SDAccel Environment Debugging Guide

UG1281 (v2018.2.xdf) October 2, 2018

XILINX®

Revision History

The following table shows the revision history for this document.

Section	Revision Summary
10/02/2018 Version 2018.2.xdf	
Throughout document	Changed `xbsak` to `xbutil`.
07/02/2018 Version 2018.2	
Entire document	Minor editorial changes.
06/06/2018 Version 2018.2	
General updates	Initial Xilinx release.

Table of Contents

Chapter 1

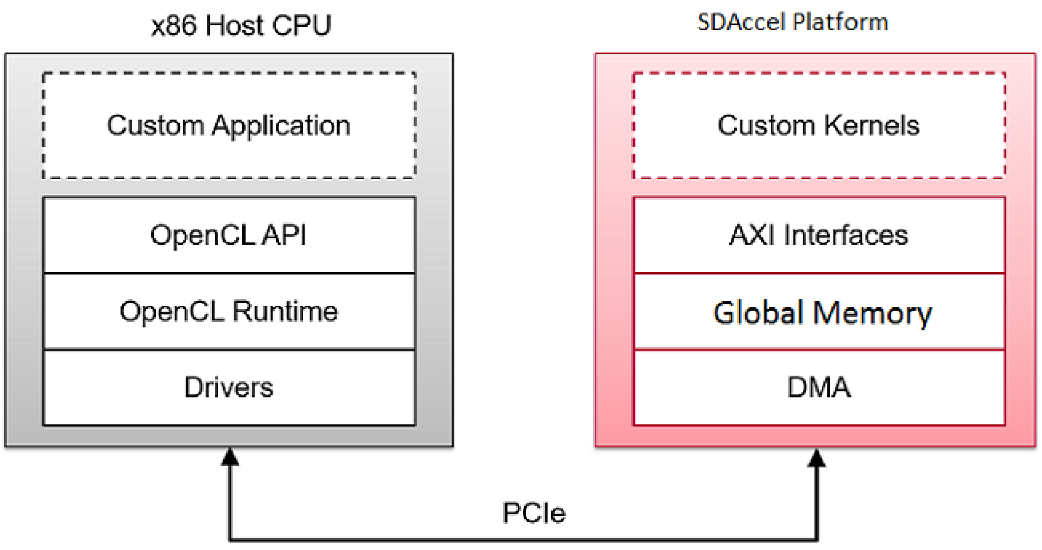
Introduction to Debugging in SDAccel

This book is intended to introduce the SDAccel™ debugging capabilities. The goal is to provide detailed instructions on how to analyze any failure encountered within the SDAccel flow. It is important to note, if no tool problem is encountered and the behavior of the design is deemed functionally correct, you should look for answers in the *SDAccel Environment Profiling and Optimization Guide* (UG1207) to determine if the performance of the design can be further improved.

Execution Model of an SDAccel Application

The SDAccel™ environment is designed to provide a simplified development experience for FPGA-based software acceleration platforms. The general structure of the SDAccel acceleration platform is shown in the following figure.

Figure 1: **Architecture of an SDAccel Application**



The custom application is running on the host x86 server and uses OpenCL™ API calls to interact with the FPGA accelerators. The SDAccel run time manages those interactions. The application is written in C/C++ using OpenCL APIs. The custom kernels are running within a Xilinx® FPGA through the SDAccel run time that manages interactions between the host application and the accelerator. Communication between the host x86 machine and the SDAccel accelerator board occurs across the PCIe® bus.

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. The kernels running on the FPGA can have one or more memory interfaces. The connection from the memory banks to those memory interfaces are programmable and determined by linking options of the compiler.

The SDAccel execution model follows these steps:

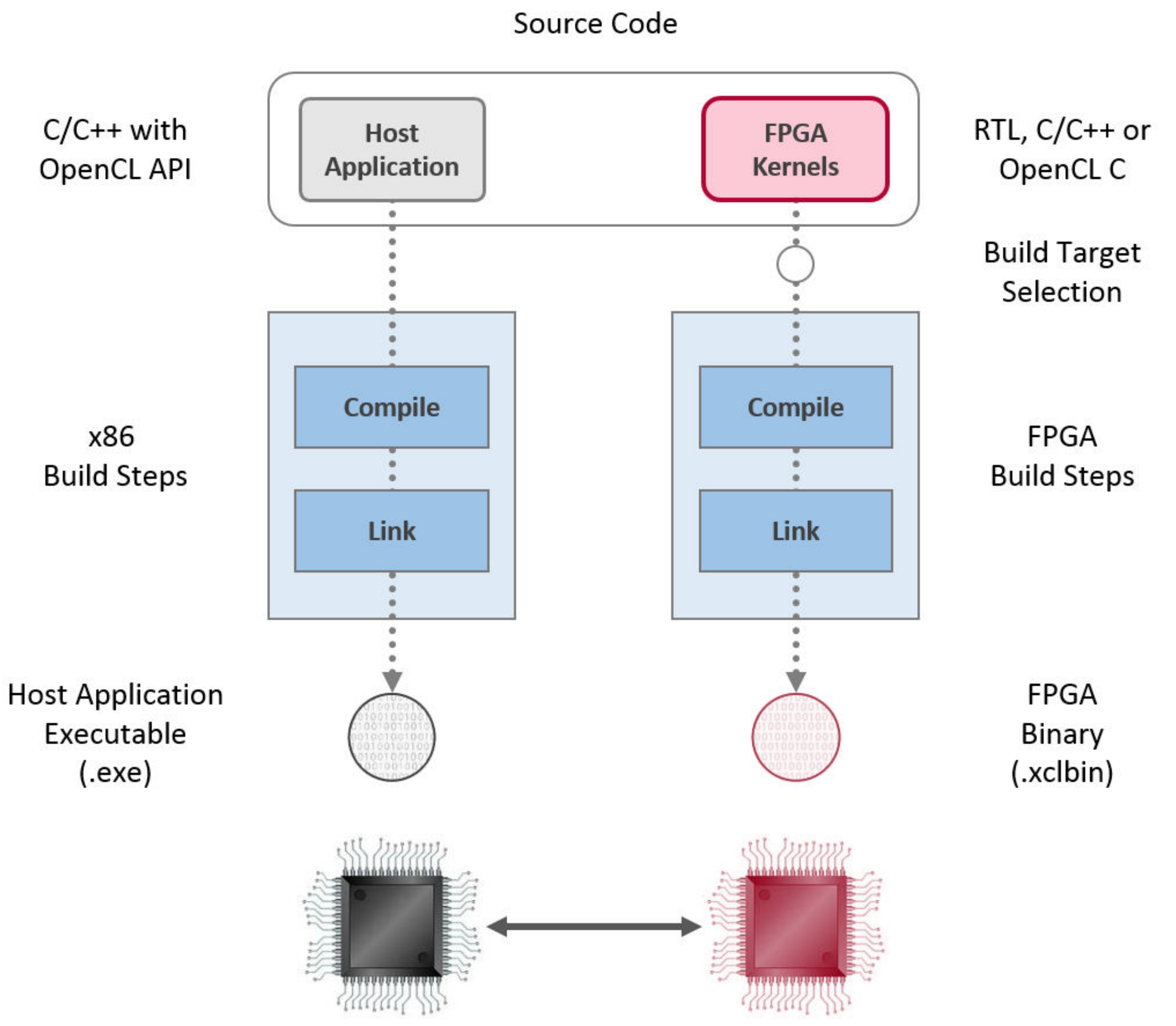
1. The host application writes the data needed by a kernel into the global memory of the SDAccel hardware platform, through the PCIe interface.
2. The host programs the kernel with its input parameters.
3. The host application triggers the execution of the kernel function on the FPGA.
4. The kernel performs the required computation, reading and writing data from global memory as necessary.
5. Kernels write data back to the memory banks, and notify the host that it has completed its task.
6. The host application reads data back from global memory into the host memory space, and continues processing as needed.

The FPGA can accommodate multiple kernel instances at one time; either different types of kernels or multiple instances of the same kernel. The SDAccel OpenCL run time transparently orchestrates the communication between the host application and the kernels in the FPGAs. The number of instances of a kernel is variable and determined by the host program and compilation options.

SDAccel Build Process

The SDAccel™ environment offers all of the features of a standard software development environment: an optimized compiler for host applications, cross-compilers for the FPGA, robust debugging environment to help you identify and resolve issues in the code, and performance profilers to let you identify the bottlenecks and optimize your code. Within this environment the SDAccel build process uses a standard compilation and linking process for both the software elements and the hardware elements of the project. As shown in the following figure, the host application is built through one process using standard GCC, and the FPGA binary is built through a separate process using the Xilinx® `xocc` compiler.

Figure 2: **Software/Hardware Build Process**



1. Host application build process using GCC:

- Each host application source file is compiled to an object file (`.o`).
- The object files (`.o`) are linked with the Xilinx SDAccel runtime shared library to create the executable (`.exe`).

2. FPGA build process using `XOCC`:

   - Each kernel is independently compiled to a Xilinx object (`.xo`) file.
     - C/C++ and OpenCL C kernels are compiled for implementation on an FPGA using the `XOCC` compiler. This step leverages the Vivado® HLS compiler. The same pragmas and attributes supported by Vivado HLS can be used in C/C++ and OpenCL C kernel source code to specify the desired kernel micro-architecture and control the result of the compilation process.
     - RTL kernels are compiled using the `package_xo` utility. The RTL kernel wizard in the SDAccel environment can be used to simplify this process.
   - The kernel `.xo` files are linked with the hardware platform (`.dsa`) to create the FPGA binary (`.xclbin`). Important architectural aspects are determined during the link step. In particular, this is where connections from kernel ports to global memory banks are established and where the number of instances for each kernel is specified.
     - When the build target is software or hardware emulation, as described below, `xocc` generates simulation models of the device contents.
     - When the build target is the system, or actual hardware, `xocc` generates the FPGA binary for the device leveraging the Vivado® Design Suite to run synthesis and implementation.

Note: The `xocc` compiler automatically uses the Vivado HLS and Vivado Design Suite tools to build the kernels to run on the FPGA platform. It uses these tools with pre-defined settings which have proven to provide good quality of results. Using the SDAccel environment and the `xocc` compiler does not require knowledge of these tools; however, hardware savvy developers can fully leverage these tools and use all their available features to implement kernels.

Build Targets

The SDAccel build process generates the host application executable (`.exe`) and the FPGA binary (`.xclbin`). The SDAccel build target defines the nature of FPGA binary generated by the build process.

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

- Software Emulation (`sw_emulator`): Both the host application code and the kernel code are compiled to run on the x86 processor. This allows iterative algorithm refinement through fast build and run loops. This target is useful to identify syntax issues, perform source-level debugging of the kernel code running together with application and verify the behavior of the system.

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

SDAccel Debug Flow Overview

This section presents the general debug flow of SDAccel™. It will take the reader through the general steps of a proven development process. This process allows the user to focus rapidly on potential errors in the design. This sets the baseline for developers indicating where to start if an error occurs in their adopted development steps.

The Debug Flow described here assumes that an SDAccel platform board is installed and the initial setup checks have passed. It is possible to configure the SDAccel environment to work with custom hardware platforms that will require a platform DSA which defines the foundational components of the board.

The SDAccel™ environment provides application-level debug features which allow the host code, the kernel code, and the interactions between them to be efficiently debugged. The recommended application-level debugging flow consists of three levels of debugging that can be summarized as:

1. Software Emulation
2. Hardware Emulation
3. Hardware Execution

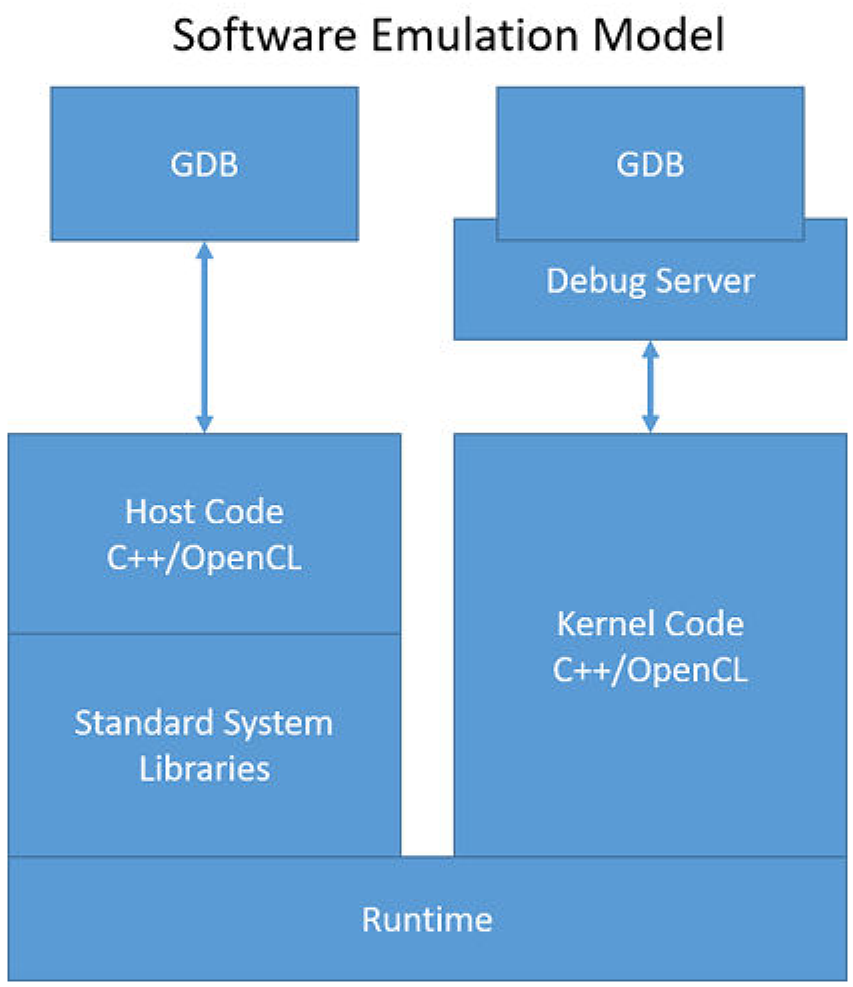
This three-tiered approach allows debugging the host and kernel code, and their interactions, at different levels of abstraction. Each of the execution models described below are supported through the SDAccel GUI as well as through a batch flow using basic compile time and run-time setup options.

Software Emulation

Purpose: Algorithm verification

Execution Model: During software emulation, all processes are running pure C/C++ models. OpenCL™ Kernel models are transformed to execute concurrently.

