5 VOL |
| 0 | 0 | 0 | 0 |

Card Power

Single field returning the total power (W) being consumed by the card.

An example of the output is given below:

Card Power (W)

33

Firewall Last Error Status

The firewall provides information when an error has been detected in hardware. This includes a timestamp and the level of the firewall. The firewall has three levels, as discussed in AXI Firewall Trips. In the following output, there are no detected firewall errors.

Table 63: Firewall Last Error Status Field Definition

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tag</td>
<td>Name of memory bank</td>
</tr>
<tr>
<td>Errors</td>
<td>Indicates if an error occurred</td>
</tr>
<tr>
<td>CE Count</td>
<td>Number of correctable errors. Number is persistent, however can be reset through xbmgt reset.</td>
</tr>
<tr>
<td>UE Count</td>
<td>Number of uncorrectable errors. The count is persistent, but it can be reset using xbmgt reset.</td>
</tr>
</tbody>
</table>
An example of the output is given below:

<table>
<thead>
<tr>
<th>Firewall Last Error Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Level 0 : 0x0(GOOD)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ECC Error Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tag     Errors      CE Count UE Count CE FFA UE FFA</td>
</tr>
<tr>
<td>bank0   (None)      0      0      0x0     0x0</td>
</tr>
<tr>
<td>bank1   (None)      0      0      0x0     0x0</td>
</tr>
<tr>
<td>bank2   (None)      0      0      0x0     0x0</td>
</tr>
<tr>
<td>bank3   (None)      0      0      0x0     0x0</td>
</tr>
</tbody>
</table>

On some cards, for example the U50, the Satellite Controller (SC) monitors operating conditions. If the card exceeds electrical or thermal limits, the SC will reset the workload on the card. In some cases, this will be seen in the `xbutil` query output as a firewall trip. It will display the time the trip occurred. An example of the post trip state is given below:

<table>
<thead>
<tr>
<th>Firewall Last Error Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Level 3 : 0x80004(RECS_CONTINUOUS_RTRANSFERS_MAX_WAIT</td>
</tr>
<tr>
<td>Error occurred on: Tue 2020-04-28 15:16:47 MDT</td>
</tr>
</tbody>
</table>

The card should be okay to use in this state.

**Memory Status**

The memory topology along with the DMA transfer metrics are provided next, followed by streaming transfers. The DMA metrics include the transfer of data between the host and card. Host to card transfers are indicated by `h2c`, while card to host transfer are defined by `c2h`.

An example of the output is given below. If no `xclbin` has been loaded, no memory status will be displayed.

<table>
<thead>
<tr>
<th>Memory Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tag         Type        Temp(C) Size    Mem Usage       BO count</td>
</tr>
<tr>
<td>[ 0] bank0       MEM_DDR4    37       16 GB   16 MB           1</td>
</tr>
<tr>
<td>[ 1] bank1       MEM_DDR4    39       16 GB   16 MB           1</td>
</tr>
<tr>
<td>[ 2] bank2       MEM_DDR4    47       16 GB   16 MB           1</td>
</tr>
<tr>
<td>[ 3] bank3       MEM_DDR4    43       16 GB   16 MB           1</td>
</tr>
<tr>
<td>[ 4] PLRAM[0]    <strong>UNUSED</strong>  N/A      128 KB 0 Byte          0</td>
</tr>
<tr>
<td>[ 5] PLRAM[1]    <strong>UNUSED</strong>  N/A      128 KB 0 Byte          0</td>
</tr>
<tr>
<td>[ 6] PLRAM[2]    <strong>UNUSED</strong>  N/A      128 KB 0 Byte          0</td>
</tr>
</tbody>
</table>

**DMA Transfer Metrics**

An example of the output is given below. If no `xclbin` has been loaded, no metrics will be displayed.

<table>
<thead>
<tr>
<th>DMA Transfer Metrics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chan[0].h2c:  12384 MB</td>
</tr>
<tr>
<td>Chan[0].c2h:  15200 MB</td>
</tr>
<tr>
<td>Chan[1].h2c:  6240 MB</td>
</tr>
<tr>
<td>Chan[1].c2h:  6144 MB</td>
</tr>
</tbody>
</table>
Streams

This is available for streaming platforms only.

An example of the output is given below:

<table>
<thead>
<tr>
<th>Tag</th>
<th>Flow ID</th>
<th>Route ID</th>
<th>Status</th>
<th>Total (B/#)</th>
<th>Pending (B/#)</th>
</tr>
</thead>
</table>

Xclbin UUID

This displays the xclbin UUID. An example of the output is given below. If no xclbin has been loaded, it will return all zeros as the UUID.

| Xclbin UUID | dfd5a66a-36aa-41c6-88bb-c85a86d15512 |

Compute Unit Status

The Compute Units (CU) present in the xclbin loaded to the card are displayed. For each CU, it displays the name, PCIe BAR address, and the status, which can be IDLE, START, and DONE. The output below shows the xclbin ID and two CUs both with IDLE status.

An example of the output is given below. If no xclbin has been loaded, no CU status will be displayed.

| Compute Unit Status | CU[ 1]: bandwidth1:kernel_1         @0x1c00000         (IDLE) | CU[ 0]: bandwidth2:kernel_2         @0x1800000         (IDLE) |

reset

**IMPORTANT! This option cannot be used with embedded processor platforms.**

The reset command resets the programmable region on the card. All running CUs in the region are stopped and reset.

It has the following command line format:

```
xbutil reset [-d card]
```

The following table lists the available option.
### Table 64: `xbutil reset` Command Option

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
<th>Required</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-d &lt;card&gt;</code></td>
<td>Specifies the target card. <code>&lt;card&gt;</code> can be specified as either the <code>card_id</code> or <code>Bus:Device:Function</code> (BDF). Defaults to <code>card_id = 0</code> if not specified. <strong>Note:</strong> Use the <code>xbutil scan</code> command to display both the <code>card_id</code> and BDF for installed cards.</td>
<td>N</td>
</tr>
</tbody>
</table>

The following example shows the output after running this command successfully:

```
All existing processes will be killed.
Are you sure you wish to proceed? [y/n]: y
```

---

## `scan` (xbutil)

The `xbutil scan` command returns detailed system information including:

- System configuration details
- XRT information
- List of all cards installed on the system, except cards in GOLDEN state.

It has the following command line format and does not have any options.

```
xbutil scan
```

The following tables lists the fields returned from various sections of the `xbutil scan` command.

### Table 65: System Configuration Field Definition

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OS Name</td>
<td>Name of the OS running on the machine</td>
</tr>
<tr>
<td>Release</td>
<td>OS release number</td>
</tr>
<tr>
<td>Version</td>
<td>OS Version</td>
</tr>
<tr>
<td>Machine</td>
<td>CPU-based architecture</td>
</tr>
<tr>
<td>Model</td>
<td>Machine model</td>
</tr>
<tr>
<td>CPU Cores</td>
<td>Number CPU cores on the machine</td>
</tr>
<tr>
<td>Memory</td>
<td>Total installed memory on the machine in MB</td>
</tr>
<tr>
<td>Glibc</td>
<td>GLIBC version installed</td>
</tr>
<tr>
<td>Distribution</td>
<td>Distribution</td>
</tr>
<tr>
<td>Now</td>
<td>Current date and time</td>
</tr>
</tbody>
</table>
Table 66: XRT Field Definition

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Version</td>
<td>XRT version</td>
</tr>
<tr>
<td>Git Hash</td>
<td>Associated GIT hash</td>
</tr>
<tr>
<td>Git Branch</td>
<td>Associated GIT branch</td>
</tr>
<tr>
<td>Build Date</td>
<td>XRT build date</td>
</tr>
<tr>
<td>XOCL</td>
<td>XOCL version</td>
</tr>
<tr>
<td>XCLMGMT</td>
<td>XCLMGMT version</td>
</tr>
</tbody>
</table>

A list of each card installed on the system will also be returned. A separate line for each card will be displayed. An example output for one card is shown below. It provides multiple fields detailing the installed card. The fields are separated with a space.

```
[0] 0000:65:00.1 xilinx_u50_gen3x16_xdma_201920_3 user(inst=128)
```

The fields are defined in the following table.

Table 67: Installed Cards Field Definition

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[card_id]</td>
<td>Provides an assigned card_id based on the card enumeration in the driver data structures. A unique card_id is assigned for each card detected. The order can change after a warm or cold reboot.</td>
</tr>
<tr>
<td>Bus : Device : Function</td>
<td>Provides the enumerated card Bus:Device:Function (BDF) for each card installed. It has the following format: [ Bus : Device : Function ] A card’s BDF is determined based on the PCIe slot it is plugged into. Note: Use the xbutil scan command to display both the card_id and BDF for installed cards.</td>
</tr>
<tr>
<td>Platform name</td>
<td>Platform name in the following format: <code>&lt;company&gt;_&lt;card&gt;_&lt;customization&gt;_&lt;major_release&gt;_&lt;minor_release&gt;</code> See Alveo Data Center Accelerator Card Platforms User Guide (UG1120) for platform naming information.</td>
</tr>
<tr>
<td>user (inst = &lt;value&gt;)</td>
<td>Returns the user function instance number associated with the card. The instance number allows you to easily find the device node for each function. In Linux OS, the device node can be found at: <code>/dev/dri/renderD&lt;inst&gt;</code> In addition, the instance can be useful when mapping the <code>dmesg</code> information to a specific card.</td>
</tr>
</tbody>
</table>
Below is an example of the `xbutil scan` output. The system configuration and XRT information sections are first displayed followed by the detected card(s). In the below example, one card is detected and assigned card_ID 0 and BDF is `0000:65:00.1`. The platform flashed and running on the card is `xilinx_u50_gen3x16_xdma_201920_3` and the user instance has been assigned 128.

INFO: Found total 1 card(s), 1 are usable
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

**System Configuration**
- OS name: Linux
- Release: 4.15.0-96-generic
- Version: #97~16.04.1-Ubuntu SMP Wed Apr 1 03:03:31 UTC 2020
- Machine: x86_64
- Model: Super Server
- CPU cores: 16
- Memory: 15703 MB
- Glibc: 2.23
- Distribution: Ubuntu 16.04.6 LTS
- Now: Tue Apr 14 21:08:05 2020

**XRT Information**
- Version: 2.5.309
- Git Hash: 9a03790c11f066a5597b133db737cf4683ad84c8
- Git Branch: 2019.2 PU2
- Build Date: 2020-02-23 18:51:37
- XOCL: 2.5.309,9a03790c11f066a5597b133db737cf4683ad84c8
- XCLMGMT: 2.5.309,9a03790c11f066a5597b133db737cf4683ad84c8

```
[0] 0000:65:00.1 xilinx_u50_gen3x16_xdma_201920_3 user(inst=128)
```

In cases where multiple cards are installed, the list of detected cards will be expanded. In the example shown below, three cards are detected and assigned card_ID 0, 1, and 2, respectively.

```
[0] 0000:a6:00.1 xilinx_u280_xdma_201920_2 user(inst=130)
[1] 0000:73:00.1 xilinx_u250_xdma_201830_2 user(inst=129)
[2] 0000:17:00.1 xilinx_u200_xdma_201830_2 user(inst=128)
```

An asterisk proceeding the listed card indicates the card it not ready. An example message given for unusable card is given below:

```
*[0] 0000:a6:00.1 xilinx_u280_xdma_201920_2(ts=0x5e172e16) user(inst=130)
WARNING: card(s) marked by '*' are not ready, run xbmgmt flash --scan --verbose to further check the details.
```

**Note:** Cards in golden state (no partition flashed) will not be displayed.

---

**status**

The `status` command reports the status of the Vitis performance monitor (SPM) and lightweight AXI protocol checker (LAPC) debug IPs contained in the `xclbin` programmed to the card.
It has the following command line format:

```
xbutil status [-d <card>] [--debug_ip_name]
```

The following table lists the available options.

**Table 68: xbutil status Command Options**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
<th>Required</th>
</tr>
</thead>
<tbody>
<tr>
<td>-d &lt;card&gt;</td>
<td>Specifies the target card. &lt;card&gt; can be specified as either the</td>
<td>N</td>
</tr>
<tr>
<td></td>
<td>card_id or Bus:Device:Function (BDF). Defaults to card_id = 0 if not</td>
<td></td>
</tr>
<tr>
<td></td>
<td>specified.</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Note:</strong> Use the xutil scan command to display both the</td>
<td></td>
</tr>
<tr>
<td></td>
<td>card_id and BDF for installed cards.</td>
<td></td>
</tr>
<tr>
<td>--&lt;ip_name&gt;</td>
<td>Returns status of the specified debug IP.</td>
<td>N</td>
</tr>
</tbody>
</table>

The `status` command displays the type and number of the debug IP on the accelerator card.

```
xutil status
```

An example output of the command is shown below. It lists all the debug IP found.

```
INFO: Found total 1 card(s), 1 are usable
Number of IPs found: 9
IPs found [<ipname>[<count>]]: aim(5) tracefunnel(1) monitorfifolite(1)
monitorfifofull(1) accelmonitor(1)
Run 'xbutil status' with option --<ipname> to get more information about
the IP
INFO: xbutil status succeeded.
```

The list of available IP is determined by the `xclbin` file compiled for use on the accelerator card and include:

- **Accelerator Monitor (AM):** Count and trace the executions of compute units. Performance Monitors are added using the `--profile.exec` option as discussed in described in `--profile Options`.

- **AXI Interface Monitor (AIM):** Count and trace transactions on AXI4 connections.

- **AXI4-Stream Monitor (ASM):** Count and trace transactions on AXI4-Stream.

- **Lightweight AXI Protocol Monitor (LAPC):** Protocol checking of AXI4. Protocol checkers are added using the `--debug.protocol` option as described in `--debug Options`.

- **Streaming Protocol Checker (SPC):** Protocol checking of AXI4-Stream.

- **Trace Funnel:** Collects trace events from all monitors. If present, then trace was enabled using the `--profile.data` option during compilation as described in `--profile Options`, and using the `timeline_trace` option during runtime as described in `xrt.ini File`. 
• **FIFO Lite:** Control of the PL FIFO that stores trace events. If present, then trace was enabled during compilation and runtime, and memory offload was specified to be a FIFO in the PL using the `--trace_memory` option during compilation as described in Vitis Compiler General Options. Trace behavior is influenced by the settings in the `xrt.ini` File.

• **FIFO Full:** The data offload of the PL FIFO that stores trace events. If present, then trace was enabled during compilation and runtime, and memory offload was specified to be a FIFO in the PL using the `--trace_memory` option during compilation.

• **TS2MM:** Takes trace events and offloads them to a memory resource (DDR, HBM, PLRAM). If present, then trace was enabled during compilation and runtime, and memory offload was specified to be a memory resource using the `--trace_memory` option during compilation as described in Vitis Compiler General Options.

You can get the status of specific IP using the following command syntax:

```
$ xbutil status --<ipname>
```

An example output using the `--aim` option is shown below:

```
$ xbutil status --aim

INFO: Found total 1 card(s), 1 are usable
AXI Interface Monitor Counters
Region or CU Type or Port           Wr Bytes Wr Trans. Rd Bytes Rd Tranx. Outstanding Cnt
runOnfpga_1  m_axi_maxiport0-DDR[1] 0        0         0        0         0
runOnfpga_1  m_axi_maxiport1-DDR[1] 0        0         0        0         0
shell        Memory to Memory       0        0         0        0         0
shell        Host to Device         0        0         0        0         0
shell        Peer to Peer           0        0         0        0         0
INFO: xbutil status succeeded.
```

The following is a code continuation of the columns:

```
INFO: Found total 1 card(s), 1 are usable
AXI Interface Monitor Counters
Region or CU Type or Port           Last Wr Addr Last Wr Data Last Rd Addr Last Rd Data
runOnfpga_1  m_axi_maxiport0-DDR[1] 0x0        0x0         0x0        0x0
runOnfpga_1  m_axi_maxiport1-DDR[1] 0x0        0x0         0x0        0x0
shell        Memory to Memory       0x0       0x0         0x0        0x0
shell        Host to Device         0x0       0x0         0x0        0x0
shell        Peer to Peer           0x0       0x0         0x0        0x0
INFO: xbutil status succeeded.
```

If no debug IPs are found in the `xclbin`, the following message will be displayed below:

```
INFO: Found total 1 card(s), 1 are usable
INFO: Failed to find any debug IPs on the platform. Ensure that a valid bitstream with debug IPs (SPM, LAPC) is successfully downloaded.
INFO: xbutil status succeeded.
```

For more information on adding performance monitor counters (AM, AIM, ASM) and LAPC in your design, see Techniques for Debugging Application Hangs.
The `top` command outputs card statistics including memory topology, DMA transfer metrics, and Compute Unit usage data. This command is similar to the Linux `top` command. When running, it continues to operate until `q` is entered in the terminal window to quit.

It has the following command line format:

```
xbutil top [-d card] [-i seconds]
```

The following table lists the available options.

**Table 69: xbutil top Command Options**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
<th>Required</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-d &lt;card&gt;</code></td>
<td>Specifies the target card. <code>&lt;card&gt;</code> can be specified as either the card_id or Bus:Device:Function (BDF). Defaults to card_id = 0 if not specified. Note: Use the xbutil scan command to display both the card_id and BDF for installed cards.</td>
<td>N</td>
</tr>
<tr>
<td><code>-i &lt;seconds&gt;</code></td>
<td>Refresh rate (in seconds). Default is 1 second.</td>
<td>N</td>
</tr>
</tbody>
</table>

For example, the following command will perform `top` with a two second refresh rate.

```
xbutil top -i 2
```

**Table 70: Top Field Definition**

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Device Memory Usage</td>
<td>Percentage of memory used per memory bank. Graphical usage bar will be displayed proportion to the amount of memory used. Bar will be similar to the following: `</td>
</tr>
<tr>
<td>Power</td>
<td>Total card power</td>
</tr>
<tr>
<td>Mem Topology</td>
<td>Assigned tag to the memory bank</td>
</tr>
<tr>
<td></td>
<td>Memory type (that is, DDR or HBM)</td>
</tr>
<tr>
<td></td>
<td>Temperature of the memory bank</td>
</tr>
<tr>
<td></td>
<td>Total available memory per memory bank</td>
</tr>
<tr>
<td></td>
<td>Current memory usage</td>
</tr>
<tr>
<td></td>
<td>Number of buffers allocated</td>
</tr>
<tr>
<td>Total DMA Transfer Metrics</td>
<td>Accumulated bytes transferred because of reboot</td>
</tr>
</tbody>
</table>
### Table 70: Top Field Definition (cont'd)

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CU Usage</td>
<td>Number of commands executed by this CU. Numbers are accumulated until the xclbin is changed where they are reset to zero.</td>
</tr>
</tbody>
</table>

The following example is an output after running this command:

```plaintext
Device Memory Usage
[0] bank0      [ ||||||||| ]                                    25.0% 
[1] bank1      [ ]                                               0.00% 
[2] bank2      [ ]                                               0.00% 
[3] bank3      [ ]                                               0.00% 

Power
34.0W

Mem Topology                                    Device Memory Usage
Tag             Type        Temp        Size    Mem Usage       BO nums
[0] bank0      MEM_DDR4    36 C        16 GB   4 GB            16
[1] bank1      MEM_DDR4    38 C        16 GB   0 Byte          0
[2] bank2      MEM_DDR4    46 C        16 GB   0 Byte          0
[3] bank3      MEM_DDR4    41 C        16 GB   0 Byte          0

Total DMA Transfer Metrics:
Chan[0].h2c: 75 GB
Chan[0].c2h: 78 GB
Chan[1].h2c: 61600 MB
Chan[1].c2h: 61440 MB

 Compute Unit Usage:
CU[@0x1800000] : 68
CU[@0x1c00000] : 68
CU[@0x2500000] : 6
```

If no xclbin is loaded, the following will be displayed:

```plaintext
Device Memory Usage
[1] bank1      [ ]                                               0.00% 

Power
23W

Mem Topology                                    Device Memory Usage
Tag             Type        Temp     Size      Mem Usage       BO nums
[1] bank1       MEM_DDR4    36       64 GB     0 Byte          0

Total DMA Transfer Metrics:
Chan[0].h2c: 0 Byte
Chan[0].c2h: 0 Byte
Chan[1].h2c: 0 Byte
Chan[1].c2h: 0 Byte
```
validate

**IMPORTANT!** This option cannot be used with embedded processor platforms.

The `validate` command generates a high-level, easy to read summary of the installed card. It validates correct installation by performing the following set of tests:

1. Validates the card found.
2. Checks PCI Express link status.
3. Runs a verify kernel on the card.
4. Performs the following data bandwidth tests:
   a. DMA test: Data transfers between host and card memory through PCI Express.
   b. DDR or HBM test: Data transfers between kernels and card memory.

It has the following command line format:

```
xbutil validate [-d card]
```

The following table lists the available option.

**Table 71: xbutil validate Command Option**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
<th>Required</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-d &lt;card&gt;</code></td>
<td>Specifies the target card. <code>&lt;card&gt;</code> can be specified as either the card_id or Bus:Device:Function (BDF). Defaults to card_id = 0 if not specified. Note: Use the xbutil scan command to display both the card_id and BDF for installed cards.</td>
<td>N</td>
</tr>
</tbody>
</table>

The following is an example of the output after running this command:

```
INFO: Found 1 cards
INFO: Validating card[0]: xilinx_u200_xdma_201830_2
INFO: == Starting AUX power connector check:
INFO: == AUX power connector check PASSED
INFO: == Starting PCIE link check:
INFO: == PCIE link check PASSED
INFO: == Starting verify kernel test:
```
INFO: == verify kernel test PASSED
INFO: == Starting DMA test:
Buffer Size: 256 MB
Host -> PCIe -> FPGA write bandwidth = 8775.99 MB/s
Host <- PCIe <- FPGA read bandwidth = 12136.8 MB/s
INFO: == DMA test PASSED
INFO: == Starting device memory bandwidth test:
............
Maximum throughput: 52428 MB/s
INFO: == device memory bandwidth test PASSED
INFO: == Starting PCIE peer-to-peer test:
P2P BAR is not enabled. Skipping validation
INFO: == PCIE peer-to-peer test SKIPPED
INFO: == Starting memory-to-memory DMA test:
bank0 -> bank1 M2M bandwidth: 12010.3 MB/s
bank0 -> bank2 M2M bandwidth: 12051.6 MB/s
bank0 -> bank3 M2M bandwidth: 12063.5 MB/s
bank1 -> bank2 M2M bandwidth: 12052.7 MB/s
bank1 -> bank3 M2M bandwidth: 12048.2 MB/s
bank2 -> bank3 M2M bandwidth: 12052.2 MB/s
INFO: == memory-to-memory DMA test PASSED
INFO: Card[0] validated successfully.
INFO: All cards validated successfully.

---

**version**

🌟 IMPORTANT! *This option cannot be used with embedded processor platforms.*

The `version` command returns the XRT build version details. It is identical to the `version` command. It has the following command line format. There are no options.

```
xbutil version
```

The following table lists the fields returned from `xbutil version` command.

**Table 72: Version Field Definition**

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XRT Build Version</td>
<td>XRT build version</td>
</tr>
<tr>
<td>Build Version Branch</td>
<td>Build version branch</td>
</tr>
<tr>
<td>Build Version Hash</td>
<td>Build version hash</td>
</tr>
<tr>
<td>Build Version Hash Date</td>
<td>Build version branch date</td>
</tr>
<tr>
<td>Build Version Date</td>
<td>Build version date</td>
</tr>
<tr>
<td>XOCL</td>
<td>XOCL version</td>
</tr>
<tr>
<td>XCLMGMT</td>
<td>XCLMGMT version</td>
</tr>
</tbody>
</table>
Below is an example output of `xbutil version` for a system with three cards installed.

<table>
<thead>
<tr>
<th>XRT Build Version</th>
<th>2.3.1301</th>
</tr>
</thead>
<tbody>
<tr>
<td>Build Version Branch</td>
<td>2019.2</td>
</tr>
<tr>
<td>Build Version Hash</td>
<td>192e706aea53163a04c574f9b3fe9ed76b6ca471</td>
</tr>
<tr>
<td>Build Version Hash Date</td>
<td>Thu, 24 Oct 2019 19:27:30 -0700</td>
</tr>
<tr>
<td>Build Version Date</td>
<td>Thu, 24 Oct 2019 20:04:29 -0700</td>
</tr>
<tr>
<td>XOCL</td>
<td>2.3.1301,192e706aea53163a04c574f9b3fe9ed76b6ca471</td>
</tr>
<tr>
<td>XCLMGMT</td>
<td>2.3.1301,192e706aea53163a04c574f9b3fe9ed76b6ca471</td>
</tr>
</tbody>
</table>

To return additional card details, use `xbmgmt flash --scan`. 
Chapter 31

xbmgmt Utility

**TIP:** The next generation of the xbmgmt command-line tool is in preview mode for the 2020.2 release of XRT. This version will replace the current xbmgmt in a future release of XRT. Refer to [https://xilinx.github.io/XRT/2020.2/html/xbmgmt2.html](https://xilinx.github.io/XRT/2020.2/html/xbmgmt2.html) for information about the new utility.

Xilinx® Board Management (xbmgmt) utility is a standalone command line tool that is included with the Xilinx Runtime (XRT) installation package. The xbmgmt command supports both Alveo Data Center accelerator cards, and embedded processor-based platforms.

Accelerator cards are partitioned into a user function and a management function to provide different levels of card access. The user function allows end users to load and run their applications, while the management function is for system administrators to manage the card. The xbutil utility interacts with the user function. The xbmgmt utility, which requires root privilege, is for interacting with the management function. The reason for splitting the function access between the two utilities is to provide some security for the management features of the tool.

**IMPORTANT!** The xbmgmt utility only works with Alveo cards that have Xilinx provided shells/platforms. XRT does not work on custom Vivado® designs.

This utility is used for card installation and administration, and requires sudo privileges when running it. The xbmgmt supported tasks include flashing the card firmware, and scanning the current device configuration.

The xbmgmt command line format is:

```bash
xbmgmt <command> [options]
```

The supported sub-commands are given below.

- **config:** Parse or update daemon/device configuration
- **flash:** Update SC firmware or shell on the device
- **help:** Print out help message for a sub-command
- **partition:** Show and download partition onto the device
- **scan:** List all detected mgmt PCIe® functions
- **version:** Print out XRT build version
**TIP:** You can use the `help` command to list the available `xbmgmt` commands and options, and access help for individual commands by using the following:

```
xbmgmt help <command>
```

For detailed help of each sub-command, use the following:

```
xbmgmt help <subcommand>
```

Set up the `xbmgmt` command using the following scripts:

- For csh shell:
  ```
  $ source /opt/xilinx/xrt/setup.csh
  ```

- For bash shell:
  ```
  $ source /opt/xilinx/xrt/setup.sh
  ```

---

### config

The `xbmgmt config` command lets you configure memory retention capability of the Alveo accelerator card.

**Table 73: xbmgt config Sub-commands**

<table>
<thead>
<tr>
<th>Sub-command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>--enable_retention</code></td>
<td>This feature preserves the contents of the DDR data during XCLBIN swapping</td>
</tr>
<tr>
<td></td>
<td>in DFX-2RP target platforms.</td>
</tr>
<tr>
<td><code>--disable_retention</code></td>
<td>This feature disables the retention of the contents of the DDR data during</td>
</tr>
<tr>
<td></td>
<td>XCLBIN swapping in DFX-2RP target platforms.</td>
</tr>
</tbody>
</table>

The `config` command has the following command line options.

```
sudo xbmgt config --enable_retention --ddr [--card <bdf>]
```

**Table 74: xbmgt config Sub-Command Options**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
<th>Required</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>--ddr</code></td>
<td>Enable or disable retention of the contents of</td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td>the DDR memory.</td>
<td></td>
</tr>
</tbody>
</table>
### Table 74: xbmgmt config Sub-Command Options (cont’d)

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
<th>Required</th>
</tr>
</thead>
<tbody>
<tr>
<td>--card &lt;bdf&gt;</td>
<td>Specifies the accelerator card to be configured as identified by its Bus:Device:Function (BDF) tag.</td>
<td>N</td>
</tr>
<tr>
<td></td>
<td><strong>TIP:</strong> Use the xbmgmt flash --scan to obtain the card BDF.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>If the BDF is not found, you will receive the following message where &lt;bdf&gt; is the BDF value entered:</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>ERROR:</strong> No mgmt PF found for &lt;bdf&gt;</td>
<td></td>
</tr>
</tbody>
</table>

### flash

**IMPORTANT! This option cannot be used with embedded processor platforms.**

The flash command has three sub-commands which are described in the following table.

The xbmgmt flash --scan command will report if a platform base is loaded. You will need to use xbmgmt partition --scan to see if the shell is loaded. Refer to Alveo Platform Loading Overview for more information on the platform base and shell.

### Table 75: xbmgmt flash Sub-commands

<table>
<thead>
<tr>
<th>Sub-command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>--scan</td>
<td>Query the card’s flashable partition running on FPGA and installed on the host system</td>
</tr>
<tr>
<td>--update</td>
<td>Flash a target platform (flashable partition) to the card</td>
</tr>
<tr>
<td>--factory_reset</td>
<td>Reset the card to factory condition</td>
</tr>
</tbody>
</table>

Each of the sub-commands are detailed below.

#### --scan

The --scan sub-command returns details of the flashable partition installed on each card along with the flashable partitions installed on the host system. In addition, it also returns additional information including SC version, BDF, Serial number, and MAC addresses.

It has the following command line format.

```
xbmgmt flash --scan [--verbose | --json]
```
Table 76: xbmgmt flash --scan Sub-command Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
<th>Required</th>
</tr>
</thead>
<tbody>
<tr>
<td>--verbose</td>
<td>Returns verbose output which includes additional fields. The --verbose and --json options are mutually exclusive.</td>
<td>N</td>
</tr>
<tr>
<td>--json</td>
<td>Returns all fields given with --verbose option in JSON format. The --verbose and --json options are mutually exclusive.</td>
<td>N</td>
</tr>
</tbody>
</table>

Using the flash --scan sub-command without any options, as shown below, will return the fields listed in the following table.

xbmgmt flash --scan

Table 77: xbmgmt flash --scan Field Definition

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Card</td>
<td>Provides the enumerated card Bus Device Function (BDF) for the card in the following format: [Bus : Device : Function]</td>
</tr>
<tr>
<td>TIP: The xbmgmt command returns a BDF which includes the management function on the card. The xbuutil scan command returns a BDF which includes the user function on the card.</td>
<td></td>
</tr>
<tr>
<td>Card type</td>
<td>Xilinx card type</td>
</tr>
<tr>
<td>Flash type</td>
<td>Returns the flash type physically installed on the card. The flash type can be:</td>
</tr>
<tr>
<td></td>
<td>• Dual QSPI: Two x4 SPI</td>
</tr>
<tr>
<td></td>
<td>• SPI: One x4 SPI</td>
</tr>
<tr>
<td></td>
<td>• OSPI: One x8 SPI</td>
</tr>
<tr>
<td>Flashable partition running on FPGA</td>
<td>Returns details on the flashable partition installed on the card:</td>
</tr>
<tr>
<td></td>
<td>• Name of the target platform running on the FPGA</td>
</tr>
<tr>
<td></td>
<td>• ID unique identification of the platform bitstream</td>
</tr>
<tr>
<td></td>
<td>• Satellite Controller (SC) version number</td>
</tr>
<tr>
<td>IMPORTANT! Flashable partition running on FPGA’s ID must match the flashable partitions installed in system or the stack will not operate correctly.</td>
<td></td>
</tr>
<tr>
<td>Flashable partitions installed in system</td>
<td>Returns details on the flashable partition installed on the host system:</td>
</tr>
<tr>
<td></td>
<td>• Name of the target platform running installed on the host system</td>
</tr>
<tr>
<td></td>
<td>• ID which represents the timestamp of the target platform</td>
</tr>
<tr>
<td></td>
<td>• SC version number</td>
</tr>
</tbody>
</table>
Below is an example output of `xbmgmt flash --scan` for a system with two different cards installed:

Card [0000:a6:00.0]
Card type: u280
Flash type: SPI
Flashable partition running on FPGA:
  xilinx_u280_xdma_201920_1,[ID=0x5da8da6e],[SC=4.3.4]
Flashable partitions installed in system:
  xilinx_u280_xdma_201920_1,[ID=0x5da8da6e],[SC=4.3.4]

Card [0000:73:00.0]
Card type: u250
Flash type: SPI
Flashable partition running on FPGA:
  xilinx_u250_xdma_201830_2,[ID=0x5d14fbe6],[SC=4.3.7]
Flashable partitions installed in system:
  xilinx_u250_xdma_201830_2,[ID=0x5d14fbe6],[SC=4.3.7]

If a card has been installed for the first time and not previously been flashed, or if the card has been factory reset the flashable partition running on FPGA will indicate this with the word GOLDEN within its name as shown in `--factory_reset`.

Using the `--verbose` option, as shown in the following example, returns additional fields specified in the following table.

**Note:** A platform has to be installed on the system to obtain the full card information.

```bash
xbmgmt flash --scan --verbose
```

### Table 78: `xbmgmt flash --scan --verbose` Field Definition

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Card Name</td>
<td>Xilinx provided card name</td>
</tr>
<tr>
<td>Card serial number (S/N)</td>
<td>Unique card serial number</td>
</tr>
</tbody>
</table>
| Configuration mode     | Returns the configuration mode in which FPGA boots up from a cold reset. The configuration modes can be:  
  - Dual QSPI: Two x4 SPI  
  - QSPI: One x4 SPI  
  - OSPI: One x8 SPI |
| Fan presence           | Represents the presence of a fan on the card.  
  A – Active cooling. Fan is present on card.  
  P – Passive cooling. Fan is not present on the card and must be cooled by host server. |
| Max power level        | Maximum available card power (Watts) supplied from PCIe and connected AUX power port. Not all cards have an AUX power port. |
| MAC address            | Returns a list of Xilinx assigned MAC addresses for the card.  
  You are free to use Xilinx assigned MAC address or provide your own.  
  An address of FF:FF:FF:FF:FF:FF implies this MAC slot has not been assigned an address. |
An example of the `xbmgmt flash --scan --verbose` output for a system with one card is given below:

Card [0000:a6:00.0]
Card type: u280
Flash type: SPI
Flashable partition running on FPGA:
  xilinx_u280_xdma_201920_1,[ID=0x5da8da6e],[SC=4.3.4]
Flashable partitions installed in system:
  xilinx_u280_xdma_201920_1,[ID=0x5da8da6e],[SC=4.3.4]
Card name ALVEO U280 PQ
Card S/N: 21760394R01L
Config mode: 7
Fan presence: A
Max power level: 225W
MAC address0: 00:0A:35:06:00:0A
MAC address1: 00:0A:35:06:00:0B
MAC address2: FF:FF:FF:FF:FF:FF
MAC address3: FF:FF:FF:FF:FF:FF

Using the `--json` sub-option returns similar information as `xbmgmt flash --scan`, but in JSON format. The following shows an example of the generated JSON output for a system with one card.

```json
{
  "card0": {
    "shellpackage": "xilinx_u280_xdma_201920_1,[ID=0x5da8da6e],[SC=4.3.4];",
    "name": "ALVEO U280 PQ",
    "serial": "21760394R01L",
    "config_mode": "7",
    "fan_presence": "A",
    "max_power": "225W",
    "mac0": "00:0A:35:06:00:0A",
    "mac1": "00:0A:35:06:00:0B",
    "mac2": "FF:FF:FF:FF:FF:FF",
    "mac3": "FF:FF:FF:FF:FF:FF"
  }
}
```

**--update**

Use the `flash --update` sub-command to change the flashable partition (target platform) on the card. It does this by flashing the specified target platform and associated satellite controller to the configurable ROM on the card. It has the following command line format:

```
xbmgmt flash --update [--shell <target_platform_name> [--id <target_platform_id>]] [--card <bdf>] [-force]
```

The following table lists the available options.
Table 79: xbmgmt --update Sub-Command Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
<th>Required</th>
</tr>
</thead>
<tbody>
<tr>
<td>--shell &lt;target_platform_name&gt; [--id]</td>
<td>Name of the target platform (flashable partition) to be flashed on the card. The target platform must be installed on the host system prior to specifying it via the target platform installation package. See the Alveo™ card installation guide for details on downloading and installing the deployment target platform. Use the <code>xbmgmt flash --scan</code> to list available target platforms installed on the host system. If neither the <code>--shell</code> and <code>--card</code> options are specified, all cards will be flashed with the compatible target platform if available on the host system. For instance, if your system has U200 and U250 cards installed and both U200 and U250 target platforms exist on the host system, then both cards will be flashed with their respective target platforms. If a card's target platform does not exist on the system, then the card will not be flashed. If the <code>--shell</code> option is specified, but the <code>--card</code> option is not, then all cards compatible with the specified <code>target_platform_name</code> will be flashed. For instance, if your system has two U200 cards installed, then both cards will be flashed with the specified <code>target_platform_name</code>. If a card's target platform does not exist on the system, then the card will not be flashed. If the <code>--shell</code> option is not specified, but the <code>--card</code> option is specified, the specified card will be flashed with the compatible target platform if available on the host system. If both the <code>--shell</code> and <code>--card</code> options are specified, the specified card will be flashed with the specified target platform if available on the host system. If the flashable partition on the card matches the flashable partition on the system, a similar message as shown below will be displayed and the card will not be updated. However, you can force the card to be flashed by using the <code>--force</code> option described below. Card [0000:65:00.0]: Status: shell is up-to-date Card(s) up-to-date and do not need to be flashed. If the specified <code>target_platform_name</code> is not installed on the host system, the card will not be updated and you will receive the following message: Specified shell not found. This <code>--id</code> sub-option specifies the ID of the target platform. Use the <code>xbmgmt flash --scan</code> to obtain the ID of the flashable partition. If the <code>--id</code> option is not specified, the card will be updated with the latest released target platform available on the host system.</td>
<td>N</td>
</tr>
</tbody>
</table>
### Table 79: `xbmgmt --update` Sub-Command Options (cont’d)

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
<th>Required</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>--card &lt;bdf&gt;</code></td>
<td>Specifies the accelerator card to be updated as identified by its Bus:Device:Function (BDF) tag. Use the <code>xbmgmt flash --scan</code> to obtain the card BDF.</td>
<td>N</td>
</tr>
<tr>
<td></td>
<td>If <code>--card</code> is not specified, then all accelerator cards in the system compatible with the specified <code>target_platform</code> will be updated.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>If neither the <code>--shell</code> and <code>--card</code> options are specified, all cards will be flashed with the compatible target platform if available on the host system.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>For instance, if your system has U200 and U250 cards installed and both U200 and U250 target platforms exist on the host system, then both cards will be flashed with their respective target platforms. If a card's target platform does not exist on the system, then the card will not be flashed.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>If the <code>--shell</code> option is specified, but the <code>--card</code> option is not, then all cards compatible with the specified <code>target_platform_name</code> will be flashed.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>For instance, if your system has two U200 cards installed, then both cards will be flashed with the specified <code>target_platform_name</code>. If a card's target platform does not exist on the system, then the card will not be flashed.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>If the <code>--shell</code> option is not specified, but the <code>--card</code> option is specified, the specified card will be flashed with the compatible target platform if available on the host system.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>If both the <code>--shell</code> and <code>--card</code> options are specified, the specified card will be flashed with the specified target platform if available on the host system.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>If the BDF is not found, you will receive the following message where <code>&lt;bdf&gt;</code> is the BDF value entered. Use the <code>xbmgmt flash --scan</code> to list the BDF values of installed cards.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ERROR: No mgmt PF found for <code>&lt;bdf&gt;</code></td>
<td></td>
</tr>
<tr>
<td><code>--force</code></td>
<td>The force option means &quot;yes&quot; to any prompt from <code>xbmgmt flash --update</code> command.</td>
<td>N</td>
</tr>
<tr>
<td></td>
<td>For instance, when flashing the card through <code>xbmgmt flash --update</code>, it will confirm that you wish to perform the update with the following prompt:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Are you sure you wish to proceed? [y/n]:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>The <code>--force</code> option will automatically set the answer to &quot;y&quot; or yes, and proceed.</td>
<td></td>
</tr>
<tr>
<td><code>--factory_reset</code></td>
<td>Use the <code>flash --factory_reset</code> sub-command to restore the flashable partition running on the FPGA to the original golden image. This command will not change the satellite controller version. It has the following command line format:</td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>xbmgmt flash --factory_reset [--card &lt;bdf&gt;]</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td>There is only one option and is given in the following table.</td>
<td></td>
</tr>
</tbody>
</table>
Table 80: `xbmgmt --factory_reset` Sub-Command Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
<th>Required</th>
</tr>
</thead>
<tbody>
<tr>
<td>--card &lt;bdf&gt;</td>
<td>Specifies the accelerator card to be factory reset as identified by its Bus:Device:Function (BDF) tag. Use the <code>xbmgmt flash --scan</code> to obtain the card BDF. If <code>--card</code> is not specified, then the card ID 0 will be reset. Card ID is not deterministic and can change after a cold or warm reboot.</td>
<td>N</td>
</tr>
</tbody>
</table>

After running the `xbmgmt flash --factory_reset` command, it is necessary to cold-reboot the system to restore the card to the original golden image.

