SDAccel Environment User Guide

UG1023 (v2017.4) July 30, 2018

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
07/30/2018	2017.4	• Restored `xocc --pk` option, added `--dk`, `--reuse_synth`, and `--reuse_impl` options. • Added references to the *SDAccel Environment Programmers Guide* (UG1277) and the *SDAccel Environment Debugging Guide* (UG1281).
03/30/2018	2017.4	• Changed `--pk` to `--profile_kernel` • Added note warning that for function prototype presented for RTL Kernel Wizard is only an example.
01/26/2018	2017.4	Changes are: • Throughout, updated formatting and command options. • Removed redundant content in Elements of SDAccel • Removed create_kernel Tcl command. • Added definition of super logic region (SLR) in User-Specified SLR Assignments for Kernels • Updated Using RTL Kernels • Updated User-Specified SLR Assignments for Kernels • Updated SDAccel Debug Command Line Flow • Moved `--optimize` option to XOCC Options for Link Mode table. • Updated `--nk` and `--sp` options in Compiling Your OpenCL Kernel Using the Xilinx OpenCL Compiler (xocc) • Marked `--device` as deprecated in Running Software and Hardware Emulation in XOCC Flow • Updated RTL Kernel Wizard • Added definition of VLNV to RTL Kernel Wizard General Settings • Added API Functions to note in Global Memory • Added functions to Private Memory. • Added content to Using Hardware Emulation • Added an example of how 16 sub-devices can be created, what xocc command and option to use, and how it is used in the Makefile in Global Memory.

Date	Version	Revision
12/20/2017	2017.4	Major reorganization and rewrite of the content in this guide, including: • Relocated Profiling and Optimization content to *SDAccel Environment Profiling and Optimization Guide* (UG1207) • Rewrite of Debugging Applications. • Rewrite of Compilation Flow.
08/16/2017	2017.2	• Reorganization of Kernel Language Support chapter. • Corrections to reflect the current release.
06/20/2017	2017.1	• Revised and reorganized Kernel Optimization Support chapter. • New section for OpenCL Installable Client Drive (ICD) Loader. • New Getting Started with Examples chapter. • New XP Parameters table in Commpliation Flow chapter. • Revised xocc Options table in Compilation chatper. • Updates to accommodate 2017.1 SDx software release.

XILINX

Table of Contents

Chapter 1

Introduction

Software is at the foundation of application specification and development. Whether the end application is targeted towards entertainment, gaming, or medicine, most products available today began as a software model or prototype that needed to be accelerated and executed on a hardware device. From this starting point, the software engineer is tasked with determining the execution device to get a solution to market and to achieve the highest possible degree of acceleration possible.

One traditional approach to accomplish this task has been to rely on processor clock frequency scaling. On its own, this approach has entered a state of diminishing returns, which has in turn led to the development of multi-core and heterogeneous computing devices. These architectures provide the software engineer with the possibility to more effectively trade-off performance and power for different form factors and computational loads. The one challenge in using these new computing architectures is the programming model of each device. At a fundamental level, all multi-core and heterogeneous computing devices require that the programmer rethink the problem to be solved in terms of explicit parallelism.

Recognizing the programming challenge of multi-core and heterogeneous compute devices, the Khronos Group industry consortium has developed the OpenCL™ programming standard. The OpenCL specification for multi-core and heterogeneous compute devices defines a single consistent programming model and system level abstraction for all hardware devices that support the standard. For a software engineer this means a single programming model to learn what can be directly used on devices from multiple vendors.

As specified by the OpenCL standard, any code that complies with the OpenCL specification is functionally portable and will execute on any computing device that supports the standard. Therefore, any code change is for performance optimization. The degree to which an OpenCL program needs to be modified for performance depends on the quality of the starting source code and the execution environment for the application.

Xilinx is an active member of the Khronos Group, collaborating on the OpenCL specification, and supports the compilation of OpenCL programs for Xilinx® FPGAs. The Xilinx SDAccel™ development environment is used for compiling OpenCL programs to execute on a Xilinx FPGA.

There are some differences between compiling a program for execution in an FPGA and a CPU/GPU environment. The following chapters in this guide describe how to use the SDAccel development environment to compile an OpenCL program for a Xilinx FPGA. This book is intended to document the features and usages of the SDAccel development environment. It is assumed that the user already has a working knowledge of OpenCL API. Though it includes some high level OpenCL concepts, it is not intended as an exhaustive technical guide on the OpenCL API. For more information on the OpenCL API, see the OpenCL specification available from the Khronos Group, and the OpenCL API introductory videos available on the Xilinx website.

Elements of SDAccel

The SDAccel™ Environment inherits many of the tools in the Xilinx® Software Development Kit (SDK), including GNU toolchains and standard libraries (for example, `glibc`) as well as the Target Communication Framework (TCF) and GDB interactive debuggers, a performance analysis perspective within the Eclipse/CDT-based GUI, and command-line tools.

The SDAccel Environment includes a system compiler (`xocc`) that generates complete hardware/software systems, an Eclipse-based user interface to create and manage projects and workflows, and a system performance estimation capability to explore different "what if" scenarios for the hardware/software interface.

The SDAccel system compiler employs underlying tools from the Vivado Design Suite (System Edition), including Vivado® HLS, IP integrator, IP libraries for data movement and interconnect, and the RTL synthesis, placement, routing, and bitstream generation tools.

TIP: *After you have done a build in SDx, you can launch Vivado. From the SDx menu, select* ***Xilinx → Vivado Integration → Open Vivado Project****.*

The principle of design reuse underlies workflows you employ with the SDAccel environment, using well established platform-based design methodologies. SDAccel solutions are compiled against a target platform, a combination of board and infrastructure components on which the kernels of an application are executed. The SDAccel Environment includes a number of platforms for application development, and others can be provided by Xilinx partners, or custom developed by FPGA design teams. The *SDAccel Environment Platform Development Guide* (UG1164) describes how to create a design using the Vivado Design Suite, specify platform properties to define and configure Platform Interfaces, and define the corresponding software run-time environment to build a platform for use in the SDAccel environment.

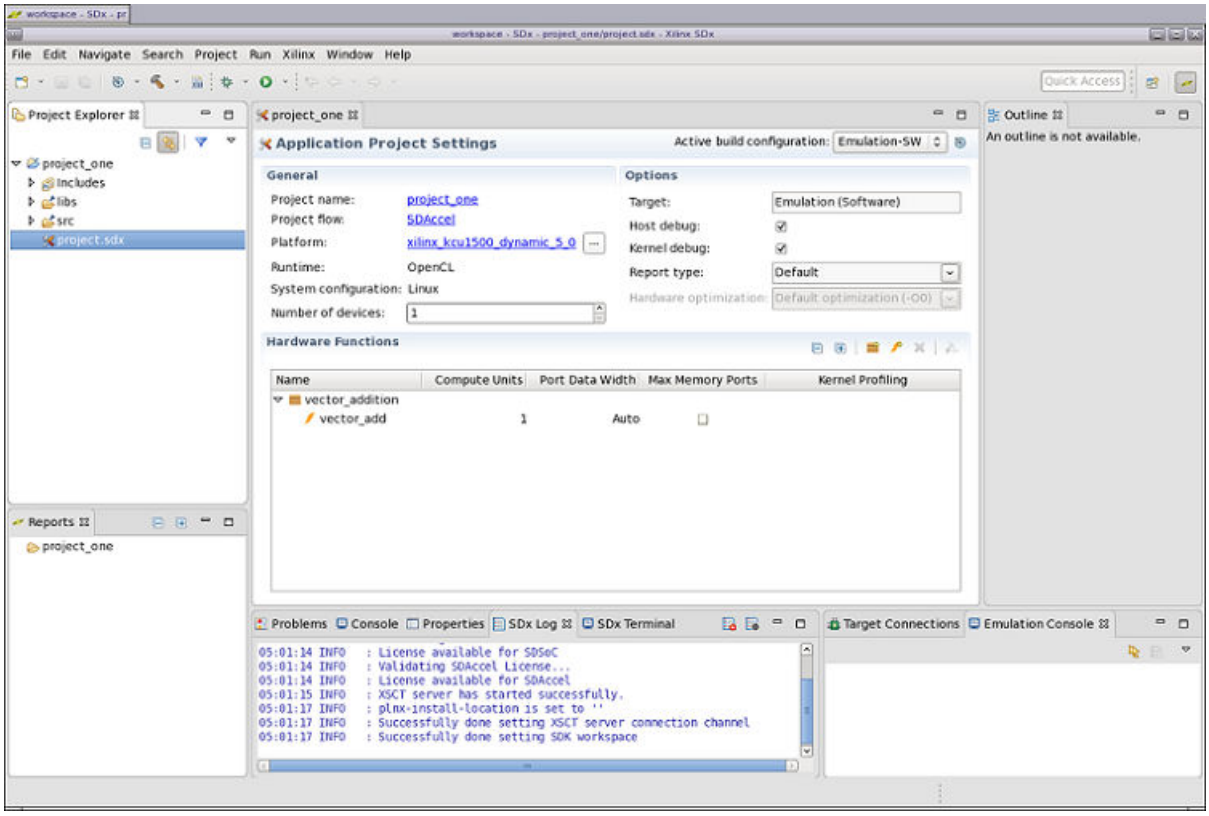
Devices can be provided by SDAccel ecosystem partners, FPGA design teams, and Xilinx.

See the *SDx Environments Release Notes, Installation, and Licensing Guide* (UG1238) for the most up-to-date list of supported devices.

Working with SDx

When a project is opened in the SDx IDE, the project is arranged in a series of different window and editor views, also known as a perspective in the IDE. The tool opens with the SDx (default) perspective.

Figure 1: **SDAccel - Default Perspective**



See Launching SDx for more information about opening the software.

As shown, the default perspective has an arrangement of Project Explorer view, Project Editor window, and the Outline view across the top, and the Report view, the Console view, and Target Connections view across the bottom. A brief description of these are:

- **Project Explorer**: Displays a tree view of the project folders and the source files, build files, and reports generated by the tool.
- **Project Editor**: This is the primary window for interacting with the project in the SDx IDE. It displays project settings, context sensitive code editors, and provides access to many commands for working with the project.
- **Outline**: Displays an outline of the current file opened in the Project Editor.

- **Report**: Displays the SDx reports of performance estimation, profile summaries, and build results.
- **Console**: Presents multiple views including the command console, problem reports, project properties, and logs and terminal views.
- **Target Connections**: Provides status for different targets connected to the SDx tool, such as the Vivado hardware server, Target Communication Framework (TCF), and QEMU networking.

You can open and close windows, using the **Window → Show View** command, and arrange them to suit your needs by dragging and dropping them into new locations in the IDE. Save the window arrangement as a perspective using the **Window → Perspective → Save Perspective As** command. This lets you define different perspectives for initial project editing, report analysis, and debug for example.

You can open different perspectives using the **Window → Perspective → Open Perspective** command. You can restore the default window arrangement by opening the SDx (default) perspective.

Command-Line Flow

In addition to the SDx IDE, the SDAccel environment provides a command line interface to support a scripted Makefile flow, or command-line execution as described in Chapter 5: Compilation Flow.

- C and C++ code for the host application can be compiled using the `xcpp` command.
- The OpenCL kernel can be compiled using the `xocc` command.
- The command line executables are located in the installation directory at `<sdx_install>/bin`.

Chapter 2

Creating an SDAccel Project

Launching SDx

IMPORTANT!: *The SDAccel™ application runs only on the Linux operating system. See the SDx Environments Release Notes, Installation, and Licensing Guide (UG1238) for a description of the software requirements for the SDAccel Environment.*

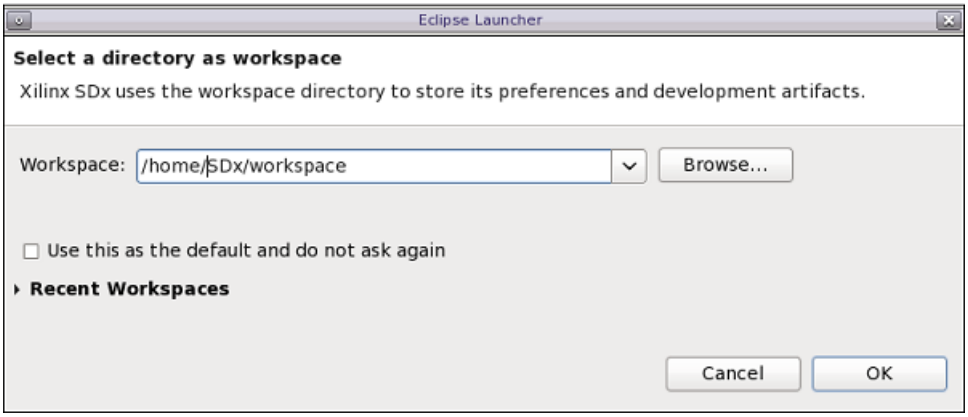
You can launch the SDx IDE directly from the desktop icon, or from the command line by one of the following methods:

- Double-clicking the SDx icon to start the application
- Launching it from the Start menu in the Windows operating system
- Using the following command from the command prompt: `sdx`

TIP: *Launching the SDx tool from the command line requires that the command shell is configured to run the application, or is an SDx Terminal window launched from the Start menu. To configure a command shell, run the `settings64.bat` file on Windows, or source the `settings64.sh` or `settings64.csh` file on Linux, from the `<install_dir>/SDx/2017.4` directory. Where `<install_dir>` is the installation folder of the SDx software.*

The SDx IDE opens, and prompts you to select a workspace when you first open the tool.

Figure 2: **Specify SDAccel Workspace**



The SDx workspace is the SDx folder that stores your projects, source files, and results while working in the tool. You can define separate workspaces for each project, or have workspaces for different types of projects, such as the SDAccel project you can create with the following instructions.

1. Use the **Browse** button to navigate to and specify the workspace, or type the appropriate path in the **Workspace** field.
2. Select the **Use this as the default and do not ask again** checkbox to set the specified workspace as your default choice and eliminate this dialog box in subsequent uses of SDx.

TIP: *You can always change the current workspace from within the SDx IDE using the **File → Switch Workspace** command.*

3. When you click **OK**, if this is the first time the SDx IDE has been launched, the **SDx Welcome** screen opens to let you specify an action. Select either **Create SDx Project**, or **File → New → SDx Project**.

Importing Sources

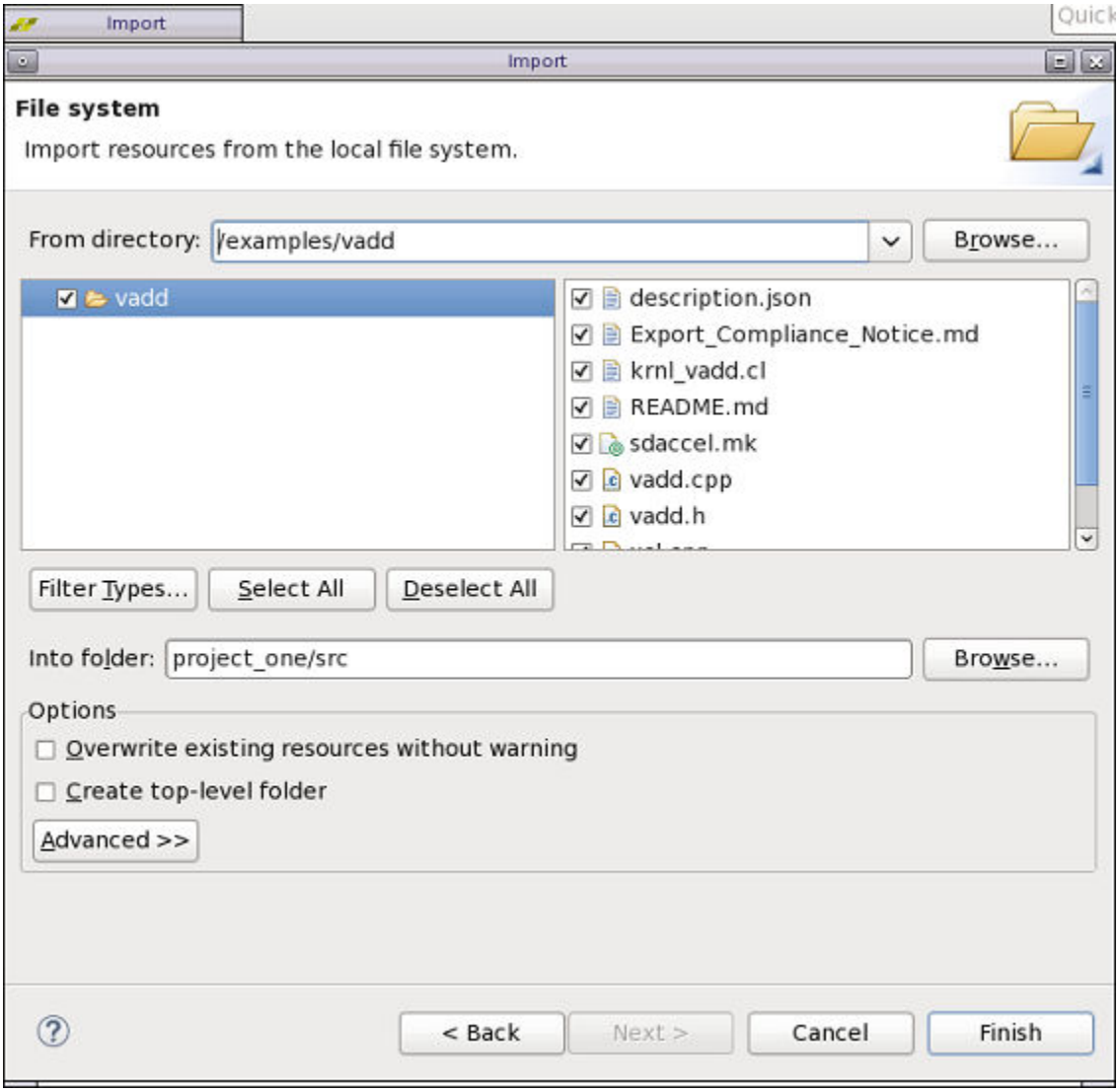
With the project open in the SDx IDE, you can import source files to add them to the project. To add files, right-click the `src` folder in the Project Explorer view, and select the **Import** command.

TIP: *You can also access this through the **File → Import** menu command.*

This displays the Import dialog box, which lets you specify the source of files you from which you are importing. The different sources include importing from archives, from existing projects, from the file system, and from a Git repository. Select the source of files you will be importing, and click **Next**.

The dialog box that diplays depends on the source of files you selected in the prior step. In the following figure you can see the File System dialog box that is displayed as result of choosing to import sources from the file system.

Figure 3: **Import File System Sources**



The File System dialog box lets you navigate to a folder in the system, and select files to import into your project. You can specify files from multiple folders, and specify the folder to import files into. You can enable the **Overwrite existing resource without warning** to simply overwrite any existing files, and enable **Create top-level folder** to have the files imported into a directory structure that matches the source file structure. If this checkbox is not enabled, which is the default, then the files will simply be imported into the specified **Into folder**.

After adding source files to your project, you are ready to begin configuring, compiling, and running the application.

Programming for SDAccel

The custom processing architecture generated by the SDAccel environment for a kernel running on a Xilinx FPGA provides opportunities for significant performance gains. However, you must take advantage of these opportunities by writing your host and kernel code specifically for acceleration on an FPGA.

The host application is running on x86 servers and uses the SDAccel runtime to manage interactions with the FPGA kernels. The host application is written in C/C++ using OpenCL APIs. The custom kernels are running within a Xilinx FPGA on an SDAccel platform.

The SDAccel hardware platform contains global memory banks. The data transfer from the host machine to kernels and from kernels to the host happens through these global memory banks. Communication between the host x86 machine and the SDAccel accelerator board occurs across the PCIe bus.

The *SDAccel Environment Programmers Guide* (UG1277) discusses how to write code for the host application to setup the SDAccel OpenCL runtime, load the kernel binary into the SDAccel platform, pass data efficiently between the host application and the kernel, and trigger the kernel on the FPGA at the appropriate time in the host application. Refer to the *SDAccel Environment Programmers Guide* for details of the host application, kernel code, and the interactions between them.

Kernel Language Support

The SDAccel™ environment supports kernels expressed in OpenCL™ C, C/C++ and RTL (Verilog or VHDL). You can use different kernel types in the same application. However, each kernel has specific requirements and coding styles that should be used.

Using OpenCL Kernels

The SDAccel™ environment supports the OpenCL™ language constructs and built in functions from the OpenCL 1.0 embedded profile. The following is an example of an OpenCL kernel for matrix multiplication that can be compiled with the SDAccel environment.

```
__kernel __attribute__ ((reqd_work_group_size(16,16,1)))
void mult(__global int* a, __global int* b, __global int* output)
{
  int r = get_local_id(0);
  int c = get_local_id(1);
  int rank = get_local_size(0);
  int running = 0;
  for(int index = 0; index < 16; index++){
    int aIndex = r*rank + index;
    int bIndex = index*rank + c;
    running += a[aIndex] * b[bIndex];
```

```
  }
  output[r*rank + c] = running;
  return;

}
```

IMPORTANT!: *Standard C libraries such as `math.h` can not be used in OpenCL C kernel. Use OpenCL built-in C functions instead.*

Using C/C++ Kernels

The kernel for matrix multiplication can be expressed in C/C++ code that can be synthesized by the Vivado® HLS tool. For kernels captured in this way, the SDAccel™ Environment supports all of the optimization techniques available in Vivado HLS. The only thing that the user has to keep in mind is that expressing kernels in this way requires compliance with a specific function signature style.

IMPORTANT!: *Global variables and printf() are not supported in HLS C/C++ kernels.*

```
void mmult(int *a, int *b, int *output)
{
#pragma HLS INTERFACE m_axi port=a offset=slave bundle=gmem
#pragma HLS INTERFACE m_axi port=b offset=slave bundle=gmem
#pragma HLS INTERFACE m_axi port=output offset=slave bundle=gmem
#pragma HLS INTERFACE s_axilite port=a bundle=control
#pragma HLS INTERFACE s_axilite port=b bundle=control
#pragma HLS INTERFACE s_axilite port=output bundle=control
#pragma HLS INTERFACE s_axilite port=return bundle=control

  const int rank = 16;
  int running = 0;
  int bufa[256];
  int bufb[256];
  int bufc[256];
  memcpy(bufa, (int *) a, 256*4);
  memcpy(bufb, (int *) b, 256*4);

  for (unsigned int c=0;c<rank;c++){
    for (unsigned int r=0;r<rank;r++){
      running=0;
      for (int index=0; index<rank; index++) {
  #pragma HLS pipeline
        int aIndex = r*rank + index;
        int bIndex = index*rank + c;
        running += bufa[aIndex] * bufb[bIndex];
      }
      bufc[r*rank + c] = running;
    }
  }

  memcpy((int *) output, bufc, 256*4);
  return;
}
```

The preceding code example is the matrix multiplication kernel expressed in C/C++ for Vivado HLS. The first thing to notice about this code is the function signature.

```
void mmult(int *a, int *b, int *output)
```

This function signature is almost identical to the signature of the kernel expressed in OpenCL C. It is important to keep in mind that by default, kernels captured in C/C++ for HLS do not have any inherent assumptions on the physical interfaces that will be used to transport the function parameter data. HLS uses pragmas embedded in the code to direct the compiler as to which physical interface to generate for a function port. For the function to be treated as a valid OpenCL kernel, the ports on the C/C++ function must be defined on the memory and control interface pragmas for HLS.

The memory interface specification is generated by the following command:

```
#pragma HLS INTERFACE m_axi port=<variable name> offset=slave
bundle=<interface name>
```

With each unique bundle name used, there is a separate AXI4 master interface created. An interface name is generated by the compiler with the following rules, taking, for example, a function argument called *arg_name*.

Using platforms version 4.x or earlier, the interface name M_AXI_ARG_NAME was used by making *arg_name* uppercase irrelevant of the original capitalization and prefixing with M_AXI_.

IMPORTANT!: *Using current platforms (versions 5 or later) the interface name m_axi_arg_name is used: the original capitalization of arg_name must be lower case and prefixed by m_axi_.*

This interface name is required for some advanced options to instruct the tools to connect the interface to a specific platform DDR memory interface.

The control interface specification is generated by the following command:

```
#pragma HLS INTERFACE s_axilite port=<variable name> bundle=<interface
name>
```

Detailed information on how these pragmas are used is available in the *SDx Pragma Reference Guide* (UG1253).

When a kernel is defined in C++, use extern "C" { ... } around the functions targeted to be kernels. The use of extern "C" instructs the compiler/linker to use the C naming and calling conventions.

When using structs it is recommended that the struct has a size in bytes that is a power of two in total. Taking into consideration that the maximum bitwidth of the underlying interface is 512 bits or 64 bytes, the recommended size of the struct is 4, 8, 16, 32 or 64 bytes.

IMPORTANT!: *To reduce the risk of misalignment between the host code and the kernel code it is recommended that the struct elements use types of the same size.*

TIP: *C++ arbitrary precision data types can be used for global memory pointers on a kernel. They are not supported for scalar kernel inputs that are passed by value.*

When using the command line flow, the kernel name is passed to `xocc` using the option `--kernel`, for example:

```
xocc .. --kernel my_c_kernel
```

For more information about `xocc` command options, see Compiling Your OpenCL Kernel Using the Xilinx OpenCL Compiler (xocc)

Using RTL Kernels

In the host application, the RTL kernel is invoked in a similar manner as HLS kernels with a function signature such as:

```
void mmult(int *a, int *b, int *output)
```

```
void mmult(unsigned int length, int *a, int *b, int *output)
```

This implies that the RTL design must have an execution model similar to that of a software function or kernel: start, execute, and end.

- It must be capable of starting when called to do so.
- It must compute all data values.
- It must return the data and then end the operation.

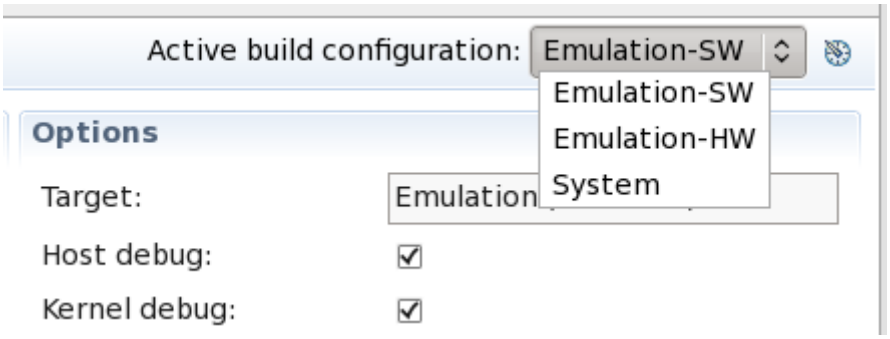
If the RTL design has a different execution model then logic must be added to ensure that the design can be executed in this manner. The RTL Kernel Wizard provides a flow that allows such changes to be performed.

For more information, see Appendix E: Creating RTL Kernels.

Building the System

Compilation of an application for execution on a Xilinx enabled OpenCL device through the SDAccel development environment is performed by selecting the compilation target. This step goes beyond compilation of host and kernel code and is also responsible for the generation of custom compute units for all of the binary containers in the solution.

Figure 4: **Active Build Configuration**



With the application project opened, you can specify the compilation target from the Project Editor window. Select **Active build configuration** to specify the compilation target. The choices are as follows:

- **Emulation-SW**: This compiles the project for use in software emulation. Kernel code can be debugged on the processor, with the host application, in the software emulation flow.
- **Emulation-HW**: This compiles the project for use in hardware emulation. This emulation flow invokes the hardware simulator to test the functionality of the logic that is executed on the FPGA compute fabric.
- **System**: Also called the "build system flow", this compiles the kernel into the FPGA hardware.

TIP: *You can also assign the compilation target from the* ***Build*** *( ) command, or from the* ***Project*** *→* ***Build Configurations*** *→* ***Set Active*** *menu command.*

The recommended flow is:

1. Perform software emulation (`sw_emu`) to confirm the functionality.
2. Perform hardware emulation (`hw_emu`) to create custom hardware and review the performance of the kernel.
3. Perform a build of the hardware `hw`system to implement the custom hardware.

As described in the Chapter 5: Compilation Flow, you can also specify the compilation target from a Makefile, or from the command-line with the following command:

```
xocc --target sw_emu|hw_emu|hw ...
```

The compilation method used is dependent on the selected kernel compilation target. The `xocc --target` option invokes different flows for kernels targeted at a processor and kernels targeted at the FPGA fabric.

Using Software Emulation

In the context of the SDAccel™ development environment, application emulation on a CPU is the same as the iterative development process that is typical of CPU/GPU programming. In this type of development style, a programmer continuously compiles and runs an application as it is being developed.

The main goal of Software emulation is to ensure functional correctness and to partition the application into kernels. Although partitioning and optimizing an application into kernels is integral to OpenCL™ development, performance is not the main goal at this stage of application development in the SDAccel environment.

For CPU-based emulation, both the host code and the kernel code are compiled to run on an x86 processor. The programmer model of iterative algorithm refinement through fast compile and run loops is preserved with speeds that are the same as a CPU compile and run cycle.

Using Hardware Emulation

The SDAccel™ Environment generates at least one custom compute unit for each kernel in an application. This means that while the Software Emulation flow is a good measure of functional correctness, it does not guarantee correctness on the FPGA execution target. Before deployment, check that the custom compute units generated by the tool are producing the correct results.

The SDAccel Environment has a Hardware Emulation flow, which enables the programmer to check the correctness of the logic generated for the custom compute units. This emulation flow invokes the hardware simulator in the SDAccel environment to test the functionality of the logic that will be executed on the FPGA compute fabric.

The memory model used for Hardware Emulation is not cycle accurate with RTL; consequently, any performance numbers shown in the profile summary report are approximate, and must be used only as a general guidance and for comparing relative performance between different kernel implementations.

Building for the Target FPGA/Platform

The SDAccel™ development environment generates custom logic for every compute unit in the binary container. Therefore, it is normal for this build step to run for a longer period of time than the other steps in the SDAccel application compilation flow.

The steps in compiling compute units targeting the FPGA fabric are as follows:

1. Generate a custom compute unit for a specific kernel.
2. Instantiate the compute units in the OpenCL® binary container.
3. Connect the compute units to memory and infrastructure elements of the target device.
4. Generate the FPGA programming file.