Figure 3: **Software Emulation**



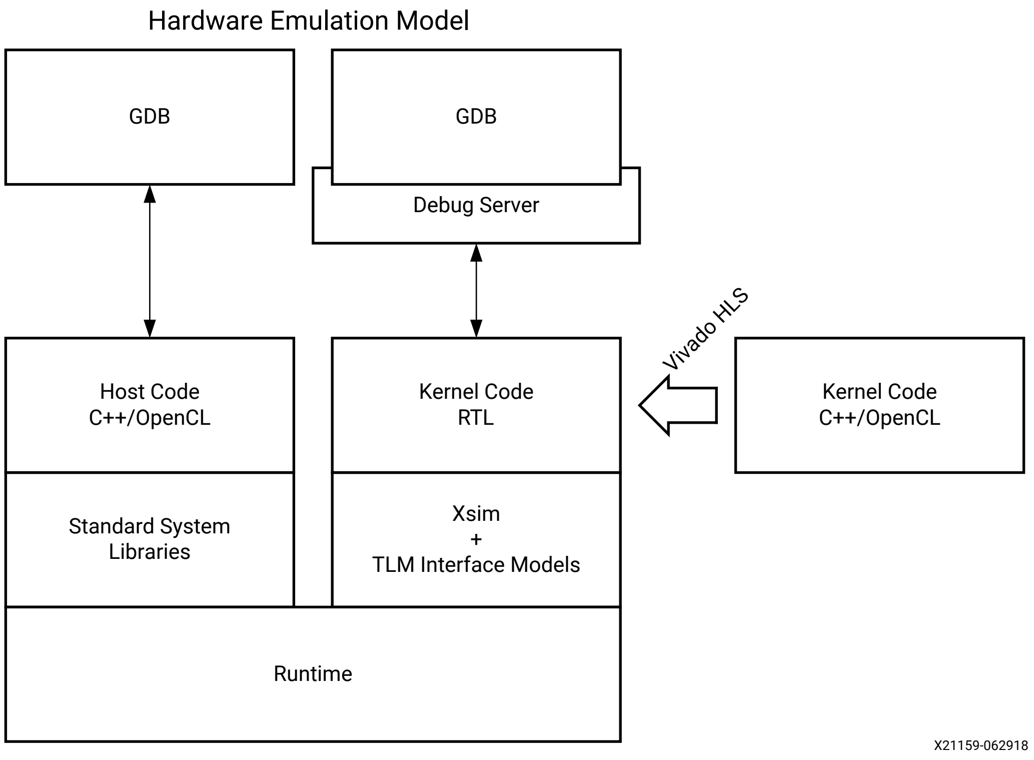
Verify both the host and kernel code are functionally correct by running software emulation. Because software emulation compiles and executes quickly, you should spend time here to iterate through the code until the host and kernel code function correctly. Both hardware emulation and hardware execution will take more time to compile and execute.

Hardware Emulation

Purpose: RTL debugging, finding protocol violations.

Execution Model: During hardware emulation the host code is executed concurrently with a simulation of the RTL model of the kernel, directly imported, or created through Vivado® HLS from the C/C++/OpenCL kernel code.

Figure 4: **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 (RTL) together with the host code C/OpenCL model. The hardware 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. Note that the interface between the models is represented by a transaction-level model (TLM) to limit impact of interface model on the overall execution time. The execution time for hardware emulation is longer than for software emulation.

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

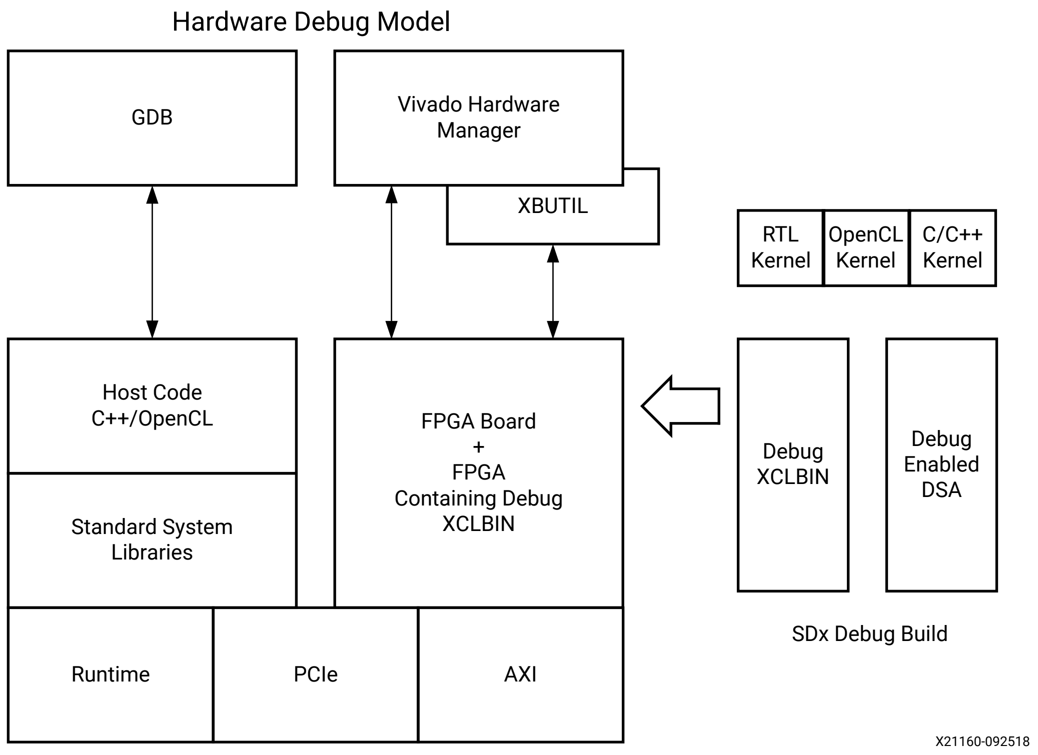
During the hardware emulation stage you can optionally modify the kernel code to improve performance. Iterate in hardware emulation until the functionality is correct and the estimated kernel performance is sufficient. See *SDAccel Environment Profiling and Optimization Guide* (UG1207) for more information.

Hardware Execution

Purpose: Final Verification of the complete system, finding protocol violations (hardware hangs), and debugging system performance.

Execution Model: During hardware execution, the actual hardware platform is used to execute the kernels. The difference between this debug configuration and the final compilation of the kernel code is the inclusion of special hardware logic in the platform, such as ILA and VIO debug cores, and AXI performance monitors for debug purposes.

Figure 5: **HW Execution**



At this stage, a system image (`xclbin`) is compiled, and executed on the actual hardware platform. Refer to the *SDAccel Environment User Guide* (UG1023) for more information on generating the `xclbin` file. At this point the kernels are confirmed to be executing correctly on the actual FPGA hardware. It is expected that your focus will shift from debugging to performance tuning. See the *SDAccel Environment Profiling and Optimization Guide* (UG1207).

Nevertheless, the hardware execution model might not be functional due to protocol issues, or issues with the hardware configuration. Towards that end, SDAccel provides specific hardware debug capabilities which include ChipScope debug cores (such as System ILAs), which can be viewed in Vivado Hardware Manager, with waveform analysis, kernel activity reports, and memory access analysis to localize these critical hardware issues.

IMPORTANT! *Debugging the kernel on the platform hardware requires additional logic to be incorporated into the overall hardware model. This means that if hardware debugging is enabled, there will be some impact on resource utilization of the FPGA, as well as some impact on the kernel performance.*

Chapter 2

SDAccel Debug Features

In this chapter, you will examine different features of the SDAccel™ environment supporting debugging efforts. This chapter introduces the debugging tools available to analyze the project and perform debugging. The next chapter illustrates debug techniques using the features described here.

SDAccel Software Debug

SDAccel™ supports typical software-like debugging for the host as well as kernel code. This flow is supported during software and hardware emulation and allows the use of break points and the analysis of variables as commonly done during software debugging.

***Note*:** The host code can still be debugged in this mode even when the actual hardware is executed.

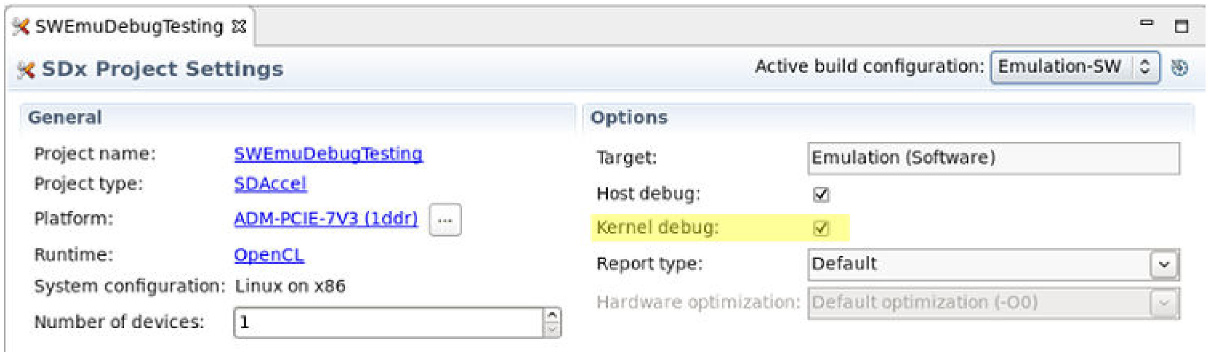
GUI Debug Flow

Running SDAccel™ in the GUI flow provides easy access to the debug capabilities. As shown in Debug Command Line Flow, setting up an executable for debugging requires many steps when performed manually. These steps are handled by the GUI when you use the GUI Debug Flow.

***Note*:** The SDAccel Debug Flow uses several different shells during debugging. This requires that the shell setup scripts such as `.bashrc` or `.cshrc` do not interfere with the SDAccel Setup such as the `LD_LIBRARY_PATH`.

Preparing the executable for debugging requires that you change the build configurations to enable the application of debug flags. You can set these options through the Project View. There are two check boxes provided in the **Options** section for the active build configuration. One enables host debug builds while the other enables debugging of the kernels. The checkboxes are named **Host debug** and **Kernel debug** respectively.

Figure 6: **Software Project Settings Options**


SWEmuDebugTesting

SDx Project Settings

Active build configuration: Emulation-SW

General

Project name: SWEmuDebugTesting
Project type: SDAccel
Platform: ADM-PCIE-7V3 (1ddr)
Runtime: OpenCL
System configuration: Linux on x86
Number of devices: 1

Options

Target: Emulation (Software)
Host debug: ✓
Kernel debug: ✓
Report type: Default
Hardware optimization: Default optimization (-O0)

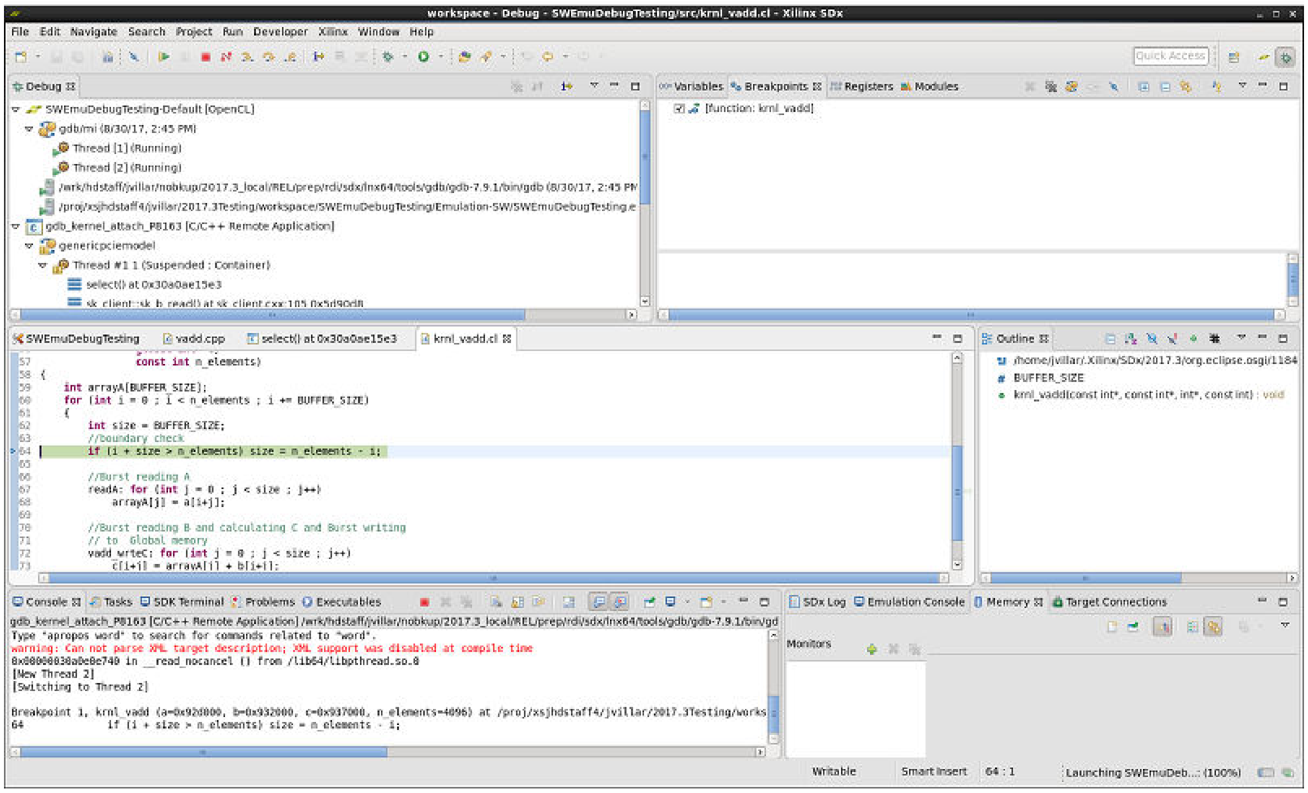
A more intuitive way to set these build options is through the context menu **Settings**. To do this, right-click on the build configuration in the Assistant view and select **Settings**. Alternatively, you can simply double-click on the build configuration. The same two checkboxes are presented. While you can enable host debug on all targets, kernel debug is only supported for software emulation and hardware emulation build targets.

This completes the setup, cleaning the build directory and rebuilding the application ensures that the project is ready for running in the GDB debug environment.

Running a GDB session from the GUI takes care of all the setup required. It automatically manages the environment setup for hardware or software emulation. It configures the SDAccel runtime to ensure debug support by the runtime environment, and manages the different consoles required for the execution of the kernel model, the host model, and the debug server.

As a result, when initiating the debug session, SDAccel asks to switch into the debug perspective, which presents several windows to manage the different debug consoles and source code windows.

