SDAccel Environment User Guide

UG1023 (v2018.2) June 20, 2018

ATTENTION! SDAccel Development Environment 2018.2 XDF users: Click here to view the 2018.2.xdf version of this guide.

Revision History

The following table shows the revision history for this document.

Section	Revision Summary
06/20/2018	
Installing, Prgamming, and Debugging Boards	Added appendix.
06/06/2018 Version 2018.2	
	This document has gone through a significant update with much of the content changed.
Chapter 3: Creating an SDAccel Project	Added discussion of the Assistant view.
Chapter 4: Programming for SDAccel	Added content on Coding the Host Application.
Chapter 5: Building the System	Added details of the Host Build and Kernel Build processes.
Chapter 7: Debugging Applications and Kernels	This chapter is now an overview of the debugging process, which is now fully presented in a separate User Guide.
Chapter 8: Command Line Flow	Added a discussion of the commands for compiling and linking the host code and kernel code.
Chapter 9: Creating RTL Kernels	Significant updates to this content.
Appendix B: Directory Structure	Describes the directory structure of application projects.
Appendix C: SLR Assignments for Kernels	Discusses the use of `--xp` to control kernel placement.
Appendix E: JTAG Fallback for Private Debug Network	Discusses remote debug issues for RTL kernels.
04/04/2018 Version 2018.1	
xbsak Commands and Options	Changed `--apm` command option to `--spm`.
Compiling Your OpenCL Kernel Using the Xilinx OpenCL Compiler	Changed `--pk` command option to `--profile_kernel`.

Table of Contents

SDAccel Introduction and Overview

The SDAccel™ environment provides a framework for developing and delivering FPGA accelerated data center applications using standard programming languages. The SDAccel environment includes a familiar software development flow with an Eclipse-based integrated development environment (IDE), and an architecturally optimizing compiler that makes efficient use of FPGA resources. Developers of accelerated applications will use a familiar software programming work flow to take advantage of FPGA acceleration with little or no prior FPGA or hardware design experience. Acceleration kernel developers can use a hardware-centric approach working through the HLS compiler with standard programming languages to produce a heterogeneous application with both software and hardware components. The software component, or application, is developed in C/C++ with OpenCL™ API calls; the hardware component, or kernel, is developed in C/C++, OpenCL, or RTL. The SDAccel environment accommodates various methodologies, allowing developers to start from either the software component or the hardware component.

Xilinx FPGAs offer many advantages over traditional CPU/GPU acceleration, including a custom architecture capable of implementing any function that can run on a processor, resulting in better performance at lower power dissipation. To realize the advantages of software acceleration on a Xilinx device, you should look to accelerate large compute-intensive portions of your application in hardware. Implementing these functions in custom hardware lets you achieve an ideal balance between performance and power. The SDAccel environment provides tools and reports to profile the performance of your host application, and determine where the opportunities for acceleration are. The tools also provide automated run-time instrumentation of cache, memory and bus usage to track real-time performance on the hardware.

The SDAccel environment targets acceleration hardware platforms such as the Xilinx® Kintex® UltraScale™ FPGA KCU1500 acceleration platform or the Virtex® UltraScale+™ FPGA VCU1525 acceleration platform, with the PCIe® bus interface to host CPU or data center servers. These acceleration platforms are designed for computationally intensive applications, specifically applications for live video transcoding, data analytics, and artificial intelligence (AI) applications using machine learning. There are also a number of available third-party acceleration platforms compatible with the SDAccel environment.

A growing number of FPGA-accelerated libraries are being offered through the SDAccel environment, such as the Xilinx® Machine Learning (ML) suite to optimize and deploy accelerated ML inference applications. Predefined accelerator functions include targeted applications like machine learning and artificial intelligence with support for many common machine learning frameworks such as Caffe, MxNet and TensorFlow, video processing, encryption, and big data analysis. These predefined accelerator libraries offered by Xilinx and third party developers can be quickly integrated into your accelerated application project to speed development.

Software Acceleration with SDAccel

When compared with processor architectures, the structures that comprise the programmable logic (PL) fabric in a Xilinx® FPGA enable a high degree of parallelism in application execution. The custom processing architecture generated by SDAccel™ for a kernel presents a different execution paradigm from CPU execution, and provides opportunity for significant performance gains. While you can simply retarget an existing application for acceleration on an FPGA, understanding the FPGA architecture and revising your host and kernel code appropriately will significantly improve performance. Refer to the *SDAccel Environment Programmers Guide* (UG1277) for more information on writing your host and kernel code, and managing data transfers between them.

CPUs have fixed resources and offer limited opportunities for parallelization of tasks or operations. A processor, regardless of its type, executes a program as a sequence of instructions generated by processor compiler tools, which transform an algorithm expressed in C/C++ into assembly language constructs that are native to the target processor. Even a simple operation, like the addition of two values, results in multiple assembly instructions that must be executed across multiple clock cycles. This is why software engineers spend so much time restructuring their algorithms to increase the cache hit rate and decrease the processor cycles used per instruction.

On the other hand, the FPGA is an inherently parallel processing device capable of implementing any function that can run on a processor. Xilinx FPGAs have an abundance resources that can be programmed and configured to implement any custom architecture and achieve virtually any level of parallelism. Unlike a processor, where all computations share the same ALU, operations in an FPGA are distributed and executed across a configurable array of processing resources. The FPGA compiler creates a unique circuit optimized for each application or algorithm. The FPGA programming fabric acts as a blank canvas to define and implement your acceleration functions.

The SDAccel compiler exercises the capabilities of the FPGA fabric through the processes of scheduling, pipelining, and dataflow.

- Scheduling: The process of identifying the data and control dependencies between different operations to determine when each will execute. The compiler analyzes dependencies between adjacent operations as well as across time, and groups operations to execute in the same clock cycle when possible, or to overlap the function calls as permitted by the dataflow dependencies.
- Pipelining: A technique to increase instruction-level parallelism in the hardware implementation of an algorithm by overlapping independent stages of operations or functions. The data dependence in the original software implementation is preserved for functional equivalence, but the required circuit is divided into a chain of independent stages. All stages in the chain run in parallel on the same clock cycle. Pipelining is a fine-grain optimization that eliminates CPU restrictions requiring the current function call or operation to fully complete before the next can begin.

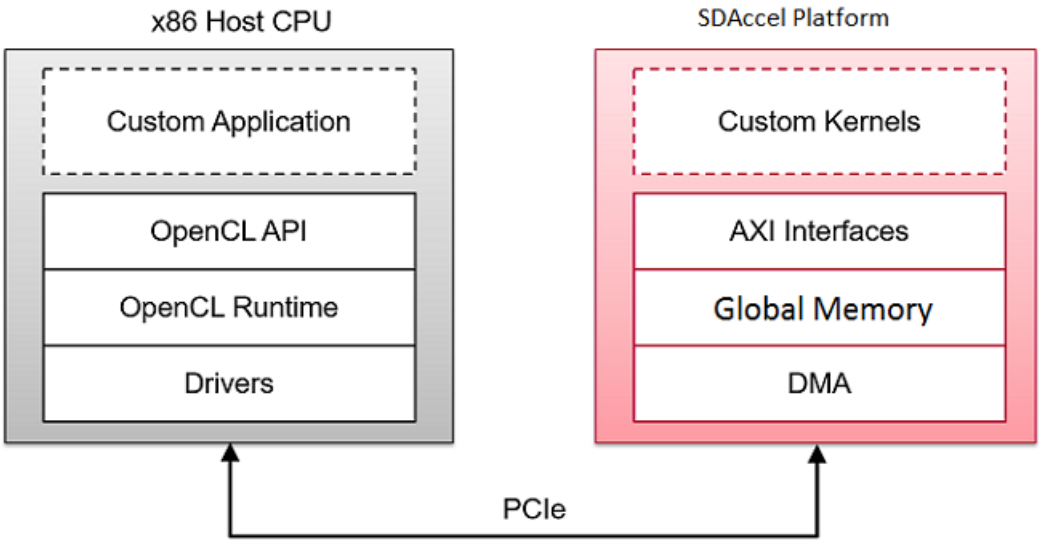
- Dataflow: Enables multiple functions implemented in the FPGA to execute in a parallel and pipelined manner instead of sequentially, implementing task-level parallelism. The compiler extracts this level of parallelism by evaluating the interactions between different functions of a program based on their inputs and outputs. In terms of software execution, this transformation applies to parallel execution of functions within a single kernel.

Another advantage of Xilinx FPGAs is the ability to be dynamically reconfigured. Like loading a compiled program into a processor, reconfiguring the FPGA during run time can re-purpose the resources of the FPGA to implement additional kernels as the accelerated application runs. This lets a single SDAccel accelerator board provide acceleration for multiple functions within an application, either sequentially or concurrently.

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:

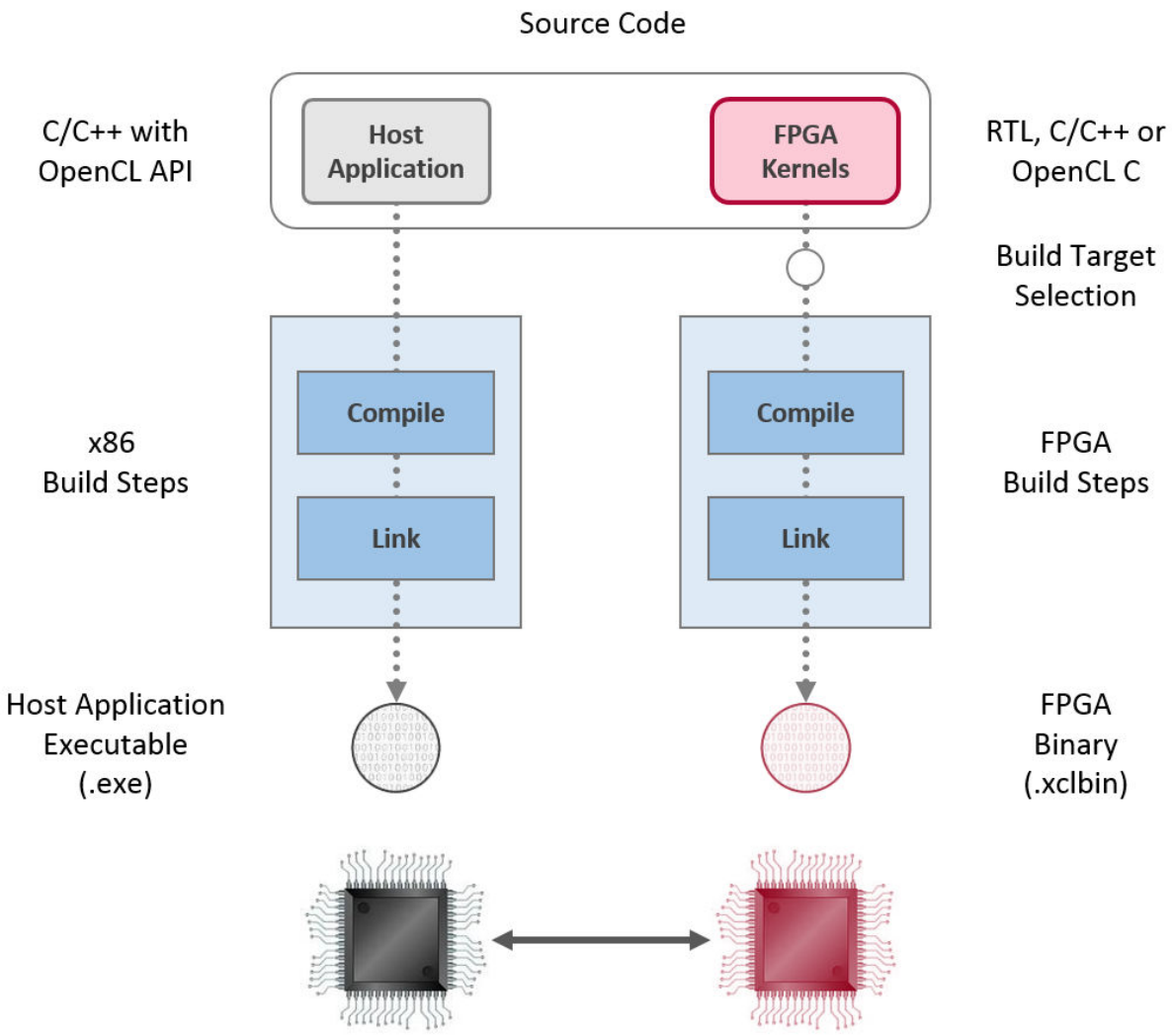
1. The host application writes the data needed by a kernel into the global memory of the SDAccel hardware platform, through the PCIe interface.
2. The host programs the kernel with its input parameters.
3. The host application triggers the execution of the kernel function on the FPGA.
4. The kernel performs 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 image below, 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_emu`): 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 Design Methodology

The SDAccel™ environment supports the two primary use cases:

- Software-centric design: This software-centric approach focuses on improving the performance of an application written by software programmers, by accelerating compute intensive functions or bottlenecks identified while profiling the application.
- Hardware-centric design: The acceleration kernel developer creates an optimized kernel that may be called as a library element by the application developer. This hardware-centric design methodology can be approached by writing the kernel in a standard programming language like C, C++, or OpenCL™, and then synthesized into RTL for implementation into programmable logic; or by using standard RTL design techniques to create and optimize an RTL kernel.

The two use cases can be combined, letting teams of software and hardware developers define accelerator kernels and develop applications to use them. This combined methodology involves different components of the application, developed by different people, potentially from different companies. You can leverage predefined kernel libraries available for use in your accelerated application, or develop all the acceleration functions within your own team.

Software-Centric Design

The software-centric approach to accelerated application development, or acceleration kernel development, uses C or C++ with OpenCL APIs. The code is written as a standard software program, with some attention to the specific architecture of the code. The software development flow typically uses the following steps:

Table 1: **Software-Centric Design Flow**

Step	Comments
Profile application.	Study SDAccel platform specification and programming. Baseline the application in terms of functionalities and performance and isolate functions to be accelerated in HW. Functions that consume the most execution time are good candidates to be offloaded and accelerated onto FPGAs

Table 1: **Software-Centric Design Flow** *(cont'd)*

Step	Comments
Code the desired kernel(s),	Convert functions to OpenCL C or C/C++ kernels without any optimization. The application code calling these kernels will also need to be converted to use OpenCL APIs for data movement and task scheduling.
Verify functionality, iterate as needed.	Run software emulation to ensure functional correctness. Run hardware emulation to generate host and kernel profiling data including: • Estimated FPGA resource usage (non-RTL) • Overall application performance, and • Visual timeline showing host calls and kernel start/stop times.
Optimize for performance, iterate as needed.	Using the various compilation reports and profiling data generated during hardware emulation and system run to assist your optimization effort. Common optimization objectives include: • Optimize data movement from the host to/from global memory, and data movement from global memory to/from the kernel. • Maximize parallelism across software requests. • Maximize parallelism across multiple kernels. • Maximize task and instruction level parallelism within kernels.

Hardware-Centric Design

A hardware-centric flows first focuses on developing and optimizing the kernel(s) and typically leverages advanced FPGA design techniques. The hardware-centric development flow typically uses the following steps:

Table 2: **Hardware-Centric Design Flow**

Step
Study SDAccel platform specification and programming model.
Estimate cycle budgets and performance requirements to define accelerator architecture and interfaces.
Develop accelerator.
Verify functionality and performance. Iterate as needed.
Optimize QoR. Iterate as needed.
Import kernel into SDAccel.
Develop sample host code to test with a dummy kernel having the same interfaces as the actual kernel.
Verify kernel works correctly with host code using hardware emulation, or running on actual hardware. Iterate as needed.
Use Activity Timeline, Profile Summary and timers in the source code to measure performance to optimize host code for performance. Iterate as needed.

Best Practices for Acceleration with SDAccel

Some specific things to keep in mind when developing your application code and hardware function in the SDAccel™ environment:

- Look to accelerate functions that have a high ratio of compute time to input and output data volume. Compute time can be greatly reduced using FPGA kernels, but data volume adds transfer latency.
- Accelerate functions that have a self-contained control structure and do not require regular synchronization with the host.
- Transfer large blocks of data from host to global device memory. One large transfer is more efficient than several smaller transfers. Aim for transfers of at least 4 Kbytes.
- Only copy data back to host when necessary. Data written to global memory by a kernel can be directly read by another kernel without copying back to the host.
- Take advantage of the multiple global memory banks to evenly distribute bandwidth across kernels.
- Maximize bandwidth usage between kernel and global memory by performing 512-bit wide bursts.
- Cache data in local memory within the kernels. Accessing local memories is much faster than accessing global memory.
- In the host application, use events and non-blocking transactions to launch multiple requests in a parallel and overlapping manner.
- In the FPGA use as many kernels as possible to execute multiple tasks in parallel and further increase performance.
- Within the kernels take advantage of tasks-level and instruction-level parallelism to maximize throughput.
- Some Xilinx FPGAs contain multiple partitions (called SLRs). Keep the kernel in the same SLR as the global memory bank that it accesses.
- Use software and hardware emulation to validate your code frequently to make sure it is functionally correct.
- Frequently review the SDAccel Guidance report as it provides clear and actionable feedback regarding deficiencies in your project.

Getting Started

Download and install the SDx™ Tool Suite according to the directions provided in the *SDx Environments Release Notes, Installation, and Licensing Guide* (UG1238).

After installing the SDx tools, you can find detailed instructions and hands-on tutorials to introduce the primary work flows for project creation, specifying functions to run in programmable logic, system compilation, debugging, and performance estimation in the *SDAccel Environment Getting Started Tutorial* (UG1021). Working through the tutorial and its labs is the best way to get an overview of the SDx environment, and should be considered a prerequisite to application development.

***Note*:** The SDx tool suite includes the entire tool stack to create a bitstream, object code, and executables. If you have installed the Xilinx® Vivado® Design Suite and the Software Development Kit (SDK) tools independently, you should not attempt to combine these installations with the SDx tools. Ensure that your tools are derived from an SDx installation (which includes the Vivado Design Suite and SDK tools).

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.*

Creating an SDAccel Project

Within the SDx™ tool you can create an SDAccel™ project using the IDE GUI. The following topic shows you how to set up a SDx workspace, create an SDAccel project, and use key features of the IDE.

TIP: *SDx is based on Eclipse, and needs a workspace environment to be set up.*

In addition to the SDx™ IDE, the SDAccel environment provides a command line interface to support a scripted flow, as described in Chapter 8: Command Line Flow. See the *SDx Command and Utility Reference Guide* (UG1279) for a full list of commands.

Using an SDx Workspace

You can launch the SDx IDE directly from the command line:

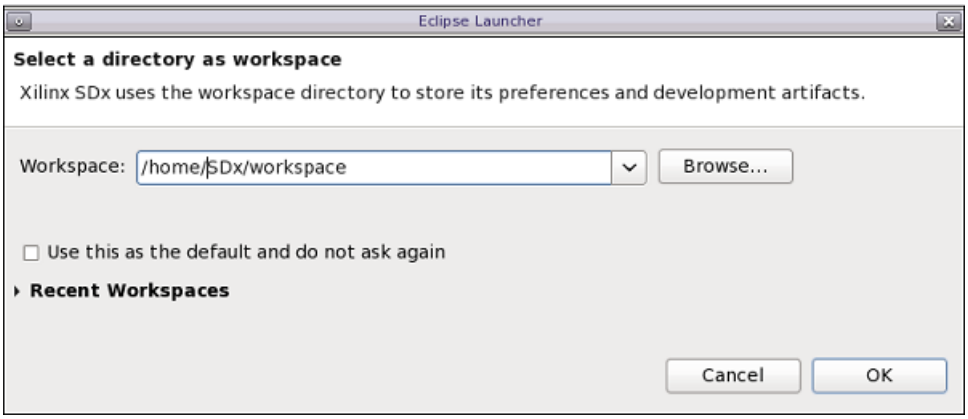
```
sdx
```

TIP: *When opening a new shell to enter an SDx™ command, ensure that you first source either the* `settings64.sh` *file or* `settings64.csh` *to setup a path to the tools. This sets up the* `$PATH` *and* `$LD_LIBRARY_PATH` *environment variables and other required settings. See the SDx Environments Release Notes, Installation, and Licensing Guide (UG1238) for more information.*

IMPORTANT!: *If you use a single computer to perform both development and deployment, be sure to open separate terminals when running the SDx tools and the* `xbinst` *board installation utility. Running both tools from the same terminal adversely affects the environment variable and causes tool issues.*

The SDx IDE opens, and prompts you to select a workspace, as shown in the following figure.

Figure 3: **Specify SDAccel Workspace**



The SDx workspace is the 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. The following instructions show you how to define a workspace for an SDAccel project.

1. Click 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** check box to set the specified workspace as your default choice and eliminate this dialog box in subsequent uses of SDx.
3. Click **OK**.

TIP: *You can change the current workspace from within the SDx IDE by selecting **File→Switch Workspace**.*

You have now created an SDx workspace.

Creating an Application Project

TIP: *Example designs are provided with the SDAccel™ tool installation, and also on the Xilinx® GitHub repository. See Appendix A: Getting Started with Examples for more information.*

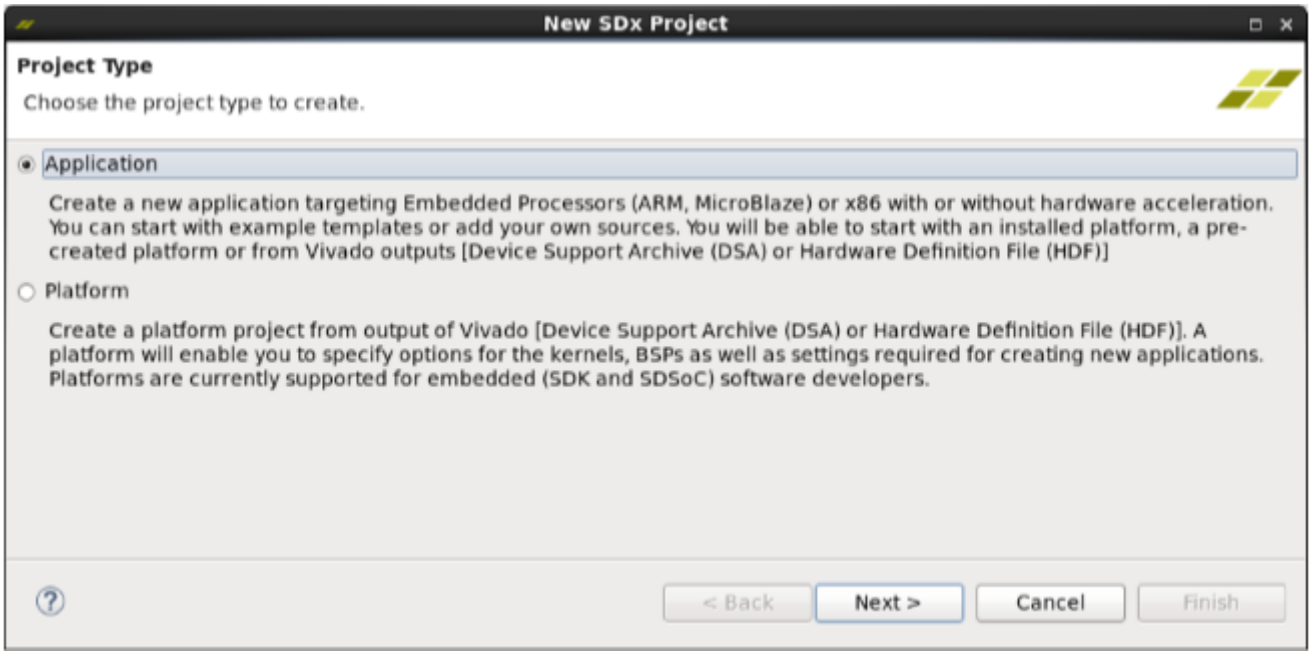
After launching the SDx™ IDE you can create a new SDx Project. Select **File → New → SDx Project**. If this is the first time the SDx IDE has been launched, the SDx Welcome screen opens to let you specify an action. Select **Create SDx Project**.

The New SDx Project wizard opens. In the Project Type page, you can choose to create either an Application or a Platform project:

- **Application Project**: A software application with portions of the application optionally accelerated onto kernelshardware functions in the programmable logic (PL) fabric.
- **Platform Project**: Defines the base hardware and software components that make up the SDAccel platform.

Select **Application** and click **Next**.

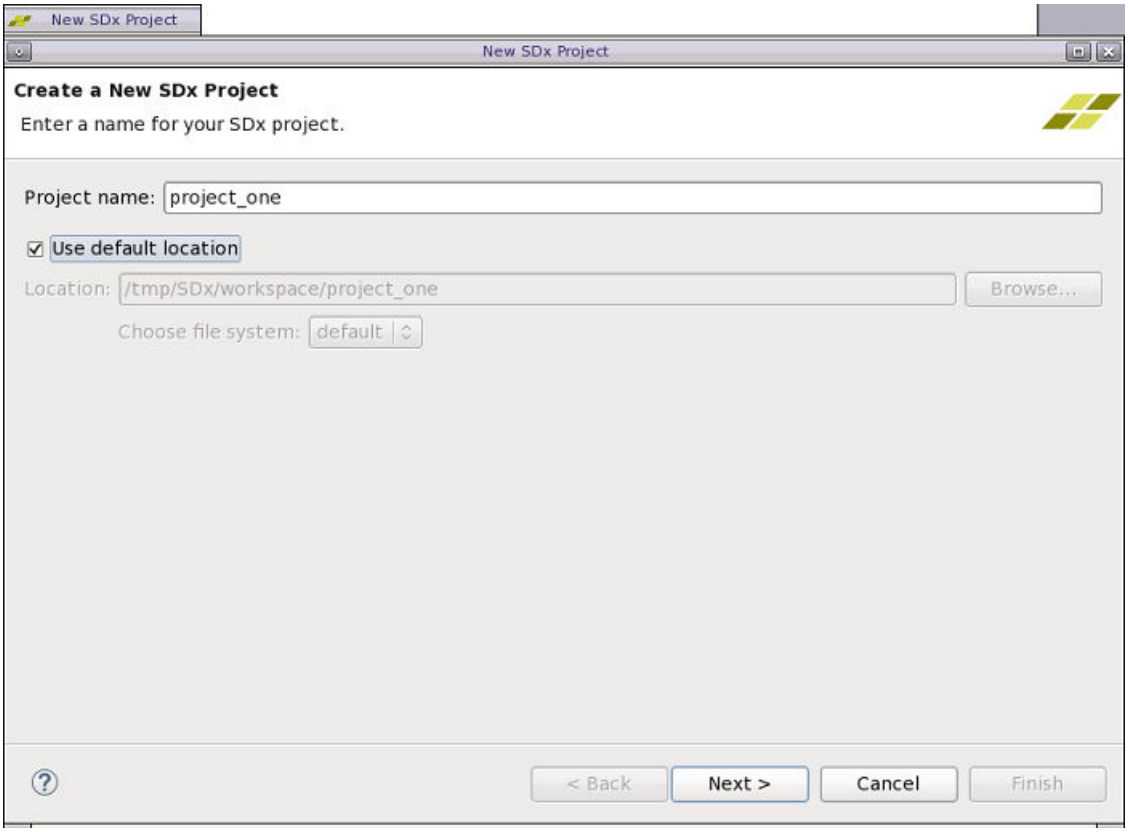
Figure 4: **Project Type**



In the Create a New SDx Project page, you can name your project.

Specify the **Project name**.

Figure 5: **Create New SDx Project**



The **Use default location** is selected by default to locate your project in a folder in the SDx workspace. You can uncheck this check box to specify that the platform project is created in a **Location** of your choice.

If you specify the location, you can use **Choose file system** to select the **default** file system, or enable the Eclipse Remote File System Explorer (**RSE**).