The generation of custom compute units for any given kernel code uses the production proven capabilities of the Xilinx® Vivado® High-Level Synthesis (HLS) tool, which is the compute unit generator in the SDAccel Environment. Based on the characteristics of the target device in the solution, the SDAccel Environment invokes the compute unit compiler to generate custom logic that maximizes performance while at the same time minimizing compute resource consumption on the FPGA fabric. Automatic optimization of a compute unit for maximum performance is not possible for all coding styles without additional user input to the compiler. The *SDAccel Environment Profiling and Optimization Guide* (UG1207) discusses the additional user input that can be provided to the SDAccel Environment to optimize the implementation of kernel operations into a custom compute unit.

After all compute units have been generated, these units are connected to the infrastructure elements provided by the target device in the solution. The infrastructure elements in a device are all of the memory, control, and I/O data planes which the device developer has defined to support an OpenCL application. The SDAccel Environment combines the custom compute units and the base device infrastructure to generate an FPGA binary which is used to program the Xilinx device during application execution.

IMPORTANT!: *The SDAccel Environment always generates a valid FPGA hardware design, but does not generate an optimal allocation of the available bandwidth in the control and memory data planes. The user can manually optimize the data bandwidth usage by selecting connection points into the memory and control data planes per compute unit.*

Chapter 3

Profiling and Optimizing the Kernel

There are three distinct areas to be considered when performing algorithm optimization in SDAccel™:

- Host optimization
- Kernel optimization
- PCIe® bandwidth optimization

Most application developers are familiar with host code optimization. This usually requires the programmers to studying algorithmic complexities, overall system performance, and data locality. There are many methodology guides and software tools to guide the developer to identify performance bottlenecks. These same techniques can be applied to the design targeting to be accelerated with SDAccel.

Consequently, as a first step, programmers should optimize their overall program performance independently of the final target. However, the main difference between SDAccel and general purpose software is that in SDAccel projects, part of the core compute algorithms are pushed onto the FPGA. This implies that the developer must be aware of algorithm concurrency, data transfers, and the fact that programmable hardware is targeted.

Generally, the programmer must identify the section of the algorithm to be accelerated. The ratio between computation and the required data transfers to the accelerator should be sufficient to avoid requiring the system bus to create an unnecessary bottleneck.

Similarly, the host needs to efficiently utilize the accelerator. This implies that the host code must be optimized to facilitate the data transfers and kernel execution, as well as performing additional pre- and post-processing, if possible.

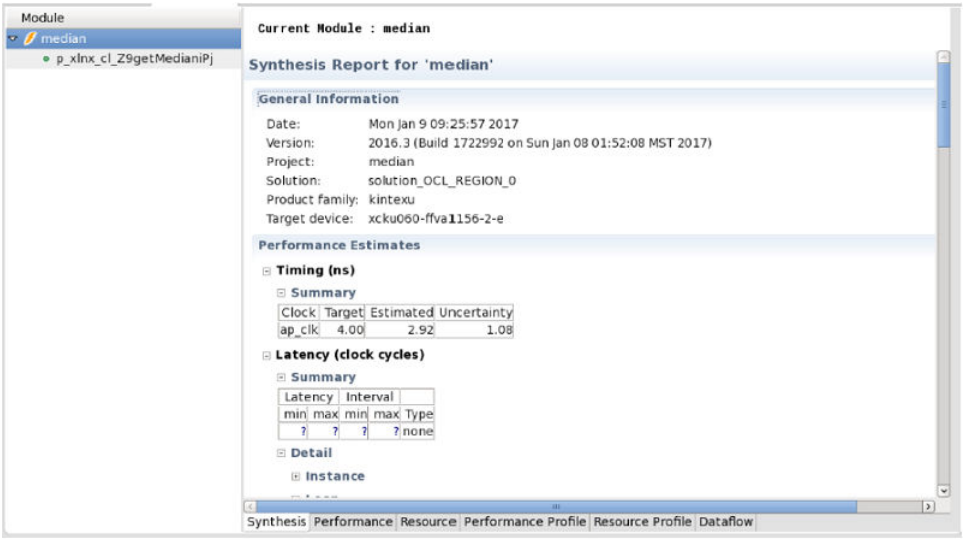
SDAccel is designed to support your efforts to optimize these areas, by generating reports that help you analyze the host code and the hardware kernels in some detail. The reports are automatically generated when you build the project, and listed in the Report view of the SDx IDE. To open a listed report, double-click the report.

The following figures show the three main reports: the HLS Report, the Application Timeline, and the Profile Summary. To access these reports from the SDx IDE, make sure the **Reports** view is visible. This view is typically below the **Project Explorer** view.

TIP: *You can use the **Window → Show View → Other** menu command to display the Reports view if it is not displayed. See Working with SDx for more information.*

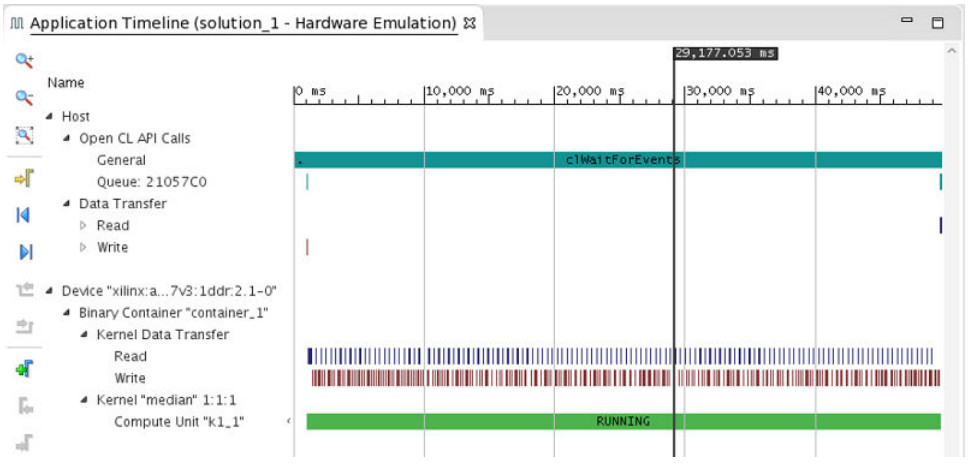
The HLS Report provides details about the High-Level Synthesis process (HLS). This tasks translates the C/C++ model into a hardware description language responsible for implementing the functionality on the FPGA. This enables the programmer to dive directly into the hardware implementation and optimize the kernel implementation.

Figure 5: **HLS Report Window**



The Application Timeline, provides a graphical representation of the OpenCL® interface calls during execution. It enables the programmer to visualize what operations are performed during what time across the complete application timeline. This enables the programmer to identify issues regarding kernel synchronization and efficient concurrent execution.

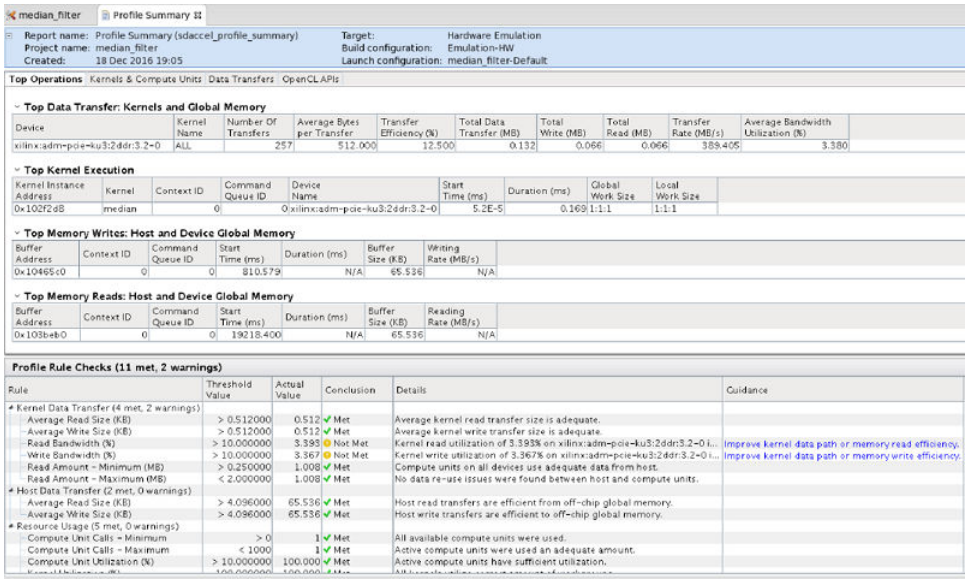
Figure 6: **Application Timeline Window**



Finally, the Profile Summary provides annotated details regarding the overall application performance. All data gathered during the execution of the program is gathered by SDAccel and grouped into categories. The profile summary enables the programmer to drill down to actual Data Transfer and Kernel Execution numbers and statistics.

Note: The profile summary also contains Profile Rule Checks (PRCs). PRCs show at a high level how the current performance numbers compare to commonly-achieved reference numbers.

Figure 7: **Profile Summary Window**



More details on each viewer, as well as the profiling and optimization methodology, common optimization steps, and even coding guidelines can be found in the *SDAccel Environment Profiling and Optimization Guide* (UG1207).

User-Specified SLR Assignments for Kernels

Dynamic platforms generally assume that the implementation tools will place a kernel in the same super logic region (SLR) as the memory bank that it accesses. If competition for logic resources leads to situations where a kernel is not automatically placed in the correct SLR, you can provide extra input to the `xocc` tools to assign the logic of the kernel into a nominated SLR. When you provide this explicit directive to place one or more kernels in an SLR, this information is shared with the memory subsystem of the dynamic platform, which automatically adds SLR-crossing pipelines to facilitate better timing closure in the platform.

IMPORTANT!: *The following features are currently only applicable to KCU1500 and VCU1525 dynamic platforms containing an SDx memory subsystem instance.*

Specifying SLR assignment information for a platform design requires two steps:

1. Create a User Post System Link Tcl file:

   To assign a kernel to a specific SLR, the user must first create a `userPostSysLink.tcl` file. This file should contain one or more of the following set property assignments:

   ```
   set_property CONFIG.SLR_ASSIGNMENTS <NAME OF SLR> \
   [get_bd_cells insert_kernel_name_here]
   ```

   Where *<NAME OF SLR>* is the string name of the SLR (SLR0, SLR1, SLR2, ...) where the you want the logic of the kernel to reside. For example:

   ```
   set_property CONFIG.SLR_ASSIGNMENTS SLR0 [get_bd_cells vadd_0]
   set_property CONFIG.SLR_ASSIGNMENTS SLR1 [get_bd_cells vadd_1]
   set_property CONFIG.SLR_ASSIGNMENTS SLR0 [get_bd_cells vadd_2]
   ```

2. Provide additional `xocc` options to apply a user post-system link Tcl file (`userPostSysLink.tcl`) to platform linking:

   When all the necessary `SLR_ASSIGNMENTS` properties are captured in the `userPostSysLink.Tcl` file,you can specify the following additional option on the `xocc` command line to apply the assignments to their design:

   ```
   --xp param:compiler.userPostSysLinkTcl=<full path to your
   userPostSysLink.tcl file>
   ```

IMPORTANT!: *The full file system path to the Tcl file must be provided. Relative paths are not supported by the `xocc` command at this time.*

Requirements for DSA Creation

The SLR automation feature of the IP integrator is currently only available when a `set_param` is enabled:

```
set_param ips.enableSLRParameter 2
```

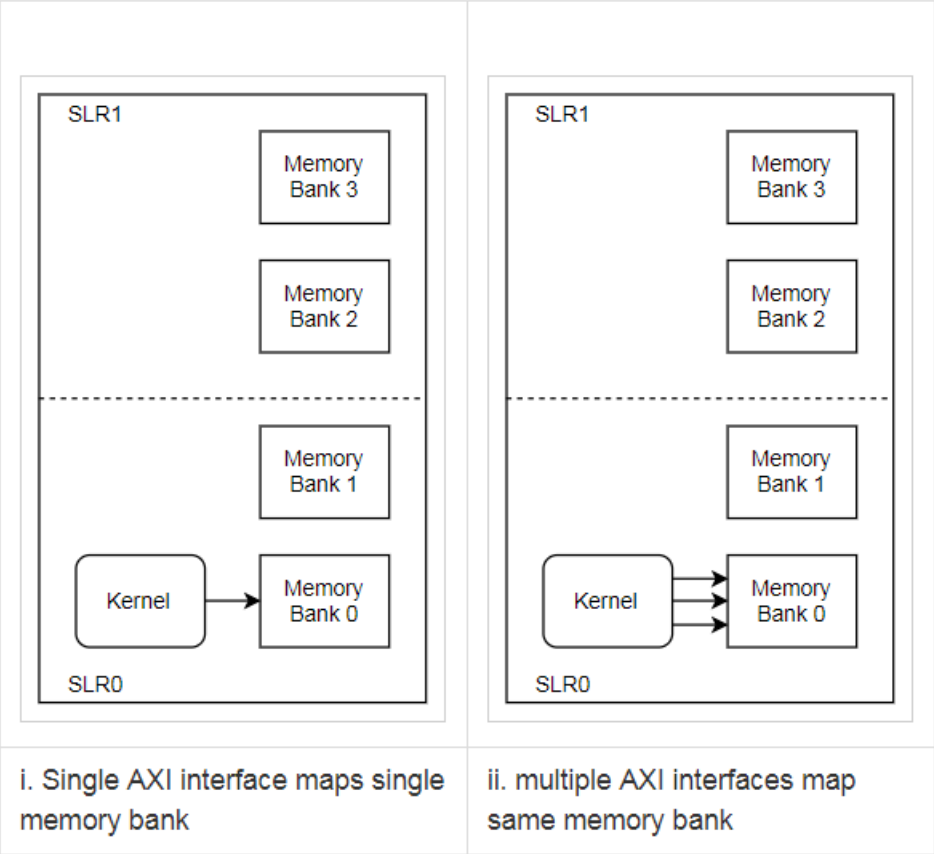
The supported platform DSAs (KCU1500/VCU1525) ensure that this parameter is enabled in their DSA pre-system link Tcl hook. The `set_param` should ideally be assigned before the project is opened or, at the very latest, before any IP integrator design is opened or created.

If applying this feature to a new DSA using 2017.4, ensure that `set_param` is applied in the DSA pre-link Tcl hook file.

Guidelines for Kernels that Access Multiple Memory Banks

The memory banks of a DSA are distributed across the SLRs of the platform and 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 memory banks 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 8: **Kernel and Memory in Same SLR**

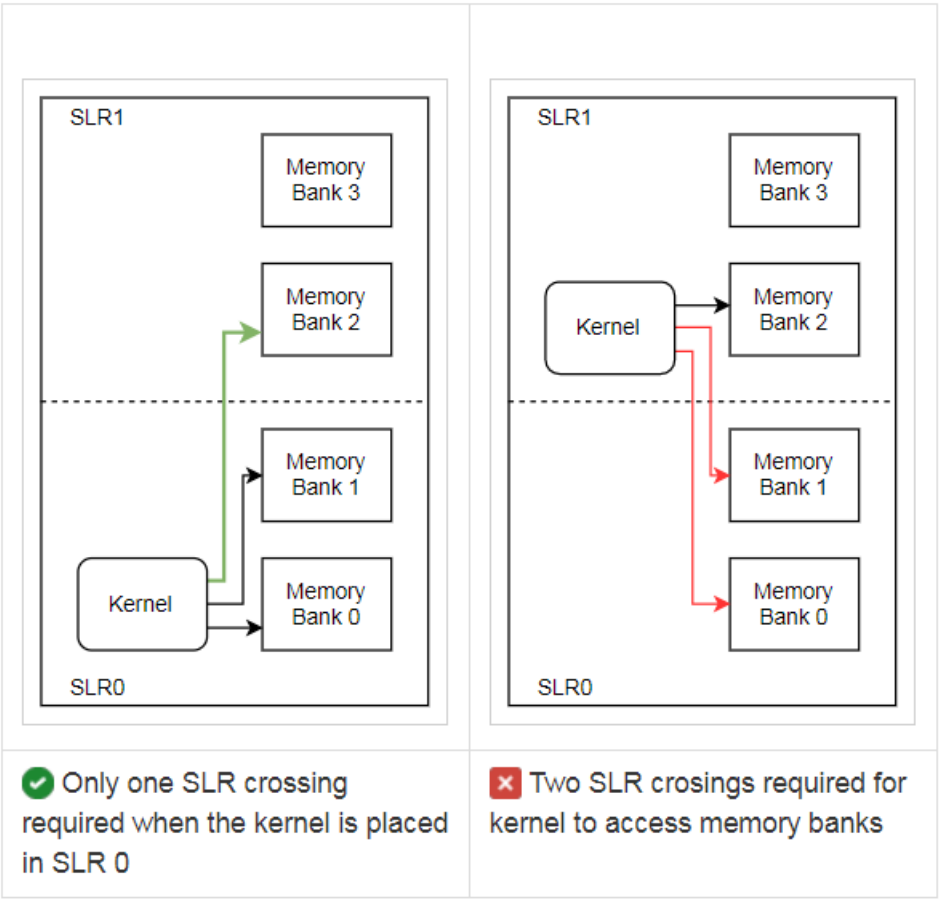


As shown in the previous figure, when a kernel has a single AXI interface that maps only a single memory bank, the DSA release information lists the SLR that is associated with the memory bank of the kernel; therefore, the SLR where the kernel would be best placed. In this simple scenario, the design 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, then 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.

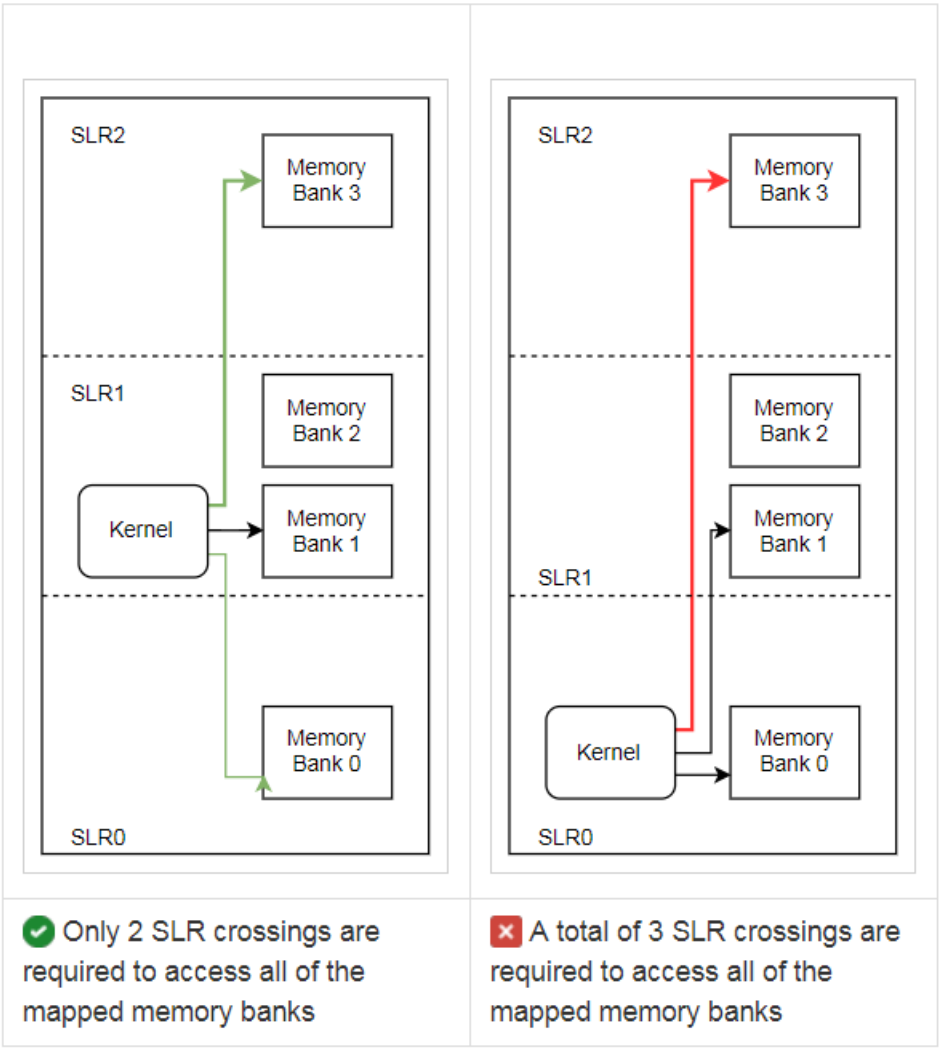
Figure 9: **Memory Bank in Adjoining SLR**



When a kernel has multiple AXI interfaces to mulitple 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 the user's design to reach their memory banks.

When the kernel is mapping memory banks from different SLRs, explicitly specify the SLR assignment for the kernel in a `userPostSysLink.tcl` file.

Figure 10: **Memory Banks Two SLRs Away**



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 4

Debugging Applications in the SDAccel Environment

The SDAccel™ Environment provides application level debug features which allow the host code, the kernel code, and the interactions between them to be debugged efficiently. The topics presented here are addressed in greater detail in the *SDAccel Environment Debugging Guide* (UG1281). Refer to that guide for more information.

The recommended application-level debugging flow consists of three levels of application debugging that can be summarized as:

1. Perform Software Emulation.

   Verify both the host and kernel code are functionally correct by running Software Emulation. It is recommended to iterate in Software Emulation, which takes little compile time and executes quickly, until the application is functioning correctly in all modes of operation.

2. Perform Hardware Emulation.

   Verify the host code and the kernel hardware implementation is correct by running Hardware Emulation on a data set. Hardware Emulation performs detailed verification using an accurate model of the hardware. Execution time for hardware emulation takes more time than software emulation.

TIP: *Xilinx recommends that you use small data sets for debug and validation.*

   It is also in this stage that you can optionally modify the kernel code to improve performance. Refer to *SDAccel Environment Profiling and Optimization Guide* (UG1207) for more information. Iterate in Hardware Emulation until the functionality is correct and the estimated kernel performance is sufficient.

3. Perform real system execution. At this stage, the final system image (`xclbin`) is generated and executed on the actual hardware. Now, the kernels are confirmed to be executing correctly on the actual FPGA hardware. It is expected that the focus shifts from debugging to actual host code performance tuning.

This three-tiered approach allows debugging at different levels of abstraction. In addition, the SDAccel Environment provides application debug features, which allow debugging of the interaction of the compiled host and kernel code, no matter what level of abstraction.

All execution models are supported through an integrated GUI flow as well as through a batch flow using basic compile time and runtime setup options. Both of these debug flows are presented in more detail in the following sections; however, before starting on the flows, the most rudimentary approach to debugging is briefly described.

Using printf() to Debug Kernels

The simplest and most rudimentary approach to debugging of algorithms is to verify key data values throughout the execution of the program. For application developers, this is a tried and trusted way of actually identifying problems within the execution of a program. Because part of the algorithm are now running on an FPGA, even this way of debugging requires additional support. Towards this end, the SDAccel™ development environment supports OpenCL™ `printf()` built-in function within the kernels in all development flows: Software Emulation, Hardware Emulation, and running kernel in actual hardware.

IMPORTANT!: *`printf()` is not supported in C/C++ kernels.*

The following is a kernel example of using `printf()` and the output when the kernel is executed with global size of 8:

```
__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;
}
```

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
```

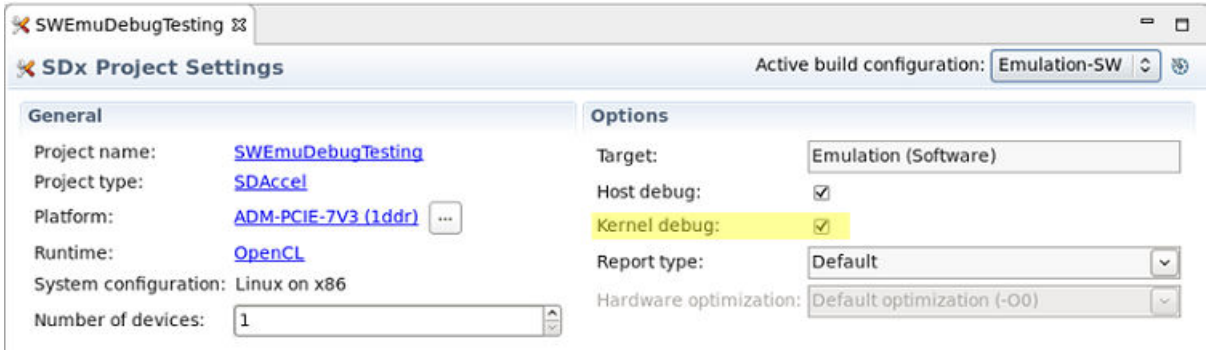
IMPORTANT!: *`printf()` messages are buffered in the global memory and unloaded when kernel execution is completed. If `printf()` is used in multiple kernels, there is no guarantee that the order of the messages from each kernel will display on the host terminal.*

SDAccel GUI Flow

Running SDAccel™ in the GUI flow provides easy access to the debug capabilities.

In the project view, check the **Kernel debug** box for the hardware emulation or software emulation target, as shown in the following figure.

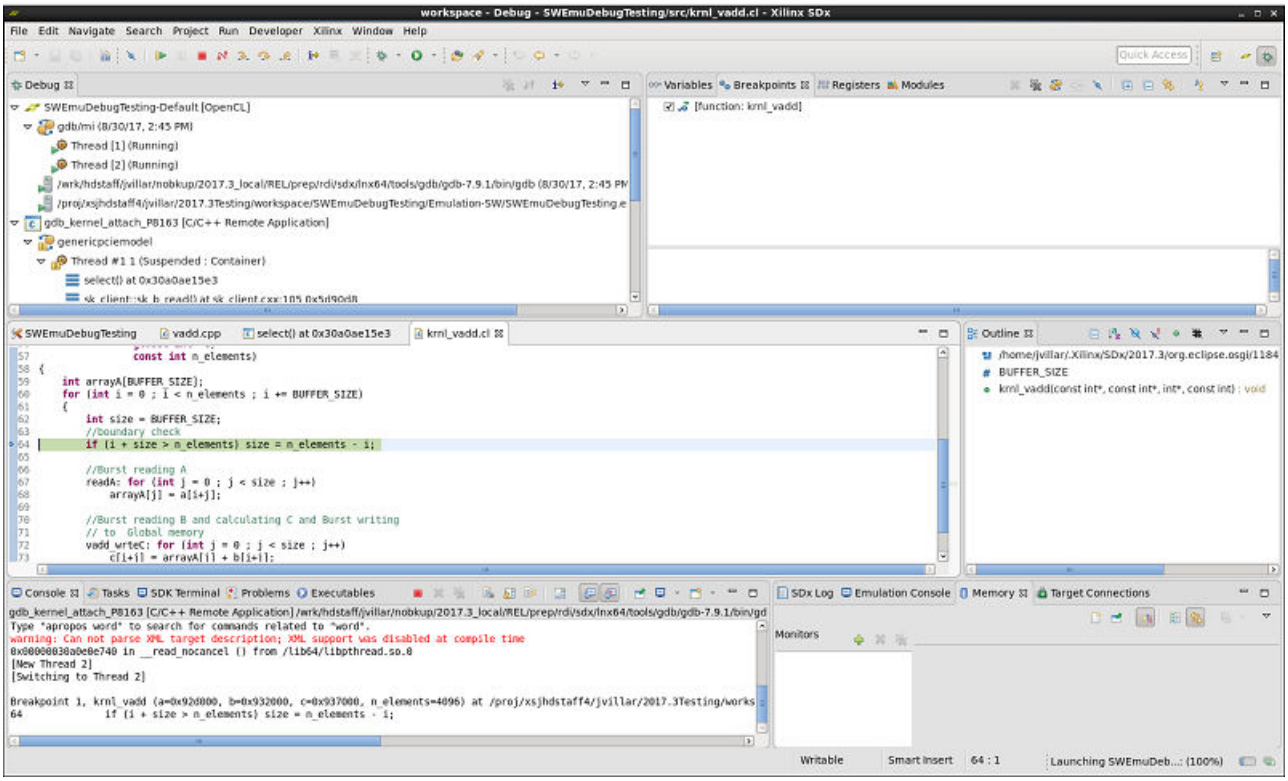
Figure 11: **Software Project Settings Options**



This completes the setup, and you can now run the default debug configuration.

When the kernel is spawned off, a breakpoint is hit at the beginning of the kernel in a separate GDB console.

Figure 12: **GDB Console**



It is now possible to add breakpoints, inspect variables, and debug the kernel as normal.

***Note*:** In hardware emulation, as the C/C++ code is translated for efficient implementation, breakpoints cannot be placed on all statements (mostly available untouched loops and functions) and similarly only preserved variables can be accessed.

See the GDB Extensions section for additional debug capabilities designed to allow easy access to common OpenCL® structures.

SDAccel Debug Command Line Flow

The Application Debug feature in the SDAccel™ Environment provides tools to debug the OpenCL® application running in all modes: Software Emulation, Hardware Emulation, or Hardware. It is referred to as *Application Debug* because it mainly deals with debugging on the host side, including the associated OpenCL APIs, and is in contrast to pure *Kernel Debug* which involves debugging OpenCL kernels (such as the accelerated portion).

There are four steps to application debugging in SDAccel using the command line flow:

1. General Environment Setup

2. Prepare the Host Code for Debug
3. Prepare Kernel Code for Debug
4. Launch GDB Standalone to Debug

IMPORTANT!: *the SDAccel Environment supports host program debugging in all flows, but kernel debugging is only supported in the emulation flows with* `gdb`*. In addition, more hardware centric debugging support such as waveform analysis is provided for the kernels.*

General Environment Setup

To run software emulation or hardware emulation from the command line, you must set the following environment variables:

Table 1: **Environment Variables and Values**

Environment Variable	Value
XCL_EMULATION_MODE	sw_emu or hw_emu
XILINX_SDX	The path to the installed SDx.
XILINX_OPENCL	The path to the installed SDx (same as ${XILINX_SDX}).
LD_LIBRARY_PATH	${LD_LIBRARY_PATH}:${XILINX_SDX}/lib/lnx64.o:${XILINX_SDX}/ runtime/lib/x86_64:${XILINX_SDX}/lib/lnx64.o/Default

Note: When using DSA 4.x or earlier, Software and Hardware Emulation require that you 4.x DSA libraries be prefixed to the `LD_LIBRARY_PATH`

Preparing the Host Code

The host program needs to be compiled with debugging information generated in the executable by adding the `-g` option to the `xcpp` command line option, as follows:

```
xcpp -g ...
```

TIP: *Because* `xcpp` *is simply a wrapper around the system compiler (*`gcc`*), the* `-g` *option enables the compiler to generate debug information.*

Preparing the Kernel

Kernel code can be debugged together with the host program in either software emulation or hardware emulation. Debugging information needs to be generated first in the binary container by passing the `-g` option to the `xocc` command line executable:

```
xocc -g -t [sw_emu | hw_emu] ...
```

The `-t` option is used to specify software emulation (`sw_emu`) or hardware emulation (`hw_emu`).

In the software emulation flow, additional runtime checks can be performed for OpenCL based kernels. The runtime checks include:

- Checking out-of-bound access made by kernel interface buffers (option: `address`)
- Checking uninitialized memory access initiated by kernel local to kernel (option: `memory`)

The options are enabled through the `--xp` option and need to be enabled during the link stage (`-l`) as shown in the following examples:

```
xocc -l -t sw_emu --xp param:compiler.fsanitize=address -o
bin_kernel.xclbin
xocc -l -t sw_emu --xp param:compiler.fsanitize=memory -o bin_kernel.xclbin
xocc -l -t sw_emu --xp param:compiler.fsanitize=address,memory -o
bin_kernel.xclbin
```

Once applied, the software emulation run produces a debug log with emulation diagnostic messages at `<project_dir>/Emulation-SW/<proj_name>-Default>/emulation_debug.log`.

Launching GDB

You can launch GDB standalone to debug the application if the host program and kernel were built with debug information (built with the `-g` flag). This flow should also work while using a graphical front-end for GDB, such as `ddd`.

Below are the instructions for launching GDB:

1. To set up the environment to run SDx, source the file below so that SDx command settings are in the `PATH`:
   - C Shell: `source <SDX_INSTALL_DIR>/settings64.csh`
   - Bash: `source <SDX_INSTALL_DIR>/settings64.sh`
2. Ensure that the environment variable `XCL_EMULATION_MODE` is set.
3. The application debug feature must be enabled at run time using an attribute in the `sdaccel.ini` file. Create an `sdaccel.ini` file in the same directory as your host executable, and include the following lines:

   ```
   [Debug]
   app_debug=true
   profile=true
   ```

4. Start `gdb` through the Xilinx® wrapper: `xgdb - args host.exe test.xclbin`

   The `xgdb` wrapper performs the following setup steps under the hood:

- Launches GDB on the host program: `gdb --args host.exe test.xclbin`
- Setup of the environment variables `PYTHONHOME` and `PYTHONPATH` to Python installation. Currently, the `gdb` in SDx expects Python 2.6 or Python 2.7. For example, if the Python available on the machine is Python 2.6 then set the environment as shown (Bash shell shown):

```
export PYTHONHOME=/usr
export PYTHONPATH=/usr/lib64/python2.6/:/usr/lib64/python2.6/lib-dynload/
```

- Sources the python script in the GDB console to enable the Xilinx GDB extensions:

```
gdb> source ${XILINX_SDX}/scripts/appdebug.py
```

Host Code Debugging

After `gdb` is launched, the host code can be debugged similar to any common application debug session. You can step through the code under GDB and examine the OpenCL objects to verify that their contents are indeed as expected at any point in the code.

Full Emulation Debug

To better mimic the hardware being emulated, Software Emulation kernels are spawned off as separate processes. If you are using GDB to debug the host code, breakpoints set on kernel lines will not be hit because the kernel code is not run within that process. To support the concurrent debugging of the host code as well as the kernel code, the SDAccel™ Environment provides a mechanism to attach to spawned kernels through the use of `sdx_server`.

As a result it is necessary to start three different terminals:

- In the first terminal, start the `sdx_server` using the following command:

```
${XILINX_SDX}/bin/sdx_server --sdx-url
```

- In a second terminal run the host code (`xgdb`). Follow the setup instructions in Launching GDB Standalone.

At this point, the first terminal running the `sdx_server` should provide a "GDB listener port NUM" on standard out. Keep track of the number returned by the `sdx_server` as the GDB listener port is used by GDB to debug the kernel process. When the GDB listener port is printed, the spawned kernel process has attached to the `sdx_server` and is waiting for commands from you. To control this process, you must start a new instance of GDB and connect to the `sdx_server`.

IMPORTANT!: *If the `sdx_server` is running, then all spawned processes compiled for debug will connect and wait for control from you. If no GDB ever attaches or provides commands, the kernel code appears to hang.*

- In a third terminal, run the `xgdb` command, and at the GDB prompt, run the following commands:
  - For Software Emulation:

    ```
    "file ${XILINX_SDX}/data/emulation/cpu_em/generic_pcie/model/
    genericpciemodel"
    ```

  - For Hardware Emulation:
    1. Locate the `sdx_server` temporary directory:`/tmp/sdx/$uid`.
    2. Find the `sdx_server` process id (PID) containing the DWARF file of this debug session.
    3. At the `gdb` command line, run: `file /tmp/sdx/$uid/$pid/NUM.DWARF`
- In either case, connect to the kernel process:

  ```
  target remote :NUM
  ```

  Where `NUM` is the number returned by the `sdx_server` as the GDB listener port.

After these commands are executed you can set breakpoints on your kernels as needed, run the `continue` command, and debug your kernel code. When the all kernel invocations have finished, the host code should continue, and the connection will drop.

In case of the software and hardware emulation flows, there are restrictions with respect to the accelerated kernel code debug interactions. As this code is preprocessed, in the software emulation flow and actually translated in the hardware emulation flow into a hardware description language (HDL) and simulated during debugging, it is not always possible to set breakpoints at all locations. Especially with hardware emulation, only a limited number of breakpoints such as on preserved loops and functions are supported. Nevertheless, this mode is useful to debug the kernel/host interface.

TIP: *When debugging Software/Hardware emulation kernels in the SDAccel GUI Environment, 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.*

Xilinx GDB Extensions

Application Debug introduces new GDB commands, provided by Xilinx, that give visibility into the host application and into the platform IPs.

***Note*:** These commands need to be enabled as described in Launching GDB Standalone.

There are two groups of commands:

- The commands that give visibility into the OpenCL runtime data structures (`cl_command_queue`, `cl_event`, and `cl_mem`). Note that the arguments to "`xprint queue`" and "`xprint mem`" are optional. The application debug internally tracks the OpenCL objects and automatically prints all valid queues and `cl_mem` if the argument is not specified. In addition, the commands do a proper validation of supplied command queue, event, and `cl_mem` arguments.

```
xprint queue [<cl_command_queue>]
xprint event <cl_event>
xprint mem [<cl_mem>]
xprint kernel
xprint all
```

- The commands that give visibility into the IPs on the platform. Note, this functionality is only available in the “System” flow and not in any of the emulation flows.

```
xstatus all
xstatus --<ipname>
xstatus --<ipname>
```

The GDB commands allow the interaction between the host code and the kernel, which is managed through the OpenCL APIs, to be analyzed during debug. Help information about the commands can be obtained using `help <command>` (like `help xprint`).

A typical example for the use of these commands is if you are seeing a hang. In this case, the host application is likely waiting for the command queue to finish or waiting on an event list. Printing the command queue using `xprint` commands can tell you what events are unfinished and you can analyze the dependencies between the events.

Advanced Waveform-Based Kernel Debugging

The C/C++ code is synthesized using High Level Synthesis (HLS) into a Hardware Description Language (HDL) and later mapped onto the FPGA (`xclbin`). Especially the hardware- centric algorithm programmer will be very familiar with another debugging approach based on waveforms. This waveform-based HDL debugging is best supported by SDAccel™ through the GUI flow during Hardware Emulation.

***Note*:** For most debugging, the HDL model will not need to be analyzed. Waveform debugging is considered an advanced debugging capability.

To run this flow:

1. Start SDx, and perform the regular setup.
2. Select **Run → Debug Configurations** to open the Debug Configurations.
3. On the Debug Configurations window, select the current debug configuration from the OpenCL® (OCL) list.
4. On the Main tab, this displays two Kernel Debug Options: **Use RTL waveform** for kernel debugging and **Launch live waveform**. Checkmark both, close the configuration window, and a debug session starts automatically.

After the hardware emulation compilation process is complete, the waveform viewer automatically opens. By default, the waveform viewer shows all interface signals and the following debug hierarchy:

• Kernel Data Transfer: This section shows AXI transfers at the OCL masters to the DDR. Data transfers from all compute units funnel through these interfaces.
These interfaces could be a different bit width than 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>*

- ◦ CU Stalls (%): This section shows a summary of stalls for the entire compute unit (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.
- ◦ Loop Pipeline Activity: This section shows the top-level loop pipeline signals for a CU. This section is only populated for flat CUs
- ◦ Data Transfers: This section shows the data transfers for all AXI masters on the CU.
- ◦ User Functions: This section lists all of the functions within the hierarchy of the CU.

Function: `<function name>`

Function Stalls: This section lists the three stall signals within this function.

Loop Pipeline Activity: This section shows the function-level loop pipeline signals for a CU. The number of pipeline regions is dictated by Vivado® HLS and can typically be identified as compute sections of loops

Function I/O: This section lists the I/O for the function. These I/O are of protocol `-m_axi`, `ap_fifo`, `ap_memory`, or `ap_none`.

***Note*:** 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 restart, HDL breakpoints, as well as HDL code lookup and waveform markers are supported.

The waveform debugging process can also enabled through the XOCC/Makefile flow. Use the following instructions to enable it:

1. Turn on debug code generation during kernel compilation.

```
xocc -g ...
```

2. Create an `sdaccel.ini` file in the same directory as the host executable with the contents below:

```
[Emulation]
launch_waveform=batch

[Debug]
profile=true
timeline_trace=true
device_profile=true
```

3. Execute Hardware Emulation. The hardware transaction data will be collected in the file named <*hardware_platform*>-<*device_id*>-<*xclbin_name*>.wdb file. This file can directly be opened through the SDAccel GUI.

***Note*:** If the `launch_waveform` option is set to `gui` in the emulation section: `[Emulation] launch_waveform=gui`, then a live waveform viewer will be spawned during the execution of the Hardware Emulation.

Chapter 5

Compilation Flow

The Xilinx® SDAccel™ Environment is used for creating and compiling applications using the OpenCL® framework onto a Xilinx FPGA. This tool suite provides a software development environment for algorithm development and emulation on x86 based workstations, as well as deployment mechanisms for Xilinx FPGAs.

The compilation of OpenCL applications into binaries for execution on an FPGA does not assume nor require FPGA design knowledge. A basic understanding of the capabilities of an FPGA is necessary during application optimization in order to maximize performance. The SDAccel environment handles the low-level details of program compilation and optimization during the generation of application specific compute units for an FPGA fabric. Therefore, using the SDAccel Environment to compile an OpenCL program does not place any additional requirements on the user beyond what is expected for compilation on a CPU or GPU target.

TIP: *The SDAccel Xilinx Open Code Compiler (XOCC), `xocc`, is a command line compiler that takes your source code and runs it through the Xilinx implementation tools to generate the bitstream and other files that are needed to program the FPGA-based accelerator boards. It supports kernels expressed in OpenCL C, C++ and RTL (SystemVerilog, Verilog, or VHDL).*

An OpenCL program can be compiled using standalone command line compilers (`xcpp` for host code and `xocc` for kernels) in the SDAccel development environments. This chapter details the compilation flow.

Creating the Xilinx OpenCL Compute Unit Binary Container

The main difference between targeting an OpenCL® application to a CPU/GPU and targeting an FPGA is the source of the compiled kernel. Both CPUs and GPUs have a fixed computing architecture onto which the kernel code is mapped by a compiler. Therefore, OpenCL programs targeted for either kind of device invokes just-in-time compilation of kernel source files from the host code. The API for invoking just-in-time kernel compilation is as follows:

```
clCreateProgramWithSource(…)
```

IMPORTANT!: *`clCreateProgramWithSource` function is not supported for kernels targeting FPGA.*

In contrast to a CPU or a GPU, consider an FPGA as a blank computing canvas onto which a compiler generates an optimized computing architecture for each kernel in the system. This inherent flexibility of the FPGA allows the developer to explore different kernel optimizations and compute unit combinations that are beyond what is possible with a fixed architecture. The only drawback to this flexibility is that the generation of a kernel-specific optimized compute architecture takes a longer time than what is acceptable for just-in-time compilation. The OpenCL standard addresses this fundamental difference between devices by allowing for an offline compilation flow.

The SDAccel™ Environment uses this offline compilation flow to generate kernel binaries. To maximize efficiency in the host program and allow the simultaneous instantiation of kernels that cooperate in the computation of a portion of an application, Xilinx® has defined the Xilinx OpenCL Compute Unit Binary format, `.xclbin`. The `xclbin` file is a binary library of kernel compute units that will be loaded together into an OpenCL context for a specific device. This format can hold either programming files for the FPGA or shared libraries for the processor. It also contains library descriptive metadata used by the Xilinx OpenCL runtime library during program execution.

This lets you generate libraries of kernels that can be loaded and executed by the host program. The OpenCL APIs for supporting kernels generated in an offline compilation flow are, as follows:

```
// An offline binary container may be loaded with
// clCreateProgramWithBinary()

cl_program p  =  clCreateProgramWithBinary(binary1);
clBuildProgram(p);

// Use the program:
// create queues, create buffers,
// transfer data to device input buffers,
// run kernels,
// read back output buffer from device...

// when done Release the program
clReleaseProgram(p);
```

Only after `clReleaseProgram()` a new call to `clCreateProgramWithBinary()` can be used to create a new program from another binary container.

IMPORTANT!: *With dynamic memory topology in version 5.0 and later, the OpenCL runtime does not allow loading a new binary container after it is done with the first one if any `cl_mem` from the first run still exist. This means applications are not able to load a new binary container unless they release all `cl_mems` with `clReleaseMemObject()` before another call to `clCreateProgramWithBinary()` in the same process.*

The library metadata included in the `xclbin` file is automatically generated by the SDAccel Environment and does not require user intervention. This data is composed of compute unit descriptive information that is automatically generated during compute unit synthesis and used by the runtime to understand the contents of an `xclbin` file.

The `xclbin` file is created by the Xilinx OpenCL Compiler (`xocc`) command line utility, which provides mechanisms modeled after `gcc` and is composed of two separate operating modes:

1. Compilation or build of kernel accelerator functions (described in C/C++/OpenCL language) into Xilinx object (`.xo`) files. This is the `-c/--compile` mode of `xocc`.
2. Linking several `.xo` files together with a platform to create the binary container needed by the host code. This is the `-l/--link` mode of `xocc`.

The `xocc` can be used standalone (or ideally in scripts or a build system like `make`), and also is fully supported by the SDx IDE. See the *SDAccel Environment Tutorial: Introduction* (UG1021)for more information.

IMPORTANT!: *With the current version of SDAccel, `xocc` must be invoked twice: once for compilation and once for linking; it cannot be used to perform both compilation and linking in a single invocation.*

Compiling Your OpenCL Kernel Using the Xilinx OpenCL Compiler (xocc)

The Xilinx® OpenCL® Compiler (`xocc`) is a standalone command line utility for compiling kernel accelerator functions and linking them with SDAccel Environment supported platforms.

The first activity in building any system is to select an acceleration platform supported by Xilinx or third-party providers, and to compile a kernel accelerator function using the `-c/--compile` option.

TIP: *The default output name for the `.xo` file is `a.xo`; rename the file so that it relates to the kernel.*

The `-c/--compile` command syntax is, as follows:

```
xocc -c --platform <platform_name> <kernel_source_file> -o
<xo_kernel_name>.xo
```

TIP: *OpenCL uses the kernel keyword to identify a kernel, but for C/C++ kernels, we need to provide the kernel name by `--kernel <kernel_name>`.*

The second activity is to link one or more kernels into the platform to create the binary container `xclbin` file using the `-l/--link` option.

TIP: *The default output name for the `xclbin` file is `a.xclbin`; rename it as needed.*

The `-l/--link` command syntax is, as follows:

```
xocc -l --platform <platform_name> <xo_kernel1_name>.xo \
[<xo_kernel2_name>.xo ..] -o<xclbin_name>.xclbin
```

In SDAccel 2017.4, the provided DSA are `xilinx_kcu1500_dynamic_5_0` and `xilinx_vcu1525_dynamic_5_0`; for other tool releases a complete list of supported platforms is included in the *SDx Environments Release Notes, Installation, and Licensing Guide* (UG1238).

The `--platform` option accepts either a platform name or alternatively an `xpfm` filename (using full or relative path) that represents a platform. This is needed when you use a platform that is not included by default in the SDAccel tool installation, for example to use a DSA and matching board from the 2017.1/2017.2 release.

The default `-target` mode is the hardware (hw) system mode and can be changed with the `--target <sw_emu|hw_emu|hw>` option.

IMPORTANT!: *The same target must be used for each pair of compile/link runs. For example, you cannot mix compiling for `sw_emu` then linking for `hw_emu`.*

Table 2: **XOCC Common Options (For Compile and Link Modes)**

Option	Valid Values	Description
`--platform <arg>`	Supported acceleration platforms by Xilinx and third-party board partners.	Required. Sets the target Xilinx device. For a list of supported devices, see the SDAccel Supported Devices in the *SDx Environments Release Notes, Installation, and Licensing Guide* (UG1238).
`--list_xdevices`	N/A	Lists the supported devices.
`--target <arg>`	[`sw_emu` | `hw_emu` | `hw`]	Specify a compile target. • `sw_emu`: Software emulation • `hw_emu`: Hardware emulation • `hw`: Hardware Default: `hw`

Table 2: **XOCC Common Options (For Compile and Link Modes)** *(cont'd)*

Option	Valid Values	Description
`-o/-output <arg>`	File name with `.xo` or `.xclbin` extension depending on mode	Optional Set output file name. Default: • `a.xo` for compile mode • `a.xclbin` for link and build mode
`--version`	N/A	Prints the version and build information.
`--help`	N/A	Print help.
`--kernel_frequency`	Frequency (MHz) of the kernel.	Sets a user defined clock frequency in MHz for the kernel, overriding a default value from the DSA.
`--pk <arg>`	<[`kernel_name`|`all`] : [`none`|`stream`|`pipe`|`memory`|`all`]	Optional. Set a stall profile type for the given kernel(s). Default: none. For example: `--pk <kernal_name>:stream`

`--profile_kernel <arg>`	<data:[`kernel_name`|`all`]:[`compute_unit_name`|`all`]:[`interface_name`|`all`]>	Optional.
		Enable profiling DDR memory traffic for kernel and host. Default: none. For example:
		`--profile_kernel data:krnl1:cu1:m_axi_gmem0`
		`--profile_kernel data:krnl2:cu2:m_axi_gmem`

`--xp`	Refer to XP Parameters.	Specify detailed parameter and property settings in the Vivado tool suite used to implement the FPGA hardware. Familiarity with the Vivado tool suite is recommended in order to make the most use of these parameters.
`-g/--debug`	N/A	Add debug constructs to the code being compiled.
`--log`	N/A	Creates a log for in the current working directory.
`--message-rules <arg>`	Message rule file name	Optional. Specify a message rule file with message controlling rules. See Using the Message Rule File for more details.
`--report <arg>`	[`estimate` | `system`]	Generate a report type specified by <*arg*>: • `estimate`: Generate estimate report in `report_estimate.xtxt` • `system`: Generate the estimate report and detailed hardware reports in the report directory.
`--save-temps`	N/A	Save intermediate files/directories created during the compilation and build process.
`--report_dir <arg>`	Directory	Specify a report directory. If the `--report` option is specified, the default is to generate all reports in the current working directory (cwd).

Table 2: **XOCC Common Options (For Compile and Link Modes)** *(cont'd)*

Option	Valid Values	Description
`--log_dir <arg>`	Directory	Specify a log directory. If the `--log` option is specified, the default is to generate the log file in the current working directory (cwd).
`--temp_dir <arg>`	Directory	Specify a temp directory. If the `--save-temps` option is specified, the default is to create the temporary compilation and build files in the current working directory (cwd).
`--export_script`	N/A	This option allows detailed control of the Vivado tool suite used to implement the FPGA hardware. Familiarity with the Vivado tool suite is recommended in order to make the most use of the Tcl file generated by this option. Generates the Tcl script, <*kernel_name*>`.tcl`, used to execute Vivado HLS but halts before Vivado HLS starts. The expectation is for the script to be modified and used with the `--custom_script` option. Not supported for `-t sw_emu` with OpenCL kernels.
`--custom_script`	<*kernel_name*>:<*path to kernel Tcl file*>	Intended for use with the <*kernel_name*>`.tcl` file generated with `--export_script`. This option allows you to customize the Tcl file used to create the kernel and execute using the customize version of the script.
input file	OpenCL or C/C++ kernel source file, or Xilinx object file (`.xo`).	Compile kernels into `a.xo` or `a.xclbin` file depending on the `xocc` mode.
`--user_ip_repo_paths <arg>`	Specify the directory location Specify existing user IP repository.	This value will be prefixed to `ip_repo_paths`.
`--remote_ip_cache<arg>`	Specify the directory location	Specify remote IP cache directory for Vivado Synthesis.
`--no_ip_cache`	N/A	Turn off IP cache for Vivado Synthesis.

Table 3: **XOCC Options for Compile Mode**

Option	Valid Values	Description
`-c/--compile`	N/A	Required, but mutually exclusive with `--link`. Run `xocc` in compile mode, generate `.xo` file.
`-k/--kernel <arg>`	Kernel to be compiled from the input `.cl` or `.c/.cpp` kernel source code.	Required for C/C++ kernels. Optional for OpenCL kernels. Compile/build only the specified kernel from the input file. Only one `-k` option is allowed per command. When an OpenCL kernel is compiled without the `-k` option, all the kernels in the input file are compiled.
`--define <arg>`	Valid macro name and definition pair. <*name*>=<*definition*>	Predefine name as a macro with definition. This option is passed to the `xocc` preprocessor.
`--include`		

Table 4: **XOCC Options for Link Mode**

Option	Valid Values	Description
`--optimize <arg>`	Valid optimization levels: 0, 1, 2, 3, s, quick `example: --optimize2` This option ONLY applies to Vivado. The compile step runs the C code through HLS and has no bearing.	These options control the default optimizations performed by the Vivado hardware synthesis engine. Familiarity with the Vivado tool suite is recommended to make better use of these settings. • 0: Default optimization. Reduce compilation time and make debugging produce the expected results. • 1: Optimize to reduce power consumption. This takes more time to compile the design. • 2: Optimize to increase kernel speed. This option increases both compilation time and the performance of the generated code. • 3: This is the highest level of optimization. This option provides the highest level performance in the generated code, but compilation time may increase considerably. • s: Optimize for size. This reduces the logic resources for the kernel • quick: Quick compilation for fast run time. This may result in reduced hardware performance, and a greater use of resources in the hardware implementation.
`--link`	N/A	Required, but mutually exclusive with `--compile`. Run `xocc` in link mode. Link `.xo` input files, generate `.xclbin` file.
`--dk <arg>`	<protocol:<*compute_unit_name*>:<*interface_name*>> For example: `protocol:cu1:m_axi_gmem`	Add Lightweight AXI Protocol Checkers (LAPC) to the specified compute units to monitor the specified interfaces.

Table 4: **XOCC Options for Link Mode** *(cont'd)*

Option	Valid Values	Description

`--nk <arg>`	<*kernel_name*>: <*compute_units*>
	For example:
	`foo:2`
	<*kernel_name*>:<*compute_units*>:<*kernel_name1*>.<*kernel_name2*>...<*kernel_nameN*>
	For example:
	`foo:3:fooA.fooB.fooC`
		Optional in link mode. Not applicable in compile mode.
		The instance name is optional. If the instance name is not specified, the first instance is named <*kernel_name*>_*1*, the second instance will be named <*kernel_name*>_*2*, and so forth.
		Instantiate the specified number of compute units for the given kernel in the `.xclbin` file.
		Default: One compute unit per kernel.

`--jobs <arg>`	Number of parallel jobs.	Optional. This option allows detailed control of the Vivado tool suite used to implement the FPGA hardware. Familiarity with the Vivado tool suite is recommended in to make the most use of the Tcl file generated by this option. Specify the number of parallel jobs to be passed to the Vivado tool suite for implementation. Increasing the number of jobs allows the hardware implementation step to spawn more parallel processes and complete faster.
`--lsf <arg>`	`bsub` command line to pass to LSF cluster. This argument is required for use with `--lsf`.	Optional. Use IBM Platform Load Sharing Facility (LSF) for Vivado implementation and synthesis. For example: `--lsf '{bsub -R "select[type=X86_64]" -N -q medium}'`
`--reuse_synth <arg>`	<*synthesized DCP*>	Import a previously synthesized DCP and only run the Vivado tool implementation and the `XCLBIN` packaging.
`--reuse_impl <arg>`	<*implemented DCP*>	Import a previously implemented DCP and only run the `XCLBIN` packaging.
`--sp <arg>`	<*kernel_inst_name*>.<*interface_name*>:<*bank*>Valid DDR4 bank names are `bank0`, `bank1`, `bank2`, and `bank3` for platforms with 4 DDR banks.	For DSA 5.x and later, the `--xp misc:map_connect` option is deprecated and replaced with the system port `--sp` option with a much simpler syntax requiring only the kernel instance name, an interface name of that kernel. and the target DDR bank for the required connection. Multiple `--sp` options can be specified to map each of the interfaces to a particular bank.

IMPORTANT!: *All examples in the SDAccel installation use Makefile to compile OpenCL applications with `xcpp` and `xocc` commands that can be used as references for compiling user applications.*

XP Parameters

Use the `--xp` switch to specify parameter values in SDAccel™. These parameters allow fine grain control over the hardware generated by SDAccel and the hardware emulation process.

IMPORTANT!: *Familiarity with the Vivado® tool suite is recommended to make the most use of these parameters. See Vivado Design Suite User Guide: High-Level Synthesis (UG902) and Vivado Design Suite User Guide: Implementation (UG904) for more information.*

Parameters are specified as `param:<param_name>=<value>`. For example:

```
xocc --xp param:compiler.enableDSAIntegrityCheck=true
-xp param:prop:kernel.foo.kernel_flags="-std=c++0x"
```

you can specify the `--xp` command option multiple times in a single `xocc` invocation, or specify the value(s) in an `xocc.ini` file with each option specified on a separate line (without `--xp` switch).

```
param:prop:solution.device_repo_paths=../dsa
param:compiler.preserveHlsOutput=1
```

Upon invocation, `xocc` first looks for an `xocc.ini` file in the `$HOME/.Xilinx/sdx` directory. If the file does not exist there, `xocc` will then look for it in the current working directory. If the same `--xp` parameter value is specified in both the command line and `xocc.ini` file, the command line value will be used.

The following table lists the `--xp` parameters and their values.

Table 5: **XP Parameter Options**

Parameter Name	Type	Default Value	Description
`param:compiler.acceleratorBinaryContent`	String	<empty>	The content to put in `xclbin`.Valid options are `bitstream` and `dcp`
`param:compiler.enableDSAIntegrityCheck`	Boolean	False	Enables the DSA Integrity Check. If this value is set to True, and SDAccel™ detects a DSA which has been modified outside of the Vivado® tool suite SDAccel halts operation.
`param:compiler.errorOnHoldViolation`	Boolean	True	Error out if there is hold violation.
`param:compiler.maxComputeUnits`	Int	`-1`	The maximum compute units allowed in the system. Any positive value will overwrite the `numComputeUnits` setting in the DSA.
`param:hw_em.enableProcSyncReset`	Boolean	false	Enable `proc_sync_reset` in `hw_em`
`param.hw_emplatformPath`	String	<empty>	

Table 5: **XP Parameter Options** *(cont'd)*

Parameter Name	Type	Default Value	Description
`param:hw_em.compiledLibs`	String	<empty>	Uses mentioned `clibs` for the specified simulator.
`param:hw_em.debugLevel`	String	OFF	The debug level of the simulator. Options are: • `OFF`: Used for optimized run times • `BATCH`: for batch runs • `GUI`: for use in GUI-mode
`param:hw_em.enableProtocolChecker`	Boolean	False	Enables the AXI protocol checker during HW emulation. This is used to confirm the accuracy of any AXI interfaces in the design.
`param:compiler.interfaceLatency`	Int	`-1`	This option specifies the expected latency on the kernel AXI bus, the number of clock cycles from when bus access is requested until it is granted.
`param:compiler.fsanitizestring`	string	<empty>	Software emulation runtime checks possible values: • `address` - Out of Bound Access • `memory` - Uninitialized Memory Access • `address` • `memory` - Both The software emulation run produces a debug log: `<project_dir>/Emulation-SW/<proj_name>-Default>/emulation_debug.log` with emulation diagnostic messages.
`param:compiler.xclDataflowFifoDepth`	Int	`-1`	Specifies the depth of FIFOs used in kernel dataflow region.
`param:compiler.interfaceWrOutstanding`	Int Range	0	Specifies how many outstanding writes to buffer are on the kernel AXI interface. Values are 1 through 256.
`param:compiler.interfaceRdOutstanding`	Int Range	0	Specifies how many outstanding reads to buffer are on the kernel AXI interface. Values are 1 through 256.
`param:compiler.interfaceWrBurstLen`	Int Range	0	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.

Table 5: **XP Parameter Options** *(cont'd)*

Parameter Name	Type	Default Value	Description
`param:compiler.interfaceRdBurstLen`	Int Range	0	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.
`misc:map_connect=<type>.kernel.<kernel_name>.<kernel_AXI_interface>.core.OCL_REGION_0.<dest_port>`	String	`<empty>`	Used to map AXI interfaces from a kernel to DDR memory banks. • `<type>` is add or remove. • `<kernel_name>` is the name of the kernel. • `<dest_port>` is a DDR memory bank M00_AXI, M01_AXI, M02_AXI or M03_AXI. This option available only for DSA 4.x and earlier,and deprecated for DSA 5.x and later: use system ports using the `--sp` documented in the `xocc Linker Options`.
`prop:kernel.<kernel_name>.kernel_flags`	String	`<empty>`	Sets specific compile flags on the kernel. *<kernel_name>*.
`prop:solution.device_repo_path`	String	`<empty>`	Specifies the path to the DSA repository. The `--platform` option with full path to the `.xpfm` platform file should be used instead.
`prop:solution.hls_pre_tcl`	String	`<empty>`	Specifies the path to a Vivado HLS Tcl file, which is executed before the C code is synthesized. This allows Vivado HLS configuration settings to be applied prior to synthesis.
`prop:solution.hls_post_tcl`	String	`<empty>`	Specifies the path to a Vivado HLS Tcl file, which is executed after the C code is synthesized.
`prop:solution.kernel_compiler_margin`	Float	12.5% of the kernel clock period.	The clock margin in ns for the kernel. This value is substracted from the kernel clock period prior to synthesis to provide some margin for P&R delays.

Table 5: **XP Parameter Options** *(cont'd)*

Parameter Name	Type	Default Value	Description

`vivado_prop:<object_type>.<object_name>.<prop_name>`	Various	Various	This allows you to specify any property used in the Vivado hardware compilation flow.`Object_type` is `run|fileset|file|project`The `object_name` and `prop_name` values are described in *Vivado Design Suite Properties Reference Guide*, (UG912)
			Examples:
			`vivado_prop:run.impl_1.{STEPS.PLACE_DESIGN.ARGS.MORE OPTIONS}={-fanout_opt}`
			`vivado_prop:fileset.current.top=foo`
			For `object_type` file, `current` is not supported.
			For object type `run` the special value of `__KERNEL__` can be used to specify run optimization settings for 'ALL' kernels, instead of the need to specify them one by one.

Running Software and Hardware Emulation in XOCC Flow

In the XOCC/Makefile flow, users manage compilation and execution of host code and kernels outside the Vivado® Xilinx SDAccel™ development environment. Follow the steps below to run software and hardware emulation:

1. Create the emulation configuration file.

   For software or hardware emulation, the runtime library needs to know what devices and how many to emulate. This information is provided to the runtime library by an emulation configuration file. SDAccel provides a utility, `emconfigutil` to automate creation of the emulation configuration file. The following are details of the `emconfigutil` command line format and options:

Option	Valid Values	Description
`--xdevice`	Target device	Required: Set target device. See the *SDx Environments Release Notes, Installation, and Licensing Guide* (UG1238) for all supported devices. This command is deprecated in Release 2017.4, and replaced with the `--platform`command.
`--nd`	Any positive integer	Optional: Number of devices. Default is 1.

Option	Valid Values	Description
`--od`	Valid directory	Optional: Output directory, `emconfig.json` file must be in the same directory as the host executable.
`--xp`	Valid Xilinx parameters and properties	Optional: Specify additional parameters and properties. For example: `--xp prop:solution.device_repo_paths=my_dsa_path` Sets the search path for the device specified in --xdevice option.
`-h`	NA	Print help messages

The `emconfigutil` command creates the configuration file `emconfig.json` in the output directory.

The `emconfig.json` file must be in the same directory as the host executable. The following example creates a configuration file targeting two `xilinx_vcu1525_dynamic_5_0` devices.