Figure 7: **GDB Console**



```
            const int n_elements)
{
    int arrayA[BUFFER_SIZE];
    for (int i = 0 ; i < n_elements ; i += BUFFER_SIZE)
    {
        int size = BUFFER_SIZE;
        //boundary check
        if (i + size > n_elements) size = n_elements - i;

        //Burst reading A
        readA: for (int j = 0 ; j < size ; j++)
            arrayA[j] = a[i+j];

        //Burst reading B and calculating C and Burst writing
        // to  Global memory
        vadd_wrteC: for (int j = 0 ; j < size ; j++)
            c[i+j] = arrayA[j] + b[i+j];
```

```
gdb_kernel_attach_P8163 [C/C++ Remote Application] /wrk/hdstaff/jvillar/nobkup/2017.3_local/REL/prep/rdi/sdx/lnx64/tools/gdb/gdb-7.9.1/bin/gd
Type "apropos word" to search for commands related to "word".
warning: Can not parse XML target description; XML support was disabled at compile time
0x00000030a0e0e740 in __read_nocancel () from /lib64/libpthread.so.0
[New Thread 2]
[Switching to Thread 2]

Breakpoint 1, krnl_vadd (a=0x92d000, b=0x932000, c=0x937000, n_elements=4096) at /proj/xsjhdstaff4/jvillar/2017.3Testing/works
64              if (i + size > n_elements) size = n_elements - i;
```

After starting the application, by default the application is stopped right at the beginning of the `main` function body in the host code. As with any GDB graphical front end, you can now set breakpoints and inspect variables in the host code. The SDAccel environment enables in a completely transparent way the same capabilities to the developer for the accelerated kernel implementation.

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

See the Xilinx OpenCL Runtime GDB Extensions section for additional debug capabilities designed to allow easy access to common SDAccel™ implementation structures.

Debug Command Line Flow

The Command Line Debug Flow in the SDAccel™ environment provides tools to debug the host and kernel application running in all modes: software emulation, hardware emulation, or hardware.

Note: The host code can be debugged using this feature in the hardware execution mode only.

There are four steps to 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 modes, 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` These environment settings are used by the runtime library to correctly execute the desired emulation. This is required in addition to building the executable for the specific emulation flow.
`XILINX_SDX`	The path to the SDx™ install.
`XILINX_OPENCL`	The path to the SDx install (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`

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 | hw] ...
```

The `-t` (or `-target`) option is used to specify the compilation target as either software emulation (`sw_emu`), hardware emulation (`hw_emu`), or hardware execution (`hw`).

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 the `param:compiler.fsanitize` directive, 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 emulation run produces a debug log with emulation diagnostic messages such as `<project_dir>/Emulation-SW/<proj_name>-Default>/emulation_debug.log`.

Launching GDB Host Code Debug

You can launch GDB standalone to debug the host program if the code is built with debug information (built with the `-g` flag). This flow should also work while using a graphical front-end for GDB, such as the Data Display Debugger (DDD) available from GNU. Please note, to perform debugging on host and kernel code simultaneously follow the instructions outlined in Launching Host and Kernel Debug.

Following 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 to the correct mode.

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
   ```

   This informs the run time library that the kernel is debug enabled.

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 the SDx environment 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
     ```

Launching Host and Kernel Debug

In software emulation, to better mimic the hardware being emulated, 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 and the kernel code, the SDAccel environment provides a mechanism to attach to spawned kernels through the use of `sdx_server`.

You must start three different terminals in the command line flow:

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

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

2. In a second terminal run the host code in `xgdb` as described in Launching GDB Host Code Debug.

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.*

3. 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/unified/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.

TIP: *When debugging software/hardware emulation kernels in the SDAccel environment GUI, 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.*

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 `sdx_server` connection will drop.

For both 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.

Utilities for Hardware Debugging

In some cases, the normal GUI and Command Line debug features are limited in their ability to isolate an issue. This is especially true when the software / hardware appears to locked up and not making any progress (hang). These kinds of system issues are best analyzed with the help of the utilities mentioned in this section.

Using Linux "dmesg"

Well designed Linux kernels and modules report issues through the kernel ring buffer. This is also true for SDAccel™ modules that allow the user to debug on the lowest linux level the interaction with the accelerator board.

***Note*:** This utility intended for use in Hardware Debug only.

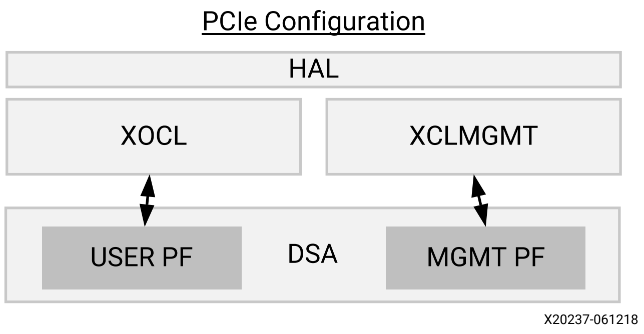
TIP: *In most cases, it is sufficient to work with the less verbose `xbutil` feature to localize a problem. Refer to the SDx Command and Utility Reference Guide (UG1279) for more information on the `xbutil` command.*

The `dmesg` utility is a linux tool that lets you read the kernel ring buffer. The kernel ring buffer holds kernel information messages in a circular buffer. A circular buffer of fixed size is used to limit the resource requirements by overwriting the oldest entry with the next incoming message.

In the SDAccel tool, the `xocl` module and `xclmgmt` driver modules write informational messages to the ring buffer. Thus, for an application hang or crash, or for that matter any unexpected behavior (like being unable to program the bitstream, etc.), the `dmesg` tool should be used to check the ring buffer.

The following image shows the layers of the software platform associated with the SDAccel board platform.

Figure 8: **Software Platform Layers**



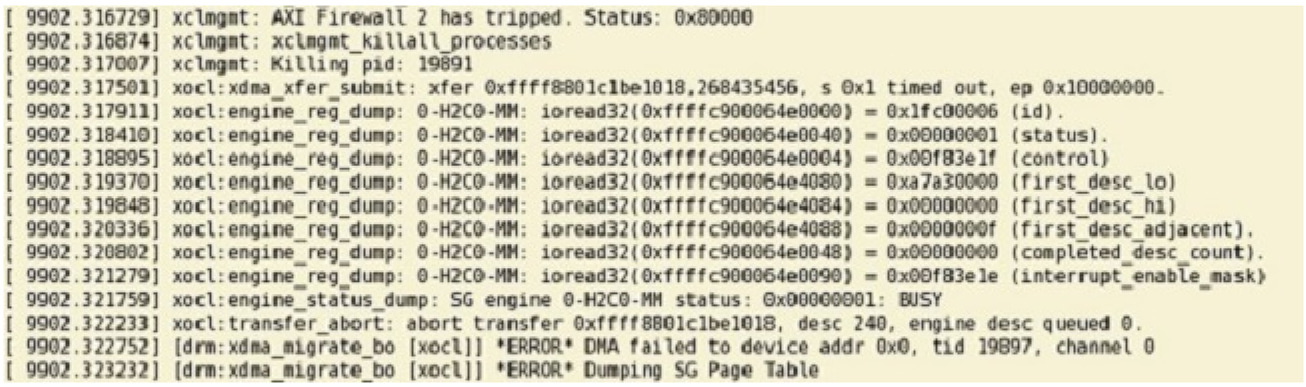
To review messages from the linux tool, you should first clear the ring buffer:

```
sudo dmesg -c
```

This will flush all messages from the ring buffer and make it easier to spot messages from the `xocl` and `xclmgmt`. After that, start your application and run `dmesg` in another terminal.

```
sudo dmesg
```

`dmesg` will print a record such as the following module reports:



```
[ 9902.316729] xclmgmt: AXI Firewall 2 has tripped. Status: 0x80000
[ 9902.316874] xclmgmt: xclmgmt_killall_processes
[ 9902.317007] xclmgmt: Killing pid: 19891
[ 9902.317501] xocl:xdma_xfer_submit: xfer 0xffff8801c1be1018,268435456, s 0x1 timed out, ep 0x10000000.
[ 9902.317911] xocl:engine_reg_dump: 0-H2C0-MM: ioread32(0xffffc900064e0000) = 0x1fc00006 (id).
[ 9902.318410] xocl:engine_reg_dump: 0-H2C0-MM: ioread32(0xffffc900064e0040) = 0x00000001 (status).
[ 9902.318895] xocl:engine_reg_dump: 0-H2C0-MM: ioread32(0xffffc900064e0004) = 0x00f83e1f (control)
[ 9902.319370] xocl:engine_reg_dump: 0-H2C0-MM: ioread32(0xffffc900064e4080) = 0xa7a30000 (first_desc_lo)
[ 9902.319848] xocl:engine_reg_dump: 0-H2C0-MM: ioread32(0xffffc900064e4084) = 0x00000000 (first_desc_hi)
[ 9902.320336] xocl:engine_reg_dump: 0-H2C0-MM: ioread32(0xffffc900064e4088) = 0x0000000f (first_desc_adjacent).
[ 9902.320802] xocl:engine_reg_dump: 0-H2C0-MM: ioread32(0xffffc900064e0048) = 0x00000000 (completed_desc_count).
[ 9902.321279] xocl:engine_reg_dump: 0-H2C0-MM: ioread32(0xffffc900064e0090) = 0x00f83e1e (interrupt_enable_mask)
[ 9902.321759] xocl:engine_status_dump: SG engine 0-H2C0-MM status: 0x00000001: BUSY
[ 9902.322233] xocl:transfer_abort: abort transfer 0xffff8801c1be1018, desc 240, engine desc queued 0.
[ 9902.322752] [drm:xdma_migrate_bo [xocl]] *ERROR* DMA failed to device addr 0x0, tid 19897, channel 0
[ 9902.323232] [drm:xdma_migrate_bo [xocl]] *ERROR* Dumping SG Page Table
```

In the example shown above, the AXI Firewall 2 has tripped which is better examined using `xbutil`. See the Debugging AXI Firewall Trips section for more information.

Using Xilinx "xbutil"

The Xilinx® Board Utility (`xbutil`) is a powerful standalone command line utility that can be used to debug lower level hardware/software interaction issues. A full description of this utility can be found in the *SDx Command and Utility Reference Guide* (UG1279).

With respect to debugging, the following `xbutil` options are of special interest:

- `query` - Provides an overall status of an SDAccel™ platform.
- `program` - Downloads a binary (`xclbin`) to the programmable region of the Xilinx device.
- `status` - Extracts the status of the SDx™ Performance Monitors (`spm`) and the Lightweight AXI Protocol Checkers (`lapc`).

Hardware Debugging Using ChipScope

After the final system image (`xclbin`) is generated and executed on the SDAccel™ platform, the entire system including the host application running on the CPU, and the accelerated kernels on the Xilinx® FPGA, can be confirmed to be executing correctly on the actual hardware. At this stage you can validate the functioning of the host code and kernel in the target hardware, and debug any issues found.

Some of the conditions that can be looked for or analyzed include the following conditions:

- Debugging system hangs that could be due to protocol violations.
  - These violations can take down the entire system.
  - These violations can cause the kernel to get invalid data or to hang.
  - It is hard to determine where or when these violations originated.
  - To debug this condition you would want an ILA triggered off of the AXI protocol checker, which would need to be configured on the SDAccel platform in use.
- Debugging problems inside the RTL kernel.
  - These are sometimes caused by the implementation: timing issues, race condition, bad design constraint.
  - Functional bug that hardware emulation did not show.
- Debugging Performance Problems.
  - For example, the frames/sec processing is not what you expect.
  - You can examine data beats and pipelining.
  - Using an ILA with trigger sequencer you could examine the burst size, pipelining, data width, and locate the bottleneck.

Checking FPGA Board for HW debug support

Supporting HW debugging requires the platform to support several IP components most notable the Debug Bridge. Please talk to your platform designer to determine if these components are included in the platform DSA.

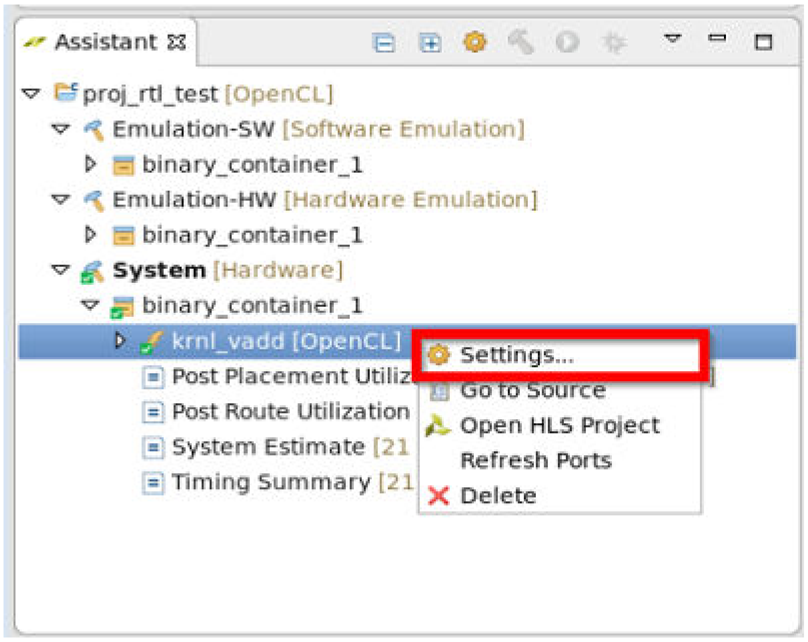
All Xilinx provided platforms fully support HW debugging:

- xilinx_kcu1500_dynamic_5_0
- xilinx_vcu1525_dynamic_5_1

Enabling ChipScope from the SDX GUI

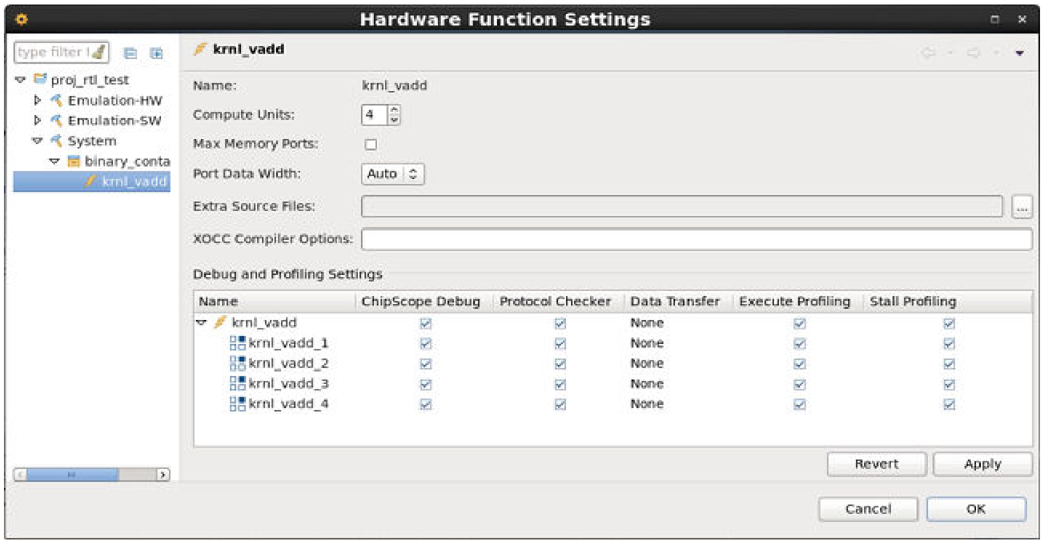
The SDx™ GUI provides options to enable ChipScope debug on all the interface ports of the compute units in the design. When enabling this option on a compute unit, the SDAccel™ compiler will add a System ILA debug core to monitor the interface ports of the compute unit. This will ensure that you can debug the interface signals on the SDAccel platform hardware as the kernel is running. You can access this through the **Settings** command by right-clicking on a kernel in the System build configuration in the **Assistant** window as shown below.