After a factory reset and cold rebooting, use `xbmgmt flash --scan` to confirm the flashable partition running on FPGA has been reverted. The partition will include the word GOLDEN within its name as shown below:

Card [0000:a6:00.0]
Card type: u280
Flash type: SPI
Flashable partition running on FPGA:
  xilinx_u280_GOLDEN_8,[SC=4.3]
Flashable partitions installed in system:
  xilinx_u280_xdma_201920_1,[ID=0x5da8da6e],[SC=4.3.4]

partition

The `xbmgmt partition` command lets you display and program shell partitions to the Alveo card.

The `partition` command is intended to be used with DFX-2RP platforms. Prior to running an application, including the validation application, it is necessary to first program the shell partition to the card using the following command:

```bash
sudo /opt/xilinx/xrt/bin/xbmgmt partition --program --name <shell_name> --card <card_bdf>
```

After the shell partition is programmed, you do not need to reprogram the accelerator card unless the system is warm or cold rebooted, or to load a different shell.

**TIP:** The platform base remains loaded even when the power is cycled on the accelerator card.

The `xbmgmt partition --scan` command reports if a platform shell has been loaded.
Table 81: xbmgmt partition Sub-commands

<table>
<thead>
<tr>
<th>Sub-command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>--scan</td>
<td>Scan and displays base and shell partitions running on the FPGA and installed in the system.</td>
</tr>
<tr>
<td>--program</td>
<td>This command is used to program the shell partition by providing the name of the shell partition. The command also supports the option to program the shell from a given interface-uuid.</td>
</tr>
</tbody>
</table>

Scan

The `--scan` sub-command returns details of the flashable partition installed on each card along with the flashable partitions installed on the host system. In addition, it also returns additional information including SC version, BDF, Serial number, and MAC addresses.

It has the following command line format.

```
xbmgmt partition --scan
```

The following is an example of the command output:

```
Card [0000:d8:00.0]
  Partitions running on FPGA:
  xilinx_u200_gen3x16_base_1
    logic-uuid: 8892e9a0478feaa2699f5df1f696470d
    interface-uuid: 1641962866f4b5e579ce90a6bdabc6f
  Partitions installed in system:
  xilinx_u200_gen3x16_xdma_shell_1_1
    logic-uuid: a21db55d2fbd60cc95eaa0a8144e1
    interface-uuid: 9437e0f859a4d9bd9e226d7e5f6be8
```

If the partition is programmed, the output will look like the following example output below.

```
alveo@alveo:~$ sudo /opt/xilinx/xrt/bin/xbmgmt partition --scan
Card [0000:65:00.0]
  Partitions running on FPGA:
  xilinx_u200_gen3x16_base_1
    logic-uuid: 3d40702f3777396cc82e0df89bafde2
    interface-uuid: 19f21ba41b9c38dffe0f1ee68910b8bb
  xilinx_u200_gen3x16_xdma_shell_1_1
    logic-uuid: fd19b2fe5a10b8cb89e35b0be02f274
    interface-uuid: 995b41d8c729d658d6700a027f412f78
  Partitions installed in system:
  xilinx_u200_gen3x16_xdma_shell_1_1
    logic-uuid: fd19b2fe5a10b8cb89e35b0be02f274
    interface-uuid: 995b41d8c729d658d6700a027f412f78
```
Program

The `--program` sub-command lets you program the specified shell partition.

It has the following command line format.

```
xmgmt partition --program --name name [--id interface-uuid] [--card bdf]
```

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
<th>Required</th>
</tr>
</thead>
<tbody>
<tr>
<td>--name &lt;name&gt;</td>
<td>Specify the name of the shell partition to program. Use the <code>xmgmt partition --scan</code> to display the names of the programmable partitions.</td>
<td>Y</td>
</tr>
<tr>
<td>--id</td>
<td>The <code>--id</code> sub-option specifies the partition interface-uuid. Use the <code>xmgmt partition --scan</code> to obtain the ID of the flashable partition.</td>
<td>N</td>
</tr>
<tr>
<td>--card &lt;bdf&gt;</td>
<td>Specifies the accelerator card to be programmed as identified by its Bus:Device:Function (BDF) tag. Use the <code>xmgmt partition --scan</code> to obtain the card BDF. If the BDF is not found, you will receive the following message where &lt;bdf&gt; is the BDF value entered:</td>
<td>N</td>
</tr>
<tr>
<td></td>
<td><strong>ERROR: No mgmt PF found for &lt;bdf&gt;</strong></td>
<td></td>
</tr>
</tbody>
</table>

**Table 82: xmgmt --program Sub-Command Options**

**scan (xmgmt)**

**IMPORTANT! This option cannot be used with embedded processor platforms.**

The `xmgmt scan` command returns a list of all the detected management PCIe functions. Each item in the list includes the card BDF, target platform name, target platform ID, and management driver instance number.

**TIP: If additional details are needed, use the `xmgmt flash --scan --verbose` command.**

It has the following command line format. There are no options.

```
xmgmt scan
```

The following table lists the fields returned from `xmgmt scan` command.
Table 83: `xbmgmt scan` Field Definition

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BDF</td>
<td>Provides the enumerated Bus:Device:Function (BDF) identifier for the card in the following format: [Bus:Device:Function]</td>
</tr>
</tbody>
</table>
| Flashable partition running on FPGA | Details on the flashable partition include:  
  - Name of the target platform flashed on the FPGA  
  - Unique ID associated with the target platform. |
| mgmt  | Returns the assigned management driver instance.  
  The instance number can easily find the device node for each function.  
  On a supported Linux distribution, the device node can be found at: `/dev/xclmgmt<inst>`.  
  In addition, the instance can be useful when mapping the `dmesg` information to a specific card. |

Below is an example output of `xbmgmt scan` for a system with two cards installed. Details for each card are on a separate line:

```
0000:d8:00.0 xilinx_u200_gen3x16_xdma_shell_1_1 mgmt(inst=55296)
0000:af:00.0 xilinx_u250_gen3x16_base_3 mgmt(inst=44800)
```

**version**

**IMPORTANT! This option cannot be used with embedded processor platforms.**

The `version` command returns the XRT build version details. It is identical to the `xbutil version` command. It has the following command line format. There are no options.

```
xbmgmt version
```

The following table lists the fields returned from `xbmgmt version` command.

Table 84: Version Field Definition

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XRT Build Version</td>
<td>XRT build version</td>
</tr>
<tr>
<td>Build Version Branch</td>
<td>Build version branch</td>
</tr>
<tr>
<td>Build Version Hash</td>
<td>Build version hash</td>
</tr>
<tr>
<td>Build Version Hash Date</td>
<td>Build version branch date</td>
</tr>
<tr>
<td>Build Version Date</td>
<td>Build version date</td>
</tr>
<tr>
<td>XOCL</td>
<td>XOCL version</td>
</tr>
<tr>
<td>XCLMGMT</td>
<td>XCLMGMT version</td>
</tr>
</tbody>
</table>
The following is an example output of `xbmgmt version`.

```
XRT Build Version: 2.3.1301
Build Version Branch: 2019.2
Build Version Hash: 192e706aea53163a04c574f9b3fe9ed76b6ca471
Build Version Hash Date: Thu, 24 Oct 2019 19:27:30 -0700
Build Version Date: Thu, 24 Oct 2019 20:04:29 -0700
XOCL: 2.3.1301,192e706aea53163a04c574f9b3fe9ed76b6ca471
XCLMGMT: 2.3.1301,192e706aea53163a04c574f9b3fe9ed76b6ca471
```
xclbinutil Utility

The xclbinutil utility can create, modify, and report xclbin content information.

The available command options are shown in the following table.

Table 85: xclbinutil Commands

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-h [ --help ]</td>
<td>Print help messages.</td>
</tr>
<tr>
<td>-i [ --input ]&lt;arg&gt;</td>
<td>Input file name. Reads xclbin into memory.</td>
</tr>
<tr>
<td>-o [ --output ]&lt;arg&gt;</td>
<td>Output file name. Writes in memory xclbin image to a file.</td>
</tr>
<tr>
<td>-v [ --verbose ]</td>
<td>Display verbose/debug information</td>
</tr>
<tr>
<td>-q [ --quiet ]</td>
<td>Minimize reporting information</td>
</tr>
<tr>
<td>--migrate-forward</td>
<td>Migrate the xclbin archive forward to the new binary format.</td>
</tr>
<tr>
<td>--remove-section&lt;arg&gt;</td>
<td>Section name to remove.</td>
</tr>
<tr>
<td>--add-section&lt;arg&gt;</td>
<td>Section name to add. Format: &lt;section&gt;:&lt;format&gt;:&lt;file&gt;</td>
</tr>
<tr>
<td>--dump-section&lt;arg&gt;</td>
<td>Section to dump. Format: &lt;section&gt;:&lt;format&gt;:&lt;file&gt;</td>
</tr>
<tr>
<td>--replace-section&lt;arg&gt;</td>
<td>Section to replace.</td>
</tr>
<tr>
<td>--key-value&lt;arg&gt;</td>
<td>Key value pairs. Format: [USER</td>
</tr>
<tr>
<td>--remove-key&lt;arg&gt;</td>
<td>Removes the given user key from the xclbin archive.</td>
</tr>
<tr>
<td>--add-signature&lt;arg&gt;</td>
<td>Adds a user defined signature to the given xclbin image.</td>
</tr>
<tr>
<td>--remove-signature</td>
<td>Removes the signature from the xclbin image.</td>
</tr>
<tr>
<td>--get-signature</td>
<td>Returns the user defined signature (if set) of the xclbin image.</td>
</tr>
<tr>
<td>--info</td>
<td>Report accelerator binary content. Including: generation and packaging data,</td>
</tr>
<tr>
<td></td>
<td>kernel signatures, connectivity, clocks, sections, etc</td>
</tr>
<tr>
<td>--list-names</td>
<td>List all possible section names (standalone option).</td>
</tr>
<tr>
<td>--version</td>
<td>Version of this executable.</td>
</tr>
<tr>
<td>--force</td>
<td>Forces a file overwrite.</td>
</tr>
</tbody>
</table>

The following are various use examples of the tool.

- Reporting xclbin information: xclbinutil --info --input binary_container_1.xclbin

- Extracting the bitstream image: xclbinutil --dump-section BITSTREAM:RAW:bitstream.bit --input binary_container_1.xclbin
• **Extracting the build metadata:**
  
  ```
  xclbinutil --dump-section
  BUILD_METADATA:HTML:buildMetadata.json --input
  binary_container_1.xclbin
  ```

  • **Removing a section:**
  
  ```
  xclbinutil --remove-section BITSTREAM --input
  binary_container_1.xclbin --output binary_container_modified.xclbin
  ```

  For most users, details about the contents and how the `xclbin` was created is desired. This information can be obtained through the `--info` option and reports information on the `xclbin`, hardware platform, clocks, memory configuration, kernel, and how the `xclbin` was generated.

  The output of the `xclbinutil` command using the `--info` option is shown below divided into sections.

  ```
  xclbinutil -i binary_container_1.xclbin --info
  ```

---

**xclbin Information**

<table>
<thead>
<tr>
<th>Generated by:</th>
<th>v++ (2020.1) on Mon Apr 13 20:19:40 MDT 2020</th>
</tr>
</thead>
<tbody>
<tr>
<td>Version:</td>
<td>2.6.436</td>
</tr>
<tr>
<td>Kernels:</td>
<td>CopyKernel</td>
</tr>
<tr>
<td>Signature:</td>
<td>Bitstream</td>
</tr>
<tr>
<td>Content:</td>
<td>Bitstream</td>
</tr>
<tr>
<td>UUID (xclbin):</td>
<td>d081de98-3fd3-4e9b-bab3-108b42c73101</td>
</tr>
<tr>
<td>UUID (IINTF):</td>
<td>862c7020a250293e32036f19956669e5</td>
</tr>
<tr>
<td>Sections:</td>
<td>DEBUG_IP_LAYOUT, BITSTREAM, MEM_TOPOLOGY,</td>
</tr>
<tr>
<td></td>
<td>IP_LAYOUT,</td>
</tr>
<tr>
<td></td>
<td>CONNECTIVITY, CLOCK_FREQ_TOPOLOGY,</td>
</tr>
<tr>
<td>BUILD_METADATA,</td>
<td>EMBEDDED_METADATA, SYSTEM_METADATA,</td>
</tr>
<tr>
<td>PARTITION_METADATA</td>
<td></td>
</tr>
</tbody>
</table>

---

**Hardware Platform Information**

<table>
<thead>
<tr>
<th>Vendor:</th>
<th>xilinx</th>
</tr>
</thead>
<tbody>
<tr>
<td>Board:</td>
<td>u200</td>
</tr>
<tr>
<td>Name:</td>
<td>xdma</td>
</tr>
<tr>
<td>Version:</td>
<td>201830.1</td>
</tr>
<tr>
<td>Generated Version:</td>
<td>Vivado 2018.3 (SW Build: 2388429)</td>
</tr>
<tr>
<td>Created:</td>
<td>Wed Nov 14 20:06:10 2018</td>
</tr>
<tr>
<td>FPGA Device:</td>
<td>xcu200</td>
</tr>
<tr>
<td>Board Vendor:</td>
<td>xilinx.com</td>
</tr>
</tbody>
</table>
Clocks

Reports the maximum kernel clock frequencies available. Both the clock names and clock indexes are provided. The clock indexes are identical to those reported in platforminfo Utility.

<table>
<thead>
<tr>
<th>Name</th>
<th>Index</th>
<th>Type</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATA_CLK</td>
<td>0</td>
<td>DATA</td>
<td>300 MHz</td>
</tr>
<tr>
<td>KERNEL_CLK</td>
<td>1</td>
<td>KERNEL</td>
<td>500 MHz</td>
</tr>
</tbody>
</table>

Memory Configuration

<table>
<thead>
<tr>
<th>Name</th>
<th>Index</th>
<th>Type</th>
<th>Base Address</th>
<th>Address Size</th>
<th>Bank Used</th>
</tr>
</thead>
<tbody>
<tr>
<td>bank0</td>
<td>0</td>
<td>MEM_DDR4</td>
<td>0x0</td>
<td>0x4000000000</td>
<td>No</td>
</tr>
<tr>
<td>bank1</td>
<td>1</td>
<td>MEM_DDR4</td>
<td>0x4000000000</td>
<td>0x4000000000</td>
<td>Yes</td>
</tr>
<tr>
<td>bank2</td>
<td>2</td>
<td>MEM_DDR4</td>
<td>0x8000000000</td>
<td>0x4000000000</td>
<td>No</td>
</tr>
<tr>
<td>bank3</td>
<td>3</td>
<td>MEM_DDR4</td>
<td>0xc000000000</td>
<td>0x4000000000</td>
<td>No</td>
</tr>
<tr>
<td>PLRAM[0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Kernel Information

For each kernel in the xclbin, the function definition, ports, and instance information is reported.

The following is an example of the reported function definition.

```
Signature: krnl_vadd (int* a, int* b, int* c, int const n_elements)
```

The following is an example of the reported ports.

```
Port: M_AXI_GMEM
Mode: master
Range (bytes): 0xFFFFFFFF
Data Width: 32 bits
Port Type: addressable

Port: M_AXI_GMEM1
Mode: master
Range (bytes): 0xFFFFFFFF
Data Width: 32 bits
Port Type: addressable

Port: S_AXI_CONTROL
Mode: slave
Range (bytes): 0x1000
Data Width: 32 bits
Port Type: addressable
```
The following is an example of the reported instance(s) of the kernel.

<table>
<thead>
<tr>
<th>Instance:</th>
<th>krnl_vadd_1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base Address:</td>
<td>0x0</td>
</tr>
<tr>
<td>Argument:</td>
<td>a</td>
</tr>
<tr>
<td>Register Offset:</td>
<td>0x10</td>
</tr>
<tr>
<td>Port:</td>
<td>M_AXI_GMEM</td>
</tr>
<tr>
<td>Memory:</td>
<td>bank1 (MEM_DDR4)</td>
</tr>
</tbody>
</table>

| Argument: | b |
| Register Offset: | 0x1C |
| Port: | M_AXI_GMEM |
| Memory: | bank1 (MEM_DDR4) |

| Argument: | c |
| Register Offset: | 0x28 |
| Port: | M_AXI_GMEM1 |
| Memory: | bank1 (MEM_DDR4) |

| Argument: | n_elements |
| Register Offset: | 0x34 |
| Port: | S_AXI_CONTROL |
| Memory: | <not applicable> |

### Tool Generation Information

The utility also reports the `{v++}` command line used to generate the `xclbin`. The Command Line section gives the actual `{v++}` command line used, while the Options section displays each option used in the command line, but in a more readable format with one option per line.

**Generated By**

Command: `{v++}`


Command Line: `v++ -t hw_emu --platform /opt/xilinx/platforms/xilinx_u200_xdma_201830_1/xilinx_u200_xdma_201830_1.xpfm --save-temps -l --connectivity.nk krnl_vadd:1 -g --messageDb binary_container_1.mdb --temp_dir binary_container_1 --report_dir binary_container_1/reports --log_dir binary_container_1/logs --remote_ip_cache /wrk/tutorials/ip_cache -o binary_container_1/xclbin binary_container_1/

Options: `-t hw_emu --platform /opt/xilinx/platforms/xilinx_u200_xdma_201830_1/xilinx_u200_xdma_201830_1.xpfm --save-temps -l --connectivity.nk krnl_vadd:1 -g --messageDb binary_container_1.mdb --temp_dir binary_container_1`
| --report_dir binary_container_1/reports |
| --log_dir binary_container_1/logs |
| --remote_ip_cache /wrk/tutorials/ip_cache |
| -obinary_container_1.xclbin binary_container_1/krnl_vadd.o |

```
# User Added Key Value Pairs
<empty>
```
xrt.ini File

The Xilinx runtime (XRT) library uses various control parameters to specify debugging, profiling, and message logging when running the host application and kernel execution. These control parameters are specified in a runtime initialization file, xrt.ini and used to configure features of XRT at start-up.

If you are a command line user, the xrt.ini file needs to be created manually and saved to the same directory as the host executable. The runtime library checks if xrt.ini exists in the same directory as the host executable and automatically reads the file to configure the runtime.

TIP: The Vitis IDE creates the xrt.ini file automatically based on your run configuration and saves it with the host executable.

Runtime Initialization File Format

The xrt.ini file is a simple text file with groups of keys and their values. Any line beginning with a semicolon (;) or a hash (#) is a comment. The group names, keys, and key values are all case sensitive.

The following is an example xrt.ini file that enables the timeline trace feature, and directs the runtime log messages to the Console view.

```ini
#Start of Debug group
[Debug]
timeline_trace = true

#Start of Runtime group
[Runtime]
runtime_log = console
```

There are three groups of initialization keys:

- Runtime
- Debug
- Emulation

The following tables list all supported keys for each group, the supported values for each key, and a short description of the purpose of the key.
### Table 86: Runtime Group

<table>
<thead>
<tr>
<th>Key</th>
<th>Valid Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>api_checks</td>
<td>[true</td>
<td>false]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• true: Enable. This is the default value.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• false: Disable.</td>
</tr>
<tr>
<td>cpu_affinity</td>
<td>{N,N,...}</td>
<td>Pins all runtime threads to specified CPUs.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Example: cpu_affinity = {4,5,6}</td>
</tr>
<tr>
<td>ert_polling</td>
<td>[true</td>
<td>false]</td>
</tr>
<tr>
<td>exclusive_cu_context</td>
<td>[true</td>
<td>false]</td>
</tr>
<tr>
<td>polling_throttle</td>
<td>&lt;value&gt;</td>
<td>Specifies the time interval in microseconds that the runtime library polls the device status when ert_polling is enabled. The default value is 0.</td>
</tr>
<tr>
<td>runtime_log</td>
<td>[null</td>
<td>console</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• null: Do not print any logs. This is the default value.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• console: Print logs to stdout</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• syslog: Print logs to Linux syslog.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• &lt;filename&gt;: Print logs to the specified file. For example, runtime_log=my_run.log.</td>
</tr>
<tr>
<td>verbosity</td>
<td>[0</td>
<td>1</td>
</tr>
</tbody>
</table>

### Table 87: Debug Group

<table>
<thead>
<tr>
<th>Key</th>
<th>Valid Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>aie_profile</td>
<td>[true</td>
<td>false]</td>
</tr>
<tr>
<td>aie_profile_interval_ms</td>
<td>&lt;int&gt;</td>
<td>Specifies the interval in milliseconds to read the AI Engine trace buffers. This is useful when you want to increase the rate of dumping data to the disk to avoid overflowing the buffers and losing data. The default interval is 20 ms.</td>
</tr>
<tr>
<td>aie_trace</td>
<td>[true</td>
<td>false]</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Note:</strong> This option is disabled when specified with profile=true.</td>
</tr>
</tbody>
</table>
Table 87: Debug Group (cont’d)

<table>
<thead>
<tr>
<th>Key</th>
<th>Valid Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>aie_trace_buffer_size</td>
<td>&lt;value{K</td>
<td>M</td>
</tr>
<tr>
<td>app_debug</td>
<td>[true</td>
<td>false]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• true: Enable.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• false: Disable. This is the default value.</td>
</tr>
<tr>
<td>continuous_trace</td>
<td>[on</td>
<td>off]</td>
</tr>
<tr>
<td>continuous_trace_interval_ms</td>
<td>&lt;int&gt;</td>
<td>Specifies the interval in milliseconds to read the trace buffers. This is useful when trace data is captured in FIFO and you want to increase the rate of dumping data to the disk to avoid overflowing the buffers and losing data. The default interval is 10 ms. Note: This option requires the use of <code>continuous_trace=true</code>.</td>
</tr>
<tr>
<td>data_transfer_trace=&lt;arg&gt;</td>
<td>[coarse</td>
<td>fine</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• coarse: Show CU transfer activity from beginning of first transfer to end of last transfer (before compute unit transfer ends).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• fine: Show all AXI-level burst data transfers.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• off: Turn off reading and reporting of device-level trace during runtime. This is the default value.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Note: For OpenCL API profiling this feature requires both <code>profile=true</code> and <code>timeline_trace=true</code> to be set.</td>
</tr>
</tbody>
</table>
### Table 87: Debug Group (cont’d)

<table>
<thead>
<tr>
<th>Key</th>
<th>Valid Values</th>
<th>Description</th>
</tr>
</thead>
</table>
| debug             | [true|false]  | Enables kernel debug functionality for software emulation and hardware emulation. If debug=true, then a separate GDB process can attach to the emulation process via xrt_server for kernel debug.  
• true: Enable.  
• false: Disable. This is the default value. |
| lop_trace         | [true|false]  | Enables or disables low overhead profiling of OpenCL API calls in the host application. Low overhead profiling produces a reduced timeline of events written to lop_trace.csv. This reports only host side events, eliminating the overhead of device side profiling and significantly reducing the impact on performance.  
• true: Enable.  
• false: Disable. This is the default value.  
**TIP:** lop_trace=true should not be specified with profile=true, or the tool will perform standard profiling and trace. |
| power_profile     | [true|false]  | When this option is enabled, power data is captured from the accelerator card during the application runtime. The power profile reports the avg/min/max power for each rail on the card. The logged data is written to power_profile_<device>.csv, and can be viewed in Vitis analyzer. |
| power_profile_interval_ms | <int> | Specifies the interval in milliseconds to read the power information. The default interval is 20 ms. |
| profile           | [true|false]  | Enables or disables OpenCL host code profiling.  
• true: Enable.  
• false: Disable. This is the default value.  
When true, the runtime generates the profile_summary.csv file for OpenCL API calls. This will summarize host code events and some device level information. When false, no profile monitoring is performed at all.  
**IMPORTANT! Because this feature accesses device information from the PL region, it will override and disable any other profiling options that access the PL region of the device, such as aie_trace and vitis_ai_profile.** |
Table 87: **Debug Group** (cont’d)

<table>
<thead>
<tr>
<th>Key</th>
<th>Valid Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>profile_api</td>
<td>[true</td>
<td>false]</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Note:</strong> This option is disabled when specified with <code>profile=true</code>.</td>
</tr>
<tr>
<td>stall_trace=&lt;arg&gt;</td>
<td>[dataflow</td>
<td>memory</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• <strong>off:</strong> Turn off stall trace information.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• <strong>all:</strong> Record all stall trace information.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• <strong>dataflow:</strong> Intra-kernel streams (for example, writing to full FIFO between dataflow blocks).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• <strong>memory:</strong> External memory stalls (for example, AXI4 read from the DDR memory).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• <strong>pipe:</strong> Inter-kernel pipe for OpenCL kernels (for example, writing to full pipe between kernels).</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Note:</strong> For OpenCL API profiling this feature requires both <code>profile=true</code> and <code>timeline_trace=true</code> to be set. For XRT API profiling, this feature requires both <code>xrt_profile=true</code> and `data_transfer_trace=fine</td>
</tr>
<tr>
<td>timeline_trace</td>
<td>[true</td>
<td>false]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• <strong>true:</strong> Enable.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• <strong>false:</strong> Disable. This is the default value.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>This option will enable data gathering for the timeline trace report. However, without adding Acceleration Monitors and AXI Performance Monitor IP into the kernels, the timeline will only show host information. At a minimum, to get compute unit start and end times, the CU needs to be built with <code>--profile.exec</code> as described in <code>--profile</code> Options.</td>
</tr>
</tbody>
</table>

**IMPORTANT!** This also requires the use of `profile=true` or no timeline data is captured.
### Table 87: Debug Group (cont’d)

<table>
<thead>
<tr>
<th>Key</th>
<th>Valid Values</th>
<th>Description</th>
</tr>
</thead>
</table>
| trace_buffer_size    | <value{K|M|G}> | Used with timeline_trace=true, this option specifies the size of the memory allocated to capture trace data. This helps to ensure you can capture enough trace data. The value is specified as the amount of memory to allocate, for example, 64K, 200M, 1G.  
**Note:** This requires the device binary (.xclbin) to be linked with the --trace_memory option as explained in Vitis Compiler General Options. |
| vitis_ai_profile     | [true|false]  | Enables DPU counter profiling for Vitis AI library.  
**Note:** This option is disabled when specified with profile=true. |
| xrt_profile          | [true|false]  | Enables capturing trace data of host applications using the XRT Native API. This option can be used with profile=true. |

### Table 88: Emulation Group

<table>
<thead>
<tr>
<th>Key</th>
<th>Valid Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>aliveness_message_interval</td>
<td>Any integer</td>
<td>Specifies the interval in seconds that aliveness messages need to be printed. The default is 300.</td>
</tr>
</tbody>
</table>
| debug_mode           | [off|batch|gui|gdb] | Specifies how the waveform is saved and displayed during emulation.  
- **off:** Do not launch simulator waveform GUI, and do not save wdb file. This is the default value.  
- **batch:** Do not launch simulator waveform GUI, but save wdb file  
- **gui:** Launch simulator waveform GUI, and save wdb file  
- **gdb:** Launch simulator in gdb mode to debug the kernel using gdb. This mode does not have waveform support and does not save any wdb file either.  
**Note:** The kernel needs to be compiled with debug enabled {v+ -g} for the waveform to be saved and displayed in the simulator GUI. |
<table>
<thead>
<tr>
<th>Key</th>
<th>Valid Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>print_infos_in_console</td>
<td>[true</td>
<td>false]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• true: Print in user’s console. This is the default value.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• false: Do not print in user console.</td>
</tr>
<tr>
<td>print_warnings_in_console</td>
<td>[true</td>
<td>false]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• true: Print in user’s console. This is the default value.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• false: Do not print in user console.</td>
</tr>
<tr>
<td>print_errors_in_console</td>
<td>[true</td>
<td>false]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• true: Print in user’s console. This is the default value.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• false: Do not print in user’s console.</td>
</tr>
<tr>
<td>user_pre_sim_script</td>
<td>Path to Tcl file</td>
<td>For the first run, run simulation in GUI mode. Add signals that you want to add. Copy the commands from the Tcl console and save into a Tcl script. For the next run, pass the Tcl script in batch mode.</td>
</tr>
<tr>
<td>user_post_sim_script</td>
<td>Path to Tcl file</td>
<td>Any post operations can be specified in the Tcl and pass to the switch. All the command provided in the Tcl will get executed after simulation is completed.</td>
</tr>
<tr>
<td>xtlm_aximm_log</td>
<td>[true</td>
<td>false]</td>
</tr>
<tr>
<td>xtlm_axis_log</td>
<td>[true</td>
<td>false]</td>
</tr>
<tr>
<td>timeout_scale</td>
<td>na/ms/sec/min</td>
<td>Timeout support for clPollStream API in emulation. Provides a scale for the timeout specified in clPollStream API. The timeout specified in the code is specified in ms, and might not work for emulation. Therefore use the timeout_scale to map ms to another scale if needed for emulation.</td>
</tr>
</tbody>
</table>

**IMPORTANT!** Timeout is not enabled in emulation by default. Use this option to enable clPollStream timeout.
Chapter 34

HLS Pragmas

Optimizations in Vitis HLS

In the Vitis software platform, a kernel defined in the C/C++ language, or OpenCL™ C, must be compiled into the register transfer level (RTL) that can be implemented into the programmable logic of a Xilinx device. The v++ compiler calls the Vitis High-Level Synthesis (HLS) tool to synthesize the RTL code from the kernel source code.

The HLS tool is intended to work with the Vitis IDE project without interaction. However, the HLS tool also provides pragmas that can be used to optimize the design, reduce latency, improve throughput performance, and reduce area and device resource usage of the resulting RTL code. These pragmas can be added directly to the source code for the kernel.

The HLS pragmas include the optimization types specified in the following table.

For detailed pragma information, refer to the Vitis HLS Flow.

Table 89: Vitis HLS Pragmas by Type

<table>
<thead>
<tr>
<th>Type</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kernel Optimization</td>
<td>• <code>pragma HLS aggregate</code></td>
</tr>
<tr>
<td></td>
<td>• <code>pragma HLS bind_op</code></td>
</tr>
<tr>
<td></td>
<td>• <code>pragma HLS bind_storage</code></td>
</tr>
<tr>
<td></td>
<td>• <code>pragma HLS expression_balance</code></td>
</tr>
<tr>
<td></td>
<td>• <code>pragma HLS latency</code></td>
</tr>
<tr>
<td></td>
<td>• <code>pragma HLS reset</code></td>
</tr>
<tr>
<td></td>
<td>• <code>pragma HLS top</code></td>
</tr>
<tr>
<td>Function Inlining</td>
<td>• <code>pragma HLS inline</code></td>
</tr>
<tr>
<td>Interface Synthesis</td>
<td>• <code>pragma HLS interface</code></td>
</tr>
<tr>
<td>Task-level Pipeline</td>
<td>• <code>pragma HLS dataflow</code></td>
</tr>
<tr>
<td></td>
<td>• <code>pragma HLS shared</code></td>
</tr>
<tr>
<td></td>
<td>• <code>pragma HLS stream</code></td>
</tr>
<tr>
<td>Pipeline</td>
<td>• <code>pragma HLS pipeline</code></td>
</tr>
<tr>
<td></td>
<td>• <code>pragma HLS occurrence</code></td>
</tr>
<tr>
<td>Loop Unrolling</td>
<td>• <code>pragma HLS unroll</code></td>
</tr>
<tr>
<td></td>
<td>• <code>pragma HLS dependence</code></td>
</tr>
</tbody>
</table>
Table 89: Vitis HLS Pragmas by Type (cont’d)

<table>
<thead>
<tr>
<th>Type</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loop Optimization</td>
<td>• <code>pragma HLS loop_flatten</code></td>
</tr>
<tr>
<td></td>
<td>• <code>pragma HLS loop_merge</code></td>
</tr>
<tr>
<td></td>
<td>• <code>pragma HLS loop_tripcount</code></td>
</tr>
<tr>
<td>Array Optimization</td>
<td>• <code>pragma HLS array_partition</code></td>
</tr>
<tr>
<td></td>
<td>• <code>pragma HLS array_reshape</code></td>
</tr>
<tr>
<td>Structure Packing</td>
<td>• <code>pragma HLS aggregate</code></td>
</tr>
<tr>
<td></td>
<td>• <code>pragma HLS dataflow</code></td>
</tr>
</tbody>
</table>
Using the Vitis Analyzer

This section contains the following chapters:

- Working with Reports
- Vitis Analyzer GUI and Window Manager
- Platform and System Diagrams
- AI Engine Graphs and Arrays
- Configuring the Vitis Analyzer
- Setting Guidance Thresholds
- Creating an Archive File

The Vitis™ analyzer is a utility that allows you to view and analyze the reports generated while building and running the application. It is intended to let you review reports generated by both the Vitis compiler when the application is built, and the Xilinx® Runtime (XRT) library when the application is run. The Vitis analyzer can be used to view reports from both the v++ command line flow, and the Vitis integrated design environment (IDE). You will launch the tool using the `vitis_analyzer` command (see Setting Up the Vitis Environment).

When first launched, the Vitis analyzer opens with a home screen that lets you open summary files, binary containers, or directories. Clicking any of these links opens a file browser that allows you to select a specific file of the type described.
• **Open Summary:** Report Summaries are collections of reports related to specific stages of application development in the Vitis tool. There are summaries created for the two steps of the build process, compile and link, and from the run process when the application is executed. Selecting **Open Summary** lets you open one of the following:

  - **Compile Summary:** The Compile Summary report is generated by the `v++` command during compilation, provides the status of the kernel compilation process. When viewing the Compile Summary report the tool also references the following reports generated during compilation: Kernel Estimate, Kernel Guidance, HLS Synthesis, and compilation log.

  - **Link Summary:** The Link Summary report is created by the `v++` command during linking and creation of the `.xclbin` file. When viewing the Link Summary report the tool also references the following reports generated during linking: System Estimate, System Guidance, Timing Summary, Utilization, Operation Trace, Platform and System Diagrams, and linking logs. When you open a Link Summary, the Vitis analyzer will automatically open the associated Compile Summaries of kernels that were linked into the `.xclbin` file.

  **TIP:** Timing Summary and Utilization are only generated when the build targets Hardware (as opposed to emulation).
- **Package Summary**: The Package Summary report is generated by the `v++ --package` command, and provides information related to the generation of emulation scripts and SD card output. When viewing the Package Summary report the tool also references the configuration file used on the command line, and the log file generated.

- **Run Summary**: The Run Summary report is created by the XRT library during the application execution, and provides a summary of the run process. When viewing the Run Summary report the tool also references the following reports generated during the application run: Guidance, Profile Summary, Application Timeline, Platform and System Diagrams, and Simulation Waveforms when enabled.

**IMPORTANT!** Reports generated during the runtime execution of an application generally require some prior setup as described in *Profiling the Application*.

When you run the application after the `v++` build process, the ID from the Link Summary is assigned to the Run Summary. When you open Run Summary and Link Summary, Vitis Analyzer links them based on the shared ID.

- **AI Engine Compile and Run Summaries**: Vitis analyzer also lets you view reports from the Versal AI Engine compile and run processes. These reports are generated by the `aiecompiler` and `aiesimulator` as described in the *Versal ACAP AI Engine Programming Environment User Guide* (UG1076).

**TIP:** In the Vitis IDE, the project hierarchy consists of a top-level system project with sub-projects which contain elements of the design: the processor application, the hardware kernels, the hardware link project, and the AI Engine project. The various summary reports described above can be found in the appropriate sub-project: Compile Summary will be in the specific hardware kernel projects, Link Summary will be in the hardware link project, Package Summary will be found in the system project, the Run Summary will be found in the processor application, and AI Engine summaries will be found in the `.//Work` folder or the `.//aiesimulator` output.

- **Open Binary Container**: Opens the selected `.xclbin` file to display the Platform Diagram and the System Diagram for the build.

- **Open Directory**: Specifies a directory to open. The tool recursively examines the contents of the directory and displays a dialog box allowing you to select which type of files to open and which individual files to open.

**TIP:** The Open Recent section of the home screen provides a list of recently opened summaries and reports to let you quickly reopen them.

The `vitis_analyzer` command lets you open the tool to the home screen, as discussed above, or specify a file to load when opening the tool. You can open a file by specifying the name of the file to open. You can open the Vitis analyzer with any of the files supported by the tool, as described in *Working with Reports*. For example:

```
vitis_analyzer project1.run_summary
```
You can access the command help for `vitis_analyzer` command by typing the following:

```
vitis_analyzer -help
```
Working with Reports

Generally, the Summary reports provide you a great overview of the specific steps in building and profiling the application to get a good view of where the application is with regard to performance and optimization. For individual kernels review the Compile Summary. For the device binary (xclbin), start with the Link Summary, which also loads the Compile Summaries for linked kernels. For embedded processor and Versal™ AI Engine systems review the Package Summary. For profiling data related to the application execution start with the Run Summary.

In addition, the File menu offers commands to let you open individual reports, and directories of reports.

**Figure 78: Open Directory**

- **Open Directory**: Specifies a directory to open. The tool recursively examines the contents of the directory and displays a dialog box allowing you to select which type of files to open and which individual files to open.
• **Open Binary Container:** FPGA binary, `<name>.xclbin`, created by the compilation and linking process as described in Section III: Building and Running the Application.

• **Open Report:** Opens one of the report files generated by the Vitis™ core development kit during compilation, linking, or running the application. The reports you can open include:
  
  • **Application Timeline:** Refer to Application Timeline.
  
  • **Profile Summary:** Refer to Profile Summary Report.
  
  • **Waveform:** Waveform database and waveform config file as described in Waveform View and Live Waveform Viewer.
  
  • **Utilization:** A resource utilization report generated by the Vivado® tool when you build the system hardware (HW) target.

The Vitis analyzer can also open Kernel Estimate, Operation Trace, AI Engine Trace, Timeline Trace, System Estimate, Log, and Timing Summary reports. Refer to Profiling the Application for more information on the individual reports generated by the build and run processes.

The Vitis analyzer displays log files rendered for improved readability. Some key additions the tool provides when viewing log files include line wrapping, message severity tagging (Error, Warning, Info), added hyperlinks to referenced files, search capability, and live log monitoring. This last feature lets you open a log file in-process and see it rendered in real time.

**Viewing Report Contents**

The features of the Vitis analyzer depend on the specific report you are viewing. When the report is structured like a spreadsheet you interact with the report like a spreadsheet, selecting rows or cells of data, and sorting columns by clicking on the column header. When the report is graphical in nature, you can interact with the report by zooming into the report to view details, and zooming out to view more information. The Vitis analyzer supports the following mouse strokes to let you quickly zoom into and out of a graphical report:

• **Zoom In:** Press and hold the left mouse button while dragging the mouse from top left to bottom right to define an area to zoom into.

• **Zoom Out:** Press and hold the left mouse button while drawing a diagonal line from lower left to upper right. This zooms out the window by a variable amount. The length of the line drawn determines the zoom factor applied. Alternatively, press Ctrl and scroll the wheel mouse button down to zoom out.

• **Zoom Fit:** Press and hold the left mouse button while drawing a diagonal line from lower right to upper left. The window zooms out to display the entire device.

• **Horizontal scrolling:** In a report such as the Application Timeline, you can hold the shift button down while scrolling the middle mouse roller to scroll across the timeline.

• **Panning:** Press and hold the wheel mouse button while dragging to pan.
Vitis Analyzer GUI and Window Manager

As shown below, the Vitis™ analyzer workspace is arranged into three views, which includes the Report Navigator, Reports, and the Source Code view. The different views can be opened, closed, and rearranged as needed.