```
$emconfigutil --xdevice xilinx_vcu1525_dynamic_5_0
 --nd 2
```

2. Set `XILINX_SDX` environment variable

   The `XILINX_SDX` environment needs to be set and pointed to the SDAccel installation path for the emulation to work. Below are examples assuming SDAccel is installed in `/opt/Xilinx/SDx/2017.4`

   C Shell:

   ```
   setenv XILINX_SDX /opt/Xilinx/SDx/2017.4
   ```

   Bash:

   ```
   export XILINX_SDX=/opt/Xilinx/SDx/2017.4
   ```

3. Set emulation mode

   Setting `XCL_EMULATION_MODE` environment variable to `sw_emu` or `hw_emu` changes the application execution to emulation mode, `sw_emu` for software emulation or `hw_emu` for hardware emulation, so that the runtime looks for the file `emconfig.json` in the same directory as the host executable and reads in the target configuration for the emulation runs.

   C Shell:

   ```
   setenv XCL_EMULATION_MODE sw_emu
   ```

Bash:

```
export  XCL_EMULATION_MODE=sw_emu
```

Unsetting the `XCL_EMULATION_MODE` environment variable will turn off the emulation mode.

4. Run Software or Hardware emulation

   With the configuration file `emconfig.json` and `XCL_EMULATION_MODE` set to `sw_emu`, execute the host application with proper arguments to run Software emulation. Alternatively, use setting `hw_emu` to run hardware emulation.

```
$./host.exe kernel.xclbin
```

Running an Application on FPGA

Use the following steps to run an application on an FPGA with host executable and kernel binaries compiled in the XOCC flow.

1. Source the `setup.sh` (Bash) or `setup.csh` (Csh/Tcsh) script file generated by the `xbinst` utility. See Appendix J: Board Installations for more details.
2. Run the application: `$./host.exe kernel.xclbin`

Using the Message Rule File

XOCC executes various Xilinx tools during kernel compilation. These tools generate many messages that provide compilation status to the users. These messages may or may not be relevant to all users depending on users' focus and design phases. A Message Rule file can be used to better manage these messages. It provides commands to promote important messages to the terminal or suppress unimportant ones. This will help users better understand the kernel compilation result and explore ways to optimize the kernel.

The Message Rule file (`.mrf`) 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-whitespace character is a comment.

Supported Commands

By default, xocc recursively scans the entire working directory and promotes all error messages to the xocc output. The `promote` and `suppress` commands below provide more control on the xocc output.

`promote`

This command indicates that matching messages should be promoted to the xocc output.

`suppress`

This command indicates that matching messages should be suppressed or filtered from the xocc output. Note that errors can not be suppressed.

Command Options

The Message Rule file can have multiple promote and suppress commands. Each command can have one and only one of the options below. The options are case sensitive.

- `-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: [XOCC 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]
  ```

- `-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`

Precedence of Message Rules

`suppress` rules take precedence over `promote` rules. If the same message ID or severity level is passed to both `promote` and `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
```

Appendix A

Getting Started with Examples

All Xilinx® SDx™ Environments are provided with examples designs. These examples can:

- Be a useful learning tool for both the SDx 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.

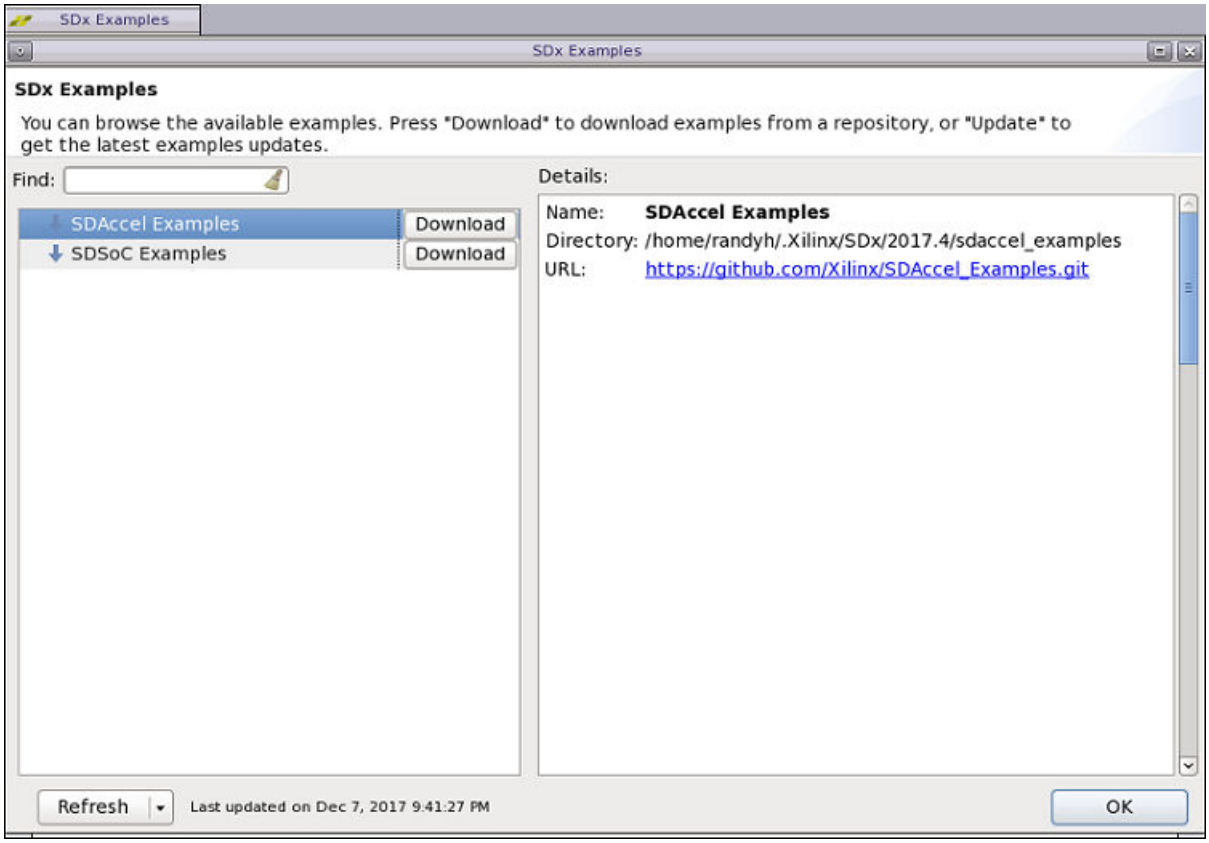
Many examples are available to be downloaded from the Xilinx GitHub repository, although a limited number are also included in the `samples` folder of the software installation.

Installing Examples

You will be asked to select a template for new projects when working through the **New SDx Project** wizard. You can also load template projects from within an existing project, through the **Xilinx → SDx Examples** command.

The first time the SDx Examples dialog box, or the Template page of the New SDx Project wizard is displayed, it will be empty if the SDAccel examples have not been downloaded. The empty dialog box is shown in the following figure. To add Templates to a new project, you must download the SDAccel Examples.

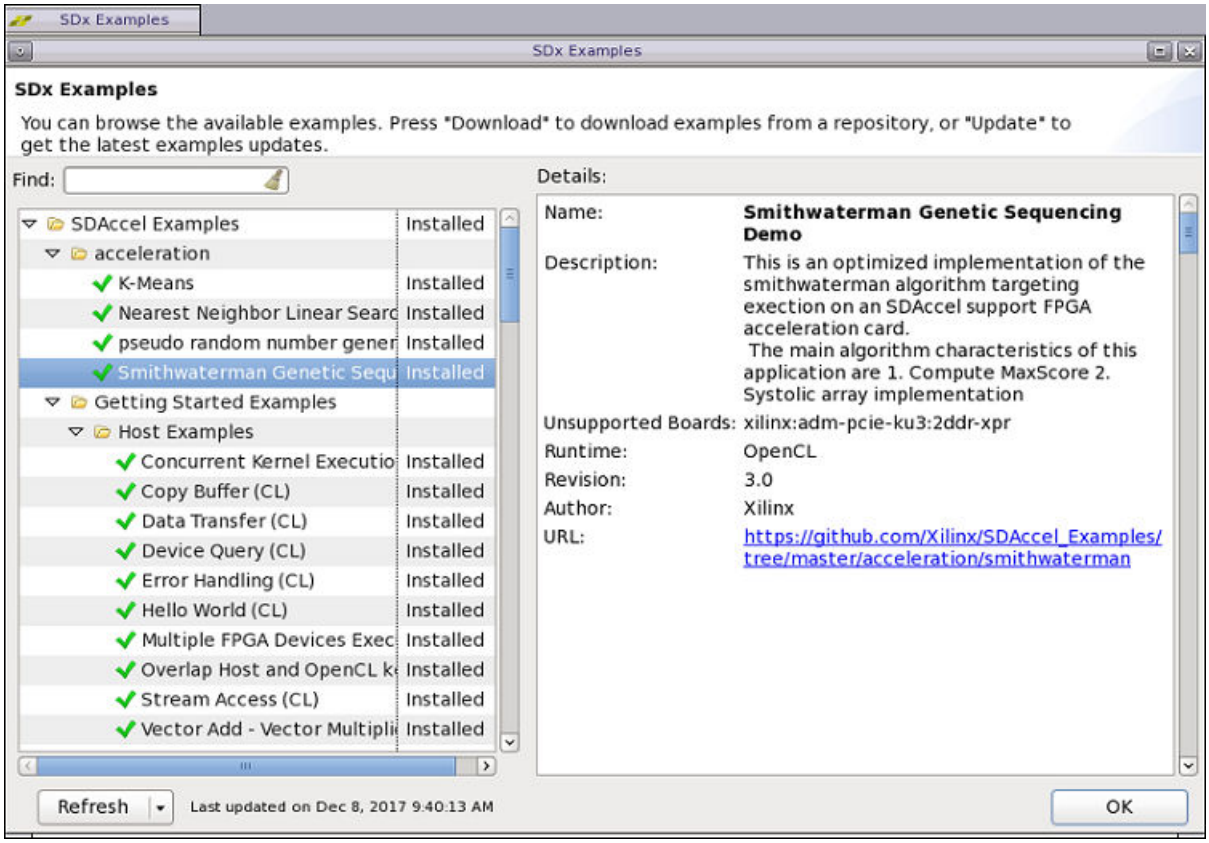
Figure 13: **SDAccel Examples - Empty**



The left side of the dialog box shows **SDAccel Examples** and **SDSoC Examples**, and has a download command for each category. The right side of the dialog box shows the directory where the examples will be downloaded to, and shows the URL where the examples will be downloaded from.

Click the **Download** button next to the **SDAccel Examples** to download the examples and populate the dialog. The examples are downloaded as shown in the following figure.

Figure 14: **SDx Examples - Populated**



The command menu at the bottom left of the SDx 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 GitHub repository.
- **Reset**: Deletes the downloaded examples from the `.Xilinx` folder.

Using Local Copies

While you must download the examples to add Templates when you create new projects, the SDx IDE always downloads the examples into your local `.Xilinx` folder. The download directory cannot be changed from the SDx Examples dialog box. However, you may want to download the example files to a different location from the `.Xilinx` folder for a number of reasons.

You can use the `git` command from a command shell to specify a new destination folder for the downloaded examples:

```
git clone https://github.com/Xilinx/SDAccel_Examples
<workspace>/examples
```

When you clone the `SDAccel_Examples` using the `git` command as shown above, you can use the example files as a resource of 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 SDx Project wizard. However, you will need to locate these files and make them local to your project manually.

You can find the needed files by searching for the file from the location of the cloned repository. For example, you could run the following command from the `examples` folder to find the `xcl2.hpp` file needed for the `vadd` example:

```
find -name xcl2.hpp
```

In addition, the Makefile in each of the example projects has a special command to localize any include files into the project. Use the following form of the `make` command in the example project:

```
make local-files
```

Appendix B

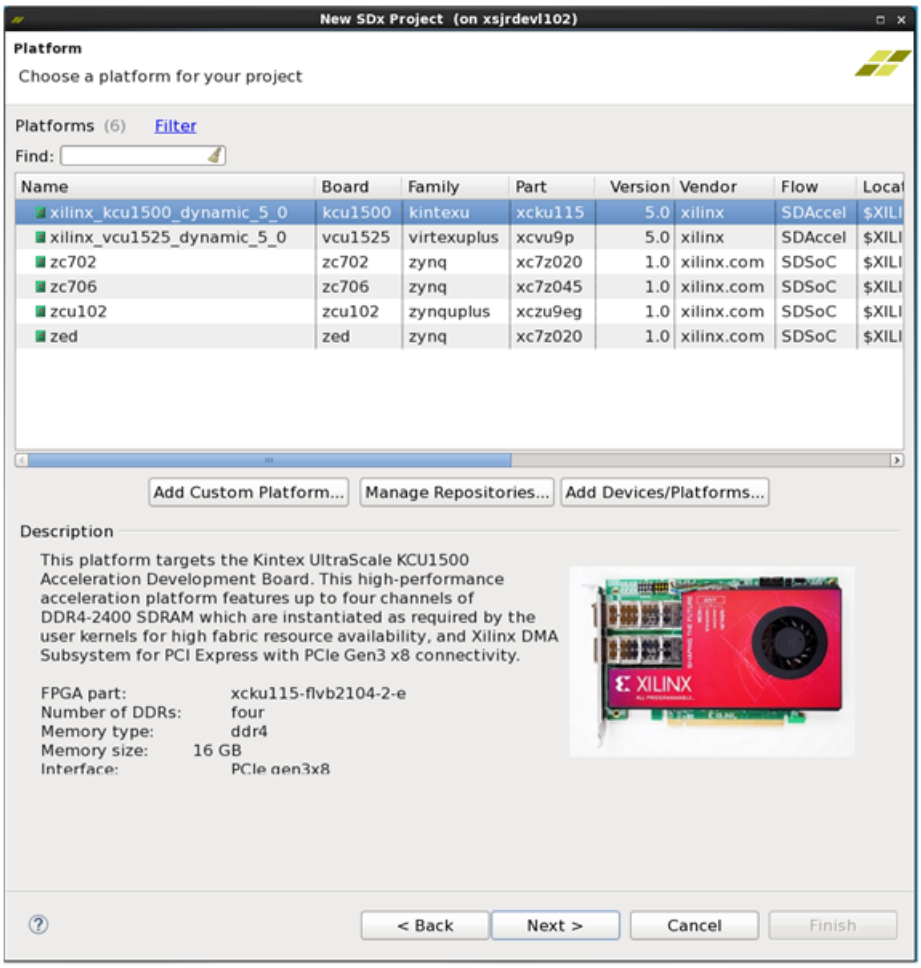
Managing Platforms and Repositories

When you are creating a project, you can manage the platforms that are available for use in SDx application projects, from the **Platform Selection** page of the SDx New Project wizard. This lets you add a new platform for a project as it is being created.

You can also manage the platforms and repositories from an opened project by clicking the Browse button () next to the **Platform** link in the **General** panel of the **Project Settings** window.

This opens the Hardware Platforms dialog box, which lets you manage the available platforms and platform repositories.

Figure 15: **Specify SDAccel Platform**



- **Add Custom Platform**: Lets you add your own platform to the list of available platforms.Navigate to the top-level directory of the custom platform, select it, and click **OK** to add the new platform. The custom platform is immediately available for selection from the list of available platforms. See *SDAccel Environment Platform Development Guide* (UG1164) for information on creating custom platforms. The **Xilinx → Add Custom Platform** command can also be used to directly add custom platforms to the tool.
- **Manage Repositories**: Lets you 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. Removing any platform from the list of repositories removes the platform from the list of available platforms.
- **Add Devices/Platforms**: Lets you manage which Xilinx® devices and platforms are installed. If a device or platform was not selected during the installation process, you can add it at a later time using this command. This command launches the SDx Installer to let you select extra content to install. The **Help → Add Devices/Platforms** command can also be used to directly add custom platforms to the tool.

Appendix C

Understanding the OpenCL Platform and Memory Model

The OpenCL™ standard describes all hardware compute resources capable of executing OpenCL applications using a common abstraction for defining a platform and the memory hierarchy. The platform is a logical abstraction model for the hardware executing the OpenCL application code. This model, which is common to all vendors implementing this standard, provides the application programmer with a unified view from which to analyze and understand how an application is mapped into hardware. Understanding how these concepts translate into physical implementations on the FPGA is necessary for application optimization.

This chapter provides a review of the OpenCL platform model and its extensions to FPGAs. It explains the mapping of the OpenCL platform and memory model into an SDAccel™ development environment-generated implementation.

OpenCL Devices and FPGAs

In the context of CPU and GPU devices, the attributes of a device are fixed and the programmer has very little influence on what the device looks like. On the other hand, this characteristic of CPU/GPU systems makes it relatively easy to obtain an off-the-shelf board. The one major limitation of this style of device is that there is no direct connection between system I/O and the OpenCL™ kernels. All transactions of data are through memory-based transfers.

An OpenCL device for an FPGA is not limited by the constraints of a CPU/GPU device. By taking advantage of the fact that the FPGA starts off as a blank computational canvas, the user can decide the level of device customization that is appropriate to support a single application or a class of applications. In determining the level of customization in a device, the programmer needs to keep in mind that kernel compute units are not placed in isolation within the FPGA fabric.

FPGAs capable of supporting OpenCL programs consist of the following:

- Connection to the host processor
- I/O peripherals
- Memory controllers

- Interconnect
- Kernel region

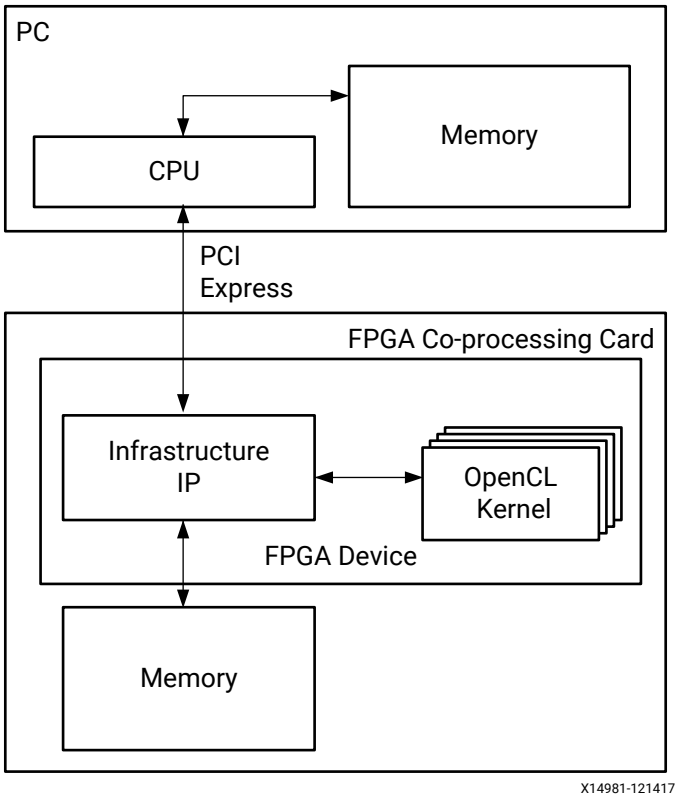
The creation of FPGAs requires FPGA design knowledge and is beyond the scope of capabilities for the SDAccel™ Environment. Devices for the SDAccel Environment are created using the Xilinx® Vivado® Design Suite for FPGA designers. The SDAccel Environment provides pre-defined devices and allows users to augment the tool with third party created devices. A methodology guide describing how to create a device for the SDAccel development environment is available upon request from Xilinx.

The devices available in the SDAccelEnvironment are for Virtex®-7, Kintex®-7, and UltraScale™ FPGAs. These devices are available in a PCIe® form factor. The PCIe form factor for Virtex-7, Kintex-7, and UltraScale devices assumes that the host processor is an x86- or Power8-based processor, and that the FPGA is used for the implementation of compute units.

Using a PCIe Reference Device

The PCIe™ base device has a distributed memory architecture, which is also found in GPU accelerated compute devices. This means that the host and the kernels access data from separate physical memory domains. Therefore, the developer has to be aware that passing buffers between the host and a device triggers memory data copies between the physical memories of the host and the device. The data transfer time must be accounted for when determining the best optimization strategy for a given application. A representative example of this type of device is shown in the following figure.

Figure 16: **PCIe Base Device**



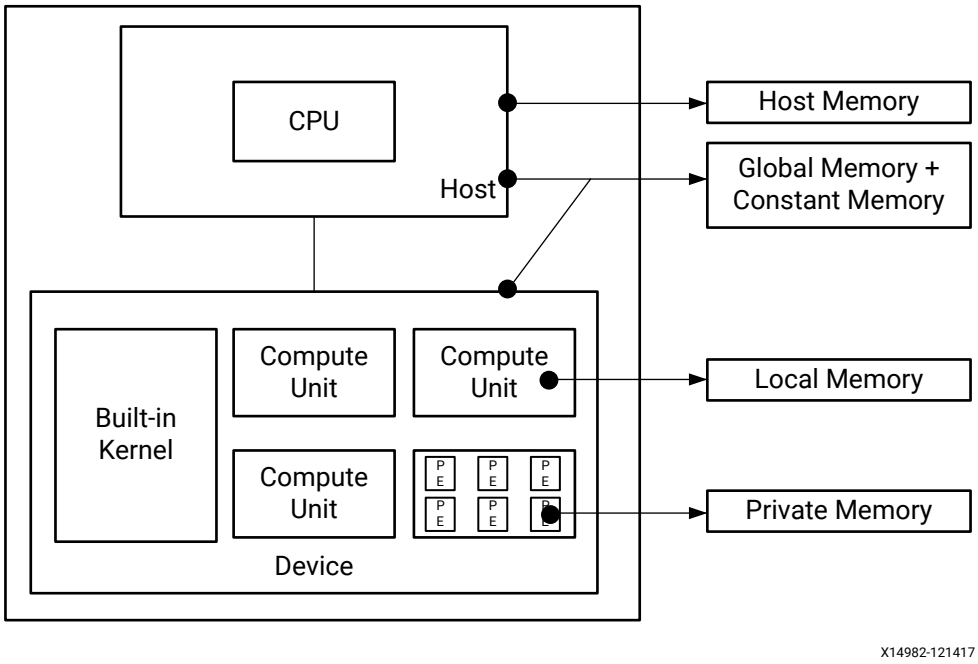
The main characteristics of devices with a PCIe form factor are as follows:

- The x86 or Power8 processor in the PC is the host processor for the OpenCL™ application.
- The infrastructure IP provided as part of the device is needed for communication to the host over the PCIe core and to access the DDR memories on the board.
- Connecting OpenCL kernels to IP other than infrastructure IP or blocks generated by the SDAccel™ development environment is not supported.
- Kernels work on data in the DDR memory attached to the FPGA.

OpenCL Memory Model

The OpenCL™ API defines the memory model to be used by all applications that comply with the standard. This hierarchical representation of memory is common across all vendors and can be applied to any OpenCL application. The vendor is responsible for defining how the OpenCL memory model maps to specific hardware. The OpenCL memory model is shown overlaid onto the OpenCL device model in the following figure.

Figure 17: **OpenCL Memory Model**



The memory hierarchy defined in the OpenCL specification has the following levels:

- Host Memory
- Global Memory
- Constant Global Memory
- Local Memory
- Private Memory

Host Memory

The host memory is defined as the region of system memory that is only visible and accessible to the host processor. The host processor has full control of this memory space and can read and write from this space without any restrictions. Kernels cannot access data located in this space. Any data needed by a kernel must be transferred into global memory so that it is accessible by a compute unit.

Global Memory

The global memory is defined as the region of system memory that is accessible to both the OpenCL™ host and device. The host is responsible for the allocation and deallocation of buffers in this memory space. There is a handshake between host and device over control of the data stored in this memory. The host processor transfers data from the host memory space into the global memory space. Then, when a kernel is launched to process the data, the host loses access rights to the buffer in global memory. The device takes over and is capable of reading and writing from the global memory until the kernel execution is complete. Upon completion of the operations associated with a kernel, the device turns control of the global memory buffer back to the host processor. After it has regained control of a buffer, the host processor can read and write data to the buffer, transfer data back to the host memory, and deallocate the buffer.

You can use the `clCreateSubDevices` function to create a sub-device for each compute unit mapped to a specific DDR/Global memory. In the Xilinx® implementation each sub-device can only have one compute unit.

The following example shows how you can create 16 sub-devices:

```
int num_sub_devices = 16;

    const cl_device_partition_property properties[3] = {
            CL_DEVICE_PARTITION_EQUALLY  ,
           1,          // Use only one compute unit
            0          //CL_DEVICE_PARTITION_BY_COUNTS_LIST_END
    };

    cl_device_id subdevice_id;
    cl_device_id sub_device_ids[num_sub_devices];
    cl_uint num_devices_ret = 0;
    err = clCreateSubDevices(device_id, properties,
        num_sub_devices, sub_device_ids, &num_devices_ret);
    if (err != CL_SUCCESS) {
            fprintf(stderr, "failed to create sub device %d!\n", err);
            return 1;
    }
```

Specify the sub-devices using the `xocc` command option `--nk` to add multiple kernels. The following shows how this is used in a Makefile.

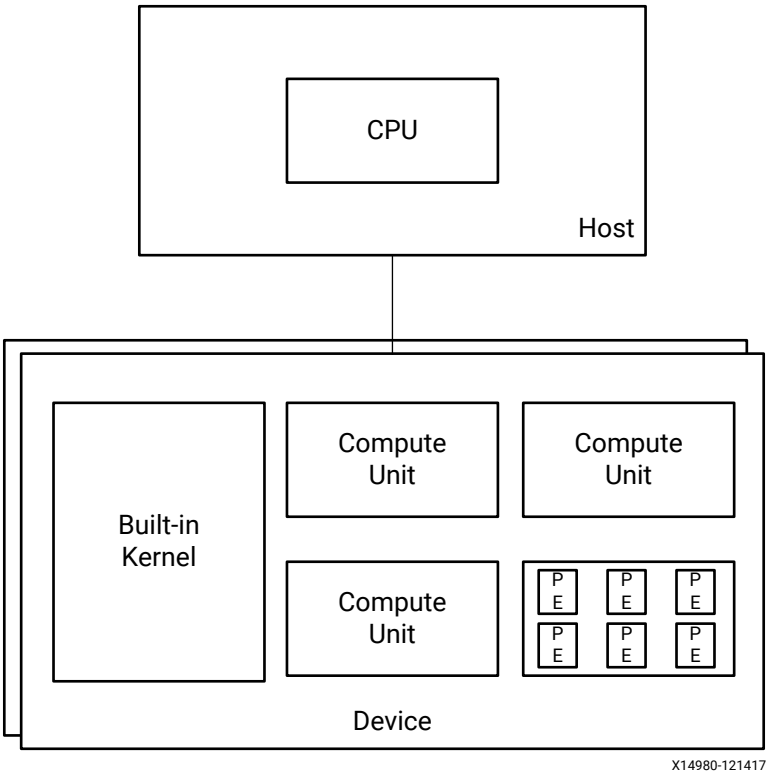
```
    KERNEL_1_COMPILE_FLAGS = --nk subf:16
    KERNEL_1_CU_1 = subf_1
    KERNEL_1_CU_2 = subf_2
    KERNEL_1_CU_3 = subf_3
    KERNEL_1_CU_4 = subf_4
    KERNEL_1_CU_5 = subf_5
    KERNEL_1_CU_6 = subf_6
    KERNEL_1_CU_7 = subf_7
    KERNEL_1_CU_8 = subf_8
    KERNEL_1_CU_9 = subf_9
    KERNEL_1_CU_10 = subf_10
    KERNEL_1_CU_11 = subf_11
    KERNEL_1_CU_12 = subf_12
```

```
    KERNEL_1_CU_13 = subf_13
    KERNEL_1_CU_14 = subf_14
    KERNEL_1_CU_15 = subf_15
    KERNEL_1_CU_16 = subf_16
    XOS_1 += $(KERNEL_1_XO)
    XOCC_LINK_OPTS_1 +=  --nk $(KERNEL_1):16:$(KERNEL_1_CU_1).
$(KERNEL_1_CU_2).
$(KERNEL_1_CU_3).
$(KERNEL_1_CU_4).
$(KERNEL_1_CU_5).
$(KERNEL_1_CU_6).
$(KERNEL_1_CU_7).
$(KERNEL_1_CU_8).
$(KERNEL_1_CU_9).
$(KERNEL_1_CU_10).
$(KERNEL_1_CU_11).
$(KERNEL_1_CU_12).
$(KERNEL_1_CU_13).
$(KERNEL_1_CU_14).
$(KERNEL_1_CU_15).
$(KERNEL_1_CU_16)
```

OpenCL Platform Model

The OpenCL™ platform model defines the logical representation of all hardware capable of executing an OpenCL program. At the most fundamental level all platforms are defined by the grouping of a processor and one or more devices. The host processor, which runs the OS for the system, is also responsible for the general bookkeeping and task launch duties associated with the execution of parallel programs such as OpenCL applications. The device is the element in the system on which the kernels of the application are executed. The device is further divided into a set of compute units. The number of compute units depends on the target hardware for a specific application. A compute unit is defined as the element in the hardware device onto which a work group of a kernel is executed. This device is responsible for executing the operations of the assigned work group to completion. In accordance to the OpenCL standard division of work groups into work items, a compute unit is further subdivided into processing elements. A processing element is the data path in the compute unit, which is responsible for executing the operations of one work item. A conceptual view of this model is shown in the following figure.

Figure 18: **OpenCL Platform Model**



An OpenCL platform always starts with a host processor. For the case of platforms created with Xilinx® devices, the host processor is an x86 or Power8 based processor communicating to the devices using a PCIe™ solution. The host processor has the following responsibilities:

- Manage the operating system and enable drivers for all devices.
- Execute the application host program.
- Set up all global memory buffers and manage data transfer between the host and the device.
- Monitor the status of all compute units in the system.

In all OpenCL platforms, the host processor tasks are executed using a common set of API functions. The implementation of the OpenCL API functions is provided by the hardware vendor and is referred to as the runtime library.

The OpenCL runtime library, which is provided by the hardware vendor, is the logical layer in a platform that is responsible for translating user commands described by the OpenCL API into hardware specific commands for a given device. For example, when the application programmer allocates a memory buffer using the `clCreateBuffer` API call, it is the responsibility of the runtime library to keep track of where the allocated buffer physically resides in the system, and of the mechanism required for buffer access. It is important for the application programmer to keep in mind that the OpenCL API is portable across vendors, but the runtime library provided by a vendor is not. Therefore, OpenCL applications have to be linked at compile time with the runtime library that is paired with the target execution device.

The other component of a platform is the device. A device in the context of an OpenCL API is the physical collection of hardware resources onto which the application kernels are executed. A platform must have at least one device available for the execution of kernels. Also, per the OpenCL platform model, all devices in a platform do not have to be of identical type.

Constant Global Memory

Constant global memory is defined as the region of system memory that is accessible with read and write access for the OpenCL™ host and with read-only access for the OpenCL device. As the name implies, the typical use for this memory is to transfer constant data needed by kernel computation from the host to the device.

Local Memory

Local memory is defined as the region of system memory that is only accessible to the OpenCL™ device. The host processor has no visibility and no control on the operations that occur in this memory space. This memory space allows read and write operations by the work items within the same compute unit. This level of memory is typically used to store and transfer data that must be shared by multiple work items.

Private Memory

Private memory is the region of system memory that is only accessible by a processing element within an OpenCL™ device. This memory space can be read from and written to by a single work item.

For devices using an FPGA, the physical mapping of the OpenCL memory model is the following:

- Host memory is any memory connected to the host processor only.
- Global and constant memories are any memory that is connected to the FPGA. These are usually memory chips that are physically connected to the FPGA. The host processor has access to these memory banks through infrastructure in the FPGA base device.

- Local memory is memory inside of the FPGA. This memory is typically implemented using block RAM elements in the FPGA fabric.
- Private memory is memory inside of the FPGA. This memory is typically implemented using registers in the FPGA fabric in order to minimize latency to the compute data path in the processing element.
- You can share data between Linux devices using the Xilinx OpenCL API Extensions, `xclGetMemObjectFd` and `xclGetMemObjectFromFd`.