Figure 9: **SDx™ Assistant View**



This brings up the **Hardware Function Settings** dialog box as shown in the following figure. You can use the **Debug and Profiling Settings** table in this dialog box to enable the **ChipScope Debug** checkbox for specific compute units of the kernel, which will enable the monitoring of all the interfaces/ports on the compute unit.

Figure 10: **SDx Hardware Function Settings**



krnl_vadd

Name: krnl_vadd

Compute Units: 4

Max Memory Ports: ☐

Port Data Width: Auto

Extra Source Files:

XOCC Compiler Options:

Debug and Profiling Settings

Name	ChipScope Debug	Protocol Checker	Data Transfer	Execute Profiling	Stall Profiling
krnl_vadd	✓	✓	None	✓	✓
krnl_vadd_1	✓	✓	None	✓	✓
krnl_vadd_2	✓	✓	None	✓	✓
krnl_vadd_3	✓	✓	None	✓	✓
krnl_vadd_4	✓	✓	None	✓	✓

TIP: *Enabling the **ChipScope Debug** option on larger designs with multiple kernels and/or compute units can result in over-utilization of the FPGA device resources. Xilinx recommends using the `xocc --dk list_ports` option on the command line to determine the number and type of interfaces on the compute units. The methodology for using the `--dk` option documented in Command Line Flow is the recommended methodology if you know which ports need to be monitored for debug as the design runs in hardware.*

Command Line Flow

The System ILA debug core provides transaction level visibility into an accelerated kernel or function running on hardware. AXI traffic of interest can also be captured and viewed using the System ILA core.

The ILA core can be instantiated in the overall hardware of an existing RTL IP design, to enable debugging features within that design, or can be inserted automatically by the compiler. The XOCC compiler provides the `--dk` option to attach System ILA cores at the interfaces to the kernels for debugging and performance monitoring purposes.

The `--dk` option to enable ILA IP core insertion has the following syntax:

```
--dk <[chipscope|list_ports]<:compute_unit_name><:interface_name>>
```

In general, the `<interface_name>` is optional. If not specified, all ports are expected to be analyzed.

The `chipscope` option requires the explicit name of the compute unit to be provided for the `<compute_unit_name>` and `<interface_name>`.

The `list_ports` option generates a list of valid compute units and port combinations in the current design.

Note:

Multiple `--dk` option switches can be specified in a single command line to additively add interface monitoring capability.

Refer to the *SDx Command and Utility Reference Guide* (UG1279) for more information on any XOCC option.

Once the design is built, you can debug the design using the Hardware Manager as described in *Vivado Design Suite User Guide: Programming and Debugging* (UG908).

Debug Techniques

This section closely examines different styles of debugging techniques. The section classifies the different approaches into software based debugging techniques and hardware oriented techniques. In the software based approaches, you are not required to actually understand the ultimate mapping of the kernel code onto the FPGA. However, this concept can only be extended to a certain amount of detail, at which point the more detailed hardware-based analysis will be required.

The section is structured along the different debug stages of SDAccel™. It starts with Functional Verification during software emulation (a purely software based approach). Next is hardware emulation, where the kernel code is converted into actual hardware representation providing more details of the final implementation. Hardware debugging as well as software debugging concepts can be applied during debugging in the hardware emulation stage. The last stage is System Verification where the actual hardware is executed. In this stage, software debugging concepts can only be applied to the host while the kernel must deploy hardware debugging concepts.

Functional Verification (Software Emulation)

Functional Verification is the process during which the software representing the system is verified towards the ultimate implementation goal. The software developer is tasked at this point with ensuring that the software behaves as intended on the given data. This is a very common task during software development and many different concepts are available.

If your software does not perform as intended, you can use the debugger to identify the root cause of the issue, or if necessary, dump datapoints during software execution. This section introduces these concepts applied to an SDx™ project.

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, printing checkpoint values in the code is a tried and trusted way of identifying problems within the execution of a program. Because part of the algorithm is now running on an FPGA, even this debugging technique requires additional support.

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.

The following is an example of using `printf()` in the kernel, 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 what order the messages from each kernel will display on the host terminal. Please note, especially when running in hardware emulation and hardware, the hardware buffer size might limit `printf` output capturing.*

Please note this feature is only supported for OpenCL kernels in all development flows.

For C/C++ kernel models `printf()` is only supported during software emulation and should be excluded from the Vivado® HLS synthesis step. In this case, any `printf()` statement should be surrounded by the following compiler macros:

```
#ifndef __SYNTHESIS__
    printf("text");
#endif
```

GDB based Debugging

In this section we briefly show how host and kernel debugging can be performed with the help of GDB. This flow should be very familiar to software developers. Therefore, this section focuses on the extensions with respect to host code debugging capabilities specifically for FPGAs and the current status with respect to kernel based hardware emulation support.

Host Code Debugging

Except for the method of launching the debugging environment described in the previous chapter, there is no difference between the SDAccel™ host code debugging and the commonly used GDB application debugging flow and features.

After `gdb` is launched, you can step through the host code in GDB and examine the C/C++/ OpenCL™ objects to verify that their contents are as expected at any point in the code.

However, as stated in the introduction especially in the case of hardware emulation, it is common to look for issues regarding protocol synchronization between the host and the kernel. The SDAccel Environment provides special GDB extensions to examine the content of the OpenCL Runtime environment from the application host. These commands are described in more detail in the next section.

Xilinx OpenCL Runtime GDB Extensions

The Xilinx® OpenCL™ Runtime Debug Environment introduces new GDB commands, that give visibility from the host application into the OpenCL runtime library.

***Note*:** If you run GDB outside of the SDAccel™ Environment, these commands need to be enabled as described in Launching GDB Host Code Debug.

There are two kinds of commands which can be called from the `gdb` command line:

- 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 environment keeps track of all the OpenCL objects and automatically prints all valid queues and `cl_mem` objects if the argument is not specified. In addition, the commands do a proper validation of supplied command `queue`, `event`, and `cl_mem` arguments.

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

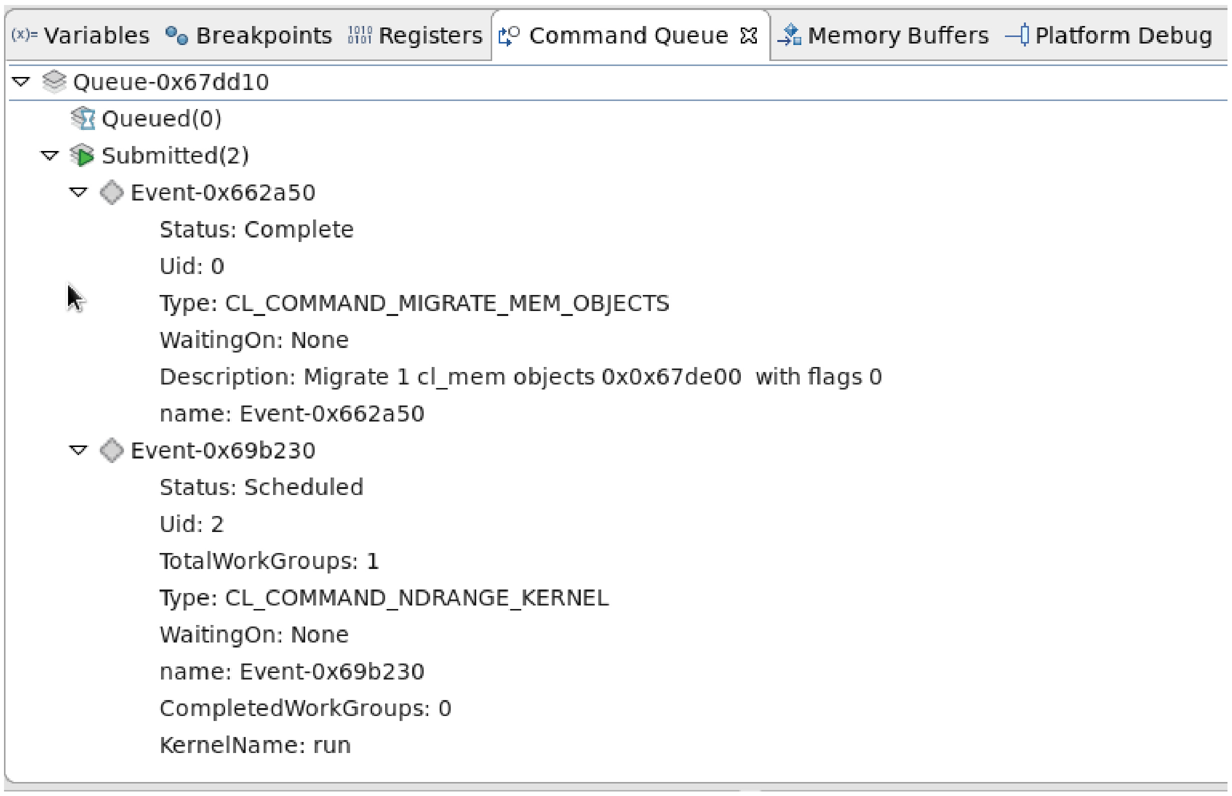
- Commands that give visibility into the IPs on the SDAccel platform. Note, this functionality is only available in the “System” flow (Hardware execution) and not in any of the emulation flows.

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

You can get help information about the commands by using `help <command>`.

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

The output of both of these commands is automatically tracked when debugging with the SDAccel GUI. In this case three tabs are provided next to the common tabs for Variables, Breakpoints, and Registers in the left upper corner of the debug perspective. These are labeled Command Queue, Memory Buffers, and Platform Debug, showing the output of `xprint queue`, `xprint mem`, and `xstatus` respectively.



```
Queue-0x67dd10
  Queued(0)
  Submitted(2)
    Event-0x662a50
        Status: Complete
        Uid: 0
        Type: CL_COMMAND_MIGRATE_MEM_OBJECTS
        WaitingOn: None
        Description: Migrate 1 cl_mem objects 0x0x67de00  with flags 0
        name: Event-0x662a50
    Event-0x69b230
        Status: Scheduled
        Uid: 2
        TotalWorkGroups: 1
        Type: CL_COMMAND_NDRANGE_KERNEL
        WaitingOn: None
        name: Event-0x69b230
        CompletedWorkGroups: 0
        KernelName: run
```

Note: The information presented in these views is only visible to the application developer while actually debugging the host code. This is the reason why this debug technique is also applicable when actual System execution (Hardware) is performed.

GDB Kernel based Debugging

GDB Kernel debugging is supported for the software emulation and hardware emulation flows. Once the GDB executable is connected to the kernel in the GUI or command-line flows, as described in Debug Command Line Flow, you can set breakpoints and query the content of variables in the kernel, similar to normal host code debugging.

This is fully supported in the software emulation flow since the kernel GDB processes simply attach to the spawned software processes.

However, during hardware emulation, the kernel source code is transformed into RTL, created by Vivado® HLS, and executed. As the RTL model is simulated, all transformations for performance optimization and concurrent hardware execution are applied. For that reason not all C/C++/OpenCL™ lines can be uniquely mapped to the RTL code, and only limited breakpoints are supported and at only specific variables can be queried. Today, the GDB tool therefore breaks on the next possible line based on requested breakpoint statements and clearly states if variables can not be queried based on the RTL transformations.

Debugging in Hardware Emulation

During hardware emulation, it is possible to deep dive into the implementation of the kernels. SDAccel™ allows you to perform typical hardware-like debugging in this mode as well as some software-like GDB based analysis on the hardware implementation.

GDB based Debugging

Debugging using a software based GDB flow is fully supported during hardware emulation. Except for the execution of the actual RTL code representing the kernel code there is no differences to the user as the GDB flow maps the RTL back into the source code description. This limits the breakpoint and observability of the variables in some cases as during the RTL generation (HLS) variables and loops might have been dissolved.

For a detailed description of the debug feature itself please see the description in the Chapter 2: SDAccel Debug Features chapter, and the extensions to GDB as presented in the GDB based Debugging section.

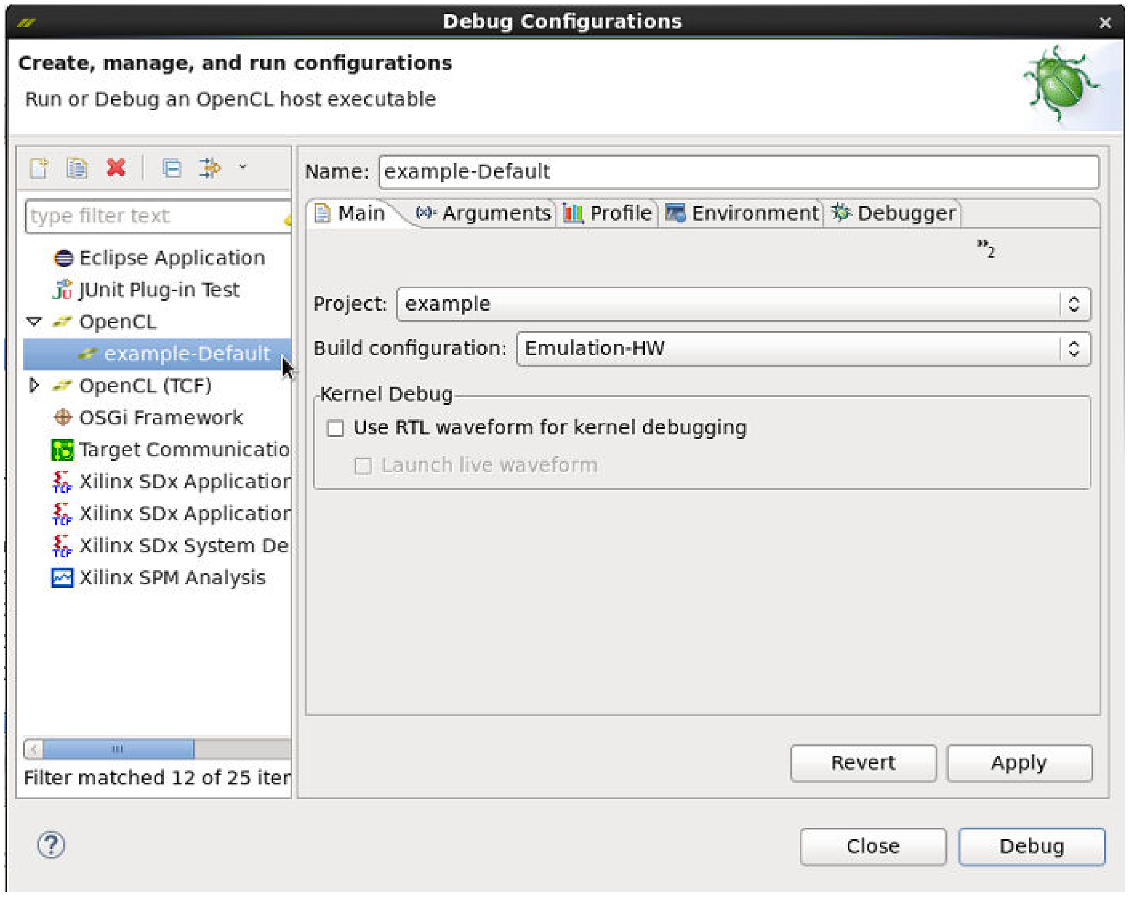
Waveform-Based Kernel Debugging

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

TIP: *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 launch configuration from the OpenCL list, as shown in the following figure.