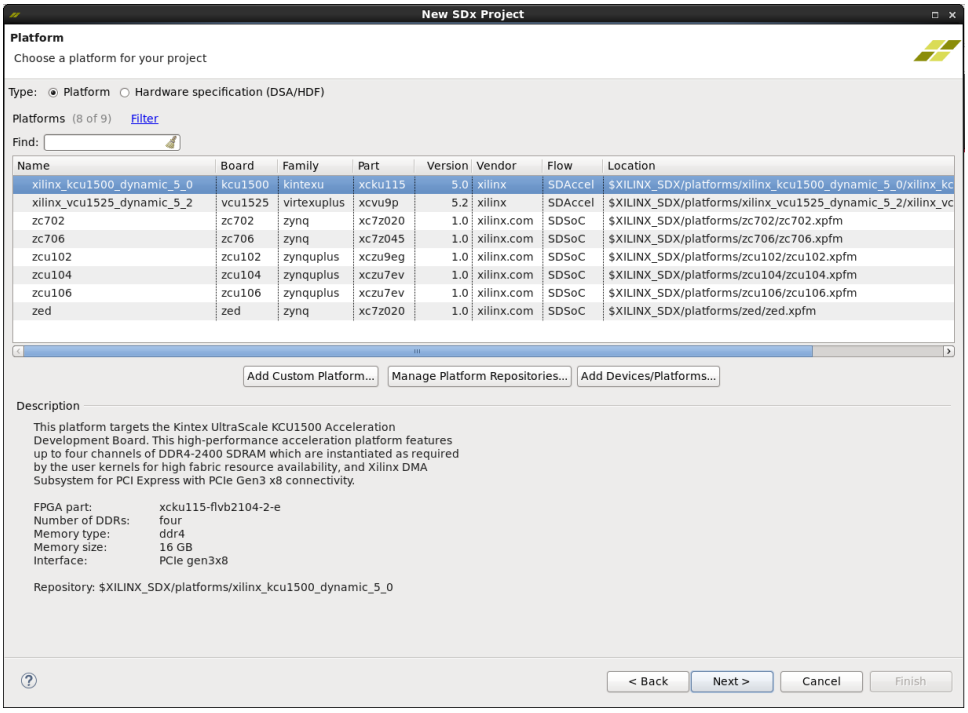
IMPORTANT!: *The project location cannot be a parent folder of an SDx workspace.*

Click **Next**.

The Platform page, similar to the one in the following figure, displays two selection options for an application: a Platform or an Application that uses a device support archive (DSA) or hardware definition file (DSA/HDF.) All SDAccel applications are based on a platform, not a hardware specification.

You can add platforms into a repository, or even develop a platform to add to projects. See Appendix D: Managing Platforms and Repositories for more information.

Figure 6: **Specify SDAccel Platform**



A platform is composed of a DSA, which describes the base hardware design, the metadata used in attaching accelerators to declared interfaces, and the software environment, which can include operating system images (for example, Linux), as well as boot-up and run-time files.

SDAccel offers pre-defined platforms for specific boards based on the following:

- Kintex® UltraScale+™
  - kcu1500
- Virtex® UltraScale+™
  - vcu1525

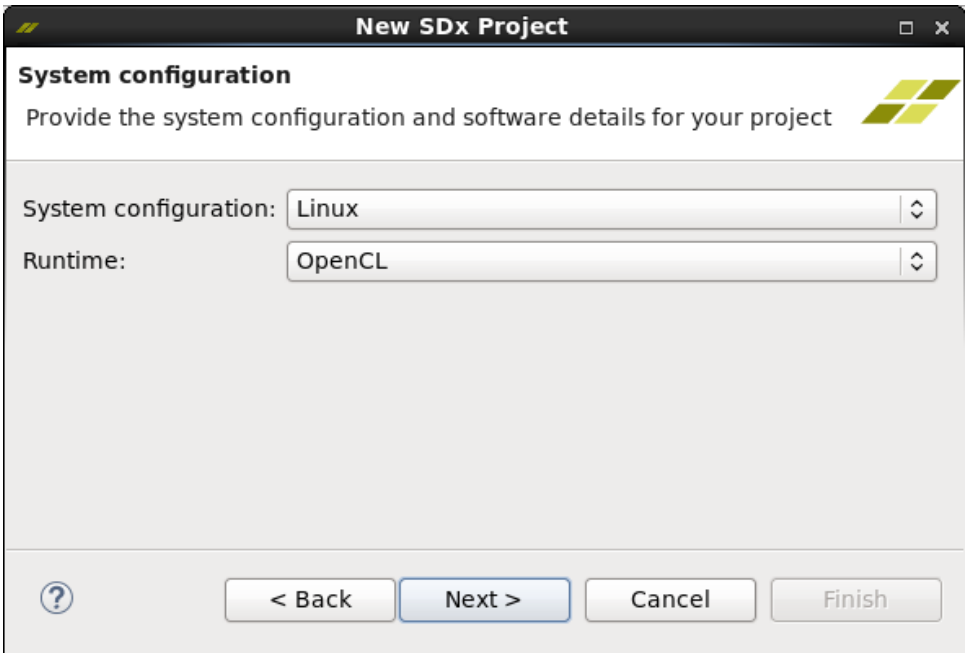
You can select the target platform for your project from the listed platforms. You can also filter the listed platform by clicking the **Filter** link, which opens the Filter dialog box so you can set filters to specify the Flow, Family, or Vendor for your target platform. By default, **Latest Revision** is checked. This option filters out older revisions of the listed platforms.

IMPORTANT!: *Your selection of a platform determines if you are working in an SDAccel project or an SDSoC™ project. Be sure to select the right platform for your project, as subsequent processes are driven by this choice. The* ***Flow*** *column indicates an SDAccel or SDSoC project.*

Select either kcu1500 or vcu1525 to use the pre-defined SDAccel platform and click **Next**.

After selecting the target platform and clicking **Next**, the System Configuration page opens, as shown in the following figure. It lets you select a **System configuration** and select **Runtime** from a list of those defined for the selected platform. The System Configuration defines the software environment that runs on the hardware platform. It specifies the operating system and the available run-time settings for the processors in the hardware platform.

Figure 7: **Specify System Configuration**

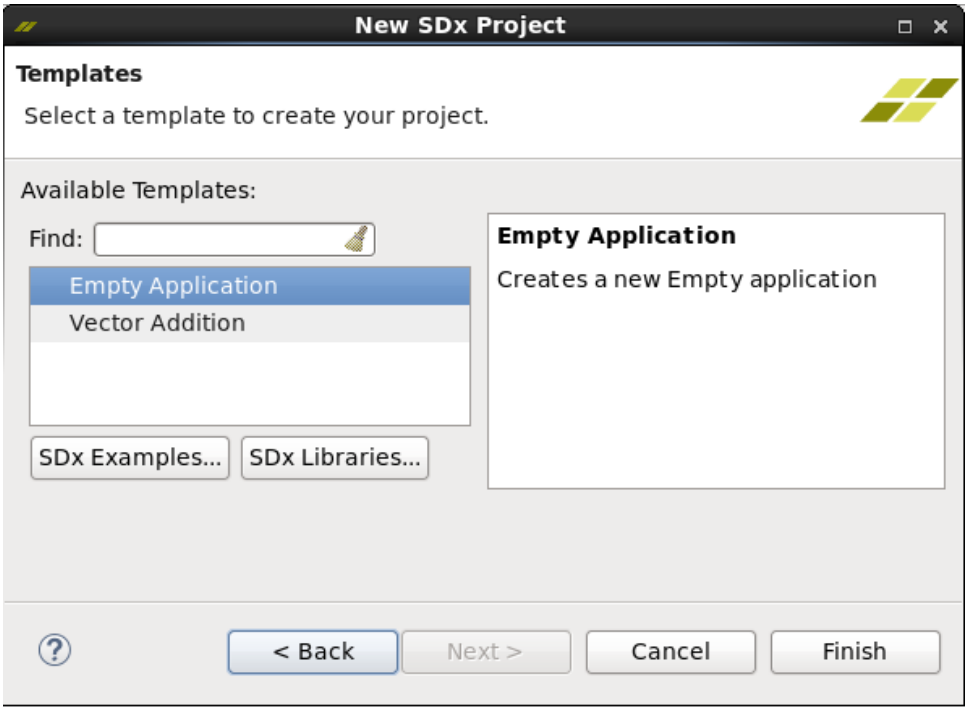


After selecting the **System Configuration** and clicking **Next**, the Templates page displays, as shown in the following figure. It lets you specify an application template for your new project. The `samples` directory within the SDx tools installation contains multiple source code example templates.

Initially, the Template dialog box has an Empty Application and a Vector Addition application.

To access additional SDAccel examples, click the **SDx Examples** button. When the SDx Examples dialog box opens, click the Download button for the SDAccel Examples. Then click **OK**. The downloaded examples are now listed in the Templates page.

Figure 8: **Application Templates**



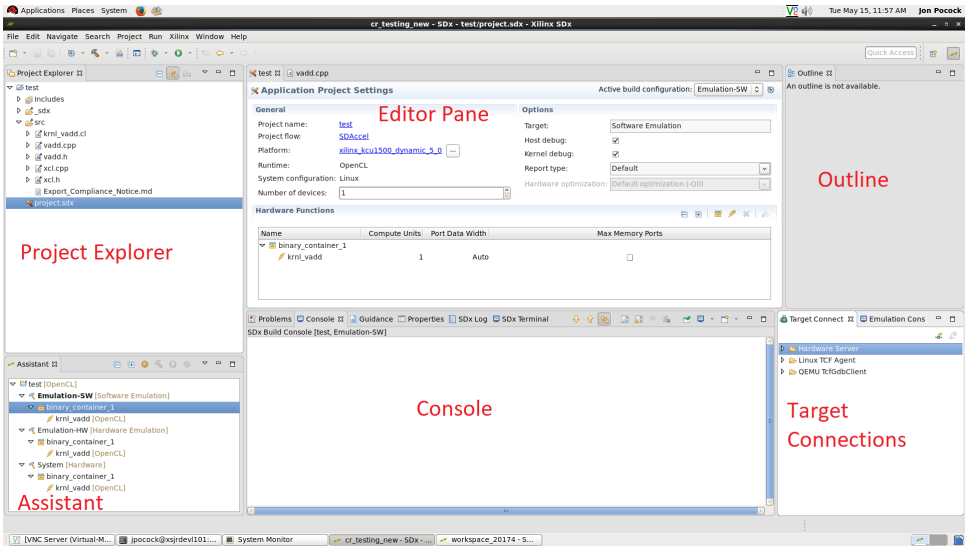
You can use the template projects as examples to learn about the SDx tool and acceleration kernels or as a foundation for your new project. Note that you must select a template. You can select **Empty Application** to create a blank project into which you can import files and build your project from scratch.

Click **Finish** to close the New SDx Project wizard and open the project.

Working with the SDx GUI

When you open a project in the SDx™ IDE, the workspace is arranged in a series of different views and editors, also known as a *perspective* in the IDE. The tool opens with the SDx (default) perspective shown in the following figure.

Figure 9: **SDAccel - Default Perspective**



Some key views/editors in the default perspective are:

- **Project Explorer:** Displays a file-oriented tree view of the project folders and their associated source files, plus the build files, and reports generated by the tool.
- **Assistant:** Provides a central location to view/edit settings, build and run your SDAccel application, launch profiling and debug sessions, and open reports.
- **Editor Area:** Displays project settings, build configurations, and provides access to many commands for working with the project.
- **Console Area:** Presents multiple views including the command console, design guidance, project properties, logs and terminal views.
- **Outline:** Displays an outline of the current source file opened in the Editor Area.
- **Target Connections:** Provides status for different targets connected to the SDx tool, such as the Vivado® hardware server, Target Communication Framework (TCF), and quick emulator (QEMU) networking.

To close a view, click the **Close** button (x) on the tab of the view. To open a view, select **Window → Show View** and select a view. You can arrange views to suit your needs by dragging and dropping them into new locations in the IDE.

To save the arrangement of views as a perspective, select **Window → Perspective → Save Perspective As**. This lets you define different perspectives for initial project editing, report analysis, and debug for example. Any changes made without saving as a perspective are stored with the workspace. To restore the default arrangement of views, select **Window → Perspective → Reset Perspective**. SDx (default) perspective.

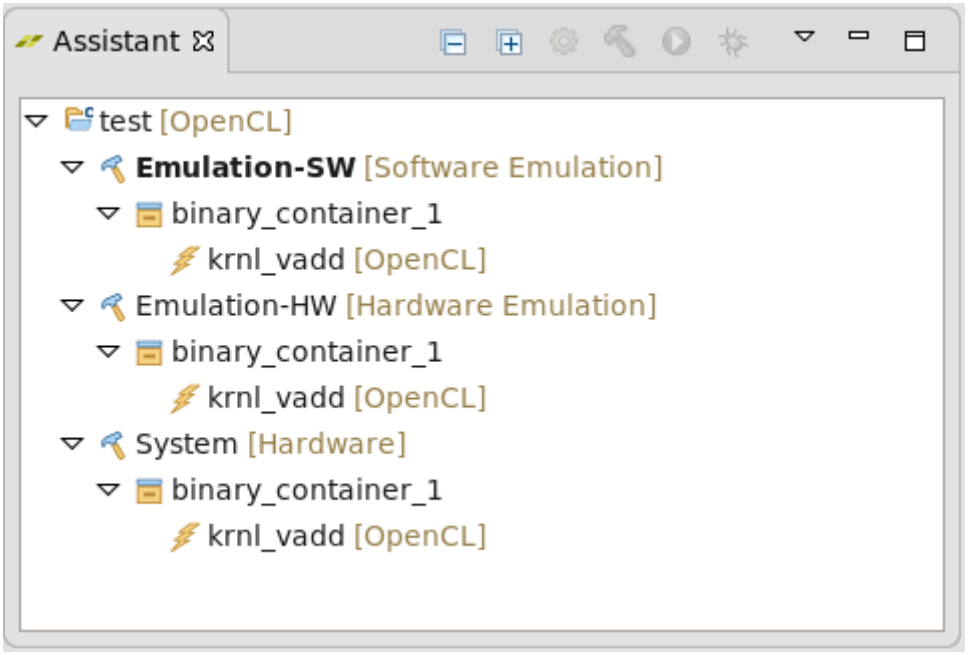
To open different perspectives, select **Window → Perspective → Open Perspective**. To restore the SDx (default) perspective, click the SDx button on the right side of the main toolbar

SDx Assistant

The Assistant view provides an SDx™-centric project tree to manage settings, builds, run times, profile, debug, and reports. It is a companion view to the Project Explorer and is opened by default directly below the Project Explorer view.

An example view of the Assistant and its tree structure is shown below.

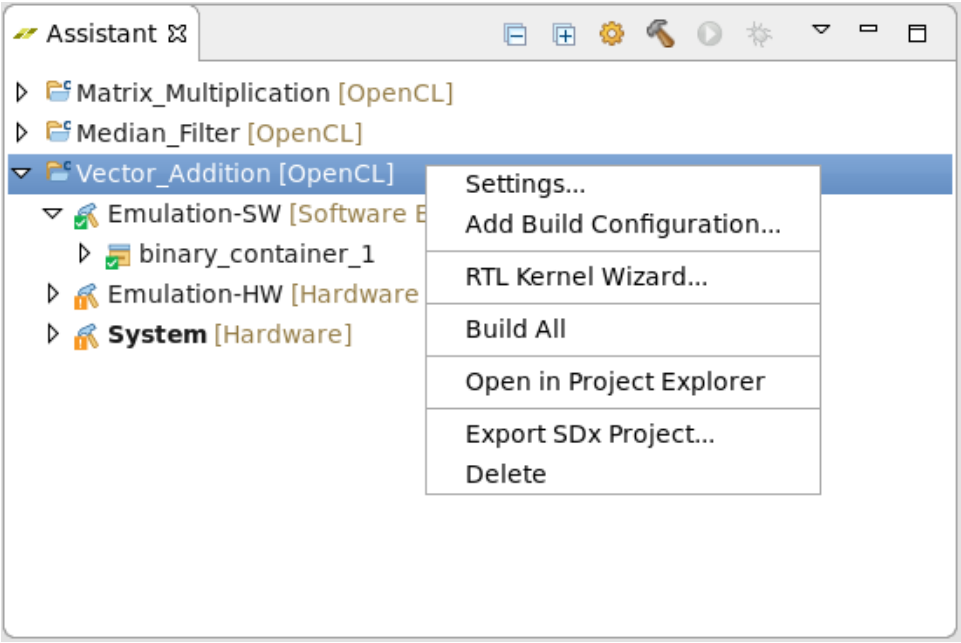
Figure 10: **Assistant Tree Structure Example**



Each item in the tree has a type-specific right-click menu with actions for that item. The actions can open dialogs, reports or views, start processes, or launch external tasks.

For example, right-clicking the 'Vector_Addition' project displays the following menu:

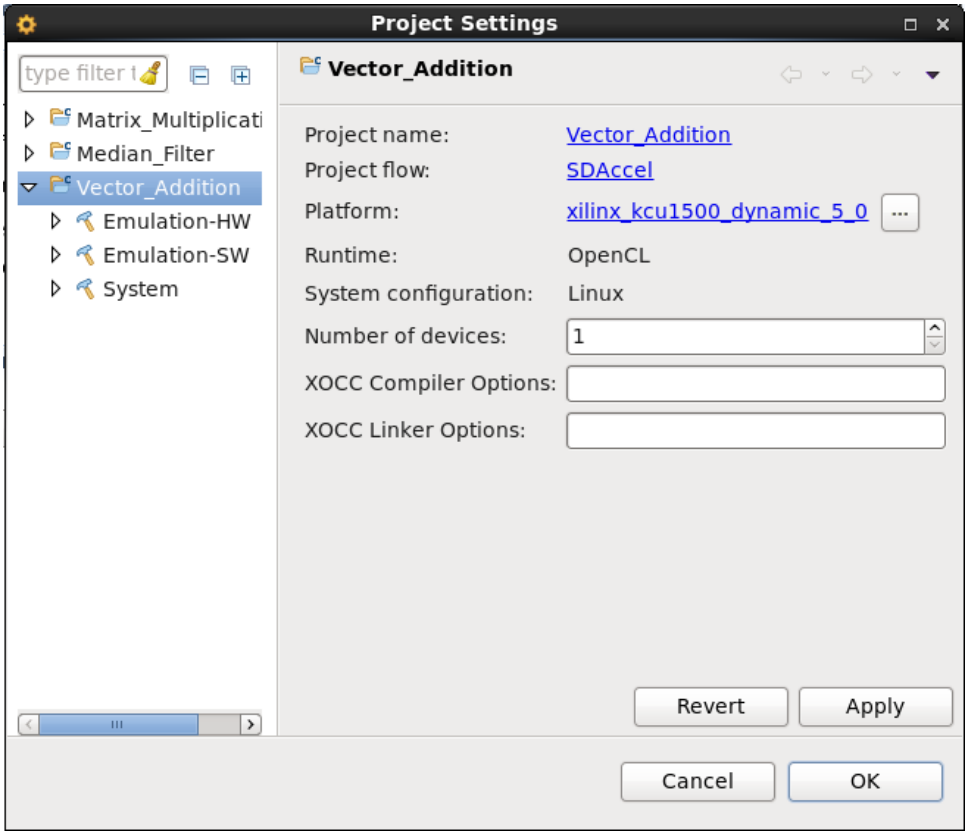
Figure 11: **Assistant Right-Click**



TIP: *Actions that are not currently available will be disabled/grayed out.*

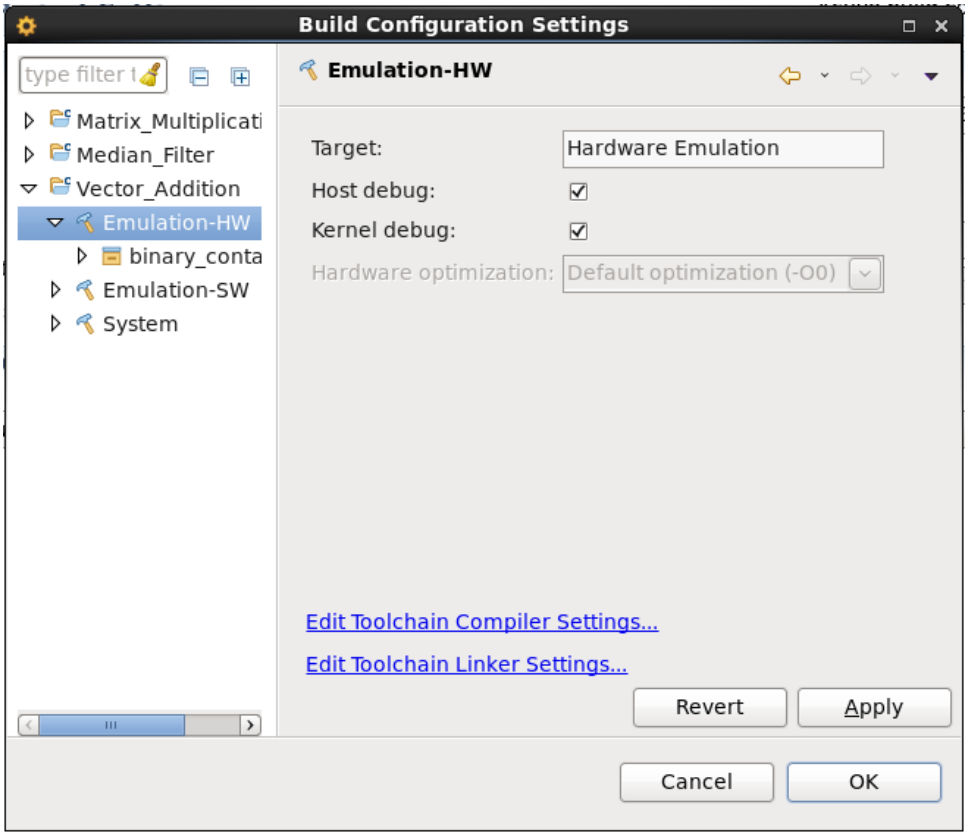
Select **Settings** to open the Project Settings dialog box.

Figure 12: **Assistant Settings Expanded**



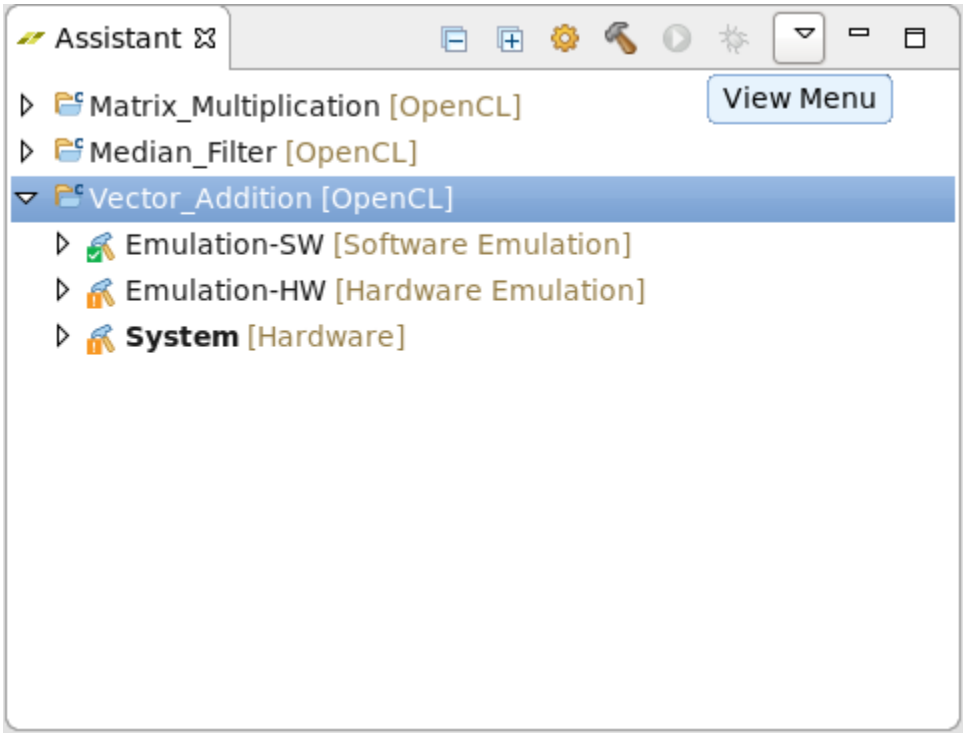
You can select the settings for the various items listed in the tree. For example, if you select the setting for the **Emulation-HW** build configuration, it displays the following. The Assistant makes it very easy to navigate through the design objects and view/update their settings.

Figure 13: **Assistant Settings Debug**



Finally, the **View** menu includes options that affect the Assistant view behavior, but do not affect project data. It can be selected by left-clicking the downward pointing arrow shown in the following graphic.

Figure 14: **Assistant View Menu**



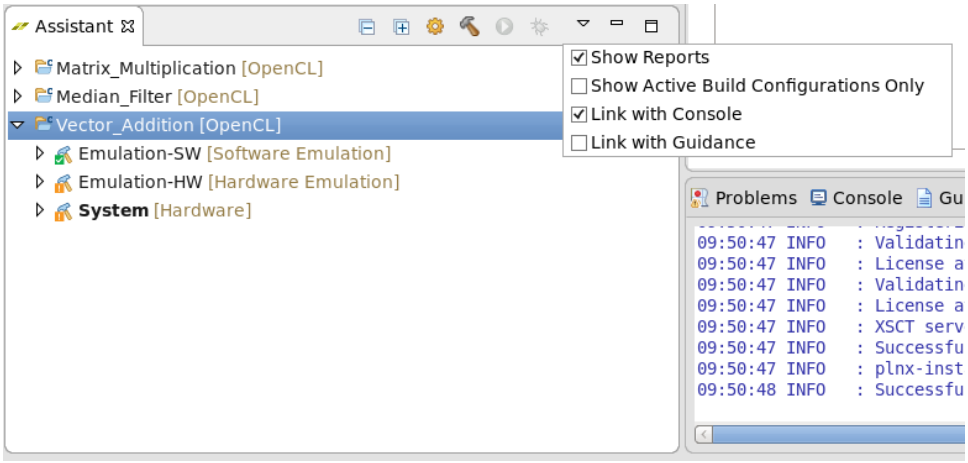
It displays the following options:

- **Show Reports:** If checked, reports will be visible in the tree. If not checked, reports will not be shown in the tree. Reports will only appear in the tree when they exist in the project, usually after a project has been built or run, with specific settings.
- **Show Active Build Configurations Only:** If checked, the tree will only show the "active" build configurations for each project. The active build configuration, in the assistant, will be the most-recently-built configuration. The active build configuration can also be changed to view the standard CDT methods (Project > Build Configurations > Set Active, or Project > Build Configurations > Manage).

  When using the assistant to iterate on a specific build, it can be helpful to see only the current build configuration.
- **Link with Console:** If checked, the build console in the Console view will switch automatically to match the current selection in the Assistant tree, if the selection is for a build configuration, or a descendant in the tree from a build configuration. If not checked, the console will not automatically switch when the Assistant selection changes.
- **Link with Guidance:**

If checked, the Guidance in the Console area automatically switches to match the current selection in the Assistant tree.

Figure 15: **Assistant Link with Console View Menu**



You can see that with a couple of clicks you can easily access many of the functions and features of the tool using the Assistant.