```
static cl_int xclGetMemObjectFd(cl_mem mem, int* fd) /* returned fd */
```

  - API `clGetMemObjectFd` returns a file pointer to which other OS processes can write. This pointer is used to transfer input data to a `cl_mem` object using `clGetMemObjectFromFd`.

```
static cl_int xclGetMemObjectFromFd(cl_context context, /
```

```
cl_device_id device, cl_mem_flags flags, int fd, cl_mem* mem) /*
returned cl_mem */
```

  - API `xclGetMemObjectFromFd` returns a cl_mem object which is used to write data to a device associated the file pointer. The following example shows how this is used:

```
int main(int argc, char** argv)
{int xilinx_bo_fd[NumOfBuffers];cl_mem buffer[NumOfBuffers];
...
// Create buffers on Device
for(size_t i = 0; i < NumOfBuffers; i++) {
 buffer[i] = clCreateBuffer(world.context,
 //CL_MEM_READ_WRITE, // | CL_MEM_EXT_PTR_XILINX,
CL_MEM_READ_WRITE,
BufferSize,
NULL,
 &err);
  }
// Transfrers buffers to device
err = clEnqueueMigrateMemObjects(world.command_queue,
NumOfBuffers,
buffer,
CL_MIGRATE_MEM_OBJECT_CONTENT_UNDEFINED,
0,
NULL,
NULL);
clFinish(world.command_queue);
...
 // Associate device buffer to file descriptor
for(size_t i = 0; i < NumOfBuffers; i++) {
err = xclGetMemObjectFd(buffer[i], &xilinx_bo_fd[i]);
if(err != CL_SUCCESS) {
printf("Error: xclGetMemObjectFd failed. Buffer #: %i\n", i);
return -1;
 }
    }
printf("Good: Got FD of all buffers\n");
...
```

```
// Transfer data from cl_mem object to file descriptor
for(size_t i = 0; i < NumOfBuffers; i++) {
err = xclGetMemObjectFromFd(world.context, world.device_id,
0,xilinx_bo_fd[i], &imported_buffer[i]);
if(err != CL_SUCCESS) {
printf("Error: xclGetMemObjectFromFd failed. Buffer #: %i\n", i);
 return -1;
 }
   }
printf("Good: Got all buffers from FDs\n");
...
... // Run kernel and process data
  }
```

OpenCL Installable Client Driver Loader

SDAccel™ supports the OpenCL™ ICD extension (`cl_khr_icd`). The OpenCL ICD Loader Library allows multiple implementations of OpenCL to co-exist on the same system. Applications may choose a platform from the list of installed platforms and hence dispatch OpenCL API calls to the correct underlying implementation.

Xilinx® does not provide the OpenCL ICD library and the following should be used to install the library on your preferred system:

On Ubuntu ICD the library is packaged with the distribution, install the following packages:

- ocl-icd-libopencl1
- opencl-headers
- ocl-icd-opencl-dev

For RHEL/CentOS 7.X use EPEL 7, install the following packages:

- ocl-icd
- ocl-icd-devel
- opencl-headers

Recommended Libraries

Xilinx recommends that you install the following libraries on your operating system.

- Independent JPEG Group's JPEG runtime library (version 6.2)
- `%sudo apt-get install libjpeg62 libjpeg62-dev`

Xilinx recommends the following packages should be installed on CentOS 7.x

- PNG reference library.
- `%sudo yum install libpng12`
- The Linux Standards Base (LSB) library. The redhat-lsb package provides utilities needed for LSB Compliant Applications.
- `%sudo yum install redhat-lsb`
- The `libtiff3` package, an older version of `libtiff` library for manipulating TIFF (Tagged Image File Format) image format files.
- `%sudo yum install redhat-lsb`

OpenCL Built-In Functions Support in the SDAccel Environment

The OpenCL™ C programming language provides a rich set of built-in functions for scalar and vector operations. Many of these functions are similar to the function names provided in common C libraries but they support scalar and vector argument types. The SDAccel™ development environment is OpenCL 1.0 embedded profile compliant. The following tables show descriptions of built-in functions in OpenCL 1.0 embedded profile and their support status in the SDAccel environment.

Work-Item Functions

Function	Description	Supported
get_global_size	Number of global work items	Yes
get_global_id	Global work item ID value	Yes
get_local_size	Number of local work items	Yes
get_local_id	Local work item ID	Yes
get_num_groups	Number of work groups	Yes
get_group_id	Work group ID	Yes
get_work_dim	Number of dimensions in use	Yes

Math Functions

Function	Description	Supported
acos	Arc Cosine function	Yes
acosh	Inverse Hyperbolic Cosine function	Yes
acospi	acox(x)/PI	Yes
asin	Arc Cosine function	Yes
asinh	Inverse Hyperbolic Cosine function	Yes
asinpi	Computes acos (x) / pi	Yes
atan	Arc Tangent function	Yes
atan2(y, x)	Arc Tangent of y / x	Yes
atanh	Hyperbolic Arc Tangent function	Yes
atanpi	Computes atan (x) / pi	Yes

Function	Description	Supported
atan2pi	Computes atan2 (y, x) / pi	Yes
cbrt	Compute cube-root	Yes
ceil	Round to integral value using the round to +ve infinity rounding mode.	Yes
copysign(x, y)	Returns x with its sign changed to match the sign of y.	Yes
cos	Cosine function	Yes
cosh	Hyperbolic Cosine function	Yes
cospi	Computes cos (x * pi)	Yes
erf	The error function encountered in integrating the normal distribution	Yes
erfc	Complementary Error function	Yes
exp	base- e exponential of x	Yes
exp2	Exponential base 2 function	Yes
exp10	Exponential base 10 function	Yes
expm1	exp(x) - 1.0	Yes
fabs	Absolute value of a floating-point number	Yes
fdim(x, y)	x - y if x > y, +0 if x is less than or equal to y.	Yes
floor	Round to integral value using the round to -ve infinityrounding mode.	Yes
fma(a, b, c)	Returns the correctly rounded floating-point representation of the sum of c with the infinitely precise product of a and b. Rounding of intermediate products shall not occur. Edge case behavior is per the IEEE 754-2008 standard.	Yes
fmax(x, y) fmax(x, float y)	Returns y if x is less than y, otherwise it returns x. If one argument is a NaN, fmax() returns the other argument. If both arguments are NaNs, fmax() returns a NaN	Yes
fmin(x, y) fmin(x, float y)	Returns y if y less than x, otherwise it returns x. If one argument is a NaN, fmax() returns the other argument. If both arguments are NaNs, fmax() returns a NaN.	Yes
fmod	Modulus. Returns x - y * trunc (x/y)	Yes
fract	Returns fmin(x - floor(x), 0x1.fffffep-1f). floor(x) is returned in iptr	Yes
frexp	Extract mantissa and exponent from x. For each component the mantissa returned is a float with magnitude in the interval [1/2, 1) or 0. Each component of x equals mantissa returned * 2exp.	Yes
hypot	Computes the value of the square root of x2 + y2 without undue overflow or underflow.	Yes
ilogb	Returns the exponent as an integer value.	Yes
ldexp	Multiply x by 2 to the power n.	Yes
lgamma	Returns the natural logarithm of the absolute value of the gamma function. The sign of the gamma function is returned in the signp argument of lgamma_r.	Yes
lgamma_r	Returns the natural logarithm of the absolute value of the gamma function. The sign of the gamma function is returned in the signp argument of lgamma_r.	Yes

Function	Description	Supported
log	Computes natural logarithm.	Yes
log2	Computes a base 2 logarithm	Yes
log10	Computes a base 10 logarithm	Yes
log1p	loge(1.0+x)	Yes
logb	Computes the exponent of x, which is the integral part of logr |x|.	Yes
mad	Approximates a * b + c. Whether or how the product of a * b is rounded and how supernormal or subnormal intermediate products are handled is not defined. mad is intended to be used where speed is preferred over accuracy30	Yes
modf	Decompose a floating-point number. The modf function breaks the argument x into integral and fractional parts, each of which has the same sign as the argument. It stores the integral part in the object pointed to by iptr.	Yes
nan	Returns a quiet NaN. The nancode may be placed in the significand of the resulting NaN.	Yes
nextafter	Next representable floating-point value following x in the direction of y	Yes
pow	Computes x to the power of y	Yes
pown	Computes x to the power of y, where y is an integer.	Yes
powr	Computes x to the power of y, where x is greater than or equal to 0.	Yes
remainder	Computes the value r such that r = x - n*y, where n is the integer nearest the exact value of x/y. If there are two integers closest to x/y, n shall be the even one. If r is zero, it is given the same sign as x.	Yes
remquo	Floating point remainder and quotient function.	Yes
rint	Round to integral value (using round to nearest even rounding mode) in floating-point format.	Yes
rootn	Compute x to the power 1/y.	Yes
round	Return the integral value nearest to x rounding halfway cases away from zero, regardless of the current rounding direction.	Yes
rsqrt	Inverse Square Root	Yes
sin	Computes the sine	Yes
sincos	Computes sine and cosine of x. The computed sine is the return value and computed cosine is returned in cosval.	Yes
sinh	Computes the hyperbolic sine	Yes
sinpi	Computes sin (pi * x).	Yes
sqrt	Computes square root.	Yes
tan	Computes the tangent.	Yes
tanh	Computes hyperbolic tangent.	Yes
tanpi	Computes tan(pi * x).	Yes
tgamma	Computes the gamma.	Yes
trunc	Round to integral value using the round to zero rounding mode.	Yes

Function	Description	Supported
half_cos	Computes cosine. x must be in the range -216... +216. This function is implemented with a minimum of 10-bits of accuracy	Yes
half_divide	Computes x / y. This function is implemented with a minimum of 10-bits of accuracy	Yes
half_exp	Computes the base- e exponential of x. implemented with a minimum of 10-bits of accuracy	Yes
half_exp2	The base- 2 exponential of x. implemented with a minimum of 10-bits of accuracy	Yes
half_exp10	The base- 10 exponential of x. implemented with a minimum of 10-bits of accuracy	Yes
half_log	Natural logarithm. implemented with a minimum of 10-bits of accuracy	Yes
half_log10	Base 10 logarithm. implemented with a minimum of 10-bits of accuracy	Yes
half_log2	Base 2 logarithm. implemented with a minimum of 10-bits of accuracy	Yes
half_powr	x to the power of y, where x is greater than or equal to 0.	Yes
half_recip	Reciprocal. Implemented with a minimum of 10-bits of accuracy	Yes
half_rsqrt	Inverse Square Root. Implemented with a minimum of 10-bits of accuracy	Yes
half_sin	Computes sine. x must be in the range -2^16... +2^16. implemented with a minimum of 10-bits of accuracy	Yes
half_sqrt	Inverse Square Root. Implemented with a minimum of 10-bits of accuracy	Yes
half_tan	The Tangent. Implemented with a minimum of 10-bits of accuracy	Yes
native_ cos	Computes cosine over an implementation-defined range. The maximum error is implementation-defined.	Yes
native_ divide	Computes x / y over an implementation-defined range. The maximum error is implementation-defined	Yes
native_ exp	Computes the base- e exponential of x over an implementation-defined range. The maximum error is implementation-defined.	Yes
native_ exp2	Computes the base- 2 exponential of x over an implementation-defined range. The maximum error is implementation-defined.	No
native_exp10	Computes the base- 10 exponential of x over an implementation-defined range. The maximum error is implementation-defined.	No
native_ log	Computes natural logarithm over an implementation-defined range. The maximum error is implementation-defined.	Yes
native_ log10	Computes a base 10 logarithm over an implementation-defined range. The maximum error is implementation-defined.	No
native_ log2	Computes a base 2 logarithm over an implementation-defined range.	No
native_ powr	Computes x to the power of y, where x is greater than or equal to 0. The range of x and y are implementation-defined. The maximum error is implementation-defined.	No

Function	Description	Supported
native_ recip	Computes reciprocal over an implementation-defined range. The maximum error is implementation-defined.	No
native_ rsqrt	Computes inverse square root over an implementation-defined range. The maximum error is implementation-defined.	No
native_ sin	Computes sine over an implementation-defined range. The maximum error is implementation-defined.	Yes
native_ sqrt	Computes inverse square root over an implementation-defined range. The maximum error is implementation-defined.	No
native_ tan	Computes tangent over an implementation-defined range. The maximum error is implementation-defined	Yes

Integer Functions

Function	Description	Supported
abs	|x|	Yes
abs-diff	|x-y| without modulo overflow	Yes
add_sat	x+y and saturate result	Yes
hadd	(x+y) >> 1 without modulo overflow	Yes
rhadd	(x+y+1) >> 1. The intermediate sum does not modulo overflow.	Yes
clz	Number of leading 0-bits in x	Yes
mad_hi	mul_hi(a,b)+c	Yes
mad24	(Fast integer function.) Multiply 24-bit integer then add the 32-bit result to 32-bit integer	Yes
mad_sat	a*b+c and saturate the result	Yes
max	The greater of x or y	Yes
min	The lessor of x or y	Yes
mul_hi	High half of the product of x and y	Yes
mul24	(Fast integer function.) Multiply 24-bit integer values a and b	Yes
rotate	result[indx]=v[indx]<<i[indx]	Yes
sub_sat	x - y and saturate the result	Yes
upsample	result[i] = ((gentype)hi[i] << 8|16|32) | lo[i]	Yes

Common Functions

Function	Description	Supported
clamp	Clamp x to range given by min, max	Yes
degrees	radians to degrees	Yes

Function	Description	Supported
max	Maximum of x and y	Yes
min	Minimum of x and y	Yes
mix	Linear blend of x and y	Yes
radians	degrees to radians	Yes
sign	Sign of x	Yes
smoothstep	Step and interpolate	Yes
step	0.0 if x < edge, else 1.0	Yes

Geometric Functions

Function	Description	Supported
clamp	Clamp x to range given by min, max	Yes
degrees	radians to degrees	Yes
cross	Cross product	Yes
dot	Dot product only float, double, half data types	Yes
dstance	Vector distance	Yes
length	Vector length	Yes
normalize	Normal vector length 1	Yes
fast_distance	Vector distance	Yes
fast_length	Vector length	Yes
fast_normalize	Normal vector length 1	Yes

Relational Functions

Function	Description	Supported
isequal	Compare of x == y.	Yes
isnotequal	Compare of x != y.	Yes
isgreater	Compare of x > y.	Yes
isgreaterequal	Compare of x >= y.	Yes
isless	Compare of x < y.	Yes
islessequal	Compare of x <= y.	Yes
islessgreater	Compare of (x < y) || (x > y).	Yes
isfinite	Test for finite value.	Yes
isinf	Test for +ve or -ve infinity.	Yes
isnan	Test for a NaN.	Yes
isnormal	Test for a normal value.	Yes
isordered	Test if arguments are ordered.	Yes
isunordered	Test if arguments are unordered.	Yes

Function	Description	Supported
signbit	Test for sign bit.	Yes
any	1 if MSB in any component of x is set; else 0.	Yes
all	1 if MSB in all components of x is set; else 0.	Yes
bitselect	Each bit of result is corresponding bit of a if corresponding bit of c is 0.	Yes
select	For each component of a vector type, result[i] = if MSB of c[i] is set ? b[i] : a[i] For scalar type, result = c ? b : a.	Yes

Vector Data Load and Store Functions

Function	Description	Supported
vloadn	Read vectors from a pointer to memory.	Yes
vstoren	Write a vector to a pointer to memory.	Yes
vload_half	Read a half float from a pointer to memory.	Yes
vload_halfn	Read a half float vector from a pointer to memory.	Yes
vstore_half	Convert float to half and write to a pointer to memory.	Yes
vstore_halfn	Convert float vector to half vector and write to a pointer to memory.	Yes
vloada_halfn	Read half float vector from a pointer to memory.	Yes
vstorea_halfn	Convert float vector to half vector and write to a pointer to memory.	Yes

Synchronization Functions

Function	Description	Supported
barrier	All work-items in a work-group executing the kernel on a processor must execute this function before any are allowed to continue execution beyond the barrier.	Yes

Explicit Memory Fence Functions

Function	Description	Supported
mem_fence	Orders loads and stores of a work-item executing a kernel	Yes

Function	Description	Supported
read_mem_fence	Read memory barrier that orders only loads	Yes
write_mem_fence	Write memory barrier that orders only stores	No

Async Copies from Global to Local Memory, Local to Global Memory Functions

Function	Description	Supported
async_work_group_copy	Must be encountered by all work-items in a workgroup executing the kernel with the same argument values; otherwise the results are undefined.	Yes
wait_group_events	Wait for events that identify the async_work_group_copy operations to complete.	Yes
prefetch	Prefetch bytes into the global cache.	No

PIPE Functions

IMPORTANT!: *OpenCL pipes must be declared in all lowercase; for example:*

```
pipe int infifo_((xcl_reqd_pipe_depth(16)));  //Cannot be 'pipe int inFifo'
```

```
pipe int outfifo_attribute_((xcl_req_pipe_depth(16))); //Cannot be 'pipe
in outFifo
```

Function	Description	Supported
read_pipe	Read packet from pipe	Yes
write_pipe	Write packet to pipe	Yes
reserve_read_pipe	Reserve entries for reading from pipe	No
reserve_write_pipe	Reserve entries for writing to pipe	No
commit_read_pipe	Indicates that all reads associated with a reservation are completed	No
commit_write_pipe	Indicates that all writes associated with a reservation are completed	No
is_valid_reserve_id	Test for a valid reservation ID	No
work_group_reserve_read_pipe	Reserve entries for reading from pipe	No
work_group_reserve_write_pipe	Reserve entries for writing to pipe	No
work_group_commit_read_pipe	Indicates that all reads associated with a reservation are completed	No
work_group_commit_write_pipe	Indicates that all writes associated with a reservation are completed	No

Function	Description	Supported
get_pipe_num_packets	Returns the number of available entries in the pipe	Yes
get_pipe_max_packets	Returns the maximum number of packets specified when pipe was created	Yes

Pipe Functions enabled by the cl_khr_subgroups extension

Function	Description	Supported
sub_group_reserve_read_pipe	Reserve entries for reading from a pipe	No
sub_group_reserve_write_pipe	Reserve entries for writing to a pipe	No
sub_group_commit_read_pipe	Indicates that all reads associated with a reservation are completed	No
sub_group_commit_write_pipe	Indicates that all writes associated with a reservation are completed	No

OpenCL 2.0 Image Objects

Table 6: **OpenCL 2.0 Image Options**

Function	Description	Supported
clCreateImage	Create an image object for a 1D image, 1D image buffer, 1D image array, 2D image, 2D image array or 3D image.	Yes
clGetSupportedImageFormats	Get the list of image formats supported by an OpenCL implementation when the Context, Image type (1D, 2D, or 3D image, 1D image buffer, 1D or 2D image array) and Image object allocation information of the image memory object is specified.	Yes
clEnqueueReadImage	Enqueue commands to read from an image or image array object to host memory.	Yes
clEnqueueWriteImage	Enqueue commands to write to an image or image array object from host memory.	Yes
clEnqueueFillImage	Enqueues a command to fill an image object with a specified color.	No
clEnqueueCopyImageToBuffer	Enqueues a command to copy an image object to a buffer object.	No
clEnqueueMapImage	Enqueues a command to map a region in an image object into the host address space and returns a pointer to this mapped region.	No
clGetImageInfo	Obtain information specific to an image object created with clCreateImage. To get information that is common to all memory objects, use the clGetMemObjectInfo function.	Yes

Appendix E

Creating RTL Kernels

You can implement a kernel in RTL and develop it using theVivado® IDE tool suite. RTL kernels offer potentially higher performance with lower area and power, but require development using RTL coding, tools, and verification methodologies.

Existing RTL-based IP and algorithms can be wrapped and migrated to the SDAccel™SDAccel Environment framework enabling those HDL-based algorithms to tool flow and runtime library. The following section describes how to implement RTL kernels.

RTL kernels should be written, designed, and tested using the recommendations in the *UltraFast Design Methodology Guide for the Vivado Design Suite*, (UG949).

There are three steps to packaging an RTL block as an RTL kernel forSDAccel applications:

1. Package the RTL block as Vivado IP.
2. Create a kernel description XML file.
3. Package the RTL kernel into a Xilinx Object (XO) file.

These steps are automated through the use of the RTL Kernel Wizard.

A fully packaged RTL Kernel is delivered as an XO file with a file extension of `.xo`. This file is a container encapsulating the Vivado IP object (including source files) and kernel XML file. The XO file can be compiled into the platform and run in hardware, or hardware emulation flows.

Programming Paradigm

There are two requirements for using an RTL design as an RTL kernel:

- Execution requirements
- Interface requirements

Execution Requirements

RTL kernels are modeled in software as functions with a void return value similar to the software interface model used in OpenCL™ and C/C++ kernels. This means that RTL kernels can only be passed scalars for input arguments and memory pointer addresses for data to be exchanged with the host application.

In the host application, the RTL kernel is invoked in a similar manner as HLS kernels with a function signature such as:

```
void mmult(int *a, int *b, int *output)
```

```
void mmult(unsigned int length, int *a, int *b, int *output)
```

This implies that the RTL design must have an execution model similar to that of a software function or kernel: start, execute, and end.

- It must be capable of starting when called to do so.
- It must compute all data values.
- It must return the data and then end the operation.

If the RTL design has a different execution model then logic must be added to ensure that the design can be executed in this manner. The RTL Kernel Wizard provides a flow that allows such changes to be performed.

Interface Requirements for Integration Into the Platform

The RTL kernel must adhere to the C function interface shown in the previous section. The RTL kernel must also integrate into a platform, which has its own requirements. The RTL kernel integrates into a platform using a slave AXI4-Lite interface for control register access (to pass kernel arguments and to start/stop the kernel). The RTL kernel can also have AXI4 master interfaces to talk to memory.

The interface requirements are discussed below in detail. In summary they are as follows:

- All scalar arguments must be passed to the RTL kernel via an AXI4 MM Slave Lite interface.
- All data accesses must be performed through one or more AXI4 MM interfaces.
- Data transfers between kernels may be performed using an AXI Stream interface.

The following signals and interfaces are required on the top level of an RTL block.

- Primary clock input port named `ap_clk`.

    - There can be a secondary optional clock input named `ap_clk_2`.
- Primary active low reset input port named `ap_rst_n`.
    - This signal should be internally pipelined to improve timing.
    - This signal is driven by a synchronous reset in the `ap_clk` clock domain.
- There can be a secondary optional active low reset input `ap_rst_n_2`.
    - This signal should be internally pipelined to improve timing.
    - This signal is driven by a synchronous reset in the `ap_clk_2` clock domain.
- One and only one AXI4-Lite slave control interface.
    - Offset 0 of the AXI4-Lite slave interface must have the following signals:
        - Bit 0: start signal - The kernel starts processing data when this bit is set.
        - Bit 1: done signal - The kernel asserts this signal when the processing is done. This bit is clear on read.
        - Bit 2: idle signal - The kernel asserts this signal when it is not processing any data. The transition from low to high should occur synchronously with assertion of done signal.
    - The host typically writes to 0x00000001 to the offset 0 control register which sets Bit 0, clears Bits 1 and 2, and polls on reading done signal until it is a "1".
- One or more AXI4 memory mapped (MM) master interfaces for global memory.
    - All AXI4 MM master interfaces must have 64-bit addresses.
    - The 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.

RTL Kernel Wizard

The RTL Kernel wizard automates some of the steps that need to be taken to ensure that the RTL IP is packaged into a Kernel that can be integrated into a system in SDAccel™.

Based on the interface information provided to the Kernel wizard, it generates an RTL wrapper for the kernel that meets the RTL kernel interface requirements.

The Wizard automatically generates the AXI4-Lite interface module including the control logic and register file. The AXI4-Lite interface module is included in the generated top-level RTL Kernel wrapper. The wrapper also includes an example kernel IP module that you replace with your RTL IP design. You need ensure correct connectivity between RTL IP with wrapper template.

The Wizard also generates a Vivado project for the top-level RTL kernel wrapper and the generated files. This enables working in the Vivado environment to update and optimize the RTL kernel.

The generated files includes a simple test bench for the generated RTL kernel wrapper and a sample host code to exercise the example RTL kernel, and that example test bench and host code must be modified to test the actual RTL IP design.

The benefit of the wizard is that allowable selections serve to guide you to understand the supported features of an RTL kernel, such as the following:

- Automating some of the steps that must be taken to ensure that the RTL IP is packaged into a Kernel that can be integrated into a system in SDAccel™.
- Steps you through the process of specifying your software function model and interface model for the RTL Kernel.
- Generates an RTL wrapper for the kernel that meets the RTL kernel interface requirements, based on the interface information provided.
- Automatically generates the AXI4-Lite interface module including the control logic and register file. The AXI-lite interface module is included in the generated top level RTL Kernel wrapper.
- Includes in the wrapper an example kernel IP module that you need to replace with your RTL IP design. The RTL IP developer must ensure correct connectivity between RTL IP with wrapper template.

A `kernel.xml` file is also generated to match the software function prototype and behavior specified in the wizard. The benefit of the wizard is that allowable selections serve to guide you to understand the supported features of an RTL kernel.

Launching the RTL Kernel Wizard

The RTL Kernel Wizard can be launched with two different methods: from the SDx™ Development Environment or from the Vivado® IDE. The SDx Development Environment provides a more seemless experience by automatically importing the generated kernel/example host code back into the SDx project.

To launch the RTL Kernel Wizard from the SDx Development Environment, do the following:

1. Launch the SDx Development Environment.

2. Create an SDx Project (Application Project Type).
3. Click **Xilinx** > **RTL Kernel Wizard**.

To launch the RTL Kernel Wizard from Vivado IDE, do the following:

1. Create a new Vivado project choosing the same device as exists on the platform you intend to target. If you do not know your target device, choose the default part.
2. Go to the IP catalog by clicking the **IP Catalog** button.
3. Type `wizard` in the IP Catalog search box.
4. Double-click **SDx Kernel Wizard** to launch the wizard.

***Note*:** Use Vivado from the SDx install so the tool versions are the same.

RTL Kernel Wizard

The wizard is organized into pages that break down the process of creating a kernel into smaller steps. To navigate between pages, use the **Next** and **Back** buttons. To finalize the kernel and build a project based on the wizards inputs, click the **OK** button. Each of the following sections describes each page and its input options.

Welcome to SDx Kernel Wizard

This page gives an abbreviated version of the workflow and the steps for following the wizard.

RTL Kernel Wizard General Settings

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 in the Vendow/Library/Name/Version (VLNV) format described in

  Vivado Design Suite User Guide: Designing with IP (UG896).
- **Kernel library:** The name of the library. Used in the VLNV. Must conform to same identifier rules **Kernel name**.

Kernel Options

Kernel type - Supports RTL and Block Design kernel types. A RTL kernel type consists of a Verilog RTL top level module with a Verilog control register module and a Verilog kernel example inside the top.Block Design kernel type also delivers a Verilog RTL top level module, but instead it instantiates an IP integrator Block Diagram inside of a verilog RTL top level module. The block design consists of MicroBlaze™ subsystem that uses a BRAM exchange memory to emulate the control registers. Example MicroBlaze software is delivered with the project to demonstrate using the MicroBlaze to control the kernel.

Clocking Options

Number of clocks - Sets the number of clocks used by the kernel. Every kernel has a primary clock and reset called `ap_clk` and `ap_rst`. All AXI interfaces on the kernel will be driven with this clock and reset. When **Number of clocks** is set to **2**, a secondary clock and related 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`, respectively. 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/slower rate than the AXI4 interfaces, which must be clocked on the primary clock. When designing with multiple clocks, proper clock domain crossing techniques must be used to ensure data integrity across all clock frequency scenarios.

Scalars Arguments

Scalar arguments are used to pass control type of information to the kernels. Scalar arguments can not be read back from the host. For each argument that is specified a corresponding control register is created to facilitate passing the argument from software to hardware.

- **Number of scalar kernel input arguments** - Specifies the number of scalar input arguments to pass to the kernel. For each of the 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.

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 the data type of the argument. This will affect the width of the control register in the generated Verilog module. The data types available are limited to the ones specified by the OpenCL version 2.0 specification. Data types that represent a bit width greater than 32 bits will 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. 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. See the Memory Performance Optimizations for AXI4 Interface section for recommendations on how to design these interfaces for optimal performance. For each interface, example AXI master logic is generated in the RTL kernel to provide a starting point and can be discarded if not used.

- **Number of AXI master interfaces:**Specifies the number of AXI master interfaces present on the Kernel. You can specify a maximum of 16 interfaces. For each interface, you can customize an interface name, data width, and number of associated arguments. Each interface contains all read and write channels.

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 (bytes): Specifies the data width of the AXI data channels. It is recommended this is matched 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.The data type of the pointer is generic and can be anything as long as it is allocated correctly in the host code.

Argument Definition

Interface: Specifies the name of the AXI Interface that the corresponding columns in the current row are associated to. This value is not directly modifiable, it is copied from the interface name defined in the previous table.

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.

Summary

This page gives a summary of the VLNV, software function prototype and hardware control registers created from options selected in the previous pages. The function prototype conveys what a kernel call would like if it was a C function.

***Note*:** The function prototype presented here is only an example. Refer to the host code in the generated example for exact details on how to set the kernel arguments for the kernel call.

The register map shows the relationship between host software ID, argument name, hardware register offset, type, and associated interface. Review this page to for correctness before proceeding to generating the kernel.

Finalizing and Generating the Kernel from the RTL Wizard

If the RTL Kernel Wizard was launched from SDx™, after clicking **OK**, the example Vivado® project opens.

If the RTL Kernel Wizard was launched from Vivado, after clicking **OK** do the following:

1. When the **Generate Output Products** window appears, select **Global** synthesis options and click **Generate**, then **OK**.
2. Right-click on the `.xci` file in the **Design Sources View**, and select **Open IP Example Design**.
3. In the open example design window, select an output directory (or accept default) and click **OK**.
4. This opens a new Vivado® project with the example design in it. You can now close the current Vivado project that the RTL Kernel Wizard was invoked from.

RTL Kernel Wizard Vivado Project

The RTL Kernel Wizard configuration dialog box customizes the specification of an RTL Kernel by specifying its I/O, control registers, and AXI4 interfaces. The next step in the process is to customize the contents of the kernel and then package those contents into a Xilinx® Object (`xo`) file. After the RTL Kernel Wizard configuration GUI has completed, a Vivado kernel project is generated and populated with the files necessary to create an RTL Kernel.

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 when combined with the rest of the RTL/Block Design becomes the acceleration kernel. The AXI4 interfaces defined at the top-level file contain a minimum subset of AXI4 signals required to generate an efficient, high throughput interface. Signals omitted will 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 starting with existing code that contains AXI signals not listed in the port list, it is possible to add these signals to the top-level ports and the IP packager will adapt to them appropriately.

Depending on the **Kernel Type** selected, the contents of the top-level file is populated either with an Verilog example and control registers, or an instantiated IP integrator block design.

RTL Kernel Type Project Flow

The RTL kernel type delivers a control register module and an example portion. Care should be taken if the control register module is modified to ensure it still aligns with the `kernel.xml` file located in the imports directory of the Vivado kernel project. The example portion may be discarded or used as a starting port for custom logic. The example consists of a simple adder function, and 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 16KB of data, performs a 32 bit "add one" operation, and then writes out 16KB of data back in place (writes to same starting pointer as the read address.)

Block Design Kernel Type Project Flow

The block design kernel type delivers an IP integrator block design (BD) as the basis of the kernel. 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 DataMover IP to create an example identical to the one in the RTL kernel type. Also, included is an SDK 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 may 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 appears, click OK.
2. The SDK application can now be invoked. Select the **File->Launch SDK** option from the Vivado menu.
3. When the Xilinx SDK GUI appears, Click **X** just to the right of the text on the Welcome tab to close the welcome dialog box. This shows an already loaded SDK project underneath.
4. From the project explorer, the source files can be found under the `<Kernel Name>_control/src` section. Modify these as appropriate.
5. When updates are complete, compile the source by selected the menu option **Project->Build All Check for errors/warnings and resolve if necessary**.
6. The ELF file is automatically updated in the GUI.
7. Run simulation to test the updated program and debug if necessary.

Simulation Testbench

A SystemVerilog simulation test bench is generated, this will exercise the kernel to ensure its operation is correct. It is populated with the checker function to verify the “add one” operation. This generated testbench can be used as a starting point in verifying the kernel functionality. It will write/read from the control registers and execute 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 the Vivado **Flow Navigator->Run Simulation**option located on the left hand side of the GUI and selected the **Run Behavioral Simulation** option. If behavioral simulation is working as expected, a post-synthesis functional simulation can be run to ensure that synthesis is 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. An XDC file is populated in the design to provide default clock frequencies for this purpose. Running synthesis is useful to determine whether the kernel will synthesize without errors. It will also provide estimates of utilization and frequency. The kernel should be able to run through synthesis successfully before it is packaged. Otherwise, errors occur during linking and it could be harder to debug. The synthesized outputs can be used when packaging the Kernel as a netlist instead of RTL. If a block design is used within the kernel, then the kernel must be packaged as netlist. To run out of context synthesis, click the **Run Synthesis** option from the Vivado **Flow Navigator->Synthesis** menu.

Software Model and Host Code Example

A C++ software model of the example “add one” operation is provided in the imports directory. It has the same name as the kernel and will have a `cpp` file extension. This software model can be modified to model the function of the kernel. In the packaging step, this model can be included with the kernel. When using SDx, this allows software emulation to be performed with the kernel. The Hardware Emulation and the System Linker always use the hardware description of the kernel.

In the `sdx_imports` directory, example C host code is provided and is called `main.c`. The host code will expect the binary container as the argument to the program. This can be automatically specified by checking the **Automatically add binary container(s) to arguments** checkbox in the **Run Configuration->Arguments** option after the host code is loaded into the SDx GUI. The host code then loads the binary as part of the `init` function. The host code will instantiate the kernel, allocate the buffers, set the kernel arguments, execute the kernel, then collect and check the results for the example “add one” function.

Generate RTL Kernel

After the kernel is designed and tested in Vivado, the final step for generating the RTL Kernel is to package the Vivado kernel project for use with SDx.

To begin the process, click the **Generate RTL Kernel** option from the Vivado **Flow Navigator->Project Manager** menu. A pop-up dialog box opens with three main packaging options:

- A Source-only kernel packages the Kernel using the RTL design sources directly.
- The 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.
- The 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 retarget the new device and it must be regenerated from the source. If the design contains a block design, then the Netlist (DCP) based kernel is the only packaging option available.

Optionally, all kernel packaging types can be packaged with the software model that can be used in software emulation. If the software model contains multiple files, then 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 the **OK** button, 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 will use those outputs, regardless if they are stale. The kernel `xo` file is generated in the `sdx_imports` 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 SDx GUI, then the example host code called `main.c` and kernel XO files are automatically imported into the SDx source folder.

Modifying an Existing RTL Kernel Generated from the Wizard

From the SDx GUI, it is possible to modify an existing generated kernel. By invoking the the Xilinx RTL Kernel Wizard menu option after a kernel has been generated, a dialog box opens that gives you the option to modify an existing kernel. Choosing the **Edit Existing Kernel Contents** option re-opens the Vivado Project, and you can then modify and generate the kernel contents again. Choosing the **Re-customize Existing Kernel Interfaces** option will re-visit the RTL Kernel Wizard configuration dialog box. Options other than **Kernel Name** can be modified and the previous Vivado Project is replaced.

***Note*:** Important: all files and changes in the previous project will be lost when the updated Vivado kernel project is generated.

Manual Development Flow for RTL Kernels

Using the RTL Kernel Wizard to create RTL kernels is highly recommended, however RTL Kernels may be created without using the wizard. This section provides details on each step of the manual development flow. The three steps to package an RTL design as an RTL kernel for SDAccel™ applications are:

1. Package the RTL block as Vivado® IP.
2. Create a kernel description XML file.
3. Package the RTL kernel into a Xilinx Object (XO) file.

Packaging an RTL Block as Vivado IP

RTL Kernels must be packaged as a Vivado® IP suitable for use in IP integrator. See *Vivado Design Suite User Guide: Creating and Packaging Custom IP* (UG1118) for details on IP packaging in Vivado.

The following interface packaging is required for the RTL Kernel:

- The AXI4-Lite interface name must be packaged as `S_AXI_CONTROL`, but the underlying AXI ports can be named differently.
- The AXI4 MM interfaces must be packaged as AXI4 master endpoints with 64 bit address support.
  - Xilinx strongly recommends that AXI4 MM 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.
- `ap_clk` and `ap_clk_2` must be packaged as clock interfaces.
- `ap_rst_n` and `ap_rst_n_2` must be packaged as active low reset interfaces.
- `ap_clk` must be packaged to be associated with all AXI4-Lite, AXI4 MM, and AXI4-Stream interfaces.

To test if the RTL kernel is packaged correctly for IP integrator, try to instantiate the packaged kernel in IP integrator. In the GUI it should show up as having interfaces for clock, reset, AXI4-Lite slave, AXI4 MM master, and AXI4-Slave only. No other ports should be present 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 then the unused AXI channels can be removed and the `READ_WRITE_MODE` will be set to read-only or write-only.

Create Kernel Description XML File

A kernel description XML file needs to be manually created for the RTL IP to be used as an RTL kernel in SDAccel environment. The following is an example of the kernel XML file:

```
<?xml version="1.0" encoding="UTF-8"?>
<root versionMajor="1" versionMinor="0">
<kernel name="input_stage" language="ip"
vlnv="xilinx.com:hls:input_stage:1.0" attributes=""
preferredWorkGroupSizeMultiple="0" workGroupSize="1">
<ports>
<port name="M_AXI_GMEM" mode="master" range="0x3FFFFFFF" dataWidth="32"
portType="addressable" base="0x0"/>
<port name="S_AXI_CONTROL" mode="slave" range="0x1000" dataWidth="32"
portType="addressable" base="0x0"/>
<port name="AXIS_P0" mode="write_only" dataWidth="32" portType="stream"/>
</ports>
<args>
<arg name="input" addressQualifier="1" id="0" port="M_AXI_GMEM"
size="0x4" offset="0x10" hostOffset="0x0" hostSize="0x4" type="int*" />
<arg name="__xcl_gv_p0" addressQualifier="4" id="" port="AXIS_P0"
size="0x4" offset="0x18" hostOffset="0x0" hostSize="0x4" type=""
memSize="0x800"/>
</args>
</kernel>
<pipe name="xcl_pipe_p0" width="0x4" depth="0x200" linkage="internal"/>
<connection srcInst="input_stage" srcPort="p0" dstInst="xcl_pipe_p0"
dstPort="S_AXIS"/>
</root>
```

The following table describes the format of the kernel XML in detail:

Table 7: **Kernel XML Format**

Tag	Attribute	Description
<root>	versionMajor	Set to 1 for the current release of SDAccel
	versionMinor	Set to 0 for the current release of SDAccel
<kernel>	name	Kernel name
	language	Always set it to "ip" for RTL kernels

	vlnv	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:
		`<spirit:vendor>xilinx.com</spirit:vendor>`
		`<spirit:library>hls</spirit:library>`
		`<spirit:name>test_sincos</spirit:name>`
		`<spirit:version>1.0</spirit:version>`
		the vlnv attribute in kernel XML will need to be set to:
		`xilinx.com:hls:test_sincos:1.0`

	attributes	Reserved. Set it to empty string.
	prefreredWorkGroupSizeMultiple	Reserved. Set it to 0.
	workGroupSize	Reserved. Set it to 1.

Table 7: **Kernel XML Format** *(cont'd)*

Tag	Attribute	Description
<port>	name	Port name. At least an AXI4 master port and an AXI4-Lite slave port are required. AXI stream port can be optionally specified to stream data between kernels. The AXI4-Lite interface name must be S_AXI_CONTROL.
	mode	• For AXI-4master port, set it to "master". • For AXI-4 slave port, set it to "slave". • For AXI Stream master port, set it to "write_only". • For AXI Stream slave port, set it "read_only".
	range	The range of the address space for the port.
	dataWidth	The width of the data that goes through the port, default is 32 bits.
	portType	Indicate whether or not the port is addressable or streaming. • For AXI4 master and slave ports, set it to "addressable". • For AXI4-Stream ports, set it to "stream".
	base	For AXI4 master and slave ports, set to "0x0". This tag is not applicable to AXI4-Stream ports.
<arg>	name	Kernel argument name.
	addressQualifier	Valid values: 0: Scalar kernel input argument 1: global memory 2: local memory 3: constant memory 4: pipe
	id	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.
	port	Indicates the port that the arg is connected to.
	size	Size of the argument. The default is 4 bytes.
	offset	Indicates the register memory address.
	type	The C data type for the argument. E.g. int*, float*
	hostOffset	Reserved. Set to 0x0.
	hostSize	Size of the argument. The default is 4 bytes.
	memSize	Not applicable to AXI-4 master and slave ports. For AXI4-Stream ports, memSize sets the depth of the FIFO created for the AXI stream ports.
The following tags specify additional information for AXI4-Stream ports. They are not applicable to AXI4 master or slave ports.		

Table 7: **Kernel XML Format** *(cont'd)*

Tag	Attribute	Description
<pipe>	For each pipe in the compute unit, the compiler inserts a FIFO for buffering the data. The pipe tag describes configuration of the FIFO.	
	name	This specifies the name for the FIFO inserted for the AXI4-Stream port. This name must be unique among all pipes used in the same compute unit.
	width	This specifies the width of FIFO in bytes. For example, 0x4 for 32-bit FIFO.
	depth	This specifies the depth of the FIFO in number of words.
	linkage	Always set to "internal".
<connection>	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.	
	srcInst	Specifies the source instance of the connection.
	srcPort	Specifies the port on the source instance for the connection.
	dstInst	Specifies the destination instance of the connection.
	dstPort	Specifies the port on the destination instance of the connection.

Package RTL Kernel into Xilinx Object File

The final step is to package the RTL IP and the kernel XML together into a Xilinx object file (`.xo`) so it can be used by the SDAccel compiler. The following is the command example in Vivado 2017.4_sda. The final RTL kernel is in the `test.xo` file.

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

Designing RTL Recommendations

Memory Performance Optimizations for AXI4 Interface

The AXI4 MM interfaces typically connect to DDR memory controllers in the platform. 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:

1. Use an AXI data width that matches the native memory controller AXI data width, typically 512 bits.
2. Do not use WRAP, FIXED, or sub-sized bursts.

3. Use burst transfer as large as possible (up to 4KByte AXI4 protocol limit).
4. Avoid use of de-asserted write strobes. De-asserted write strobes can cause ECC logic in the DDR memory controller to perform read-modify-write operations.
5. Use pipelined AXI transactions
6. Avoid using threads if an AXI interface is only connected to one DDR controller.
7. Avoid generating write address commands if the kernel does not have the ability to deliver the full write transaction (non-blocking write requests).
8. Avoid generating read address commands if the kernel does not have the capacity to accept all the read data without backpressure (non-blocking read requests).
9. If a read-only or write-only interfaces are desired, then the ports of the unused channels can be commented out in the top level RTL file before the project is packaged into a kernel.
10. Using multiple threads can cause larger resource requirements in the infrastructure IP between the kernel and the memory controllers.

Quality of Results Considerations

The following recommendations help improve results for timing and area:

1. Pipeline all reset inputs and internally distribute resets avoiding high fanout nets.
2. Reset only essential control logic FFs.
3. Consider registering input and output signals to the extent possible.
4. Understand the size of the kernel relative to the capacity of the target platforms to ensure fit, especially if multiple kernels will be instantiated.
5. Recognize platforms that use Stack Silicon Interconnect (SSI) Technology. These devices have multiple dice and any logic that must cross between them should be Flip Flop (FF) to FF timing paths.

Debug and Verification Considerations

1. 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 (AXI VIP) is available in the Vivado® IP catalog and can help with verification of AXI interfaces. The RTL kernel example designs contain an AXI VIP based test bench with sample stimulus files.
2. The hardware emulation flow should not be used for functional verification because it does not accurately represent the range of possible protocol signalling conditions that real AXI traffic in hardware may incur. Hardware emulation should be used to test the host code software integration or to view the interaction between multiple kernels.

Appendix F

xbinst Command Reference

The FPGA acceleration card plugged into the host machine must have the associated Linux kernel driver, firmware, and runtime libraries installed before it can be used for running user applications. SDAccel™ provides a Xilinx® board installation utility, `xbinst` to generate all necessary files for the platform support package for the FPGA card. It also generates an installation script to compile and install the driver, firmware, and runtime libraries.

The `xbinst` utility requires superuser privileges on the host machine to run. The supported options are listed below:

Table 8: **xbinst Options**

Short Option	Long Option	Valid Values	Description
`-h`	`--help`	NA	Print Usage Message.
`-f <arg>`	`--platform <arg>`	Supported platform from SDAccel installation or the full path to platform definition file for custom platforms	Required. All installed platforms are listed at SDAccel Developer Zone: Platforms.
`-d <arg>`	`--destination <arg>`	Valid path on the file system.	Required destination directory for driver and firmware for the specified platform.

Follow the instructions below to install the driver and firmware for the FPGA card on the host machine. The ADM-PCIE-7V3 DSA is used as an example. Replace it with the DSA for the actual card plugged into the system.

All commands need to be run with superuser privileges.

1. Create a board installation directory.

```
sudo mkdir -p /opt/dsa/xilinx_kcu1500_dynamic_5_0
```

2. Run `xbinst` to generate all necessary files.

```
$ sudo xbinst --platform xilinx_kcu1500_dynamic_5_0
-d /opt/dsa/<file_path>/xilinx_kcu1500_dynamic_5_0
****** xbinst v2017.4_sdx
  **** Copyright 1986-2017 Xilinx, Inc. All Rights Reserved.

INFO: [XBINST 60-267] Packaging for ..
INFO: [XBINST 60-268] Packaging for ..Complete
INFO: [XBINST 60-667] xbinst has successfully created a board
installation directory
at <file_path>/*_dsa
```

   If you installed a custom platform, the full path to the platform package file needs to be provided to the xbinst command as shown in the following example:

```
$ sudo xbinst -f /platform/repo/vendor_board_name_version.xpfm -d
custom_platform
```

3. Install the driver, firmware, and runtime libraries.

```
$cd <verson>_dsa/xbinst/
$ sudo ./install.sh
```

   This will do the following:

   - Compile and install Linux kernel device drivers.
   - Install the firmware files to the Linux firmware area.
   - Generate a `setup.sh` (Bash) or `setup.csh` (for csh/tcsh) to set up the runtime environment. Users must source the setup script before running any application on the target FPGA card.

Appendix G

Xilinx Board Swiss Army Knife Utility

Xilinx® Board Swiss Army Knife (`xbsak`) utility is a standalone command line utility that can perform the following board administration and debug tasks independent of SDAccel™ runtime library (`.so`) and for the SDAccel tools installation:

- Board administration tasks:
  - Flash device configuration memory of the board.
  - Reboot boards without rebooting the host.
  - Reset hung boards.
  - Query board status, sensors and PCIe® AER registers.
- Debug operations:
  - Download the SDAccel binary. `.xclbin`) to FPGA.
  - Test DMA for PCIe bandwidth.
  - Show status of compute units.

The `xbsak` utility automatically adds itself to your path when you run the `setup.{csh|sh}` file created by `xbinst` when it created the deployent files for the deployment machince.

xbsak Commands and Options

The following documents the `xbsak` command line format, and the available commands and options .

`xbsak` <*command*> [*options*]

Commands and Options

- `help`: Print help messages.
- `list` : List all supported devices installed on the server in the format of [*device_id*] : *device_name*.

The following is an example output where the device ID is 0 and the device name is `xilinx_vcu1525_dynamic_5_0`.

```
$xbsak list
Linux:xxxx
Distribution: yyyy
GLIBC: 2.17
---
XILINX_OPENCL="/opt/dsa/xilinx_vcu1525_dynamic_5_0/xbinst"
LD_LIBRARY_PATH="/opt/dsa/xilinx_vcu1525_dynamic_5_0/xbinst/runtime/lib/
x86_64:"
---
INFO: Found 1 device(s)
[0] xilinx:vcu1525:dynamic:5.0
```

- `query` [`-d` *device_id*] [`-r` *region_id*]

  Query the specified device and programmable region on the device to get detailed status information.

  . `-d` *device_id* - Specify the target device. Optional. Default=0 if not specified.

  . `-r` *region_id* - Specify the target region. Optional. Default=0 if not specified.

- `boot` [`-d` *<device_id>*]: Attempts to boot the device <*device_id*>from its configuration memory, retain the PCIe® link without rebooting the host, and trigger a re-enumeration of the PCIe bus and bus re-scanning. If this option command does not complete as expected, `xbsak` notifies the user and requests for hard reboot (sometimes called cold reboot: shutdown and restart of the machine)

- `clock` [`-d` *<device_id>*] [`-r` *<region_id>*] `-f` *<clock1_freq>* [`-g` *<clock2_freq>*]: Set frequencies of clocks driving the computing units.

  . `-d` *<device_id>* - Specify the target device. Optional. Default=0 if not specified.

  . `-r` *<region_id>* - Specify the target region. Optional. Default=0 if not specified.

  . `-f` *<clock1_freq>* - Specify clock frequency in MHz for the first clock. Required. All platforms have this clock.

  . `-g` *<clock2_freq>* - Specify clock frequency in MHz for the second clock. Optional: Some platforms have this clock to support IP based kernels.

- `dmatest` [`-d` *device_id*] [`-b` *blocksize*]

  Test throughput of data transfer between the host machine and global memory on the device.

  . `-d` *<device_id>* - Specify the target device. Optional. Default=0 if not specified.

  . `-b` *<blocksize>* - Specify the test block size in KB. Optional: Default=65536 or 64MB if not specified. The block size can be specified in both decimal or hexadecimal formats. e.g. Both `-b 1024` and `-b 0x400` set the block size to 1024KB or 1MB.

- `flash [-d device_id] -m primary_mcs [-n secondary_mcs]`

  This option helps program the configuration memory (flash memory device) on the FPGA board with specified configuration files for the FPGA device to boot from. If you know the platform installed currently on the board then you can use this method.

  If you do not know what DSA is installed or want to program from a starting point, you need to externally flash using the USB JTAG or USB cable as detailed in the board installation section for your FPGA board.

  . `-d <device_id>`: Specify the target device. Optional. Default=0 if not specified.

  . `-m <primary_mcs>`: Specify the primary configuration file. Required. All platforms have at least one configuration memory.

  . `-n <secondary_mcs>`: Specify the secondary configuration file. Optional. Some platform have two configuration memories and the secondary configuration file is required for the second configuration memory.

IMPORTANT!:

For the flash programming function of the `xbsak` to access and program the flash configuration memory on the board, certain hardware features must be present in the current platform programmed onto the FPGA. This means the flash programming function works with boards already programmed with a given firmware only, which are listed in the following table.

The following are required steps:

1. *Have two `xbinst`-generated deployment areas.*
   a. *The first area corresponds tot the **current** platform deployed on the board. This provides the necessary `xbsak` that matches the current drivers installed on the deployment machine; most often you already have this area present on the deployment machine. You must source the `setup.{sh,csh}` from this area to use `xbsak` matching the installed drivers.*
   b. *The second area corresponds to the **new** platform to be deployed on the board; this provides the following:*
      i. *Configuration file(s) needed for the primary mcs `(-m)` and optional secondary mcs `(-n)` options*
      ii. *New drivers needed to be deployed on the machine*
2. *Find the new configuration files in the new platform with the command: `$ find -name '*.mcs'` OR `$ find /path/to/new/platform -name '*.mcs'`*
3. *Run `xbsak` to program the configuration file(s) onto the configuration memory of the board with the `-m` and `-n` options.*
4. *Install the drivers of the new platform in the OS (to replace the old drivers) with `$ sudo ./install.sh -k yes -f yes`; the `f` and `k` options will force the install of the OS kernel drivers.*

The table lists the boards and their minimum required platform firmware versions for the flash programming to function. The entries show the platform names to use for the `xbinst` *command.*

Board	Platform Firmware used in xbinst 2017.2	Platform Firmware using in xbinst 2017.4
ADM-PCIE-7V3	`xilinx:adm-pcie-7v3:1ddr:3.0` or newer	
ADM-PCIE-KU3	. `xilinx:adm-pcie-ku3:2ddr-xpr:3.3` or newer . `xilinx:adm-pcie-ku3:2ddr:3.3` or newer . `xilinx:adm-pcie-ku3:1ddr:3.3 or newer`	
KCU1500	-	`xilinx_kcu1500_dynamic_5_0`
VCU1525	-	`xilinx_vcu1525_dynamic_5_0`
XIL-ACCEL-RD-KU115	`xilinx:xil-accel-rd-ku115:4ddr-xpr:3.2` or newer	

- `program` [`-d` *`device_id`*] [`-r` *`region_id`*] `-p` *`xclbin`*

  Download the OpenCL binary to the programmable region on the device.

  . `-d` *`device_id`* - Specify the target device. Optional. Default=0 if not specified.

  . `-r` *`region_id`* - Specify the target region. Optional. Default=0 if not specified.

  . `-p` *`xclbin`* - Specify the OpenCL binary file. Required.

- `reset` [`-d` *`device_id`*] [`-r` *`region_id`*]

  Reset the programmable region on the device. All running compute units in the region will be stopped and reset.

  . `-d` *`device_id`* - Specify the target device. Optional. Default=0 if not specified.

  . `-r` *`region_id`* - Specify the target region. Optional. Default=0 if not specified.

- `status` [`--apm` | `--lapc`]

Displays the status of any AXI Performance Monitor (apm) or Lightweight AXI Protocol Checkers (lapc) that are available in the base platform.

  - `--apm` - Returns the value of the AXI Performance Monitor (apm) counters. This option is only applicable if one or more AXI Performance Monitors are available in the base platform.
  - `--lapc` - Returns the values of the violations codes detected by the Lightweight AXI Protocol Checkers (lapc). This option is only applicable if one or more Lightweight AXI Protocol Checkers are available in the base platform.

- `- scan`

  Scans the system and displays any Xilinx PCIe devices, associated drivers and pertinent system information.

Using the Runtime Initialization File

The SDAccel™ runtime library uses various parameters to control debug, profiling, and message logging during host application and kernel execution in software emulation, hardware emulation, and system run on the acceleration board. These control parameters are specified in a runtime initialization file.

For command line users, the runtime initialization file needs to be created manually. The file must be named `sdaccel.ini` and saved in the same directory as the host executable.

For SDx GUI users, the project manager creates `sdaccel.ini` file automatically based on users run configuration and saves it next to the host executable.

The runtime library will check if `sdaccel.ini` exists in the same directory as the host executable and automatically read the parameters from the file during start-up if it finds it.

Runtime Initialization File Format

The runtime initialization file is a text file with groups of keys and their values. Any line beginning with semicolon (;) or hash (#) is a comment. The group names, keys, and key values are all case sensitive.

The following is a simple example that turns on profile timeline trace and sends the runtime log messages to console.

```
#Start of Debug group
[Debug]
timeline_trace = true

#Start of Runtime group
[Runtime]
runtime_log = console
```

The following table lists all supported groups, keys, valid key values and short descriptions on the function of the keys.

Key	Valid Values	Descriptions
`[Debug] Group`		

Key	Valid Values	Descriptions
`debug`	`[true|false]`	Enable or disable kernel debug. • true: enable • false: disable • Default: false
`profile`	`[true|false]`	Enable or disable OpenCL code profiling. • true: enable • false: disable • Default: true
`timeline_trace`	`[true|false]`	Enable or disable profile timeline trace • true: enable • false: disable • Default: false
`device_profile`	`[true|false]`	Enable or disable device profiling. • true: enable • false: disable • Default: false
`[Runtime] Group`		
`api_checks`	`[true|false]`	Enable or disable OpenCL API checks. • true: enable • false: disable • Default: true
`runtime_log`	`null console syslog filename`	Specify where the runtime logs are printed • null: Do not print any logs. • console: Print logs to `stdout` • syslog: Print logs to Linux syslog • filename: Print logs to the specified file. e.g. `runtime_log=my_run.log` • Default: null

Key	Valid Values	Descriptions
`polling_throttle`	An integer	Specify the time interval in microseconds that the runtime library polls the device status. • Default: 0
`[Emulation] Group`		
`aliveness_message_interval`	Any integer	Specify the interval in seconds that aliveness messages need to be printed • Default 300
`print_infos_in_console`	`[true|false]`	Controls the printing of emulation info messages to users console. Emulation info messages are always logged into a file called emulation_debug.log • true = print in users console • false = don't print in users console • Default: true
`print_warnings_in_console`	`[true|false]`	Controls the printing emulation warning messages to users console. Emulation warning messages are always logged into a file called emulation_debug.log. • true = print in users console • false = do not print in users console • Default: true
`print_errors_in_console`	`[true|false]`	Controls printing emulation error messages in users console. Emulation error messages are always logged into file called emulation_debug.log. • true = print in users console • false = don't print in users console • Default: true
`enable_oob`	`[true|false]`	Enable or disable diagnostics of out of bound access during emulation. A warning is reported if there is any out of bound access. • true: enable • false: disable • Default: false

Key	Valid Values	Descriptions
`launch_waveform`	`[off|batch|gui]`	Specify how the waveform is saved and displayed during emulation. • off: Do not launch simulator waveform GUI, and do not save `wdb` file • batch: Do not launch simulator waveform GUI, but save `wdb` file • gui: Launch simulator waveform GUI, and save `wdb` file • Default: off **Note:** The kernel needs to be compiled with debug enabled (`xocc -g`) for the waveform to be saved and displayed in the simulator GUI.

Converting Tcl Compilation Flow to XOCC

Starting with the 2016.3 release of SDAccel™, Tcl based compilation flow is no longer supported. Direct command line access via XOCC (and MakeFile) is the main entry point to use SDAccel services. This appendix provides guidance on how to convert Tcl based compilation script to XOCC based command line options. All examples in the SDAccel installation use Makefile/ XCOCC for compilation and can be used as additional reference.

The following sections list Tcl commands and their equivalent XOCC options. All XOCC options need to be provided to the XCOCC command in a single command line.

Solution

- Tcl

```
create_solution -name example_alpha -dir . -force
```

- XOCC

  XOCC does not require a solution to be explicitly created.

Host Code Management and Compilation

- Tcl

```
add_files "test-cl.c"
```

- XOCC

  In XOCC flow, the host code is managed by the user and compiled with the `xcpp` command line or Makefile to generate host executable.

`xcpp` is an SDAccel wrapper around the underlying system compiler / linkers (such as GCC/G++) to create a uniform frontend for the host computer compilation. One goal of the wrapper is to isolate from potential incompatibilities between preinstalled system compilers and the required SDAccel compiler versions.