Debug Configurations

Create, manage, and run configurations

Run or Debug an OpenCL host executable

Name: example-Default

Project: example

Build configuration: Emulation-HW

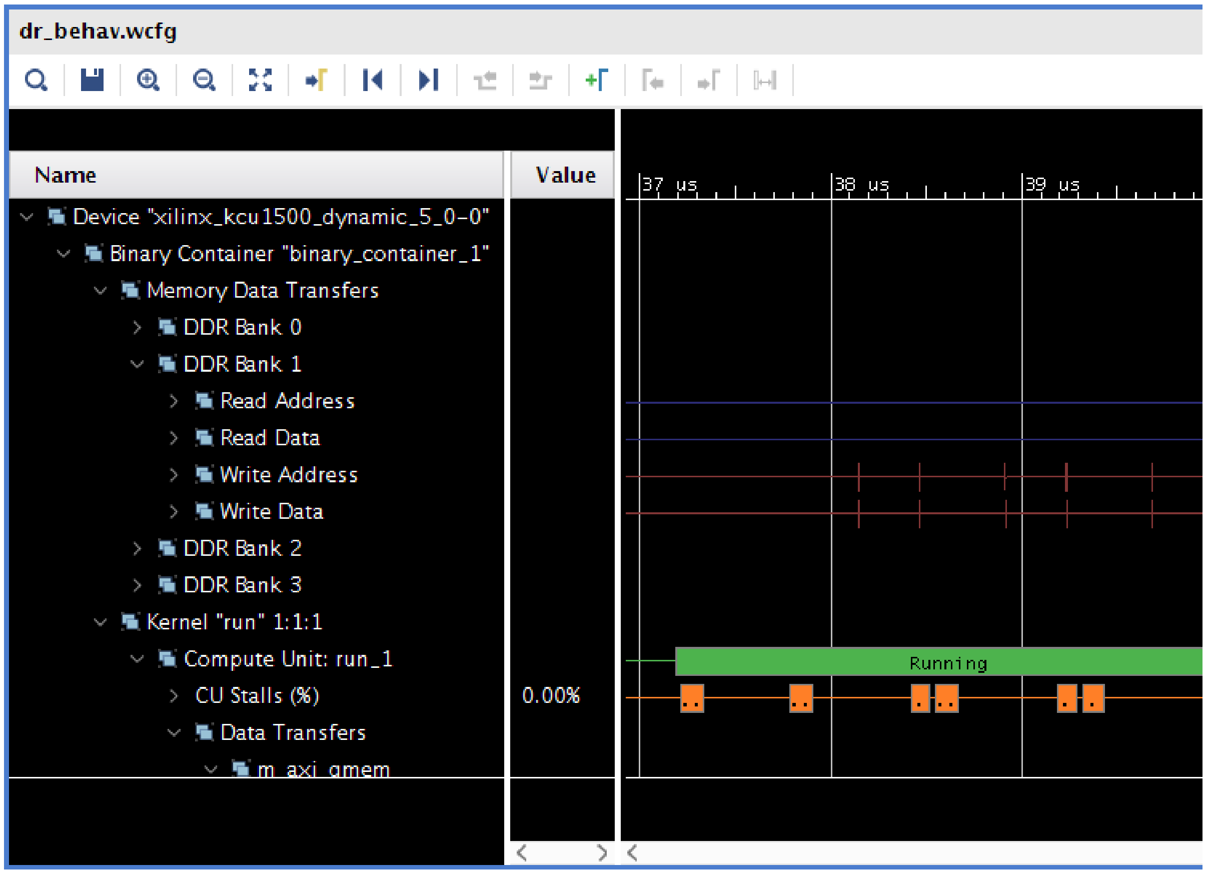
Kernel Debug

☐ Use RTL waveform for kernel debugging

☐ Launch live waveform

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. **Use RTL waveform for kernel debugging** will ensure that a simulation waveform database is generated, while the **Launch live waveform** option will actually spawn the Waveform viewer during the actual simulation allowing the user full control to the simulation engines and waveform display.

If the live waveform viewer is activated, the waveform viewer automatically opens when running the executable. By default, the waveform viewer shows all interface signals and the following debug hierarchy:



```
Device "xilinx_kcu1500_dynamic_5_0-0"
  Binary Container "binary_container_1"
    Memory Data Transfers
      DDR Bank 0
      DDR Bank 1
        Read Address
        Read Data
        Write Address
        Write Data
      DDR Bank 2
      DDR Bank 3
    Kernel "run" 1:1:1
      Compute Unit: run_1
        CU Stalls (%)          0.00%
        Data Transfers
          m_axi_gmem
```

- Memory Data Transfers: Shows data transfers from all compute units funnel through these interfaces.

TIP: *These interfaces could be a different bit width from the compute units. If so, then the burst lengths would be different. For example, a burst of sixteen 32-bit words at a compute unit would be a burst of one 512-bit word at the OCL master.*

- `Kernel` *`<kernel name> <workgroup size>`* `Compute Unit` *`<CU name>`*
  - **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.

- **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>`
    - **Dataflow/Pipeline Activity**: This section shows the function-level loop dataflow/ pipeline signals for a CU.
    - **Function Stalls**: This section lists the three stall signals within this function.
    - **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`.

TIP: *As with any waveform debugger, additional debug data of internal signals can be added by selecting the instance of interest from the scope menu and the signals of interest from the object menu. Similarly, debug controls such as HDL breakpoints, as well as HDL code lookup and waveform markers are supported. Refer to the Vivado Design Suite User Guide: Logic Simulation (UG900) for more information on working with the waveform viewer.*

The waveform debugging process can also be enabled through the XOCC command line. 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
data_transfer_trace=fine
```

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.

TIP: *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.*

System Verification, Hardware Hangs

Application Hang

This section discusses debugging issues related to the interaction of the host code and the accelerated kernels. Problems with these interactions manifest themselves as issues such as machine hangs or application hangs. Although, the GDB debug environment might help with isolating the errors in some cases (`xprint`), such as hangs associated with specific kernels, these issues are best debugged using the `dmesg` and `xbutil` commands as shown here. The commands and all their associated options are described in Utilities for Hardware Debugging.

In the following, we examine three of the more common issues encountered:

- AXI firewall trips
- Kernel hangs due to AXI violations
- Host application hangs accessing DDR memory

If this process of hardware debugging does not resolve the problem, it is necessary to perform hardware debugging using ChipScope. See the Debugging with ChipScope section.

AXI Firewall Trips

The AXI firewall should prevent host hangs. This is why Xilinx recommends the AXI Protocol Firewall IP to be included in SDAccel Platforms. When the firewall trips one of the first checks you perform should be to see if the host code and kernels are setup to utilize the same memory banks. The following is one of the simplest methods to perform this check:

1. Use `xbutil` to program the FPGA:

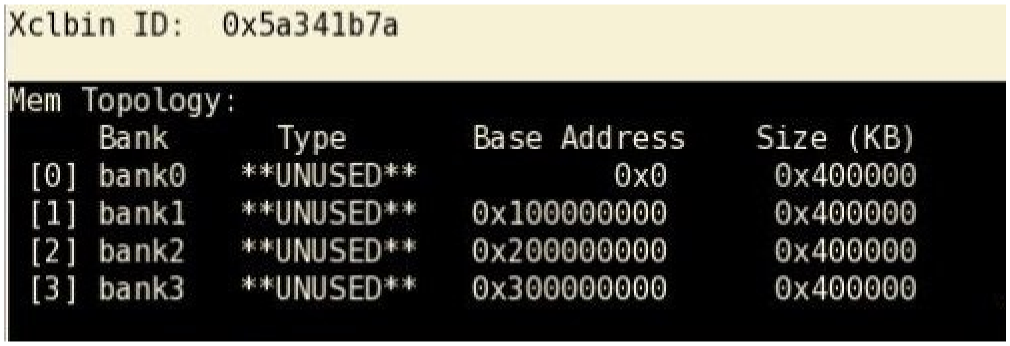
   ```
   xbutil program -p <xclbin>
   ```

2. Run the `xbutil` query option to check memory topology:

   ```
   xbutil query
   ```

   In the following example, there is no memory bank associated with the kernels:

Figure 11: **Results of xbutil query**



```
Xclbin ID:  0x5a341b7a

Mem Topology:
     Bank       Type         Base Address     Size (KB)
  [0] bank0   **UNUSED**             0x0      0x400000
  [1] bank1   **UNUSED**     0x100000000      0x400000
  [2] bank2   **UNUSED**     0x200000000      0x400000
  [3] bank3   **UNUSED**     0x300000000      0x400000
```

3. If the host code expects any DDR banks to be used, this report would definitely indicate an issue. In this case, it is necessary to check kernel and host code expectations. If the host code is using the Xilinx OpenCL™ extensions, it is necessary to check what DDR banks should be used by the kernel. These should match the `xocc -sp` arguments provided.

Kernel Hangs due to AXI Violations

It is possible for the kernels to hang due to bad AXI transactions between the kernels and the memory controller. To debug these issues, it is required to instrument the kernels.

1. The SDAccel environment provides two options for instrumentation to be applied during XOCC linking (`-l`). Please note, both of these add hardware to your implementation, and based on utilization it might be necessary to actually limit instrumentation.

   a. Add Lightweight AXI Protocol Checkers (`lapc`). These protocol checkers are added using the `--dk` option. The following syntax is used:

   ```
   --dk <[protocol|list_ports]<:compute_unit_name><:interface_name>>
   ```

   In general, the `<interface_name>` is optional. If not specified, all ports are expected to be analyzed.

   The `protocol` option is used to define protocol checkers to be inserted. This option can accept a special keyword, `all`, for `<compute_unit_name>` and/or `<interface_name>`.

   The `list_ports` option generates a list of valid compute units and port combinations in the current design.

Note: Multiple `--dk` option switches can be specified in a single command line to additively add interface monitoring capability.

b. Adding SDx™ Performance Monitors (`spm`) enables the listing of detailed communication statistics (counters). Although, this is most useful for performance analysis, it provides insight during debugging on pending port activities. The Performance Monitors are added using the `profile_kernel` option. The basic syntax for `profile_kernel` option is:

```
--profile_kernel data:<krnl_name|all>:<cu_name|all>:<intrfc_name|
all>:<counters|all>
```

Three fields are required to determine the precise interface to which the performance monitor is applied to. However in case resource utilization isn't an issue, the keyword `all` enables the user to apply the monitoring to all existing kernels, compute units, and interfaces with a single option. Otherwise, the user can specify the `kernel_name`, `cu_name`, and `interface_name` explicitly to limit instrumentation.

The last option, `<counters|all>`, allows you to restrict the information gathering to just `counters` for large designs, while `all` (default) includes the collection of actual trace information.

Note: Multiple `--profile_kernel` option switches can be specified in a single command line to additively add performance monitoring capability.

```
--profile_kernel data:kernel1:cu1:m_axi_gmem0
--profile_kernel data:kernel1:cu1:m_axi_gmem1
--profile_kernel data:kernel2:cu2:m_axi_gmem
```

2. Once the application is rebuilt, rerun the host application using the `xclbin` with the added SPM IP and LAPC IP.

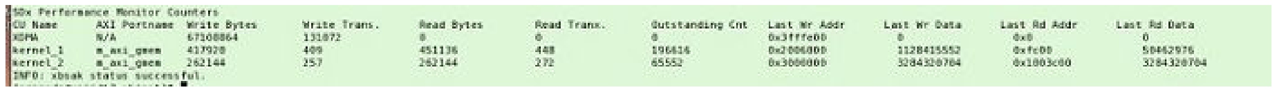
3. When the application hangs you can use `xbutil status` to check for any errors or anomalies.

4. Check the SPM output:

   - Run `xbutil status --spm` a couple of times to check if any counters are moving. If they are moving then the kernels are active.

TIP: *Testing SPM output is also supported through GDB debugging using the command extension `xstatus spm`.*

   - If the counters are stagnate, then "outstanding counts" greater than zero could mean some AXI transactions are hung.



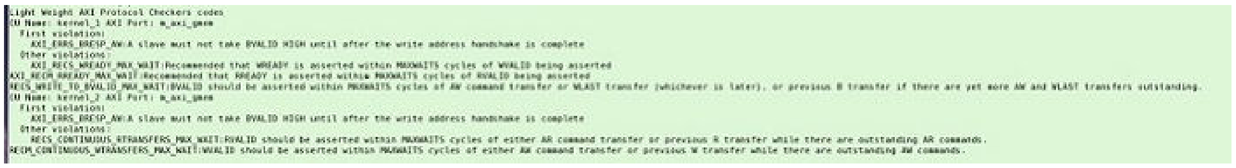
```
SDx Performance Monitor Counters
CU Name     AXI Portname  Write Bytes    Write Trans.   Read Bytes    Read Tranx.    Outstanding Cnt  Last Wr Addr   Last Wr Data   Last Rd Addr   Last Rd Data
XDMA        N/A           67108864       131072         0             0              0                0x3fffe00      0              0x0            0
kernel_1    m_axi_gmem    417920         409            451136        448            196616           0x2000000      1128415552     0xfc00         50462976
kernel_2    m_axi_gmem    262144         257            262144        272            65552            0x3000000      3284320704     0x1003c00      3284320704
INFO: xbsak status successful.
```

5. Check the LAPC output:

   - Run `xbutil status --lapc` to check if there are any AXI violations.

TIP: *Testing LAPC output is also supported through GDB debugging using the command extension `xstatus lapc`.*

- If there are any AXI violations, it implies that there are problems in the kernel implementation.



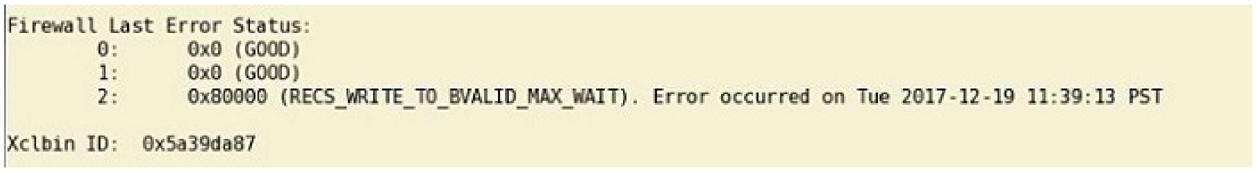
```
Light Weight AXI Protocol Checkers codes
CU Name: kernel_1 AXI Port: m_axi_gmem
  First violation:
    AXI_ERRS_BRESP_AW:A slave must not take BVALID HIGH until after the write address handshake is complete
  Other violations:
    AXI_RECS_WREADY_MAX_WAIT:Recommended that WREADY is asserted within MAXWAITS cycles of WVALID being asserted
AXI_RECM_RREADY_MAX_WAIT:Recommended that RREADY is asserted within MAXWAITS cycles of RVALID being asserted
RECS_WRITE_TO_BVALID_MAX_WAIT:BVALID should be asserted within MAXWAITS cycles of AW command transfer or WLAST transfer (whichever is later), or previous B transfer if there are yet more AW and WLAST transfers outstanding.
CU Name: kernel_2 AXI Port: m_axi_gmem
  First violation:
    AXI_ERRS_BRESP_AW:A slave must not take BVALID HIGH until after the write address handshake is complete
  Other violations:
    RECS_CONTINUOUS_RTRANSFERS_MAX_WAIT:RVALID should be asserted within MAXWAITS cycles of either AR command transfer or previous R transfer while there are outstanding AR commands.
RECM_CONTINUOUS_WTRANSFERS_MAX_WAIT:WVALID should be asserted within MAXWAITS cycles of either AW command transfer or previous W transfer while there are outstanding AW commands.
```

Host Application Hangs Accessing DDR Memory

It is also possible that the application hangs, or crashes, if the DMA transfers initiated from the host code don't complete. This does not mean that the host code is wrong. It could also be that the kernels issued illegal transactions and locked up the AXI.

1. If the platform has an AXI firewall, such as in the SDAccel™ platforms, it will trip. The driver will issue a SIGBUS, kill the application, and reset the device. You can check this by running `xbutil query`. The following figure shows such an error:

Figure 12: **Firewall Status**