SDx Project Export and Import

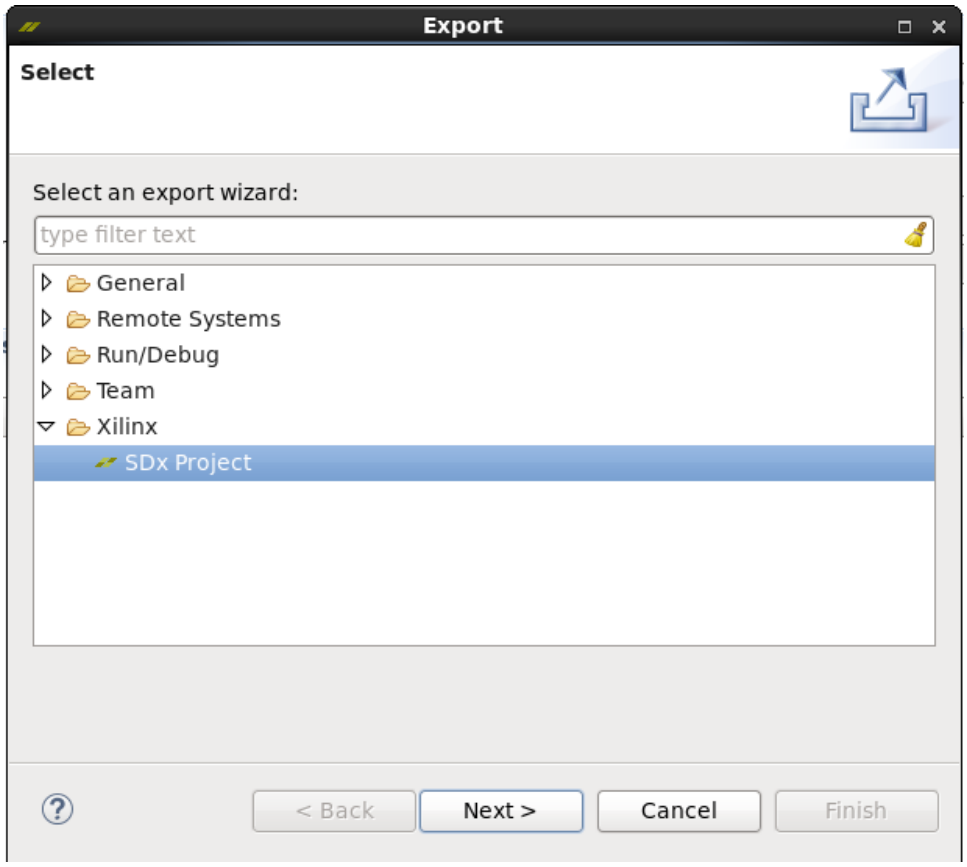
SDx™ provides a simplified method for exporting/importing one or more SDx projects within your workspace. You can optionally include associated project build folders.

Export an SDx Project

When exporting a project, it archives the project in a zip file with all the relevant files necessary to import into another workspace.

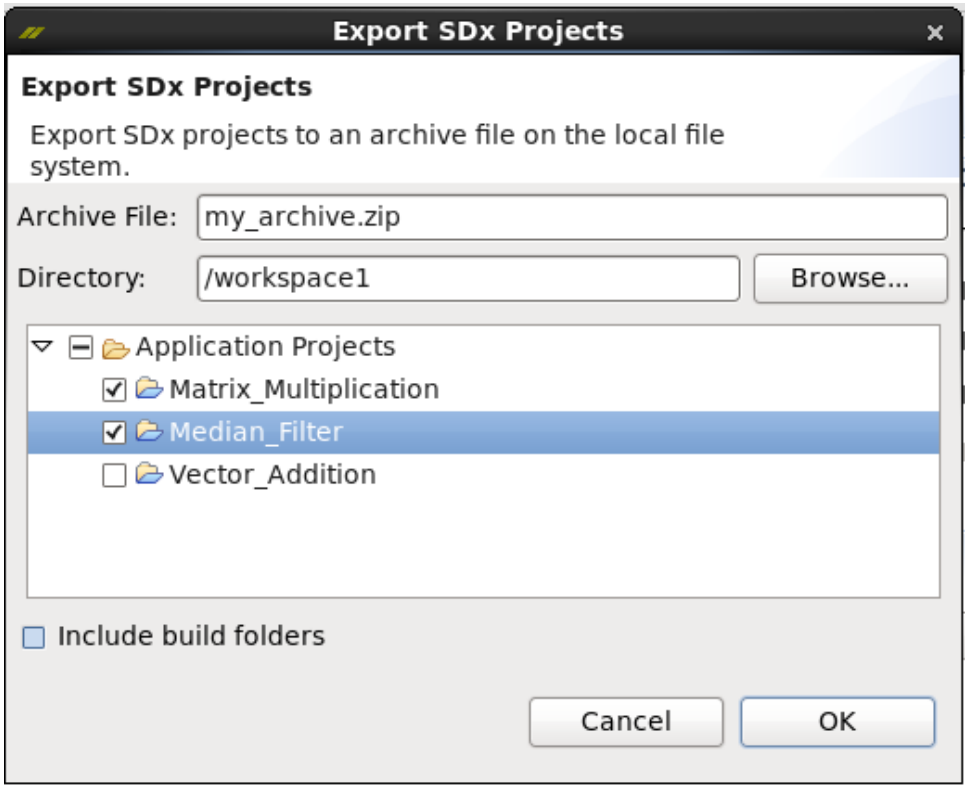
1. To export a project, select **File** → **Export** from the main menu.
2. When the Export window opens, select the SDx™ Project import wizard under the Xilinx folder as shown in the following figure and click **Next**.

Figure 16: **Export Select Export Wizard**



3. When the Export SDx Projects window opens, the projects in the workspace will be displayed as shown in the following figure. Select the desired projects to be included in the archive by checking the respective check boxes. Enter the name of the archive file and the directory location to where you wish to save the file. In addition, you can optionally include the associated project build folders in the archive by checking the Include build folders check box. The build folders include all the build related files including reports and bit files for example.
4. Click **OK** to save the archive zip file.

Figure 17: **Export Filename Build Folder**

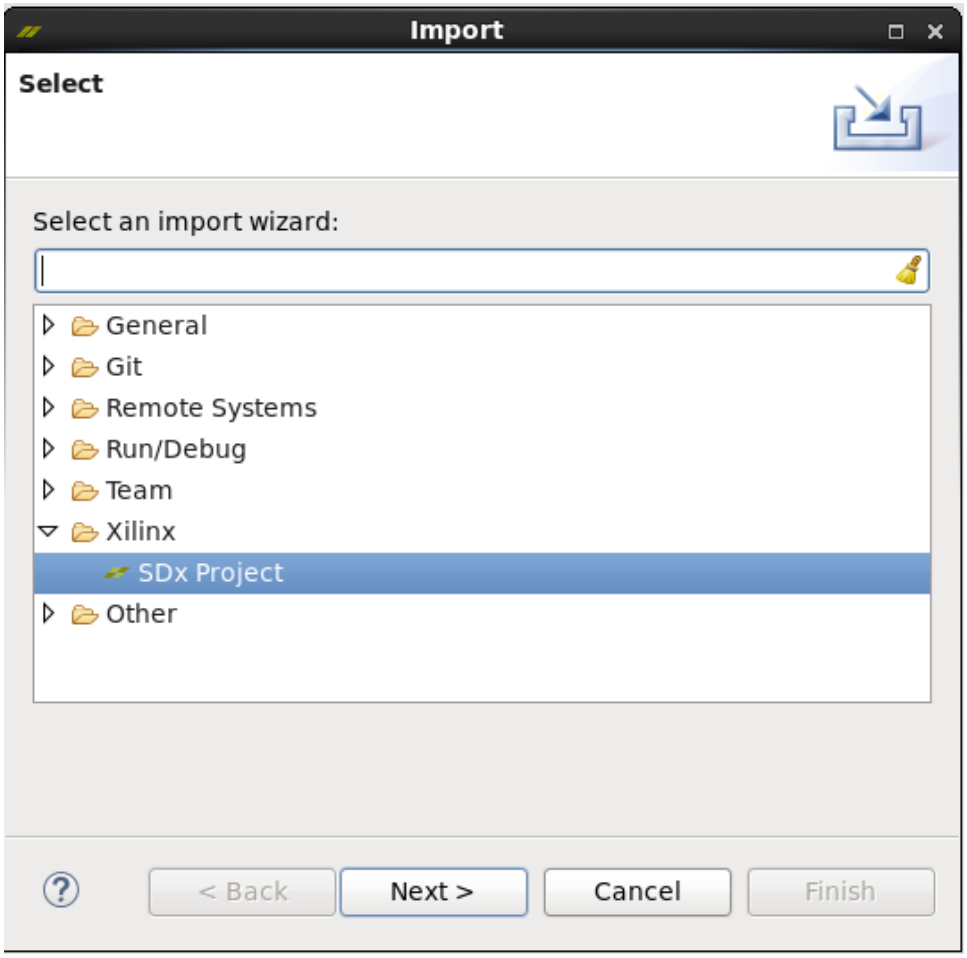


The SDx projects have been successfully archived and can be imported into a different workspace.

Import an SDx Project

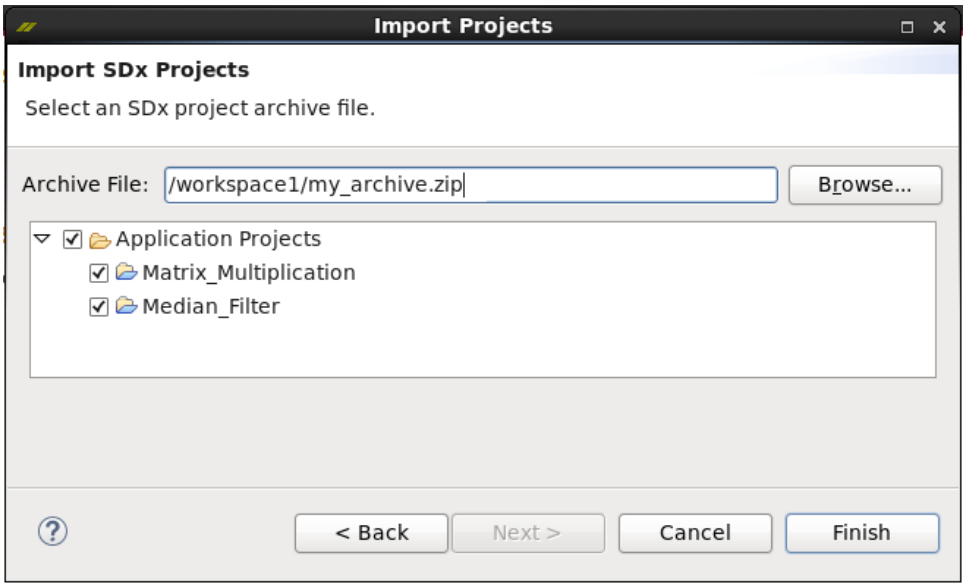
1. To import an SDx™ project, select **File** → **Import** from the top menu.
2. From the Import window, select the SDx Project import wizard under the Xilinx® folder as shown in the following figure and click **Next**.

Figure 18: **Import Select an Import Wizard**



3. This will open the Import SDx Projects window. Browse and select the desired archive file. It will display the archived projects.
4. Select the projects you wish to import using the check boxs. In the following figure both projects are selected for import.
5. Click **Finish** to import the projects into your workspace.

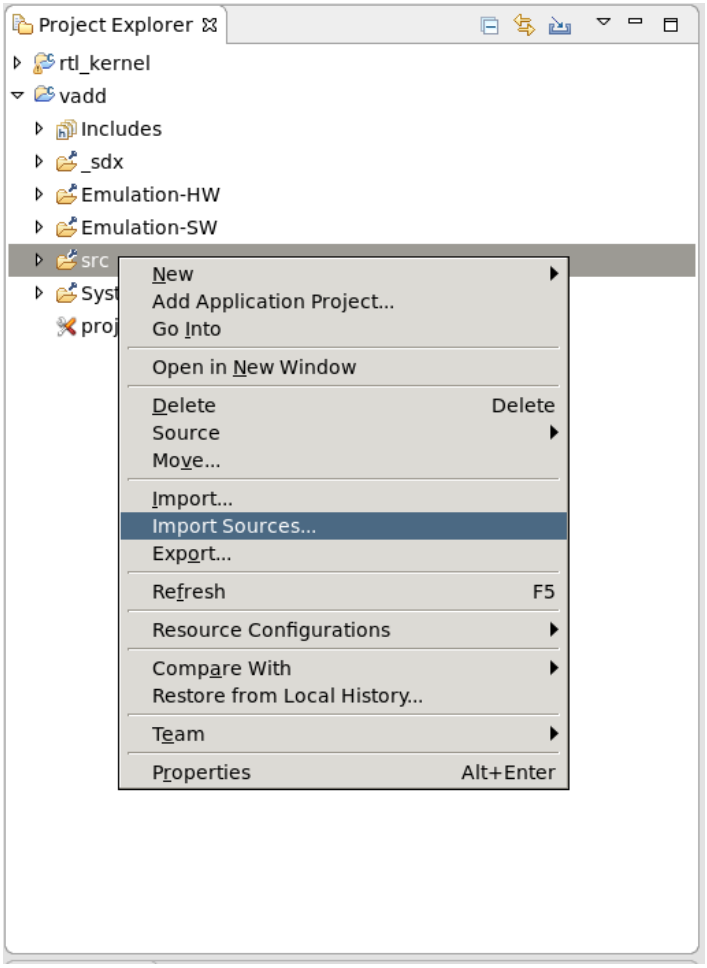
Figure 19: **import Archive Filename**



Adding Sources

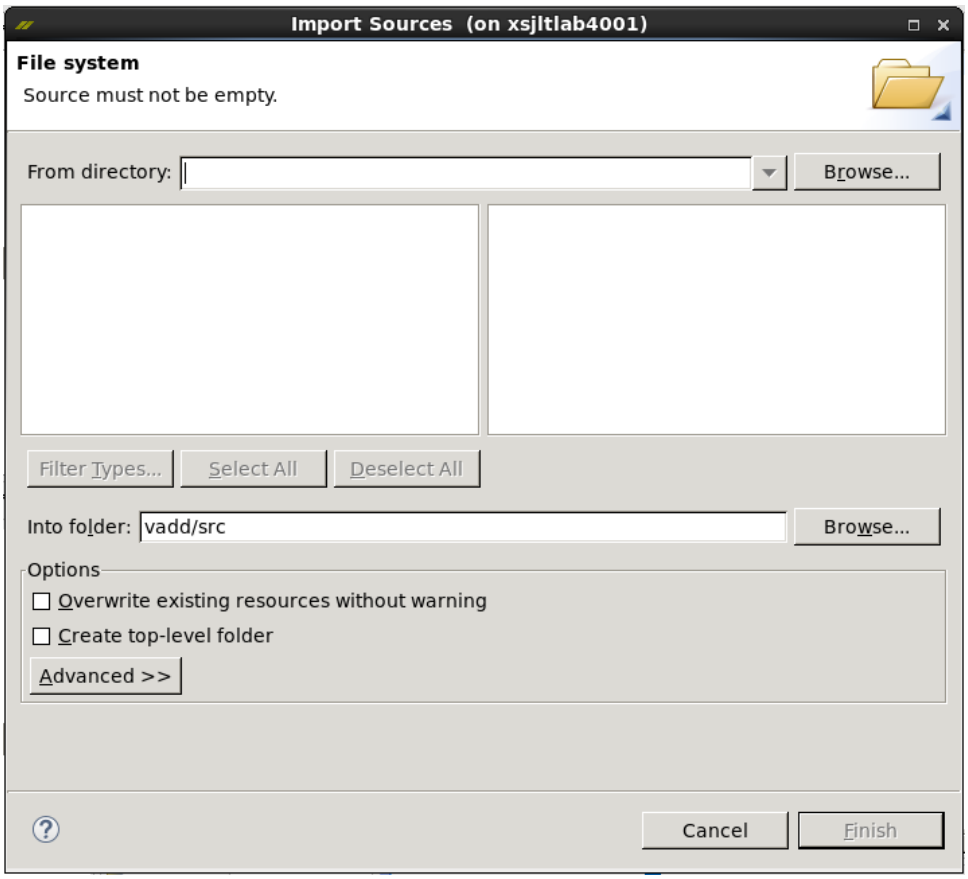
A project consists of many different sources including those for host application code, kernel functions and even pre-compiled XO files. With the project open in the SDx IDE, you can add these source files to the project by right-clicking the `src` folder in the Project Explorer, and selecting **Import Sources...** as shown in the following image.

Figure 20: **Import Sources Right Click**



This displays the Import Sources dialog box shown in the following image. In the dialog box, click the **Browse** button to select the directory from which you want to import sources. Select the desired sources and click **Finish**.

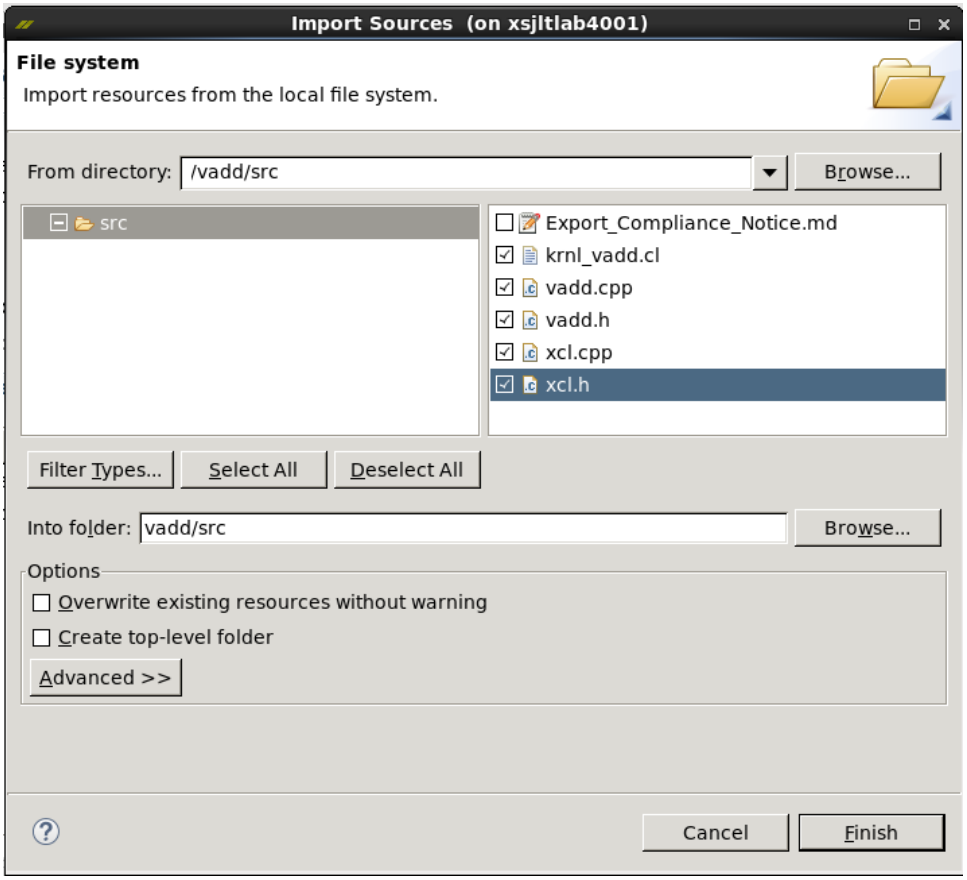
Figure 21: **Import Sources Directory**



The source files in that directory will be displayed. Select the desired sources to import by selecting the appropriate check-boxes and click **Finish**. In the following image C/C++, OpenCL™ and header files will be imported into the project.

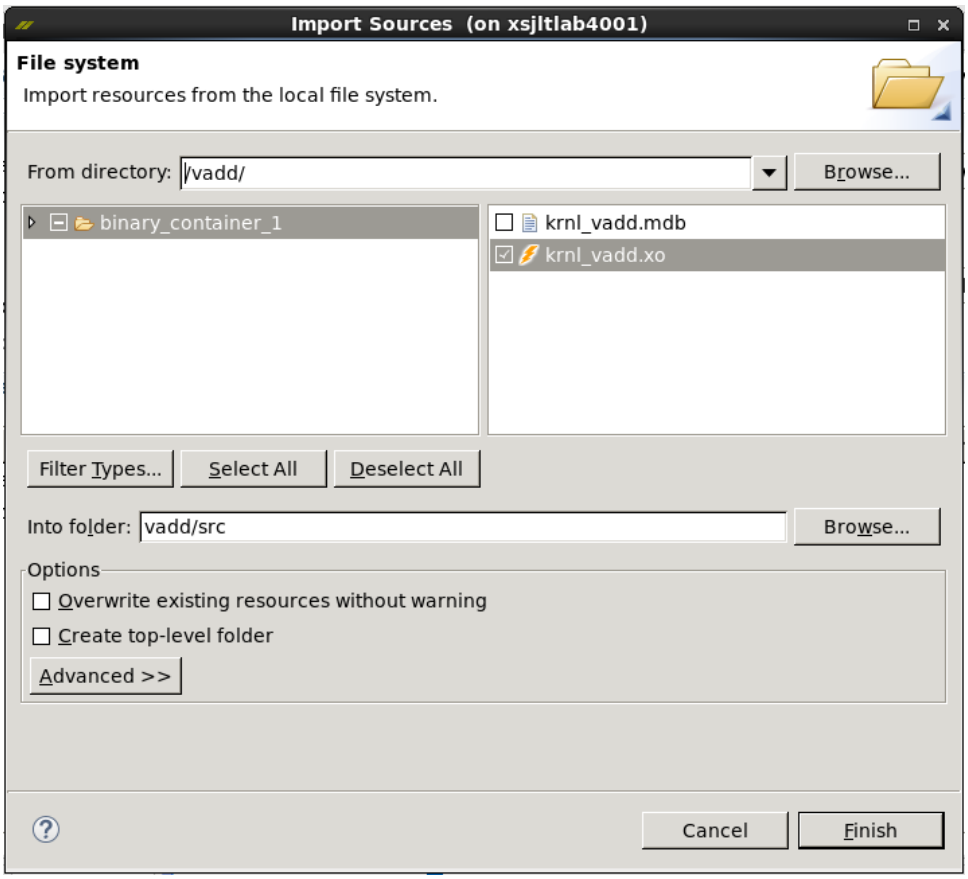
IMPORTANT!: *When you import source files into a workspace, it copies the file into the workspace. Any changes to the files are lost if you delete the workspace.*

Figure 22: **Import Sources Dialog Box**



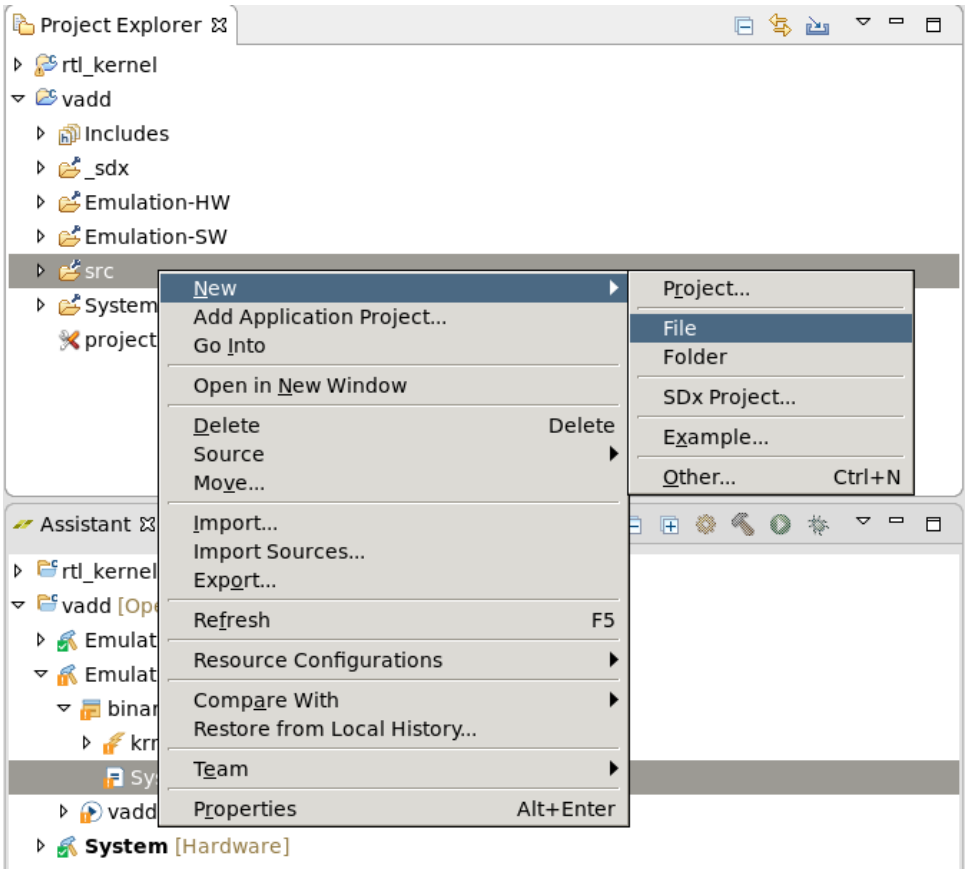
Similarly you can import compiled kernels (`.xo` files) into the project through the **Import Sources...** selection. In the following image, the `krnl_vadd.xo` file will be imported into the project.

Figure 23: **Import XO Sources**



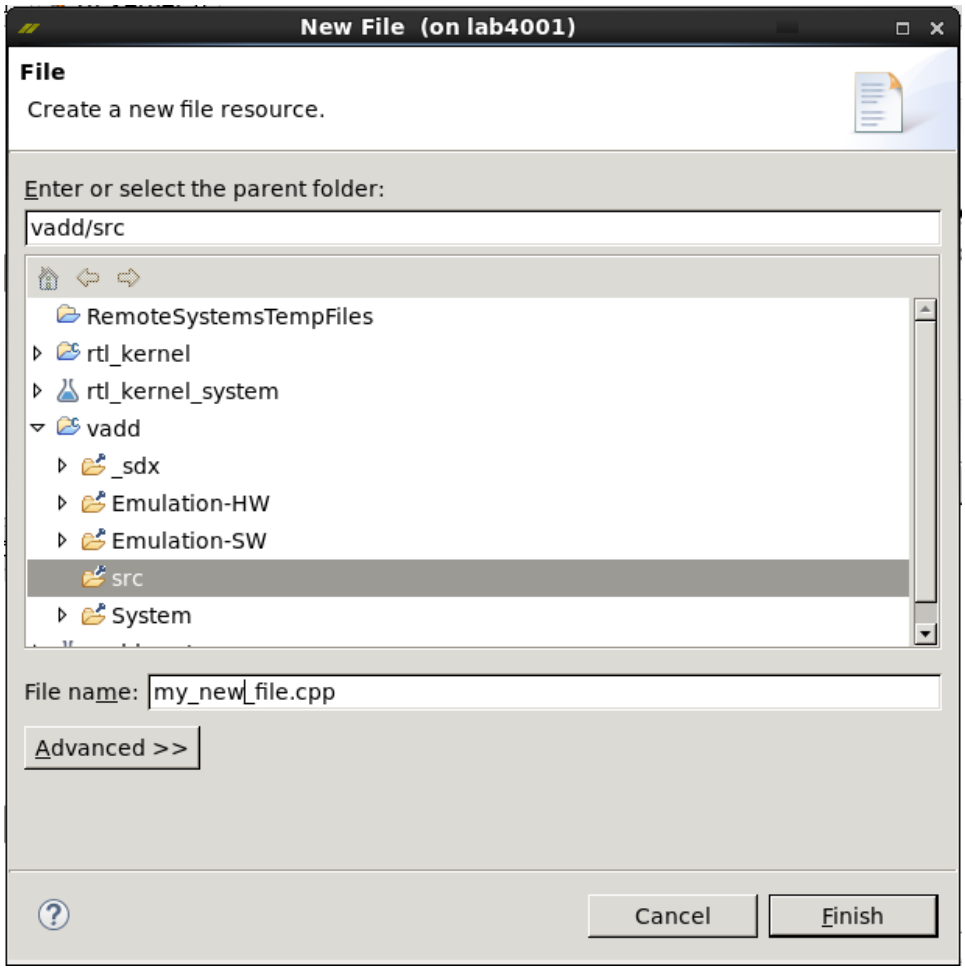
In addition to importing sources, you can also create and edit new source files in the GUI. With the project open in the SDx IDE, right-click the `src` folder and select **New → File** as shown in the following image.

Figure 24: **New File Select**



Select the folder in which to create the new file and enter a filename as shown in the image below. Click **Finish** to add the file to the project.

Figure 25: **New File Name**



After adding source files to your project, you are ready to begin configuring, compiling, and running the application. You can open a source file by expanding the `src` folder in the Project Explorer and double-clicking on the file.

Programming for SDAccel

The custom processing architecture generated by SDAccel™ 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 following topics discuss 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.

In addition, the FPGA fabric can support multiple kernels running simultaneously. Therefore, you can create multiple instances of a single kernel, or configure multiple kernels on the same device, to increase the performance of the host application. Kernels running on the FPGA can have one or more memory interfaces to connect to the global memory of the platform. You will manage both the number of kernels running on the FPGA, and the specific memory banks accessed by the kernel through XOCC linking options during the build process.