- **Report Navigator**: On the left side, this view lists all open summary files and associated reports. You can use this view to quickly find and open a report. In the figure above you can see the Link Summary is opened, and the Compile Summary is contained within it, and all their related reports are listed in the Report Navigator.
When you click any file in Report Navigator, it opens as a new tab in the Report view. Opening a file adds a green dot next to the report name in Report Navigator to let you quickly determine if a report is already open in the tool.

**TIP:** You can right-click the Compile Summary file to **Open HLS Project**, or right-click the Link Summary or Run Summary to **Open Vivado Project** if needed. The Vivado project cannot be opened for the Software Emulation build.

- **Reports:** The center area displays the contents of the summary files and open reports. You can have multiple reports open in the Reports view, and quickly change from one report to another by selecting the window tab at the top of the view.

  All the reports related to the Compile Summary, Link Summary, or Run Summary are grouped together within a single container. You can arrange the reports for a container in different ways, using the New Horizontal Group, New Vertical Group, or Float commands for the reports in a container. Multiple Summary reports can be opened and the contents managed as collections.

  Reports that are currently opened in Vitis analyzer will display an "out-of-date" banner if the summary files or reports have been updated on the disk, due to recompiling or rerunning the application for instance. You can keep working in the open report, or reload the updated files.

- **Source Code:** The optional Source Code view is opened on the right side of the workspace. This lets you view and edit kernel source code, based on feedback from the System Guidance report for instance. You can open the Source Code window by selecting a link in the Guidance report, or by right-clicking the Compile Summary in the Report Navigator and clicking **Open Source**.

- **Global Search:** Vitis analyzer provides a search field next to the **Help** command in the main menu at the top of the display. You can use it to search the loaded Summary, associated reports, and source files. The search dialog box provides options to limit the scope of the search of the search terms, as shown in the following figure.
The Report Navigator and Source Code views can be collapsed by clicking the Minimize button in the toolbar, and restored by clicking the tab for the collapsed view.

To close all the open source code views, select the File → Close All Sources command.

To close all the open reports associated with a Summary report, such as the Link Summary, right-click the Summary in the Report Navigator view and select Close Tabs. This closes all open reports associated with the summary in the Report view.

To close a Summary file, such as the Link Summary, right-click the file in the Report Navigator area and select Close File. Closing the Summary file closes all associated reports and files. Therefore, closing the Link Summary also ends the Compile Summary for the build.

To close all files displayed in the Report Navigator select the File → Close All Files command. This returns Vitis analyzer to the home screen.

Diff Two Text Files

The Vitis analyzer lets you compare two reports of the same type. This opens a report window similar to the one shown below.
The text-based comparison reports include:

- Kernel estimate
- System estimate
- Timing summary
- Log files

To compare reports, you must have two of the same type of reports listed in the Report Navigator window, or opened in the Reports view. Right-click a supported report in the Report Navigator, or in the Report view, and select the **Diff with >** command. This command lets you specify another report of the same type to compare with the currently selected report.

---

**Cross-Probing between Reports**

The Vitis analyzer supports a variety of selectable objects within different reports and views:
• **Compute Units (CU):** Selectable in the System Diagram and associated Compute Units table. Selecting the kernel selects associated compute units and vice versa. The CU is found in the Utilization report, the Profile Summary, the Application Timeline, and the Waveform view.

• **CU ports:** Selectable in the System Diagram.

• **Kernels:** Selectable in the System Diagram and associated Kernels table. Note that selecting the kernel also selects the CUs and vice versa. Kernels are found in the Link Summary, Utilization report, System Guidance (under Accelerators), Profile Summary, and the Waveform view.

• **Kernel ports:** Selectable in the System Diagram.

• **Function arguments:** Selectable the System Diagram and Kernels table.

• **AXI interconnects:** Selectable in the System Diagram. This selects all connections to a memory bank.

• **AXI ports:** Selectable in the System Diagram. These are "flattened", for example, they are the same for all kernels. Shown in the Profile Summary, and the Waveform view (data transfers).

• **Memory resources:** Selectable in the Platform and System Diagrams, and associated Memories table. Shown in the Profile Summary (data transfers: kernels to global memory).

• **Host CPU:** Selectable in the Platform and System Diagrams.

*Figure 82: Cross-Probing Reports*
The Vitis analyzer supports cross-probing between reports, such as within the System Diagram and from the Guidance View to other views. The Guidance view will provide an actionable resolution for a violation reported, and you can use cross-probing from the violation to quickly navigate to other reports and views.

Cross-probing can be bidirectional or unidirectional, depending on the report. The Guidance report lets you select objects in other reports, but does not support cross-probing from other reports or views.

- **Bidirectional** cross-probing between the System Diagram and Profile Summary report. Selecting a kernel, compute unit or compute unit port in one selects it in the other. Selecting a kernel also selects associated CUs in the reports.

- **Unidirectional** cross-probing from the Guidance to the System Diagram and Profile Summary report. The Details column of the guidance report displays hyperlinks that correspond to design objects such as kernels, CUs, etc.
  - Clicking a kernel, compute unit, or compute unit port hyperlink in Guidance selects it in the System Diagram and Profile Summary.
  - Clicking a memory or kernel argument hyperlink selects it in the System Diagram, but not the Profile Summary.
  - Clicking a kernel port hyperlink in Guidance selects a CU port in the System Diagram.
  - In some cases, the Details column displays a hyperlink for a value, for example, **82.601%**.
    - Clicking a value hyperlink selects the corresponding design object and navigates to the associated section in the Profile Summary report.
    - If the report is already open but is hidden behind another tab, it will be brought to the front.
    - If the report is not open, clicking a value hyperlink will open the report.
  - Guidance hyperlinks also have tooltips explaining what the click action does.
  - Additionally, selecting other objects such as the host, memories, AXI interconnects, and kernel arguments in the System Diagram does not cross-probe to the Profile Summary because the report does not represent these as selectable objects.
Platform and System Diagrams

The Platform and System Diagrams display a representation of the platform resources, and the kernel code integrated onto the platform. They can be viewed in the Vitis™ analyzer from the Link Summary, the Run Summary, or the .xclbin for a project.

The Platform Diagram is a block diagram of the target platform, before the .xclbin is loaded. This diagram shows all DDR banks and PLRAM available, and their available connections. A table at the bottom displays details of bank names with types of memories, their sizes and which SLR region these are available.

The System Diagram shows memory banks or PLRAMs used by the .xclbin. You can also see how the function arguments of Compute Units are connected to AXI4 interfaces. A table at the bottom of the System Diagram displays information for each Compute Unit, Kernels, and Memories. For designs that include AI Engine kernels, the System Diagram also displays information related to those kernels. Features of the System Diagram include the following:

- Name of the kernel with an indication which SLR this is available.
- LUT%
- Register %
- BRAM % used
- URAM % used
- DSP % used
When Run Summary is loaded, the System Diagram includes profile data from the run. The Vitis analyzer automatically runs `perf_analyze` when opening a Run Summary that contains a `profile_summary.csv` as well as optionally `profile_kernels.csv` for Hardware Emulation. The profile data is added to the table at the bottom of the System Diagram and can also be displayed in the diagram, as shown in the figure above.

The resource information from the table can also be displayed in a box next to each kernel or CU in the System Diagram. The **Settings** command lets you display or hide Unused Memory, Interface Ports, Profile Info, and Resource info.

---

**Figure 83: System Diagram with Profile Data**

**Figure 84: Show Port Info**
The ports on a Compute Unit can display the transfer rates on the system diagram, as well as CU Utilization percentage. CU port transfer rates are taken from the Kernel Transfer section of the Profile Summary report. CU utilization statistics are taken from the Compute Unit Utilization section of Profile Summary. The performance data is available as long as Profiling was enabled for Hardware and Hardware Emulation run, using the Vitis compiler --profile option as described in --profile Options.
Chapter 38

AI Engine Graphs and Arrays

The AI Engine Graph and Array diagrams provide a quick overview of the structure of the ADF graph application implemented in the AI Engine tiles.

Figure 85: AI Engine Graph Application

The AI Engine Graph view shows the connectivity of the ADF graph as seen by the AI Engine compiler. The canvas shows nodes representing kernels, kernel arguments, memory buffers, and primary inputs and outputs, with edges drawn to represent the connectivity between these elements.

To the right of the canvas, a Settings panel that can be used to customize the view of the graph, by letting you show or hide elements of the displayed graph.
Below the graphical view is a set of tables containing all of the objects drawn in the graphical view: kernels, connections, paths, memories, etc. Selecting an element in one of these tables will cross-probe to the graphical view and vice versa.

*Figure 86: AI Engine Array Diagram*

The Array view shows how the ADF graph is spatially placed on the AI Engine tile array. The array canvas contains all the core and memory components of the array. Kernels are drawn in the core where they are programmed, and connectivity between cores and/or memory are shown with connectivity lines. There is an abstract representation of the PL area as needed to model PL cores and interfaces to the programmable logic region.

To the right of the canvas, a Settings panel that can be used to customize the view of the array, by letting you show or hide elements of the resources and/or tile array.
Chapter 39

Configuring the Vitis Analyzer

The **Tools → Settings** command opens the Vitis Analyzer Settings dialog box as shown below.

*Figure 87: Settings Dialog Box*

In the General settings, the following can be configured:

- **Default Directory**: Specifies the default directory used by the Vitis™ analyzer when it is opened.
- **Recent**: Configures the tool to restore the workspace when reopening the Vitis analyzer, and specify the number of entries to display for **File → Open Recent** commands.
- **Report Tabs**: Defines the number of reports and views that can be opened in the main Reports window.

In the Display settings you can configure the following features of the display:

- **Scaling**: Sets the font scaling to make the display easier to read on high resolution monitors. Use OS font scaling uses the value set by the OS for your primary monitor. User-defined scaling allows you to specify a value specific to the Vitis analyzer.
• **Spacing**: Sets the amount of space used by the Vitis IDE. Comfortable is the default setting. Compact reduces the amount of space between elements to fit more elements into a smaller space.

The Reports section configures the Vitis analyzer to also open specified reports when opening the Compile Summary, Link Summary, Run Summary, or Binary Container reports:

• **Compile Summary**: Select which reports are listed in the Report Navigator view, and opened with the Compile Summary.

• **Link Summary**: Select which reports are listed and opened with the Link Summary.

• **Run Summary**: Select which reports are listed and opened with the Run Summary.

• **Binary Container**: Select which reports are listed and opened with the Binary Container.

• **AI Engine Compile Summary**: Select which reports are listed and opened with the AI Engine Compile Summary.

• **AI Engine Run Summary**: Select which reports are listed and opened with the AI Engine Run Summary.

For Window Behavior settings, the following can be configured:

• **Warnings**: Shows warning when exiting or just exits the Vitis analyzer.

• **Alerts**: Issues an alert when you are running the tool on an unsupported operating system.

After configuring the tool, click **OK**, **Apply**, or **Cancel**. You can also use the **Restore** command to restore the defaults settings of the tool.

**Layout**

The Vitis analyzer also lets you arrange reports in different layouts, and save those layouts for reuse. This allows you to arrange two reports for comparison, or to check one report against the other. The **Layout** menu in the main menu lets you switch between existing layouts, or save a custom report layout using the **Save Layout As** command. You can also **Reset Layout** to return it to its saved configuration, or **Remove Layout** to eliminate saved layout.
Chapter 40

Setting Guidance Thresholds

Guidance messages reported in the Run Guidance report are triggered by specific rules and value thresholds that are defined within the tool. Some of these rules and value thresholds are user-modifiable within Vitis™ analyzer. When the Run Guidance report is opened, the modifiable values will appear as links in the Threshold column of the report, as shown in the following figure.

Figure 88: Run Guidance Threshold

<table>
<thead>
<tr>
<th>Name</th>
<th>Threshold</th>
<th>Actual</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Profiling</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Kernel: SgdLR_1(2)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Compute Unit: SgdLR_1(1)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>UNUSED_CUS</td>
<td>&gt; 0</td>
<td>0</td>
<td>Compute unit SgdLR_1 on device xilinx_u280_xdma_201920_3-0 was used 0 time(s).</td>
</tr>
<tr>
<td>KERNEL_PORT_DATA_WIDTH</td>
<td>&lt; 512</td>
<td>32</td>
<td>Port m_wu_pmem1 has a data width of 32.</td>
</tr>
<tr>
<td>System(2)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Device: xilinx_u280_xdma_201920_3-0(1)</td>
<td>&gt; 0.000</td>
<td>0.000</td>
<td>Device xilinx_u280_xdma_201920_3-0 was utilized for 0.000 msec.</td>
</tr>
<tr>
<td>Host Data Transfers</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HOST_MIGRATE_MEM</td>
<td>&gt; 0</td>
<td>0</td>
<td>Migrate Memory OpenGL APIs were used 0 time(s).</td>
</tr>
</tbody>
</table>

Clicking a link in the Threshold column will display the Edit Guidance Threshold dialog box, as shown below. The top part of the dialog box displays the current operator and threshold for the selected rule, and the lower part lets you edit the value.

Figure 89: Edit Guidance Threshold

Edit threshold for guidance rule DEVICE_UTIL
Default setting: > 0.0

Threshold value: 12 (>= 0)  

Update guidance report

OK  Cancel
The figure above shows the Threshold value being redefined with a user-specified value. In some cases, a range of accepted values is displayed and the tool checks that the provided value falls within that range. In the example above, the specified value must be greater than or equal to 0.

The Edit Guidance Threshold dialog box also provides a Reset threshold command that allows you to reset the user-defined value to the hard-coded value provided by the tool. This is the seen in the figure above.

**TIP:** The menu bar for the Run Guidance report also has a Reset Guidance Thresholds command to let you clear all user-modified thresholds from the report.

The Update guidance report check box lets you specify if the guidance report will be rerun after the value is updated. If you deselect this check box, the threshold value will be changed as specified, and the report will be marked out-of-date due to the new user-specified threshold. You will need to manually Reload the report to see the impact of the user values.

Click OK to change the value, or Cancel to close the dialog box without change.

**Import/Export Rule Thresholds**

The rules as to which threshold values can be modified, and the default values of all rules, are stored within the tool installation. You can override these rules with a custom rule file, which you can export and import into different projects in Vitis analyzer.

When you have customized one or more rules in the Run Guidance report, an Export User Guidance Thresholds command becomes active in the toolbar menu of the report. You can see this command displayed in the top figure above. This lets you export the customized value thresholds to reuse them in other designs.

A sample file is shown below:

```plaintext
profile_rules =
{
    {
        id = "HOST_MIGRATE_MEM";
        value = '1';
    },
    {
        id = "DEVICE_UTIL";
        value = '3';
    }
}:
version = '1';
```

The user threshold file does not need to contain all guidance rules, only the rules where the value has changed, and only the rule id and value fields are needed:

- **id** is the name of the rule as displayed in the Name column of the Run Guidance report.
- **value** is the user-specified value, displayed in the Threshold column.
The Vitis analyzer IDE also lets you **Import User Guidance Thresholds** to reuse the custom thresholds in other projects. You can export a `user-thresholds.cfg` file, edit the values, and import the file as needed. Importing a user-thresholds file regenerates the Run Guidance report to use the imported values.

**IMPORTANT!** Importing a user-guidance threshold file will overwrite any existing user-modified thresholds.
Creating an Archive File

At times while developing your project, you might introduce changes in your host or kernel code that will completely alter the contents or quality of the various build and run summaries produced by the tool. Your existing reports and analysis data will be overwritten unless you manually save the relevant files. The Archive Summary command lets you save all relevant files with an open Link Summary or Run Summary.

**TIP:** This feature lets you quickly share design reports with other team members by sharing the archive summary.

Select **File → Archive Summary** menu command, or right-click a summary in the Report Navigator and select **Archive Summary.** This opens the Archive Summary dialog box as shown below.

![Archive Summary Dialog Box](image)

Archive file names must have the extension of `link_summary.archive`, or `run_summary.archive` to be recognized by the Vitis analyzer. The contents of the archive depend on the summary report being archived.

The contents of the Link Summary include these file extensions:

- **Binary container(s):** .xclbin
- **Kernel(s):** .xo
The contents of the Run Summary include these file extensions:

- **Run summary**: .run_summary
- **System diagram**: .run_summary
- **Platform diagram**: .run_summary
- **Guidance**: Not required, generated using via perf_analyze and the profile_summary.csv
- **Profile summary**: profile_summary.csv
- **Application timeline**: timeline_trace.csv
- **Waveform report**: .wdb

**IMPORTANT!** For AI Engine designs, when a `run_summary` contains a trace report but does not have an entry for the AI Engine work directory, Vitis analyzer cannot run `hwanalyze` to display the trace report. In this case, the tool will prompt you for the location of the AI Engine `compile_summary`, and will update the `run_summary` file on disk. If you archive the `run_summary` before providing this location, you will not be able to view the trace report from the archived `run_summary`.

You can also choose to save the `.xclbin` file extension, any compiled kernel object file extensions (.o and .xo), and the original source file extensions (.cpp, .c, and .cl) used to generate the summary reports.

**TIP:** Guidance is not saved because this is dynamically generated by Vitis™ analyzer from `profile_summary.csv`, and optionally `profile_kernels.csv`.

To open an existing archive file, use the **File → Open Summary** command and browse for the archive file. You can also open an archive file when launching Vitis analyzer:

```
vitis_analyzer design.archive
```

There is also a command line form of Archive Summary that you learn more about with:

```
archive_summary -help
```
Using the Vitis IDE

This section contains the following chapters:

- Vitis Command Options
- Creating a Vitis IDE Project
- Building the System
- Vitis IDE Debug Flow
- Configuring the Vitis IDE
- Project Export and Import
- Getting Started with Examples
- RTL Kernel Wizard
Vitis Command Options

In the Vitis™ integrated design environment (IDE), you can create a new application project or platform development project.

The *vitis* command launches the Vitis IDE with your defined options. It provides options for specifying the workspace and options of the project. The following sections describe the *vitis* command options.

**Display Options**

The following options display the specified information intended for review.

- **-help**: Displays help information for the Vitis core development kit command options.
- **-debug**: Launches the Vitis IDE to run debug on a command-line project.

  **TIP:** To view the help for the *vitis -debug* command, use `-debug -help`.

- **-version**: Displays the Vitis core development kit release version.

**Command Options**

The following command options specify how the *vitis* command is configured for the current workspace and project.

- **-workspace <workspace location>**: Specify the workspace directory for Vitis IDE projects.
- **{-lp <repository_path>}**: Add `<repository_path>` to the list of Driver/OS/Library search directories.
- **-eclipseargs <eclipse arguments>**: Eclipse-specific arguments are passed to Eclipse.
- **-vmargs <java vm arguments>**: Additional arguments to be passed to Java VM.
Chapter 43

Creating a Vitis IDE Project

In the Vitis™ IDE, you can create a new application project, or platform development project. The following section shows you how to set up a workspace, create a new Vitis IDE project, and use key features of the IDE.

Launch a Vitis IDE Workspace

1. Launch the Vitis IDE directly from the following command line.

   `$vitis`

   **IMPORTANT!** When opening a new target platform, to enter a Vitis core development kit command, ensure that you set it up as described in Setting Up the Vitis Environment.

   The Vitis IDE opens.

2. Select a workspace as shown in the following figure.
The workspace is the folder that stores your projects, source files, and results while working in the IDE. You can define separate workspaces for each project, or have a single workspace with multiple projects and types. The following instructions show you how to define a workspace for a Vitis IDE project.

3. Click **Browse** to navigate to and specify the workspace, or type the appropriate path in the Workspace field.

4. Optionally enable **Use this as the default and do not ask again** to set the specified workspace as your default choice and eliminate this dialog box in subsequent uses of the IDE.

   **Note:** To restore the dialog box, navigate to **Window → Preference → Additional → General → Startup and Shutdown → Workspaces**, and select **Prompt for workspace on startup**.

5. Click **Launch**.

   **TIP:** To change the current workspace from within the Vitis IDE, select **File → Switch Workspace**.

You have now created a workspace and can populate it with projects.

---

**Create an Application Project**

**TIP:** Example designs are provided with the Vitis core development kit installation and also on the Xilinx Vitis Examples GitHub repository. For more information, see **Getting Started with Examples**.

After launching the Vitis IDE, you can create a new Application Project.

1. Select **File → New → Vitis Application Project**, or if this is the first time the Vitis IDE has been launched, you can select **Create Application Project** on the Welcome screen.
The New Application Project wizard opens displaying a Welcome page that explains the process for new users. You can disable this from being shown again by enabling Skip welcome page next time.

2. Click Next to open the Platform page of the New Application Project wizard to specify a target platform.

A target platform is composed of a base hardware design and the meta-data used in attaching accelerators to declared interfaces. Use the Select a platform from repository tab to choose a platform for your project. You can enter a value in the Find field to limit the choices displayed to make it easier to locate the required platform. The bottom portion displays information related to the currently selected platform, as shown in the following figure.
Note: For platforms supported by a specific release refer to the Release Notes in the Section I: Getting Started with Vitis.

You can also add custom defined or third-party platforms into a repository. For more information, see Managing Platforms and Repositories.

3. In the Application Project Details page, specify the name in the Application project name field, as shown in the following figure.
By default, the tool creates a new system project for your application project. However, you can also add your application project to an existing system project, if one exists. The system project is a top-level manager for different projects that combine to create the system view.

4. Click **Next** to proceed.

   **Note:** If you selected Data Center accelerator card as your project platform in Step 2, the following page is not displayed and you can skip to Step 6.

5. If you select an **Embedded Acceleration** target platform on the Platform page, as displayed in the Flow column, the Domain page opens next as shown in the following figure.
Select a **Domain** from the list of existing domains on the platform, and Domain details are populated from your selection. The Domain defines the processor and operating used for running the host program on the target platform. You must also set the following Application Settings for the project to build correctly on the embedded platform:

- **Sysroot path**: The *sysroot* is part of the platform where the basic system root file structure is defined. The Sysroot path lets you define a new *sysroot* for your application.

- **Root FS**: Specify the location of the root file system.

- **Kernel Image**: Specify the location of the operating system kernel.

These option can be changed after the project is created from the **System Project Settings** in the Project Editor window.

6. Click **Next** to open the Templates page letting you select an application acceleration template for your new project.
You can select an **Empty Application** to create a blank project that you can import source files into and build from scratch. Also, you can use one of the provided template projects as a foundation for your new application project to help start your project, or help you learn the tool.

_TIP: Click the **Vitis IDE Examples** button, or the **Vitis IDE Libraries** button to install additional examples as discussed in **Getting Started with Examples**._

7. Click **Finish** to close the New Application Project wizard and open the project in the IDE.

_TIP: The Vitis IDE opens in the Design perspective as described in **Understanding the Vitis IDE**. Review this information if you are unfamiliar with the display._

When a new application acceleration project is created in the Vitis IDE, it includes a top-level system project, and nested within an application project for the host-code, a hardware kernels project for compiling kernel objects, and a **hw_link** project that is used for linking hardware kernels to the target platform and to each other. These projects are displayed in the Explorer view as shown in the following figure.
Managing Platforms and Repositories

You can manage the platforms that are available for use in Vitis IDE projects, from Xilinx → Add Custom Platform in the main menu of an open project, or from the Platform page present on both New Application and New Platform wizards.

Figure 91: New Platform Project
From the Platform page, manage the available platforms and platform repositories using one of the following options:

- **Add (➕):** Add your own platform to the list of available platforms. To add a new platform, navigate to the top-level directory of the custom platform, select it, and click OK. The custom platform is immediately available for selection from the list of available platforms.

- **Manage (🔧):** Add or remove standard and custom platforms. If a custom platform is added, the path to the new platform is automatically added to the repositories. When a platform is removed from the list of repositories, it no longer displays in the list of available platforms.

---

**Understanding the Vitis IDE**

When you open a project in the Vitis IDE, the workspace is arranged in a series of different views and editors, also known as a **perspective** in the Eclipse-based IDE. The tool opens with the Design perspective shown in the following figure.

*Figure 92: Vitis IDE – Default Perspective*

Some key views and editors in the default perspective include:
• **Explorer view:** Displays a file-oriented tree view of the project folders and their associated source files, plus the build files, and reports generated by the tool. You can use this to explore your project file hierarchy.

• **Assistant view:** Provides a central location to view and manage the projects of the workspace, and the build and run configurations of the project. You can interact with the various project settings and reports of the different configurations. From this view, you can build and run your Vitis IDE application projects, and launch the Vitis analyzer to view reports and performance data as explained in Section VI: Using the Vitis Analyzer.

• **Project Editor view:** Displays the current project, the target platform, the active build configuration, and specified hardware functions; allows you to directly edit project settings.

• **Console view:** Presents multiple views including the command console, design guidance, project properties, logs, and terminal views.

The Vitis IDE includes several predefined perspectives, such as the Vitis IDE perspective, the Debug perspective, and the Performance Analysis perspective. To quickly switch between perspectives, click the perspective name in the upper right of the Vitis IDE.

You can arrange views to suit your needs by dragging and dropping them into new locations in the IDE, and the arrangement of views is saved in the current perspective. You can close windows by selecting the Close (X) button on the View tab. You can open new windows by using the **Window → Show View** command and selecting a specific view.

To restore a perspective to the default arrangement of views, make the perspective active and select **Window → Reset Perspective**.

To open different perspectives, select **Window → Open Perspective**.

---

**Adding Sources**

A project consists of many different source files, including C/C++ files and headers for the Host Application, C/C++ Kernels, compiled Xilinx object (XO) files containing RTL kernels as discussed in RTL Kernels, or HLS kernels as described in Compiling Kernels with the Vitis HLS. You can add these source files as needed to support your application.

Each individual project within a system project requires its own source and data files. The host application code (host.cpp) is added to the ./src folder in the processor application project, and the kernel code (kernel.cpp), or the compiled kernel.xo files are added to the ./src folder of the kernel application project. You can add these files using the **Import Sources** command as explained in the next section.
Add Source Files

1. With the project open in the Vitis IDE, to add source files, right-click the src folder in the Project Explorer, and click Import Sources.

This displays the Import Sources dialog box shown in the following figure.

2. In the dialog box, for the From directory field, click the Browse button to select the directory from which you will import sources.

3. In the Into folder field, make sure the folder specified is the src folder of the project.

4. Select the desired source files by enabling the check box next to the file name, and click Finish.

IMPORTANT! When you import source files into a workspace, it copies the file into the workspace. Any changes to the files are lost if you delete the workspace.

After adding source files to your project, you are ready to begin configuring, building, and running the application. To open a source file in the built-in text editor, expand the src folder in the Project Explorer and double-click on a specific file.
Create and Edit New Source Files

You can also create and edit new source files directly in the Vitis IDE.

1. From the open project, right-click the src folder and select **New → File**.
   
   The New File dialog box is displayed as shown in the following figure.

   ![New File Dialog](image)

   - **Enter or select the parent folder:**
     - K-means/src

   - **File name:** mySourceFile.cpp

2. Select the folder in which to create the new file and enter a file name.

3. Click **Finish** to add the file to the project.

After adding source files to your project, you are ready to begin configuring, building, and running the application. To open a source file in the built-in text editor, expand the src folder in the Project Explorer and double-click on a specific file.
Working in the Project Editor View

Building the system requires compiling and linking both the host program and the FPGA binary (xclbin). Your defined application project includes a top-level system project, host processor project, hardware kernel project, and hw_link project. Both the host and kernel projects contain source code in the src folder, as imported or created in the project. Any of these projects can be opened in the Project Editor view, shown in the following figure, which gives a top-level view of the project and its various build configurations.

*Figure 93: Project Editor View*

Depending on the type of project you are viewing, system project, host, kernel, or link, the Project Editor provides the following details:

- General information about the project name
- Target platform
- Active build configuration
- Several configuration options related to the selected project

These include boot files for system project, debug options for the host or kernel projects, and a menu to select the report level of the hardware kernel project as discussed in Controlling Report Generation.
The bottom portion of the Project Editor view displays Application Projects contained in the top-level system project, as shown in the figure above, or displays the Hardware Functions that will be compiled in a hardware kernel project, or assigned to the binary container in the `hw_link` project to be built into the `xclbin`.

To specify a function to be compiled in a hardware kernel project, click the **Add Hardware Function** button in the upper right of the Hardware Functions pane. This opens the Add Hardware Functions dialog box displaying a list of functions defined in the source code of the current project, as shown below.

*Figure 94: Adding Hardware Functions to a Binary Container*

Select a function from the list to specify the hardware function, and click **OK**. The selected function becomes the target of the build process for the hardware kernel project and is also added to the device binary in the `hw_link` project.
Working in the Assistant View

The Assistant view provides a project tree to manage build configurations, run configurations, and set the attributes of these configurations. It is a companion view to the Explorer view and displays directly below it in the default Vitis IDE perspective. The following figure shows an example Assistant view and its tree structure.

*Figure 95: Assistant View*

The objects displayed in the Assistant view hierarchy include the top-level system project, host project, hardware kernel projects, and the `hw_link` project. For each of these projects the different build configurations are also displayed: the software emulation and hardware emulation build configurations, and the hardware build configuration. The build configurations define the build target as described in Build Targets, and specify options for the compilation and linking process.

**TIP:** The status of build configurations can be quickly determined from the icons displayed in the figure above:

- Emulation-SW builds are complete as indicated by the green check box.
Emulation-HW builds need to be updated for the hardware kernel and hw_link projects, as indicated by the yellow exclamation (!).

Hardware builds are not built.

When you select a build configuration, such as Emulation-HW build and click the Settings icon (⚙️), the Vitis Build Configuration Settings dialog box opens. You will use this Settings dialog box to configure the build process for the specific emulation or hardware target.

Within the hierarchy of these build configuration is the binary container (or .xclbin), the hardware function or functions, the run configuration, and any reports or summaries generated by the build or run process. When you select the hardware function for a specific build configuration and click the Settings icon, the Vitis Hardware Function Settings dialog box is displayed. You will use this dialog box to specify the number of compute units for each kernel, assign compute units to SLRs, and assign kernel ports to global memory.

After a specific build configuration has been built, launch configurations become available for the project. There are two types of launch configurations:

- Run configurations specify the profile used for running the compiled and linked application; it defines the environment and options for running the host application and kernel code. Use the Run command (🚀) on the toolbar menu to access run configurations. This opens the Vitis Run and Debug Configuration Settings dialog box, where you can configure the run before launching it.

- Debug configurations specify the profile for debugging the application. It launches the environment needed to interactively debug both the host and kernel code. You can access debug configurations through the Debug command (🐞) on the toolbar menu. Refer to Vitis IDE Debug Flow for more information.

**Figure 96: Assistant View Menu**

Within the Assistant view, the View menu includes options that affect what the Assistant view displays, or affect how the Assistant view interacts with other views. Open the View menu by left-clicking the menu command to display the following options:

- **Show Active Build Configurations Only**: When enabled, the Assistant view will only show the active build configuration for each project. This option can be useful to reduce the clutter in the Assistant view. To change the active configuration, select **Active build configuration** in the Project Editor view.
• **Link with Console:** When enabled, the build console in the Console view switches automatically to match the currently selected build configuration in the Assistant view. If not enabled, the build console does not automatically change to match the Assistant view.

• **Link with Guidance:** When enabled, the Guidance tab of the Console view automatically switches to match the current selection in the Assistant view.

For each of the build configurations, reports are generated during the build and run process and are displayed in the Assistant view. The different reports are grouped into Compile Summary, Link Summary, Package Summary, and Run Summary which can be viewed in the Vitis analyzer tool as described in Section VI: Using the Vitis Analyzer. You can right-click one of these summary reports in the Assistant view and select **Open in Vitis Analyzer** as shown below.

![Figure 97: Open in Vitis Analyzer](image)

### Output Directories from the Vitis IDE

The following example shows the directory structure automatically generated by the Vitis IDE for a sample application acceleration project which includes:

- Top-level system project
- Host application project
- Hardware kernel compilation project
- Hardware kernel linking project

The default directory structure of the Vitis IDE is similar but not identical to that created by the command-line flow.

```bash
### Vitis IDE Directory Structure
workspace
  >pathwayKernel - a hardware kernel project in the Vitis application acceleration development flow. The top-level project can contain multiple kernel projects.
  >Emulation-HW - the hardware emulation build folder.
    >build
      > The build folder for the compiled hardware kernel (.XO)
    >guidance.html
```
Section VII: Using the Vitis IDE

Chapter 43: Creating a Vitis IDE Project

>guidance.json
>guidance.pb
>makefile
>mmult-compile.cfg
>pathwayKernel_Emulation-HW.build.ui.log
>xcd.log

>Emulation-SW
>Files related to the Software Emulation Build
>pathwayKernel.prj - Hardware kernel project
>src - Hardware kernel source files
>kernel.cpp
>pathwayTest - The host application project

>Emulation-HW
>au250_image.jpg
>bd_d216_drr4_mem00_0_microblaze.mcs_bd.tcl
>bd_d216_drr4_mem00_0_microblaze.mcs.hwh
>bd_d216_drr4_mem01_0_microblaze.mcs_bd.tcl
>bd_d216_drr4_mem01_0_microblaze.mcs.hwh
>bd_d216_drr4_mem02_0_microblaze.mcs_bd.tcl
>bd_d216_drr4_mem02_0_microblaze.mcs.hwh
>bd_d216_drr4_mem03_0_microblaze.mcs_bd.tcl
>bd_d216_drr4_mem03_0_microblaze.mcs.hwh
>bd_d216_interconnect_DDR4_MEM00_0_bd.tcl
>bd_d216_interconnect_DDR4_MEM00_0.hwh
>bd_d216_interconnect_DDR4_MEM01_0_bd.tcl
>bd_d216_interconnect_DDR4_MEM01_0.hwh
>bd_d216_interconnect_DDR4_MEM02_0_bd.tcl
>bd_d216_interconnect_DDR4_MEM02_0.hwh
>bd_d216_interconnect_DDR4_MEM03_0_bd.tcl
>bd_d216_interconnect_DDR4_MEM03_0.hwh
>bd_d216_interconnect_PLRAM_MEM00_0_bd.tcl
>bd_d216_interconnect_PLRAM_MEM00_0.hwh
>bd_d216_interconnect_PLRAM_MEM01_0_bd.tcl
>bd_d216_interconnect_PLRAM_MEM01_0.hwh
>bd_d216_interconnect_PLRAM_MEM02_0_bd.tcl
>bd_d216_interconnect_PLRAM_MEM02_0.hwh
>bd_d216_interconnect_PLRAM_MEM03_0_bd.tcl
>bd_d216_interconnect_PLRAM_MEM03_0.hwh
>bd_d216_interconnect_S00_AXI_0_bd.tcl
>bd_d216_interconnect_S00_AXI_0.hwh
>bd_d216_interconnect_S01_AXI_0_bd.tcl
>bd_d216_interconnect_S01_AXI_0.hwh
>bd_d216_interconnect_S02_AXI_0_bd.tcl
>bd_d216_interconnect_S02_AXI_0.hwh
>bd_d216_interconnect_S03_AXI_0_bd.tcl
>bd_d216_interconnect_S03_AXI_0.hwh
>binary_container_1.xclbin
>board
>board files for the selected hardware platform
>dsa.xml
>emconfig.json
>emu
>dynamic_post_sys_link.tcl
>dynamic_pre_sys_link.tcl
>emu.bd
>emu.bxml
>emu.xml
>hdl
>emu_wrapper.v
>ip
>IP files from the selected platform
>ipshared
>
Section VII: Using the Vitis IDE

Chapter 43: Creating a Vitis IDE Project
Section VII: Using the Vitis IDE

Chapter 43: Creating a Vitis IDE Project
Section VII: Using the Vitis IDE
Chapter 43: Creating a Vitis IDE Project

>xo
>   >ip_repo
>   >mmult
>       >cpu_sources
>       >debug
>       >kernel.xml
>       >mmult.design.xml
>   >link.spr
>   >link.steps.log
>   >run_link
>   >sys_link
>   >vivado
>   >logs
>       >link
>       >reports
>           >link
>           >binary_container_1-link.cfg
>           >binary_container_1.mdb
>           >binary_container_1.xclbin
>           >binary_container_1.xclbin.info
>           >binary_container_1.xclbin.link_summary
>           >binary_container_1.xclbin.sh
>           >guidance.html
>           >guidance.json
>           >guidance.pb
>           >makefile
>           >pathwayTest_system_hw_link_Emulation-HW.build.ui.log
>           >xcd.log
Building the System

When building the system, it best practice to use the three available build targets described in Build Targets. Each build target is represented in a separate build configuration in the Assistant view. Work through these build configurations in the following order:

- **Emulation-SW**: Build for software emulation (sw_emu) to confirm the algorithm functionality of both the host program and kernel code working together.

- **Emulation-HW**: Build for hardware emulation (hw_emu) to compile the kernel into a hardware description language (HDL), confirm the correctness of the generated logic, and evaluate its simulated performance.

- **Hardware**: Perform a system hardware build (hw) to implement the application running on the target platform.

Before launching the build command, configure each of these build configurations to ensure it meets your needs. Select the specific build configuration, and click the Settings icon to open the Build Configuration Settings dialog box. For more information on using this dialog box, refer to Vitis Build Configuration Settings.

Beyond the build configuration settings, many of the settings that will affect your application are contained in the Hardware Function, accessed through the Vitis Hardware Function Settings dialog box. It is a good idea to review each of the Settings dialog boxes as discussed in Configuring the Vitis IDE.

From the Assistant view, with the various options of the build configuration specified, you can start the build process by selecting a build configuration and clicking on the Build button. The Vitis core development kit uses a two part build process that generates the FPGA binary (.xclbin) for the hardware kernels using the Vitis™ compiler v++ command, and compiles and links the host program code using the g++ compiler.

After the build process is complete, the Assistant view shows the specific build configuration with a green check mark to indicate it has been successfully built, as shown in the following figure. You can open any of the build reports, such as the Compile Summary in the hardware function, or the Link Summary in the binary container. Right-click the report in the Assistant view and select Open in Vitis Analyzer.
With the build complete, you can now run the application in the context provided by the specific build configuration. For instance, exercise a C-model of the host program and FPGA binary working together in the Emulation-SW build, or review the host program and the RTL kernel code in simulation in the Emulation-HW build, or run the application on the target platform in the Hardware build.

To run the application from within the Vitis IDE, select the build configuration, and click the Run button ( ) to launch the default run configuration. You can also right-click the build configuration and use the Run menu to select a specific run configuration, or edit a run configuration as described in Vitis Run and Debug Configuration Settings.

**TIP:** The Vitis IDE creates a folder named after the run configuration in the specific build configuration being run. For instance ./project/Emulation-HW/run_config. The output files and logs from the application run are written to this folder. All arguments passed to the host program should be written relative to this folder.

---

**Vitis IDE Guidance View**

After building or running a specific build configuration, the Guidance tab of the Console view displays a list of errors, warnings, and suggestions related to the build and run process. The Guidance view is automatically populated and displayed in the tabs located in the Console view. You can review the guidance messages to make any changes that might be needed in your code or build process.

After running hardware emulation, the Guidance view might look like the following figure.
Figure 99: Guidance for the Build

To simplify sorting through the Guidance view information, the Vitis IDE lets you search, and filter the Guidance view to locate specific guidance rule entries. You can collapse or expand the tree view, or even suppress the hierarchical tree representation and visualize a condensed representation of the guidance rules. Finally, you can select what is shown in the Guidance view by enabling or disabling the display of warnings, as well as rules that have been met, and also restrict the specific content based on the source of the messages such as build and emulation.

By default, the Guidance view shows all guidance information for the project selected in the drop down. To restrict the content to an individual build or run step, do the following:

1. Select Window → Preferences
2. Select the category Guidance.
3. Deselect Group guidance rule checks by project.

Working with Vivado Tools from the Vitis IDE

The Vitis core development kit calls the Vivado Design Suite during the linking process to automatically run RTL synthesis and implementation when generating the FPGA binary (.xclbin). You also have the option of launching the Vivado tool directly from within the Vitis IDE to interact with the project for synthesizing and implementing the FPGA binary. There are three commands to support interacting with the Vivado tool from the Vitis IDE, accessed through the Xilinx → Vivado Integration menu:
TIP: The *hw_link* project must be opened and be the current project in the IDE for these options to be available.

- **Open Vivado Project**: This automatically opens the Vivado project (.xpr) associated with the Hardware build configuration. In order for this feature to work, you must have previously completed the Hardware build so that a Vivado project exists for the build.

  Opening the Vivado project launches the Vivado IDE and opens the implementation design checkpoint (DCP) file to edit the project, to let you manage the results of synthesis and implementation more directly. You can then use the results of this effort for generating the FPGA binary by selecting **Import Design Checkpoint**.