```
Firewall Last Error Status:
        0:      0x0 (GOOD)
        1:      0x0 (GOOD)
        2:      0x80000 (RECS_WRITE_TO_BVALID_MAX_WAIT). Error occurred on Tue 2017-12-19 11:39:13 PST

Xclbin ID:  0x5a39da87
```

TIP: *If the firewall did not trip, then the Linux tool,* `dmesg`*, could provide additional insight. Please see Using Linux "dmesg" for more information.*

2. When you know that the firewall has tripped, it is important to determine the cause for the DMA timeout. The issue could be an illegal DMA transfer, or the kernels misbehaved. However, a side-effect of the AXI firewall tripping is that "health-check" functionality in the driver would reset the board after killing the application. Any information on the device that could help with debugging the root cause is lost. To debug this problem, you can disable the health-check thread in `xclmgmt` kernel module to capture the error. This uses common Unix kernel tools in the following sequence:

   a. `sudo modinfo xclmgmt`: This command lists the current configuration of the module and will list if the health_check parameter is on or off. It also returns the path to the `xclmgmt` module.

   b. `sudo rmmod xclmgmt`: This will remove and therefore disable the `xclmgmt` kernel module.

   c. `sudo insmod <path to module>/xclmgmt.ko health_check=0`: This will reinstall the `xclmgmt` kernel module with the health_check disabled.

TIP: *The path to this module is reported in the output of the call to* `modinfo`*.*

3. With the health check disabled, re-run the application. You can utilize the kernel instrumentation to isolate this issue as previously described in the section on Kernel Hangs.

Debugging with ChipScope

You can use the ChipScope debugging environment and the Vivado® Hardware Manager to help you debug your host application and kernels quickly and more effectively. In order to do this, at least one of the following must be true:

- Your SDAccel™ application project has been instrumented with debug cores, using the `--dk` compiler switch (as described in Hardware Debugging Using ChipScope).
- The RTL kernels used in your project must have been instantiated with debug cores (as described in Adding Debug IP to RTL Kernels).

These tools enable a wide range of capabilities from logic to system level debug while your kernel is running in hardware.

The following steps are required to run XVC and HW servers, host applications, and finally trigger and arm the debug cores in Vivado Hardware Manager.

1. Add debug IP to your RTL kernels. This applies only to RTL kernels. See Adding Debug IP to RTL Kernels.
2. Instrument the host application to pause at appropriate point in the host execution where you want to debug. See Debugging through the Host Application.
3. Set up the environment for hardware debug. You can do this manually or using a script that automates this for you. The following steps are described in Manual Setup for Hardware Debug and Automated Setup for Hardware Debug:
   a. Run the required XVC and HW servers.
   b. Execute the host application and pause at the appropriate point in the host execution to enable setup of ILA triggers.
   c. Open Vivado Hardware Manager and connect to XVC server.
   d. Set up ILA trigger conditions for the design.
   e. Continue with host application.
   f. Inspect results in the Vivado Hardware Manager.
   g. Rerun iteratively from Step b (above) as required.

Adding Debug IP to RTL Kernels

IMPORTANT! *This debug technique requires familiarity with the Vivado® Design Suite, and RTL design.*

You need to instantiate debug cores like the Integrated Logic Analyzer (ILA) and Virtual Input/ Output (VIO) in your RTL kernel code to debug the kernel logic. From within the Vivado Design Suite, edit the RTL kernel to instantiate an ILA IP customization, or a VIO IP, into the RTL code, similar to using any other IP in Vivado IDE. Refer to the *Vivado Design Suite User Guide: Programming and Debugging* (UG908) to learn more about using the ILA or other debug cores in the RTL Insertion flow and to learn about using the HDL generate statement technique to enable/disable debug core generation.

TIP: *The best time to add debug cores to your RTL kernel is when you create it. Refer to "Debugging" in the UltraFast Design Methodology Guide for the Vivado Design Suite (UG949) for more information.*

You can also add the ILA debug core using a Tcl script from within an open Vivado project as shown in the following code example:

```
create_ip -name ila -vendor xilinx.com -library ip -version 6.2 -
module_name ila_0
set_property -dict [list CONFIG.C_PROBE6_WIDTH {32} CONFIG.C_PROBE3_WIDTH
{64} \
CONFIG.C_NUM_OF_PROBES {7} CONFIG.C_EN_STRG_QUAL {1}
CONFIG.C_INPUT_PIPE_STAGES {2} \
CONFIG.C_ADV_TRIGGER {true} CONFIG.ALL_PROBE_SAME_MU_CNT {4}
CONFIG.C_PROBE6_MU_CNT {4} \
CONFIG.C_PROBE5_MU_CNT {4} CONFIG.C_PROBE4_MU_CNT {4}
CONFIG.C_PROBE3_MU_CNT {4} \
CONFIG.C_PROBE2_MU_CNT {4} CONFIG.C_PROBE1_MU_CNT {4}
CONFIG.C_PROBE0_MU_CNT {4}] [get_ips ila_0]
```

The following is an example of an ILA debug core instantiated into the RTL kernel source file of the RTL Kernel Debug example design on GitHub. The ILA monitors the output of the combinatorial adder as specified in the `src/hdl/krnl_vadd_rtl_int.sv` file.

```
    // ILA monitoring combinatorial adder
    ila_0 i_ila_0 (
        .clk(ap_clk),              // input wire        clk
        .probe0(areset),           // input wire [0:0]  probe0
        .probe1(rd_fifo_tvalid_n), // input wire [0:0]  probe1
        .probe2(rd_fifo_tready),   // input wire [0:0]  probe2
        .probe3(rd_fifo_tdata),    // input wire [63:0] probe3
        .probe4(adder_tvalid),     // input wire [0:0]  probe4
        .probe5(adder_tready_n),   // input wire [0:0]  probe5
        .probe6(adder_tdata)       // input wire [31:0] probe6
    );
```

After the RTL kernel has been instrumented for debug with the appropriate debug cores, you can analyze the hardware in the ChipScope tool as described in Debugging with ChipScope.

Debugging through the Host Application

To debug the host application, working with the kernel code running on the SDAccel™ platform, the application host code must be modified to ensure that you can set up the ILA trigger conditions after the kernel has been programmed into the device, but before actually starting the kernel.

Pausing a C++ Host Application

The following code example is from the `src/host.cpp` code from the RTL Kernel example on GitHub:

```
....
    std::string binaryFile = xcl::find_binary_file(device_name,"vadd");

    cl::Program::Binaries bins = xcl::import_binary_file(binaryFile);
    devices.resize(1);
    cl::Program program(context, devices, bins);
    cl::Kernel krnl_vadd(program,"krnl_vadd_rtl");

    //if (interactive == true)
        //wait_for_enter("\nPress ENTER to continue after setting up ILA 
trigger...");

    //Allocate Buffer in Global Memory
    std::vector<cl::Memory> inBufVec, outBufVec;
    cl::Buffer buffer_r1(context,CL_MEM_USE_HOST_PTR | CL_MEM_READ_ONLY,
            vector_size_bytes, source_input1.data());
    ...

    //Copy input data to device global memory
    q.enqueueMigrateMemObjects(inBufVec,0/* 0 means from host*/);

    //Set the Kernel Arguments
    ...

    //Launch the Kernel
    q.enqueueTask(krnl_vadd);
```

Notice the addition, or presence of the conditional `if (interactive)` test, and the use of the `wait_for_enter` function to pause the host application to give the ILA time to set up the required triggers and prepare to capture data from the kernel. After the Vivado® Hardware Manager is set up and configured properly, you can press `Enter` to continue running the host application.

Pausing the Host Application Using GDB

Alternatively, instead of making changes to the host application to pause before a kernel execution, you can run a GDB session from the SDx™ GUI. You can then set a breakpoint prior to the kernel execution in the host application. When the breakpoint is reached you can setup the debug ILA triggers in Vivado® Hardware Manager and arm the trigger, and then resume the kernel execution in GDB.

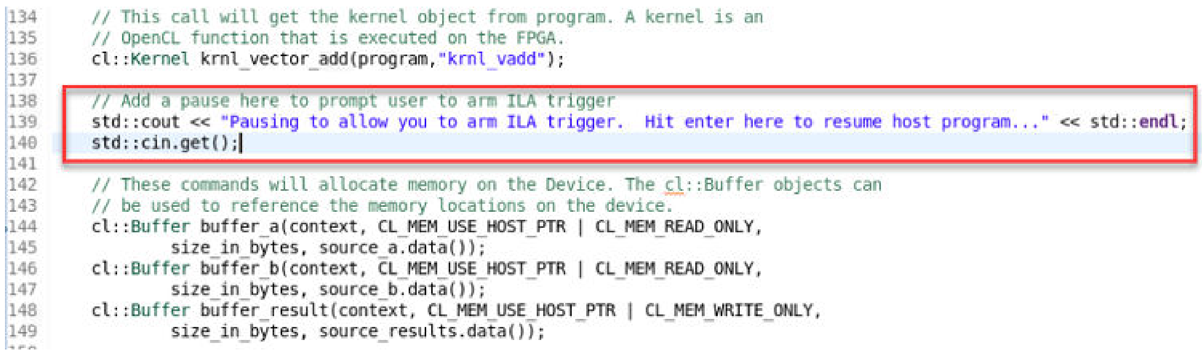
Automated Setup for Hardware Debug

1. Set up your SDx™ environment by sourcing the appropriate `settings64.sh/.csh` file found in your SDx install area.
2. Start `xvc_pcie` and `hw_server` apps using the `sdx_debug_hw` script (note that the `/dev/xvc_*` character device will differ depending on the platform and SDx release – for the vcu1525 5.1 DSA in 2018.2 SDx release, the file is `/dev/xvc_pub.m1025`).

```
sdx_debug_hw --xvc_pcie /dev/xvc_pub.m1025 --hw_server
launching xvc_pcie...
xvc_pcie -d /dev/xvc_pub.m1025 -s TCP::10200
launching hw_server...
hw_server -sTCP::3121
```

3. In the SDx IDE, modify the host code to include a "pause" statement AFTER the kernel has been created/downloaded and BEFORE the kernel execution is started, then recompile the host program.

   - For C++ host code, add a "pause" after the function that wraps the `clCreateKernel()` function call. The following snippet is from the Vector Add template design's C++ host code:

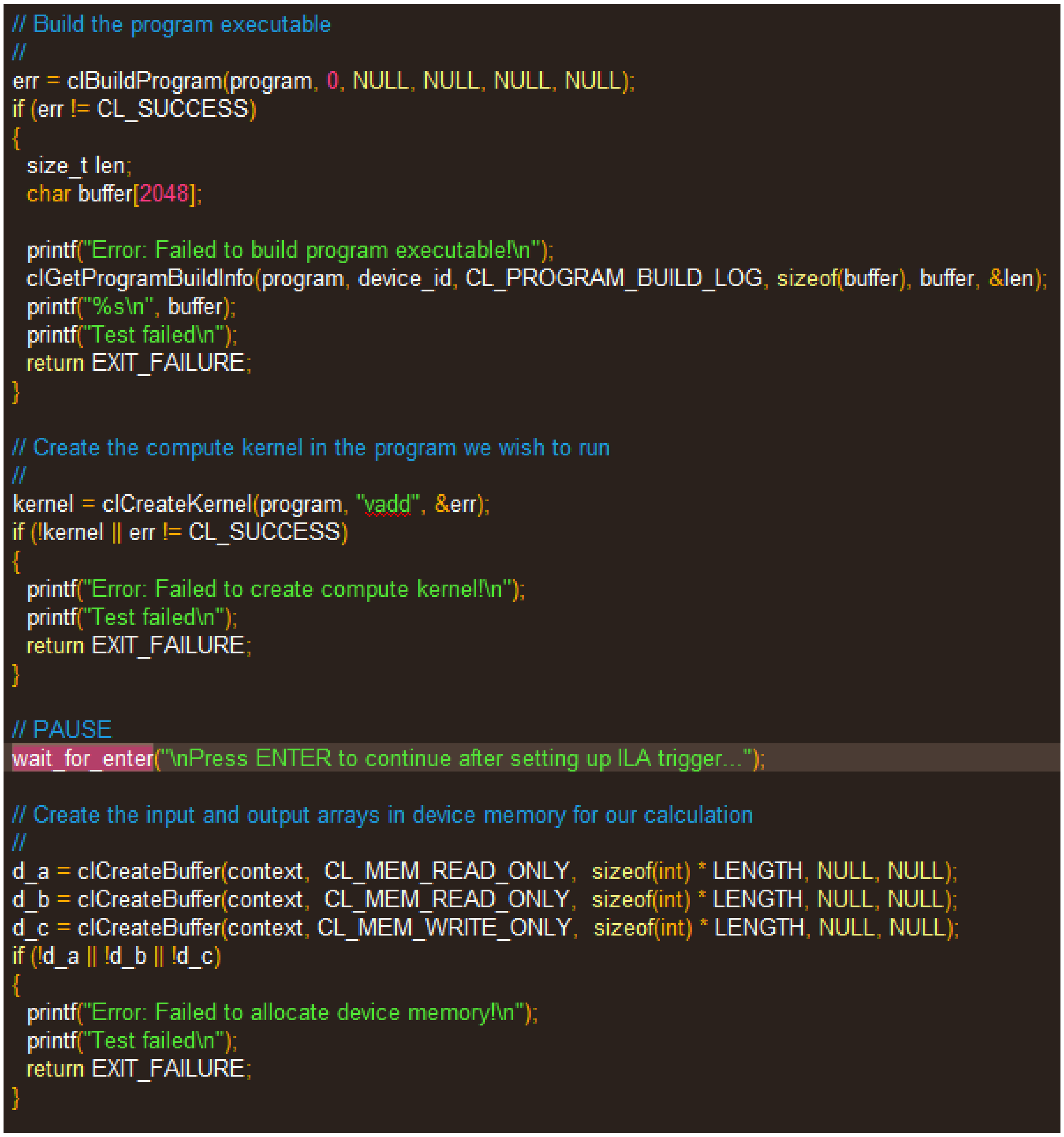