```
xcpp -g -Wall -DFPGA_DEVICE -I/opt/SDx/2017.4/runtime/include/1_2 -c
test-cl.cpp -o test-cl.o
xcpp  -L/opt/SDx/2017.4/runtime/lib/x86_64 -lxilinxopencl -llmx6.0 -
lstdc++ test-cl.o -o mmult_ex
```

Device

- Tcl

```
add_device -vbnv xilinx:adm-pcie-7v3:1ddr:3.0
set_property device_repo_paths /path/to/custom_dsa [current_solution]
```

- XOCC

```
--platform xilinx:adm-pcie-7v3:1ddr:3.0
--xp prop:solution.device_repo_paths=/path/to/custom_dsa
```

Kernel Definition

- Tcl

```
create_kernel mmult -type clc
add_files -kernel [get_kernels mmult] "mmult1.cl"
```

- XOCC

```
--kernel mmult mmul1.cl
```

Setting Kernel Compile Flags

- Tcl

```
set_property kernel_flags "-DUSE2DDR=1" [get_kernels mmult]
```

- XOCC

```
-DUSE2DDR=1
```

Binary Container Definition

- Tcl

```
create_opencl_binary bin_mmult
set_property region "OCL_REGION_0" [get_opencl_binary bin_mmult]
create_compute_unit -opencl_binary [get_opencl_binary bin_mmult] -
kernel [get_kernels mmult] -name k1
create_compute_unit -opencl_binary [get_opencl_binary bin_mmult] -
kernel [get_kernels mmult] -name k2
```

- XOCC

```
--output bin_mmult.xclbin --nk mmult:2:k1.k2
```

Compile for Software Emulation

- Tcl

```
compile_emulation -flow cpu -opencl_binary [get_opencl_binary bin_mmult]
```

- XOCC

```
--target sw_emu
```

Compile for Hardware Emulation

- Tcl

```
compile_emulation -flow hardware -opencl_binary [get_opencl_binary
bin_mmult]
```

- XOCC

```
--target hw_emu
```

Run CPU and Hardware emulation Emulation

- Tcl

```
run_emulation -flow cpu -args "bin_mmult.xclbin"
run_emulation -flow hardware -args "bin_mmult.xclbin"
```

- XOCC

  Refer to Running Software and Hardware Emulation in XOCC Flow for details.

Build System

- Tcl

```
build_system
```

- XOCC

```
--target hw
```

Report Estimate

- Tcl

```
report_estimate
```

- XOCC

```
--report estimate
```

Package System

- Tcl

```
package_system
```

- XOCC

  XOCC does not have an equivalent option for `package_system`. Packaging of libraries and drivers for deployment is provided by another command, `xbinst`. Below is a simple command line example.

```
xbinst -f xilinx:adm-pcie-7v3:1ddr:3.0 -d 7v3
```

Putting All Together

- Tcl

```
create_solution -name example_alpha -dir . -force
add_device -vbnv xilinx:adm-pcie-7v3:1ddr:3.0

create_kernel mmult -type clc
add_files -kernel [get_kernels mmult] "mmult1.cl"
set_property kernel_flags "-DUSE2DDR=1" [get_kernels mmult]

create_opencl_binary bin_mmult
set_property region "OCL_REGION_0" [get_opencl_binary bin_mmult]
create_compute_unit -opencl_binary [get_opencl_binary bin_mmult] -
kernel [get_kernels mmult] -name k1
```

```
create_compute_unit -opencl_binary [get_opencl_binary bin_mmult] -
kernel [get_kernels mmult] -name k2

compile_emulation -flow cpu -opencl_binary [get_opencl_binary bin_mmult]
run_emulation -flow cpu -args "bin_mmult.xclbin"

compile_emulation -flow hardware -opencl_binary [get_opencl_binary
bin_mmult]
run_emulation -flow cpu -args "bin_mmult.xclbin"

build_system
```

- XOCC

  The following is the equivalent XOCC command line options of the Tcl commands above for compilation of Software emulation. Change `--target sw_emu` to `--target hw_emu` for compilation for hardware emulation or `--target hw` for compilation for system run.

```
xocc --target sw_emu --platform xilinx:adm-pcie-7v3:1ddr:3.0 \
     -DUSE2DDR=1 \
     --output bin_mmult.xclbin --nk mmutl:2:k1.k2
     --kernel mmult mmul1.c
```

Appendix J

Board Installations

SDAccel™ applications are executed in hardware using one of the listed FPGA cards or boards.

You can develop and run applications using two or more computers: a development computer to develop applications, on which you must have SDAccel installed; and one or more deployment computers, to run applications on, inside of which you must have an FPGA accelerator board installed. Also, you can use a single computer to perform both development and deployment roles; both and SDAccel and an accelerator card must be installed.

The procedure for installing a hardware accelerator board on a host computer is provided here and includes instructions for using the `xbinst` board installation utility, and the Vivado® Design Suite, to install the board. When you install SDAccel, the `xbinst` utility, the Vivado Design Suite, and any required cable drivers are included.

Follow the installation instructions appropriate for your FPGA accelerator board:

- Installing the KCU1500 Card
- Installing the VCU1525 Card

Installing the KCU1500 Card

The XIL-ACCEL-RD-KU115 card is a high-performance reconfigurable computing card for data center applications. It features the following:

- Kintex® UltraScale™ XCKU1500-2FLVB2104E FPGA
- Four GB DDR4 banks (16GB total)

Step 1: Preparing Board Installation Files

SDAccel™ provides a utility, `xbinst`, that generates firmware and driver files for the target board plugged into the host computer.

Run the following commands to prepare files for the target card installation. See the Appendix F: xbinst Command Reference for more details on `xbinst` utility.

***Note*:** Depending on the target location, you might need to run some commands with `root` or `sudo` privilege. You must change access permissions to enabel read access for all machine users.

Run the following commands:

```
$ sudo mkdir -p /opt/dsa/xilinx_kcu1500_dynamic_5_0
$ sudo xbinst --platform xilinx_kcu1500_dynamic_5_0
-d /opt/dsa/xilinx_kcu1500_dynamic_5_0

****** xbinst v2017.4 (64-bit)
  **** SW Build 2080852 on Sun Dec 10 18:06:50 MST 2017
    ** Copyright 1986-2017 Xilinx, Inc. All Rights Reserved.

INFO: [XBINST 60-895] Target platform:
/opt/Xilinx/SDx/2017.4/platforms/xilinx_kcu1500_dynamic_5_0/
xilinx_kcu1500_dynamic_5_0.xpfm
INFO: [XBINST 60-267] Packaging for PCIe...
INFO: Adding section [CLEARING_BITSTREAM (1)] using: 'xilinx_kcu1500_'
(1686170 Bytes)
INFO: Adding section [FIRMWARE (3)] using: 'mgmt' (14548 Bytes)
INFO: Adding section [SCHED_FIRMWARE (5)] using: 'sched' (9488 Bytes)
Successfully completed 'xclbincat'
INFO: [XBINST 60-268] Packaging for PCIe...COMPLETE
INFO: [XBINST 60-667] xbinst has successfully created a board installation
directory at /opt/dsa/xilinx_kcu1500_dynamic_5_0.
```

***Note*:** The actual version and build date could vary based on the release of the tools.

Make a note of the board installation directory. This procedure uses `/opt/dsa/xilinx_kcu1500_dynamic_5_0` as an example.

Copy the following files to the programming computer:

```
/opt/dsa/xilinx_kcu1500_dynamic_5_0/xbinst/firmware/
xilinx_kcu1500_dynamic_5_0_primary.mcs
```

```
/opt/dsa/xilinx_kcu1500_dynamic_5_0/xbinst/firmware/
xilinx_kcu1500_dynamic_5_0_secondary.mcs
```

Make a note of the file location on the programming computer because it is required to program the configuration memory in a later step.

Step 2: Setting up the Card and Computer

1. Make sure the host computer is completely turned off.
2. Install the KCU1500 card into an open PCIE® slot in the hots computer.

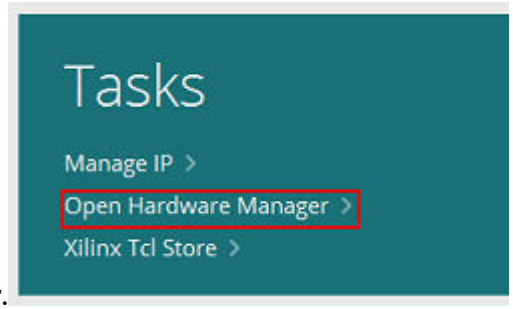
Follow host computer manufacturer recommendations to ensure proper mounting and adequate cooling.

3. Turn on the host computer.

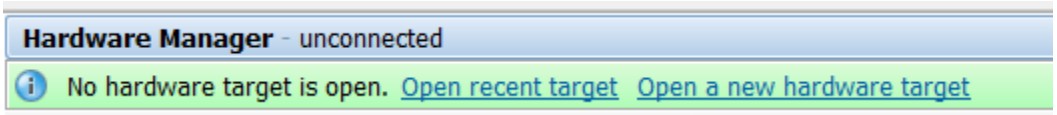
Step 3: Programming the Base Platform

All applications compiled by the SDAccel™ compiler for the KCU1500 card are compiled against a specific device. A device is a combination of interfaces and infrastructure components on the card, which are required for proper execution of the user program. The base device program or firmware is different for all devices. This program must be loaded onto the FPGA before the user applicationis loaded. To program the firmware program:

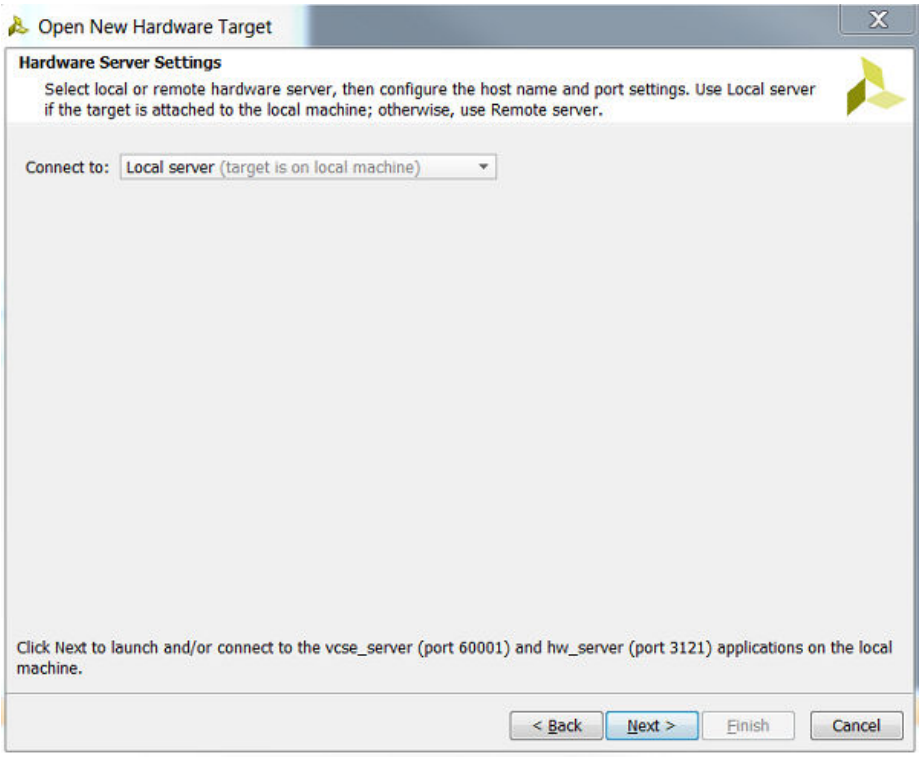
1. Connect the KCU1500 card to the programming computer with an installation of Vivado® Design Suite using a USB cable as shown in the following figure.
2. On the programming computer, start the Vivado® Design Suite, then on the **Welcome** page, select **Open Hardware Manager**.



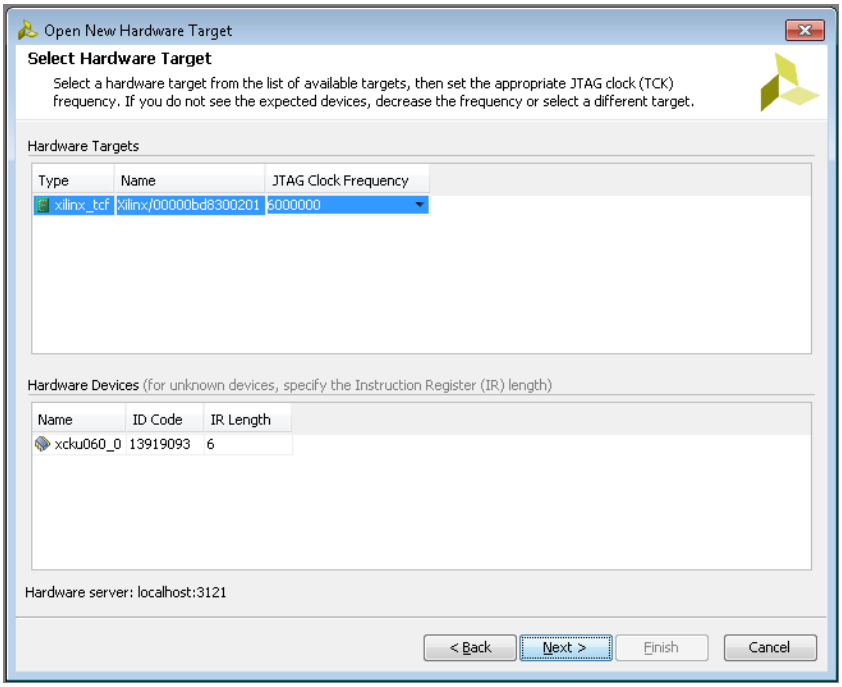
3. Select **Open a New Hardware Target**.



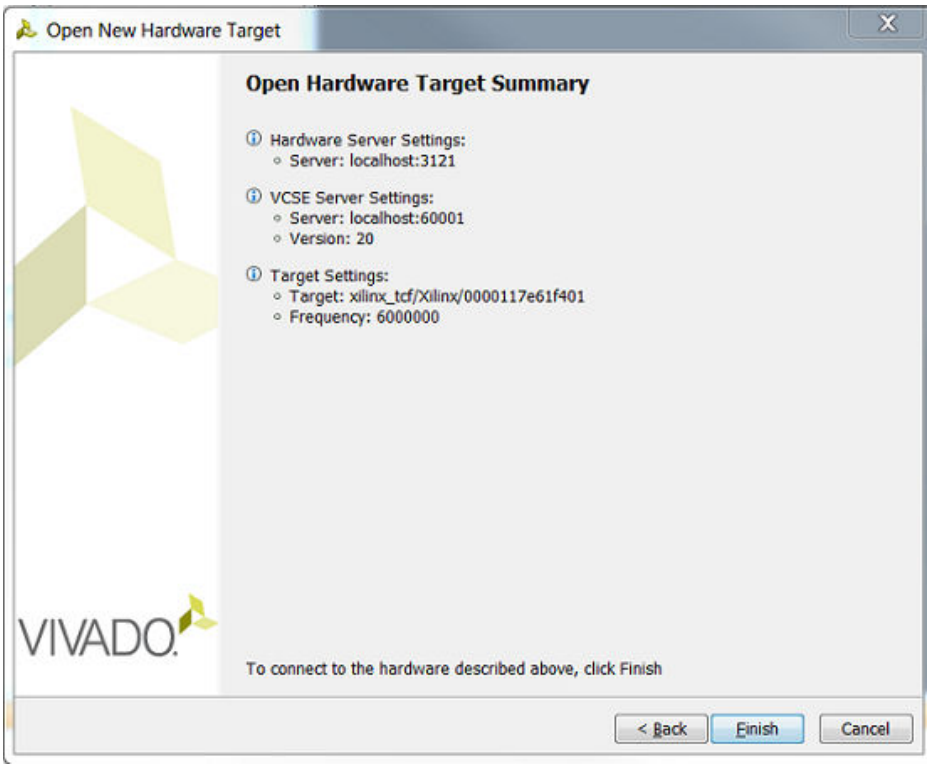
4. Click **Next**.
5. Select **Local server** in the **Connect to** field and click **Next**.



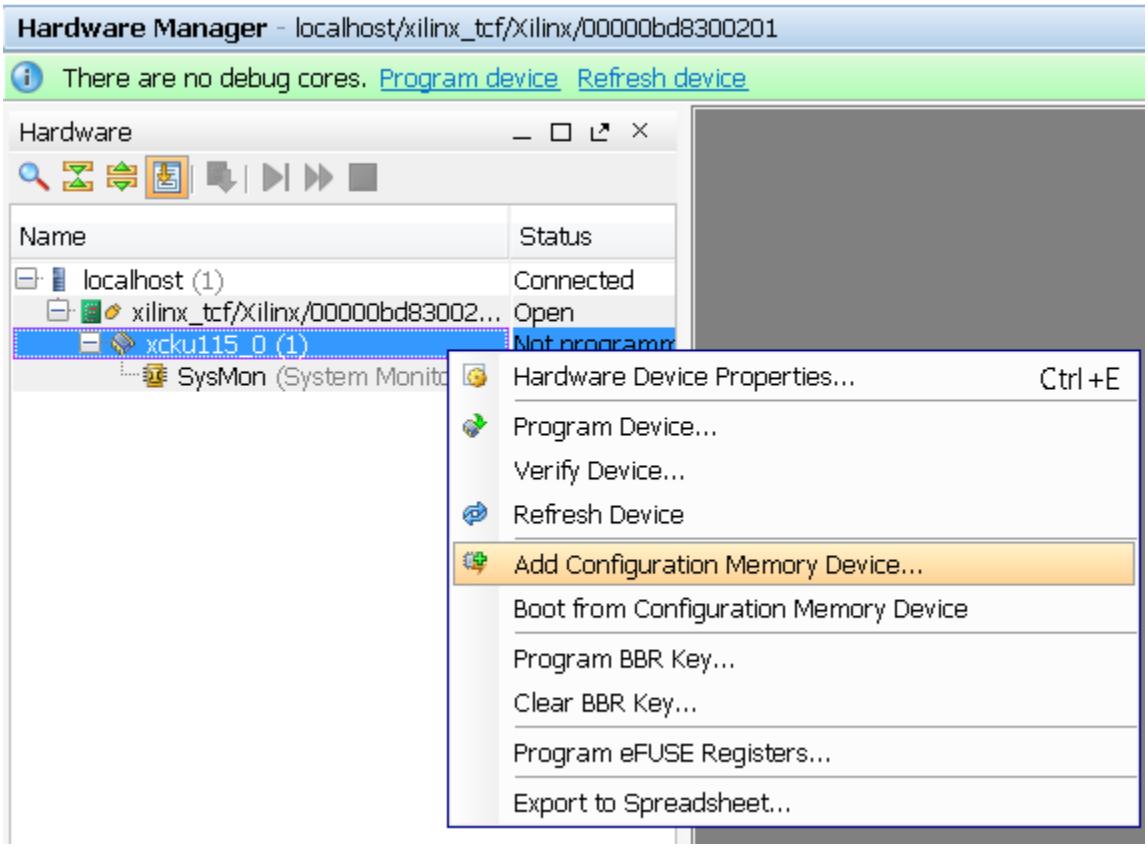
6. In the **Open New Hardware Target** window, select **xilinx_tcf** and click **Next**.



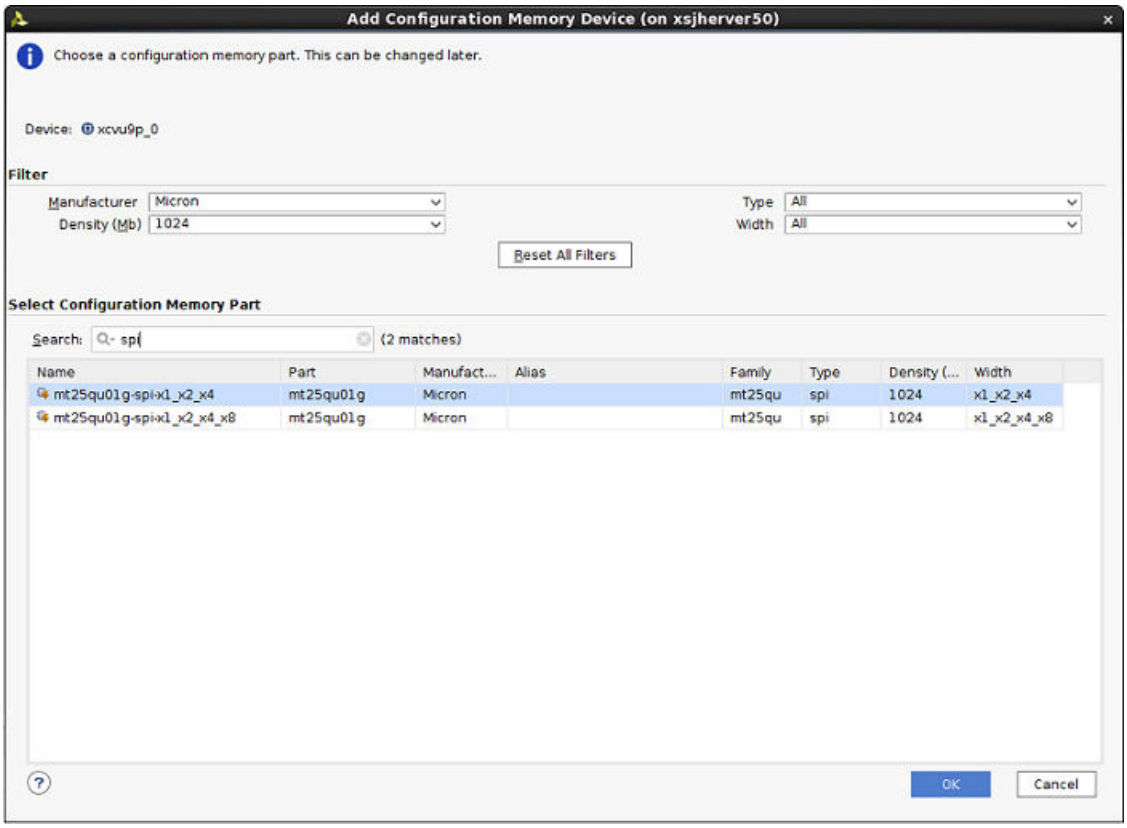
7. In the **Open Hardware Target Summary** window, click **Finish**.



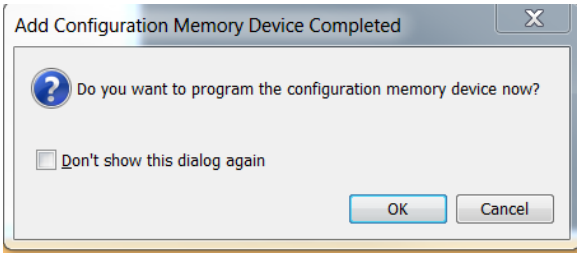
8. Right-click your FPGA (`xcku115_0`) and select **Add Configuration Memory Device**.



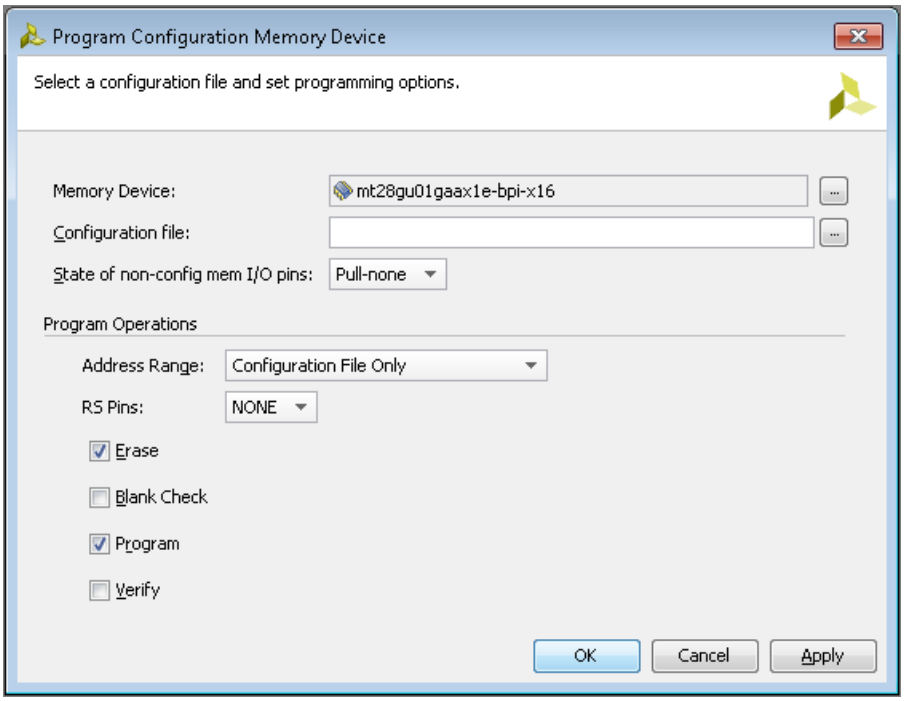
9. Select `mt25qu512-spi-x1_x2_x4_x8` as the configuration memory.



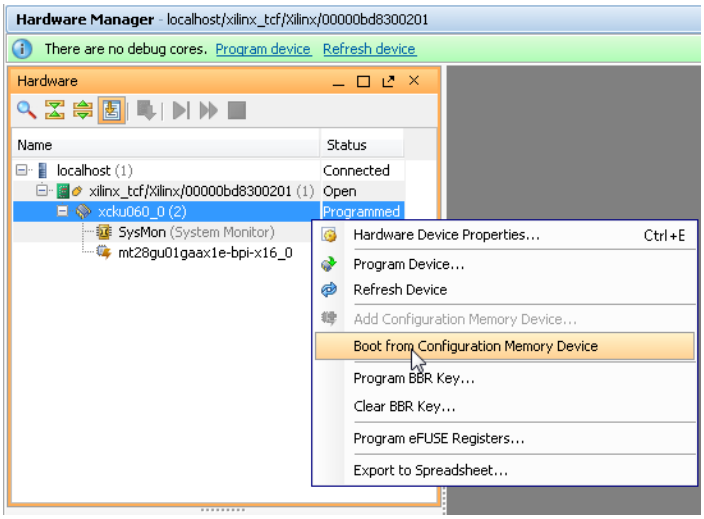
10. Click **OK** to program the configuration memory.



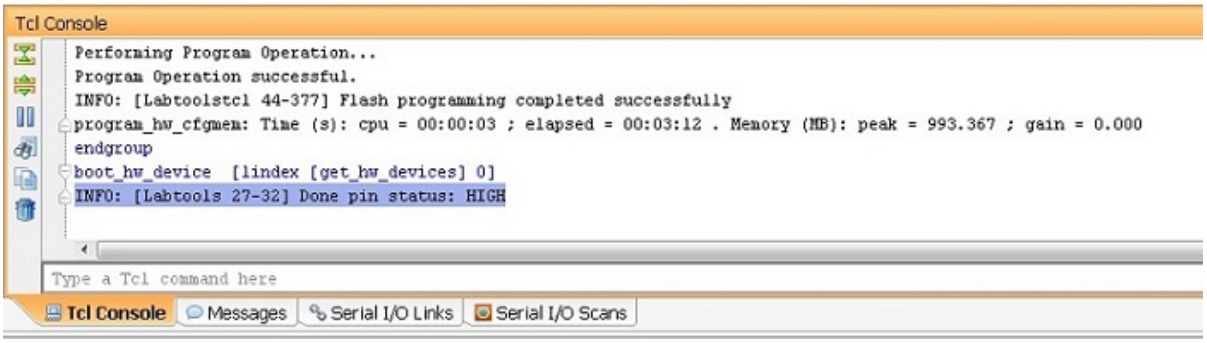
11. In the **Programming Configuration Memory Device** window, go to the **Configuration File** entry box, browse to and select the MCS file (`xilinx_kcu1500_dynamic_5_0_primary.mcs`) that you copied to the programming computer in Step 1.
12. Go to the **Configuration File 2** entry box, browse to, and select the second MCS file (`xilinx_kcu1500_dynamic_5_0_secondary.mcs`).
13. Verify all other settings as shown in the **Program Configuration Memory Device** window. Click **OK** to start programming the configuration memory.



14. After the memory has been configured, right-click the FPGA (**xcku115_0**) and select **Boot from Configuration Memory Device**



15. The Tcl console displays `Done pin status: HIGH` after the FPGA device is booted successfully.



16. Reboot the host computer.

***Note*:** Programming of the device firmware is required only once per device support archive (dsa). All applications targeting the same dsa can share a single programming instance of the card firmware.

Step 4: Installing Driver for the Card

You must install proper drivers for the card before you can use it to run SDAccel™ applications. Use the following instructions to install the required drivers.

1. Change to the board installation directory generated in Step 1 and run installation script:

```
$cd /opt/dsa/xilinx_kcu1500_dynamic_5_0/xbinst
$sudo ./install.sh -f yes -k yes
```

   This will do the following.

   - Compile and force (`-f yes`) the installation of Linux kernel device drivers.
   - Install the firmware to the Linux firmware area.
   - Install Xilinx OpenCL Installable Client Driver (ICD) to `/etc/OpenCL/vendors`. The OpenCL ICD allows multiple implementations of OpenCL to co-exist on the same system. It allows applications to choose a platform from the list of installed platforms and dispatches OpenCL API calls to the underlying implementation.

2. Generate a `setup.sh` (Bash) or `setup.csh` (for csh/tcsh) to set up the runtime environment. You must source the setup script before running any applicationon the target FPGA card: for example: `source /opt/dsa/xilinx_kcu1500_dynamic_5_0/xbinst/setup.sh`.

Step 5: Verifying Successful Board Installation

During execution of `xbinst`, a simple prebuild executable and the associated `xclbin` is included in the "test" directory. This executable lets you check that the drivers and DSA are correctly installed and that your setup is operating as expected. The following are the steps required to perform the installation validation:

1. Source the `setup.sh` (Bash) or `setup.csh` (csh/tcsh):

```
$ source /opt/dsa/xilinx_kcu1500_dynamic_5_0/xbinst/setup.[sh|csh]
```

2. Change into the test directory and run the validation executable.

***Note*:** If the test directory is not write enabled, it might be necessary to run the executable from a different directory with an absolute path to the `verify.xclbin`.

3. Type the following:

```
$ cd /opt/dsa/<file_path>xilinx_kcu1500_dynamic_5_0/xbinst/test \
$ ./verify.exe verify.xclbin
```

Or, type:

```
$ cd /tmp
$ /opt/dsa/<file_path>/xilinx_kcu1500_dynamic_5_0/xbinst/verify.exe \
$ /opt/dsa/<file_path>/xilinx_kcu1500_dynamic_5_0/xbinst./verify .xclbin
```

Your system is correctly functioning if the output looks similar to the following:

```
$ CL_PLATFORM_VENDOR Xilinx
$ CL_PLATFORM_NAME Xilinx
$ loading verify.xclbin
$ RESULT:
$ Hello World
```

Installing the VCU1525 Card

The Xilinx® VCU1525 card is a high-performance reconfigurable computing card for data center applications. It features:

- A Virtex® Zynq® UltraScale+™ FPGA
- Four 16GB of DDR4 banks (64GB total)

Step 1: Prepare Board Installation Files

SDAccel™ provides the `xbinst`utility, that generates firmware and driver files for the target board plugged in the host computer. Run the following commands to prepare files for the target card installation. See Appendix F: xbinst Command Reference for more details on the `xbinst` utility.

Depending on the target location, some commands must be run with `root` or `sudo` privilege; otherwise, you would have to change access permissions to enable read access for all users on that system.

Use the following commands:

```
$ sudo mkdir /opt/dsa/xilinx'vcu1525_dynamic_5_0
$ sudo xbinst ---platform xilinx_vcu_1525_dynamic_5_0 -d
```

```
/opt/dsa/xilinx_vcu_1525_dynamic_5_0
```

You should see output similar to the following:

```
****** xbinst v2017.4 (64-bit)
  **** SW Build 2080852 on Sun Dec 10 18:06:50 MST 2017
    ** Copyright 1986-2017 Xilinx, Inc. All Rights Reserved.