- **Import Design Checkpoint**: Lets you specify a Vivado DCP file to use as the basis for the Hardware build, and for generating the FPGA binary.

- **Import Vivado Settings**: Lets you specify a configuration file used by the Vivado tools, as described in *[Vitis Compiler Configuration File](#)*, for use during the linking process.

Using the Vivado IDE in standalone mode enables the exploration of various synthesis and implementation options for further optimizing the kernel for performance and area. There are additional options available to let you interact with the FPGA build process. See *[Managing Vivado Synthesis and Implementation Results](#)* for more information.

**IMPORTANT!** The optimization switches applied in the standalone project are not automatically incorporated back into the Vitis IDE build configurations. You need to ensure that the various synthesis and implementation properties are specified for the build using the `v++ --config` file options. For more information, refer to *[Vitis Compiler Command](#)*.
Chapter 45

Vitis IDE Debug Flow

The Vitis™ IDE provides easy access to the debug capabilities. When performed manually, setting up an executable for debugging requires many steps. When you use the debug flow, these steps are handled automatically by the Vitis IDE.

**Note:** The debug flow in the Vitis IDE relies on shell scripts during debugging. This requires that the setup files, such as .bashrc or .cshrc, do not interfere with the environment setup, such as the `LD_LIBRARY_PATH`.

To prepare the executable for debugging, you must change the build configurations to enable **Host debug** and **Kernel debug**. Set these options in the Project Editor view in the Vitis IDE, as shown in the following figure. There are two check boxes provided in the Options section for the Active build configuration:

- **Host debug** enables debugging constructs in the host compilation, and is available for all build types.
- **Kernel debug** enables debugging of the kernels, but is only available in software and hardware emulation builds. To enable debug in hardware builds, use the Chipscope Debug settings as described in **Vitis Hardware Function Settings**.

These check boxes enable the `-g`, or `--debug` options in the g++ and Vitis compilers.

*Figure 100: Project Editor View Debug Options*

You can also enable the debug features from the Build Configuration Settings dialog box, as shown in **Vitis Build Configuration Settings**, by selecting the build configuration in the Assistant view and clicking the **Settings** button. Alternatively, you can double-click the build configuration. The same two check boxes are presented. While you can enable host debug on all targets, kernel debug is only supported for software emulation and hardware emulation build targets.
Running a GDB session from the Vitis IDE takes care of all the required setup. It automatically manages the environment setup for hardware or software emulation. It configures XRT to ensure debug support when the application is running, as described in xrt.ini File, and manages the different consoles required for the execution of the host code, the kernel code, and the debug server.

When running on an embedded platform, the Vitis IDE also configures and launches the QEMU system mode, the logic simulator for the PL kernel, and manages synchronization between them. For more information, refer to launch_emulator Utility.

After setting up the build configuration for debug, clean the build directory, and rebuild the application to ensure that the project is ready to run in the GDB debug environment.

To launch a debug session, select the build configuration in the Assistant view, and click Debug (avr). When launching the debug session in the Vitis IDE, the perspective switches to the Debug perspective, which is configured to present additional windows to manage the different debug views and source code windows. The following figure shows the Debug perspective.

*Figure 101: Debug Perspective*
After launching the debug environment, by default, the application is stopped at the beginning of the main function body in the host code. As with any GDB graphical front end, you can now set breakpoints and inspect variables in the host code. The Vitis IDE enables the same capabilities for the accelerated kernel implementation in a transparent way. For more information, refer to Debugging Applications and Kernels.

**Note:** In hardware emulation, because the C/C++/OpenCL™ kernel code is translated for efficient implementation, breakpoints cannot be placed on all statements. Mostly, untouched loops and functions are available for breakpoints, and only preserved variables can be accessed.

---

**Using the Standalone Debug Flow**

The Vitis IDE lets you open the debug tool for projects that have been built using the command line flow.

**Launching Standalone Debug for Embedded Platforms**

The standalone debug flow supports both the embedded processor application acceleration flow (embedded_accel) or the embedded processor software development flow (embedded). For embedded platforms, the application is running on the Arm processor of the device, the files that are required to boot the system, and load the application and kernel, are on a remote system, but the debug tools are running on the local system, and the data and reports generated need to be moved from the embedded system to the local system. The process for debugging in that environment requires more setup and configuration.

Running standalone debug in the Vitis IDE for the embedded_accel flow is a two-step process.

1. You must first launch the QEMU emulator environment using the `launch_sw_emu.sh` or the `launch_hw_emu.sh` script, that is generated during the --package process.
2. Then you must launch the Vitis IDE in standalone debug mode using the -debug option.

To run standalone debug in the Vitis IDE for the embedded flow, you must first launch the QEMU emulator environment using the `launch_hw_emu.sh` script, that is generated during the --package process.

The files required for emulation of the system are also defined by the --package command. This means that launching the standalone debug process for embedded platforms is reliant on the output of the package process, including the emulation script. An example command to launch the emulation environment would include the following.

```
launch_hw_emu.sh -pid-file emulation.pid -no-reboot -forward-port 1440 1534
\-enable-debug
```

Where:
• **enable-debug**: Opens two different command shells to launch QEMU and XSIM, and enables the GDB connection to the QEMU shell.

• **forward-port**: Forwards the TCP port from target to host for connecting to the QEMU shell. The QEMU port default is 1440. You can change it if necessary, for example, to 1446, but you must specify it for both the `launch_emulation` command or script and in the `vitis -debug` command line.

• **no-reboot**: Exit the QEMU environment when done.

• **pid-file**: Write the process ID to the specified file, used to kill the process, if necessary.

For hardware emulation, this launches two terminal windows running the QEMU system mode, and the Vivado simulator for simulating the PL kernel.

After the terminals and emulation are up and running, you can launch the Vitis IDE in standalone debug mode in a separate command shell:

```
vitis -debug -flow embedded_accel -target hw_emu -exe vadd.elf \ 
-program-args vadd.xclbin -kernels vadd
```

Where:

• **vitis -debug**: Launches the Vitis IDE in standalone debug mode.

• **flow embedded_accel**: Specifies the application acceleration flow on an embedded processor platform.

• **target hw_emu**: Indicates the target build being debugged.

• **exe vadd.elf**: Indicates the executable application to run and debug.

• **program-args vadd.xclbin**: Specifies the .xclbin file to be loaded as an argument to the executable.

There are more options that can be specified as described in **vitis -debug Command Line**, and these options might be needed depending on the configuration of your application and build environment.

The default for embedded systems searches for the executable and the .xclbin file, and any other required input files, on the `/mnt` folder of the emulation environment, or the embedded system. You can change this by specifying the **-target-work-dir** when launching the tool. This launches the Vitis IDE with the Debug perspective enabled, running a debug configuration for the specified executable application and kernel code. From this point you can do all the debug activities like step in/step over/viewing variables/adding break points within the GUI-based debug environment.
Launching Standalone Debug for Data Center Platforms

Launching standalone debug for Data Center applications is a bit simpler. In this case, you need to identify the build target and the executable to run and debug. The Data Center platforms do not require an emulation environment.

The following example launches the Vitis standalone debug for the data_center flow targeting the software emulation build. It specifies the executable, `host.exe`, which is looked for in the current directory, and specifies the kernel to debug.

```
vitis -debug -flow data_center -target sw_emu -exe host.exe -kernels krnl_vadd
```

By default, the standalone debug flow looks in the current directory for specified files and to write results. You can specify the `-work-dir` option to indicate a different working directory from the default. This might be necessary when the .xclbin file is built in a different directory.

This launches the Vitis IDE with the Debug perspective enabled, letting you perform debug activities like step in/step over/viewing variables/adding break points within the GUI-based debug environment.

---

**vitis -debug Command Line**

**Command Line Usage**

The Vitis software platform standalone debug feature lets you launch the Vitis IDE for debugging an existing command line project. In the following sections, an explanation of each of the command line options is described with examples of launching the standalone debug environment for different platforms and target builds.

- **-debug**

  ```
vitis -debug
  ```

  Launches the Vitis IDE in standalone debug mode.

- **-flow**

  ```
  -flow [ data_center | embedded_accel | embedded ]
  ```

  Specifies the type of application project being debugged. This configures the Vitis IDE for debugging Data Center applications running on Alveo cards; for example, application acceleration projects running on embedded platforms, such as the zcu104_base platform, or embedded software projects.
IMPORTANT! For embedded and embedded_accel flows, you must launch the QEMU system emulator using the launch_hw_emu.sh or launch_sw_emu.sh script generated during the --package step as described in Packaging for Embedded Platforms, or using the launch_emulator command.

-workspace

-exe

-target

-program-args

Specifies the Vitis IDE workspace to use when opening the application project in debug mode. If this option is not specified, the tool will create a directory named workspace in the current working directory. If a directory named workspace already exists, the tool will use that as the workspace.

Specifies the file name and path to the application (host) executable.

For example:

vitis -debug -exe .host.elf

Specifies the build target to use for debugging.

TIP: This only applies for data_center and embedded_accel flows.

For example:

vitis -debug -target hw_emu

Specifies the command line arguments to be passed to the host application at runtime. If not specified, the tool will pass the .xclbin as a program argument when the data_center or embedded_accel flows are selected.

For example:

vitis -debug -program-args ./xclbin in.dat
-kernels

-kernels <list of kernels>

Specifies the list of kernels to debug. Multiple kernel names can be specified, separated by commas. Listed kernels are defined as function-level breakpoints, so the debugger stops when kernel execution starts. If a kernel is not specified, no function-level debugging is provided.

This is valid only for data_center flows and is not supported for embedded or embedded_accel flows.

For example:

vitis -debug -kernels mmult madd

-work-dir

-work-dir <path_to_working_directory>

Specifies the working directory to save generated output files and reports. This is valid for data_center and embedded_accel flows.

For the data_center flow, this is the directory where the specified .exe will be launched. For embedded_accel flow, the launch directory will be defined by -target-work-dir.

TIP: If not specified, the current working directory is used as the working directory.

-target-work-dir

-target-work-dir <Target working directory>

This is the directory on the target board OS, and the QEMU environment, where the executable will be launched. This is valid for embedded_accel and also for embedded flows using a Linux OS.

TIP: If not specified, the target working directory is /mnt.

-xrt-ini

-xrt-ini <path_to_xrt.ini>

Specifies the location of the xrt.ini file. Valid for data_center and embedded_accel flows.

If the location is specified, it will be looked for in the same directory as the application .exe or in the working directory.
**-os**

```bash
-os [ linux | baremetal ]
```

Specifies the OS running on the target board. This is valid for the embedded flows.

**-host**

```bash
-host <host_name or ip_address>
```

Specifies the name or IP address of the host system where the TCF agent or hw_server is running. Valid for embedded_accel and embedded flows. If not specified, it the default host name is localhost for bare metal, and the default IP address is 192.168.0.1 for Linux target OS.

**-port**

```bash
-port <port number>
```

Port for TCF agent running on target Linux, or the port for hw_server running on local host for bare metal target. If not specified, the port is 1534 for tcf-agent and 3121 for hw_server.

**-launch-script**

```bash
-launch-script <path_to_tcl_script>
```

Specify a Tcl script to be sourced before attaching the application to the debugger. This is valid only for embedded flow with bare metal OS. The Tcl script can contain commands to initialize the board, download the application, add breakpoints and make the target ready for the debugger to attach.
Configuring the Vitis IDE

From the Assistant view, use the Settings button (⚙️) to configure a selected project or configuration object. For more information, refer to the following topics:

- Vitis Project Settings
- Vitis Build Configuration Settings
- Vitis Hardware Function Settings
- Vitis Binary Container Settings
- Vitis Toolchain Settings
- Vitis Run and Debug Configuration Settings

Vitis Project Settings

To edit the Vitis™ project settings, select the top-level system project in the Assistant view and click the Settings button (⚙️) to bring up the Project Settings dialog box. This dialog box lets you specify both linking and compile options for the Vitis compiler v++ command to let you customize the project build process.
Figure 102: Project Settings

- **Project name**: Name of the project. Click the link to open the Properties dialog box for the project.
- **Platform**: Target platform for this project. Click the link to open the Platform Description dialog box. Click **Browse** to change the platform.
- **Runtime**: Displays the runtime used in this project.
- **Options**: Options are displayed when the Host application has been written using the XRT native C++ API.
- **Applications**: Lists the different sub-projects included with the system project, and also displays the current build target.
Vitis Build Configuration Settings

To edit the settings for any of the build configurations under a project, select the build configuration in the Assistant view and click the Settings button (⚙️) to bring up the Build Configuration Settings dialog box. The specific features of the dialog box vary depending on the type of project and the build target you have selected. In this dialog box, you can enable host and kernel debug, specify the level of information to report during the build process, and specify the level of optimization for the hardware build.

**Figure 103: Build Configuration Settings**

The various options displayed on this dialog box include:
- **Target:** The build configuration target as described in Build Targets.

- **Host debug:** Select to enable debug of the host code.

- **Kernel debug:** Select to enable debug of the kernel code.

- **Report level:** Specify what report level to generate as described in Controlling Report Generation.

- **Hardware optimization:** Specify how much effort to use on optimizing the hardware. Hardware optimization is a compute intensive task. Higher levels of optimization might result in more optimal hardware but with increased build time. This option is only available in the Build Configuration System.

The Build Configuration dialog box also contains links to Edit Toolchain Compiler Settings and Edit Toolchain Linker Settings. These links provide access to all the settings in the standard Eclipse environment, and can be used to configure the G++ compiler, the V++ compiler, and the emconfigutil command as described in Vitis Toolchain Settings.

---

**Vitis Hardware Function Settings**

To edit the settings for the hardware functions, expand the build target for the kernels project in the Assistant view, select the hardware function, and click the **Settings** button (⚙️). This displays the Hardware Function Settings dialog box as shown in the following figure.

---

**TIP:** Notice in the figure below that the dct kernel (or hardware function) is selected for the Emulation-HW build target of the hardware kernels project (dct_project_kernels).
This dialog box lets you set options related to the hardware function, and the v++ compilation process for the selected build target. Specific options include:

- **Stall Profiling:** Enables the `--profile.stall` option for the kernel as explained in `--profile Options`.

- **Max memory ports:** For OpenCL kernels, when enabled, generates a separate physical memory interface (`m_axi`) for every global memory buffer declared in the kernel function signature. If not enabled, a single physical memory interface is created for the memory mapped kernel ports.

- **Port data width:** For OpenCL kernels, specify the width of the data port.

- **Extra source files:** Define any additional source files required by this hardware function, such as input data files.
• **V++ compiler options:** Specify Vitis compiler options for the selected hardware function.

• **Supported targets:** Specify which of the three build targets you are defining in the Hardware Function Settings dialog box. You can select one or all build targets.

• **Kernel Arguments:** Displays the names and attributes of the arguments of the hardware kernel.

• **V++ Compiler Command Line:** Displays the current v++ command line with any compilation options you have specified.

**TIP:** The settings specified by the Hardware Function Settings dialog box are written to a configuration file used by the Vitis compiler with the `--config` option as described in Vitis Compiler Configuration File. The configuration file is a link; when you place your mouse over the link, it displays the contents of the configuration file.

---

**Vitis Binary Container Settings**

To edit the settings for a binary container, expand the build target for the `hw_link` project in the Assistant view, select the `binary_container`, and click the **Settings** button (⚙️). This displays the Binary Container Settings dialog box as shown in the following figure.

**TIP:** The options displayed in the Binary Container Settings dialog box depend on the specific build target selected, and will vary between software emulation, hardware emulation, and hardware.
This dialog box lets you specify a new name for the binary container, and to specify link options for the `v++` command. Specific options include:

- **Name**: Specify the name of the binary container.
- **Trace Memory**: Specify the type and amount of memory to use for storing event trace data for profiling the application. This relates to the `--trace_memory` option of the Vitis compiler.
- **V++ Linker Options**: Enter link options for the selected binary container, in the specified build configuration. For more information on the available options, refer to [Vitis Compiler Command](#).
- **Compute Units**: Specify the number of kernels to add to the device binary as explained in `--connectivity Options`. The field can be edited, and when you have entered the value, additional kernels will be displayed in the dialog box.
- **Memory**: Specify global memory assignments for each port of a compute unit as discussed in [Mapping Kernel Ports to Memory](#).
- **SLR**: Define the SLR placement for each compute unit of the kernel as discussed in Assigning Compute Units to SLRs.
- **ChipScope Debug**: Add monitors to capture hardware trace debug information. This specifies the `--debug.chipscope` option.
• **Protocol Checker:** Add AXI Protocol Checker to your design. This relates to the `-debug.protocol` option.

• **Data Transfer:** Add performance monitors to capture information related to data transferred between compute unit and global memory. Captured data includes counters, trace, or both. This relates to the `--profile.data` option, and to `xrt.ini` file settings as explained in [*Enabling Profiling in Your Application*](#).

• **Execute Profiling:** Add an accelerator monitor to capture the start and end of compute unit executions. This relates to the `--profile.exec` option.

• **Stall Profiling:** Add an accelerator monitor with functionality to capture stalls in the flow of data inside a kernel, between two kernels, or between the kernel and external memory. This relates to the `--profile.stall` option.

• **V++ Linker Command Line:** Displays the current `v++` command line with any link options you have specified.

**TIP:** The settings specified by the Binary Container Settings dialog box are written to a configuration file used by the Vitis compiler with the `--config` option as described in [*Vitis Compiler Configuration File*](#). The configuration file is a link; when you place your mouse over the link, it displays the contents of the configuration file.

---

**Vitis Toolchain Settings**

The toolchain settings provide a standard Eclipse-based view of the project, providing all options for the C/C++ build in the Vitis IDE.

From the Build Configuration Settings dialog box, click **Edit Toolchain Compiler Settings** or **Edit Toolchain Linker Settings** to bring up the compiler and linker Settings dialog box containing all of the C/C++ build settings. This dialog box lets you set standard C++ paths, include paths, libraries, project wide defines, and host defines.
The contents of the Toolchain Settings dialog box depends on the specific Build Configuration dialog box you launched it from. You can launch it from the host application project, the hardware kernels project, or the hw_link project. The specific settings available include the following:

- **emconfigutil**: Specify the command line options for `emconfigutil Utility`. See [emconfigutil Settings](#).

- **GCC Host Compiler**: Specify `g++` linker arguments that must be passed during the host compilation process. See [G++ Host Compiler and Linker Settings](#).

- **GCC Host Linker**: Specify `g++` linker arguments that must be passed during the host linking process. See [G++ Host Compiler and Linker Settings](#).
• **V++ Kernel Compiler**: Specify the `v++` command and any additional options that must be passed when calling the `v++` command for the kernel compilation process. See *V++ Compiler and Linker Settings*.

• **V++ Kernel Linker**: Specify the `v++` command and any additional options to be passed when calling the `v++` command for the kernel linking process. See *V++ Compiler and Linker Settings*.

### G++ Host Compiler and Linker Settings

*Figure 107: GCC Compiler and Linker Settings*

You can open the **GCC Compiler and Linker Settings** dialog box from the Build Configuration Settings dialog box of a host application project. The arguments for the `g++` compiler used by the Vitis core development kit can be accessed under the GCC Host Compiler section of the Toolchain Settings.
- **Dialect**: Specify the command options that select the C++ language standard to use. Standard dialect options include C++ 98, C++ 2011, and C++ 2014 (1Y).

- **Preprocessor**: Specify preprocessor arguments to the host compiler such as symbol definitions. The default symbols already defined include the platform so that the host code can check for the specific platform.

- **Includes**: Specify the include paths and include files.

- **Optimization**: Specify the compiler optimization flags and other optimization settings.

- **Debugging**: Specify the debug level and other debugging flags.

- **Warnings**: Specify options related to compiler warnings.

- **Miscellaneous**: Specify any other flags that are passed to the g++ compiler.

### GCC Linker Options

The linker arguments for the Vitis technology G++ Host Linker are provided through the options available here. Specific sections include general options, libraries and library paths, miscellaneous linker options, and shared libraries.

### V++ Compiler and Linker Settings

*Figure 108: V++ Compiler Settings*

You can open the **V++ Compiler Settings** dialog box from a hardware kernel project Build Configuration Settings dialog box. The V++ Kernel Compiler settings shows the `v++` command used during compilation, and any additional options that must be passed when calling the `v++` command for the kernel compilation process. The `v++` command options can be symbols, include paths, or miscellaneous valid options.

- **Symbols**: Click **Symbols** under Vitis compiler to define any symbols that are passed with the `-D` option when calling the `v++` command.
• **Includes**: To add include paths to the Vitis compiler, select **Includes** and click the **Add** button.

• **Miscellaneous**: Vitis specific settings, such as the Vitis compiler and linker flags, which are not part of the standard C/C++ tool chain, can be added as flags in the Miscellaneous section. For more information on the available compiler options, refer to **Vitis Compiler Command**.

*Figure 109: V++ Linker Settings*

You can open the **V++ Linker Settings** dialog box from a **hw_link** project Build Configuration Settings dialog box. The **V++ Linker Settings** shows the `v++` command used during linking, as well as any additional options to be passed when calling the `v++` command for the kernel linking process.

Any additional options that need to be passed to the Vitis compiler can be added as flags in the Miscellaneous section. For more information, refer to **Vitis Compiler Command** for the available options in the linking process.

**emconfigutil Settings**

Select the **emconfigutil** command options in the **GCC Compiler and Linker Settings** dialog box, which you can open from the Build Configuration Settings dialog box of a host application project. The **Command** field specifies the `emconfigutil` command that will launch the Vitis IDE, as described in **emconfigutil Utility**. You can also specify any options for the command line as needed for your project.

The Vitis IDE creates the `emconfig.json` file by running the specified `emconfigutil` command before launching an emulation run configuration.
Vitis Run and Debug Configuration Settings

Launching the compiled, linked, and packaged application, to run or to debug within the Vitis IDE requires the use of the Launch Configuration dialog box, as shown in the figure below. When the build process has completed, the tool enables both the Run button and the Debug button in the Assistant view to let you specify a Launch Configuration to use.

**TIP:** The Launch Configuration is the same for both running and debugging the application. There are differences in the steps that the tool follows when running the application or launching the debugger, but you can use the same configuration for both purposes.

To edit the settings for a launch configuration, select a build target and click the Run button to open the Run Configurations dialog box. The Run Configuration dialog box, as shown below, lets you specify debug options, enable profiling of the running application, and specify the types of profiling data to collect.
Create, manage, and run configurations

Debug a system using Vitis Debugger.

**TIP:** The options displayed in the Launch Configuration dialog box vary for Data Center and Embedded Processor platforms, and also for the software and hardware emulation builds, and the hardware build.
- **Name:** Specify the name of the run configuration. The Vitis IDE creates a folder named after the run configuration in the specific build configuration being run. For instance `./project/Emulation-HW/run_config`. The output files and logs from the application run are written to this folder. All arguments passed to the host program should be written relative to this folder.

- **Project:** Display the current project, but can be changed to other open projects.

- **Build Configuration:** Select the build target that the launch configuration applies to, or it applies to the active build configuration.

- **Disable Build Before Launch:** When enabled this check box prevents the tool from rebuilding the project before running or debugging it.

- **Target:** Specify the run or debug target for the configuration. Note that emulation builds targets the Linux TCF agent, while the hardware build requires the `hw_server`, as described in Debugging Applications and Kernels.

- **Remote Working Directory:** For embedded processor systems, specifies the mount disk for the QEMU environment, or for the physical device.

- **Program Arguments:** Display the Programs Argument dialog box. This lets you specify command line arguments for your application if any are needed. Enabling the **Automatically update arguments** check box allows the tool to automatically specify the `xclbin` file as an argument for the application. The `xclbin` file is appended at the end of the command line, after any other specified arguments.

  **Note:** Program arguments should be specified relative to the run configuration folder that is created during the run. You can refer to the Vitis log window in the IDE to review a copy of the command line used to launch the application.

- **Override Application Options:** Generate an options file for the `launch_emulator` command, that you can manually edit to customize as needed for your purposes. Click the **Generate** button to create a new `launch_options.cfg` file, or use the **Browse** button to locate an existing one.

- **Xilinx Runtime Profiling:** This specifies the profiling and event trace features that are enabled during the application run. The specified options are stored in a configuration file and written to the `xrt.ini` file to use while running the application. Click the **Edit** button to open the Xilinx Runtime Profiling dialog box as shown below.
• **Runtime Profiling:** Specify the type of profiling you want enabled for the run. You can specify a form of profiling, or profiling and timeline capture as described in Enabling Profiling in Your Application.

  **TIP:** For hardware builds you can also enable Power Profiling for the accelerator card.

• **Device Profiling:** Specify level of profiling performed on the hardware kernels. This option relates to the data_transfer_trace as described in xrt.ini File. Coarse device profiling shows data transfer activity of the CU. Fine device profiling shows all AXI-level transactions on the ports.

• **Collect Stall Trace:** Indicate the capture of stall data for a variety of conditions, as described in --profile Options, and xrt.ini File.

• **Device Trace Offload:** Specify the amount of global memory to allocate to the capture of trace data, and also to enable continuous trace offload at the specified interval to prevent the loss of application timeline data in the case of an interrupted process, as described in Enabling Profiling in Your Application.
The Launch Configuration dialog box has additional tabs to help configure the runtime environment for your application. The three tabs are:

- **Target Setup**: Mainly for embedded platforms with a bare-metal application. It provides options to initialize, manage, and reset the board, device, and program.
- **Environment**: Lets you set up and manage environment variables needed for the Vitis IDE.
- **Common**: This tab is inherited from Eclipse. These are settings and feature common to the Eclipse environment.
Project Export and Import

The Vitis™ IDE provides a simplified method for exporting or importing one or more Vitis IDE projects within your workspace, or import from GIT repositories. You can optionally include associated project build folders.

Export a Vitis Project

When exporting a project, the project is archived in a zip file with all the relevant files needed to import to another workspace.

1. To export a project, select File → Export from the main menu.

   The Export Vitis Projects page opens, where you select the project or projects in the current workspace to export as shown in the following figure.

   ![Export Vitis Projects dialog](image)

   - Archive File: `vitis_export_archive.ide.zip`  
   - Directory: `/tmp`  
   - Include build folders

2. To change the name for the archive, edit the Archive File field.
3. To include the current build configurations, enable **Include build folders** at the bottom of the window.

**TIP:** This can significantly increase the size of the archive, but might be necessary in some cases.

4. To create the archive with your selected files, click **OK** to create the archive.

The selected Vitis IDE projects will be archived in the specified file and location, and can be imported into the Vitis IDE under a different workspace, on a different computer, by a different user.

---

**Import a Vitis Project**

1. To import a project, select **File → Import** from the top menu.

   This opens the Import Projects page to select the import file type. There are two types of files you can select to import:

   - **Vitis project exported zip files:** Lets you import projects previously exported from the Vitis IDE as discussed in Export a Vitis Project.

   - **Eclipse workspace or zip file:** Lets you import projects from another Vitis IDE workspace.

   - **Import projects from Git:** Lets you import projects from either a local previously cloned Git repository, or from a specified Git URL, as described in the next topic.

2. The following figure shows the page that is opened when you select **Eclipse workspace or zip file** and click **Next**.
3. For Select root directory, point to a workspace for the Vitis IDE, and specify the following options as needed:
   - **Search for nested projects**: Looks for projects inside other projects in the workspace.
   - **Copy projects into workspace**: Creates a physical copy of the project in the current open workspace.
   - **Close newly created imported projects upon completion**: Closes the projects in the open workspace after they are created.

4. Click **Finish** to import the projects into the open workspace in the Vitis IDE.

---

**Import Projects from Git**

1. From the Import Projects wizard, select the **Import projects from Git** option, and select **Next** to proceed.

   This opens the Select Repository Source page. There are two types of repositories you can select:
   - **Existing local repository**: Selects an existing Git repository that has already been cloned locally. When you select this option, the Select a Git Repository page displays the list of currently cloned local repositories found by the Vitis IDE. Click **Next** to continue.

   **TIP**: You can create a local Git repository by cloning the URL as described below, or by using the `git clone <url>` command from the Linux shell.
• **Clone URL**: Lets you specify a Git URL to clone to the specified location. When you select this option the Source Git Repository page of the Import Projects wizard is displayed as shown below.

![Source Git Repository Page](image)

1. On the Source Git Repository page, specify the following and click **Next** to continue:
   - **Location**: Specify the URL for the repository. The host and repository path are extracted from the provided URL. Some URLs that might be of interest include:
     - Vitis Acceleration Examples: [https://github.com/Xilinx/Vitis_Accel_Examples](https://github.com/Xilinx/Vitis_Accel_Examples)
   - **Connection**: Specifies the connection protocol used to connect. Use these fields to customize the connection if necessary.
   - **Authentication**: Specifies the User ID and Password to access the repository if one is required.
2. In the Branch Selection page, you can select one or more branches to clone. Click Next to proceed.

3. In the Local Destination page, specify the **Destination Directory** where the repository will be cloned. Click Next to proceed.

2. After opening the local repository or cloning the URL to create a new local repository, the Select a wizard to import projects page of the Import Project wizard is given. As shown below, this page lets you import an Eclipse project, import a project using the New Project wizard, or import a general project.

3. Select the method for importing the project and click Next to continue. Depending on which method you chose, you will be directed to the New Project wizard as described in **Creating a Vitis IDE Project**, or you will be guided through the process of importing an Eclipse project or general project.
Getting Started with Examples

The Vitis™ core development kit is provided with example designs. These examples can:

- Be a useful learning tool for both the Vitis IDE and compilation flows such as makefile flows.
- Help you quickly get started in creating a new application project.
- Demonstrate useful coding styles.
- Highlight important optimization techniques.

Every target platform provided within the Vitis IDE contains sample designs to get you started, and are accessible through the project creation flow as described in Create an Application Project.

A limited number of sample designs are available in the `<vitis_root>/samples` folder, and many examples are also available for download from the Xilinx® GitHub repository. Each of these designs is provided with a Makefile, so you can build, emulate, and run the code entirely on the command line if preferred.

Installing Examples and Libraries

You can download and install sample applications from the Templates page when working through the New Application Project wizard, or from within an existing project by selecting Xilinx → Examples. This displays the Vitis IDE Examples dialog box as shown in the following figure.
The left side of the dialog box shows Vitis IDE Examples Repository, and has a download command for each category. The right side of the dialog box shows the directory to where the examples downloaded and the URL from where the examples are downloaded. Click **Download** next to Vitis IDE Examples to download the examples and populate the dialog box.

The command menu at the bottom left of the Vitis IDE Examples dialog box provides two commands to manage the repository of examples:

- **Refresh**: Refreshes the list of downloaded examples to download any updates from the Vitis Examples GitHub repository.
- **Reset**: Deletes the downloaded examples from the .Xilinx folder.

**TIP:** Corporate firewalls can restrict outbound connections. Specific proxy settings might be necessary.

You can also download Vitis Accelerated Libraries from the **New Application Project** wizard, or by selecting the **Xilinx → Libraries** menu command. For more information on the available libraries and their uses, refer to **Vitis Libraries**.
Using Local Copies

While you must download the examples to add templates when you create new projects, the Vitis IDE always downloads the examples into your local .Xilinx/vitis/<version> folder:

- **On Windows**: C:\Users\<user_name>\Xilinx\vitis\<version>
- **On Linux**: ~/.Xilinx/vitis/<version>

The download directory cannot be changed from the Vitis IDE Examples dialog box. You might want to download the example files to a different location from the .Xilinx folder. To perform this, use the git command from a command shell to specify a new destination folder for the downloaded examples:

```
git clone https://github.com/Xilinx/Vitis_Examples <workspace>/examples
```

When you clone the examples using the git command as shown above, you can use the example files as a resource for application and kernel code to use in your own projects. However, many of the files use include statements to include other example files that are managed in the makefiles of the various examples. These include files are automatically populated into the src folder of a project when the template is added through the New Vitis Project wizard. To make the files local, locate the files and manually make them local to your project.
You can find the needed files by searching for the file from the location of the cloned repository. For example, you can run the following command from the examples folder to find the xcl2.hpp file needed for the vadd example.

```
find -name xcl2.hpp
```
Chapter 49

RTL Kernel Wizard

The RTL kernel wizard automates some of the steps you need to take to ensure that the RTL IP is packaged into a kernel object (XO) file that can be used by the Vitis™ compiler. The RTL Kernel wizard:

• Steps you through the process of specifying the interface requirements for your RTL kernel, and generates a top-level RTL wrapper based on the provided information.

• Automatically generates an AXI4-Lite interface module including the control logic and register file, included in the top level wrapper.

• Includes an example kernel IP module in the top-level wrapper that you can replace with your own RTL IP design, after ensuring correct connectivity between your RTL IP and the wrapper.

• Automatically generates a kernel.xml file to match the kernel specification from the wizard.

• Generates a simple simulation test bench for the generated RTL kernel wrapper.

• Generates an example host program to run and debug the RTL kernel.

The RTL Kernel wizard can be accessed from the Vitis IDE, or from the Vivado® IP catalog. In either case it creates a Vivado project containing an example design to act as a template for defining your own RTL kernel.

The example design consists of a simple RTL IP adder, called VADD, that you can use to guide you through the process of mapping your own RTL IP into the generated top-level wrapper. The connections include clock(s), reset(s), s_axilite control interface, m_axi interfaces, and optionally axis streaming interfaces.

The Wizard also generates a simple test bench for the generated RTL kernel wrapper, and a sample host code to exercise the example RTL kernel. This example test bench and host code must be modified to test the your RTL IP design accordingly.

Launch the RTL Kernel Wizard

The RTL Kernel Wizard can be launched from the Vitis IDE, or from the Vivado IDE.

TIP: Running the wizard from the Vitis IDE automatically imports the generated RTL kernel, and example host code, into the current application project when the process is complete.
To launch the RTL Kernel Wizard from within the Vitis IDE, select the Xilinx→RTL Kernel Wizard menu item from an open application project. For details on working with the GUI, refer to Section VII: Using the Vitis IDE.

To launch the RTL Kernel Wizard from the Vivado IDE:

1. Create a new Vivado project, select the target platform when choosing a board for the project.
2. In the Flow Navigator, click the IP catalog command.
3. Type RTL Kernel in the IP catalog search box.
4. Double-click RTL Kernel Wizard to launch the wizard.

Using the RTL Kernel Wizard

The RTL Kernel wizard is organized into multiple pages that break down the process of defining an RTL kernel. The pages of the wizard include:

1. General Settings
2. Scalars
3. Global Memory
4. Streaming Interfaces
5. Summary

To navigate between pages, click Next and Back as needed.

To finalize the kernel and build a project based on the kernel specification, click OK on the Summary page.

General Settings

The following figure shows the three settings in the General Settings page.
The following are three settings in the General Settings page.

**Kernel Identification**

- **Kernel name**: The kernel name. This will be the name of the IP, top-level module name, kernel, and C/C++ functional model. This identifier shall conform to C and Verilog identifier naming rules. It must also conform to Vivado IP integrator naming rules, which prohibits underscores except when placed in between alphanumeric characters.

- **Kernel vendor**: The name of the vendor. Used in the Vendor/Library/Name/Version (VLNV) format described in the *Vivado Design Suite User Guide: Designing with IP* (UG896).

- **Kernel library**: The name of the library. Used in the VLNV. Must conform to the same identifier rules.

**Kernel options**

- **Kernel type**: The RTL Kernel wizard currently supports two types of kernels: RTL, and Block Design.
• **RTL:** The RTL type kernel consists of a Verilog RTL top-level module with a Verilog control register module and a Verilog kernel example inside the top-level module.

• **Block Design:** The block design type kernel also delivers a Verilog top-level module, but instead it instantiates an IP integrator block diagram inside of the top-level. The block design consists of a MicroBlaze™ subsystem that uses a block RAM exchange memory to emulate the control registers. Example MicroBlaze software is delivered with the project to demonstrate using the MicroBlaze to control the kernel.

• **Kernel control interface:** There are three types of control interfaces available for the RTL kernel. `ap_ctrl_hs`, `ap_ctrl_chain`, and `ap_ctrl_none`. This defines the `hwControlProtocol` for the `<kernel>` tag as described in RTL Kernel XML File.

### Clock and Reset options

• **Number of clocks:** Sets the number of clocks used by the kernel. Every RTL kernel has one primary clock called `ap_clk` and an optional reset called `ap_rst_n`. All AXI interfaces on the kernel are driven with this clock.

When setting **Number of clocks** to 2, a secondary clock and optional reset are provided to be used by the kernel internally. The secondary clock and reset are called `ap_clk_2` and `ap_rst_n_2`. This secondary clock supports independent frequency scaling and is independent from the primary clock. The secondary clock is useful if the kernel clock needs to run at a faster or slower rate than the AXI4 interfaces, which must be clocked on the primary clock.

**IMPORTANT!** When designing with multiple clocks, proper clock domain crossing techniques must be used to ensure data integrity across all clock frequency scenarios. Refer to UltraFast Design Methodology Guide for Xilinx FPGAs and SoCs (UG949) for more information.

• **Has reset:** Specifies whether to include a top-level reset input port to the kernel. Omitting a reset can be useful to improve routing congestion of large designs. Any registers that would normally have a reset in the design should have proper initial values to ensure correctness. If enabled, there is a reset port included with each clock. Block Design type kernels must have a reset input.

### Scalars

Scalar arguments are used to pass control type information to the kernels. Scalar arguments cannot be read back from the host. For each argument that is specified, a corresponding register is created to facilitate passing the argument from software to hardware. See the following figure.
**Number of scalar kernel input arguments:** Specifies the number of scalar input arguments to pass to the kernel. For each number specified, a table row is generated that allows customization of the argument name and argument type. There is no required minimum number of scalars and the maximum allowed by the wizard is 64.

The following is the scalar input argument definition:

- **Argument name:** The argument name is used in the generated Verilog control register module as an output signal. Each argument is assigned an ID value. This ID value is used to access the argument from the host software. The ID value assignments can be found on the summary page of this wizard. To ensure maximum compatibility, the argument name follows the same identifier rules as the kernel name.

- **Argument type:** Specifies the data type, and hence bit-width, of the argument. This affects the register width in the generated RTL kernel module. The data types available are limited to the ones specified by the OpenCL C Specification Version 2.0 in "6.1.1 Built-in Scalar Data Types" section. The specification provides the associated bit-widths for each data type. The RTL wizard reserves 64 bits for all scalars in the register map regardless of their argument type. If the argument type is 32 bits or less, the RTL Wizard sets the upper 32 bits (of the 64 bits allocated) as a reserved address location. Data types that represent a bit width greater than 32 bits require two write operations to the control registers.
Global Memory

Global memory is accessed by the kernel through AXI4 master interfaces. Each AXI4 interface operates independently of each other, and each AXI4 interface can be connected to one or more memory controllers to off-chip memory such as DDR4. Global memory is primarily used to pass large data sets to and from the kernel from the host. It can also be used to pass data between kernels. For recommendations on how to design these interfaces for optimal performance, see Memory Performance Optimizations for AXI4 Interface.

TIP: For each interface, the RTL Kernel wizard generates example AXI master logic in the top-level wrapper to provide a starting point that can be discarded if not needed.

- **Number of AXI master interfaces:** Specify the number of interfaces present on the kernel. The maximum is 16 interfaces. For each interface, you can customize an interface name, data width, and the number of associated arguments. Each interface contains all read and write channels. The default names proposed by the RTL kernel wizard are `m00_axi` and `m01_axi`. If not changed, these names will have to be used when assigning an interface to global memory as described in Mapping Kernel Ports to Memory.
AXI master definition (table columns)

- **Interface name**: Specifies the name of the interface. To ensure maximum compatibility, the argument name follows the same identifier rules as the kernel name.

- **Width (in bytes)**: Specifies the data width of the AXI data channels. Xilinx recommends matching to the native data width of the memory controller AXI4 slave interface. The memory controller slave interface is typically 64 bytes (512 bits) wide.

- **Number of arguments**: Specifies the number of arguments to associate with this interface. Each argument represents a data pointer to global memory that the kernel can access.

**Argument definition**

- **Interface**: Specifies the name of the AXI Interface. This value is copied from the interface name defined in the table, and cannot be modified here.

- **Argument name**: Specifies the name of the pointer argument as it appears on the function prototype signature. Each argument is assigned an ID value. This ID value is used to access the argument from the host software as described in *Host Application*. The ID value assignments can be found on the summary page of this wizard. To ensure maximum compatibility, the argument name follows the same identifier rules as the kernel name. The argument name is used in the generated RTL kernel control register module as an output signal.