```
    // This call will get the kernel object from program. A kernel is an
    // OpenCL function that is executed on the FPGA.
    cl::Kernel krnl_vector_add(program,"krnl_vadd");

    // Add a pause here to prompt user to arm ILA trigger
    std::cout << "Pausing to allow you to arm ILA trigger.  Hit enter here to resume host program..." << std::endl;
    std::cin.get();

    // These commands will allocate memory on the Device. The cl::Buffer objects can
    // be used to reference the memory locations on the device.
    cl::Buffer buffer_a(context, CL_MEM_USE_HOST_PTR | CL_MEM_READ_ONLY,
            size_in_bytes, source_a.data());
    cl::Buffer buffer_b(context, CL_MEM_USE_HOST_PTR | CL_MEM_READ_ONLY,
            size_in_bytes, source_b.data());
    cl::Buffer buffer_result(context, CL_MEM_USE_HOST_PTR | CL_MEM_WRITE_ONLY,
            size_in_bytes, source_results.data());
```

   - For C-language host code, add a "pause" after the "clCreateKernel()" function call:



```
// Build the program executable
//
err = clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
if (err != CL_SUCCESS)
{
  size_t len;
  char buffer[2048];

  printf("Error: Failed to build program executable!\n");
  clGetProgramBuildInfo(program, device_id, CL_PROGRAM_BUILD_LOG, sizeof(buffer), buffer, &len);
  printf("%s\n", buffer);
  printf("Test failed\n");
  return EXIT_FAILURE;
}

// Create the compute kernel in the program we wish to run
//
kernel = clCreateKernel(program, "vadd", &err);
if (!kernel || err != CL_SUCCESS)
{
  printf("Error: Failed to create compute kernel!\n");
  printf("Test failed\n");
  return EXIT_FAILURE;
}

// PAUSE
wait_for_enter("\nPress ENTER to continue after setting up ILA trigger...");

// Create the input and output arrays in device memory for our calculation
//
d_a = clCreateBuffer(context,  CL_MEM_READ_ONLY,  sizeof(int) * LENGTH, NULL, NULL);
d_b = clCreateBuffer(context,  CL_MEM_READ_ONLY,  sizeof(int) * LENGTH, NULL, NULL);
d_c = clCreateBuffer(context, CL_MEM_WRITE_ONLY,  sizeof(int) * LENGTH, NULL, NULL);
if (!d_a || !d_b || !d_c)
{
  printf("Error: Failed to allocate device memory!\n");
  printf("Test failed\n");
  return EXIT_FAILURE;
}
```

4. Run your modified host program.

```
vadd_test.exe ./binary_container_1.xclbin
Loading: './binary_container_1.xclbin'
Pausing to allow you to arm ILA trigger. Hit enter here to resume host
program...
```

5. Launch Vivado® Design Suite using the `sdx_debug_hw` script located in your SDAccel installation directory.

```
> sdx_debug_hw --vivado --host xcoltlab40 --ltx_file ../workspace/
vadd_test/System/pfm_top_wrapper.ltx
```

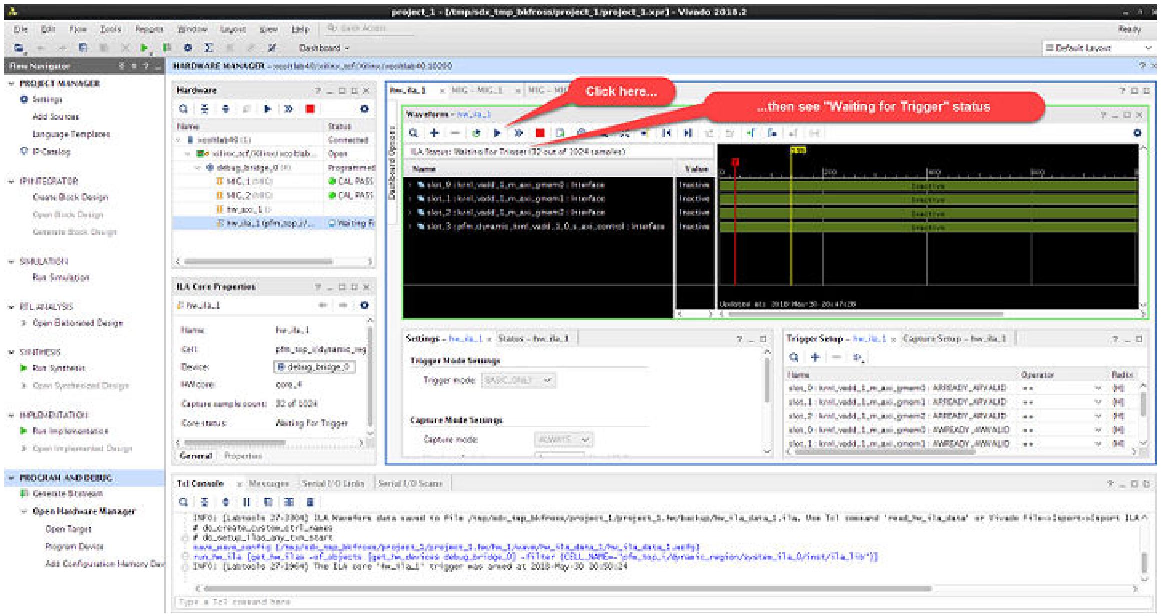
The command window displays the following:

```
launching vivado... ['vivado', '-source', 'sdx_hw_debug.tcl', '-
tclargs', '/tmp/sdx_tmp/project_1/project_1.xpr', 'workspace/vadd_test/
System/pfm_top_wrapper.ltx', 'xcoltlab40', '10200', '3121']

****** Vivado v2018.2 (64-bit)
  **** SW Build 2245749 on Wed May 30 12:36:19 MDT 2018
  **** IP Build 2245576 on Wed May 30 15:12:50 MDT 2018
    ** Copyright 1986-2018 Xilinx, Inc. All Rights Reserved.

start_gui
```

6. In Vivado Design Suite, run the ILA trigger.

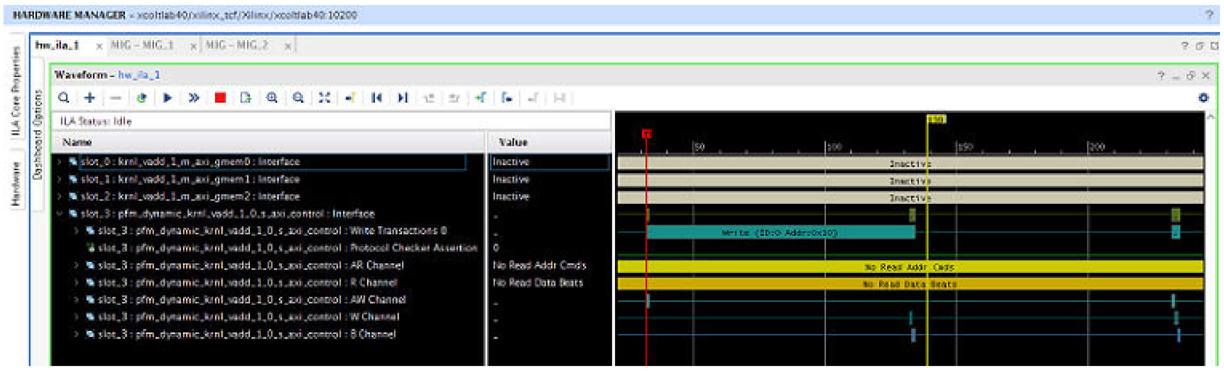


7. Press **Enter** to "un-pause" host program.

```
vadd_test.exe ./binary_container_1.xclbin
Loading: './binary_container_1.xclbin'
Pausing to allow you to arm ILA trigger.  Hit enter here to resume host
program...

TEST PASSED
```

8. In the Vivado Design Suite, see the interface transactions on the kernel compute unit's slave control interface in the Waveform view.


HARDWARE MANAGER - xcoltlab40/xilinx_tcf/Xilinx/xcoltlab40:10200
hw_ila_1
MIG - MIG_1
MIG - MIG_2
Waveform - hw_ila_1
ILA Status: Idle
Name
Value
slot_0 : krnl_vadd_1_m_axi_gmem0 : Interface
slot_1 : krnl_vadd_1_m_axi_gmem1 : Interface
slot_2 : krnl_vadd_1_m_axi_gmem2 : Interface
slot_3 : pfm_dynamic_krnl_vadd_1_0_s_axi_control : Interface
slot_3 : pfm_dynamic_krnl_vadd_1_0_s_axi_control : Write Transactions 0
slot_3 : pfm_dynamic_krnl_vadd_1_0_s_axi_control : Protocol Checker Assertion
slot_3 : pfm_dynamic_krnl_vadd_1_0_s_axi_control : AR Channel
slot_3 : pfm_dynamic_krnl_vadd_1_0_s_axi_control : R Channel
slot_3 : pfm_dynamic_krnl_vadd_1_0_s_axi_control : AW Channel
slot_3 : pfm_dynamic_krnl_vadd_1_0_s_axi_control : W Channel
slot_3 : pfm_dynamic_krnl_vadd_1_0_s_axi_control : B Channel
Inactive
Inactive
Inactive
0
No Read Addr Cmds
No Read Data Beats
50
100
150
200
Write (ID:0 Addr:0x10)
No Read Addr Cmds
No Read Data Beats
Hardware
ILA Core Properties
Dashboard Options

Manual Setup for Hardware Debug

Starting Debug Servers on Amazon F1 Instance

Instructions to start the debug servers on an Amazon F1 instance can be found here: https://github.com/aws/aws-fpga/blob/master/hdk/docs/Virtual_JTAG_XVC.md

Starting Debug Servers on Nimbix and Other Cloud Platforms

There are two steps required to start the debug servers prior to debugging the design in Vivado® Hardware Manager.

1. Source the SDx™ `settings64.csh` or `settings64.sh` environment setup script and run the `xvc_pcie` driver. The filename passed to `xvc_pcie` must match the character driver file installed with the kernel device driver.