The content discussed here is provided in greater detail in the *SDAccel Environment Programmers Guide* (UG1277). Refer to that guide for details of the host application, kernel code, and the interactions between them.

Coding the Host Application

When creating the host application, you must manage the required overhead to setup and configure the SDAccel™ runtime, program and launch the kernel, pass data back and forth between the host application and the kernel, as well as address the primary function of the application.

Setting up the Runtime

Within every host application you must setup the environment to identify the OpenCL™ platform and the device IDs, specify a context, create a command queue, build a program, and spawn one or more kernels. The program identifies and configures the kernel, and transfers data between the host code and the kernel. In the host code, this process could use the following steps:

TIP: *The following code examples are taken from the IDCT example design.*

1. To set up the OpenCL™ runtime environment, you need to identify the Xilinx® platform using the `clGetPlatformIDs` and `clGetPlatformInfo` commands. For example:

```
// get all platforms
std::vector<cl_platform_id> platforms(platform_count);
clGetPlatformIDs(platform_count, platforms.data(), nullptr);

for (int p = 0; p < (int)platform_count; ++p) {
  platform_id = platforms[p];
  clGetPlatformInfo(platform_id,CL_PLATFORM_VENDOR,1000,(void
*)cl_platform_vendor,NULL);
  clGetPlatformInfo(platform_id,CL_PLATFORM_NAME,1000,(void
*)cl_platform_name,NULL);
  if(!strcmp(cl_platform_vendor,"Xilinx")) {...}
```

2. Identify the Xilinx® devices on the platform available for enqueuing kernels, using the `clGetDeviceIDs` command. Finding the device IDs requires the platform ID discovered in the prior step. For example:

```
clGetDeviceIDs(platform_id, CL_DEVICE_TYPE_ACCELERATOR, 1, &device_id,
NULL);
```

3. Setup the context using `clCreateContext`. The context is the environment that work-items execute, and identifies devices to be assigned transactions from the command queue. The example below shows the creation of the context:

```
cl_context cntxt = clCreateContext(0, 1, &device_id, NULL, NULL, &err);
```

4. Define the command queue using `clCreateCommandQueue`. The command queue is a list of commands or kernels waiting to be assigned to a device. You can setup the command queue to handle commands in the order submitted, or to be asynchronous so that a command can be executed as soon as it is ready. Use the out-of-order command queue, or multiple in-order command queues, for concurrent kernel execution on the FPGA. An example follows:

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

5. Finally, in the host code you need to set up the program, which contains and configures the kernels to be passed to the command queue by the host application. The `clCreateProgramWithBinary` command is used to build the program from the compiled kernel binary (`.xclbin`). The following example shows the creation of the program:

```
char *krnl_bin;
size_t krnl_size;
krnl_size = load_file_to_memory(binaryName, &krnl_bin);

cl_program program = clCreateProgramWithBinary(context, 1,
                     (const cl_device_id* ) &device_id, &krnl_size,
                     (const unsigned char**) &krnl_bin, NULL, &err);
```

TIP: *An FPGA binary file is created containing all of the kernels in the SDAccel application project.*

Transferring Data To/From the SDAccel Platform

With the program established, you can transfer the data required by the kernel to the SDAccel platform prior to triggering the kernel. The simplest way to send data back and forth from the kernel is using `clCreateBuffer`, `clEnqueueReadBuffer`, and `clEnqueueWriteBuffer` commands. However, to transfer the data required ahead of the transaction, use the `clEnqueueMigrateMemObjects` command. Using this command results reduced latency in the application. The following code example demonstrates this:

```
// Move Buffer over input vector
mBlockExt.obj = blocks->data() + mNumBlocks64*64*start;
mQExt.obj     = q->data();
mInBuffer[0] = clCreateBuffer(mContext,
              CL_MEM_EXT_PTR_XILINX | CL_MEM_USE_HOST_PTR |
CL_MEM_READ_ONLY,
             mNumBlocks64*64*sizeof(int16_t),
             &mBlockExt,
             &err);

// Schedule actual writing of data
clEnqueueMigrateMemObjects(mQ, 2, mInBuffer, 0, 0, nullptr,
&inEvVec[mCount]);
```

TIP: *By default, all the memory interfaces from all the kernels are connected to a single global memory bank. However, if the device contains multiple DDR banks, you can customize the DDR bank connections by modifying the default connection. This improves kernel performance by enabling multiple kernels to concurrently read and write data from separate DDR banks. See* Using the --sp Switch to Assign Kernel Interfaces to Memory Banks *for more information.*

Setting up the Kernel

With the program established, you can setup the kernel, execute the kernel, and manage event synchronization between the host application and the kernel.

1. Create a kernel from the program and the loaded FPGA binary using the `clCreateKernel` command:

```
// Create Kernel
cl_kernel krnl = clCreateKernel(program, "krnl_idct", &err);
```

2. Set the kernel arguments using the `clSetKernelArg`. You can use this command to define the scalar and array type arguments for a specific instance of the kernel.

```
// Set the kernel arguments
clSetKernelArg(mKernel, 0, sizeof(cl_mem), &mInBuffer[0]);
clSetKernelArg(mKernel, 1, sizeof(cl_mem), &mInBuffer[1]);
clSetKernelArg(mKernel, 2, sizeof(cl_mem), &mOutBuffer[0]);
clSetKernelArg(mKernel, 3, sizeof(int), &m_dev_ignore_dc);
clSetKernelArg(mKernel, 4, sizeof(unsigned int), &mNumBlocks64);
```

3. The kernel is triggered to run on the FPGA by using the `clEnqueueTask`. The kernel is placed into the command queue and either waits for its turn, or is executed when ready, depending on the nature of the queue.

```
clEnqueueTask(mQ, mKernel, 1, &inEvVec[mCount], &runEvVec[mCount]);
```

4. Because the clEnqueueTask command is asynchronous in nature, and will return immediately after the command is enqueued in the command queue, you may need to manage the scheduling of events within the host application. In order to resolve the dependencies among the commands in the host application, you can use `clWaitForEvents` or `clFinish` commands to pause or block execution of the host program. For example:

```
// Execution waits until all commands in the command queue are finished
clFinish(command_queue);

clWaitForEvents(1, &readevent); // Wait for clEnqueueReadBuffer event
to finish
```

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.

Kernels created from OpenCL C and C/C++ are well suited to software and algorithm developers. It makes it easier to start from an existing C/C++ application and accelerate portions of it.

Writing OpenCL C Kernels

The SDAccel™ environment supports the OpenCL™ C language constructs and built-in functions from the OpenCL 1.0 embedded profile. The following is an example of an OpenCL C 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` cannot be used in the OpenCL C kernel. Use OpenCL built-in C functions instead.*

Writing C/C++ Kernels

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

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

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 HLS C/C++ 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.

***Note*:** 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_`.

1. 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 bit width of the underlying interface is 512 bits or 64 bytes, the recommended size of the struct is 4, 8, 16, 32 or 64 bytes. 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.

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.

For more information about `xocc` command options, refer to the *SDx Command and Utility Reference Guide* (UG1279), *SDAccel Environment Profiling and Optimization Guide* (UG1207), and *SDAccel Environment Programmers Guide* (UG1277).

Writing RTL Kernels

An RTL kernel has both software and hardware requirements to allow it to be used in the SDAccel environment framework. From a software side, the RTL kernel must be designed such that it:

- is capable of starting when called to do so by the host.
- compute all data values.
- returns the data and then ends the operation.

On the hardware side, it requires the following interfaces:

- A single slave AXI4-Lite interface used to access control registers (to pass scalar arguments and to start/stop the kernel).
- At least one of the following interfaces. (Can have both interfaces).

  1. AXI4 master interface to communicate with memory.

  2. AXI4-Stream interface for transferring data between kernels.

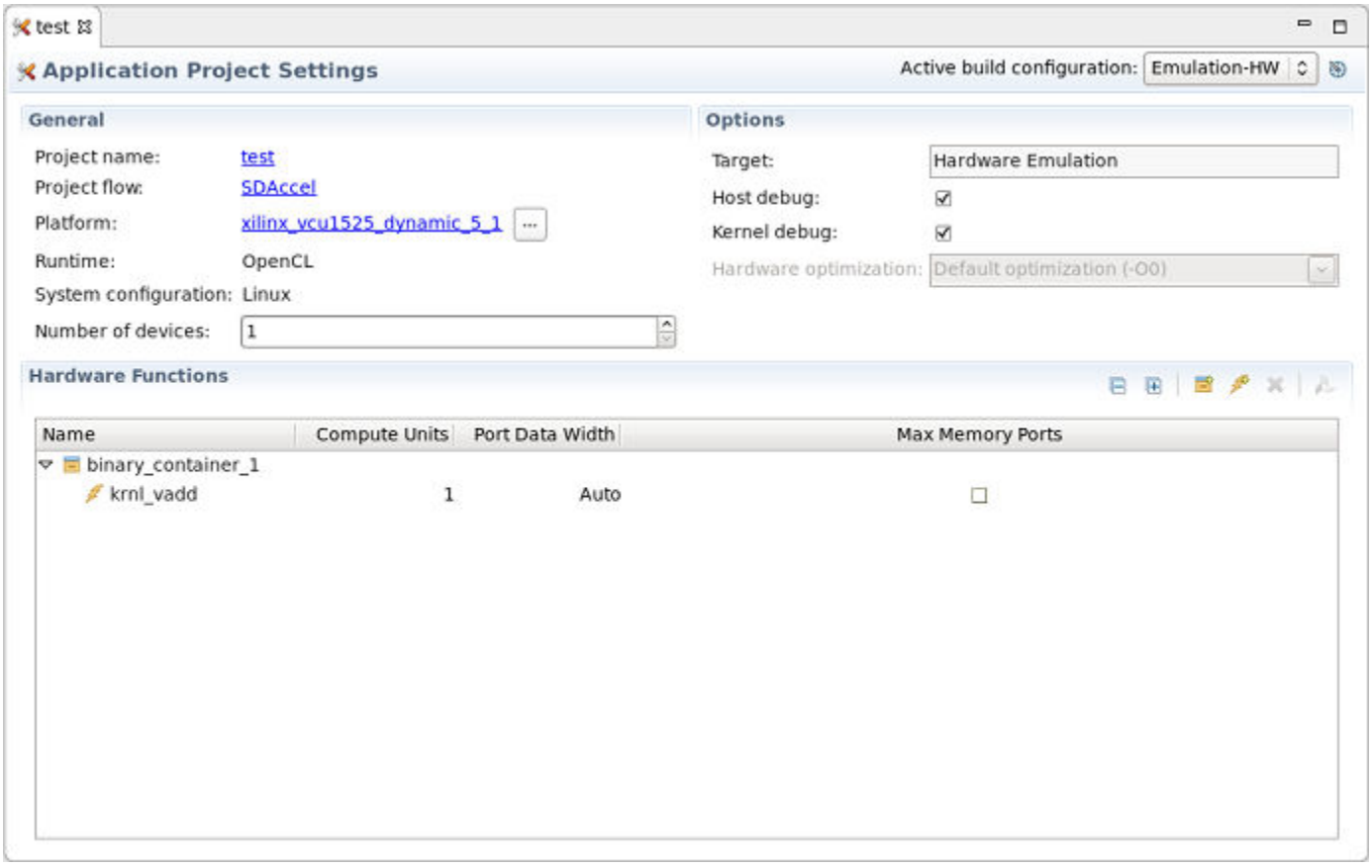
For complete details on creating and using RTL kernels, see Chapter 9: Creating RTL Kernels.

Building the System

Building the system requires building both the hardware (kernels) and the software (host code) side of the system. When a compute unit is instantiated in the host code, the host needs to know how the compute units are connected to the memory and infrastructure elements of the target device.

The Editor pane, shown below, gives a top-level view of the build configuration. It provides general information about the active build configuration, including the project name, current platform, and selected system configuration (OS and runtime). It also displays several build options including the selected build target, and options for enabling host and kernel debugging. For more details on build targets see Build Targets while Chapter 7: Debugging Applications and Kernels gives details on using the debug options.

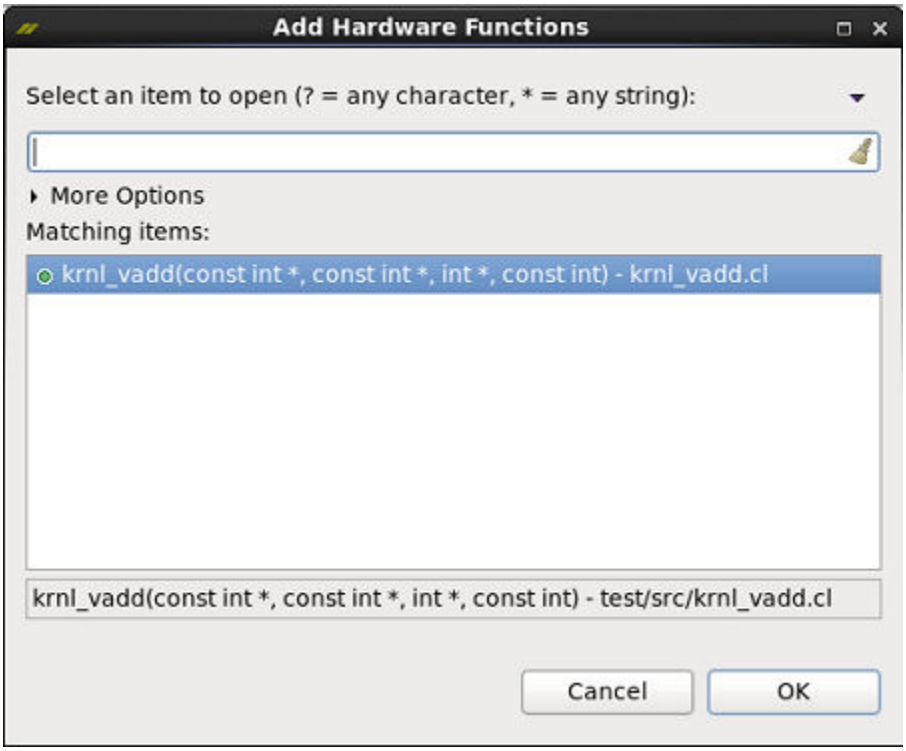
Figure 26: **Project Editor view**



The bottom portion of the Editor lists the current kernels used in the project. The kernels are listed under the binary container. In the above example, the kernel `krnl_vadd` has been added to `binary_container_1`. To add a binary container left-click the icon. You can rename the binary container by left-clicking on the default name and entering a new name.

To add a kernel to the binary container, left-click the icon located in the Hardware Functions window. It will display a list of kernels defined in the project. Select the desired kernel from the Add Hardware Functions dialog box as shown below.

Figure 27: **Adding hardware functions to a binary container**



In the Compute Units column, next to the kernel, enter a value to instantiate multiple instances of the kernel (called compute units) as described in Using the --nk Switch to Instantiate Multiple Kernels.

With the various options of the active build configuration specified, you can start the build process by clicking on the Build ( ) command.

The SDAccel build process generates the host application executable (`.exe`) and the FPGA binary (`.xclbin`). The SDAccel™ environment manages two separate independent build flows:

- Host code (software) build and
- Kernel code (hardware) build

SDAccel uses a standard compilation and linking process for both these software and hardware elements of the project. The steps to build both the host and kernel code to generate the selected build target are described in the following sections.

Host Build

The host code (written in C/C++ using OpenCL™ APIs) is compiled and linked by the Xilinx® C++ (XCPP) compiler and generates a host executable (`.exe` file) which executes on the host CPU.

TIP: *XCPP is based on GCC, and therefore supports many standard GCC options which are not documented here. For information refer to the* GCC Option Index.

Host Compilation

Each host application source file is compiled using the `-c` option that generates an object file (`.o`).

```
xcpp ... -c <file_name1> ... <file_nameN>
```

The name of the output object file can optionally be specified with the `-o` option.

```
xcpp ... -o <outut_file_name>
```

You can produce debugging information using the `-g` option.

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

Host Linking

The generated object files (`.o`) are linked with the Xilinx SDAccel runtime shared library to create the executable (`.exe`). Linking is performed using the -l option.

```
xcpp ... -l <object_file1.o> ... <object_fileN.o>
```

In the GUI flow, the host code and the kernel code are compiled and linked by clicking on the Build () command.

Kernel Build

The kernel code is written in C, C++, or OpenCL™ C, and is built by the Xilinx® OpenCL Compiler (XOCC); a command line utility modeled after GCC. The final output of `xocc` is the generation of the FPGA binary (`.xclbin`) which links the kernel `.xo` files and the hardware platform (`.dsa`). Generation of the `.xclbin` is a two step build process requiring kernel compilation and linking.

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 Getting Started Tutorial* (UG1021) for more information.

Kernel Compilation

During compilation, `xocc` compiles kernel accelerator functions (written in C/C++ or OpenCL language) into Xilinx object (`.xo`) files. Each kernel is compiled into separate `.xo` files. This is the `-c/--compile` mode of `xocc`.

Kernels written in RTL are compiled using the `package_xo` command line utility. This utility, similar to `xocc -c` , also generates `.xo` files which are subsequently used in the linking stage. See Chapter 9: Creating RTL Kernels for more information.

The compilation is dependent on the selected build target, which is discussed in greater detail in Build Targets. You can specify the build target using the `xocc -target` option as shown below.

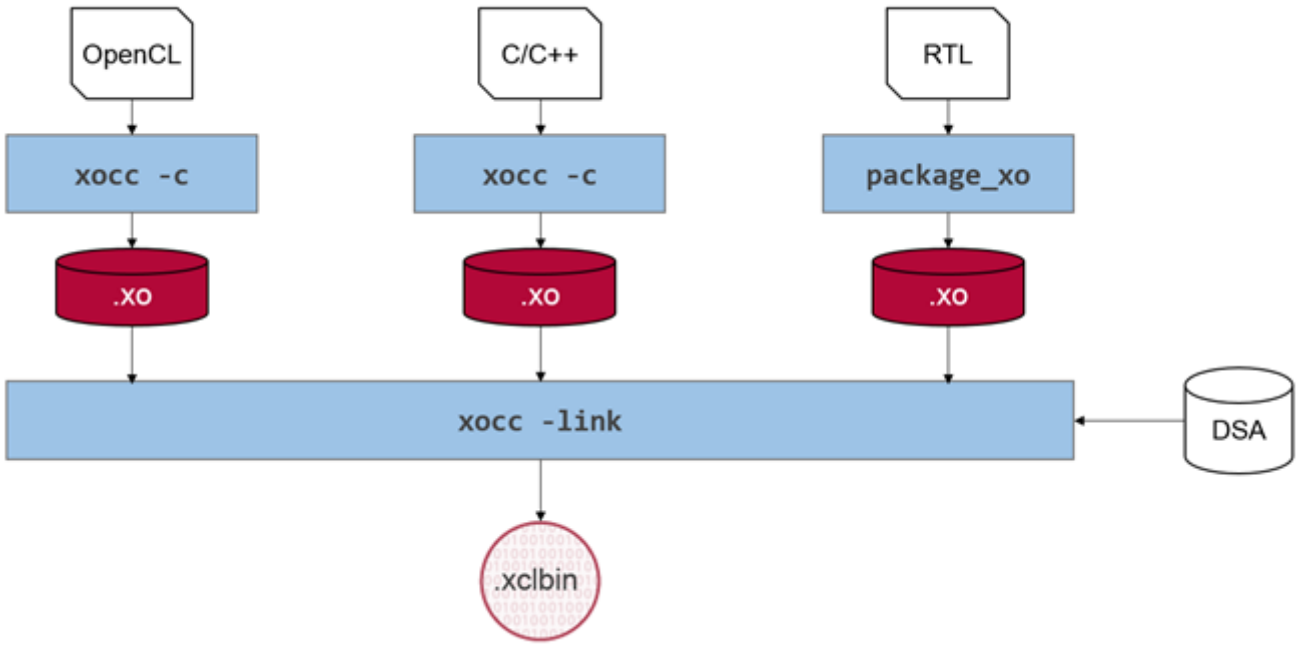
```
xocc --target sw_emu|hw_emu|hw ...
```

- For software emulation (`sw_emu`), the kernel source code is used during emulation.
- For hardware emulation (`hw_emu`), the synthesized RTL code is used for simulation in the hardware emulation flow.
- When the target build is system (`hw`), `xocc` generates the FPGA binary and the system can be run on hardware.

Kernel Linking

As discussed above, the kernel compilation process results in a Xilinx object file (`.xo`) whether the kernel is described in OpenCL, C, C++, or RTL. During the linking stage, `.xo` files from different kernels are linked with the hardware platform (`.dsa`) to create the FPGA binary container file (`.xclbin`) which is needed by the host code. This flow is shown in the figure below.

Figure 28: **SDAccel Linking Process**



X#####-061118

The XOCC command to link files is:

```
xocc ... -l
```

Using the --nk Switch to Instantiate Multiple Kernels

During the linking stage, you can specify the number of kernel instances to be run on the Xilinx® FPGA. Each specified instance of a kernel is referred to as a compute unit.

You can generate multiple hardware instances of the same kernel by specifying the number of instances for each kernel in the XOCC link command. This allows the same kernel function to run in parallel at application runtime to improve the performance of the host application, using different device resources on the FPGA.

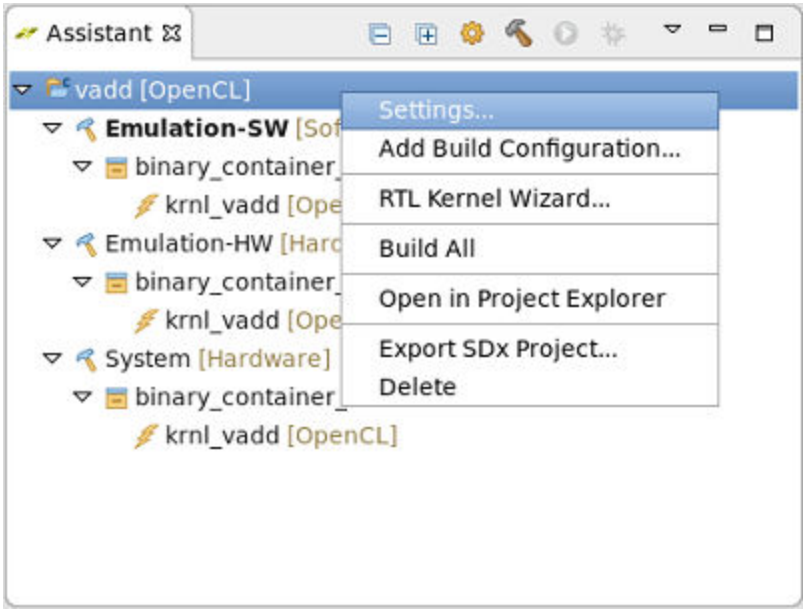
In the command-line flow, the `xocc --nk` option specifies the number of compute units of a given kernel to instantiate into the `.xclbin` file. The syntax of the command is as follows:

```
xocc ... --nk <kernel_name>:<compute_units>:<cu_name1>:…:<cu_nameN>
```

TIP: *While the compute unit instance name is optional, you are recommended to use it as an instance name is required for options like* `--sp`.

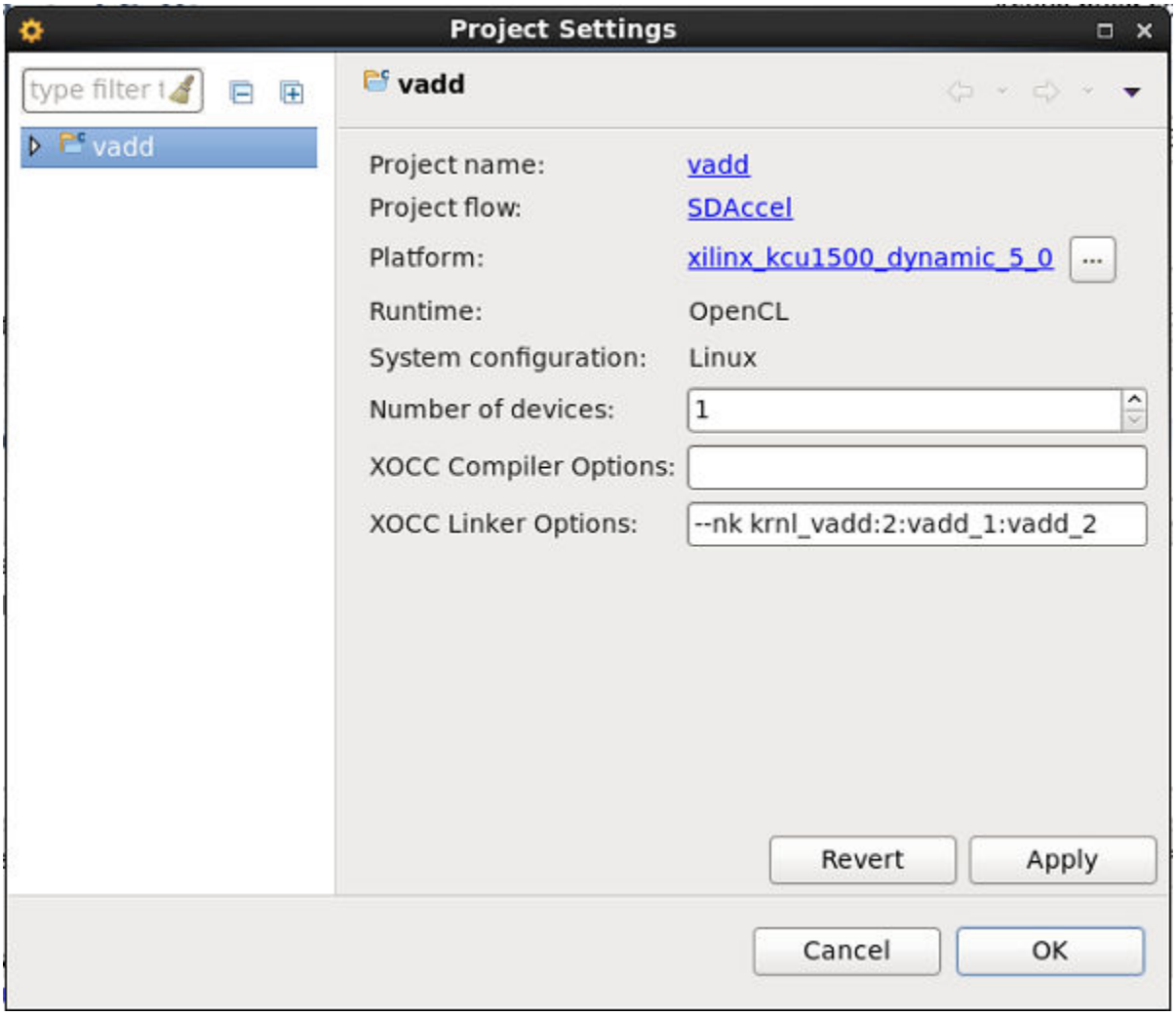
In the GUI flow, the number of compute units can be specified by right-clicking the top-level kernel within the **Assistant** view, and selecting **Settings**.

Figure 29: **Assistant Top-Level**



From within the Project Settings dialog box, enter the `--nk` option in the **XOCC Linker Options** field as shown below.

Figure 30: **Instantiate Multiple Compute Units**



In the figure above, two compute units of the `vadd` kernel will be linked into the FPGA binary (`.xclbin`), addressable as `vadd_1` and `vadd_2`.

Using the --sp Switch to Assign Kernel Interfaces to Memory Banks

During linking, the memory interfaces from all kernels are connected to a single global memory bank through the default kernel compilation process. As a result, only one memory interface can transfer data to and from the DDR bank at one time, limiting the performance of the kernel. If the FPGA contains only one global memory bank, this is the only option. However, if the device contains multiple banks, you can customize the memory bank connections by modifying the default connection.

Global memory is the DDR memory accessible by a platform. SDAccel™ platforms can have access to multiple global memory banks, and you can specify connections from the kernel interfaces to global memory. Even if there is only one compute unit instantiated, by assigning the input to DDR1, and the output to DDR2, for instance, you can accelerate the performance of your kernel and host application. In applications with multiple kernel instances running concurrently, this can result in significant performance gains.