### Streaming Interfaces

The streaming interfaces page allows configuration of AXI4-Stream interfaces on the kernel. Streaming interfaces are only available on select platforms and if the chosen platform does not support streaming, then the page does not appear. Streaming interfaces are used for direct host-to-kernel and kernel-to-host communication, as well as continuously operating kernels as described in *Streaming Data Transfers*. 
• **Number of AXI4-Stream interfaces**: Specifies the number of AXI4-Stream interfaces that exist on the kernel. A maximum of 32 interfaces can be enabled per kernel. Xilinx recommends keeping the number of interfaces as low as possible to reduce the amount of area consumed.

• **Name**: Specifies the name of the interface. To ensure maximum compatibility, the argument name follows the same identifier rules as the kernel name.

• **Mode**: Specifies whether the interface is a master or slave interface. An AXI4-Stream slave interface is a read-only interface, and the RTL kernel can be sent data with the `clWriteStream` API from the host program. An AXI4-Stream master interface is a write-only interface, and the host program can receive data through the interface with the `clReadStream` API.

• **Width (bytes)**: Specifies the `TDATA` width (in bytes) of the AXI4-Stream interface. This interface width is limited to 1 to 64 bytes in powers of 2.

The streaming interface uses the `TDATA/TKEEP/TLAST` signals of the AXI4-Stream protocol. Stream transactions consists of a series of transfers where the final transfer is terminated with the assertion of the `TLAST` signal. Stream transfers must adhere to the following:

• AXI4-Stream transfer occurs when `TVALID/TREADY` are both asserted.
• **TDATA** must be 8, 16, 32, 64, 128, 256, or 512 bits wide.

• **TKEEP** (per byte) must be all 1s when **TLAST** is 0.

• **TKEEP** can be used to signal a ragged tail when **TLAST** is 1. For example, on a 4-byte interface, **TKEEP** can only be \(0b0001\), \(0b0011\), \(0b0111\), or \(0b1111\) to specify the last transfer is 1-byte, 2 bytes, 3 bytes, or 4 bytes in size, respectively.

• **TKEEP** cannot be all zeros (even if **TLAST** is 1).

• **TLAST** must be asserted at the end of a packet.

• **TREADY** input/**TVALID** output should be low if kernel is not started to avoid lost transfers.

### Summary

This section summarizes the VLNV for the RTL kernel IP, the software function prototype, and hardware control registers created from options selected in the previous pages. The function prototype conveys what a kernel call would be like if it was a C function. See the host code generated example of how to set the kernel arguments for the kernel call. The register map shows the relationship between the host software ID, argument name, hardware register offset, type, and associated interface. Review this section for correctness before proceeding to generate the kernel.
Click **OK** to generate the top-level wrapper for the RTL kernel, the VADD temporary RTL kernel IP, the `kernel.xml` file, the simulation test bench, and the example `host.cpp` code. After these files are created, the RTL Kernel wizard opens a project in the Vivado Design Suite to let you complete kernel development.

### Using the RTL Kernel Project in Vivado IDE

If you launched the RTL Kernel wizard from the Vitis IDE, after clicking **OK** on the Summary page, the Vivado Design Suite open with an example IP project to let you complete your RTL kernel code.

If you launched the RTL Kernel wizard from within the Vivado IP catalog, after clicking **OK** on the Summary page, an RTL Kernel Wizard IP is instantiated into your current project. From there you must take the following steps:
1. When the Generate Output Products dialog box appears, click **Skip** to close it.

2. Right-click the `<kernel_name>.xci` file that is added to the Sources view, and select **Open IP Example Design**.

3. In the Open Example Design dialog box, specify the **Example project directory**, or accept the default value, and click **OK**.

   **TIP:** An example project is created for the RTL kernel IP. This example IP project is the same as the example project created if you launch the RTL Kernel wizard from the Vitis IDE, and is where you will complete the development work for your kernel.

4. You can now close the original Vivado project from which you launched the RTL Kernel wizard.

Depending on the **Kernel Type** you selected for the kernel options, the example IP project is populated with a top-level RTL kernel file that contains either a Verilog example and control registers as described in **RTL Type Kernel Project**, or an instantiated IP integrator block design as described in **Block Design Type Kernel Project**. The top-level Verilog file contains the expected input/output signals and parameters. These top-level ports are matched to the kernel specification file (`kernel.xml`) and can be combined with your RTL code, or /block design, to complete the RTL kernel.

The AXI4 interfaces defined in the top-level file contain a minimum subset of AXI4 signals required to generate an efficient, high throughput interface. Signals that are not present inherit optimized defaults when connected to the rest of the AXI system. These optimized defaults allow the system to omit AXI features that are not required, saving area and reducing complexity. If your RTL code or block design contains AXI signals that were omitted, you can add these signals to the ports in the top-level RTL kernel file, and the IP packager will adapt to them appropriately.

The next step in the process customizes the contents of the kernel and then packages those contents into a Xilinx Object (`xo`) file.

### RTL Type Kernel Project

The RTL type kernel delivers a top-level Verilog design consisting of control register and the `Vadd` sub-modules example design. The following figure illustrates the top-level design configured with two AXI4-master interfaces. Care should be taken if the Control Register module is modified to ensure that it still aligns with the `kernel.xml` file located in the imports directory of the Vivado kernel project. The example block can be replaced with your custom logic or used as a starting point for your design.
The Vadd example block, shown in the following figure, consists of a simple adder function, an AXI4 read master, and an AXI4 write master. Each defined AXI4 interface has independent example adder code. The first associated argument of each interface is used as the data pointer for the example. Each example reads 16 KB of data, performs a 32-bit add one operation, and then writes out 16 KB of data back in place (the read and write address are the same).
The following table describes some important files in the example IP project, relative to the root of the Vivado project for the kernel, where `<kernel_name>` is the name of the kernel you specified in the RTL Kernel wizard.

**Table 90: RTL Kernel Wizard Source and Test Bench File**

<table>
<thead>
<tr>
<th>Filename</th>
<th>Description</th>
<th>Delivered with Kernel Type</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;kernel_name&gt;_ex.xpr</code></td>
<td>Vivado project file</td>
<td>All</td>
</tr>
<tr>
<td>imports directory</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;kernel_name&gt;.v</code></td>
<td>Kernel top-level module</td>
<td>All</td>
</tr>
<tr>
<td><code>&lt;kernel_name&gt;_control_s_axi.v</code></td>
<td>RTL control register module</td>
<td>RTL</td>
</tr>
<tr>
<td><code>&lt;kernel_name&gt;_example.sv</code></td>
<td>RTL example block</td>
<td>RTL</td>
</tr>
<tr>
<td><code>&lt;kernel_name&gt;_example_vadd.sv</code></td>
<td>RTL example AXI4 vector add block</td>
<td>RTL</td>
</tr>
<tr>
<td><code>&lt;kernel_name&gt;_example_axi_read_master.sv</code></td>
<td>RTL example AXI4 read master</td>
<td>RTL</td>
</tr>
<tr>
<td><code>&lt;kernel_name&gt;_example_axi_write_master.sv</code></td>
<td>RTL example AXI4 write master</td>
<td>RTL</td>
</tr>
<tr>
<td><code>&lt;kernel_name&gt;_example_adder.sv</code></td>
<td>RTL example AXI4-Stream adder block</td>
<td>RTL</td>
</tr>
<tr>
<td><code>&lt;kernel_name&gt;_example_counter.sv</code></td>
<td>RTL example counter</td>
<td>RTL</td>
</tr>
</tbody>
</table>
Table 90: RTL Kernel Wizard Source and Test Bench File (cont’d)

<table>
<thead>
<tr>
<th>Filename</th>
<th>Description</th>
<th>Delivered with Kernel Type</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;kernel_name&gt;_exdes_tb_basic.sv</code></td>
<td>Simulation test bench</td>
<td>All</td>
</tr>
<tr>
<td><code>&lt;kernel_name&gt;_cmodel.cpp</code></td>
<td>Software C-Model example for software emulation.</td>
<td>All</td>
</tr>
<tr>
<td><code>&lt;kernel_name&gt;_ooc.xdc</code></td>
<td>Out-of-context Xilinx constraints file</td>
<td>All</td>
</tr>
<tr>
<td><code>&lt;kernel_name&gt;_user.xdc</code></td>
<td>Xilinx constraints file for kernel user constraints.</td>
<td>All</td>
</tr>
<tr>
<td>kernel.xml</td>
<td>Kernel description file</td>
<td>All</td>
</tr>
<tr>
<td>package_kernel.tcl</td>
<td>Kernel packaging script proc definitions</td>
<td>All</td>
</tr>
<tr>
<td>post_synth_impl.tcl</td>
<td>Tcl post-implementation file</td>
<td>All</td>
</tr>
<tr>
<td>exports directory</td>
<td></td>
<td></td>
</tr>
<tr>
<td>src/host_example.cpp</td>
<td>Host code example</td>
<td>All</td>
</tr>
<tr>
<td>makefile</td>
<td>Makefile example</td>
<td>All</td>
</tr>
</tbody>
</table>

**Block Design Type Kernel Project**

The block design type kernel delivers an IP integrator block design (.bd) at the top-level of the example project. A MicroBlaze processor subsystem is used to sample the control registers and to control the flow of the kernel. The MicroBlaze processor system uses a block RAM as an exchange memory between the host and the kernel instead of a register file.
For each AXI interface, a DMA and math operation sub-blocks are created to provide an example of how to control the kernel execution. The example uses the MicroBlaze AXI4-Stream interfaces to control the AXI Data Mover IP to create an example identical to the one in the RTL kernel type. Also, included is a Vitis IDE project to compile and link an ELF file for the MicroBlaze core. This ELF file is loaded into the Vivado kernel project and initialized directly into the MicroBlaze instruction memory.

The following steps can be used to modify the MicroBlaze processor program:
1. If the design has been updated, you might need to run the Export Hardware option. The option can be found in the **File → Export → Export Hardware** menu location. When the Export Hardware dialog opens, click **OK**.

2. The core development kit application can now be invoked. Select **Tools → Launch Vitis** from the main menu.

3. When the Vitis IDE opens, click **X** just to the right of the text on the Welcome tab to close the welcome dialog box. This shows an already loaded Vitis IDE project underneath.

4. From the Project Explorer, the source files are under the `<Kernel Name>_control/src` section. Modify these as appropriate.

5. When updates are complete, compile the source by selecting the menu option **Project → Build All → Check for errors/warnings and resolve if necessary**. The ELF file is automatically updated in the IDE.

6. Run simulation to test the updated program and debug if necessary.

### Simulation Test Bench

A SystemVerilog test bench is generated for simulating the example IP project. This test bench exercises the RTL kernel to ensure its operation is correct. It is populated with the checker function to verify the `add one` operation.

This generated test bench can be used as a starting point in verifying the kernel functionality. It writes/reads from the control registers and executes the kernel multiple times while also including a simple reset test. It is also useful for debugging AXI issues, reset issues, bugs during multiple iterations, and kernel functionality. Compared to hardware emulation, it executes a more rigorous test of the hardware corner cases, but does not test the interaction between host code and kernel.

To run a simulation, click **Vivado Flow Navigator → Run Simulation** located on the left hand side of the GUI and select **Run Behavioral Simulation**. If behavioral simulation is working as expected, a post-synthesis functional simulation can be run to ensure that synthesis results are matched with the behavioral model.

### Out-of-Context Synthesis

The Vivado kernel project is configured to run synthesis and implementation in out-of-context (OOC) mode. A Xilinx Design Constraints (XDC) file is populated in the design to provide default clock frequencies for this purpose.

You should always synthesize the RTL kernel before packaging it with the `package_xo` command. Running synthesis is useful to determine whether the kernel synthesizes without errors. It also provides estimates of resource utilization and operating frequency. Without presynthesizing the RTL kernel you could encounter errors during the `v++` linking process, and it could be much harder to debug the cause.
To run OOC synthesis, click **Run Synthesis** from the **Vivado Flow Navigator → Synthesis** menu.

The synthesized outputs can also be used to package the RTL kernel with a netlist source, instead of RTL source.

**IMPORTANT!** A **block design type kernel must be packaged as a netlist using the** `package_xo` **command.**

### Software Model and Host Code Example

A C++ software model of the **add one example operation**, `<kernel_name>_cmodel.cpp`, is provided in the `.imports` directory. This software model can also be modified to model the function of your kernel. When running `package_xo`, this model can be included with the kernel source files to enable software emulation for the kernel. The hardware emulation and system builds always use the RTL description of the kernel.

In the `.exports/src` directory, an example host program is provided and is called `host_example.cpp`. The host program takes the binary container as an argument to the program. The host code loads the binary as part of the `init` function. The host code instantiates the kernel, allocates the buffers, sets the kernel arguments, executes the kernel, and then collects and checks the results for the example **add one function**.

For information on using the host program and kernel code in an application, refer to **Creating a Vitis IDE Project**.

### Generate RTL Kernel

After the kernel is designed and tested in the example IP project in the Vivado IDE, the final step is to generate the RTL kernel object (XO) file for use by the Vitis compiler.

Click the **Generate RTL Kernel** command from the **Vivado Flow Navigator → Project Manager** menu. The Generate RTL Kernel dialog box opens with three main packaging options:

- **Sources only kernel**: Packages the kernel using the RTL design sources directly.
- **Pre-synthesized kernel**: Packages the kernel with the RTL design sources with a synthesized cached output that can be used later on in the flow to avoid re-synthesizing. If the target platform changes, the packaged kernel might fall back to the RTL design sources instead of using the cached output.
- **Netlist (DCP) based kernel**: Packages the kernel as a block box, using the netlist generated by the synthesized output of the kernel. This output can be optionally encrypted if necessary. If the target platform changes, the kernel might not be able to re-target the new device and it must be regenerated from the source. If the design contains a block design, the netlist (DCP) based kernel is the only packaging option available.
Optionally, the Software Emulation Sources field lets you specify a software model for your kernel that can be used during software emulation. If the software model contains multiple files, provide a space in between each file in the Source files list, or use the GUI to select multiple files using the CTRL key when selecting the file.

After you click OK, the kernel output products are generated. If the pre-synthesized kernel or netlist kernel option is chosen, then synthesis can run. If synthesis has previously run, it uses those outputs, regardless if they are stale. The kernel Xilinx Object (XO) file is generated in the exports directory of the Vivado kernel project.

At this point, you can close the Vivado kernel project. If the Vivado kernel project was invoked from the Vitis IDE, the example host code called host_example.cpp and kernel Xilinx Object (XO) files are automatically imported into the ./src folder of the application project in the Vitis IDE.

Modifying an Existing RTL Kernel Generated from the Wizard

From the Vitis IDE, you can modify an existing RTL kernel by selecting it from the ./src folder of an application project where it is in use. Right-click the XO file in the Project Explorer view, and select RTL Kernel Wizard. The Vitis IDE attempts to open the Vivado project for the selected RTL kernel.

TIP: If the Vitis IDE is unable to find the Vivado project, it returns an error and does not let you edit the RTL kernel.

A dialog box opens displaying two options to edit an existing RTL kernel. Selecting Edit Existing Kernel Contents re-opens the Vivado Project, letting you modify and regenerate the kernel contents. Selecting Re-customize Existing Kernel Interfaces opens the RTL Kernel wizard. Options other than the Kernel Name can be modified, and the previous Vivado project is replaced.

IMPORTANT! All files and changes in the previous Vivado project are lost when the updated RTL kernel project is created.
Using Vitis Embedded Platforms

This section contains the following chapters:

- Vitis Embedded Platforms
- Using Vitis Embedded Platforms
- Creating Embedded Platforms in Vitis
Vitis Embedded Platforms

Introduction

The Vitis™ unified software platform provides a Platform+Kernel structure to help developers focus on the applications. The decoupling of platform and kernel helps make the platform reusable with multiple kinds of kernels and vice versa.

Xilinx provides pre-built platforms for Alveo™ and embedded evaluation boards. You are free to create your own embedded platforms or customize the Xilinx embedded platforms.

The Vitis software platform is an environment for creating embedded software and accelerated applications on heterogeneous platforms based on FPGAs, Zynq®-7000 SoCs, and Zynq® UltraScale+™ MPSoCs. This document focuses on using embedded platform for Zynq UltraScale+ MPSoC.

Platform Types

The Vitis target platforms can be customized with unique hardware and software components. There are two general types of platforms: fixed platforms and extensible platforms. The first type of platform supports embedded software development and it is a direct analog to the hardware definition file that was previously used for software development with the Xilinx SDK tool. The second type of platform supports application acceleration, and it includes hardware for supporting acceleration kernels, controlling AI Engine for Versal™ ACAP, and software for a target running Linux and the Xilinx Runtime (XRT) library. For more information on the XRT library, see https://github.com/Xilinx/XRT.

The following figure shows the traditional SDK flow for embedded software application development. A Xilinx Shell Archive (XSA) is exported from the Vivado® Design Suite. It is used by SDK for board support package (BSP) generation and creating software applications that apply the BSP.
The following figure shows the Vitis embedded software development flow that supersedes SDK from 2019.2 onwards. The hardware specification is now referred to as the XSA and is exported from a Vivado design but is formatted differently and has a .xsa filename extension. The Vitis core tools create a platform, BSP, and software boot components such as the FSBL and PMU firmware for this type of PMU firmware for fixed XSA and are associated with the Vitis platform. Software applications targeting the platform can then be developed with the Vitis core tools and do not require Linux and the XRT library. See the Vitis Embedded Software Development Flow Documentation in the Vitis Unified Software Platform Documentation (UG1416) for more information.
For product developers who want to accelerate their applications, platforms with hardware and software components that support acceleration kernels can be created with the Vitis core tools. The Vivado Design Suite is used to generate and write a second type of XSA containing a few additional IP blocks and metadata to support kernel connectivity. The following figure shows the acceleration software development flow.

**Figure 124: Vitis Acceleration Kernel Flow**

The Vitis core tool supports application development in multiple languages (OpenCL™, C, C++) but the applications must target a Vitis target platform. A target platform consists of hardware and software components as shown in the following figure. The target platform view on the left side of the page is for the Vitis embedded software development flow, whereas the right side of page shows a platform that supports acceleration kernels. The differences include acceleration kernel requirements of a target with Linux + XRT, metadata, and kernel interface declarations.

**Note:** Custom platform generation sources are available in [https://github.com/Xilinx/Vitis_Embedded_Platform_Source](https://github.com/Xilinx/Vitis_Embedded_Platform_Source).
Figure 125: Vitis Target Platforms

<table>
<thead>
<tr>
<th>Software Component</th>
<th>Hardware Component</th>
</tr>
</thead>
<tbody>
<tr>
<td>Domains:</td>
<td>Domains:</td>
</tr>
<tr>
<td>- Standalone / Baremetal</td>
<td>- Linux + XRT required for acceleration kernel support</td>
</tr>
<tr>
<td>- Linux</td>
<td>- Acceleration</td>
</tr>
<tr>
<td>- FreeRTOS</td>
<td></td>
</tr>
</tbody>
</table>

**Fixed XSA:** (Embedded Software Development)
- IP integrator block design
- Xilinx Devices
  - Zynq UltraScale+ MPSoC
  - Zynq-7000 SoC

**Extensible XSA:** (Supports Acceleration Kernels)
- IP integrator block design
- Declared interfaces for acceleration kernels
- Platform (PFM) metadata and properties
- Xilinx Devices
  - Versal ACAP
  - Zynq UltraScale+ MPSoC
  - Zynq-7000 SoC

Figure 126: Vitis Platform Project Flow

- Vivado → PetaLinux → Vitis/XSCT → Target Platform
- zynqmp_fsbl.elf
- bl31.elf
- pmufw.elf
- u-boot.elf
- sysroot
- rootfs
- system.dtb
Platform Naming Convention

The platform name is used when creating acceleration applications in Vitis or targeting platforms when using the Vitis compiler (v++). Pre-built platform images also use the same file name and directory name.

Pre-built Vitis embedded platforms use the following naming convention.

\<Vendor\>_\<Board\>_\<Feature\>_\<Supported Vitis Tool Version\>_\<Release Version\>

Where:

- **\<Vendor\>**: The board vendor. For all Xilinx-created pre-built platforms, use xilinx.
- **\<Feature\>**: The special function of this platform. For example:
  - base indicates that it connects all possible resources for you to use in an acceleration application.
  - DFX indicates that it supports Xilinx Dynamic Function eXchange (DFX).
- **\<Supported Vitis Tool Version\>**: The specific version of the Vitis development platform that the platform is designed for. This also indicates the version of the Vivado® Design Suite tools that the pre-built platform is created by.
- **\<Release Version\>**: The release version of the platform. The first version is 1.

For example, the following platform names follow the naming convention:

- xilinx_zcu102_base_202020_1
- xilinx_zcu104_base_202020_1
- xilinx_zc706_base_202020_1
- xilinx_zcu102_base_dfx_202020_1

**Note**: Platform source code uses a git branch for versioning. The directory name is \<Board\>_\<Feature\> (for example, zcu102_base). The platform generated from the source in https://github.com/Xilinx/Vitis_Embedded_Platform_Source has the name xilinx_zcu102_base_202020_1.

Embedded Platform Components and Architecture

A platform is the starting point of your Vitis design. Vitis applications are built on top of the platforms.
An embedded platform includes a hardware platform and a software platform.

**Hardware Platform**

The hardware platform is the static, unchanging portion of your hardware design. It includes the Xilinx Support Archive (XSA) file exported from the Vivado Design Suite.

The hardware platform describes platform hardware setup and the acceleration resources that can be used by acceleration applications, for example, Input and output interfaces, clocks, AXI buses, and interrupts. Vitis adds kernels and infrastructure modules to the hardware design as needed to facilitate data movement. Acceleration kernels can share data with platform IPs, but cannot change or modify them. For information about setting up the hardware platform, refer to [Installing Xilinx Runtime and Platforms](#).

**Software Platform**

The software platform is the environment that runs the software to control acceleration kernels for acceleration applications. It includes the domain setup and boot components setup.

By default, all Xilinx pre-built platforms have a Linux domain that has enabled Xilinx Runtime (XRT) so that acceleration applications can run on this environment. The pre-built binaries for Linux kernel image and rootsfs are located in a separate download file on the PetaLinux download page. See the "Common images for Embedded Vitis platforms" section of the Xilinx download center. Because the device tree is unique to each platform, it is provided as a component with the Linux XRT domain inside the platform.

Linux Domain Components must be provided when there is a Linux domain in the embedded platform. These components can be generated by PetaLinux, Yocto, or third-party frameworks. Because these components can be shared across all Xilinx demo boards for the given FPGA family, a common Linux component image generated by PetaLinux is provided for Zynq-7000 SoC and Zynq UltraScale+ MPSoC devices.

The following Linux images can be downloaded from the PetaLinux download page:

- **Root File System (RFS):** Includes binaries, libraries, and setups for a Linux file system. In the Xilinx-provided common rootsfs, XRT has been installed so that acceleration application can run on this Linux environment.

- **Kernel Image:** A compiled Linux kernel. The common kernel image provided by Xilinx includes most Xilinx peripheral drivers.

- **Sysroot:** Used for cross compilation. It provides the libraries to be linked when compiling applications for a target system.

**Note:** Optionally, you can pack Linux domain components into embedded platforms. When creating a Linux application in the Vitis IDE, the Linux domain components in the platform setting will be the default and initial settings if they have been set in the platform. You can overwrite these settings with components installed elsewhere.
Xilinx pre-built embedded platforms and pre-built common Linux components are provided in separate download files. You can regenerate the common Linux components from the platform source files hosted on the Vitis Embedded Platform GitHub repository by setting the environment variable `COMMON_RFS_KHRNL_SYSROOT=FALSE` before running `make`.

---

**Installing Embedded Platforms**

Pre-built Vitis embedded platforms must be downloaded from the Xilinx website:

- Download required Vitis platforms from the [Vitis Embedded Platforms download page](#).
- Download common Linux components from the "Common images for Embedded Vitis platforms" section of the [PetaLinux download page](#).

After you download the embedded platforms, there are three ways to include them in your project:

- Extract the pre-built platforms to `/opt/xilinx/platforms`.
- Extract the pre-built platforms to any folder. Set the `PLATFORM_REPO_PATHS` environment variable to the folder path.
  ```bash
  export PLATFORM_REPO_PATHS=/path/to/platforms
  ```
- Extract the pre-built platforms to any folder. In the Vitis IDE, select Xilinx → Add Custom Platform and select the folder path.

Common Linux components can be extracted to any folder. When you create a Linux application, you will specify the path for the components in the Vitis New Application Project wizard.
Chapter 51

Using Vitis Embedded Platforms

Packaging Images

A new packaging stage is added to the Vitis™ compiler (v++) in 2020.1.

In 2019.2, v++ had two stages:

- `-c` or `--compile` to compile acceleration kernels
- `-l` or `--link` to link acceleration kernels with platform logic.

In the 2020.1 release, the new stage is `-p` or `--package`. The command `V++ --package` generates both `boot.bin` and the `sd_card` image files.

The `--package` command supports both initramfs and Ext4 rootfs images.

In the Vitis IDE, the package stage is automatically called during the build process. You can add additional package options in the system project detail page by double-clicking the `.sprj` file. Package log files, command configuration files, and output files are stored in the `package` directory under the Emulation-SW, Emulation-HW or Hardware directories.

In command line mode, you can pass in package options as v++ options or configuration files. For more detailed information about the v++ `--package` option, refer to the Vitis Compiler Command or the v++ `-help` command, and the Xilinx Vitis Acceleration Examples GitHub repository.

Packaging Images with Ext4 rootfs in the Vitis IDE

When `ext4 rootfs` is provided to the Vitis IDE, the generated `sd_card.img` file includes the following:

- The xclbin file for PL kernel
- The host application
- The Linux kernel image
- The device tree
- The u-boot configuration file: `boot.scr`
• The platform initialization script `init.sh` and platform name in `platform_desc.txt`.

  **Note:** `init.sh` sets up the environment variable `XILINX_XRT` and copies the `platform_desc.txt` file to `/etc/xocl.txt`. This must be executed manually.

• The `ext4` rootfs in the Ext4 partition

To package an image with Ext4 rootfs in the Vitis IDE:

1. Select **File → New → Application Project** to create a new application project in the Vitis IDE.
2. Select the platform (for example, `xilinx_zcu102_base_202010_1`), and click **Next**.
3. Provide a name for the application project (for example, `vadd`)
4. For the System Project selection, select **Create New**.
5. For the Target processor, select the processor that can run the Linux domain (for example, `psu_cortexa53 SMP`), and click **Next**.
6. In the Domain page, select `xrt` and provide the following application settings:
   - Sysroot path (for example, `xilinx-zynqmp-common-v2020.1/sysroots/aarch64-xilinx-linux`)
   - Root FS (for example, `xilinx-zynqmp-common-v2020.1/rootfs.ext4`)
   - Kernel Image (for example, `xilinx-zynqmp-common-v2020.1/Image`)
7. Click **Next**.
8. Select an application template (for example, **Vector Addition**) and click **Finish**.
9. Select the system project and click the **Build** button to build the project.
10. Verify that the `sd_card.img` file was created in the package directory under the **Emulation-SW**, **Emulation-HW** or **Hardware** directory.

**TIP:** To change the path for `sysroot`, `rootfs`, or `kernel` after the application project has been created, double-click the `.sprj` file and change the path in the Options dialog box.

### Packaging Images with initramfs rootfs in the Vitis IDE

When `initramfs rootfs (rootfs.cpio)` is provided to the Vitis IDE, the generated `sd_card.img` includes the following:

• The `xclbin` file for PL kernel
• The host application
• The Linux kernel image
• The device tree
- The `boot.scr`
- The `init.sh`, `platform_desc.txt`, and `initramfs` rootfs in FAT32 partition

**Note:** The `sd_card.img` file does not contain the ext4 partition.

1. In the Vitis IDE, select **File → New → Application Project** to create a new application project.
2. Select the platform (for example, `xilinx_zcu102_base_202010_1`), and click **Next**.
3. Provide the application project name (for example, `vadd`).
4. Select **Create New**.
5. For the target processor, select the processor that can run the Linux domain (for example, `psu_cortexa53 SMP`), and click **Next**.
6. In the Domain page, select the `xrt` domain and provide the application settings as follows:
   a. **Sysroot path** (for example, `your_linux_component_dir/sysroots/aarch64-xilinx-linux`)
   b. **Root FS** (for example, `your_linux_component_dir/rootfs.cpio.gz.u-boot`)
   c. **Kernel Image** (for example, `your_linux_component_dir/Image`)
7. Click **Next**.
8. Select the application template (for example, **Vector Addition**).
9. Select the system project and click the **Build ( сохрани) button to build the project.**
10. Verify that the `sd_card.img` file was created in the `package` directory under the `Emulation-SW, Emulation-HW` or `Hardware` directory.

**Note:** The common Linux components package does not provide initramfs rootfs. For more information about generating initramfs rootfs, refer to *PetaLinux Tools Documentation: Reference Guide* (UG1144).

---

**TIP:** To change the path for `sysroot`, `rootfs`, or `kernel` after the application project has been created, double-click the `.sprj` file and change the path in the Options dialog box.

---

### Writing Images to the SD Card

You can use the Vitis unified software platform accelerated flow to target an embedded platform. This facilitates packaging and creating an SD image with RootFS as an EXT4 partition, because `initramfs` uses Double Data Rate SDRAM (DDR SDRAM) for file system storage. It limits the real usable DDR memory for Linux kernel and applications when the file system size increases. It cannot retain RootFS changes after reboot.

To write EXT4 RootFS to an SD Card:
1. Prepare an SD card binary image file with FAT32 partition for boot and EXT4 partition for RootFS.

2. Write SD card images to the SD card. You can use various tools to do this, such as Etcher on Windows or dd command on Linux.

   **Note:** Refer to Xilinx Answer Record 73711 for detailed information about these tools.

There are various ways to prepare an SD card image. You can use the v++ package tool to generate it, or use an open source tool. A pre-built SD card image is also provided in the base platform package, which you can download from the Vitis Embedded Platforms download page.

The pre-built SD card image (pre-built/sd_card.img) has two partitions:

- **FAT32 partition:** 1 GB size, initialized with the kernel image provided by common Linux components.

- **EXT4 partition:** 2 GB size, initialized with RootFS provided by common Linux components.

To make the pre-built SD card image boot, you must copy the following boot components to the FAT32 partition:

- `pre-built/BOOT.BIN`
- `boot.scr`, `system.dtb`, `init.sh`, and `platform_desc.txt` in the `xrt/image` directory

The pre-built SD card image can be used for evaluation usage and by Windows users. It does not require Vitis or PetaLinux to be installed.

   **Note:** The `v++ --package` with Ext4 partition is not supported on Windows.

   **Note:** `init.sh` sets up the environment variable `XILINX_XRT` and copies the `platform_desc.txt` file to `/etc/xocl.txt`. You must manually run this after Linux boots up before running any acceleration applications.

### Configuring the PL Kernel in DFX Platforms and Non-DFX Platforms

The Xilinx Dynamic Function eXchange (DFX) feature can change some blocks of PL function while keeping other areas of PL working, allowing you to configure PL kernels on the fly. To use the DFX feature, when the `xclbin` file is generated, configure it with your host application. The new kernels in the `xclbin` take effect immediately without requiring a reboot.

For platforms without DFX features, PL kernel must be packed into `boot.bin`. Copy it to the FAT32 partition on your SD card and reboot the system. Then, configure the `xclbin` file with your host application.
The xclbin file contains both bit files for PL kernel and metadata to describe these kernel features and connections. Programming the xclbin file on DFX platforms loads the bit file and metadata; programming on non-DFX platforms only loads the metadata.

Running an Acceleration Application on the Board

If you are using the common Linux components that are provided by Xilinx perform the following to run an acceleration application on the platform:

1. To the SD card, write the sd_card.img generated by the Vitis compiler command v++ --package.
2. Boot the board.
3. Run the command cd /mnt/sd-mmcblk0p1/.
4. Run the command source init.sh.
5. Run acceleration application. For example, for vector addition, run ./vadd ./binary_container_1.xclbin.

Acceleration application uses Xilinx Runtime (XRT) to communicate with acceleration kernels. To set up the environment for XRT, run init.sh. This command does the following:

- It sets the environment variable XILINX_XRT to /usr to allow the application to find the XRT environment.
- It copies platform_desc.txt to /etc/xocl.txt to inform XRT which platform it is running on.

Note: This was done automatically for embedded platforms in the 2019.2 release of Vitis. Because automatically running init.sh may introduce security breaches, common Linux rootfs did no run init.sh by default.

Note: If the sd_card.img file has already been written to the SD card and you are only updating the application, you can save time in the debugging phase by copying all files from <Vitis System Project>/Hardware/package.sd_card to FAT32 partition on SD card to replace existing files. The Ext4 partition does not change in sd_card.img.
Software Package Management in PetaLinux Rootfs

The package management feature is new for the Vitis 2020.1 release. All PetaLinux rootfs software packages are hosted on http://petalinux.xilinx.com/sswreleases/rel-v2020/feeds. You can install these software packages to rootfs when running Linux on the target board as long as the board has Internet access.

To use this feature, you must enable package manager DNF in rootfs. The rootfs in Xilinx-provided pre-built Linux components provides the DNF package management features by default.

To set up a package feed URL:

2. Download the repo file that matches your SoC device to target board.

   # Example: ZCU102 uses ZU9EG devices

   # Example: ZCU104 uses ZU7EV devices

3. Copy the downloaded repo file to /etc/yum.repos.d/.
4. Clean the cache.

   dnf clean all

To manage packages, use DNF:

- Listing available packages: Use the command dnf repoquery.
- Installing packages from a Xilinx repository: Use the command dnf install <pkg name>.

Here are some basic functions you can run with the DNF package manager.

- Listing available packages: Use the command dnf repoquery.
- Installing packages from a Xilinx repository: Use the command dnf install <pkg name>.
- Installing packages from a local package file: Use the command dnf install <pkg file name>.
- Installing packages to sysroot:
When packages are installed on the rootfs of a running board, target has the latest binaries and libraries. When cross compiling on host is needed, these libraries must be added to host side sysroot.

A `sysroot_overlay` script is provided in XRT to extract RPM and update sysroot. This script will extract RPM libraries and include a file update in sysroot.

Besides XRT, this script supports all RPMs for various software packages.

- **Getting the `sysroot_overlay.sh`:** Use the command
  ```bash
  ```

The `sysroot` command description is:

  ```bash
  ./sysroots_overlay.sh --sysroot --rpms-file
  ```

Where:

- `--sysroot` is the sysroot to be overlaid.
- `--rpms-file` is the RPMs file that contains the RPM file paths to be overlaid.

**Examples**

The following example is a command to install updated XRT to the common sysroot:

  ```bash
  ./sysroots_overlay.sh -s sysroots/aarch64-xilinx-linux/ -r $PWD/rpm.txt
  ```

This example shows the contents of an `rpm.txt` file:

  ```bash
  ./xrt-dev-202010.2.6.0-r0.aarch64.rpm
  ./xrt-202010.2.6.0-r0.aarch64.rpm
  ```

**Note:** This script works only for the local RPMs. You must download RPMs to your host machine to install them to the common sysroot.
Chapter 52

Creating Embedded Platforms in Vitis

Platform Creation Basics

In the Vitis™ environment acceleration application development flow, the project is divided into two distinct elements: the platform and the processing subsystem. The platform contains essential IP blocks (such as PS for SoCs, NoC and AI Engine for Versal™ ACAPs) and board interface IP blocks (such as high-speed I/Os and memory controllers). The processing subsystem contains the application-specific part of the system and can be composed of both programmable logic and AI Engine blocks. This approach promotes separation of concerns, facilitates concurrent development, and encourages reusability. The application developer is insulated from the low-level details of the platform and can focus on the specifics of the processing subsystem. The platform developer can focus on system bring-up and tuning I/O performance without having to worry about the processing subsystem. This means that the application developer can integrate the subsystem on different platforms, and a platform can be reused with different processing subsystems.

Xilinx provides pre-built platforms for Alveo™ cards and embedded evaluation boards. You can download these platforms from the Xilinx Download Center. Efficiently leveraging the decoupling of platforms and subsystems is central to the methodology and the productivity gains offered by the Vitis environment. For embedded designs, Xilinx recommends a parallel development process where the application team starts working on the subsystem using a Xilinx pre-built platform while the platform team works independently on bringing-up the custom platform. Rapid progress can be made by working in this manner. Using a pre-built platform means that the subsystem can be developed, integrated, and tested independently using a pre-verified, known-good foundation. After the subsystem is in a sufficiently advanced and stable state, the subsystem can be integrated with appropriate versions of the custom platform. Overall, this approach greatly streamlines the system integration process.

The following figure shows how to create a customized embedded platform.
To create a platform, you must have a base bootable design as a starting point. This design can be a Xilinx base platform design, an existing working design, or a design created from scratch. The following base components must be included in your base bootable design:

- A base hardware design exported from Vivado® Design Suite
- A base software design that includes Linux kernel, root file system, and device tree

After you have working hardware and board through a Vivado® Design Suite design, converting it into a Vitis environment platform requires adding properties to the base components to meet the requirements of the Vitis environment. In general, platform creation consists of the following steps:

1. Add hardware interface parameters and interrupt support in your Vivado® Design Suite project and export the XSA.
2. Update the software platform components to enable application acceleration software stacks (enable XRT, update device tree, and so on).
3. Package and generate the platform using XSCT commands or the Vitis IDE.

The Vitis environment uses the properties in the hardware project to recognize the resources in the platforms and link kernels to the platforms. The Vitis environment uses the software stacks to take control of the kernels.

For details on Vitis environment embedded platform creation, see the Vitis Unified Software Platform Documentation (UG1416). For step-by-step instructions, see the Vitis Platform Creation tutorial.

---

**Platform Creation Requirements**

The base design you created in a Vitis platform is static after the platform creation process is complete.

Vitis does modify parameters based on certain IPs (for example, SmartConnect, NoCs) by adding additional master/slave interfaces. In some situations, PS/CIPS interfaces can also be modified and Versal™ ACAP and AI Engine IP is instantiated in the platform.

The following table shows the workflows to validate the base system on your board.
Table 91: Platform Workflows

<table>
<thead>
<tr>
<th>Workflow</th>
<th>Development</th>
<th>Validation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic board bring-up</td>
<td>Processor basic parameter setup.</td>
<td>Standalone Hello world and Memory Test application run properly.</td>
</tr>
<tr>
<td>Advanced hardware setup</td>
<td>Enable advanced I/O in Processing System (such as USB, Ethernet, Flash, PCIe®, or RC). Add I/O related IP in PL (such as MIPI, EMAC, or HDMI_TX). Add non-Vitis IP (such as AXI BRAM Controller, or Video Processing Subsystem (VPSS) IP).</td>
<td>If these IP have standalone drivers, test them.</td>
</tr>
<tr>
<td>Base software setup</td>
<td>Create PetaLinux project based on hardware platform.</td>
<td>Linux boots up successfully. Peripherals work properly in Linux.</td>
</tr>
</tbody>
</table>

Base Component Requirements

Every hardware platform design must contain a Processing System IP block from the IP catalog.

- Versal ACAP, Zynq® UltraScale+™ MPSoC, and Zynq-7000 SoC devices are supported.
- MicroBlaze™ processors are not supported for controlling acceleration kernels, but can be part of the base hardware.

Creating an Embedded Platform

Adding Hardware Interfaces

The following table shows the possible Vitis inputs and the minimal requirements for an acceleration embedded platform.

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Types Vitis Can Use</th>
<th>Minimum Requirements for AXI MM Kernels</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control Interfaces</td>
<td>AXI Master Interfaces from PS or from AXI Interconnect IP or SmartConnect IP</td>
<td>One AXI4-Lite Master for kernel control</td>
</tr>
<tr>
<td>Memory Interfaces</td>
<td>AXI Slave Interfaces</td>
<td>One memory interface for data exchange</td>
</tr>
<tr>
<td>Streaming Interfaces</td>
<td>AXI4-Stream Interfaces</td>
<td>Not required</td>
</tr>
<tr>
<td>Clock</td>
<td>Multiple clock signals</td>
<td>One clock</td>
</tr>
<tr>
<td>Interrupt</td>
<td>Multiple interrupt signals</td>
<td>One Interrupt</td>
</tr>
</tbody>
</table>
General Requirements

• Every IP used in the platform design that is not part of the standard Vivado IP catalog must be local to the Vivado Design Suite project. References to IP repository paths external to the project are not supported when creating extensible XSA.