```
>xvc_pcie -d /dev/xvc_pub.m1025
```

2. Start the XVC server on port 10201 and the `hw_server` on port 3121.

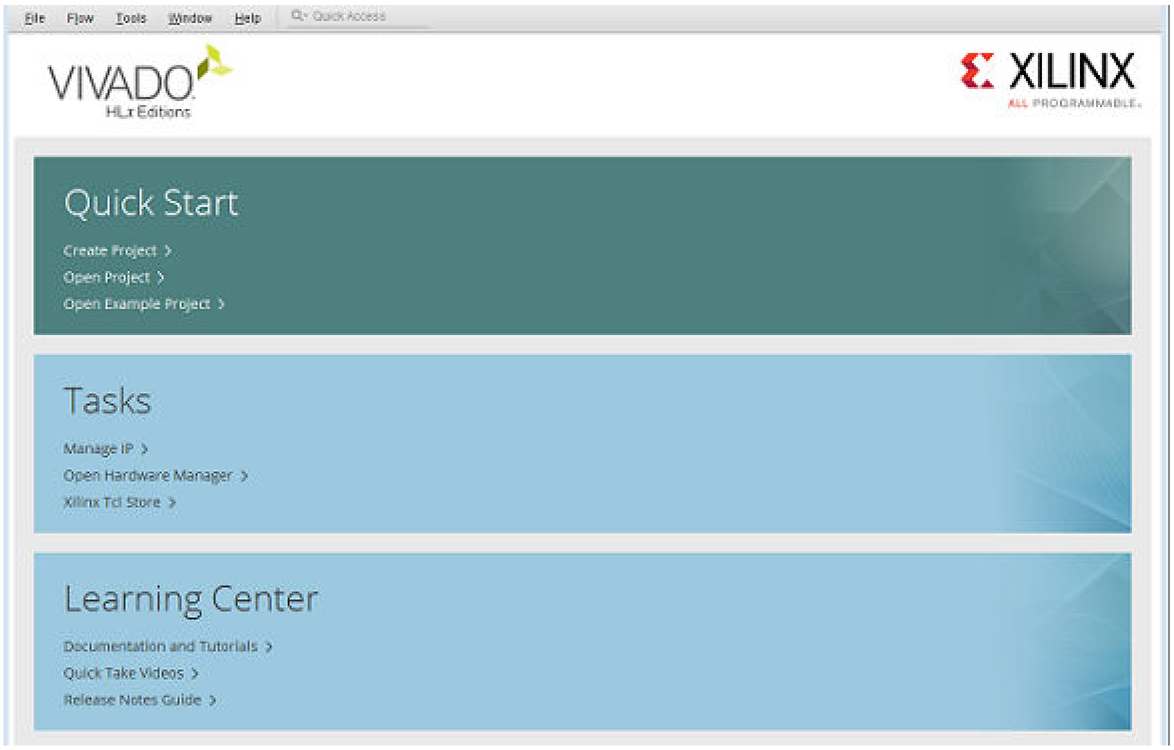
```
>hw_server "set auto-open-servers xilinx-xvc:localhost:10201" -e "set
always-open-jtag 1"
```

Debugging Designs using Vivado Hardware Manager

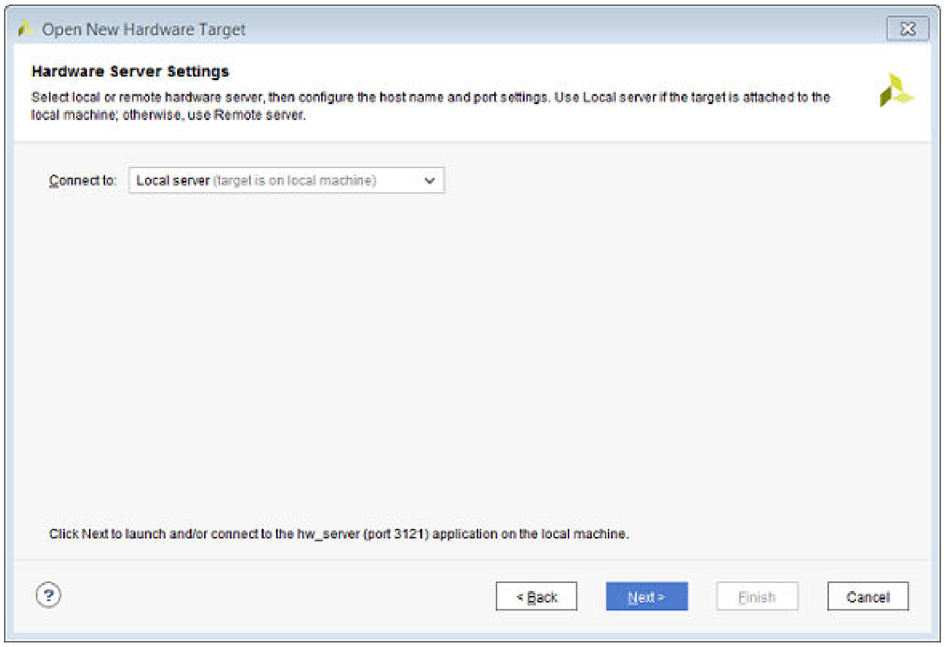
Traditionally, a physical JTAG connection is used to debug FPGAs. The SDAccel platforms have leveraged XVC for a debug flow that enables debug in the cloud. To take advantage of this capability, SDAccel enables running the XVC Server. The XVC Server is an implementation of Xilinx Virtual Cable (XVC) protocol, which allows the Vivado® Design Suite to connect to a local or remote target FPGA for debug, using standard Xilinx debug cores like the Integrated Logic Analyzer IP (ILA), or the Virtual Input/Output IP (VIO), and others.

The Vivado Hardware Manager (Vivado Design Suite or Vivado Lab Edition) can be running on the target instance or it can be running remotely on a different host. The TCP port on which the XVC Server is listening must be accessible to the host running Vivado Hardware Manager. To connect the Vivado Hardware Manager to XVC server on the target, the following steps should be followed on the machine hosting the Vivado tools:

1. Launch the Vivado Lab Edition, or the full Vivado Design Suite.
2. Select **Open Hardware Manager** from the **Tasks** menu, as shown in the following figure.



3. Connect to the Vivado tools `hw_server`, specifying a local or remote connection, and the **Host name** and **Port**, as shown below.



Open New Hardware Target

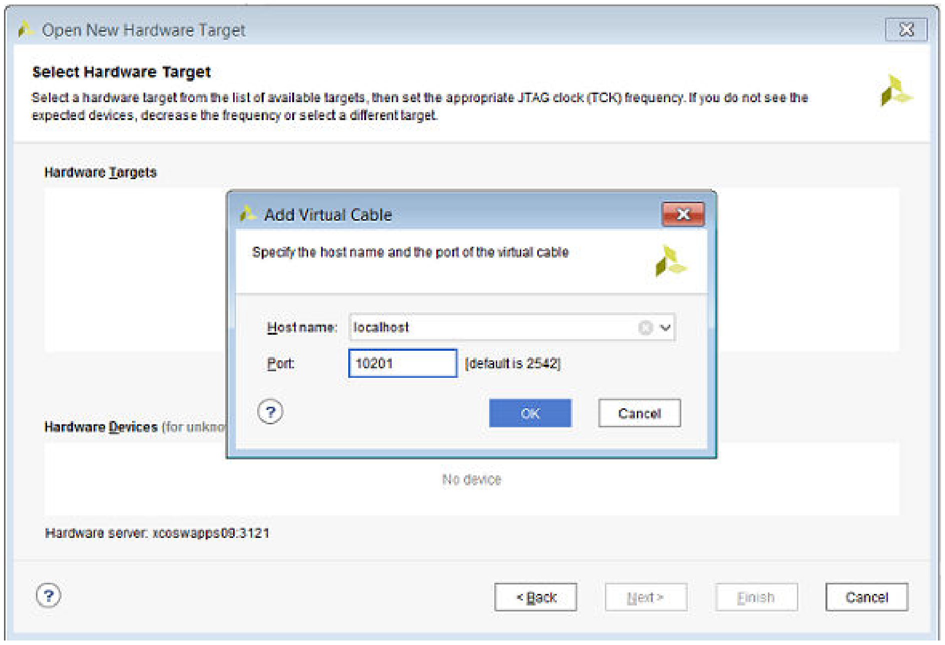
Hardware Server Settings

Select local or remote hardware server, then configure the host name and port settings. Use Local server if the target is attached to the local machine; otherwise, use Remote server.

Connect to: Local server (target is on local machine)

Click Next to launch and/or connect to the hw_server (port 3121) application on the local machine.

4. Connect to the target instance Virtual JTAG XVC server.



Open New Hardware Target

Select Hardware Target

Select a hardware target from the list of available targets, then set the appropriate JTAG clock (TCK) frequency. If you do not see the expected devices, decrease the frequency or select a different target.

Hardware Targets

Add Virtual Cable

Specify the host name and the port of the virtual cable

Host name: localhost

Port: 10201 [default is 2542]

Hardware Devices (for unkno

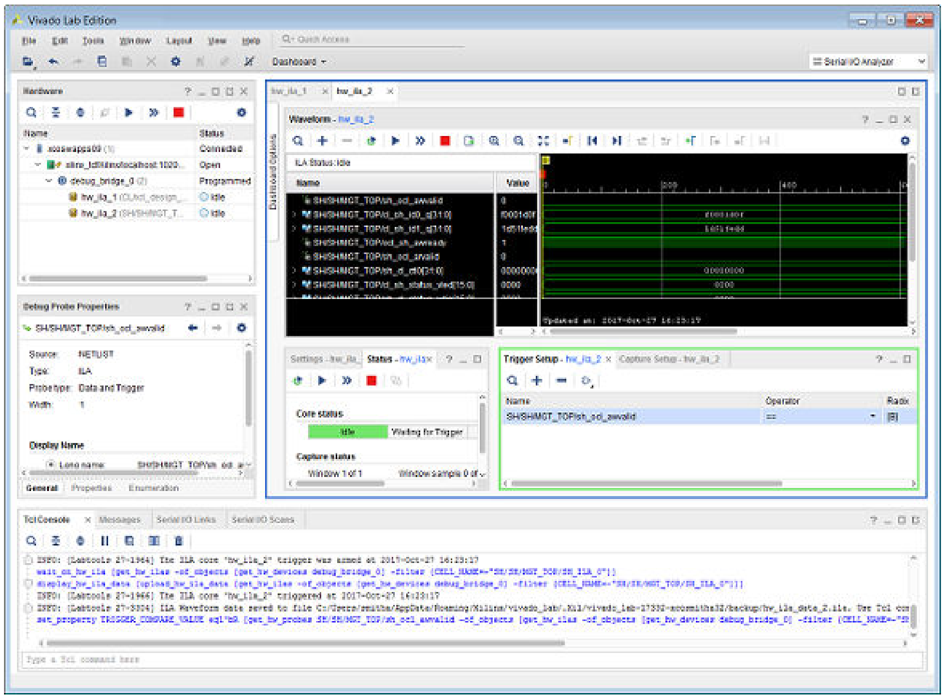
No device

Hardware server: xcoswapps09:3121

5. Select the debug bridge instance from the **Hardware** window of the ChipScope tool.
6. In the **Hardware Device Properties** window select the appropriate probes file for your design by clicking the icon next to the Probes file entry, selecting the file, and clicking **OK**. This will refresh the hardware device and it should now show the debug cores present in your design.

TIP: *The probes file (`.ltx`) is written out during the implementation of the kernel by the Vivado tool, if the kernel has debug cores as specified in Hardware Debugging Using ChipScope.*

7. The ChipScope tool can now be used to debug the kernels running on the SDAccel platform. Refer to the *Vivado Design Suite User Guide: Programming and Debugging* (UG908) for more information on working with the tool.



```
INFO: [Labtools 27-1964] The ILA core 'hw_ila_2' trigger was armed at 2017-Oct-27 16:23:17
wait_on_hw_ila [get_hw_ilas -of_objects [get_hw_devices debug_bridge_0] -filter {CELL_NAME=~"SH/SH/MGT_TOP/SH_ILA_0"}]
display_hw_ila_data [upload_hw_ila_data [get_hw_ilas -of_objects [get_hw_devices debug_bridge_0] -filter {CELL_NAME=~"SH/SH/MGT_TOP/SH_ILA_0"}]]
INFO: [Labtools 27-1966] The ILA core 'hw_ila_2' triggered at 2017-Oct-27 16:23:17
INFO: [Labtools 27-3304] ILA Waveform data saved to file C:/Users/smitha/AppData/Roaming/Xilinx/vivado_lab/.Xil/vivado_lab-27332-xcosmitha32/backup/hw_ila_data_2.ila. Use Tcl com
set_property TRIGGER_COMPARE_VALUE eq1'bX [get_hw_probes SH/SH/MGT_TOP/sh_ocl_awvalid -of_objects [get_hw_ilas -of_objects [get_hw_devices debug_bridge_0] -filter {CELL_NAME=~"SH
```

Appendix A

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

Xilinx Resources

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

Documentation Navigator and Design Hubs

Xilinx® Documentation Navigator (DocNav) provides access to Xilinx documents, videos, and support resources, which you can filter and search to find information. DocNav is installed with the SDSoC™ and SDAccel™ development environments. To open it:

- On Windows, select **Start** → **All Programs** → **Xilinx Design Tools** → **DocNav**.
- At the Linux command prompt, enter `docnav`.

Xilinx Design Hubs provide links to documentation organized by design tasks and other topics, which you can use to learn key concepts and address frequently asked questions. To access the Design Hubs:

- In DocNav, click the **Design Hubs View** tab.
- On the Xilinx website, see the Design Hubs page.

***Note*:** For more information on DocNav, see the Documentation Navigator page on the Xilinx website.

References

1. *SDAccel Environments Release Notes, Installation, and Licensing Guide* (UG1238)
2. *SDAccel Environment User Guide* (UG1023)
3. *SDAccel Environment Profiling and Optimization Guide* (UG1207)
4. *SDAccel Environment Tutorial: Introduction* (UG1021)
5. SDAccel Development Environment web page
6. Vivado® Design Suite Documentation
7. *Vivado Design Suite User Guide: Designing IP Subsystems using IP Integrator* (UG994)
8. *Vivado Design Suite: Creating and Packaging Custom IP* (UG1118)
9. *Vivado Design Suite User Guide: Partial Reconfiguration* (UG909)
10. *Vivado Design Suite User Guide: High-Level Synthesis* (UG902)
11. *UltraFast Design Methodology Guide for the Vivado Design Suite* (UG949)
12. *Vivado Design Suite Properties Reference Guide* (UG912)
13. Khronos Group web page: Documentation for the OpenCL standard
14. Xilinx Virtex UltraScale+ FPGA VCU1525 Acceleration Development Kit
15. Xilinx Kintex UltraScale FPGA KCU1500 Acceleration Development Kit

Please Read: Important Legal Notices

The information disclosed to you hereunder (the "Materials") is provided solely for the selection and use of Xilinx products. To the maximum extent permitted by applicable law: (1) Materials are made available "AS IS" and with all faults, Xilinx hereby DISCLAIMS ALL WARRANTIES AND CONDITIONS, EXPRESS, IMPLIED, OR STATUTORY, INCLUDING BUT NOT LIMITED TO WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, OR FITNESS FOR ANY

PARTICULAR PURPOSE; and (2) Xilinx shall not be liable (whether in contract or tort, including negligence, or under any other theory of liability) for any loss or damage of any kind or nature related to, arising under, or in connection with, the Materials (including your use of the Materials), including for any direct, indirect, special, incidental, or consequential loss or damage (including loss of data, profits, goodwill, or any type of loss or damage suffered as a result of any action brought by a third party) even if such damage or loss was reasonably foreseeable or Xilinx had been advised of the possibility of the same. Xilinx assumes no obligation to correct any errors contained in the Materials or to notify you of updates to the Materials or to product specifications. You may not reproduce, modify, distribute, or publicly display the Materials without prior written consent. Certain products are subject to the terms and conditions of Xilinx's limited warranty, please refer to Xilinx's Terms of Sale which can be viewed at https://www.xilinx.com/legal.htm#tos; IP cores may be subject to warranty and support terms contained in a license issued to you by Xilinx. Xilinx products are not designed or intended to be fail-safe or for use in any application requiring fail-safe performance; you assume sole risk and liability for use of Xilinx products in such critical applications, please refer to Xilinx's Terms of Sale which can be viewed at https://www.xilinx.com/legal.htm#tos.

AUTOMOTIVE APPLICATIONS DISCLAIMER

AUTOMOTIVE PRODUCTS (IDENTIFIED AS "XA" IN THE PART NUMBER) ARE NOT WARRANTED FOR USE IN THE DEPLOYMENT OF AIRBAGS OR FOR USE IN APPLICATIONS THAT AFFECT CONTROL OF A VEHICLE ("SAFETY APPLICATION") UNLESS THERE IS A SAFETY CONCEPT OR REDUNDANCY FEATURE CONSISTENT WITH THE ISO 26262 AUTOMOTIVE SAFETY STANDARD ("SAFETY DESIGN"). CUSTOMER SHALL, PRIOR TO USING OR DISTRIBUTING ANY SYSTEMS THAT INCORPORATE PRODUCTS, THOROUGHLY TEST SUCH SYSTEMS FOR SAFETY PURPOSES. USE OF PRODUCTS IN A SAFETY APPLICATION WITHOUT A SAFETY DESIGN IS FULLY AT THE RISK OF CUSTOMER, SUBJECT ONLY TO APPLICABLE LAWS AND REGULATIONS GOVERNING LIMITATIONS ON PRODUCT LIABILITY.

Copyright

© Copyright 2018 Xilinx, Inc. Xilinx, the Xilinx logo, Artix, ISE, Kintex, Spartan, Virtex, Vivado, Zynq, and other designated brands included herein are trademarks of Xilinx in the United States and other countries. OpenCL and the OpenCL logo are trademarks of Apple Inc. used by permission by Khronos. All other trademarks are the property of their respective owners.