Reassigning the default memory bank access between the host application and the kernel, requires you to take the following steps:

1. In the host application, allocate buffers to specific global memory banks when they are created.
2. In the kernel, bundle kernel arguments associated with a specific memory bank into a single interface.
3. During XOCC linking, use the `--sp` option to assign the kernel interface to the proper memory bank.

Details of coding the host application can be found in the *SDAccel Environment Programmers Guide* (UG1277), in "Memory Data Transfer To/From the FPGA Device." In short, you must assign the buffers to specific global memory banks at the time the buffer is created, using the `cl_mem_ext_ptr_t` type.

In the kernel interface description, HLS INTERFACE pragmas are used to denote the interfaces coming to and from the global memory banks. These pragmas dictate how the memory interfaces are mapped to the global memory banks, as shown in the following code example:

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

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

Refer to "Memory Data Inputs and Outputs" in *SDAccel Environment Programmers Guide* (UG1277) for more information.

Finally, during XOCC linking, in the command-line flow the `xocc --sp` option assigns the kernel interfaces to available memory banks. The syntax of the option is as follows:

```
xocc --sp <cu_name>.<interface>:<bank_num>
```

Where:

- `cu_name`: The name of the compute unit on the FPGA. This is the name of the kernel, or the name of the compute unit used in the `--nk` option.

- `interface`: The name of the interface on the kernel function, as defined by pragmas in the kernel code.
- `bank_num`: Global memory bank #, expressed as bank0, bank1, bank2, and bank3 for platforms with four DDR banks.

The `--sp` switch can be added through the SDx GUI as discussed previously in Using the --nk Switch to Instantiate Multiple Kernels. Right-click the top-level kernel in the **Assistant** view, and select **Settings**. From within the Project Settings dialog box, enter the `--sp` option in the XOCC Linker Options field.

Using the --xp Switch to Optimize Results

When compiling or linking, fine grain control over the hardware generated by SDAccel™ for hardware emulation and system builds can be specified using the `--xp` switch.

The `--xp` switch is paired with parameters to configure the Vivado® Design Suite. For instance, the `--xp` switch can configure the optimization, placement and timing results of the hardware implementation.

The `--xp` can also be used to set up emulation and compile options. Specific examples of these parameters include setting the clock margin, specifying the depth of FIFOs used in the kernel dataflow region, and specifying the number of outstanding writes and reads to buffer on the kernel AXI interface. A full list of the parameters can be found in the *SDx Command and Utility Reference Guide* (UG1279)

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

In the command-line flow, the `xocc --xp` option is specified as follows:

```
xocc --xp param:compiler.enableDSAIntegrityCheck=true
```

tool suite is necessary to make the most use of these parameters.Parameters are specified as `param:<param_name>=<value>`, which includes:

- `param` keyword.
- `param_name`: The name of a parameter to apply.
- value: An appropriate value for the parameter.

IMPORTANT!: *The XOCC linker does not check the validity of the parameter or value. You must be careful to apply valid values or the downstream tools may not work properly.*

For example:

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

You must repeat the `--xp` switch for each `param` used in the XOCC command, or specify the value(s) in an `xocc.ini` file with each option specified on a separate line (without the `--xp` switch). For example:

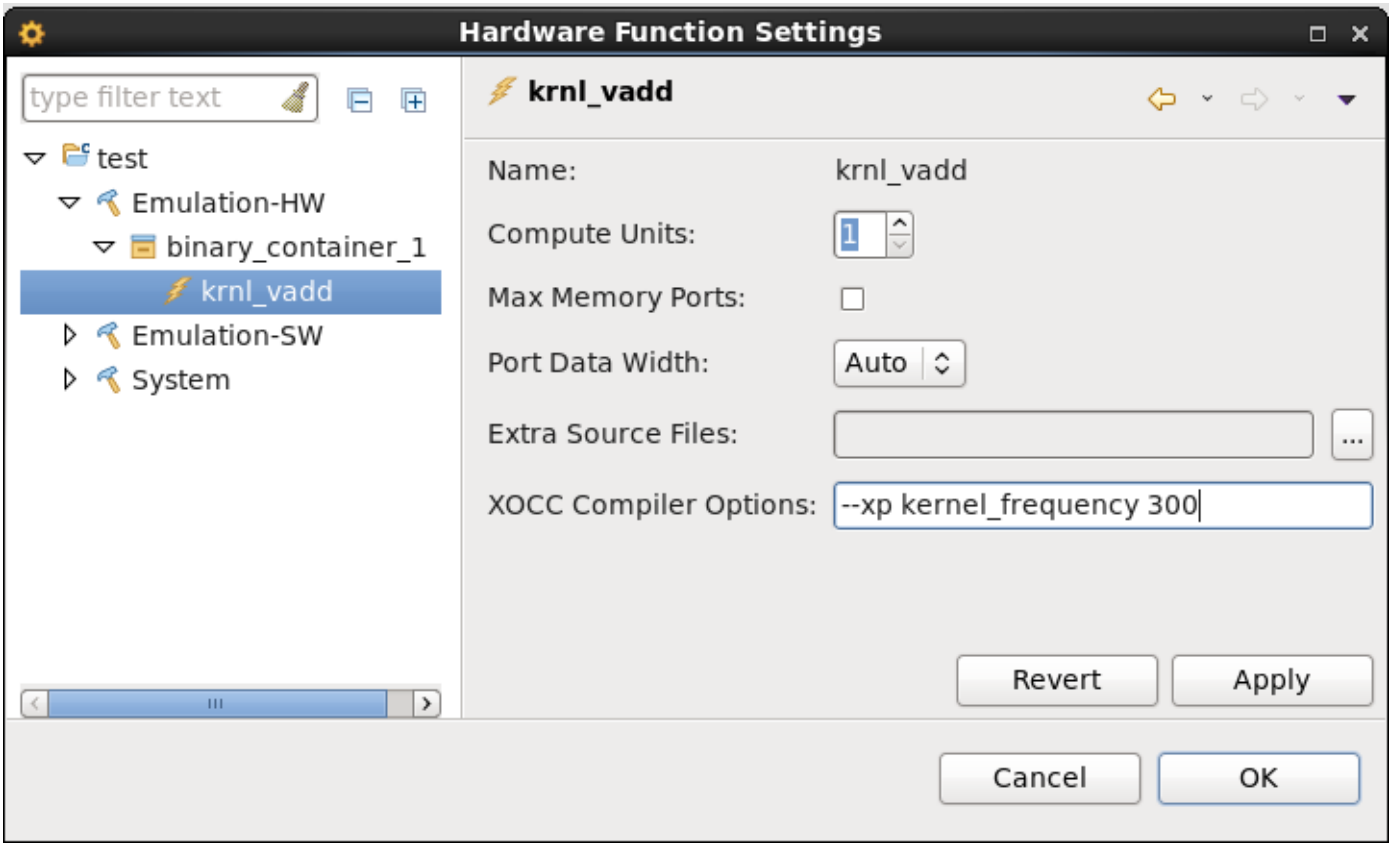
```
param:compiler.enableDSAIntegrityCheck=true
param:prop:kernel.foo.kernel_flags="-std=c++0x"
```

An `xocc.ini` is an initialization file that contains `--xp` settings. It should be located in the same directory as the build configuration. Under the GUI flow, if no `xocc.ini` is present, it will use the GUI build settings. Under a `Makefile` flow, if no `xocc.ini` file is present, it will use the configurations within the Makefile.

The `--xp` switch can be added through the SDx GUI as discussed previously in Using the --nk Switch to Instantiate Multiple Kernels. Right-click the top-level kernel in the **Assistant** view, and select **Settings**. From within the Project Settings dialog box, enter the `--xp` option in the **XOCC Linker Options** field.

You can also add XOCC compiler options and `--xp` parameters to kernels by right-clicking the kernel in the Assistant view. The following image shows the settings for the *krnl_vadd* kernel.

Figure 31: **Assistant XOCC Compile Settings**



Using -R to Control Report Generation

The XOCC `-R` switch controls the level of report generation during the link stage for hardware emulation and system targets. Builds that generate fewer reports will typically run more quickly.

The command -line option is as follows:

```
xocc -R<report_level>
```

Where:

- -R0 – Minimal reports and no intermediate DCPs
- -R1 – Includes R0 reports plus:
- ◦ Identifies design characteristics to review for each kernel (`report_failfast`)
  - ◦ Identifies design characteristics to review for full design post-opt (`report_failfast`)
  - ◦ Saves post-opt DCP
- -R2 – Includes R1 reports plus:

- Vivado® default reporting including DCP after each implementation step
- Identifies design characteristics to review for each SLR after placement (`report_failfast`)

TIP: *The `report_failfast` is a utility which highlights potential device utilization challenges, clock constraint problems and potential unreachable target frequency (MHz).*

The `-R` switch can also be added through the SDx GUI as discussed previously in Using the --nk Switch to Instantiate Multiple Kernels. Right-click the top-level kernel in the **Assistant** view, and select **Settings**. From within the Project Settings dialog box, enter the `-R` option in the **XOCC Linker Options** field.

Build Targets

The SDAccel build target defines the nature of FPGA binary generated by the build process. There are three different build targets, two emulation targets (software and hardware emulation) used for debug and validation purposes and the default hardware target used to generate the actual FPGA binary.

Software Emulation

The main goal of software emulation is to ensure functional correctness and to partition the application into kernels. For SW emulation, both the host code and the kernel code are compiled to run on the host x86 processor. The programmer model of iterative algorithm refinement through fast compile and run loops is preserved. SW emulation has compile and execution times that are the same as a CPU. Refer to the *SDAccel Environment Debugging Guide* (UG1281) for more information on running software emulation.

In the context of the SDAccel development environment, software 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.

For RTL kernels, software emulation can be supported if a C model is associated with the kernel. The RTL kernel wizard packaging step provides an option to associate C model files with the RTL kernel for support of software emulation flows.

Hardware Emulation

While the Software Emulation flow is a good measure of functional correctness, it does not guarantee correctness on the FPGA execution target. The Hardware Emulation flow enables the programmer to check the correctness of the logic generated for the custom compute units before deployment on hardware, where a compute unit is an instantiation of a kernel.

The SDAccel™ Environment generates at least one custom compute unit for each kernel in an application. Each kernel is compiled to a hardware model (RTL). During emulation kernels are executed with a hardware simulator, but the rest of the system still uses a C simulator. This allows the SDAccel environment to test the functionality of the logic that will be executed on the FPGA compute fabric.

In addition, hardware emulation provides performance and resource estimation, allowing the programmer to get an insight into the design.

In hardware emulation, compile and execution times are longer in software emulation; thus Xilinx recommends that you use small data sets for debug and validation.

IMPORTANT!: *The DDR memory model and the Memory Interface Generator (MIG) model used in Hardware Emulation are high-level simulation models. These models are good for simulation performance, however they approximate latency values and are not cycle-accurate like the kernels. 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.*

System

When the build target is system, xocc generates the FPGA binary for the device by running synthesis and implementation on the design. The binary includes 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 build flow. However, because the kernels will be running on actual hardware, their execution times will be extremely fast.

The generation of custom compute units uses the Vivado® High-Level Synthesis (HLS) tool, which is the compute unit generator in the application compilation flow..SDAccel Environment. 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 and performs default connections from the kernel to global memory. It is recommended that you explicitly define optimal connections. See* Appendix C: SLR Assignments for Kernels *for details.*

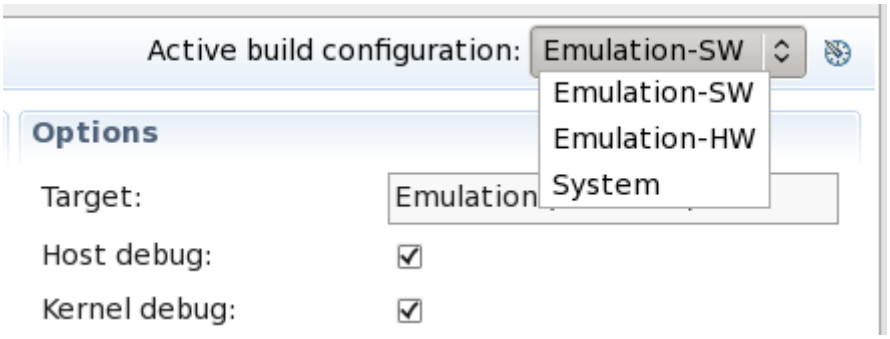
Specifying a Target

You can specify the target build from the command-line with the following command:

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

Similarly, from within the GUI, the build target can be specified by selecting the **Active build configuration** pull-down tab in the Project Editor window. This provides three choices: Emulation-SW, Emulation-HW and System as shown in the following figure.

Figure 32: **Active Build Configuration**



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

After setting the active build configuration, build the system from the Project > Build Project menu command.

The recommended build flow is detailed in Debugging Flows.

Profiling and Optimization

The SDAccel™ environment generates various system and kernel resource performance reports during compilation. It also collects profiling data during application execution in both emulation and system mode configurations. Examples of the data reported includes:

- Host and device timeline events
- OpenCL™ API call sequence
- Kernel execution sequence
- FPGA trace data including AXI transactions
- Kernel start and stop signals

Together the reports and profiling data can be used to isolate performance bottlenecks in the application and optimize the design to improve performance.

Optimizing an application requires optimizing both the application host code and any hardware accelerated kernels. The host code must be optimized to facilitate data transfers and kernel execution, while the kernel should be optimized for performance and resource usage.

There are four distinct areas to be considered when performing algorithm optimization in SDAccel: System resource usage and performance, Kernel optimization, Host optimization and PCIe® bandwidth optimization. The following SDAccel reports and graphical tools support your efforts to profile and optimize these areas:

- System Estimate
- Design Guidance
- HLS Report
- Profile Summary
- Application Timeline
- Waveform View and Live Waveform Viewer

Reports are automatically generated after running the active build via the SDAccel GUI or xocc/ Makefile flows.

Separate sets of reports are generated for all three build configurations and can be found in the respective report directories.

IMPORTANT!: *The high level synthesis (HLS) report and HLS guidance are only generated for hardware emulation and system build configurations for C and OpenCL kernels, not for RTL kernels.*

The Profile Summary and Application Timeline reports are generated for all three build configurations and are located under the default application sub-directory.

Reports can be viewed in a web browser or spreadsheet viewer for the SDAccel GUI. To access these reports from the SDx™ integrated design environment, make sure the Assistant view is visible and double-click the desired report.

This following sections briefly describe the various reports and graphical visualization tools, and how they can be used to profile and optimize your design. For complete details on each report along with optimization steps, and coding guidelines see the *SDAccel Environment Profiling and Optimization Guide* (UG1207)

Design Guidance

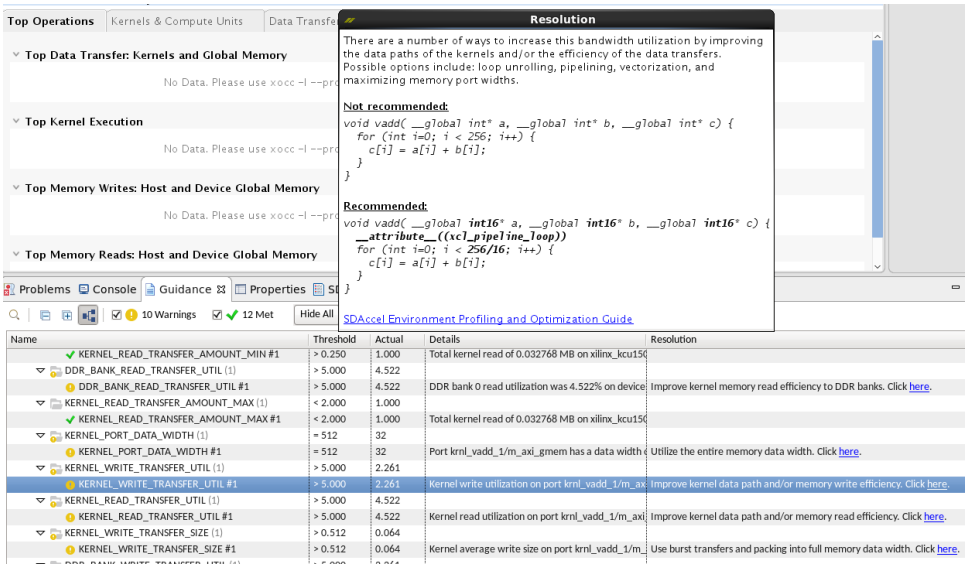
The SDAccel™ environment has a comprehensive design guidance tool that provides immediate actionable guidance to the software application developers for detected issues in their designs. Guidance is generated from HLS, the SDx™ Profiler and Vivado® when invoked from `xocc`. The generated design guidance can have several severity levels; errors, advisories, warnings and critical warnings are provided during software and hardware emulation and system builds.

The guidance includes hyperlinks, examples, and links to documentation. This improves productivity for current users by quickly highlighting issues and propels new users to more quickly become experts in using SDAccel.

Design guidance is automatically generated after building or running a design in the SDx™ GUI with results contained in the Guidance view located in the console area of the SDx GUI. Hovering over the guidance will highlight solutions and suggestions.

The following image shows an example of guidance given by the SDx GUI. It details ways to increase the bandwidth use of the kernels. Clicking a link displays an expanded view of the actionable guidance. In this case, it displays guidance for maximizing use of global memory bandwith.

Figure 33: **Design Guidance Example**



TIP: *In the Assistant you can right-click on a build configuration and select* ***Show Guidance****.*

To enable the design guidance tool via the command line, add the following parameter to the `xocc` call.

```
--xp param:compiler.useSdxRuleService=true
```

There will be one HTML guidance report for each command line run of `xocc`, including compile and link. The report files are generated in the `--report_dir` location under the specific XO name.

The name of the report file is given below, where t is the XO name:

- `xocc_compile_t_guidance.html` for `xocc` compilation or
- `xocc_link_t_guidance.html` for `xocc` linking

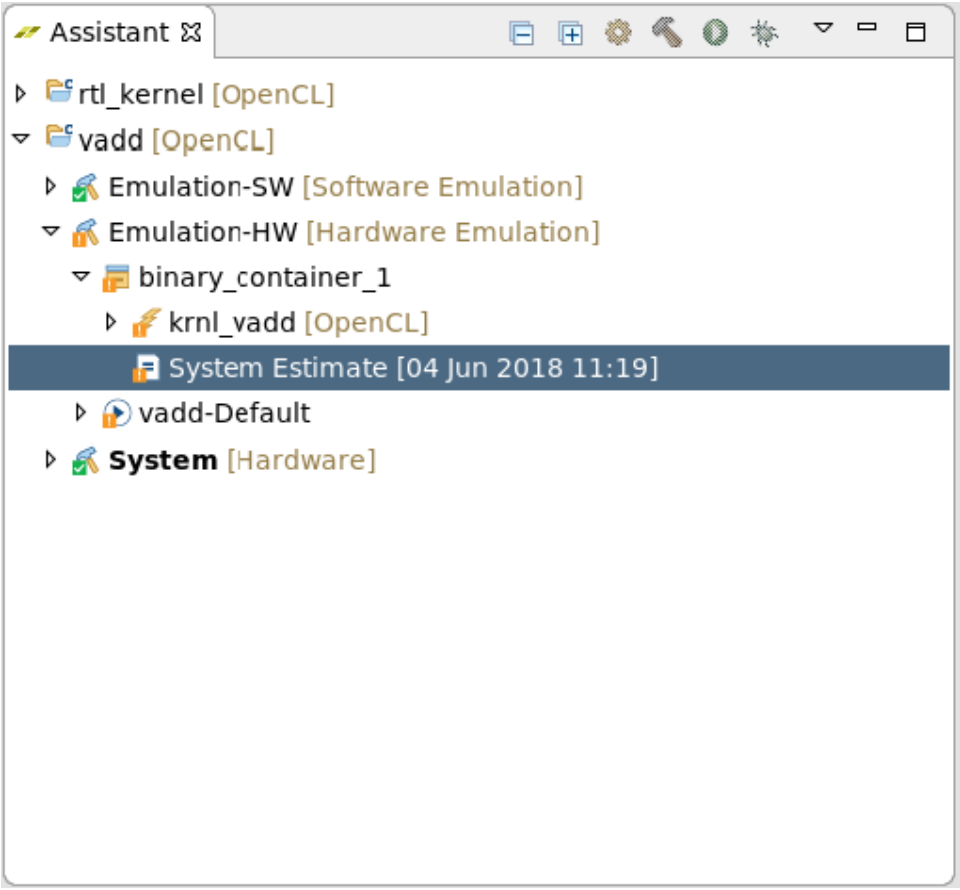
The profile design guidance helps you interpret the profiling results and know exactly where to focus on to improve performance. Specific details of the reports and additional design guidance details can be found in *SDAccel Environment Profiling and Optimization Guide* (UG1207).

Estimating Performance

The System Estimate report will only be available for C/C++ and OpenCL™ kernels because it is generated by Vivado® HLS, not RTL kernels. The report provides estimates on FPGA resource usage and the frequency at which the hardware accelerated kernels can operate. It is automatically generated for Emulation-HW and System builds, and can be found under the respective directory of the Assistant view.

TIP: *The time to generate the System Estimate report in Hardware Emulation build is much shorter than during System builds which provide actual and not estimated resources. Thus it is recommended to iterate in Hardware Emulation and optimize before performing a System build.*

Figure 34: **System Estimate Assistant View**

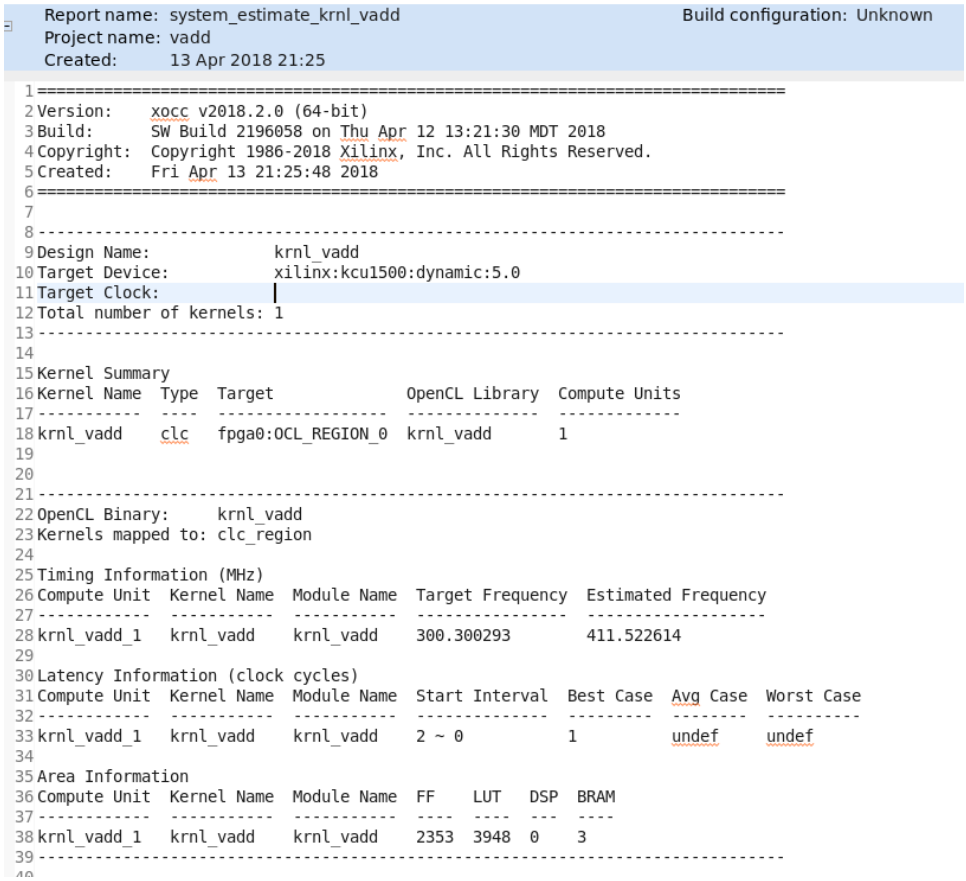


The report contains high-level details of the user kernels including resource usage and estimated frequency. The results can be used to guide the design optimization. For instance, if the target frequency is not met, it might be necessary to revisit the source code.

An example report is shown in the following graphic. It shows the *krnl_vadd* kernel:

- It is estimated to operate at a frequency of 411 MHz which exceeds the 300 MHz targeted frequency.
- In the best case it has a latency of one cycle.
- Estimated FPGA resource usage of 2353 FF, 3948 LUTS, no DSPs and 3 BRAMs.

Figure 35: **System Estimate**



```
Report name:  system_estimate_krnl_vadd                    Build configuration:  Unknown
Project name: vadd
Created:      13 Apr 2018 21:25

================================================================================
Version:    xocc v2018.2.0 (64-bit)
Build:      SW Build 2196058 on Thu Apr 12 13:21:30 MDT 2018
Copyright:  Copyright 1986-2018 Xilinx, Inc. All Rights Reserved.
Created:    Fri Apr 13 21:25:48 2018
================================================================================

--------------------------------------------------------------------------------
Design Name:             krnl_vadd
Target Device:           xilinx:kcu1500:dynamic:5.0
Target Clock:
Total number of kernels: 1
--------------------------------------------------------------------------------

Kernel Summary
Kernel Name  Type  Target              OpenCL Library  Compute Units
-----------  ----  ------------------  --------------  -------------
krnl_vadd    clc   fpga0:OCL_REGION_0  krnl_vadd       1


--------------------------------------------------------------------------------
OpenCL Binary:     krnl_vadd
Kernels mapped to: clc_region

Timing Information (MHz)
Compute Unit  Kernel Name  Module Name  Target Frequency  Estimated Frequency
------------  -----------  -----------  ----------------  -------------------
krnl_vadd_1   krnl_vadd    krnl_vadd    300.300293        411.522614

Latency Information (clock cycles)
Compute Unit  Kernel Name  Module Name  Start Interval  Best Case  Avg Case  Worst Case
------------  -----------  -----------  --------------  ---------  --------  ----------
krnl_vadd_1   krnl_vadd    krnl_vadd    2 ~ 0           1          undef     undef

Area Information
Compute Unit  Kernel Name  Module Name  FF    LUT   DSP  BRAM
------------  -----------  -----------  ----  ----  ---  ----
krnl_vadd_1   krnl_vadd    krnl_vadd    2353  3948  0    3
--------------------------------------------------------------------------------
```

When using the command line flow, you can generate the system estimate report with the following option:

```
xocc .. --report estimate
```

For additional details on the System Estimate report see the *SDAccel Environment Profiling and Optimization Guide* (UG1207)

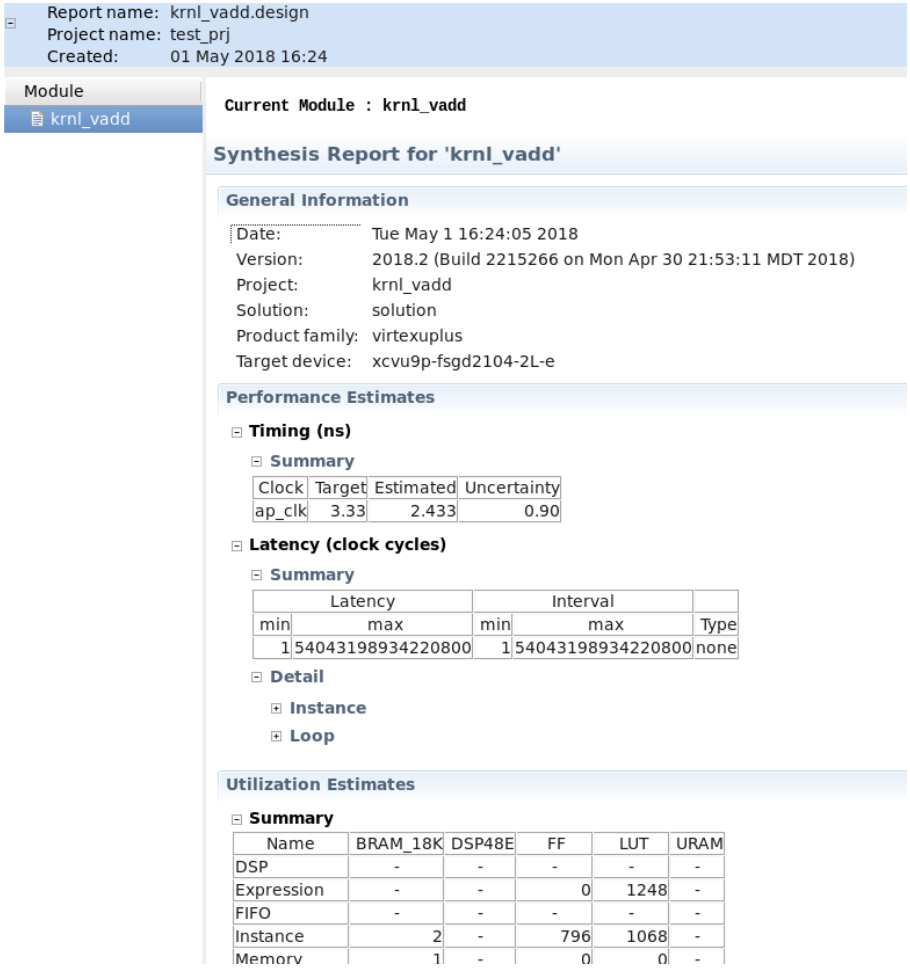
HLS Report

The HLS Report provides details about the High-Level Synthesis process (HLS) of a user kernel and is generated in Hardware emulation and System builds. This process translates the C/C++ and C kernel into a hardware description language responsible for implementing the functionality on the FPGA. It provides estimated FPGA resource usage, operating frequency, latency and interface signals of the custom-generated hardware logic. These details provide the programmer many insights to guide kernel optimization.