INFO: [XBINST 60-895]    Target platform: /opt/Xilinx/SDx/2017.4/platforms/
xilinx_vcu1525_dynamic_5_0/xilinx_vcu1525_dynamic_5_0.xpfm
INFO: [XBINST 60-267] Packaging for PCIe...
INFO: Adding section [FIRMWARE (3)] using: 'mgmt' (14548 Bytes)
INFO: Adding section [SCHED_FIRMWARE (5)] using: 'sched' (9488 Bytes)
Successfully completed 'xclbincat'
INFO: [XBINST 60-268] Packaging for PCIe...COMPLETE
INFO: [XBINST 60-667] xbinst has successfully created a board installation
directory at /opt/dsa/ xilinx_vcu1525_dynamic_5_0.
```

Make a note of the file location on the programming computer because it will be required for programming the configuration memory at a later stage.

Step 2. Setting Up the Card and Computer

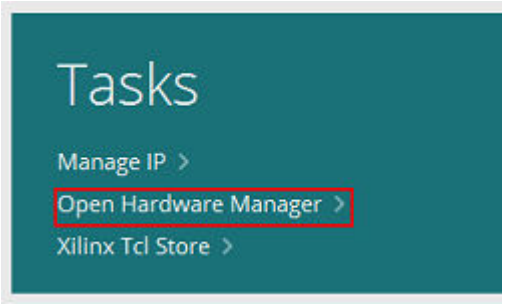
1. Make sure the host computer is completely turned off.
2. Install the Xilinx® VCU1525 FPGA board in an open PCIe® slot on the host computer.
3. Turn on the host computer.

Note: Follow the host computer manufacturer recommendations to ensure proper mounting and adequate cooling.

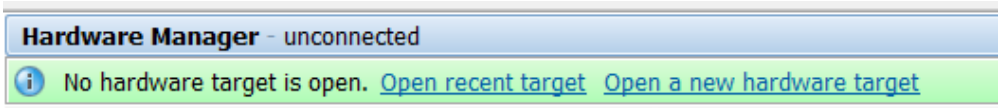
Step 3: Programming the Base Platform

All applications compiled by the SDAccel™ compiler for the data card are compiled against a specific device. A device is a combination of interfaces and infrastructure components on the card that are required for proper execution of the user program. The base device program or firmware is different for all devices. This program must be loaded onto the FPGA before the user application is loaded. To program the firmware program:

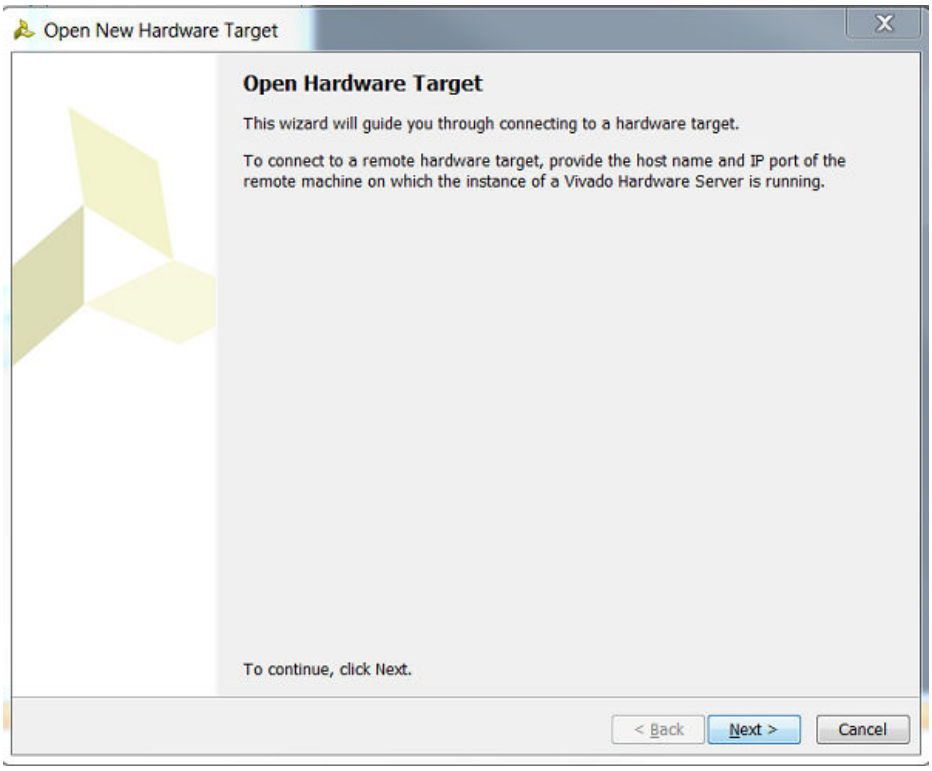
1. Using a micro USB cable, connect between the Xilinx® VCU1525 card and the programming computer with an installation of the Vivado Design Suite.
2. On the programming computer, start the Vivado Design Suite **Open Hardware Manager**.



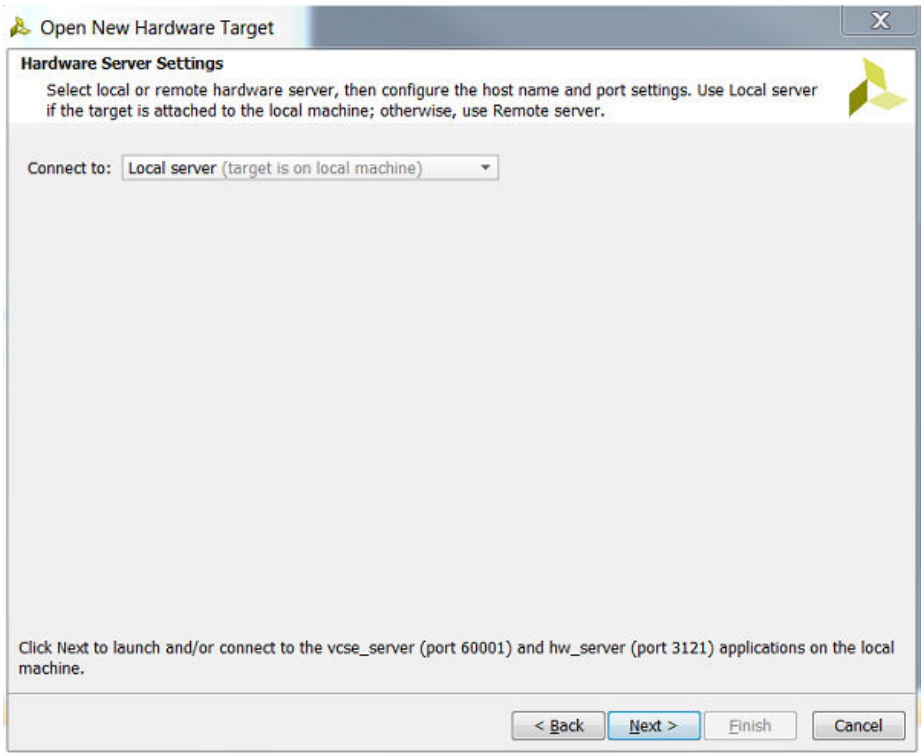
3. Select **Open a New Hardware Target**.



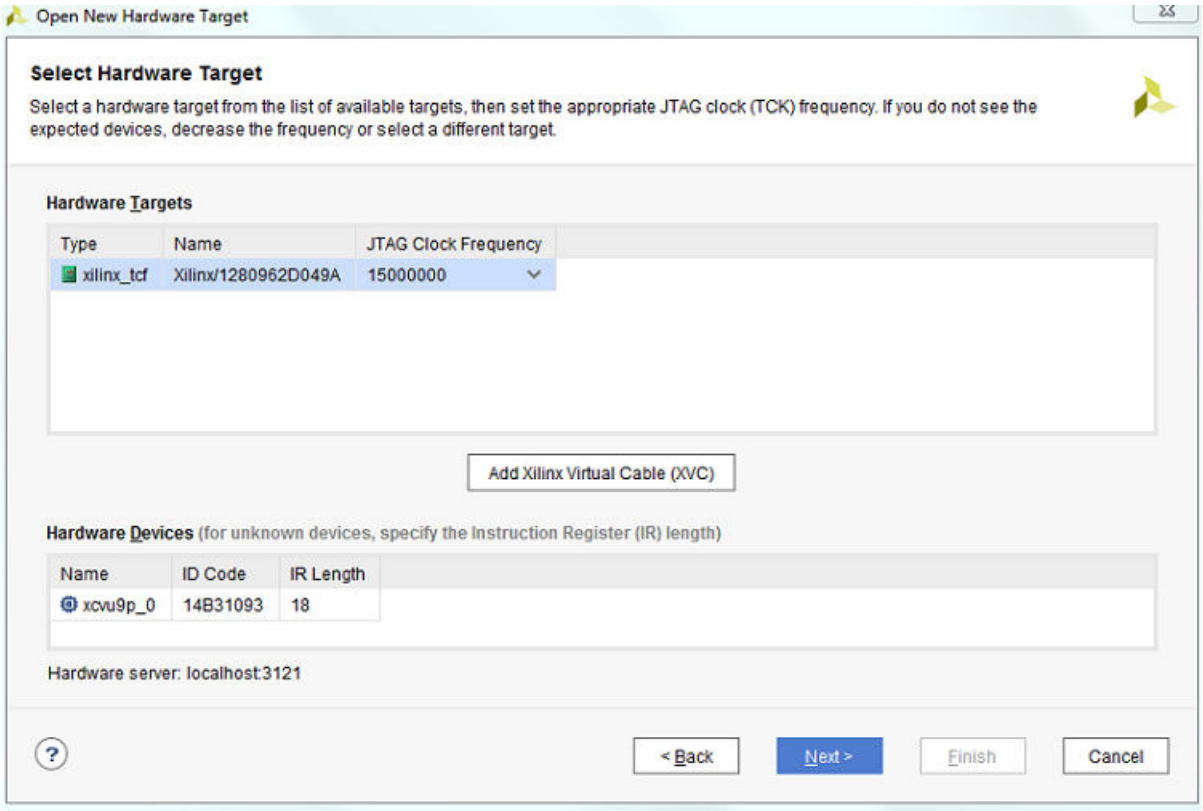
4. Click **Next** in the Open Hardware Target window.



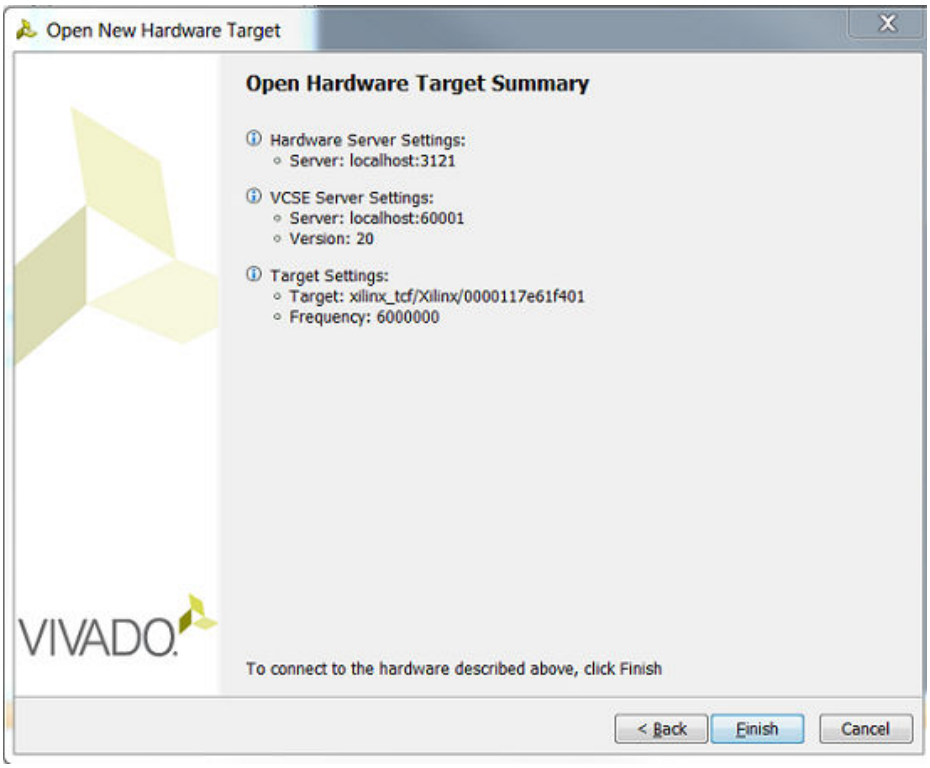
5. Select **Local server** in the **Connect to** field and click **Next**.



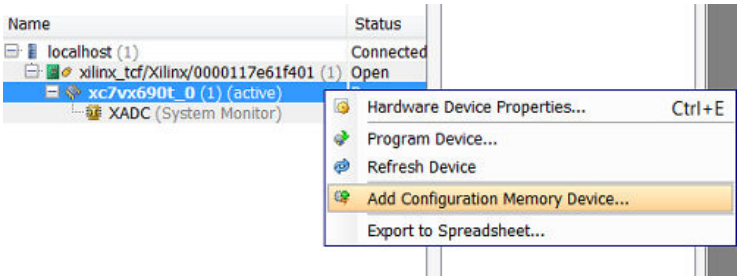
6. In the Open New Hardware Target window, select **xilinx_tcf** and click **Next**.



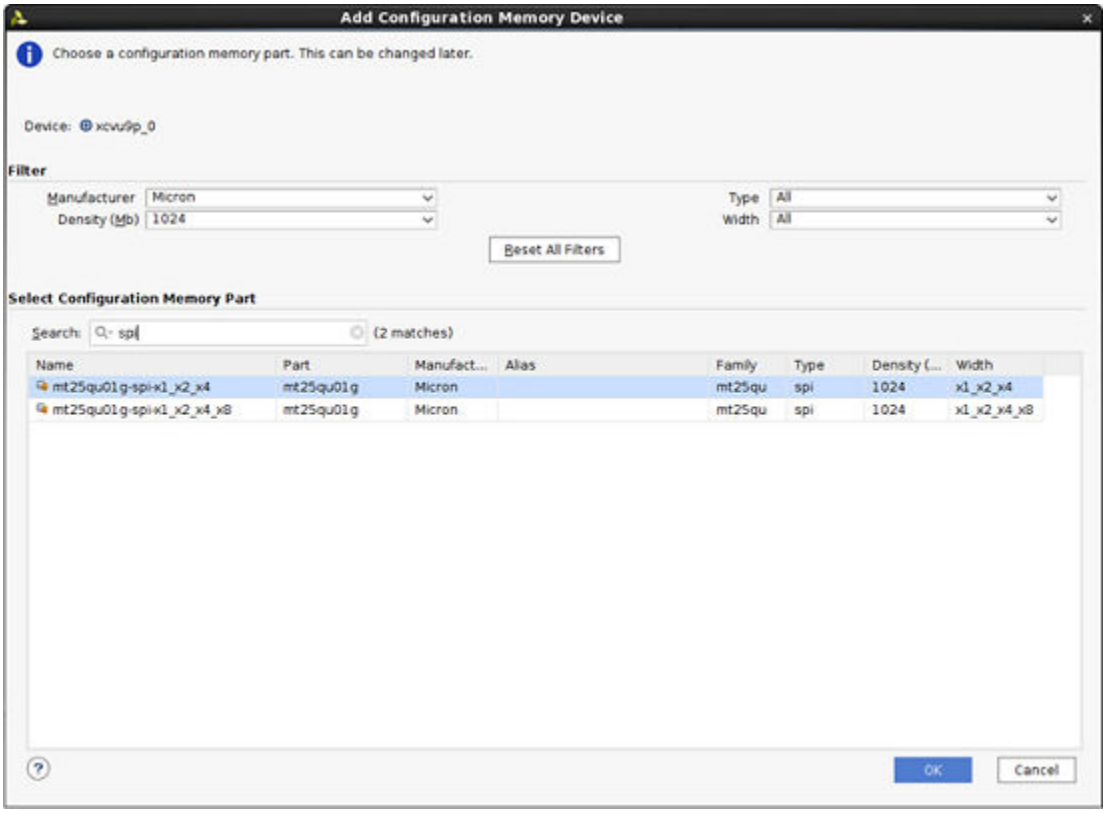
7. In the Open Hardware Target Summary window, click **Finish**.



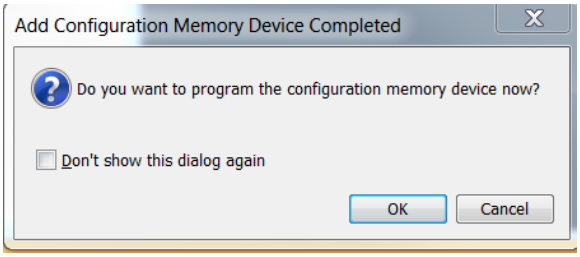
8. Right-click the FPGA (**vu9p_0**) and select **Add Configuration Memory Device**.



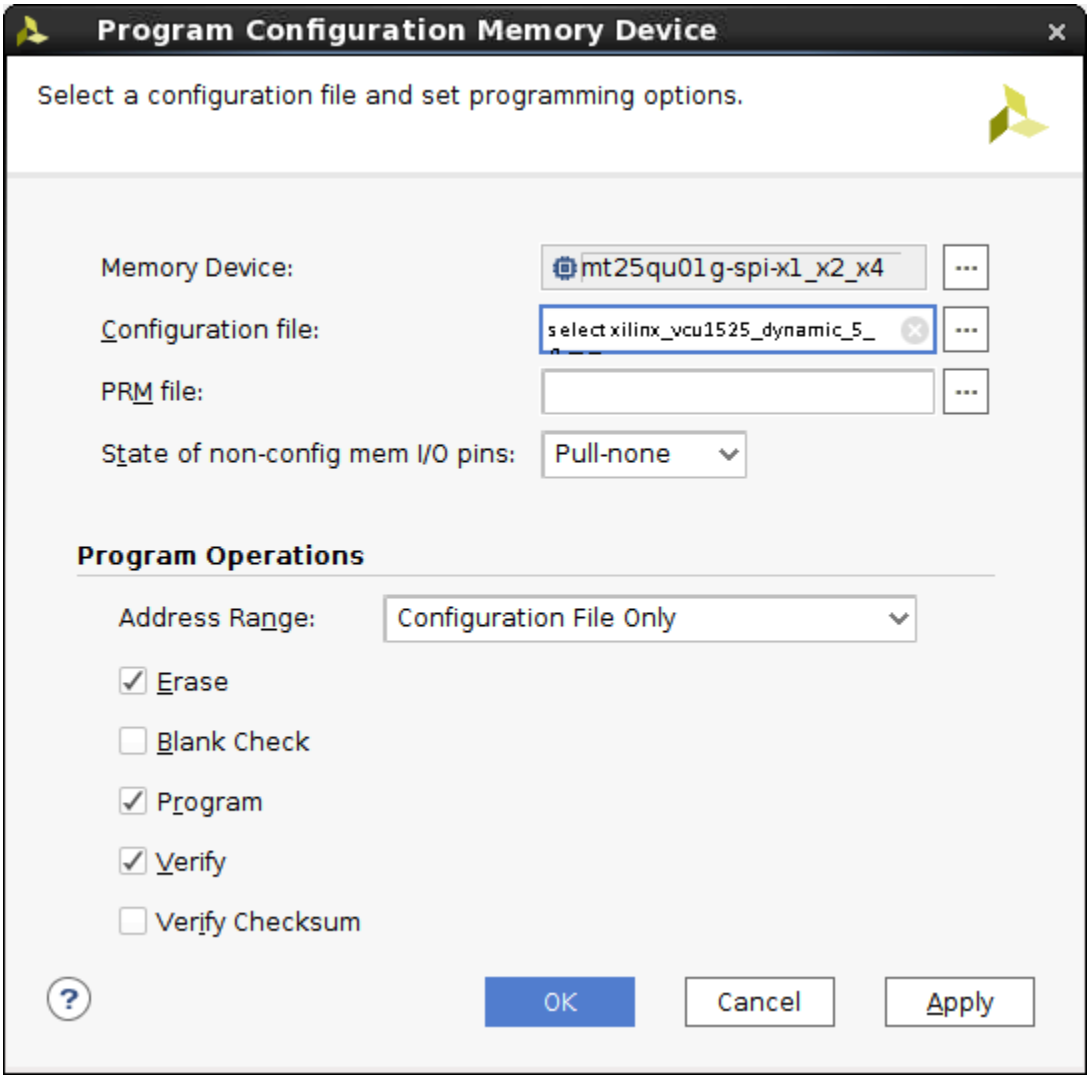
9. Select **mt25qu01g-spi-x1_x2_x4** as the configuration memory.



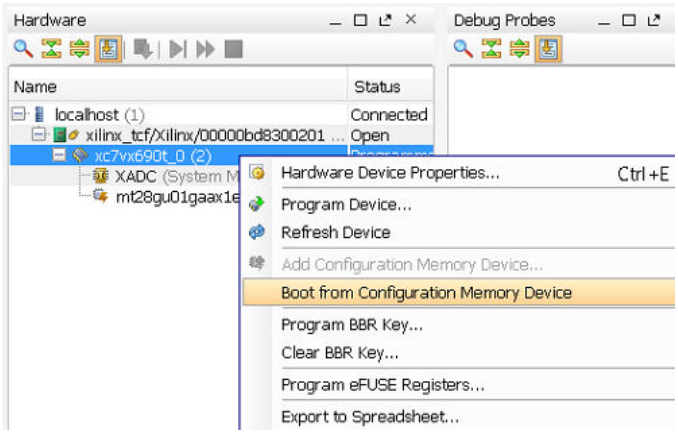
10. Click **OK** to program the configuration memory.



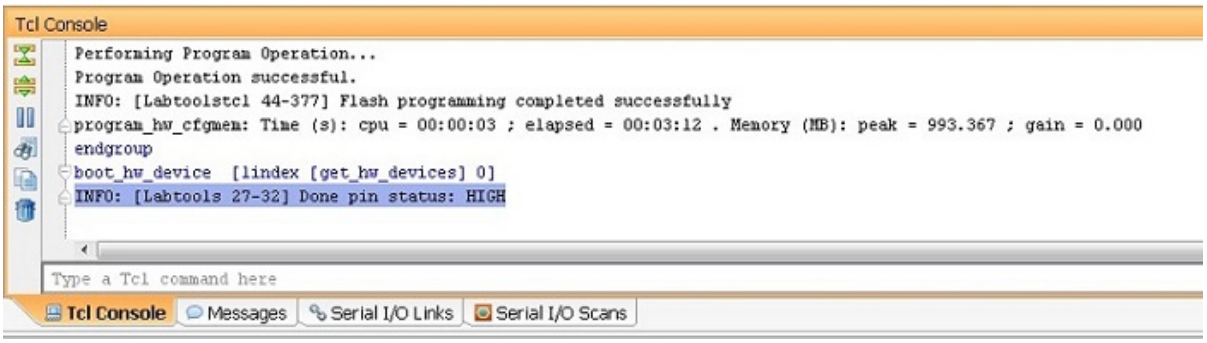
11. In the **Programming Configuration Memory Device** window, go to the **Configuration File** entry box, browse to, and select the MCS file (**xilinx_vcu1525_dynamic_5_0.mcs**) that you copied to the programming computer in Step 1. Do not use a configuration file 2. Verify all other settings as shown in the **Program Configuration Memory Device** window. Click **OK** to start programming the configuration memory.



12. After the memory has been configured, right-click the FPGA (**xcku115_0**) and select **Boot From Configuration Device**.



The Tcl Console displays `Done pin status: HIGH` after the FPGA is booted successfully.



13. Reboot the host computer.

***Note*:** Programming of the device firmware is required only once per device. All applications targeting the same device can share a single programming instance of the card firmware.

Step 4: Installing Driver for the Card

You must install proper drivers for the card before you can use it to run SDAccel™ applications. The following instructions describe how to install the required drivers.

Change to the board installation directory generated in Step 1, and run the installation script:

```
$ cd /opt/dsa/xilinx_vcu1525_dynamic_5_0/xbinst
$ sudo ./install.sh -f yes -k yes
```

This will do the following:

- Compile and force (`-f yes`) the install of Linux kernel device drivers.
- Install the firmware to the Linux firmware area.

- Install the Xilinx® OpenCL™ Installable Client Driver (ICD) to `/etc/OpenCL/vendors`. The OpenCL ICD allows multiple implementations of OpenCL to co-exist on the same system. It allows applications to choose a platform from the list of installed platforms and dispatches OpenCL API calls to the underlying implementation.
- Generate a `setup.sh` (Bash) or `setup.csh` (for csh/tcsh) to set up the runtime environment. You must source the setup script before running any application on the target FPGA card. For example:

```
$ source /opt/dsa/xilinx_kcu1500_dynamic_5_0/xbinst/setup.sh
```

Note: If `-k no` is used,only the setup scripts are generated, no driver is generated and this can be run without `sudo` privileges.

Step 5: Verifying Successful Board Installation

During execution of `xbinst`, a simple prebuild executable and the associated `xclbin` is included in the "test" directory. This executable lets you check that the drivers and DSA are correctly installed and that your setup is operating as expected. The following are the steps required to perform the installation validation:

1. Source the `setup.sh` (Bash) or `setup.csh` (csh/tcsh):

```
$ source /opt/dsa/xilinx_kcu1500_dynamic_5_0/xbinst/setup.[sh|csh]
```

2. Change into the test directory and run the validation executable.

Note: If the test directory is not write enabled, it might be necessary to run the executable from a different directory with an absolute path to the `verify.xclbin`.

3. Type the following:

```
$ cd /opt/dsa/<file_path>xilinx_kcu1500_dynamic_5_0/xbinst/test \
$ ./verify.exe verify.xclbin
```

Or, type:

```
$ cd /tmp
$ /opt/dsa/<file_path>/xilinx_kcu1500_dynamic_5_0/xbinst/verify.exe \
$ /opt/dsa/<file_path>/xilinx_kcu1500_dynamic_5_0/xbinst./verify .xclbin
```

Your system is correctly functioning if the output looks similar to the following:

```
$ CL_PLATFORM_VENDOR Xilinx
$ CL_PLATFORM_NAME Xilinx
$ loading verify.xclbin
$ RESULT:
$ Hello World
```

Appendix K

Additional Resources and Legal Notices

Xilinx Resources

For support resources such as Answers, Documentation, Downloads, and Forums, see Xilinx Support.

Solution Centers

See the Xilinx Solution Centers for support on devices, software tools, and intellectual property at all stages of the design cycle. Topics include design assistance, advisories, and troubleshooting tips

References

1. *SDx Environments Release Notes, Installation, and Licensing Guide* (UG1238)
2. *SDAccel Environment User Guide* (UG1023)
3. *SDAccel Environment Optimization Guide* (UG1207)
4. *SDAccel Environment Tutorial: Introduction* (UG1021)
5. *SDAccel Environment Platform Development Guide* (UG1164)
6. SDAccel Development Environment web page
7. Vivado® Design Suite Documentation
8. *Vivado Design Suite User Guide: Designing IP Subsystems using IP Integrator* (UG994)
9. *Vivado Design Suite User Guide: Creating and Packaging Custom IP* (UG1118)
10. *Vivado Design Suite User Guide: Partial Reconfiguration* (UG909)
11. *Vivado Design Suite User Guide: High level Synthesis* (UG902)
12. *UltraFast Design Methodology Guide for the Vivado Design Suite* (UG949)
13. *Vivado Design Suite Properties Reference Guide* (UG912)
14. Khronos Group web page: Documentation for the OpenCL standard

15. Alpha Data web page: Documentation for the ADM-PCIE-7V3 Card
16. Pico Computing web page: Documentation for the M-505-K325T card and the EX400 Card

Please Read: Important Legal Notices

Copyright