• Any platform interface, used for linking to kernels by the Vitis compiler, must be an AXI4, AXI4-Lite, AXI4-Stream, interrupt, clock, or reset type of interface.

• Any platform IP that has an AXI interface for linking to kernels by the Vitis compiler must also have associated clock pins to enable v++ to correctly infer and insert clock domain crossing logic when needed.

• Custom bus type and hardware interfaces on the platform or on kernels are not supported through v++ linker --connectivity.sp and --connectivity.sc directives. If a data bus with a custom bus type needs to be connected to kernels by the Vitis compiler, it must be converted to an AXI4, AXI4-Lite, or AXI4-Stream interface.

Project Type

The Vivado project type needs to be set to extensible Vitis platform type.

When creating a new project, select Project is an extensible Vitis platform.

To change an existing Vivado project to an extensible Vitis platform project, select Project Manager → Settings in the Flow Navigator and enable Project is an extensible Vitis platform.

```tcl
set_property platform.extensible true [current_project]
```

Adding Platform Interfaces

If a component in block design has a PFM property, this component can be recognized by v++ linker and can be used by the acceleration kernel.

In Vivado IDE, the PFM interface properties can be set in the Platform Setup window if the project is created as an extensible platform project. Click Window menu → Platform Setup to open the settings. They can be defined manually in the Tcl Console, or by a Tcl script as well.

The four Platform Interface Tcl APIs include:

• AXI memory-mapped interfaces:

```tcl
set_property PFM.AXI_PORT { <port_name> {parameters} <port2> {parameters} ...} [get_bd_cells <cell_name>]
```

• AXI4-Stream interfaces:

```tcl
set_property PFM.AXIS_PORT { <port_name> {parameters} <port2> {parameters} ...} [get_bd_cells <cell_name>]
```
• Clocks and resets:

```tcl
set_property PFM.CLOCK { <port_name> {parameters} <port2> {parameters} ...} [get_bd_cells <cell_name>]
```

• Interrupts:

```tcl
set_property PFM.IRQ {pin_name {id id_number range irq_count}} [get_bd_cells <cell_name>]
```

The requirements for the PFM Properties are:

• The value of the PFM interface properties must be specified as a Tcl dictionary, a list of name/"value" pairs.

  IMPORTANT! The "value" must be quoted, and both the name and value are case-sensitive.

• A bd_cell can have multiple PFM interface definitions. However, for each type of PFM interface, all ports are required to be set in a single set_property Tcl command.

• For each PFM interface property, the name specified for the port object must match the name of an external port or interface on a bd_cell. Each external port or interface object may only have one PFM interface definition.

• Each different type of PFM interface can have different parameters.

• Setting the PFM property with a NULL (""") string will delete previously defined PFM interfaces.

Adding AXI Interfaces

To support AXI memory mapped kernels, the platform needs to declare at least one AXI control interface with AXI memory-mapped master port (M_AXI_GP) and one memory interface with AXI Slave port. They can be exported from PS block directly or have an interconnect IP connected. If the platform does not work with AXI memory mapped kernels, these interfaces are not required.

The following is the Tcl command syntax:

```tcl
set_property PFM.AXI_PORT { <port_name> {parameters} <port2> {parameters} ...} [get_bd_cells <cell_name>]
```

The AXI control interfaces and AXI memory interfaces share the same PFM.AXI property. They have different memport types.

• AXI control interface can be defined as M_AXI_GP. Memory interfaces use other types: S_AXI_HP, S_AXI_ACP, S_AXI_HPC, or MIG.

• The sptags property for M_AXI_GP port is not supported.

• sptag ID: (Optional) A user-defined ID that should start with an alphabetic character. The ID is case-sensitive. The system port tag (sptag) is a symbolic identifier that represents a class of platform port connections, such as S_AXI_HP, S_AXI_ACP, or M_AXI_GP. Multiple block design platform ports can share the same sptag.
• memory: (Optional) Specify the associated MIG IP instance and address_segment. The memory tag is a unique identifier that combines the Cell Name and Base Name columns in the IP integrator Address Editor. This tag will be associated with connections to the Memory Subsystem HIP, where multiple block design platform ports can share the same memory tag.

Exporting AXI interconnect master and slave ports involves the following requirements:

• All ports on the interconnect used within the platform must precede in index order any declared platform interfaces.

• There can be no gaps in the port indexing.

• The maximum number of master IDs for the S_AXI_ACP port is 8, so on a connected AXI interconnect, available ports to declare must be one of {S00_AXI, S01_AXI, ..., S07_AXI}. Do not declare any ports that are used within the platform itself. Declaring as many as possible will allow sds++ to avoid cascaded axi_interconnects.

• The maximum number of master IDs for an S_AXI_HP or MIG port is 16, so on a connected AXI interconnect, available ports to declare must be one of {S00_AXI, S01_AXI, ..., S15_AXI}. Do not declare any ports that are used within the platform itself. Declaring as many as possible will allow v++ to avoid cascaded axi_interconnects in generated user systems.

• The maximum number of master ports declared on an interconnect connected to an M_AXI_GP port is 64, so on a connected AXI interconnect, available ports to declare must be one of {M00_AXI, M01_AXI, ..., M63_AXI}. Do not declare any ports that are used within the platform itself. Declaring as many as possible will allow v++ to avoid cascaded axi_interconnects in generated user systems.

The following shows an example of defining an AXI master ports on AXI Interconnect IP:

```tcl
set parVal []
for {set i 2} {$i < 64} {incr i} {
    lappend parVal M[format %02d $i]_AXI {
        memport "M_AXI_GP"
    }
}
set_property PFM.AXI_PORT $parVal [get_bd_cells /axi_interconnect_0]
```

The following shows an example of defining AXI memory ports with MIG on SmartConnect IP:

```tcl
set parVal []
for {set i 1} {$i < 16} {incr i} {
    lappend parVal S[format %02d $i]_AXI {
        memport "MIG" sptag "Bank0"
    }
}
set_property PFM.AXI_PORT $parVal [get_bd_cells /smartconnect_0]
```

The following is an example of the PFM.AXI_PORT setting for control interface and memory interface.

```tcl
set_property PFM.AXI_PORT {
    M_AXI_HPM1_FPD {
        memport "M_AXI_GP"
    }
    S_AXI_HPC0_FPD {
        memport "S_AXI_HPC" sptag "HPC0" memory "zynq_ultra_ps_e_0
    } HPC0_DDR_LOW"
    S_AXI_HPC1_FPD {
        memport "S_AXI_HPC" sptag "HPC1" memory "zynq_ultra_ps_e_0"
    }
```
Note:
- zynq_ultra_ps_e_0 is the instance name of the Zynq UltraScale+ MPSoC module.
- HPC0_DDR_LOW is the address range name.

Adding AXI4-Stream Interfaces

To support AXI4-Stream stream kernels, the platform needs to declare the corresponding master or slave AXI4-Stream interfaces.

AXI4-Stream kernel interfaces are specified with the PFM.AXIS_PORT sptag interface property and a matching `connectivity.sc` command argument to the `v++` linker.

The following is the TCL command syntax:

```tcl
set_property PFM.AXIS_PORT { <port_name> {parameters} <port2> {parameters} ...} [get_bd_cells <cell_name>]
```

Argument Description
- **Port_name**: AXI4-Stream port name.
- **Parameters**: type value: Streaming interface port type. Valid values for type include:
  - M_AXIS: A general-purpose AXI master port
  - S_AXIS: A high-performance AXI slave port

Example

```tcl
set_property PFM.AXIS_PORT {AXIS_P0 {type "S_AXIS"}} [get_bd_cells /zynq_ultra_ps_e_0]
```

Note: Refer to `v++` linker config file syntax for how to link AXI4-Stream interface between kernels and platforms (`--connectivity.sc`).

Adding Clock and Resets

You can export any clock source with the platform, but for each clock you must also export synchronized reset signals using a Processor System Reset IP block in the platform. The PFM.CLOCK property can be set on a BD cell, external port, or external interface.
The following is the Tcl command for setting the PFM.CLOCK property:

```tcl
set_property PFM.CLOCK { <port_name> {parameters} \ <port2> {parameters} ...} [get_bd_cells <cell_name>]
```

### Adding Interrupts

Vitis provides a way to automatically connect the kernel output IRQ signal to an IRQ in the platform during the *v++* link stage. The following shows the Tcl command syntax:

```tcl
set_property PFM.IRQ {pin_name {id id_number}} bd_cell
set_property PFM.IRQ {port_name {id id_number range irq_count}} [get_bd_cell <cell_name>]
```

**Argument Description**

- **Port_name**: IRQ port name of bd_cell.
- **id_number**: Integer from 0 to 127 to specify the IRQ number or the starting number if range is specified.
- **irq_count**: Used for labeling interfaces that are otherwise subject to parameter propagation for specifying sizing of a bus (for example, interrupt controller intr interface).

The example shows how to enable 32 IRQ inputs to *axi_intc_0* intr port.

```tcl
set_property PFM.IRQ {intr {id 0 range 32}} [get_bd_cells /axi_intc_0]
```

The example shows how to enable 63 IRQ with cascaded interrupt controller in VCK190 base platform.

```tcl
set_property PFM.IRQ {intr {id 0 range 32}} [get_bd_cells /axi_intc_0]
set_property PFM.IRQ {In0 [id 32] In1 [id 33] In2 [id 34] In3 [id 35] In4 [id 36] In5 [id 37] In6 [id 38] In7 [id 39] In8 [id 40] \ In9 [id 41] In10 [id 42] In11 [id 43] In12 [id 44] In13 [id 45] In14 [id 46] In15 [id 47] In16 [id 48] In17 [id 49] In18 [id 50] \ In19 [id 51] In20 [id 52] In21 [id 53] In22 [id 54] In23 [id 55] In24 [id 56] In25 [id 57] In26 [id 58] In27 [id 59] In28 [id 60] \ xlconcat_0 \ In29 [id 61] In30 [id 62]} [get_bd_cells /xlconcat_0]
```

### Exporting Extensible Platforms

Hardware platforms are encapsulated in XSA file format. There are two kinds of XSA formats: fixed XSA for software development and extensible XSA for acceleration projects. To create a Vitis embedded platform for acceleration flow, you must use an extensible XSA.

When Vivado project type is set to extensible Vitis platform, export platform menu is available in File → Export or Window → Platform Setup → Export Platform button.
In the Export Hardware Platform Window, select platform type. There are four types of platform. If the exported platform will only be used to generate binaries run on hardware board, choose hardware. If it is expected to run hardware emulation with this platform, choose hardware emulation or hardware and hardware emulation. The differences between these two options is that if some modules in the current design is not supported by emulation, you should create an emulation specific design, export as "hardware emulation" platform and then use "Combine XSAs" option to combine a hardware XSA and a hardware emulation XSA into one XSA that is capable of performing both jobs.

For simple designs:

1. Select Hardware and Hardware Emulation, click Next.
2. Select Pre-synthesis for Platform State. Post-implementation is only needed when creating DFX platforms. Click Next.
4. Input the XSA file name and the export target directory. Click Next.
5. Check summary and click Finish.

You can also perform this in the command line using the following command:

```
set_property pfm_name {vendor:board:name:version} [get_files <bd_file>]
write_hw_platform -hw -force <XSA file>
```

To create and combine a hardware XSA and a hardware emulation XSA, use the following commands:

```
write_hw_platform -hw <hw_platform>
write_hw_platform -hw_emu <hw_emu_platform>
combine_hw_platform -hw <hw_platform> -hw_emu <hw_emu_platform> -o <combined_platform>
```

### Updating Software Components

#### Adding XRT to the Root Filesystem

Vitis acceleration application uses XRT to control hardware. XRT provides a unified programming interface across Alveo™ Data Center accelerator cards to embedded use cases.

You must add the XRT kernel driver (zocl) and the user space library (xrt-dev) to rootfs and sysroot. Package xrt-dev enables you to compile Vitis applications that use the XRT API.

#### Updating the Device Tree for ZOCL

The zocl driver interface requires a device tree node to enable the interrupt connection.
The following is an example of the zocl device node.

```
&amba {
    zyxclmm_drm {
        compatible = "xlnx,zocl";
        status = 'okay';
        interrupt-parent = <&axi_intc_0>;
        interrupts = <0  4>, <1  4>, <2  4>, <3  4>,
                    <4  4>, <5  4>, <6  4>, <7  4>,
                    <8  4>, <9  4>, <10 4>, <11 4>,
                    <12 4>, <13 4>, <14 4>, <15 4>,
                    <16 4>, <17 4>, <18 4>, <19 4>,
                    <20 4>, <21 4>, <22 4>, <23 4>,
                    <24 4>, <25 4>, <26 4>, <27 4>,
                    <28 4>, <29 4>, <30 4>, <31 4>;
    }
};
```

For more information, refer to the XRT documentation: [https://xilinx.github.io/XRT/master/html/yocto.html](https://xilinx.github.io/XRT/master/html/yocto.html).

**Update Interrupt Controller Input Number**

In the block diagram, the interrupt controller has not been connected to acceleration kernels. The auto-generated device tree reflects the hardware design of the block diagram and does not consider that v++ linker would connect the interrupt signals of kernels to interrupt controller. To enable these interrupts, override the interrupt input number of the interrupt controllers.

The following is an example on how to override AXI Interconnect node parameters in the `system-user.dtsi`.

```
&axi_intc_0 {
    xlnx,kind-of-intr = <0x0>;
    xlnx,num-intr-inputs = <0x20>;
    interrupt-parent = <&gic>;
    interrupts = <0 89 4>;
};
```

**Declaring the Platform with /etc/xocl.txt**

Platform name can be written into `/etc/xocl.txt` in the embedded platform rootfs, so that XRT knows which platform it is. Host application can use XRT API to get the platform name and check the comparability with XCLBIN and host application with platform.

**Adjusting the CMA Size**

XRT uses CMA for buffer object allocation. You must reserve sufficient memory for CMA in `bootargs` or the device tree to prevent running out of memory during acceleration application runtime.
Packaging a Vitis Acceleration Platform

With all requirements prepared for Vitis acceleration platforms, you can package them together and generate the final Vitis acceleration platform. You can do this using either the Vitis IDE or the Xilinx Software Command-Line Tool (XSCT).

• In the Vitis IDE, select File → New → Platform Project to create a Vitis platform.

• Using XSCT, you can use the platform command to create a platform and the domain command to add domains into a platform. For more information about XSCT, refer to Xilinx Software Command-Line Tool in the Embedded Software Development flow of the Vitis Unified Software Platform Documentation (UG1416).

The platform is an encapsulation of multiple hardware and software components. This encapsulation makes it easier to hand off deliveries from hardware-oriented engineers to application developers.

The following files and information are packaged into the platform.

• **Hardware Specification**: This is an extensible XSA file.

• **Software Components**: These are added to the platform as a Linux domain that enables OpenCL runtime. Software components include the following:
  
  • Boot components
    
    • BIF file that describes the boot components and their properties for Bootgen to generate the boot.bin file.
    
    • A boot components directory that includes all the files described in the BIF file.
  
  • Image directory (optional): Contents in this directory will be copied into the FAT32 partition of the final SD card image.
  
  • Linux domain: The platform requires a Linux domain. The kernel, RootFS, and sysroot info can be added when creating a platform, or when creating an application.
  
  • Emulation support files (optional)

Root Filesystem

FAT32 and Ext4 partition types are supported by Vitis. The root filesystem is optional in platform creation step because it can be assigned during Vitis application creation step.

An image directory needs to be set during platform creation. All contents in this directory will be packaged into final SD card image. If the target file system is FAT32, the files will be placed to SD card root directory; if the target file system is Ext4, the files will be placed to root directory of the first FAT32 partition.
## Boot Components

A BIF file must be provided so that the application build process can package the boot image. The following is an example of a BIF file:

```c
/* linux */
the_ROM_image:
{
    [fsbl_config] a53_x64
    [bootloader] <fsbl.elf>
    [pmufw_image] <pmufw.elf>
    [destination_device=pl] <bitstream>
    [destination_cpu=a53-0, exception_level=el-3, trustzone] <bl31.elf>
    [destination_cpu=a53-0, exception_level=el-2] <u-boot.elf>
}
```

A boot components directory, including all the files described in the BIF, should also be provided. In this example, the components directory provides `fsbl.elf`, `pmufw.elf`, `bl31.elf`, and `u-boot.elf`. These boot components can be generated by PetaLinux.

In the Vitis application building and packaging state, `v++` looks for the files in the boot components directory and replaces the placeholders with real file names and paths. It then calls Bootgen to generate `BOOT.BIN`.

## Testing Your Platform

Before delivering the platform to the application developers, you should run some basic platform tests to make sure it works properly for acceleration applications.

Generally, we need to make sure the platform can pass these tests:

- **Boot test**: The Vivado project generated implementation result BIT file (from Adding Hardware Interfaces) and PetaLinux generated images (from Updating Software Components) should be able to successfully boot to the Linux console.

- **Platforminfo test**: The platform generated in Packaging a Vitis Acceleration Platform should have a proper platforminfo report for clock and memory information.

- **XRT basic test**: The XRT `xbutil query` utility should be able to run on the target board and properly report platform information.

- **Vadd test**: Use Vitis to generate a vector addition sample application with the platform. The generated application and xclbin should print `test pass` on the board.

## Enabling Hardware Emulation for Extensible XSA

The following steps are used for custom platform developers.
1. Create a Vivado project with the necessary BD, RTL, test bench, and other sources.
   a. Note that in 2020.1, only BD can be used in HW EMU, but starting 2020.2, other sources will also be allowed.
   b. For Versal ACAPs only, test bench needs to include BD wrapper instead of including BD directly, because Vitis does performs jobs on this level to insert NoC into simulation.
   c. For Versal ACAPs only, to enable AI Engine in the Vitis platform, the AI Engine block needs to be configured to have only one slave AXI4 Memory-Mapped port enabled and connected to NoC. Vitis based on AI Engine Graph software will make additional auto-connections during v++ linking stage.
   d. For DFX platforms, specify correct PFM properties in the dynamic region BD so that the Vitis tools can attach accelerators correctly.

2. Update the design HW Emulation packaging into XSA.
   a. Before packaging the design into XSA, it is important that your design step through the simulator correctly.
   b. For Versal ACAPs only, prepare the platform design to enable SystemC models. Update the CIPS and NoC IP setting to change SELECTED_SIM_MODEL property to TLM. This ensures that for CIPS IP, the design uses QEMU model on which SW can be run. Following Tcl command can be used in the design. Also, set the parameter to enable SystemC simulation in Vivado:

```
foreach tlmCell [get_bd_cells * -hierarchical -filter {VLNV =~ "*:*:axi_noC:*" || VLNV =~ "*:*:versal_cips:*"]} [set_property SELECTED_SIM_MODEL tlm $tlmCell ]
set_param bd.generateHybridSystemC true
```

   c. Create a test bench in `sim_1` fileset and instantiate the `<top>` module of your design. For Versal ACAPs, Vivado requires that the user test bench should not instantiate the `<top>` module directly. Instead, it should instantiate `<top>_sim_wrapper` module. A file called `<top>_sim_wrapper.v` is generated when you call the `launch_simulation -scripts_only` command. The interface of this module is the same as your `<top>` module, but it instantiates additional simulations models related to an aggregated NoC module created from various logical NoC modules instantiated in the design.

   d. Compile the design, go through the above steps, and start simulation. Because the design is configured to use QEMU, the CIPS IP will not generate any transactions because there is no SW present when doing simulation in Vivado simulator. You will see the following ERROR message in the Vivado simulation, but it indicates that the basic design loads correctly in simulator.

```
##############################################################
# Simulation does not work as Versal CIPS Emulation
(SELECTED_SIM_MODE=tlm) only works with Vitis tool(launch_emulator
tool in Vitis)
```
# ################################################################################

ERROR: [Simtcl 6-50] Simulation engine failed to start: The Simulation shut down unexpectedly during initialization.

Note: To confirm that the design will have correct transactions, you can optionally perform a simulation of the design using the CIPS VIP first before changing it to use TLM (QEMU). First, you must keep the SELECTED_SIM_MODEL property to be RTL for NoC and CIPS IP. Also, create a different test bench which drives the CIPS VIP and also meet the requirement of the NoC Verilog model. Refer to the CIPS VIP and NoC IP documentation for additional details on how to set up test bench for Verilog-based simulation.

3. Package the HW Emulation only XSA.
   a. Use Vivado File → Export → Export Platform to export a Hardware Emulation platform or use the following Tcl command:

```
set_property platform.platform_state 'pre_synth' [current_project]
write_hw_platform -hw_emu -file platform_hw_emu.xsa
```

   b. This XSA can be used with pre-built Linux images or with PetaLinux to create a custom Linux image to create a full platform. Then, the remainder of the Vitis tools can be used to add a kernel to design with the XRT.

Special Considerations for Embedded Platform Creation

Divide Logic Functions to Platform and Kernel

While the designs on FPGA and SoC are getting more complex, it is common for multiple developers or teams to work on a design together. The Vitis software platform provides a clear boundary for application developers and platform developers. Platform developers might include board developers, BSP developers, system software developers, and so on.

In the view of a system architect, some logic functions might be in a gray area: they can be packaged grouped with platforms, or they can work as an acceleration kernel. To help divide the system blocks, here are some general guidelines.

- The basic consideration for classifying a function as a kernel or platform is whether it is an application-related logic.
- Platforms should be more stable than applications. Application function changes should only happen in the software and kernel.
- Platforms abstract hardware. When changing a hardware board, the application should need no change, or very little change if necessary, to target to the new hardware.
- Follow constraints and limitations of the Vitis tool. For example:
  - Only three types of interfaces are supported by Vitis acceleration kernels: AXI MM, AXI4-Lite, and AXI4-Stream.
AXI Kernel does not support external I/O pins.

The following table shows the recommended platforms and kernels for logic types.

<table>
<thead>
<tr>
<th>Logic</th>
<th>Platform</th>
<th>Kernel</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hard Processors (PS of Zynq and Zynq UltraScale+ MPSoC)</td>
<td>Only in Platform</td>
<td></td>
</tr>
<tr>
<td>Soft Processors</td>
<td>Preferred in Platform</td>
<td>OK as an RTL kernel</td>
</tr>
<tr>
<td>I/O Block (External pins, MIPI, PHY, etc.)</td>
<td>Only in Platform</td>
<td></td>
</tr>
<tr>
<td>Related IP for I/O Block (DMA for PCIe®, MAC for Ethernet, etc.)</td>
<td>Generally in platform because the interface between I/O and IP are not AXI.</td>
<td>OK as Kernel if the interfaces between I/O block and IP are AXI.</td>
</tr>
<tr>
<td>IP with non-AXI interface</td>
<td>Only in Platform</td>
<td>OK if the interface can be changed to AXI MM or AXI4-Stream</td>
</tr>
<tr>
<td>Traditional memory mapped IP which has Linux driver (VPSS, etc.)</td>
<td>Only in Platform</td>
<td></td>
</tr>
<tr>
<td>HLS AXI memory mapped IP</td>
<td>OK in Platform. You have to write control software.</td>
<td>Preferred as Kernel. Controlled by XRT.</td>
</tr>
<tr>
<td>Acceleration memory mapped IP follows Vitis kernel register standard and open to XRT</td>
<td>Preferred as Kernel</td>
<td></td>
</tr>
<tr>
<td>Vitis Libraries</td>
<td>Only work as Kernel</td>
<td></td>
</tr>
<tr>
<td>Free running IP with AXI4-Stream interface</td>
<td>OK</td>
<td>OK</td>
</tr>
</tbody>
</table>

References

For more information on embedded platforms, see the following links:

- Vitis Embedded Acceleration Platform Creation Tutorial
- Vitis Embedded Platform Source
Section IX

Additional Information

This section contains the following chapters:

- OpenCL Kernel Development
- Migrating to a New Target Platform
Chapter 53

OpenCL Kernel Development

The following OpenCL™ kernel discussion is based on the information provided in the C/C++ Kernels topic. The same programming techniques for accelerating the performance of a kernel apply to both C/C++ and OpenCL kernels. However, the OpenCL kernel uses the __attribute syntax in place of pragmas. For details of the available attributes, refer to OpenCL Attributes.

The following code examples show some of the elements of an OpenCL kernel for the Vitis™ application acceleration development flow. This is not intended to be a primer on OpenCL or kernel development, but to merely highlight some of the key difference between OpenCL and C/C++ kernels.

Kernel Signature

In C/C++ kernels, the kernel is identified on the Vitis compiler command line using the v++ --kernel option. However, in OpenCL code, the __kernel keyword identifies a kernel in the code. You can have multiple kernels defined in a single .cl file, and the Vitis compiler will compile all of the kernels, unless you specify the --kernel option to identify which kernel to compile.

```
__kernel __attribute__ ((reqd_work_group_size(1, 1, 1)))
void apply_watermark(__global const TYPE * __restrict input,
    __global TYPE * __restrict output, int width, int height) {
    ...
}
```

TIP: The complete code for the kernel function above, apply_watermark, can be found in the Global Memory Two Banks (CL) example in the Vitis Accel Examples GitHub repository.

In the example above, you can see the watermark kernel has two pointer type arguments: input and output, and has two scalar type int arguments: width and height.

In C/C++ kernels, these arguments would need to be identified with the HLS INTERFACE pragmas. However, in the OpenCL kernel, the Vitis compiler, and Vitis HLS recognize the kernel arguments, and compile them as needed: pointer arguments into m_axi interfaces, and scalar arguments into s_axilite interfaces.
Kernel Optimizations

Because the kernel is running in programmable logic on the target platform, optimizing your task to the environment is an important element of application design. Most of the optimization techniques discussed in C/C++ Kernels can be applied to OpenCL kernels. Instead of applying the HLS pragmas used for C/C++ kernels, you will use the `__attribute__` keyword described in OpenCL Attributes. Following is an example:

```c
// Process the whole image
__attribute__((xcl_pipeline_loop))
image_traverse: for (uint idx = 0, x = 0, y = 0 ; idx < size ; ++idx, x+= DATA_SIZE)
{
   ...
}
```

The example above specifies that the `for` loop, `image_traverse`, should be pipelined to improve the performance of the kernel. The target II in this case is 1. For more information, refer to `xcl_pipeline_loop`.

In the following code example, the watermark function uses the `opencl_unroll_hint` attribute to let the Vitis compiler unroll the loop to reduce latency and improve performance. However, in this case the `__attribute__` is only a suggestion that the compiler can ignore if needed. For details, refer to `opencl_unroll_hint`.

```c
// Unrolling below loop to process all 16 pixels concurrently
__attribute__((opencl_unroll_hint))
watermark: for (int i = 0 ; i < DATA_SIZE ; i++)
{
   ...
}
```

For more information, review the OpenCL Attributes topics to see what specific optimizations are supported for OpenCL kernels, and review the C/C++ Kernels content to see how these optimizations can be applied in your kernel design.

Setting Data Width in OpenCL Kernels

For OpenCL kernels, the API provides attributes to support incrementing AXI data width usage. To eliminate manual code modifications, the following OpenCL attributes are interpreted to perform data path widening and vectorization of the algorithm:

- `vec_type_hint`
- `reqd_work_group_size`
- `xcl_zero_global_work_offset`
Examine the combined functionality on the following case:

```c
__attribute__((reqd_work_group_size(64, 1, 1)))
__attribute__((vec_type_hint(int)))
__attribute__((xcl_zero_global_work_offset))
__kernel void vector_add(__global int* c, __global const int* a, __global const int* b) {
    size_t idx = get_global_id(0);
    c[idx] = a[idx] + b[idx];
}
```

In this case, the hard coded interface is a 32-bit wide data path `(int *c, int* a, int *b)`, which drastically limits the memory throughput if implemented directly. However, the automatic widening and transformation is applied, based on the values of the three attributes.

- **__attribute__((vec_type_hint(int)))**: Declares that `int` is the main type used for computation and memory transfer (32-bit). This knowledge is used to calculate the vectorization/widening factor based on the target bandwidth of the AXI interface (512 bits). In this example the factor would be $16 = 512 \text{ bits} / 32\text{.bit}$. This implies that in theory, 16 values could be processed if vectorization can be applied.

- **__attribute__((reqd_work_group_size(X, Y, Z)))**: Defines the total number of work items (where X, Y, and Z are positive constants). $X*Y*Z$ is the maximum number of work items therefore defining the maximum possible vectorization factor which would saturate the memory bandwidth. In this example, the total number of work items is $64*1*1=64$.

  The actual vectorization factor to be applied will be the greatest common divider of the vectorization factor defined by the actual coded type or the `vec_type_hint`, and the maximum possible vectorization factor defined through `reqd_work_group_size`.

  The quotient of maximum possible vectorization factor divided by the actual vectorization factor provides the remaining loop count of the OpenCL description. As this loop is pipelined, it can be advantageous to have several remaining loop iterations to take advantage of a pipelined implementation. This is especially true if the vectorized OpenCL code has long latency.

- **__attribute__((xcl_zero_global_work_offset))**: The `__attribute__((xcl_zero_global_work_offset))` instructs the compiler that no global offset parameter is used at runtime, and all accesses are aligned. This gives the compiler valuable information with regard to alignment of the work groups, which in turn usually propagates to the alignment of the memory accesses (less hardware).

  It should be noted, that the application of these transformations changes the actual design to be synthesized. Partially unrolled loops require reshaping of local arrays in which data is stored. This usually behaves nicely, but can interact poorly in rare situations.

  For example:
• For partitioned arrays, when the partition factor is not divisible by the unrolling/vectorization factor.
  ○ The resulting access requires a lot of multiplexers and will create a difficult issue for the scheduler (might severely increase memory usage and compilation time). Xilinx recommends using partitioning factors that are powers of two (as the vectorization factor is always a power of two).

• If the loop being vectorized has an unrelated resource constraint, the scheduler complains about II not being met.
  ○ This is not necessarily correlated with a loss of performance (usually it is still performing better) because the II is computed on the unrolled loop (which has therefore a multiplied throughput for each iteration).
  ○ The scheduler informs you of the possible resources constraints and resolving those will further improve the performance.
  ○ Note that a common occurrence is that a local array does not get automatically reshaped (usually because it is accessed in a later section of the code in non-vectorizable method).

### Reducing Kernel to Kernel Communication Latency in OpenCL Kernels

The OpenCL API 2.0 specification introduces a new memory object called a pipe. A pipe stores data organized as a FIFO. Pipe objects can only be accessed using built-in functions that read from and write to a pipe. Pipe objects are not accessible from the host. Pipes can be used to stream data from one kernel to another inside the FPGA without having to use the external memory, which greatly improves the overall system latency. For more information, see Pipe Functions on Version 2.0 of the OpenCL C Specification from Khronos Group.

In the Vitis IDE, pipes must be statically defined outside of all kernel functions. Dynamic pipe allocation using the OpenCL 2.x `clCreatePipe` API is not supported. The depth of a pipe must be specified by using the OpenCL attribute `xcl_reqd_pipe_depth` in the pipe declaration. For more information, see `xcl_reqd_pipe_depth`.

As specified in `xcl_reqd_pipe_depth`, the valid depth values are as follows: 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768.

A given pipe can have one and only one producer and consumer in different kernels.

```c
pipe int p0 __attribute__((xcl_reqd_pipe_depth(32)));
```

Pipes can be accessed using the Xilinx extended `read_pipe_block()` and `write_pipe_block()` functions in blocking mode.

**TIP:** Reading or writing to pipes using the non-blocking `read_pipe()` or `write_pipe()` functions is not supported.
The status of pipes can be queried using OpenCL `get_pipe_num_packets()` and `get_pipe_max_packets()` built-in functions.

The following function signatures are the currently supported pipe functions, where `gentype` indicates the built-in OpenCL C scalar integer or floating-point data types.

```c
int read_pipe_block (pipe gentype p, gentype *ptr)
int write_pipe_block (pipe gentype p, const gentype *ptr)
```

The following “dataflow/dataflow_pipes_ocl” from Xilinx Getting Started Examples on GitHub uses pipes to pass data from one processing stage to the next using blocking `read_pipe_block()` and `write_pipe_block()` functions:

```c
pipe int p0 __attribute__((xcl_reqd_pipe_depth(32)));
pipe int p1 __attribute__((xcl_reqd_pipe_depth(32)));
// Input Stage Kernel : Read Data from Global Memory and write into Pipe P0
kernel __attribute__ ((reqd_work_group_size(1, 1, 1)))
void input_stage(__global int *input, int size)
{
    __attribute__((xcl_pipeline_loop))
    mem_rd: for (int i = 0 ; i < size ; i++)
    {
        //blocking Write command to pipe P0
        write_pipe_block(p0, &input[i]);
    }
}
// Adder Stage Kernel: Read Input data from Pipe P0 and write the result into Pipe P1
// into Pipe P1
kernel __attribute__ ((reqd_work_group_size(1, 1, 1)))
void adder_stage(int inc, int size)
{
    __attribute__((xcl_pipeline_loop))
    execute: for(int i = 0 ; i < size ;  i++)
    {
        int input_data, output_data;
        //blocking read command to Pipe P0
        read_pipe_block(p0, &input_data);
        output_data = input_data + inc;
        //blocking write command to Pipe P1
        write_pipe_block(p1, &output_data);
    }
}
// Output Stage Kernel: Read result from Pipe P1 and write the result to Global Memory
// Memory
kernel __attribute__ ((reqd_work_group_size(1, 1, 1)))
void output_stage(__global int *output, int size)
{
    __attribute__((xcl_pipeline_loop))
    mem_wr: for (int i = 0 ; i < size ; i++)
    {
        //blocking read command to Pipe P1
        read_pipe_block(p1, &output[i]);
    }
}
```
The Device Traceline view shows the detailed activities and stalls on the OpenCL pipes after hardware emulation is run. This information can be used to choose the correct FIFO sizes to achieve the optimal application area and performance.

Figure 128: Device Traceline View

---

OpenCL Attributes

This section describes OpenCL™ attributes that can be added to source code to assist system optimization by the Vitis core development kit, and Vitis HLS tool synthesis.

The Vitis core development kit provides OpenCL attributes to optimize your code for data movement and kernel performance. The goal of data movement optimization is to maximize the system level data throughput by maximizing interface bandwidth usage and DDR bandwidth usage. The goal of kernel computation optimization is to create processing logic that can consume all the data as soon as they arrive at kernel interfaces. This is generally achieved by expanding the processing code to match the data path with techniques, such as function inlining and pipelining, loop unrolling, array partitioning, dataflowing, and so on.

The following table includes the OpenCL attributes are specified by type.
Table 92: OpenCL Attributes by Type

<table>
<thead>
<tr>
<th>Type</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kernel Optimization</td>
<td>• reqd_work_group_size</td>
</tr>
<tr>
<td></td>
<td>• vec_type_hint</td>
</tr>
<tr>
<td></td>
<td>• work_group_size_hint</td>
</tr>
<tr>
<td></td>
<td>• xcl_latency</td>
</tr>
<tr>
<td></td>
<td>• xcl_max_work_group_size</td>
</tr>
<tr>
<td></td>
<td>• xcl_zero_global_work_offset</td>
</tr>
<tr>
<td>Function Inlining</td>
<td>• always_inline</td>
</tr>
<tr>
<td>Task-level Pipeline</td>
<td>• xcl_dataflow</td>
</tr>
<tr>
<td></td>
<td>• xcl_reqd_pipe_depth</td>
</tr>
<tr>
<td>Pipeline</td>
<td>• xcl_pipeline_loop</td>
</tr>
<tr>
<td></td>
<td>• xcl_pipeline_workitems</td>
</tr>
<tr>
<td>Loop Optimization</td>
<td>• opencl_unroll_hint</td>
</tr>
<tr>
<td></td>
<td>• xcl_loop_tripcount</td>
</tr>
<tr>
<td></td>
<td>• xcl_pipeline_loop</td>
</tr>
<tr>
<td>Array Optimization</td>
<td>• xcl_array_partition</td>
</tr>
<tr>
<td></td>
<td>• xcl_array_reshape</td>
</tr>
</tbody>
</table>

*Note:* Array variables only accept a single array optimization attribute.

**TIP:** The Vitis compiler also supports many of the standard attributes supported by gcc, such as:

- ALWAYS_INLINE
- NOINLINE
- UNROLL
- NOUNROLL

**always_inline**

**Description**

The ALWAYS_INLINE attribute indicates that a function must be inlined. This attribute is a standard feature of GCC, and a standard feature of the Vitis compiler.

**TIP:** The NOINLINE attribute is also a standard feature of GCC, and is also supported by the Vitis compiler.

This attribute enables a compiler optimization to have a function inlined into the calling function. The inlined function is dissolved and no longer appears as a separate level of hierarchy in the RTL.
In some cases, inlining a function allows operations within the function to be shared and optimized more effectively with surrounding operations in the calling function. However, an inlined function can no longer be shared with other functions, so the logic might be duplicated between the inlined function and a separate instance of the function which can be more broadly shared. While this can improve performance, this will also increase the area required for implementing the RTL.

For OpenCL kernels, the Vitis compiler uses its own rules to inline or not inline a function. To directly control inlining functions, use the ALWAYS_INLINE or NOINLINE attributes.

By default, inlining is only performed on the next level of function hierarchy, not sub-functions.

**IMPORTANT!** When used with the XCL_DATAFLOW attribute, the compiler will ignore the ALWAYS_INLINE attribute and not inline the function.

**Syntax**

Place the attribute in the OpenCL API source before the function definition to always have it inlined whenever the function is called.

```c
__attribute__((always_inline))
```

**Examples**

This example adds the ALWAYS_INLINE attribute to function foo:

```c
__attribute__((always_inline))
void foo ( a, b, c, d ) {
    ...
}
```

This example prevents the inlining of the function foo:

```c
__attribute__((noinline))
void foo ( a, b, c, d ) {
    ...
}
```

**See Also**

- [https://gcc.gnu.org](https://gcc.gnu.org)

**opencl_unroll_hint**

**Description**

**IMPORTANT!** This is a compiler hint which the compiler can ignore.
Loop unrolling is an optimization technique available in the Vitis compiler. The purpose of the loop unroll optimization is to expose concurrency to the compiler. This newly exposed concurrency reduces latency and improves performance, but also consumes more FPGA fabric resources.

The OPENCL_UNROLL_HINT attribute is part of the OpenCL Specification, and specifies that loops (for, while, do) can be unrolled by the Vitis compiler. See Loop Unrolling for more information.

The OPENCL_UNROLL_HINT attribute qualifier must appear immediately before the loop to be affected. You can use this attribute to specify full unrolling of the loop, partial unrolling by a specified amount, or to disable unrolling of the loop.

Syntax

Place the attribute in the OpenCL source before the loop definition:

```
__attribute__((opencl_unroll_hint(<n>)))
```

Where:

- `<n>` is an optional loop unrolling factor and must be a positive integer, or compile time constant expression. An unroll factor of 1 disables unrolling.

**TIP:** If `<n>` is not specified, the compiler automatically determines the unrolling factor for the loop.

Examples

The following example unrolls the `for` loop by a factor of 2. This results in two parallel loop iterations instead of four sequential iterations for the compute unit to complete the operation.

```
__attribute__((opencl_unroll_hint(2)))
for(int i = 0; i < LENGTH; i++) {
    bufc[i] = bufa[i] * bufb[i];
}
```

Conceptually the compiler transforms the loop above to the following code.

```
for(int i = 0; i < LENGTH; i+=2) {
    bufc[i] = bufa[i] * bufb[i];
    bufc[i+1] = bufa[i+1] * bufb[i+1];
}
```

See Also

- [https://www.khronos.org/](https://www.khronos.org/)
- *The OpenCL C Specification*
**reqd_work_group_size**

**Description**

When OpenCL API kernels are submitted for execution on an OpenCL device, they execute within an index space, called an ND range, which can have 1, 2, or 3 dimensions. This is called the global size in the OpenCL API. The work-group size defines the amount of the ND range that can be processed by a single invocation of a kernel compute unit (CU). The work-group size is also called the local size in the OpenCL API. The OpenCL compiler can determine the work-group size based on the properties of the kernel and selected device. After the work-group size (local size) is determined, the ND range (global size) is divided automatically into work-groups, and the work-groups are scheduled for execution on the device.