The HLS Report can be opened by selecting the report in the Assistant and double-clicking.

An example of the HLS report follows.

Figure 36: **HLS Report**



When running from the command line, this report can be found in the following directory:

```
_x/<kernel_name>.<target>.<platform>/<kernel_name>/<kernel_name>/
solution/syn/report
```

For additional details on the System Estimate report see the *SDAccel Environment Profiling and Optimization Guide* (UG1207).

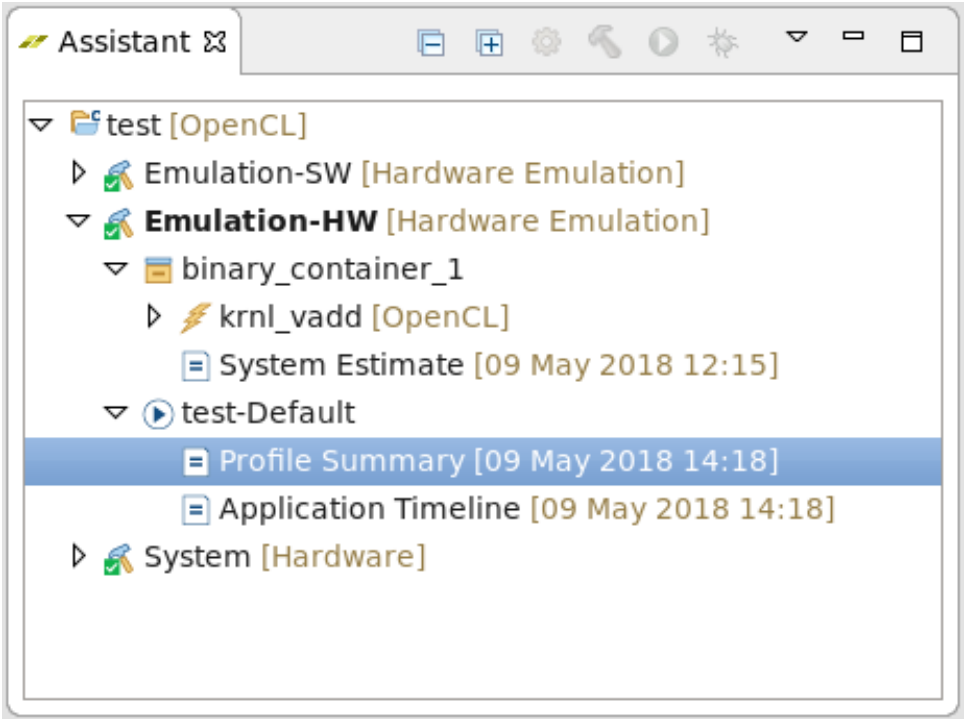
Profile Summary Report

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

TIP: *The Profile Summary report is automatically generated for all build configurations. However, with the Emulation-SW build, the report will not include any data transfer details under kernel execution efficiency and data transfer efficiency. This information is only generated in Emulation-HW or System build configurations.*

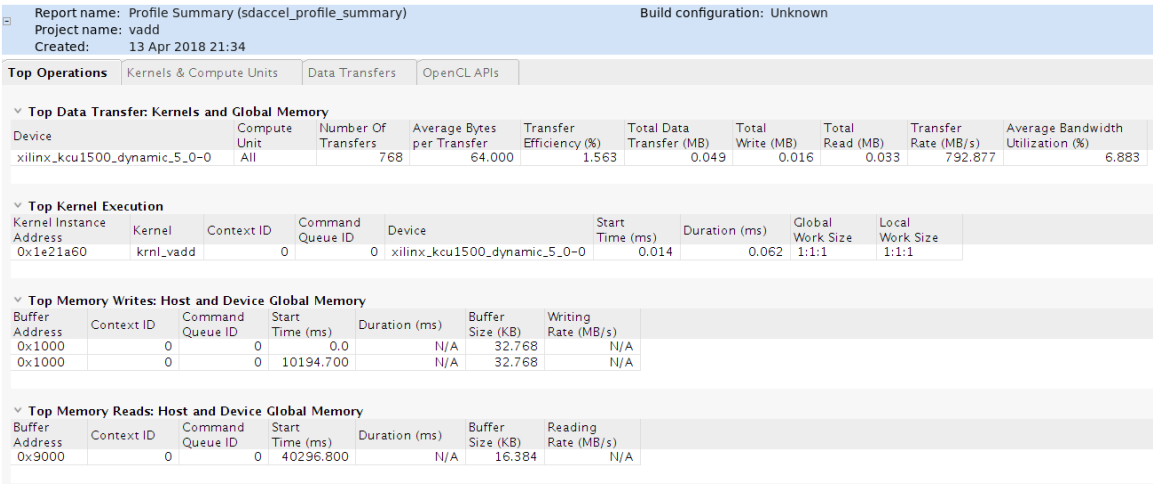
To open the Profile Summary report in the SDx™ IDE, double-click the Profile Summary report under the Assistant as shown in the following image.

Figure 37: **Opening Profile Summary Report**



The report opens. An example of the Profile Summary report is shown below.

Figure 38: **Profile Summary**



The report has multiple tabs that can be selected. A description of each tab is given in the following table.

Table 3: **Profile Summary**

Tab	Description
Top Operations	Kernels and Global Memory. This tab shows a summary of top operations. It displays the profile data for top data transfers between FPGA and device memory.
Kernels & Compute Units	Displays the profile data for all kernels and compute units.
Data Transfers	Host and Global Memory. This table displays the profile data for all read and write transfers between the host and device memory via the PCIe® link. It also displays data transfers between kernels and global memory, if enabled.
OpenCL APIs	Displays the profile data for all OpenCL™ C host API function calls executed in the host application.

For command line users, ensure the profile summary data is generated by using the `-g` option during compilation of the kernel:

```
xocc .. -g
```

The profile summary is affected by `-g` option only for hardware emulation. For System runs, `--profile_kernel` is needed at compile time for stalls. Link time setting of `--profile-kernel` enables device data transfers and compute unit utilization tables for profile summary. Runtime setting of `profile=true` enables profile in runtime.

Subsequently link the bitstream file (`xclbin`) with the `--profile_kernel` option:

```
xocc -g -l --profile_kernel data:all:all:all ...
```

See the *SDAccel Environment Profiling and Optimization Guide* (UG1207) for specific details on the report and command line flow.

Application Timeline

Application Timeline collects and displays host and device events on a common timeline to help you understand and visualize the overall health and performance of your systems. These events include:

- OpenCL™ API calls from the host code.
- Device trace data including Compute units, AXI transaction start/stop.
- Host events and kernel start/stops.

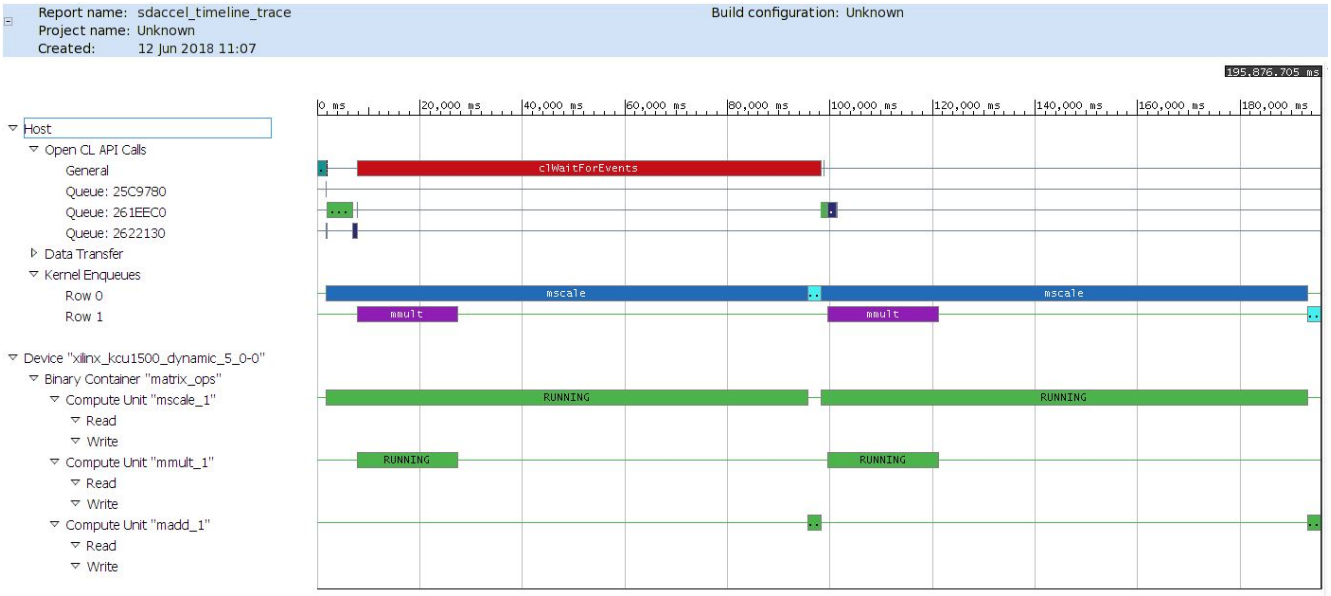
This graphical representation enables the programmer to identify issues regarding kernel synchronization and efficient concurrent execution.

TIP: *By default, timeline and device trace data are only collected during hardware emulation and not System. Turning on device profiling is intrusive and can negatively affect overall performance. This feature should be used for system performance debugging only. To collect data during system testing, update the run config setting. Details can be found in the SDAccel Environment Profiling and Optimization Guide (UG1207).*

Double-click **Application Timeline** in the Reports window to open the Application Timeline window.

Following is a snapshot of the Application Timeline window that displays host and device events on a common timeline. The host events include creating the program, running the kernel, and host read/write access to the (double data rate) DDR. This information helps you to understand details of application execution and identify potential areas for improvements.

Figure 39: **Application Timeline**



X21073-061418

Specifically, it shows:

- **OpenCL Calls:** When host OpenCL call occurs
- **Duration of data transfers:** Start, stop, and duration of data transfers
- **Kernel Execution:** Execution time of the kernel

Timeline data can be enabled and collected via the command line flow, however, viewing must be done via the GUI. Follow these instructions for enabling timeline data collection in the Command Line Flow. Details can be found in the *SDAccel Environment Profiling and Optimization Guide* (UG1207).

For system compilation and linking you need to use the `-profile_kernel` option otherwise it is ignored. During hardware emulation, this data is generated by default.

```
xocc … -profile_kernel
```

It is then necessary to enable data gathering during runtime by adding the following lines to the `sdaccel.ini` file which is located in the same directory as the host executable.

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

With these additions, trace data are captured in a `.csv` file during application execution. The `.csv` report needs to be converted to the Application Timeline format using the `sdx_analyze` utility before they can be opened and displayed in the SDAccel GUI.

```
sdx_analyze trace sdaccel_timeline_trace.csv
```

This creates the `sdaccel_timeline_trace.wdb` file by default, which can be opened via the GUI. To visualize the timeline report start the SDx IDE and select a workspace as described in Using an SDx Workspace

Select **File → Open File**, browse to the `.wdb` file generated during hardware emulation or system run, and open it.

Waveform View and Live Waveform Viewer

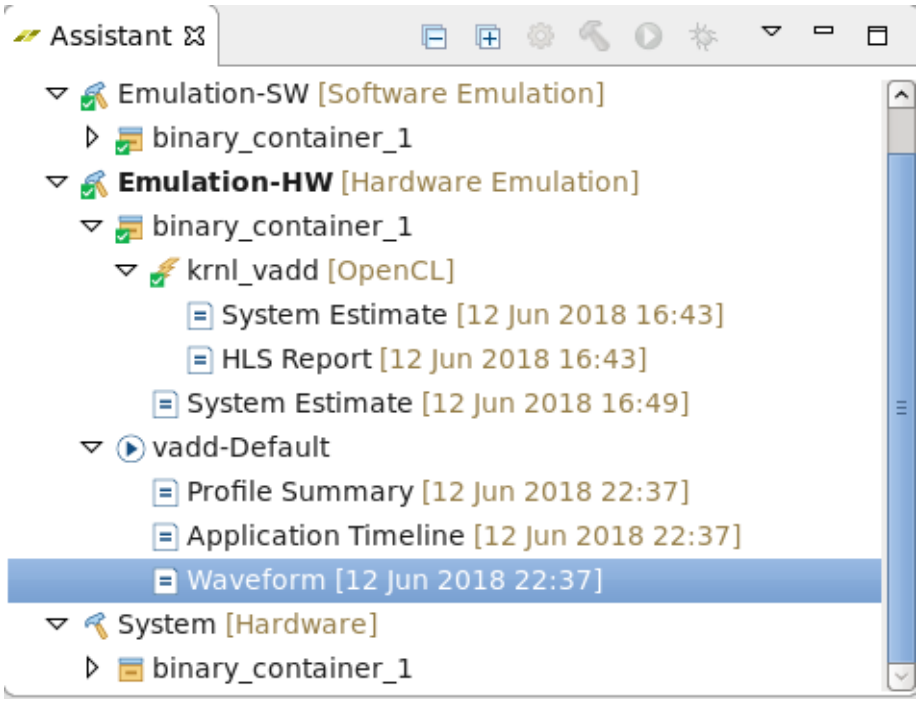
The SDx™ Development Environment can generate a Waveform View when running hardware emulation. It displays in-depth details on the emulation results at the system level, compute unit (CU) level, and at the function level. The details include data transfers between the kernel and global memory and data flow via inter-kernel pipes.These details provide many insights into the performance bottleneck from the system level down to the individual function call to help developers optimize their applications.

The Live Waveform Viewer is similar to the Waveform view, however, it provides even lower-level details. It can also be opened using `xsim`, a Xilinx tool used by hardware designers.

Waveform View and Live Waveform Viewer data are not collected by default because it requires the run time to generate simulation waveform during hardware emulation, which consumes more time and disk space. The *SDAccel Environment Profiling and Optimization Guide* (UG1207) describes setups required to enable data collection for the Waveform View and Live Waveform Viewer for both GUI and command line.

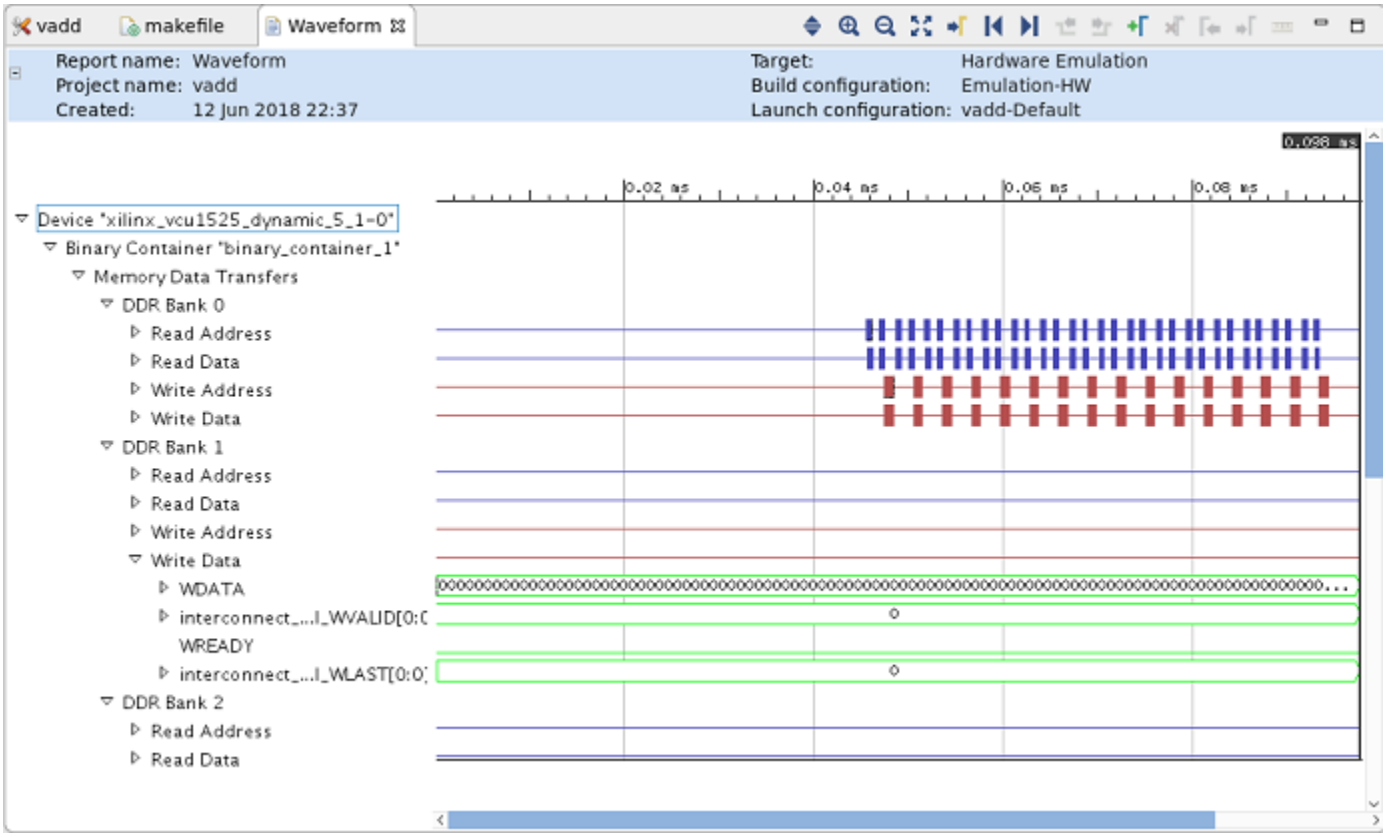
Double click the **Waveform** in the Assistant window (shown in the following image) to open the Waveform View window.

Figure 40: **Opening Waveform View**



An example of the Waveform View is shown below.

Figure 41: **Waveform View Example**

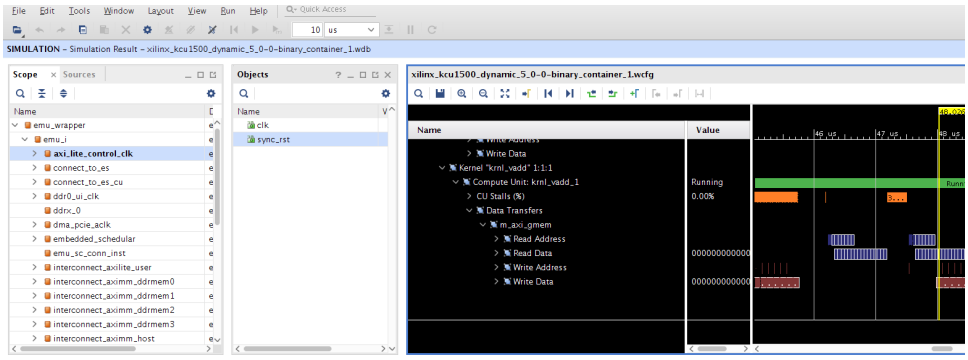


The Live Waveform Viewer can be viewed if the Launch live waveform is selected in the Run Configuration Main tab. Or if the Launch live waveform is not selected, you can open the waveform (`.wdb`) via `xsim` through the Linux command line. The `.wdb` file will be located in the sub-directory Emulation-HW/vadd-Default within the project directory. Use the following Linux line command to open `xsim` (your filename may differ):

```
xsim -gui xilinx_kcu1500_dynamic_5_0-0-binary_container_1.wdb &
```

An example of the `xsim` Live Waveform Viewer is shown in the following image.

Figure 42: **Detailed Kernel Trace View**



Debugging Applications and Kernels

The SDAccel™ environment provides application-level debug features and techniques that allow the host code, the kernel code, and the interactions between them to be debugged efficiently. These features and techniques are split between software-centric and more detailed low-level hardware centric flows.

In addition, for hardware-centric debugging, designs running on hardware can be debugged via both PCIe® (using Xilinx® Virtual Cable, XVC) and JTAG (via USB-JTAG cables) without changing the design.

Debugging Features and Techniques

There are several features and techniques that you can use to debug your design. The following table lists the features or techniques that can be used for debugging for the three different build configurations. Each feature and technique are described in more detail in the following table.

Table 4: **Features and Techniques for Debugging Different Build Configurations**

Feature/ Technique	OS	Host	Kernel	FPGA (Platform)
Software Emulation	dmesg	GDB	GDB	xbsak
Hardware Emulation	dmesg	GDB	GDB Kernel Waveform Viewer	xbsak
System	dmesg	GDB	Kernel Waveform Viewer ILA MicroBlaze™ Debug	xbsak

These features and techniques can be divided into software and hardware-centric debugging features as shown in the following table.

Table 5: **Software and Hardware Debugging Features and Techniques**

Software-centric	Hardware-centric
GNU Debugger (GDB)	Kernel waveform viewer
Xbsak	Integrated Logic Analyzer (ILA)
Linux dmesg	MicroBlaze Debugger

Together you can use them to isolate and debug issues from functional all the way to protocol and board hangs.

Debugging Flows

The recommended application-level debugging flow consists of three levels of debugging:

- Perform software emulation (`sw_emu`:) to confirm the algorithm functionality.
- Perform hardware emulation (`hw_emu`) to create custom hardware and confirm the functionality and performance on FPGAs.
- Perform a System build (hardware) `hw` to implement the custom hardware.

Each provides specific insights into the design and makes debugging easier. All flows are supported through an integrated GUI flow as well as through a batch flow using basic compile time and run-time setup options. A brief description of each flow follows.

Software Emulation

Software emulation can be used to validate functional correctness of the host and kernel (written in C/C++ or OpenCL™). The GDB can be used to debug both the host and kernel code. 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.

Hardware Emulation

Hardware Emulation can be used to validate the host code, profile host and kernel performance, give estimated FPGA resource usage as well as verify the kernel using an accurate model of the hardware (RTL). The execution time for hardware emulation takes more time than software emulation; thus Xilinx recommends that you use small data sets for debug and validation. Again, the GDB can be used to debug the host and kernels. Iterate in Hardware Emulation until the functionality is correct and the estimated kernel performance is sufficient (See the *SDAccel Environment Profiling and Optimization Guide* (UG1207) for optimization details).

System

Finally, in hardware execution (System) the complete system is validated on actual hardware (to ensure kernels are executing correctly). SDAccel™ provides specific hardware debug capabilities which include waveform analysis, kernel activity reports, MicroBlaze™ debugging as well as memory access analysis to isolate these critical hardware issues. Hardware debugging requires additional logic to be incorporated into the overall hardware model and will impact FPGA resources and performance. This additional logic can be removed in the final compilation.

GDB

GDB debugging allows you to add breakpoints, inspect variables, and debug the kernel or host code. This familiar software debug flow allows quick debugging to validate the functionality. However, SDAccel™ also provides special GDB extensions to examine the content of the OpenCL™ run-time environment from the application host. These can be used to debug protocol synchronization issues between the host and the kernel.

The SDAccel environment supports GDB host program debugging in all flows, but kernel debugging is limited to software and hardware emulation flows. Debugging information needs to be generated first in the binary container by passing the `-g` option to the `xocc` command line executable or enabled by setting the appropriate box in the GUI options.

In software and hardware emulation flows, there are restrictions with respect to the accelerated kernel code debug interactions. Because this code is preprocessed in the software emulation flow and translated into a hardware description language (HDL) in the hardware emulation flow, it is not always possible to set breakpoints at all locations especially in hardware emulation.

For more details, see the *SDAccel Environment Debugging Guide* (UG1281)

Linux “dmesg”

Debugging hangs in the system can be difficult; however, SDAccel™ provides a method to debug the interaction with Linux and the hardware platform via the `dmesg` Linux command. When the software or hardware appears to lock up, you can use the `dmesg` command to print a record of the transactions and kernel information messages. The detailed report can help to isolate and resolve the issue.

Kernel Waveform Viewer

SDAccel™ provides waveform-based HDL debugging through the GUI flow in hardware emulation mode. The waveform is opened in the Vivado® waveform viewer which will be familiar to Vivado users. It allows you to display kernel interface and internal signals and includes debug controls such as restart, HDL breakpoints, as well as HDL code lookup and waveform markers. In addition it provides top-level DDR data transfers (per bank) along with kernel-specific details including compute unit stalls, loop pipeline activity and data transfers.

For details see the *SDAccel Environment Profiling and Optimization Guide* (UG1207)

ILA

SDAccel™ provides insertion of the system Integrated Logic Analyzers (ILA) into a design to capture and view AXI transaction level activity via the interface signals between the kernel and global memory. The ILA provides, for example, custom event triggering on one or more signals to allow waveform capture at system speeds. The waveforms can be analyzed in a viewer and used to debug hardware such as protocol violations or performance issues and can be crucial for debugging difficult situation like application hangs. This low-level hardware-centric debug technique will be known by Vivado® users. See the *Vivado Design Suite User Guide: Programming and Debugging* (UG908) for complete details.

Note: The ILA core requires system resources, including logic and local memory to capture and store the signal data.

System ILAs can be inserted into the design using `xocc` command with `-dk` options.

For example,

```
xocc -dk chipscope:<compute_unit_name>:<interface_name>
```

Captured data can be accessed via the Xilinx® Virtual Cable (XVC) using the Vivado tools.

MicroBlaze Debug

RTL kernel designers often use MicroBlaze™ embedded processors as part of their accelerator designs. SDAccel™ allows software debugging of the embedded MicroBlaze processor in RTL kernel designs. To debug the software applications running on those embedded MicroBlaze designs, you need to include a MicroBlaze Debug Module (MDM) IP in you designs.

In the data center environment, debugging must be performed using the Xilinx® Virtual Cable (XVC) protocol over PCIe®.

To enable MicroBlaze debug in an RTL kernel via MDM using XVC-over-PCIe, the following steps must be done:

1. MDM must be enabled to include an "External BSCAN" interface.
2. The external BSCAN interface of each MDM in the RTL kernel must be brought up to a top-level interface of the RTL kernel.
3. The Debug Bridge (BSCAN-to-DebugHub mode) slave in the dynamic region must be enabled to include a BSCAN Master interface for each MDM External BSCAN interface exposed on the RTL Kernel.
4. Each BSCAN Master interface of the Debug Bridge from step 3 above should be connected to the corresponding External BSCAN interface of the RTL Kernel from step 2 above.

On the host side, the RTL kernel designers will use SDK to connect to the XVC server running on the host to connect to the desired MicroBlaze/MDM in their designs.

Xilinx Board Swiss Army Knife

Finally, the Xilinx Board Swiss Army Knife Utility (XBSAK) is a powerful standalone command line utility that can be used to debug lower level hardware / software interaction issues.