Although the OpenCL compiler can define the work-group size, the specification of the REQD_WORK_GROUP_SIZE attribute on the kernel to define the work-group size is highly recommended for FPGA implementations of the kernel. The attribute is recommended for performance optimization during the generation of the custom logic for a kernel.

---

**TIP:** In the case of an FPGA implementation, the specification of the REQD_WORK_GROUP_SIZE attribute is highly recommended as it can be used for performance optimization during the generation of the custom logic for a kernel.

---

OpenCL kernel functions are executed exactly one time for each point in the ND range index space. This unit of work for each point in the ND range is called a work-item. Work-items are organized into work-groups, which are the unit of work scheduled onto compute units. The optional REQD_WORK_GROUP_SIZE attribute defines the work-group size of a compute unit that must be used as the local_work_size argument to clEnqueueNDRangeKernel. This allows the compiler to optimize the generated code appropriately for this kernel.

**Syntax**

Place this attribute before the kernel definition, or before the primary function specified for the kernel.

```
__attribute__((reqd_work_group_size(<X>, <Y>, <Z>)))
```

Where:

- `<X>`, `<Y>`, `<Z>`: Specifies the ND range of the kernel. This represents each dimension of a three dimensional matrix specifying the size of the work-group for the kernel.
Examples

The following OpenCL C kernel code shows a vector addition design where two arrays of data are summed into a third array. The required size of the work-group is 16x1x1. This kernel will execute 16 times to produce a valid result.

```c
#include <clc.h>

__attribute__ ((reqd_work_group_size(16, 1, 1)))
__kernel void vadd(__global int* a,
                  __global int* b,
                  __global int* c)
{
  int idx = get_global_id(0);
  c[idx] = a[idx] + b[idx];
}
```

See Also

- [https://www.khronos.org/](https://www.khronos.org/)
- *The OpenCL C Specification*

**vec_type_hint**

**Description**

```
IMPORTANT! This is a compiler hint which the compiler can ignore.
```

The optional `__attribute__((vec_type_hint(<type>)))` is part of the OpenCL Language Specification, and hints to the OpenCL compiler representing the computational width of the kernel, providing a basis for calculating processor bandwidth usage when the compiler is looking to auto-vectorize the code.

By default, the kernel is assumed to have the `__attribute__((vec_type_hint(int)))` qualifier. This lets you specify a different vectorization type.

Implicit in autovectorization is the assumption that any libraries called from the kernel must be re-compilable at runtime to handle cases where the compiler decides to merge or separate work items. This means that these libraries can never be hard-coded binaries or that hard-coded binaries must be accompanied either by source or some re-targetable intermediate representation. This might be a code security question for some.
Syntax

Place this attribute before the kernel definition, or before the primary function specified for the kernel.

```c
__attribute__((vec_type_hint(<type>)))
```

Where:

- `<type>`: is one of the built-in vector types listed in the following table, or the constituent scalar element types.

  Note: When not specified, the kernel is assumed to have an INT type.

### Table 93: Vector Types

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>char&lt;n&gt;</td>
<td>A vector of <code>&lt;n&gt;</code> 8-bit signed two’s complement integer values.</td>
</tr>
<tr>
<td>uchar&lt;n&gt;</td>
<td>A vector of <code>&lt;n&gt;</code> 8-bit unsigned integer values.</td>
</tr>
<tr>
<td>short&lt;n&gt;</td>
<td>A vector of <code>&lt;n&gt;</code> 16-bit signed two’s complement integer values.</td>
</tr>
<tr>
<td>ushort&lt;n&gt;</td>
<td>A vector of <code>&lt;n&gt;</code> 16-bit unsigned integer values.</td>
</tr>
<tr>
<td>int&lt;n&gt;</td>
<td>A vector of <code>&lt;n&gt;</code> 32-bit signed two’s complement integer values.</td>
</tr>
<tr>
<td>uint&lt;n&gt;</td>
<td>A vector of <code>&lt;n&gt;</code> 32-bit unsigned integer values.</td>
</tr>
<tr>
<td>long&lt;n&gt;</td>
<td>A vector of <code>&lt;n&gt;</code> 64-bit signed two’s complement integer values.</td>
</tr>
<tr>
<td>ulong&lt;n&gt;</td>
<td>A vector of <code>&lt;n&gt;</code> 64-bit unsigned integer values.</td>
</tr>
<tr>
<td>float&lt;n&gt;</td>
<td>A vector of <code>&lt;n&gt;</code> 32-bit floating-point values.</td>
</tr>
<tr>
<td>double&lt;n&gt;</td>
<td>A vector of <code>&lt;n&gt;</code> 64-bit floating-point values.</td>
</tr>
</tbody>
</table>

Note: `<n>` is assumed to be 1 when not specified. The vector data type names defined above where `<n>` is any value other than 2, 3, 4, 8 and 16, are also reserved. Therefore, `<n>` can only be specified as 2, 3, 4, 8, and 16.

Examples

The following example autovectorizes assuming double-wide integer as the basic computation width.

```c
#include <clc.h>
// For VHLS OpenCL C kernels, the full work group is synthesized
__attribute__((vec_type_hint(double)))
__attribute__((reqd_work_group_size(16, 1, 1)))
__kernel void
...
```
See Also

- https://www.khronos.org/
- *The OpenCL C Specification*

**work_group_size_hint**

**Description**

**IMPORTANT! This is a compiler hint, which the compiler might ignore.**

The work-group size in the OpenCL API standard defines the size of the ND range space that can be handled by a single invocation of a kernel compute unit. When OpenCL kernels are submitted for execution on an OpenCL device, they execute within an index space, called an ND range, which can have 1, 2, or 3 dimensions.

OpenCL kernel functions are executed exactly one time for each point in the ND range index space. This unit of work for each point in the ND range is called a work-item. Unlike *for* loops in C, where loop iterations are executed sequentially and in-order, an OpenCL runtime and device is free to execute work-items in parallel and in any order.

Work-items are organized into work-groups, which are the unit of work scheduled onto compute units. The optional WORK_GROUP_SIZE_HINT attribute is part of the OpenCL Language Specification, and is a hint to the compiler that indicates the work-group size value most likely to be specified by the `local_work_size` argument to `clEnqueueNDRangeKernel`. This allows the compiler to optimize the generated code according to the expected value.

**TIP:** In the case of an FPGA implementation, the specification of the REQD_WORK_GROUP_SIZE attribute, instead of the WORK_GROUP_SIZE_HINT is highly recommended because it can be used for performance optimization during the generation of the custom logic for a kernel.

**Syntax**

Place this attribute before the kernel definition, or before the primary function specified for the kernel:

```
__attribute__((work_group_size_hint(<X>, <Y>, <Z>)))
```

Where:

- `<X>`, `<Y>`, `<Z>`: Specifies the ND range of the kernel. This represents each dimension of a three dimensional matrix specifying the size of the work-group for the kernel.
Examples

The following example is a hint to the compiler that the kernel will most likely be executed with a work-group size of 1.

```c
__attribute__((work_group_size_hint(1, 1, 1)))
__kernel void ...
```

See Also

- [https://www.khronos.org/](https://www.khronos.org/)
- *The OpenCL C Specification*

### xcl_array_partition

**Description**

**IMPORTANT!** Array variables only accept one attribute. While XCL_ARRAY_PARTITION does support multi-dimensional arrays, you can only reshape one dimension of the array with a single attribute.

An advantage of using the FPGA over other compute devices for OpenCL programs is the ability for the application programmer to customize the memory architecture all throughout the system and into the compute unit. By default, the Vitis compiler generates a memory architecture within the compute unit that maximizes local and private memory bandwidth based on static code analysis of the kernel code. Further optimization of these memories is possible based on attributes in the kernel source code, which can be used to specify physical layouts and implementations of local and private memories. The attribute in the Vitis compiler to control the physical layout of memories in a compute unit is `array_partition`.

For one-dimensional arrays, the XCL_ARRAY_PARTITION attribute implements an array declared within kernel code as multiple physical memories instead of a single physical memory. The selection of which partitioning scheme to use depends on the specific application and its performance goals. The array partitioning schemes available in the Vitis compiler are cyclic, block, and complete.

**Syntax**

Place the attribute with the definition of the array variable.

```c
__attribute__((xcl_array_partition(<type>, <factor>, <dimension>)))
```

Where:

- `<type>`: Specifies one of the following partition types:
- **cyclic**: Cyclic partitioning is the implementation of an array as a set of smaller physical memories that can be accessed simultaneously by the logic in the compute unit. The array is partitioned cyclically by putting one element into each memory before coming back to the first memory to repeat the cycle until the array is fully partitioned.

- **block**: Block partitioning is the physical implementation of an array as a set of smaller memories that can be accessed simultaneously by the logic inside the compute unit. In this case, each memory block is filled with elements from the array before moving on to the next memory.

- **complete**: Complete partitioning decomposes the array into individual elements. For a one-dimensional array, this corresponds to resolving a memory into individual registers. The default `<type>` is `complete`.

  - `<factor>`: For cyclic type partitioning, the `<factor>` specifies how many physical memories to partition the original array into in the kernel code. For block type partitioning, the `<factor>` specifies the number of elements from the original array to store in each physical memory.

  **IMPORTANT!** For complete type partitioning, the `<factor>` is not specified.

  - `<dimension>`: Specifies which array dimension to partition. Specified as an integer from 1 to `<N>`. Vitis core development kit supports arrays of N dimensions and can partition the array on any single dimension.

**Example 1**

For example, consider the following array declaration.

```c
int buffer[16];
```

The integer array, named buffer, stores 16 values that are 32-bits wide each. Cyclic partitioning can be applied to this array with the following declaration.

```c
int buffer[16] __attribute__((xcl_array_partition(cyclic,4,1)));
```

In this example, the cyclic `<partition_type>` attribute tells the Vitis compiler to distribute the contents of the array among four physical memories. This attribute increases the immediate memory bandwidth for operations accessing the array buffer by a factor of four.

All arrays inside a compute unit in the context of the Vitis core development kit are capable of sustaining a maximum of two concurrent accesses. By dividing the original array in the code into four physical memories, the resulting compute unit can sustain a maximum of eight concurrent accesses to the array buffer.
Example 2

Using the same integer array as found in Example 1, block partitioning can be applied to the array with the following declaration.

```c
int buffer[16] __attribute__((xcl_array_partition(block,4,1)));
```

Because the size of the block is four, the Vitis compiler will generate four physical memories, sequentially filling each memory with data from the array.

Example 3

Using the same integer array as found in Example 1, complete partitioning can be applied to the array with the following declaration.

```c
int buffer[16] __attribute__((xcl_array_partition(complete, 1)));
```

In this example, the array is completely partitioned into distributed RAM, or 16 independent registers in the programmable logic of the kernel. Because complete is the default, the same effect can also be accomplished with the following declaration.

```c
int buffer[16] __attribute__((xcl_array_partition));
```

While this creates an implementation with the highest possible memory bandwidth, it is not suited to all applications. The way in which data is accessed by the kernel code through either constant or data dependent indexes affects the amount of supporting logic that the Vitis compiler has to build around each register to ensure functional equivalence with the usage in the original code. As a general best practice guideline for the Vitis core development kit, the complete partitioning attribute is best suited for arrays in which at least one dimension of the array is accessed through the use of constant indexes.

See Also
- `xcl_array_reshape`
- `pragma HLS array_partition`
- `Vitis HLS Flow`

**xcl_array_reshape**

**Description**

> IMPORTANT! Array variables only accept one attribute. While the XCL_ARRAY_RESHAPE attribute does support multi-dimensional arrays, you can only reshape one dimension of the array with a single attribute.

This attribute combines array partitioning with vertical array mapping.
The XCL_ARRAY_RESHAPE attribute combines the effect of XCL_ARRAY_PARTITION, breaking an array into smaller arrays, and concatenating elements of arrays by increasing bit-widths. This reduces the number of block RAM consumed while providing parallel access to the data. This attribute creates a new array with fewer elements but with greater bit-width, allowing more data to be accessed in a single clock cycle.

Given the following code:

```c
void foo (...) {
    int array1[N] __attribute__((xcl_array_reshape(block, 2, 1)));
    int array2[N] __attribute__((xcl_array_reshape(cycle, 2, 1)));
    int array3[N] __attribute__((xcl_array_reshape(complete, 1)));
    ...
}
```

The ARRAY_RESHAPE attribute transforms the arrays into the form shown in the following figure.

**Figure 129: ARRAY_RESHAPE**

<table>
<thead>
<tr>
<th>Array</th>
<th>Block</th>
<th>Cyclic</th>
<th>Complete</th>
</tr>
</thead>
<tbody>
<tr>
<td>array1[N]</td>
<td>0 1 2 ... N-3 N-2 N-1</td>
<td>block MSB LSB</td>
<td>array4[N/2] N/2 ... N-2 N-1</td>
</tr>
<tr>
<td>array2[N]</td>
<td>0 1 2 ... N-3 N-2 N-1</td>
<td>cyclic MSB LSB</td>
<td>array5[N/2] 1 ... N-3 N-1</td>
</tr>
<tr>
<td>array3[N]</td>
<td>0 1 2 ... N-3 N-2 N-1</td>
<td>complete MSB LSB</td>
<td>array6[1]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>... 1 0</td>
</tr>
</tbody>
</table>

**Syntax**

Place the attribute with the definition of the array variable.

```c
__attribute__((xcl_array_reshape(<type>,<factor>,
<dimension>)))
```

Where:

- `<type>`: Specifies one of the following partition types:
  - cyclic: Cyclic partitioning is the implementation of an array as a set of smaller physical memories that can be accessed simultaneously by the logic in the compute unit. The array is partitioned cyclically by putting one element into each memory before coming back to the first memory to repeat the cycle until the array is fully partitioned.
• **block**: Block partitioning is the physical implementation of an array as a set of smaller memories that can be accessed simultaneously by the logic inside the compute unit. In this case, each memory block is filled with elements from the array before moving on to the next memory.

• **complete**: Complete partitioning decomposes the array into individual elements. For a one-dimensional array, this corresponds to resolving a memory into individual registers. The default `<type>` is complete.

• `<factor>`: For cyclic type partitioning, the `<factor>` specifies how many physical memories to partition the original array into in the kernel code. For Block type partitioning, the `<factor>` specifies the number of elements from the original array to store in each physical memory.

**IMPORTANT!** For complete type partitioning, the `<factor>` should not be specified.

• `<dimension>`: Specifies which array dimension to partition. Specified as an integer from 1 to `<N>`. The Vitis core development kit supports arrays of `<N>` dimensions and can partition the array on any single dimension.

### Example 1
Reshapes (partition and maps) an 8-bit array with 17 elements, AB[17], into a new 32-bit array with five elements using block mapping.

```c
int AB[17] __attribute__((xcl_array_reshape(block,4,1)));
```

**TIP:** A `<factor>` of 4 indicates that the array should be divided into four. As a result, the 17 elements are reshaped into an array of five elements, with four times the bit-width. In this case, the last element, AB[17], is mapped to the lower eight bits of the fifth element, and the rest of the fifth element is empty.

### Example 2
Reshapes the two-dimensional array AB[6][4] into a new array of dimension [6][2], in which dimension 2 has twice the bit-width:

```c
int AB[6][4] __attribute__((xcl_array_reshape(block,2,2)));
```

### Example 3
Reshapes the three-dimensional 8-bit array, AB[4][2][2] in function foo, into a new single element array (a register), 128-bits wide (4×2×2×8):

```c
int AB[4][2][2] __attribute__((xcl_array_reshape(complete,0)));
```

**TIP:** A `<dimension>` of 0 means to reshape all dimensions of the array.
See Also

- xcl_array_partition
- pragma HLS array_reshape
- Vitis HLS Flow

xcl_dataflow

Description

Enables task-level pipelining, allowing functions and loops to overlap in their operation, increasing the concurrency of the RTL implementation, and increasing the overall throughput of the design.

All operations are performed sequentially in a C description. In the absence of any directives that limit resources, such as `pragma HLS allocation`, the Vitis HLS tool seeks to minimize latency and improve concurrency. However, data dependencies can limit this. For example, functions or loops that access arrays must finish all read/write accesses to the arrays before they complete. This prevents the next function or loop that consumes the data from starting operation. The dataflow optimization enables the operations in a function or loop to start operation before the previous function or loop completes all its operations.

When dataflow optimization is specified, the HLS tool analyzes the dataflow between sequential functions or loops and creates channels (based on ping-pong RAMs or FIFOs) that allow consumer functions or loops to start operation before the producer functions or loops have completed. This allows functions or loops to operate in parallel, which decreases latency and improves the throughput of the RTL.

If no initiation interval (number of cycles between the start of one function or loop and the next) is specified, the HLS tool attempts to minimize the initiation interval and start operation as soon as data is available.

TIP: The HLS tool provides dataflow configuration settings. The `config_dataflow` command specifies the default memory channel and FIFO depth used in dataflow optimization.

For the DATAFLOW optimization to work, the data must flow through the design from one task to the next. The following coding styles prevent the HLS tool from performing the DATAFLOW optimization:

- Single-producer-consumer violations
- Bypassing tasks
- Feedback between tasks
- Conditional execution of tasks
- Loops with multiple exit conditions
IMPORTANT! If any of these coding styles are present, the HLS tool issues a message and does not perform DATAFLOW optimization.

Finally, the DATAFLOW optimization has no hierarchical implementation. If a sub-function or loop contains additional tasks that might benefit from the DATAFLOW optimization, you must apply the optimization to the loop, the sub-function, or inline the sub-function.

Syntax

Assign the XCL_DATAFLOW attribute before the function definition or the loop definition:

```c
__attribute__((xcl_dataflow))
```

Examples

Specifies dataflow optimization within function `foo`.

```c
__attribute__((xcl_dataflow))
void foo ( a, b, c, d ) {
    ...
}
```

See Also

- `pragma HLS dataflow`
- `Vitis HLS Flow`

**xcl_latency**

Description

The XCL_LATENCY attribute specifies a minimum, or maximum latency value, or both, for the completion of functions, loops, and regions. Latency is defined as the number of clock cycles required to produce an output. Function or region latency is the number of clock cycles required for the code to compute all output values, and return. Loop latency is the number of cycles to execute all iterations of the loop. See "Performance Metrics Example" of Vitis High-Level Synthesis User Guide (UG1399).

The Vitis HLS tool always tries to minimize latency in the design. When the XCL_LATENCY attribute is specified, the tool behavior is as follows:

- When latency is greater than the minimum, or less than the maximum: The constraint is satisfied. No further optimizations are performed.
- When latency is less than the minimum: If the HLS tool can achieve less than the minimum specified latency, it extends the latency to the specified value, potentially increasing sharing.
• When latency is greater than the maximum: If the HLS tool cannot schedule within the maximum limit, it increases effort to achieve the specified constraint. If it still fails to meet the maximum latency, it issues a warning, and produces a design with the smallest achievable latency in excess of the maximum.

**TIP:** You can also use the XCL_LATENCY attribute to limit the efforts of the tool to find a optimum solution. Specifying latency constraints for scopes within the code: loops, functions, or regions, reduces the possible solutions within that scope and improves tool runtime. For more information, refer to “Improving Synthesis Runtime and Capacity” of Vitis High-Level Synthesis User Guide (UG1399).

**Syntax**

Assign the XCL_LATENCY attribute before the body of the function, loop, or region:

```c
__attribute__((xcl_latency(min, max)))
```

Where:

- `<min>`: Specifies the minimum latency for the function, loop, or region of code.
- `<max>`: Specifies the maximum latency for the function, loop, or region of code.

**Examples**

The `for` loop in the `test` function is specified to have a minimum latency of 4 and a maximum latency of 8.

```c
__kernel void test(__global float *A, __global float *B, __global float *C, int id)
{
    for (unsigned int i = 0; i < id; i++)
        __attribute__((xcl_latency(4, 12)))
            C[id] = A[id] * B[id];
}
```

**See Also**

- `pragma HLS latency`
- `Vitis HLS Flow`

**xcl_loop_tripcount**

**Description**

The XCL_LOOP_TRIPCOUNT attribute can be applied to a loop to manually specify the total number of iterations performed by the loop.
**IMPORTANT!** The XCL_LOOP_TRIPCOUNT attribute is for analysis only, and does not impact the results of synthesis.

The Vivado High-Level Synthesis (HLS) reports the total latency of each loop, which is the number of clock cycles to execute all iterations of the loop. The loop latency is therefore a function of the number of loop iterations, or tripcount.

The tripcount can be a constant value. It can depend on the value of variables used in the loop expression (for example, x<y), or depend on control statements used inside the loop. In some cases, the HLS tool cannot determine the tripcount, and the latency is unknown. This includes cases in which the variables used to determine the tripcount are:

- Input arguments, or
- Variables calculated by dynamic operation.

In cases where the loop latency is unknown or cannot be calculated, the XCL_LOOP_TRIPCOUNT attribute lets you specify minimum, maximum, and average iterations for a loop. This lets the tool analyze how the loop latency contributes to the total design latency in the reports, and helps you determine appropriate optimizations for the design.

**Syntax**

Place the attribute in the OpenCL source before the loop declaration.

```opencl
__attribute__((xcl_loop_tripcount(<min>, <max>, <average>)))
```

Where:

- `<min>`: Specifies the minimum number of loop iterations.
- `<max>`: Specifies the maximum number of loop iterations.
- `<avg>`: Specifies the average number of loop iterations.

**Examples**

In this example, the WHILE loop in function `f` is specified to have a minimum tripcount of 2, a maximum tripcount of 64, and an average tripcount of 33.

```opencl
__kernel void f(__global int *a) {
    unsigned i = 0;
    __attribute__((xcl_loop_tripcount(2, 64, 33)))
    while(i < 64) {
        a[i] = i;
        i++;
    }
}
```
See Also

- pragma HLS occurrence
- Vitis HLS Flow

**xcl_max_work_group_size**

**Description**

Use this attribute instead of REQD_WORK_GROUP_SIZE when you need to specify a larger kernel than the 4K size.

Extends the default maximum work group size supported in the Vitis core development kit by the reqd_work_group_size attribute. Vitis core development kit supports work size larger than 4096 with the XCL_MAX_WORK_GROUP_SIZE attribute.

**Note**: The actual workgroup size limit is dependent on the Xilinx device selected for the platform.

**Syntax**

Place this attribute before the kernel definition, or before the primary function specified for the kernel:

```c
__attribute__((xcl_max_work_group_size(<X>, <Y>, <Z>)))
```

Where:

- `<X>, <Y>, <Z>`: Specifies the ND range of the kernel. This represents each dimension of a three dimensional matrix specifying the size of the work-group for the kernel.

**Examples**

Below is the kernel source code for an un-optimized adder. No attributes were specified for this design, other than the work size equal to the size of the matrices (for example, 64x64). That is, iterating over an entire workgroup will fully add the input matrices, a and b, and output the result. All three are global integer pointers, which means each value in the matrices is four bytes, and is stored in off-chip DDR global memory.

```c
#define RANK 64
__kernel __attribute__((reqd_work_group_size(RANK, RANK, 1)))
void madd(__global int* a, __global int* b, __global int* output) {
    int index = get_local_id(1)*get_local_size(0) + get_local_id(0);
    output[index] = a[index] + b[index];
}
```

This local work size of (64, 64, 1) is the same as the global work size. This setting creates a total work size of 4096.
Note: This is the largest work size that Vitis core development kit supports with the standard OpenCL attribute `REQD_WORK_GROUP_SIZE`. Vitis core development kit supports work size larger than 4096 with the Xilinx attribute `xcl_max_work_group_size`.

Any matrix larger than 64x64 would need to only use one dimension to define the work size. That is, a 128x128 matrix could be operated on by a kernel with a work size of (128, 1, 1), where each invocation operates on an entire row or column of data.

See Also

- https://www.khronos.org/
- The OpenCL C Specification

xcl_pipeline_loop

Description

You can pipeline a loop to improve latency and maximize kernel throughput and performance.

Although unrolling loops increases concurrency, it does not address the issue of keeping all elements in a kernel data path busy at all times. Even in an unrolled case, loop control dependencies can lead to sequential behavior. The sequential behavior of operations results in idle hardware and a loss of performance.

Xilinx addresses this issue by introducing a vendor extension on top of the OpenCL 2.0 API specification for loop pipelining using the XCL_PIPELINE_LOOP attribute.

By default, the v++ compiler automatically pipelines loops with a trip count more than 64, or unrolls loops with a trip count less than 64. This should provide good results. However, you can choose to pipeline loops (instead of the automatic unrolling) by explicitly specifying the NOUNROLL attribute and XCL_PIPELINE_LOOP attribute before the loop.

Syntax

Place the attribute in the OpenCL source before the loop definition:

```c
__attribute__((xcl_pipeline_loop(<II_number>)))
```

Where:

- `<II_number>`: Specifies the desired initiation interval (II) for the pipeline. The Vitis HLS tool tries to meet this request; however, based on data dependencies, the loop might have a larger initiation interval. When the II is not specified, the default is 1.
Examples

The following example specifies an II target of 3 for the for loop in the specified function:

```c
__kernel void f(__global int *a) {
    __attribute__((xcl_pipeline_loop(3)))
    for (unsigned i = 0; i < 64; ++i)
        a[i] = i;
}
```

See Also

- pragma HLS pipeline
- Vitis HLS Flow

**xcl_pipeline_workitems**

**Description**

Pipeline a work item to improve latency and throughput. Work item pipelining is the extension of loop pipelining to the kernel work group. This is necessary for maximizing kernel throughput and performance.

**Syntax**

Place the attribute in the OpenCL API source before the elements to pipeline:

```c
__attribute__((xcl_pipeline_workitems))
```

**Example 1**

To handle the reqd_work_group_size attribute in the following example, Vitis technology automatically inserts a loop nest to handle the three-dimensional characteristics of the ND range (3,1,1). As a result of the added loop nest, the execution profile of this kernel is like an unpipelined loop. Adding the XCL_PIPELINE_WORKITEMS attribute adds concurrency and improves the throughput of the code.

```c
__kernel __attribute__((reqd_work_group_size(3,1,1)))
void foo(...) {
    ...
    __attribute__((xcl_pipeline_workitems)) {
        int tid = get_global_id(0);
        op_Read(tid);
        op_Compute(tid);
        op_Write(tid);
    }
    ...
}
```
Example 2

The following example adds the work-item pipeline to the appropriate elements of the kernel:

```c
__kernel __attribute__ ((reqd_work_group_size(8, 8, 1)))
void madd(__global int* a, __global int* b, __global int* output)
{
    int rank = get_local_size(0);
    __local unsigned int bufa[64];
    __local unsigned int bufb[64];
    __attribute__((xcl_pipeline_workitems)) {
        int x = get_local_id(0);
        int y = get_local_id(1);
        bufa[x*rank + y] = a[x*rank + y];
        bufb[x*rank + y] = b[x*rank + y];
    }
    barrier(CLK_LOCAL_MEM_FENCE);
    __attribute__((xcl_pipeline_workitems)) {
        int index = get_local_id(1)*rank + get_local_id(0);
        output[index] = bufa[index] + bufb[index];
    }
}
```

See Also

- pragma HLS pipeline
- Vitis HLS Flow

**xcl_reqd_pipe_depth**

**Description**

**IMPORTANT!** Pipes must be declared in lower case alphanumerics. `printf()` is also not supported with variables used in pipes.

The OpenCL framework 2.0 specification introduces a new memory object called pipe. A pipe stores data organized as a FIFO. Pipes can be used to stream data from one kernel to another inside the FPGA without using the external memory, which greatly improves the overall system latency.

In the Vitis core development kit, pipes must be statically defined outside of all kernel functions. The depth of a pipe must be specified by using the XCL_REQD_PIPE_DEPTH attribute in the pipe declaration:

```c
pipe int p0 __attribute__((xcl_reqd_pipe_depth(512)));
```

Pipes can only be accessed using standard OpenCL `read_pipe()` and `write_pipe()` built-in functions in non-blocking mode, or using Xilinx-extended `read_pipe_block()` and `write_pipe_block()` functions in blocking mode.
IMPORTANT! Vitis HLS only supports blocking mode for reading and writing pipes. The non-block `read_pipe/write_block` functions are not supported.

IMPORTANT! A given pipe can have one and only one producer and consumer in different kernels.

Pipe objects are not accessible from the host CPU. The status of pipes can be queried using OpenCL `get_pipe_num_packets()` and `get_pipe_max_packets()` built-in functions. For more details on these built-in functions, see *The OpenCL C Specification* from Khronos OpenCL Working Group.

Syntax

This attribute must be assigned at the declaration of the pipe object:

```
pipe int <id> __attribute__((xcl_reqd_pipe_depth(<n>)));
```

Where:

- `<id>`: Specifies an identifier for the pipe, which must consist of lower-case alphanumeric characters. For example, `<infifo1>` not `<inFifo1>`.
- `<n>`: Specifies the depth of the pipe. Valid depth values are 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768.

Examples

The following is the `dataflow_pipes_ocl` example from Xilinx GitHub that use pipes to pass data from one processing stage to the next using blocking `read_pipe_block()` and `write_pipe_block()` functions:

```c
pipe int p0 __attribute__((xcl_reqd_pipe_depth(32)));
pipe int p1 __attribute__((xcl_reqd_pipe_depth(32)));
// Input Stage Kernel: Read Data from Global Memory and write into Pipe P0
kernel __attribute__((reqd_work_group_size(1, 1, 1)))
void input_stage(__global int *input, int size)
{
  __attribute__((xcl_pipeline_loop))
  mem_rd: for (int i = 0 ; i < size ; i++)
  {
    //blocking Write command to pipe P0
    write_pipe_block(p0, &input[i]);
  }
}
// Adder Stage Kernel: Read Input data from Pipe P0 and write the result
// into Pipe P1
kernel __attribute__((reqd_work_group_size(1, 1, 1)))
void adder_stage(int inc, int size)
{
  __attribute__((xcl_pipeline_loop))
  execute: for(int i = 0 ; i < size ; i++)
  {
    int input_data, output_data;
    //blocking read command to Pipe P0
```
read_pipe_block(p0, &input_data);
output_data = input_data + inc;
//blocking write command to Pipe P1
write_pipe_block(p1, &output_data);
}
]
// Output Stage Kernel: Read result from Pipe P1 and write the result to
// Global Memory
kernel __attribute__((reqd_work_group_size(1, 1, 1)))
void output_stage(__global int *output, int size)
{
__attribute__((xcl_pipeline_loop))
mem_wr: for (int i = 0 ; i < size ; i++)
{
//blocking read command to Pipe P1
read_pipe_block(p1, &output[i]);
}
}
}

See Also

- [https://www.khronos.org/](https://www.khronos.org/)
- *The OpenCL C Specification*

**xcl_zero_global_work_offset**

**Description**

If you use `clEnqueueNDRangeKernel` with the `global_work_offset` set to NULL or all zeros, use this attribute to tell the compiler that the `global_work_offset` is always zero.

This attribute can improve memory performance when you have memory accesses like:

\[
A[get_global_id(x)] = \ldots;
\]

**Note:** You can specify `REQUED_WORK_GROUP_SIZE`, `VEC_TYPE_HINT`, and `XCL_ZERO_GLOBAL_WORK_OFFSET` together to maximize performance.

**Syntax**

Place this attribute before the kernel definition or the primary function specified for the kernel.

```c
__kernel __attribute__((xcl_zero_global_work_offset))
void test(__global short *input, __global short *output, __constant short *constants) [ ]
```

See Also

- `reqd_work_group_size`
- `vec_type_hint`
- `clEnqueueNDRangeKernel`
Migrating to a New Target Platform

This migration content is intended for users who need to migrate their accelerated Vitis™ technology application from one target platform to another. For example, moving an application from an Alveo™ U200 Data Center accelerator card, to an Alveo U280 card.

The following sections are included:

- **Design Migration**: An overview of the Design Migration Process including the physical aspects of FPGA devices.
- **Migrating Releases**: Any changes to the host code and design constraints if a new release is used.
- **Modifying Kernel Placement**: Controlling kernel placements and DDR interface connections.
- **Address Timing**: Timing issues in the new target platform which might require additional options to achieve performance.

Design Migration

When migrating an application implemented in one target platform to another, it is important to understand the differences between the target platforms and the impact those differences have on the design.

Key considerations:

- Is there a change in the release?
- Does the new target platform contain a different target platform?
- Do the kernels need to be redistributed across the Super Logic Regions (SLRs)?
- Does the design meet the required frequency (timing) performance in the new platform?

The following diagram summarizes the migration flow described in this guide and the topics to consider during the migration process.
Figure 130: Target Platform Migration Flowchart

IMPORTANT! Before starting to migrate a design, it is important to understand the architecture of an FPGA and the target platform.

Understanding an FPGA Architecture

Before migrating any design to a new target platform, you should have a fundamental understanding of the FPGA architecture. The following diagram shows the floorplan of a Xilinx® FPGA device. The concepts to understand are:

- SSI Devices
- SLRs
- SLR routing resources
- Memory interfaces
Figure 131: Physical View of Xilinx FPGA with Four SLR Regions

TIP: The FPGA floorplan shown above is for a SSI device with four SLRs where each SLR contains a DDR Memory interface.

Stacked Silicon Interconnect Devices

A SSI device is one in which multiple silicon dies are connected together through silicon interconnect, and packaged into a single device. An SSI device enables high-bandwidth connectivity between multiple die by providing a much greater number of connections. It also imposes much lower latency and consumes dramatically lower power than either a multiple FPGA or a multi-chip module approach, while enabling the integration of massive quantities of interconnect logic, transceivers, and on-chip resources within a single package. The advantages of SSI devices are detailed in Xilinx Stacked Silicon Interconnect Technology Delivers Breakthrough FPGA Capacity, Bandwidth, and Power Efficiency (WP380).
Super Logic Region

An SLR is a single FPGA die slice contained in an SSI device. Multiple SLR components are assembled to make up an SSI device. Each SLR contains the active circuitry common to most Xilinx FPGA devices. This circuitry includes large numbers of:

- LUTs
- Registers
- I/O Components
- Gigabit Transceivers
- Block Memory
- DSP Blocks

One or more kernels can be implemented within an SLR. A single kernel cannot be implemented across multiple SLRs.

SLR Routing Resources

The custom hardware implemented on the FPGA is connected via on-chip routing resources. There are two types of routing resources in an SSI device:

- **Intra-SLR Resources**: Intra-SLR routing resource are the fast resources used to connect the hardware logic. The Vitis technology automatically uses the most optimal resources to connect the hardware elements when implementing kernels.

- **Super Long Line (SLL) Resources**: SLLs are routing resources running between SLRs, used to connect logic from one region to the next. These routing resources are slower than intra-SLR routes. However, when a kernel is placed in one SLR, and the DDR it connects to is in another, the Vitis technology automatically implements dedicated hardware to use SLL routing resources without any impact to performance. More information on managing placement are provided in Modifying Kernel Placement.

Memory Interfaces

Each SLR contains one or more memory interfaces. These memory interfaces are used to connect to the DDR memory where the data in the host buffers is copied before kernel execution. Each kernel will read data from the DDR memory and write the results back to the same DDR memory. The memory interface connects to the pins on the FPGA and includes the memory controller logic.

Understanding Target Platforms

In the Vitis technology, a target platform is the hardware design that is implemented onto the FPGA before any custom logic, or accelerators are added. The target platform defines the attributes of the FPGA and is composed of two regions:
• Static region which contains kernel and device management logic.
• Dynamic region where the custom logic of the accelerated kernels is placed.

The figure below shows an FPGA with the target platform applied.

*Figure 132: Target Platform on an FPGA with Four SLR Regions*

The target platform, which is a static region that cannot be modified, contains the logic required to operate the FPGA, and transfer data to and from the dynamic region. The static region, shown above in gray, might exist within a single SLR, or as in the above example, might span multiple SLRs. The static region contains:

• DDR memory interface controllers
• PCIe® interface logic
• XDMA logic
• Firewall logic, etc.
The dynamic region is the area shown in white above. This region contains all the reconfigurable components of the target platform and is the region where all the accelerator kernels are placed.

Because the static region consumes some of the hardware resources available on the device, the custom logic to be implemented in the dynamic region can only use the remaining resources. In the example shown above, the target platform defines that all four DDR memory interfaces on the FPGA can be used. This will require resources for the memory controller used in the DDR interface.

Details on how much logic can be implemented in the dynamic region of each target platform is provided in the Vitis Software Platform Release Notes. This topic is also addressed in Modifying Kernel Placement.

Migrating Releases

Before migrating to a new target platform, you should also determine if you will need to target the new platform to a different release of the Vitis technology. If you intend to target a new release, Xilinx highly recommends to first target the existing platform using the new software release to confirm there are no changes required, and then migrate to a new target platform.

There are two steps to follow when targeting a new release with an existing platform:

- Host Code Migration
- Release Migration

**IMPORTANT! Before migrating to a new release, Xilinx recommends that you review the Vitis Software Platform Release Notes.**

**Host Code Migration**

The **XILINX_XRT** environment variable is used to specify the location of the XRT library environment and must be set before you compile the host code. When the XRT library environment has been installed, the **XILINX_XRT** environment variable can be set by sourcing the `/opt/xilinx/xrt/setup.csh`, or `/opt/xilinx/xrt/setup.sh` file as appropriate. Secondly, ensure that your **LD_LIBRARY_PATH** variable also points to the XRT library installation area.

To compile and run the host code, source the `<INSTALL_DIR>/settings64.csh` or `<INSTALL_DIR>/settings64.sh` file from the Vitis installation.

If you are using the GUI, it will automatically incorporate the new XRT library location and generate the `makefile` when you build your project.
However, if you are using your own custom makefile, you must use the XILINX_XRT environment variable to set up the XRT library.

- Include directories are now specified as: `-I${XILINX_XRT}/include` and `-I${XILINX_XRT}/include/CL`
- Library path is now: `-L${XILINX_XRT}/lib`
- OpenCL library will be: `libxilinxopencl.so`, use `-lxilinxopencl` in your makefile

## Release Migration

After migrating the host code, build the code on the existing target platform using the new release of the Vitis technology. Verify that you can run the project in the Vitis unified software platform using the new release, ensure it completes successfully, and meets the timing requirements.

Issues which can occur when using a new release are:

- Changes to C libraries or library files.
- Changes to kernel path names.
- Changes to the HLS pragmas or pragma options embedded in the kernel code.
- Changes to C/C++/OpenCL compiler support.
- Changes to the performance of kernels: this might require adjustments to the pragmas in the existing kernel code.

Address these issues using the same techniques you would use during the development of any kernel. At this stage, ensure the throughput performance of the target platform using the new release meets your requirements. If there are changes to the final timing (the maximum clock frequency), you can address these when you have moved to the new target platform. This is covered in Address Timing.

## Modifying Kernel Placement

The primary issue when targeting a new platform is ensuring that an existing kernel placement will work in the new target platform. Each target platform has an FPGA defined by a static region. As shown in the figure below, the target platform(s) can be different.

- The target platform on the left has four SLRs, and the static region is spread across all four SLRs.
- The target platform on the right has only three SLRs, and the static region is fully-contained in SLR1.
This section explains how to modify the placement of the kernels.

Implications of a New Hardware Platform

The figure below highlights the issue of kernel placement when migrating to a new target platform. In the example below:

- Existing kernel, kernel_B, is too large to fit into SLR2 of the new target platform because most of the SLR is consumed by the static region.
- The existing kernel, kernel_D, must be relocated to a new SLR because the new target platform does not have four SLRs like the existing platform.
When migrating to a new platform, you need to take the following actions:

- Understand the resources available in each SLR of the new target platform, as documented in the Vitis Software Platform Release Notes.
- Understand the resources required by each kernel in the design.
- Use the `v++ --config` option to specify which SLR each kernel is placed in, and which DDR bank each kernel connects to. For more details, refer to Assigning Compute Units to SLRs and Mapping Kernel Ports to Memory.

These items are addressed in the remainder of this section.

### Determining Where to Place the Kernels

To determine where to place kernels, two pieces of information are required:
• Resources available in each SLR of the hardware platform (.xsa).
• Resources required for each kernel.

With these two pieces of information you will then determine which kernel or kernels can be placed in each SLR of the target platform.

Keep in mind when performing these calculation that 10% of the available resources can be used by system infrastructure:

• Infrastructure logic can be used to connect a kernel to a DDR interface if it has to cross an SLR boundary.
• In an FPGA, resources are also used for signal routing. It is never possible to use 100% of all available resources in an FPGA because signal routing also requires resources.