While this tool provides board administration commands, it also provides the following important debug operations:

Table 6: **Debug Operations**

Operation	Description
Query	Queries the specified device and programmable region on the device to get detailed status information of compute units. Can be used to isolate hangs of kernels.
Clock	Sets frequencies of clocks driving the computing units.
Download	Downloads the SDAccel™ binary (xclbin) to the programmable region of the device.
status	Displays the status of the SDx™ Performance Monitors (spm) and the Lightweight AXI Protocol Checkers (LAPC).
mem	Reads and writes to DDR memory.

A description of this utility can be found in the *SDx Command and Utility Reference Guide* (UG1279).

Command Line Flow

In addition to using the SDAccel™ GUI to create projects and build hardware accelerated applications, the SDx™ system tools can be invoked with a command-line interface in a command or shell window, in which a path to the tools has been set.

TIP: *To configure a command shell, source the* `settings64.sh` *or* `settings64.csh` *file on Linux, from the* `<install_dir>/SDx/<version>` *directory, where* `<install_dir>` *is the installation folder of the SDx software, and* `<version>` *is the software release.*

You will recall from Building the System that an SDAccel application project is compiled in a two part process: the software build process for the host application, and the hardware build process for the accelerated kernel. The host application is compiled using XCPP, a GCC compatible compiler.

The SDAccel Xilinx® Open Code Compiler (XOCC), `xocc`, is a command line compiler that takes your source code and runs it through the Vivado® 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).

This chapter walks through the command-line flow to show you how to build the software and hardware components from the command-line, or a script. The following topics show the steps and options used to create an `.exe` file for the host code, and an `.xclbin` file for the kernel in a sample project. See the *SDx Command and Utility Reference Guide* (UG1279) for details on the XOCC tool and associated options.

Host Code Compilation and Linking

Compiling

The host code (written in C/C++ using OpenCL™ APIs) is compiled by Xilinx® C++ (XCPP) compiler and generates host executable (`.exe` file) which executes on the host CPU. XCPP is a wrapper which uses standard `gcc` compiler along with standard gcc switches and command line arguments which should be familiar to the software developer and are not elaborated here.

TIP: *XCPP is based on GCC, and therefore supports many standard GCC options which are not documented here. For information refer to the GCC Option Index.*

An example of the `xcpp` command used to compile a design is given below:

```
xcpp -DSDX_PLATFORM=xilinx_vcu1525_dynamic_5_1 -I/lin64/SDx/2018.2/runtime/
include/1_2/
-I/lin64/Vivado/2018.2/include/ -g -Wall -c -o vadd.o vadd.cpp
```

The various options used are detailed on the right-hand side in brackets.

Figure 43: **Host Code Compilation**

```
$ xcpp
-DSDX_PLATFORM=xilinx_vcu1525_dynamic_5_1      (define macro)
-I/lin64/SDx/2018.2/runtime/include/1_2/       (include directory of header files)
-I/lin64/Vivado/2018.2/include/                (include directory of header files)
-I<user include directory>                     (user include file directory)
-g                                             (produce debugging information)
-Wall                                          (all warnings)
-c                                             (compile)
-o vadd.o                                      (create vadd.o output file)
vadd.cpp                                       (source file)
```

Linking

The generated object files (`.o`) are linked with the Xilinx® SDAccel™ runtime shared library to create the executable (`.exe`).

An example of the `xcpp` command used to link the design is given below:

```
xcpp -o vadd.exe -W1 -lxilinxopencl -lpthread -lrt -lstdc++ \
-L/lin64/SDx/2018.2.runtime/lib/x86_64 -rpath,/lnx64/lib/csim vadd.o
```

The various options used are detailed on the right-hand side in parentheses.

Figure 44: **Linking the Host Code**

```
$ xcpp
-o vadd.exe                                    (create vadd.exe output file)
-Wl                                            (specify warning level)
-lxilinxopencl -lpthread -lrt -lstdc++         (link with various library files)
-L/lin64/SDx/2018.2/runtime/lib/x86_64         (look in directories for library files)
-rpath,/lnx64/lib/csim                         (designate run-time search path)
vadd.o                                         (source file)
```

Kernel Code Compilation and Linking

Compiling

The first stage in building any system is to compile a kernel accelerator function. Compilation is done using the XOCC compiler. There are multiple `xocc` options that need to be used to correctly compile your kernel. These options are discussed below.

- Kernel source files are specified on the `xocc` command by directly listing the source files. Multiple source files can be added.
- The `-k` / `--kernel` option is used to specify the kernel name associated with the source files.

```
xocc … -k <kernel_name> <kernel_source_file> … <kernel_source_file>
```

- A platform on which the kernel is to be targeted needs to be specified. Specify the platform using the `-platform xocc` option. Xilinx® platforms include `xilinx_vcu1525_dynamic_5_1` and `xilinx_kcu1500_dynamic_5_0`.

```
xocc … --platform <platform_name>
```

- Specify the build target with the `-target xocc` option. By default, the target is set to `hw`. However, as discussed in Build Targets, the target can be one of the following:
  - `sw_emu` for software emulation
  - `hw_emu` for hardware emulation
  - `hw` for building on the target board
- The name of the generated output file can optionally be specified using the `-o` option. The default output file name is `<kernel>.xo`.

```
xocc .. -o <xo_kernel_name> .xo
```

- System and estimate reports can optionally be generated using the `-R`/`-report_level` options. Furthermore, you can optionally specify report, log and temp directories using the `-report_dir`, `--log_dir` and `-temp_dir` options respectively. These can be useful for organizing the generated files.

```
xocc ... --report_level 2 --report_dir <report_dir_name>
```

- Finally use the `xocc -c/--compile` option to compile the kernel. This will generate an `.xo` file that can be used in the subsequent link stage.

```
xocc ... -c
```

Putting it all together in an example:

```
xocc -c -k krnl_vadd --platform xilinx_vcu1525_dynamic_5_1 vadd.cl vadd.h \
-o krnl_vadd.xo --report_level 2 --report_dir reports
```

This will:

- Compile the kernel
- Kernel named `krnl_vadd`
- Use `xilinx_vcu1525_dynamic_5_1` platform.
- Use `vadd_file1.cppvadd_file2.cpp` source files.
- Output file named `krnl_vadd.xo`
- Generate reports and write them to the directory `reports`

Linking

As discussed in Chapter 5: Building the System, the second part of the build process links one or more kernels into the platform to create the binary container `xclbin` file. Similar to compiling, linking requires several options.

- The `.xo` source files are specified on the `xocc` command by directly listing the source files. Multiple source files can be added.

```
xocc … <kernel_xo_file.xo> … <kernel_xo_file.xo>
```

- You must specify the platform with the `-platform` option. The platform specified must be identical to that specified in the compile stage.

```
xocc … --platform <platform_name>
```

- Like the compile stage, you can specify the name of the generated output file using the `-o` option. The output file in the link stage will be an `.xclbin` file. The default output name is a `.xclbin`.

```
xocc .. -o <xclbin_name> .xclbin
```

- As described in Using the --nk Switch, the `--nk` option instantiates the specified number of compute units for the given kernel in the `.xclbin` file. While the compute unit instance name is optional, it is recommended to add one.

```
xocc ... --nk <kernel_name>: <compute_units>:<kernel_name1>:
…:<kernel_nameN>
```

- As described in Using the --sp Switch, you can optionally use the `-sp` option to specify the connection of a kernel interface to the target DDR bank. Multiple `--sp` options can be specified to map each of the interfaces to a particular bank.

```
xocc ... -sp <kernel_inst_name>.<interface_name>:<bank>
```

- Finally, linking is also done using the `-l`/`--link` option.

```
xocc ... -l
```

Putting it all together in an example:

```
xocc -l --platform xilinx_vcu1525_dynamic_5_1 -nk krnl_vadd:1:krnl_vadd1 \
--sp krnl_vadd1.m_axi_gmem:bank3 -o vadd.xclbin
```

This will:

- Link the kernel
- Use the `xilinx_vcu1525_dynamic_5_1` platform
- Create one compute unit called `krnl_vadd1`
- Map `krnl_vadd1`, port `m_axi_gmem` to DDR bank3
- Name output `.xclbin` file `vadd.xclbin`

Using the sdaccel.ini 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 the `sdaccel.ini` file automatically based on your 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 reads the parameters from the file during start-up if it finds it.

Run-time Initialization File Format

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

The following is a simple example that turns on profile timeline trace and sends the run-time log messages to the 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`	To Be Supplied	To Be Supplied
`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: false
`timeline_trace`	`[true|false]`	Enable or disable profile timeline trace • true: enable • false: disable • Default: false

Key	Valid Values	Descriptions
`device_profile`	`[true|false]`	Enable or disable device profiling. • true: enable • false: disable • Default: false
`[Runtime] Group`	To Be Supplied	To Be Supplied
`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 run-time 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. For example, runtime_log=my_run.log • Default: null
`polling_throttle`	An integer	Specify the time interval in microseconds that the run-time library polls the device status. Default: 0
`[Emulation] Group`	To Be Supplied	To Be Supplied
`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 = do not print in user console • Default: true

Key	Valid Values	Descriptions
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 user 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 = do not print in user 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
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.

Creating RTL Kernels

Many hardware engineers have existing RTL IP (including Vivado® IP integrator based designs), or just feel comfortable implementing a kernel in RTL and develop it using Vivado. The SDAccel™ environment allows these designs to be integrated into the flow to create accelerated kernels. However, RTL-based IP must be migrated to the SDAccel Environment framework. This migration requires packaging RTL-based IP as an SDAccel kernel enabling it to be used within the tool flow and run-time library.

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

Requirements for Using an RTL Design as an RTL Kernel

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

- Software requirements
- Interface requirements

It is necessary to add or modify the RTL code to meet these requirements to allow the RTL kernel to operate within the SDAccel™ framework.

Software Requirements

RTL kernels are are seen by the software application 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, 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

An RTL kernel must adhere to the C function interface, shown in the previous section, for integration into a platform. These interfaces include:

- A single slave AXI4-Lite interface used to access control registers (to pass scalar arguments and to start/stop the kernel).
- At least one of the following interfaces. (Can have both interfaces).

  1. AXI4 master interface to communicate with memory.

  2. AXI4-Stream interface for transferring data between kernels.

The various interface requirements are summarized in the following table. Note in some instances the port names must be written exactly.

Table 7: **RTL Kernel Interface and Port Requirements**

Port or Interface	Description	Comment
ap_clk	Primary clock input port	Name must be exact.
ap_clk_2	Secondary optional clock input port	• Name must be exact. • Optional port.
ap_rst_n	Primary active-Low reset input port	• Name must be exact. • This signal should be internally pipelined to improve timing. • This signal is driven by a synchronous reset in the ap_clk clock domain.
ap_rst_n_2	secondary optional active-Low reset input	• Name must be exact. • Optional port. • This signal should be internally pipelined to improve timing. • This signal is driven by a synchronous reset in the ap_clk_2 clock domain.
S_AXI_CONTROL	One and only one AXI4-Lite slave control interface	• 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 cleared 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 the 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.
AXI4_MASTER	One or more AXI4 master interfaces for global memory access	• All AXI4 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.

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™.

The benefit of the wizard are:

- Automates 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 AXI4-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 a wrapper template.
- A `kernel.xml` file is generated to match the software function prototype and behavior specified in the wizard.

The RTL Kernel Wizard generates a Vivado® project containing an example design consisting of a simple adder RTL IP, called VADD. In addition, it generates an associated RTL wrapper matching the desired interface, control logic and register map (described above) based on the user Wizard input. It is this wrapper which is retained and used to wrap your RTL IP into an RTL Kernel accessible by the SDAccel framework.

The generated RTL IP (VADD) will be replaced by your RTL and connected to the wrapper. The connections include clock(s), reset(s), AXI4-Lite interface, memory interfaces, and optionally streaming interfaces. The number of connections will be based on the interface information provided to the kernel wizard (for example, choosing two AXI4-Memory interfaces). It is necessary to manually make these connections to your IP and validate the design.

The Wizard generates a Vivado® project for the top-level RTL kernel wrapper and the generated files. This enables you to easily update and optimize the RTL kernel.

Furthermore, the Wizard also generates a simple test bench for the generated RTL kernel wrapper and a sample host code to exercise the example RTL kernel. This example test bench and host code must be modified to test the your RTL IP design accordingly.

Using the Kernel Wizard is described in the following subsections.

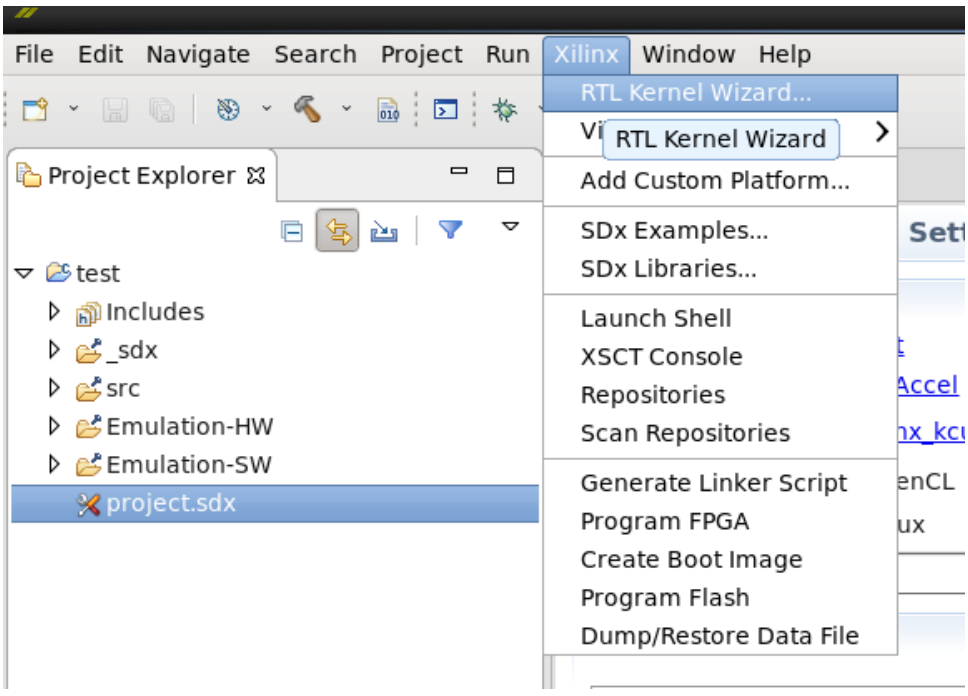
Launching the RTL Kernel Wizard

The RTL Kernel Wizard can be launched with two different methods: from the SDx™ Development Environment or from the Vivado® Integrated Design Environment (IDE). The SDx Development Environment provides a more seamless 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**. (See the following image.)

Figure 45: **Launching the 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.

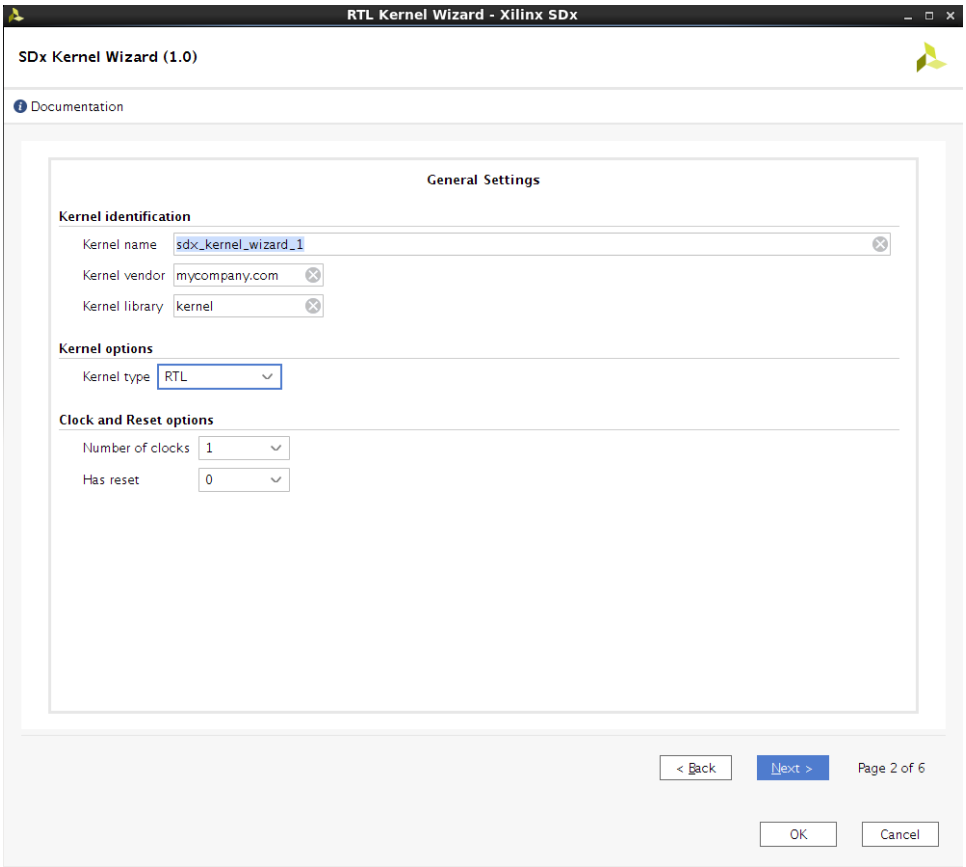
Using the RTL Kernel Wizard

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

RTL Kernel Wizard General Settings

The following graphic shows the three settings in the General Settings tab.

Figure 46: **Kernel Wizard General Settings**



Kernel Identification

Following are the three settings in the General Settings tab.

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

Kernel Options

Kernel type:

An 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-level module. The 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 a MicroBlaze™ subsystem that uses a block RAM exchange memory to emulate the control registers. Example MicroBlaze software is delivered with the project to demonstrate using the MicroBlaze to control the kernel – supports RTL and block design kernel types.

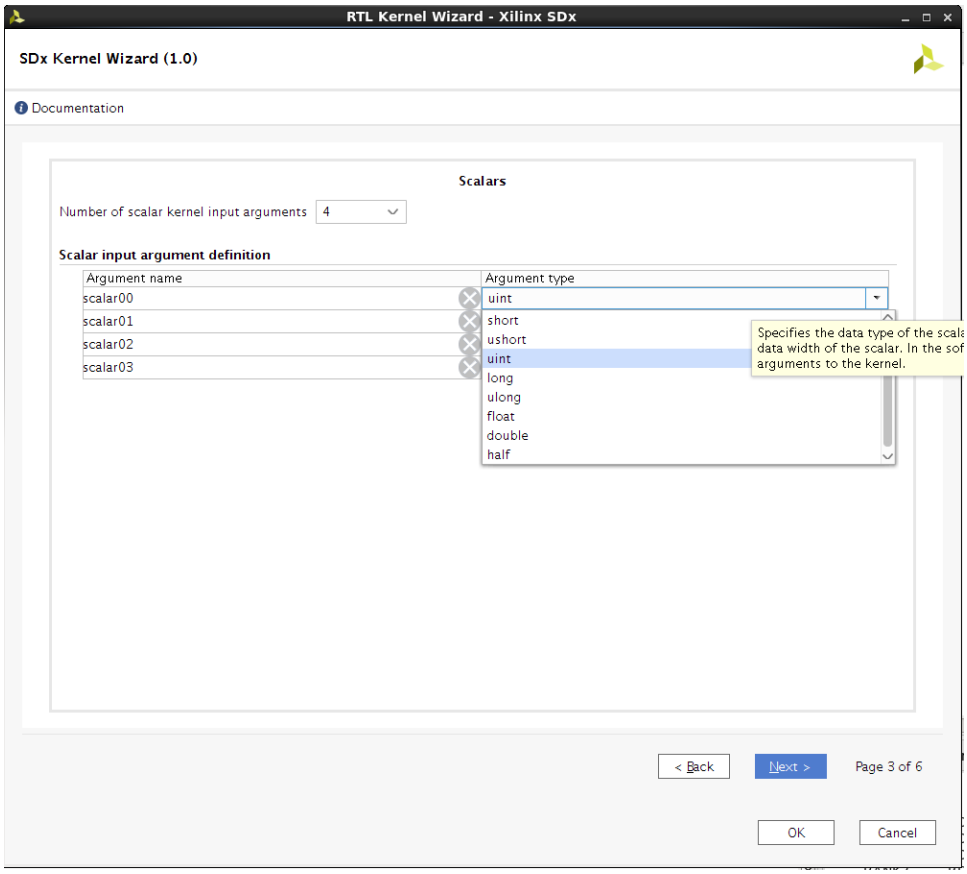
Clock and Reset Options

- **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 are driven with this clock and reset. When selecting **Number of clocks** 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 or 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
- **Has Reset:** Specifies whether to include a top level reset input port to the kernel. Omitting a reset can be useful to improve routing congestion of large designs. Any registers that would normally have a reset in the design should have proper initial values to ensure correctness. If enabled, there will be a reset port included with each clock. Block Design type kernels must have a reset input.

Scalars Arguments

Scalar arguments are used to pass control type information to the kernels. Scalar arguments cannot be read back from the host. For each argument that is specified, a corresponding control register is created to facilitate passing the argument from software to hardware. See the following image.

Figure 47: **Kernel Wizard Scalars**



- Number of scalar kernel input arguments: Specifies the number of scalar input arguments to pass to the kernel. For each number specified, a table row is generated that allows customization of the argument name and argument type. There is no required minimum number of scalars and the maximum allowed by the wizard is 64.

Following is the scalar input argument definition.

- Argument name: The argument name is used in the generated Verilog control register module as an output signal. Each argument is assigned an ID value. This ID value is used to access the argument from the host software. The ID value assignments can be found on the summary page of this wizard. To ensure maximum compatibility, the argument name follows the same identifier rules as the kernel name.

- Argument type: Specifies the data type of the argument. This affects 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 require two write operations to the control registers.

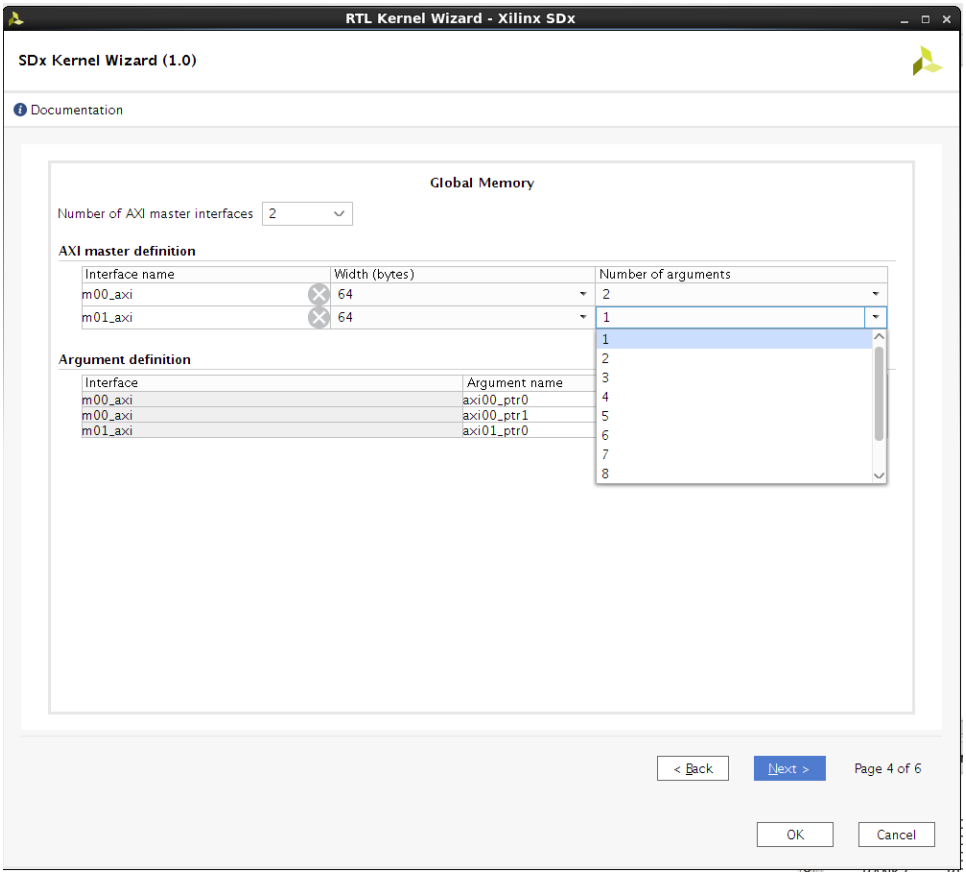
Global Memory

Global memory is accessed by the kernel through AXI4 master interfaces (See Global Memory figure). 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 the number of associated arguments. Each interface contains all read and write channels.

For RTL kernels the port names are generated during the import process by the RTL kernel wizard. The default names proposed by the RTL kernel wizard are m00_axi and m01_axi. If not changed, these names will have to be used when assigning a DDR bank through the `--sp` option.

Figure 48: **Global Memory**



AXI Master Definition (table columns)

- **Interface Name:** Specifies the name of the interface. To ensure maximum compatibility, the argument name follows the same identifier rules as the kernel name.
- **Width in bytes:** Specifies the data width of the AXI data channels. Xilinx recommends to match 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

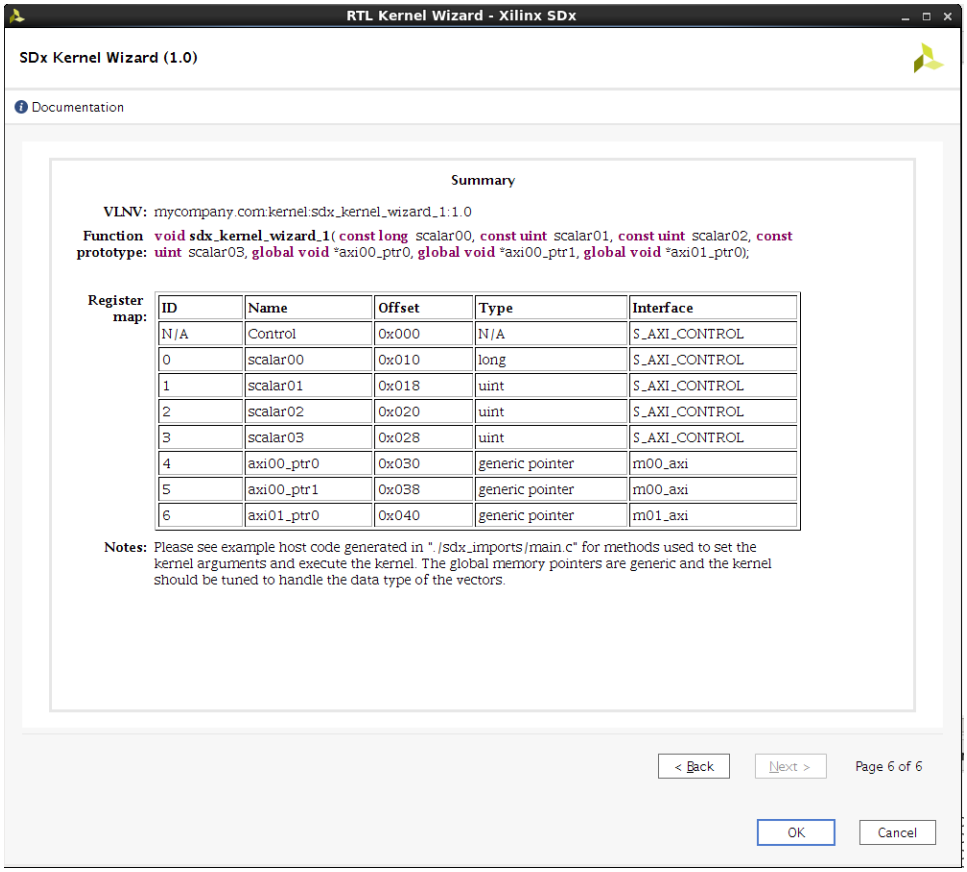
- **Interface:** Specifies the name of the AXI Interface that the corresponding columns in the current row are associated with. This value is not directly modifiable; it is copied from the interface name defined in the previous table.

- **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 section summarizes VLNV, the software function prototype, and hardware control registers created from options selected in the previous pages. The function prototype conveys what a kernel call would be like if it was a C function. See the host code generated example of how to set the kernel arguments for the kernel call. The register map shows the relationship between the host software ID, argument name, hardware register offset, type, and associated interface. Review this section for correctness before proceeding to generate the kernel.