Available SLR Resources

The resources available in each SLR of the various platforms supported by a release can be found in the Vitis Software Platform Release Notes. The table shows an example target platform. In this example:

• SLR description indicates which SLR contains static and/or dynamic regions.
• Resources available in each SLR (LUTs, Registers, RAM, etc.) are listed.

This allows you to determine what resources are available in each SLR.

Table 94: SLR Resources of a Hardware Platform

<table>
<thead>
<tr>
<th>Area</th>
<th>SLR 0</th>
<th>SLR 1</th>
<th>SLR 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>SLR description</td>
<td>Bottom of device; dedicated to dynamic region.</td>
<td>Middle of device; shared by dynamic and static region resources.</td>
<td>Top of device; dedicated to dynamic region.</td>
</tr>
<tr>
<td>Dynamic region Pblock name</td>
<td>pfa_top_i_dynamic_region_pblock_dynamc_SLR0</td>
<td>pfa_top_i_dynamic_region_pblock_dynamc_SLR1</td>
<td>pfa_top_i_dynamic_region_pblock_dynamc_SLR2</td>
</tr>
<tr>
<td>Compute unit placement syntax</td>
<td>set_property CONFIG.SLR_ASSIGNMENTS SLR0[get_bd_cells&lt;cu_name&gt;]</td>
<td>set_property CONFIG.SLR_ASSIGNMENTS SLR1[get_bd_cells&lt;cu_name&gt;]</td>
<td>set_property CONFIG.SLR_ASSIGNMENTS SLR2[get_bd_cells&lt;cu_name&gt;]</td>
</tr>
</tbody>
</table>

Global memory resources available in dynamic region

<table>
<thead>
<tr>
<th>Memory channels; system port name</th>
<th>bank0 (16 GB DDR4)</th>
<th>bank1 (16 GB DDR4, in static region)</th>
<th>bank2 (16 GB DDR4, in dynamic region)</th>
<th>bank3 (16 GB DDR4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLB LUT</td>
<td>388K</td>
<td>199K</td>
<td>388K</td>
<td></td>
</tr>
<tr>
<td>CLB Register</td>
<td>776K</td>
<td>399K</td>
<td>776K</td>
<td></td>
</tr>
<tr>
<td>Block RAM Tile</td>
<td>720</td>
<td>420</td>
<td>720</td>
<td></td>
</tr>
<tr>
<td>UltraRAM</td>
<td>320</td>
<td>160</td>
<td>320</td>
<td></td>
</tr>
<tr>
<td>DSP</td>
<td>2280</td>
<td>1320</td>
<td>2280</td>
<td></td>
</tr>
</tbody>
</table>
Kernel Resources

The resources for each kernel can be obtained from the System Estimate report.

The System Estimate report is available in the Assistant view after either the Hardware Emulation or Hardware run are complete. An example of this report is shown below.

Figure 135: System Estimate Report

<table>
<thead>
<tr>
<th>Area Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compute Unit</td>
</tr>
<tr>
<td>smithwaterman_1</td>
</tr>
</tbody>
</table>

- FF refers to the CLB Registers noted in the platform resources for each SLR.
- LUT refers to the CLB LUTs noted in the platform resources for each SLR.
- DSP refers to the DSPs noted in the platform resources for each SLR.
- BRAM refers to the block RAM Tile noted in the platform resources for each SLR.

This information can help you determine the proper SLR assignments for each kernel.

Assigning Kernels to SLRs

Each kernel in a design can be assigned to an SLR region using the `connectivity.slr` option in a configuration file specified for the `v++ --config` command line option. Refer to Assigning Compute Units to SLRs for more information.

When placing kernels, Xilinx recommends assigning the specific DDR memory bank that the kernel will connect to using the `connectivity.sp config` option as described in Mapping Kernel Ports to Memory.

For example, the figure below shows an existing target platform that has four SLRs, and a new target platform with three SLRs. The static region is also structured differently between the two platforms. In this migration example:

- Kernel_A is mapped to SLR0.
- Kernel_B, which no longer fits in SLR1, is remapped to SLR0, where there are available resources.
- Kernel_C is mapped to SLR2.
- Kernel_D is remapped to SLR2, where there are available resources.

The kernel mappings are illustrated in the figure below.
Specifying Kernel Placement

For the example above, the configuration file to assign the kernels would be similar to the following:

```
[connectivity]
nk=kernel:4:kernel_A.kernel_B.kernel_C.kernel_D

slr=kernel_A:SLR0
slr=kernel_B:SLR0
slr=kernel_C:SLR2
slr=kernel_D:SLR2
```

The `v++` command line to place each of the kernels as shown in the figure above would be:

```
v++ -l --config config.cfg ...
```
Specifying Kernel DDR Interfaces

You should also specify the kernel DDR memory interface when specifying kernel placements. Specifying the DDR interface ensures the automatic pipelining of kernel connections to a DDR interface in a different SLR. This ensures there is no degradation in timing which can reduce the maximum clock frequency.

In this example, using the kernel placements in the above figure:

- Kernel_A is connected to Memory Bank 0.
- Kernel_B is connected to Memory Bank 1.
- Kernel_C is connected to Memory Bank 2.
- Kernel_D is connected to Memory Bank 1.

The configuration file to perform these connections would be as follows, and passed through the `v++ --config` command:

```text
[connectivity]
k=kernel:4:kernel_A.kernel_B.kernel_C.kernel_D
slr=kernel_A:SLR0
slr=kernel_B:SLR0
slr=kernel_C:SLR2
slr=kernel_D:SLR2
sp=kernel_A.arg1:DDR[0]
sp=kernel_B.arg1:DDR[1]
sp=kernel_C.arg1:DDR[2]
sp=kernel_D.arg1:DDR[1]
```

**IMPORTANT!** When using the `connectivity.sp` option to assign kernel ports to memory banks, you must map all interfaces/ports of the kernel. Refer to Mapping Kernel Ports to Memory for more information.

Address Timing

Perform a system run and if it completes with no violations, then the migration is successful.

If timing has not been met you might need to specify some custom constraints to help meet timing. Refer to *UltraFast Design Methodology Timing Closure Quick Reference Guide (UG1292)* for more information on meeting timing.
Custom Constraints

Custom Tcl constraints for floorplanning, placement, and timing of the kernels will need to be reviewed in the context of the new target platform (.xsa). For example, if a kernel needs to be moved to a different SLR in the new target platform, the placement constraints for that kernel will also need to be modified.

In general, timing is expected to be comparable between different target platforms that are based on the 9P Virtex UltraScale device. Any custom Tcl constraints for timing closure will need to be evaluated and might need to be modified for the new platform.

Custom constraints can be passed to the Vivado® tools using the [advanced] directives of the v++ configuration file specified by the --config option. Refer to Managing Vivado Synthesis and Implementation Results more information.

Timing Closure Considerations

Design performance and timing closure can vary when moving across Vitis releases or target platform(s), especially when one of the following conditions is true:

- Device or SLR resource utilization was higher than the typical guideline:
  - LUT utilization was higher than 70%
  - DSP, RAMB, and UltraRAM utilization was higher than 80%
  - FD utilization was higher than 50%
- High effort compilation strategies were needed to close timing.

The utilization guidelines provide a threshold above which the compilation of the design can take longer, or performance can be lower than initially estimated. For larger designs which usually require using more than one SLR, specify the kernel/DDR association with the v++ --config option, as described in Mapping Kernel Ports to Memory, while verifying that any floorplan constraint ensures the following:

- The utilization of each SLR is below the recommended guidelines.
- The utilization is balanced across SLRs if one type of hardware resource needs to be higher than the guideline.

For designs with overall high utilization, increasing the amount of pipelining in the kernels, at the cost of higher latency, can greatly help timing closure and achieving higher performance.

For quickly reviewing all aspects listed above, use the fail-fast reports generated throughout the Vitis application acceleration development flow using the -R option as described below (refer to Controlling Report Generation for more information):
v++ -R 1

- `report_failfast` is run at the end of each kernel synthesis step
- `report_failfast` is run after `opt_design` on the entire design
- `opt_design` DCP is saved

v++ -R 2

- Same reports as with `-R 1`, plus:
- `report_failfast` is post-placement for each SLR
- Additional reports and intermediate DCPs are generated

All reports and DCPs can be found in the implementation directory, including kernel synthesis reports:

```
<runDir>/_x/link/vivado/prj/prj.runs/impl_1
```

For more information about timing closure and the fail-fast report, see the UltraFast Design Methodology Guide for Xilinx FPGAs and SoCs (UG949).
Section X

Additional Resources and Legal Notices

Xilinx Resources

For support resources such as Answers, Documentation, Downloads, and Forums, see Xilinx Support.

Documentation Navigator and Design Hubs

Xilinx® Documentation Navigator (DocNav) provides access to Xilinx documents, videos, and support resources, which you can filter and search to find information. To open DocNav:

- From the Vivado® IDE, select Help → Documentation and Tutorials.
- On Windows, select Start → All Programs → Xilinx Design Tools → DocNav.
- At the Linux command prompt, enter docnav.

Xilinx Design Hubs provide links to documentation organized by design tasks and other topics, which you can use to learn key concepts and address frequently asked questions. To access the Design Hubs:

- In DocNav, click the Design Hubs View tab.
- On the Xilinx website, see the Design Hubs page.

Note: For more information on DocNav, see the Documentation Navigator page on the Xilinx website.
## Revision History

### Getting Started with Vitis Revision History

The following table shows the revision history for Section I: Getting Started with Vitis.

<table>
<thead>
<tr>
<th>Section</th>
<th>Revision Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>03/22/2021 Version 2020.2</td>
<td></td>
</tr>
<tr>
<td>OpenCL Installable Client Driver Loader</td>
<td>Updated installation information for RHEL/CentOS.</td>
</tr>
<tr>
<td>12/15/2020 Version 2020.2</td>
<td></td>
</tr>
<tr>
<td>Installing Xilinx Runtime and Platforms</td>
<td>Minor update.</td>
</tr>
<tr>
<td>Vitis Software Platform Release Notes</td>
<td>Updated section and subsections.</td>
</tr>
<tr>
<td>Installation</td>
<td>Updated section and subsections.</td>
</tr>
<tr>
<td>Execution Model</td>
<td>Minor update.</td>
</tr>
<tr>
<td>Data Center Application Acceleration Development Flow</td>
<td>Added.</td>
</tr>
<tr>
<td>Embedded Processor Application Acceleration Development Flow</td>
<td>Added.</td>
</tr>
<tr>
<td>Tutorials and Examples</td>
<td>Updated link.</td>
</tr>
<tr>
<td>08/20/2020 Version 2020.1</td>
<td></td>
</tr>
<tr>
<td>Entire section</td>
<td>No updates to this section.</td>
</tr>
<tr>
<td>06/24/2020 Version 2020.1</td>
<td></td>
</tr>
<tr>
<td>Installation Requirements</td>
<td>Updated operating system requirements.</td>
</tr>
<tr>
<td>06/03/2020 Version 2020.1</td>
<td></td>
</tr>
<tr>
<td>General updates</td>
<td>Updated figures and tool commands.</td>
</tr>
</tbody>
</table>

### Developing Applications Revision History

The following table shows the revision history for Section II: Developing Applications.

<table>
<thead>
<tr>
<th>Section</th>
<th>Revision Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>03/22/2021 Version 2020.2</td>
<td></td>
</tr>
<tr>
<td>Entire section</td>
<td>No changes to this section.</td>
</tr>
<tr>
<td>12/15/2020 Version 2020.2</td>
<td></td>
</tr>
<tr>
<td>Entire section</td>
<td>No updates to this section.</td>
</tr>
<tr>
<td>Kernel Properties</td>
<td>Minor update.</td>
</tr>
<tr>
<td>Kernel Execution Modes</td>
<td>Added table.</td>
</tr>
<tr>
<td>Kernel Interfaces</td>
<td>Minor update.</td>
</tr>
<tr>
<td>Host Application</td>
<td>Minor update.</td>
</tr>
</tbody>
</table>
### Section III: Building and Running the Application

#### Revision Summary

<table>
<thead>
<tr>
<th>Section</th>
<th>Revision Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Setting Kernel Arguments</td>
<td>Minor update.</td>
</tr>
<tr>
<td>Buffer Allocation on the Device</td>
<td>Minor update.</td>
</tr>
<tr>
<td>Buffer Creation and Data Transfer</td>
<td>Updated section.</td>
</tr>
<tr>
<td>Letting XRT Allocate Buffers</td>
<td>Updated section.</td>
</tr>
<tr>
<td>Using Host Pointer Buffers</td>
<td>Minor update.</td>
</tr>
<tr>
<td>Allocating Page-Aligned Host Memory</td>
<td>Minor update.</td>
</tr>
<tr>
<td>Kernel Execution</td>
<td>Minor update.</td>
</tr>
<tr>
<td>Summary</td>
<td>Minor update.</td>
</tr>
<tr>
<td>Memory Interface Width Considerations</td>
<td>Minor update.</td>
</tr>
<tr>
<td>Reading and Writing by Burst</td>
<td>Minor update.</td>
</tr>
<tr>
<td>Requirements of an RTL Kernel</td>
<td>Minor update.</td>
</tr>
<tr>
<td>Kernel Interface Requirements</td>
<td>Updated table.</td>
</tr>
<tr>
<td>Kernel Controls</td>
<td>Updated section.</td>
</tr>
<tr>
<td>Interrupt</td>
<td>Updated section.</td>
</tr>
<tr>
<td>RTL Kernel Development Flow</td>
<td>Updated section.</td>
</tr>
<tr>
<td>Package the RTL Code as a Vivado IP</td>
<td>Updated section.</td>
</tr>
<tr>
<td>Creating the XO File from the RTL Kernel</td>
<td>Updated section.</td>
</tr>
</tbody>
</table>

**08/20/2020 Version 2020.1**

No updates to this section.

**06/24/2020 Version 2020.1**

Updated instructions.

**06/03/2020 Version 2020.1**

Added information for supporting the automatic widening of interfaces.

**Building and Running the Application Revision History**

The following table shows the revision history for Section III: Building and Running the Application.

<table>
<thead>
<tr>
<th>Section</th>
<th>Revision Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mapping Kernel Ports to Memory</td>
<td>Updated section.</td>
</tr>
<tr>
<td>HBM Configuration and Use</td>
<td>Added this section.</td>
</tr>
</tbody>
</table>

**03/22/2021 Version 2020.2**
# Revision Summary

<table>
<thead>
<tr>
<th>Section</th>
<th>Revision Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>PLRAM Configuration and Use</td>
<td>Added this section.</td>
</tr>
<tr>
<td>Using the -vivado and -advanced Options</td>
<td>Added information about disabling global buffer insertion during placement.</td>
</tr>
<tr>
<td>RTL Simulator Support</td>
<td>Updated Vivado properties for all simulators.</td>
</tr>
<tr>
<td>Using I/O Traffic Generators</td>
<td>Updated information about running traffic generators.</td>
</tr>
<tr>
<td>12/15/2020 Version 2020.2</td>
<td></td>
</tr>
<tr>
<td>Running Emulation on an Embedded Processor Platform</td>
<td>Updated section.</td>
</tr>
<tr>
<td>Software Emulation</td>
<td>Updated section.</td>
</tr>
<tr>
<td>Hardware Emulation</td>
<td>Updated section.</td>
</tr>
<tr>
<td>System Hardware Target</td>
<td>Minor update.</td>
</tr>
<tr>
<td>Building the Host Program</td>
<td>Minor update.</td>
</tr>
<tr>
<td>Compiling and Linking for x86</td>
<td>Minor update.</td>
</tr>
<tr>
<td>Compiling and Linking for Arm</td>
<td>Minor update.</td>
</tr>
<tr>
<td>Compiling Kernels with the Vitis Compiler</td>
<td>Updated argument naming.</td>
</tr>
<tr>
<td>Linking the Kernels</td>
<td>Updated argument naming.</td>
</tr>
<tr>
<td>Mapping Kernel Ports to Memory</td>
<td>Added Using Slave-Bridge to Connect to Host Memory section.</td>
</tr>
<tr>
<td>Managing Vivado Synthesis and Implementation Results</td>
<td>Split/updated subsections and moved out of Linking. the Kernels</td>
</tr>
<tr>
<td>Packaging the System</td>
<td>Updated section and subsections.</td>
</tr>
<tr>
<td>Output Directories of the v++ Command</td>
<td>Updated section.</td>
</tr>
<tr>
<td>Running Emulation</td>
<td>Split into subsections and added new topics.</td>
</tr>
<tr>
<td>08/20/2020 Version 2020.1</td>
<td></td>
</tr>
<tr>
<td>Entire section</td>
<td>No updates to this section.</td>
</tr>
<tr>
<td>06/24/2020 Version 2020.1</td>
<td></td>
</tr>
<tr>
<td>Entire section</td>
<td>Editorial updates only. No technical content updates.</td>
</tr>
<tr>
<td>06/03/2020 Version 2020.1</td>
<td></td>
</tr>
<tr>
<td>Compiling Kernels with the Vitis HLS</td>
<td>Updated to Vitis HLS.</td>
</tr>
<tr>
<td>Packaging the System</td>
<td>Added chapter.</td>
</tr>
<tr>
<td>Entire section</td>
<td>Updated figures and tool commands.</td>
</tr>
</tbody>
</table>

## Profiling, Optimizing, and Debugging the Application

The following table shows the revision history for Section IV: Profiling, Optimizing, and Debugging the Application.

<table>
<thead>
<tr>
<th>Section</th>
<th>Revision Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>03/22/2021 Version 2020.2</td>
<td></td>
</tr>
<tr>
<td>Entire section</td>
<td>No changes to this section.</td>
</tr>
<tr>
<td>12/15/2020 Version 2020.2</td>
<td></td>
</tr>
<tr>
<td>Entire section</td>
<td>No updates to this section.</td>
</tr>
<tr>
<td>Section</td>
<td>Revision Summary</td>
</tr>
<tr>
<td>--------------------------------------------------------------</td>
<td>-------------------------------------------------------</td>
</tr>
<tr>
<td>Profiling the Application</td>
<td>Minor update.</td>
</tr>
<tr>
<td>Baselining Functionality and Performance</td>
<td>Minor update.</td>
</tr>
<tr>
<td>Enabling Profiling in Your Application</td>
<td>Minor update.</td>
</tr>
<tr>
<td>Custom Profiling of the Host Application</td>
<td>Added subtopics.</td>
</tr>
<tr>
<td>Guidance</td>
<td>Added tip note.</td>
</tr>
<tr>
<td>Opening the System Estimate Report</td>
<td>Minor update.</td>
</tr>
<tr>
<td>HLS Report</td>
<td>Minor update.</td>
</tr>
<tr>
<td>Profile Summary Report</td>
<td>Minor update.</td>
</tr>
<tr>
<td>Generating and Opening the Profile Summary Report</td>
<td>Updated section.</td>
</tr>
<tr>
<td>Interpreting the Profile Summary</td>
<td>Updated section.</td>
</tr>
<tr>
<td>Generating and Opening the Application Timeline</td>
<td>Updated section.</td>
</tr>
<tr>
<td>Interpreting the Application Timeline</td>
<td>Updated Read and Write descriptions.</td>
</tr>
<tr>
<td>Enabling Low Overhead Profiling</td>
<td>Updated section.</td>
</tr>
<tr>
<td>Waveform View and Live Waveform Viewer</td>
<td>Minor update.</td>
</tr>
<tr>
<td>Generating and Opening the Waveform Reports</td>
<td>Minor update.</td>
</tr>
<tr>
<td>Host Optimization</td>
<td>Minor update.</td>
</tr>
<tr>
<td>Reducing Overhead of Kernel Enqueing</td>
<td>Minor update.</td>
</tr>
<tr>
<td>Optimizing Data Movement</td>
<td>Updated section.</td>
</tr>
<tr>
<td>Overlapping Data Transfers with Kernel Computation</td>
<td>Updated event figures.</td>
</tr>
<tr>
<td>Buffer Memory Segmentation</td>
<td>Minor update.</td>
</tr>
<tr>
<td>Multiple In-Order Command Queues</td>
<td>Updated figure.</td>
</tr>
<tr>
<td>Optimizing Kernel Computation</td>
<td>Updated section.</td>
</tr>
<tr>
<td>Assigning DDR Bank in Host Code</td>
<td>Added important note.</td>
</tr>
<tr>
<td>Using Burst Data Transfers</td>
<td>Minor update.</td>
</tr>
<tr>
<td>Optimizing Computational Parallelism</td>
<td>Minor update.</td>
</tr>
<tr>
<td>Debug Techniques for Hardware Emulation</td>
<td>Added.</td>
</tr>
<tr>
<td>Enabling Kernels for Debugging with ChipScope</td>
<td>Updated section.</td>
</tr>
<tr>
<td>System ILA</td>
<td>Removed section.</td>
</tr>
<tr>
<td>Debugging with ChipScope</td>
<td>Minor update.</td>
</tr>
<tr>
<td>Kernel Hangs Due to AXI Violations</td>
<td>Minor update.</td>
</tr>
<tr>
<td>Hardware Debug for Embedded Processors</td>
<td>Minor update.</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>08/20/2020 Version 2020.1</td>
<td>No updates to this section.</td>
</tr>
<tr>
<td>Entire section</td>
<td></td>
</tr>
<tr>
<td>06/24/2020 Version 2020.1</td>
<td>Editorial updates only. No technical content updates.</td>
</tr>
<tr>
<td>Entire section</td>
<td></td>
</tr>
<tr>
<td>06/03/2020 Version 2020.1</td>
<td>Added information emulation debug in embedded processors.</td>
</tr>
</tbody>
</table>
### Section X: Additional Resources and Legal Notices

<table>
<thead>
<tr>
<th>Section</th>
<th>Revision Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hardware Debug for Embedded Processors</td>
<td>Added information hardware debug in embedded processors.</td>
</tr>
<tr>
<td>Enabling Profiling in Your Application</td>
<td>Added section.</td>
</tr>
<tr>
<td>Guidance</td>
<td>Added more information about the types of guidance available.</td>
</tr>
<tr>
<td>HLS Report</td>
<td>Updated for Vitis HLS.</td>
</tr>
<tr>
<td>Profile Summary Report</td>
<td>Added more information about generating and interpreting the profile summary report.</td>
</tr>
<tr>
<td>Enabling Low Overhead Profiling</td>
<td>Added section.</td>
</tr>
<tr>
<td>Enabling Kernels for Debugging with Chipscope</td>
<td>Added information about the AXI Protocol Checker.</td>
</tr>
<tr>
<td>Debugging on Embedded Processor Platforms</td>
<td>Added section.</td>
</tr>
<tr>
<td>General updates</td>
<td>Updated figures and tool commands.</td>
</tr>
</tbody>
</table>

### Vitis Environment Reference Materials Revision History

The following table shows the revision history for Section V: Vitis Environment Reference Materials.

<table>
<thead>
<tr>
<th>Section</th>
<th>Revision Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>---clock Options</td>
<td>Updated Important notice.</td>
</tr>
<tr>
<td>---profile Options</td>
<td>Added --profile.trace_memory.</td>
</tr>
<tr>
<td>---vivado Options</td>
<td>In --vivado.prop option description, changed fanout_opt option to no_bufg_opt.</td>
</tr>
<tr>
<td>12/15/2020 Version 2020.2</td>
<td></td>
</tr>
<tr>
<td>--advanced Options</td>
<td>Updated table</td>
</tr>
<tr>
<td>--hls Options</td>
<td>Minor updates</td>
</tr>
<tr>
<td>flash</td>
<td>Added xbmgmt flash --scan description.</td>
</tr>
<tr>
<td>partition</td>
<td>Added partition command description.</td>
</tr>
<tr>
<td>Vitis Compiler General Options</td>
<td>Added advanced, clock, connectivity, hls, linkhook, package, profile, reuse_bit, and vivado options. Added package description.</td>
</tr>
<tr>
<td>--advanced Options</td>
<td>Updated table</td>
</tr>
<tr>
<td>--clock Options</td>
<td>Updated section.</td>
</tr>
<tr>
<td>--debug Options</td>
<td>Added.</td>
</tr>
<tr>
<td>--hls Options</td>
<td>Updated section.</td>
</tr>
<tr>
<td>--linkhook Options</td>
<td>Updated section.</td>
</tr>
<tr>
<td>--package Options</td>
<td>Minor updates.</td>
</tr>
<tr>
<td>--profile Options</td>
<td>Added.</td>
</tr>
<tr>
<td>--vivado Options</td>
<td>Updated section.</td>
</tr>
<tr>
<td>kernelinfo Utility</td>
<td>Minor updates.</td>
</tr>
<tr>
<td>launch_emulator Utility</td>
<td>Updated section.</td>
</tr>
<tr>
<td>package_xo Command</td>
<td>Updated section.</td>
</tr>
<tr>
<td>platforminfo Utility</td>
<td>Added force and code update.</td>
</tr>
</tbody>
</table>
### Section X: Additional Resources and Legal Notices

<table>
<thead>
<tr>
<th>Section</th>
<th>Revision Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTL Kernel XML File</td>
<td>Title and table update.</td>
</tr>
<tr>
<td>xbutil Utility</td>
<td>Added tip.</td>
</tr>
<tr>
<td>p2p</td>
<td>Minor updates.</td>
</tr>
<tr>
<td>query</td>
<td>Fixed Firewall description.</td>
</tr>
<tr>
<td>status</td>
<td>Minor updates.</td>
</tr>
<tr>
<td>xbmgmt Utility</td>
<td>Added tip.</td>
</tr>
<tr>
<td>config</td>
<td>Minor updates.</td>
</tr>
<tr>
<td>flash</td>
<td>Minor updates.</td>
</tr>
<tr>
<td>partition</td>
<td>Added table.</td>
</tr>
<tr>
<td>xrt.ini File</td>
<td>Minor updates.</td>
</tr>
</tbody>
</table>

#### 08/20/2020 Version 2020.1

Entire section

No updates to this section.

#### 06/24/2020 Version 2020.1

<table>
<thead>
<tr>
<th>Vitis Compiler General Options</th>
<th>Updated the details of the following commands:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>• <code>--custom_script</code></td>
</tr>
<tr>
<td></td>
<td>• <code>--export_script</code></td>
</tr>
<tr>
<td></td>
<td>• <code>--advanced.param</code></td>
</tr>
</tbody>
</table>

#### 06/03/2020 Version 2020.1

| Vitis Compiler Command        | Updated for Vitis HLS.                         |
| Vitis Compiler General Options| Made significant updates to the following options: |
|                               | • `--custom_script`                            |
|                               | • `--dk`                                       |
|                               | • `--from_step`                                |
|                               | • `--no_ip_cache`                              |
|                               | • `--remote_ip_cache`                          |
|                               | • `--reuse_impl`                               |
|                               | • `--to_step`                                  |
|                               | • `--trace_memory`                             |
| `--advanced Options`          | Made significant updates to the following options: |
|                               | • `--advanced.param`                           |
|                               | • `--advanced.prop`                            |
| `--clock Options`             | Added new options:                             |
|                               | • `--clock.defaultTolerance`                   |
|                               | • `--clock.tolerance`                          |
| `--connectivity Options`      | Updated `--connectivity.sc` option             |
| `--linkhook Options`          | Added new section.                             |
| `--package Options`           | Added new section.                             |
| `--vivado Options`            | Made significant updates to the `--vivado.prop` option. |
| `launch_emulator Utility`     | Added new section.                             |
| `manage_ipcache Utility`      | Added new section.                             |
| `platforminfo Utility`        | Added new section.                             |
|                               | Added platforminfo example for embedded processors. |
**Revision Summary**

<table>
<thead>
<tr>
<th>Section</th>
<th>Revision Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>xbutil Utility</td>
<td>Added additional information about the <code>xbutil</code> utility. Made significant updates to all commands in this section.</td>
</tr>
<tr>
<td>xbmgmt Utility</td>
<td>Added additional information about the <code>xbmgmt</code> utility. Made significant updates to all commands in this section.</td>
</tr>
<tr>
<td>xclbinutil Utility</td>
<td>Made significant updates to the following sections:</td>
</tr>
<tr>
<td></td>
<td>• <code>xclbin</code> Information</td>
</tr>
<tr>
<td></td>
<td>• Tool Generation Information</td>
</tr>
<tr>
<td>xrt.ini File</td>
<td>Added the following new keys:</td>
</tr>
<tr>
<td></td>
<td>• Runtime group: <code>ert_polling</code>, <code>exclusive_cu_context</code></td>
</tr>
<tr>
<td></td>
<td>• Debug group: <code>continuous_trace</code>, <code>continuous_trace_interval_ms</code>, <code>lop_trace</code>, <code>power_profile</code></td>
</tr>
<tr>
<td></td>
<td>• Emulation group: <code>debug_mode</code> (replaces <code>launch_waveform</code>, <code>user_pre_sim_script</code>, <code>user_post_sim_script</code>, <code>xtlm_aximm_log</code>, <code>xtlm_axis_log</code>, <code>timeout_scale</code>)</td>
</tr>
<tr>
<td>HLS Pragmas</td>
<td>Added new pragmas to table.</td>
</tr>
<tr>
<td></td>
<td><em>Note</em>: Starting in the 2020.1 release, all pragma descriptions are located in Vitis HLS Flow.</td>
</tr>
</tbody>
</table>

**Using the Vitis Analyzer Revision History**

The following table shows the revision history for Section VI: Using the Vitis Analyzer.

<table>
<thead>
<tr>
<th>Section</th>
<th>Revision Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><strong>03/22/2021 Version 2020.2</strong></td>
</tr>
<tr>
<td>Entire section</td>
<td>No changes to this section.</td>
</tr>
<tr>
<td></td>
<td><strong>12/15/2020 Version 2020.2</strong></td>
</tr>
<tr>
<td>Entire section</td>
<td>No updates to this section.</td>
</tr>
<tr>
<td></td>
<td><strong>11/23/2020 Version 2020.2</strong></td>
</tr>
<tr>
<td>Section VI: Using the Vitis Analyzer</td>
<td>Added Versal AI Engine description.</td>
</tr>
<tr>
<td>Working with Reports</td>
<td>Minor updates.</td>
</tr>
<tr>
<td>Vitis Analyzer GUI and Window Manager</td>
<td>Updated section.</td>
</tr>
<tr>
<td>Platform and System Diagrams</td>
<td>Minor updates.</td>
</tr>
<tr>
<td>AI Engine Graphs and Arrays</td>
<td>Added.</td>
</tr>
<tr>
<td>Configuring the Vitis Analyzer</td>
<td>Updated section.</td>
</tr>
<tr>
<td>Setting Guidance Thresholds</td>
<td>Added.</td>
</tr>
<tr>
<td>Creating an Archive File</td>
<td>Minor updates.</td>
</tr>
<tr>
<td></td>
<td><strong>08/20/2020 Version 2020.1</strong></td>
</tr>
<tr>
<td>Entire section</td>
<td>No updates to this section.</td>
</tr>
<tr>
<td></td>
<td><strong>06/24/2020 Version 2020.1</strong></td>
</tr>
<tr>
<td>Entire section</td>
<td>Editorial updates only. No technical content updates.</td>
</tr>
</tbody>
</table>
Using the Vitis IDE Revision History

The following table shows the revision history for Section VII: Using the Vitis IDE.

<table>
<thead>
<tr>
<th>Section</th>
<th>Revision Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>03/22/2021 Version 2020.2</td>
<td>Entire section</td>
</tr>
<tr>
<td></td>
<td>No changes to this section.</td>
</tr>
<tr>
<td>12/15/2020 Version 2020.2</td>
<td>Managing Platforms and Repositories</td>
</tr>
<tr>
<td></td>
<td>Updated additional tab descriptions.</td>
</tr>
<tr>
<td></td>
<td>Adding Sources</td>
</tr>
<tr>
<td></td>
<td>Minor update.</td>
</tr>
<tr>
<td></td>
<td>Building the System</td>
</tr>
<tr>
<td></td>
<td>Added tip note.</td>
</tr>
<tr>
<td></td>
<td>Vitis Run and Debug Configuration Settings</td>
</tr>
<tr>
<td></td>
<td>Minor update.</td>
</tr>
<tr>
<td>11/23/2020 Version 2020.2</td>
<td>Create an Application Project</td>
</tr>
<tr>
<td></td>
<td>Updated section.</td>
</tr>
<tr>
<td></td>
<td>Understanding the Vitis IDE</td>
</tr>
<tr>
<td></td>
<td>Minor update.</td>
</tr>
<tr>
<td></td>
<td>Adding Sources</td>
</tr>
<tr>
<td></td>
<td>Minor update.</td>
</tr>
<tr>
<td></td>
<td>Working in the Project Editor View</td>
</tr>
<tr>
<td></td>
<td>Updated section.</td>
</tr>
<tr>
<td></td>
<td>Working in the Assistant View</td>
</tr>
<tr>
<td></td>
<td>Updated section.</td>
</tr>
<tr>
<td></td>
<td>Output Directories from the Vitis IDE</td>
</tr>
<tr>
<td></td>
<td>Updated section.</td>
</tr>
<tr>
<td></td>
<td>Configuring the Vitis IDE</td>
</tr>
<tr>
<td></td>
<td>Updated section and subsections.</td>
</tr>
<tr>
<td></td>
<td>Generate RTL Kernel</td>
</tr>
<tr>
<td></td>
<td>Updated section.</td>
</tr>
<tr>
<td>08/20/2020 Version 2020.1</td>
<td>Entire section</td>
</tr>
<tr>
<td></td>
<td>No updates to this section.</td>
</tr>
<tr>
<td>06/24/2020 Version 2020.1</td>
<td>Entire section</td>
</tr>
<tr>
<td></td>
<td>Editorial updates only. No technical content updates.</td>
</tr>
<tr>
<td>06/03/2020 Version 2020.1</td>
<td>Getting Started with Examples</td>
</tr>
<tr>
<td></td>
<td>Added information working with and using Vitis libraries.</td>
</tr>
<tr>
<td></td>
<td>Vitis Command Options</td>
</tr>
<tr>
<td></td>
<td>Added the <code>--debug</code> Command.</td>
</tr>
<tr>
<td></td>
<td>Create an Application Project</td>
</tr>
<tr>
<td></td>
<td>Updated with new information for IDE changes and steps in flow.</td>
</tr>
<tr>
<td></td>
<td>Vitis IDE Debug Flow</td>
</tr>
<tr>
<td></td>
<td>Added new sections:</td>
</tr>
<tr>
<td></td>
<td>• Using the Standalone Debug Flow</td>
</tr>
<tr>
<td></td>
<td>• <code>vitis -debug Command Line</code></td>
</tr>
</tbody>
</table>
Using Vitis Embedded Platforms Revision History

The following table shows the revision history for Section VIII: Using Vitis Embedded Platforms.

<table>
<thead>
<tr>
<th>Section</th>
<th>Revision Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Platform Creation Basics</td>
<td>Section rewritten.</td>
</tr>
<tr>
<td>Adding Hardware Interfaces</td>
<td>Added content to General Requirements.</td>
</tr>
<tr>
<td>12/15/2020 Version 2020.2</td>
<td></td>
</tr>
<tr>
<td>Adding Hardware Interfaces</td>
<td>Minor updates.</td>
</tr>
<tr>
<td>Platform Naming Convention</td>
<td>Minor updates.</td>
</tr>
<tr>
<td>Software Package Management in PetaLinux rootfs</td>
<td>Updated link.</td>
</tr>
<tr>
<td>Adding Hardware Interfaces</td>
<td>Updated section.</td>
</tr>
<tr>
<td>Enabling Hardware Emulation for Extensible XSA</td>
<td>Added.</td>
</tr>
<tr>
<td>08/20/2020 Version 2020.1</td>
<td></td>
</tr>
<tr>
<td>Creating Embedded Platforms in Vitis</td>
<td>Added.</td>
</tr>
<tr>
<td>06/24/2020 Version 2020.1</td>
<td>Rewrite for this version.</td>
</tr>
</tbody>
</table>

Migrating to a New Target Platform Revision History

The following table shows the revision history for Migrating to a New Target Platform.

<table>
<thead>
<tr>
<th>Section</th>
<th>Revision Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Entire section</td>
<td>No changes to this section.</td>
</tr>
<tr>
<td>12/15/2020 Version 2020.2</td>
<td></td>
</tr>
<tr>
<td>Entire section</td>
<td>No updates to this section.</td>
</tr>
<tr>
<td>Entire section</td>
<td>No updates to this section.</td>
</tr>
<tr>
<td>08/20/2020 Version 2020.1</td>
<td></td>
</tr>
<tr>
<td>Entire section</td>
<td>No updates to this section.</td>
</tr>
<tr>
<td>06/24/2020 Version 2020.1</td>
<td></td>
</tr>
<tr>
<td>Entire section</td>
<td>Editorial updates only. No technical content updates.</td>
</tr>
<tr>
<td>06/03/2020 Version 2020.1</td>
<td></td>
</tr>
<tr>
<td>Entire section</td>
<td>Editorial updates only. No technical content updates.</td>
</tr>
</tbody>
</table>
Please Read: Important Legal Notices

The information disclosed to you hereunder (the "Materials") is provided solely for the selection and use of Xilinx products. To the maximum extent permitted by applicable law: (1) Materials are made available "AS IS" and with all faults, Xilinx hereby DISCLAIMS ALL WARRANTIES AND CONDITIONS, EXPRESS, IMPLIED, OR STATUTORY, INCLUDING BUT NOT LIMITED TO WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, OR FITNESS FOR ANY PARTICULAR PURPOSE; and (2) Xilinx shall not be liable (whether in contract or tort, including negligence, or under any other theory of liability) for any loss or damage of any kind or nature related to, arising under, or in connection with, the Materials (including your use of the Materials), including for any direct, indirect, special, incidental, or consequential loss or damage (including loss of data, profits, goodwill, or any type of loss or damage suffered as a result of any action brought by a third party) even if such damage or loss was reasonably foreseeable or Xilinx had been advised of the possibility of the same. Xilinx assumes no obligation to correct any errors contained in the Materials or to notify you of updates to the Materials or to product specifications. You may not reproduce, modify, distribute, or publicly display the Materials without prior written consent. Certain products are subject to the terms and conditions of Xilinx's limited warranty, please refer to Xilinx's Terms of Sale which can be viewed at https://www.xilinx.com/legal.htm#tos; IP cores may be subject to warranty and support terms contained in a license issued to you by Xilinx. Xilinx products are not designed or intended to be fail-safe or for use in any application requiring fail-safe performance; you assume sole risk and liability for use of Xilinx products in such critical applications, please refer to Xilinx's Terms of Sale which can be viewed at https://www.xilinx.com/legal.htm#tos.

AUTOMOTIVE APPLICATIONS DISCLAIMER

AUTOMOTIVE PRODUCTS (IDENTIFIED AS "XA" IN THE PART NUMBER) ARE NOT WARRANTED FOR USE IN THE DEPLOYMENT OF AIRBAGS OR FOR USE IN APPLICATIONS THAT AFFECT CONTROL OF A VEHICLE ("SAFETY APPLICATION") UNLESS THERE IS A SAFETY CONCEPT OR REDUNDANCY FEATURE CONSISTENT WITH THE ISO 26262 AUTOMOTIVE SAFETY STANDARD ("SAFETY DESIGN"). CUSTOMER SHALL, PRIOR TO USING OR DISTRIBUTING ANY SYSTEMS THAT INCORPORATE PRODUCTS, THOROUGHLY TEST SUCH SYSTEMS FOR SAFETY PURPOSES. USE OF PRODUCTS IN A SAFETY APPLICATION WITHOUT A SAFETY DESIGN IS FULLY AT THE RISK OF CUSTOMER, SUBJECT ONLY TO APPLICABLE LAWS AND REGULATIONS GOVERNING LIMITATIONS ON PRODUCT LIABILITY.
Copyright

© Copyright 2019-2021 Xilinx, Inc. Xilinx, the Xilinx logo, Alveo, Artix, Kintex, Spartan, Versal, Virtex, Vivado, Zynq, and other designated brands included herein are trademarks of Xilinx in the United States and other countries. AMBA, AMBA Designer, Arm, ARM1176JZ-S, CoreSight, Cortex, PrimeCell, Mali, and MPCore are trademarks of Arm Limited in the EU and other countries. OpenCL and the OpenCL logo are trademarks of Apple Inc. used by permission by Khronos. PCI, PCIe, and PCI Express are trademarks of PCI-SIG and used under license. All other trademarks are the property of their respective owners.