Figure 49: **Kernel Wizard Summary**



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 click **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 customizes the contents of the kernel and then packages 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.

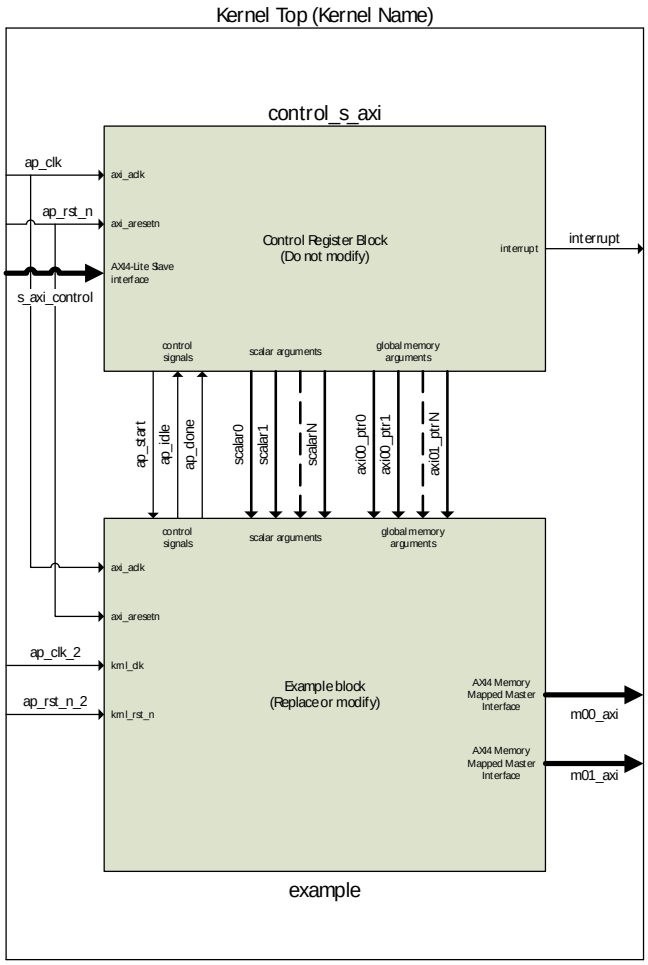
The top-level Verilog file contains the expected input/output signals and parameters. These top-level ports are matched to the kernel specification file (`kernel.xml`) and when combined with the rest of the RTL/block design becomes the acceleration kernel. The AXI4 interfaces defined at the top-level 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 a Verilog example and control registers or an instantiated IP integrator block design.

RTL Kernel Type Project Flow

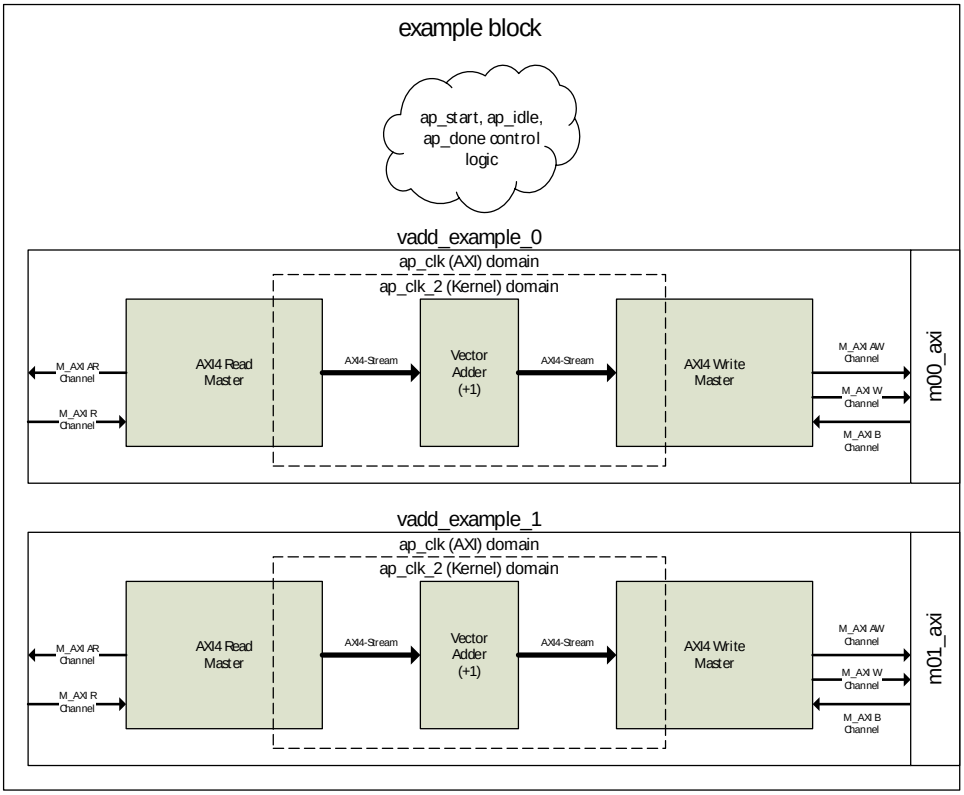
The RTL kernel type delivers a top-level Verilog design consisting of control register and Vadd sub-modules. The following figure illustrates the top-level design configured with two AXI4-master interfaces. Care should be taken if the control register module is modified to ensure that it still aligns with the `kernel.xml` file located in the imports directory of the Vivado® kernel project. The example sub-module can be replaced with your custom logic or used as a starting point for your design.

Figure 50: **Kernel Type RTL Top**



The Vadd sub-module, shown in the following figure, consists of a simple adder function, an AXI4 read master, and an AXI4 write master. Each defined AXI4 interface has independent example adder code. The first associated argument of each interface is used as the data pointer for the example. Each example reads 16 KB of data, performs a 32 bit “add one” operation, and then writes out 16 KB of data back in place (the read and write address are the same).

Figure 51: **Kernel Type RTL Example**



The following table describes important files relative to the root of the Vivado project for the kernel, where <kernel_name> is the name of the kernel chosen in the wizard.

Table 8: **RTL Wizard Source and Test Bench File**

Filename	Description	Delivered with Kernel Type
<kernel_name>_ex.xpr	Vivado project file	All
imports directory		
<kernel_name>.v	Kernel top level module	All
<kernel_name>_control_s_axi.v	RTL control register module	RTL
<kernel_name>_example.sv	RTL example block	RTL
<kernel_name>_example_vadd.sv	RTL example AXI4 vector add block	RTL
<kernel_name>_example_axi_read_master.sv	RTL example AXI4 read master	RTL
<kernel_name>_example_axi_write_master.sv	RTL example AXI4 write master	RTL
<kernel_name>_example_adder.sv	RTL example AXI4-Stream adder block	RTL
<kernel_name>_example_counter.sv	RTL example counter	RTL
<kernel_name>_exdes_tb_basic.sv	Simulation test bench	All
<kernel_name>.cpp	Software C-Model example for software emulation.	All

Table 8: **RTL Wizard Source and Test Bench File** *(cont'd)*

Filename	Description	Delivered with Kernel Type
<kernel_name>_ooc.xdc	Out-of-context Xilinx constraints file	All
<kernel_name>_user.xdc	Xilinx constraints file for kernel user constraints.	All
kernel.xml	Kernel description file	All
package_kernel.tcl	Kernel packaging script proc definitions	All
post_synth_impl.tcl	Tcl post-implementation file	All
sdx_imports directory		
main.c	Host code example	All
makefile	Makefile example	All
<kernel_name>_ex.sdk/<kernel_name>_control/src directory		
kernel_control.h	MicroBlaze™ C header file	Block Design
kernel_control.c	MicroBlaze C file	Block Design
<kernel_name>_ex.sdk/<kernel_name>_control/Debug directory		
<kernel_name>_control.elf	MicroBlaze™ elf file	Block Design
<kernel_name>_ex.src/sources_1/<kernel_name>_bd directory		
<kernel_name>_bd.bd	Vivado Block Diagram file	Block Design

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 might need to run the Export Hardware option. The option can be found in the File > Export > Export Hardware menu location. When the export Hardware dialog appears, click **OK**.
2. The software development kit (SDK) application can now be invoked. Select **File** → **Launch** → **SDK** 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 selecting the menu option **Project->Build All Check for errors/warnings and resolve if necessary**. The ELF file is automatically updated in the GUI.
6. Run simulation to test the updated program and debug if necessary.

Simulation Test Bench

When 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 test bench 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 Vivado® **Flow Navigator → Run Simulation** located on the left hand side of the GUI and select **Run Behavioral Simulation**. If behavioral simulation is working as expected, a post-synthesis functional simulation can be run to ensure that synthesis 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. A Xilinx® Design Constraints (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 usage 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, the kernel must be packaged as a netlist. To run out of context synthesis, click **Run Synthesis** 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 uses 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 selecting **Automatically add binary container(s) to arguments** in Run Configuration > Arguments 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, and 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 **Generate RTL Kernel** 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, design checkpoint (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, 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, provide a space in between each file in the Source files list, or use the GUI to select multiple files using the **CTRL** key when selecting the file.

After you click **OK**, the kernel output products are generated. If the pre-synthesized kernel or netlist kernel option is chosen, then synthesis can run. If synthesis has previously run, it will use those outputs, regardless if they are stale. The kernel Xilinx Object `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, the example host code called `main.c` and kernel Xilinx Object (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 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. Selecting **Edit Existing Kernel Contents** re-opens the Vivado® Project, and you can then modify and generate the kernel contents again. Selecting **Re-customize Existing Kernel Interfaces** will revisit the RTL Kernel Wizard configuration dialog box. Options other than Kernel Name can be modified and the previous Vivado project is replaced.

IMPORTANT!: *all files and changes in the previous project are 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 can 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 the IP integrator. See the *Vivado Design Suite: 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 interfaces must be packaged as AXI4 master endpoints with 64-bit address support.

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

- `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, and AXI4-Stream interfaces.

To test if the RTL kernel is packaged correctly for the IP integrator, try to instantiate the packaged kernel in the IP integrator. In the GUI it should show up as having interfaces for clock, reset, AXI4-Lite slave, AXI4 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, 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 the 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 9: **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

Table 9: **Kernel XML Format** *(cont'd)*

Tag	Attribute	Description
<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.
	preferredWorkGroupSizeMultiple	Reserved. Set it to 0.
	workGroupSize	Reserved. Set it to 1.
<port>	name	Port name. At least an AXI4 master port and an AXI4-Lite slave port are required. The AXI4-Stream port can be optionally specified to stream data between kernels. The AXI4-Lite interface name must be S_AXI_CONTROL.
	mode	• For AXI4 master port, set it to "master." • For AXI4 slave port, set it to "slave." • For AXI4-Stream master port, set it to "write_only." • For AXI4-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.

Table 9: **Kernel XML Format** *(cont'd)*

Tag	Attribute	Description
<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. For example, 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 AXI4- Stream ports.
The following tags specify additional information for AXI4-Stream ports. They are not applicable to AXI4 master or slave ports.		
<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

For the final step, 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 command line will package the `.xo` file. 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

While the RTL Wizard assists in packaging RTL designs for use within the SDx™ flow, the underlying RTL kernels should be designed with recommendations from the *UltraFast Design Methodology Guide for the Vivado Design Suite* (UG949).

Thus, in addition to adhering to the interface and packaging requirements, the kernels should be designed with performance goals in mind. Specifically

- Memory Performance Optimizations for AXI4 Interface
- Quality of Results Considerations
- Debug and Verification Considerations

which are described in the following subsections.

Memory Performance Optimizations for AXI4 Interface

The AXI4 interfaces typically connects 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:

- Use an AXI data width that matches the native memory controller AXI data width, typically 512 bits.
- Do not use WRAP, FIXED, or sub-sized bursts.
- Use burst transfer as large as possible (up to 4K Byte AXI4 protocol limit).
- Avoid use of deasserted write strobes. Deasserted write strobes can cause error-correction code (ECC) logic in the DDR memory controller to perform read-modify-write operations.
- Use pipelined AXI transactions.

- Avoid using threads if an AXI interface is only connected to one DDR controller.
- Avoid generating write address commands if the kernel does not have the ability to deliver the full write transaction (non-blocking write requests).
- Avoid generating read address commands if the kernel does not have the capacity to accept all the read data without back pressure (non-blocking read requests).
- If a read-only or write-only interfaces are desired, the ports of the unused channels can be commented out in the top level RTL file before the project is packaged into a kernel.
- Using multiple threads can cause larger resource requirements in the infrastructure IP between the kernel and the memory controllers.

Quality of Results Considerations

The following recommendations help improve results for timing and area:

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

Debug and Verification Considerations

- RTL kernels should be verified in their own test bench using advanced verification techniques including verification components, randomization, and protocol checkers. The AXI Verification IP (AXI VIP) is available in the Vivado® IP catalog and can help with the verification of AXI interfaces. The RTL kernel example designs contain an AXI VIP based test bench with sample stimulus files.
- The hardware emulation flow should not be used for functional verification because it does not accurately represent the range of possible protocol signaling conditions that real AXI traffic in hardware can incur. Hardware emulation should be used to test the host code software integration or to view the interaction between multiple kernels.

Packaging an RTL Block as an RTL Kernel

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 an associated kernel description XML file.
3. Package the RTL kernel into a Xilinx® Object (XO) file.

These steps are an automated 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 associated kernel XML file. The `.xo` file can be compiled into the platform and run in hardware or hardware emulation flows.

Appendix A

Getting Started with Examples

All Xilinx® SDx™ environments are provided with example 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.

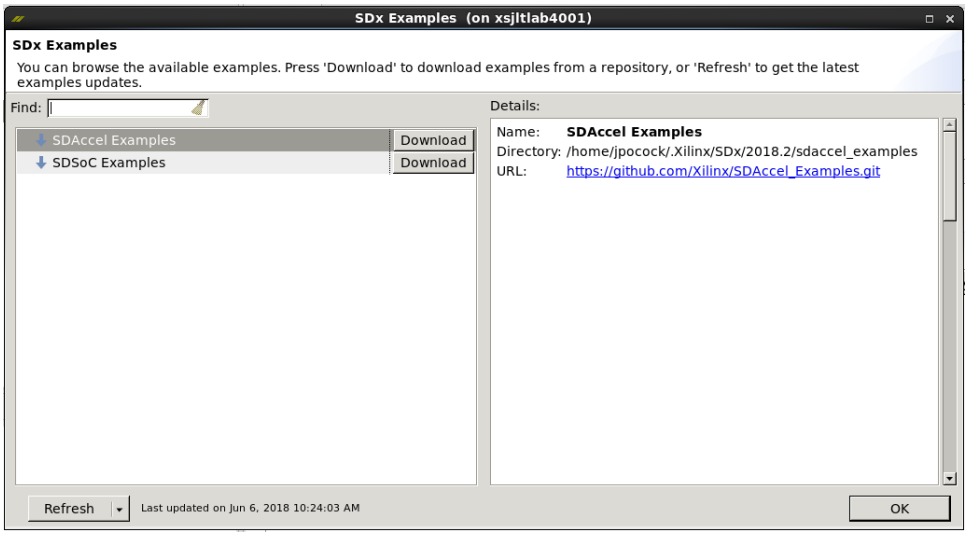
Every platform provided with SDAccel contains sample designs to get the user started, accessible through project creation as described in Creating an Application Project. Furthermore, each of these designs, which are found in `<sdx_root>/samples` provides a Makefile so you can build, emulate, and run the code working entirely on the command line if you prefer. In addition, 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, by selecting **Xilinx → SDx Examples**.

The first time the SDx™ Examples dialog box, or the Template page of the New SDx Project wizard displays, it is 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.

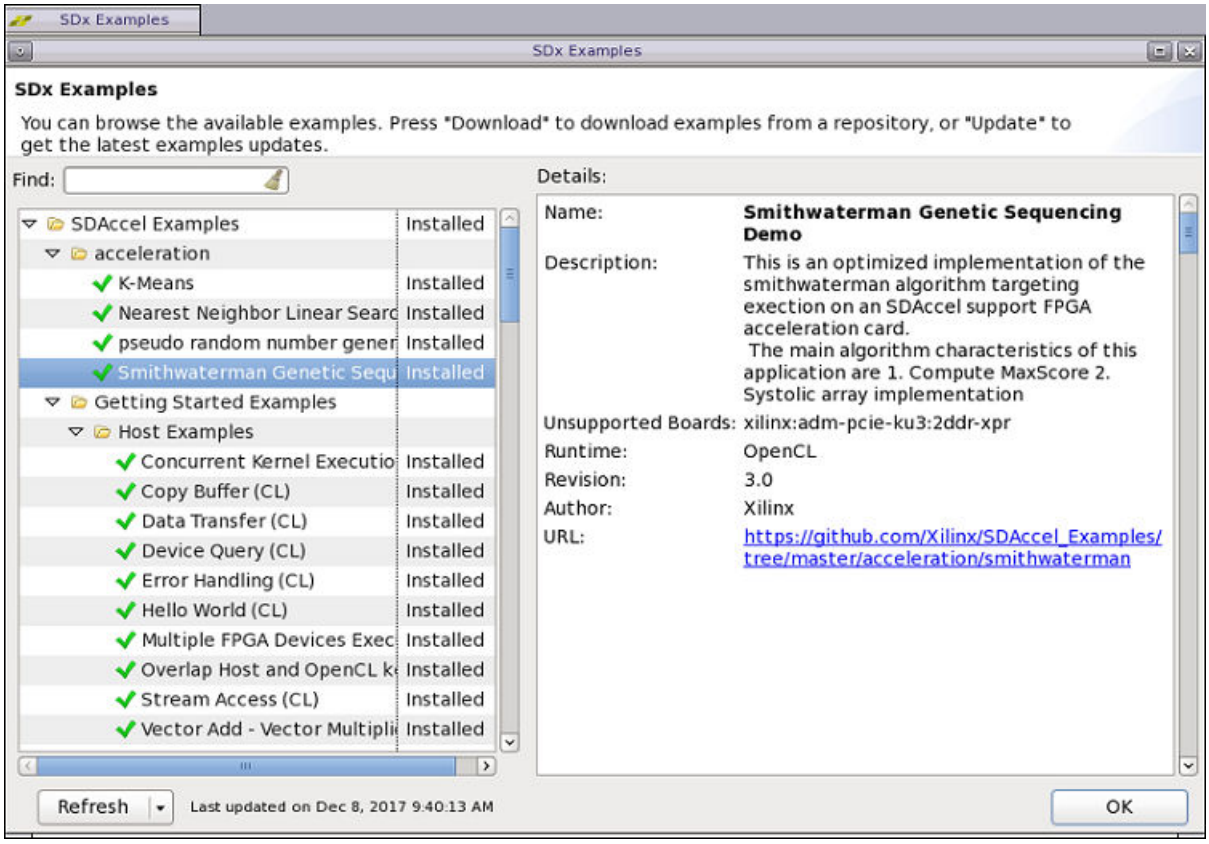
Figure 52: **SDAccel Examples - Empty**



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

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

Figure 53: **SDAccel 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/SDx/<version>`folder:

- On Windows: `C:\Users\<user_name>\.Xilinx\SDx\<version>`
- On Linux: `~/.Xilinx/SDx/<version>`

The download directory cannot be changed from the SDx Examples dialog box. However, you might want to download the example files to a different location from the `.Xilinx` folder.

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 examples using the `git` command as shown above, you can use the example files as a resource for application and kernel code to use in your own projects. However, many of the files use include statements to include other example files that are managed in the Makefiles of the various examples. These include files are automatically populated into the `src` folder of a project when the Template is added through the New 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 can 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
```

Directory Structure

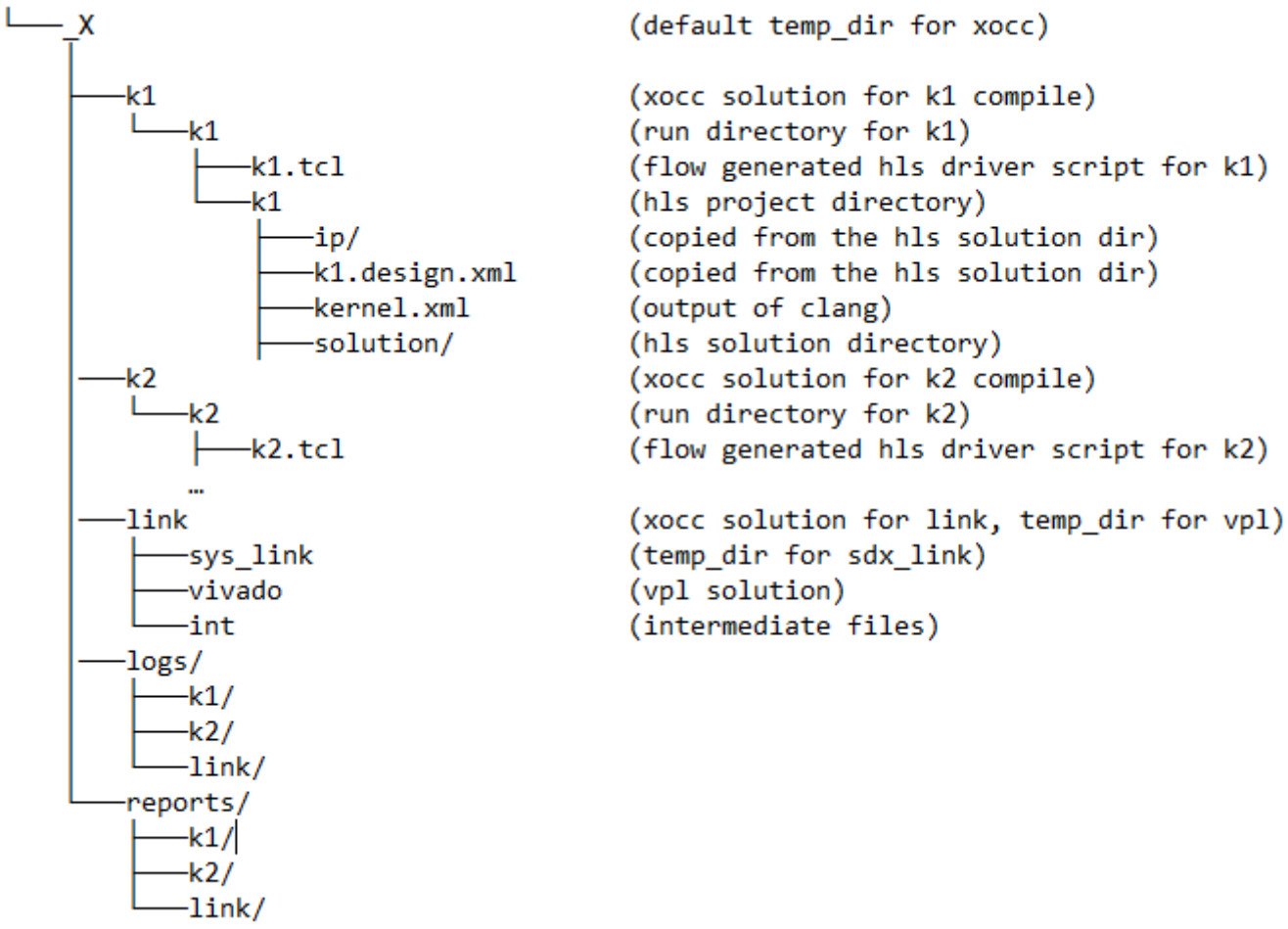
The directory structure generated by the GUI and make flows has been organized to allow you to easily find and access files. By navigating each compile, link, log and report directories, you can easily reach generated files. Furthermore, each kernel will also have a directory created.

Command Line

When using `xocc` via the command line, by default it creates the following directory structure during compile and link. The `.xo` and `.xclbin` are always generated in the working directory. All the intermediate files are created under the `_x` directory (default name of the temp_dir).

The following example shows the generated directory structure for two `xocc` compile runs (k1 and k2) and one `xocc` link (`design.xclbin`). The `k1.xo`, `k2.xo` and `design.xclbin` files are located in the working directory. The `_x` directory contains the associated k1 and k2 kernel compile subdirectories. The link, logs, and reports directories contain the respective information on the builds.

Figure 54: **Command Line Directory Structure**



You can optionally change the directory structure using the following xocc options:

```
--log_dir <dir_name (full or relative path)>
```

```
--report_dir< dir_name (full or relative path)>
```

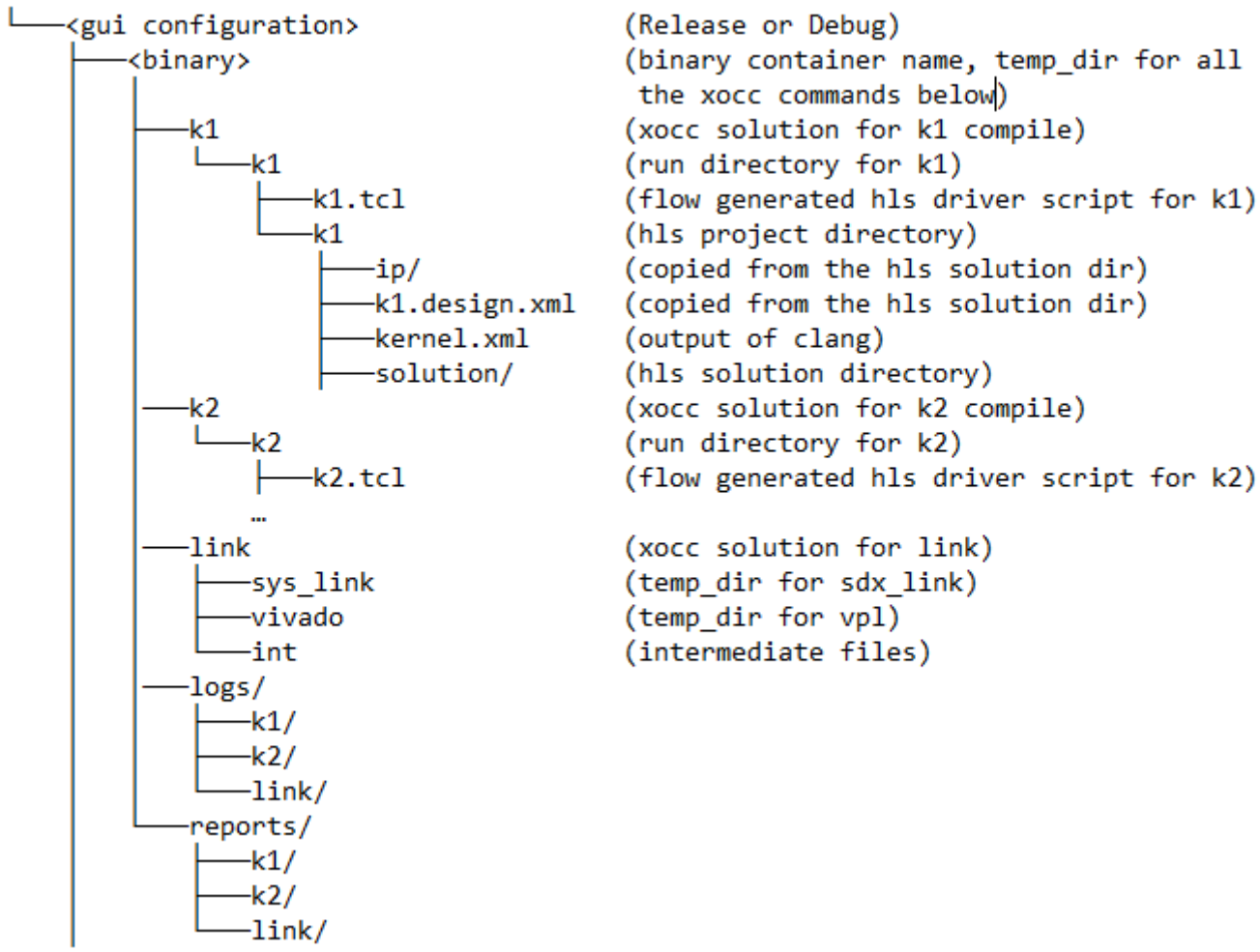
```
--temp_dir< dir_name (full or relative path)>
```

See *SDx Command and Utility Reference Guide* (UG1279) for details on the `xocc` command options.

GUI

Though similar, the default directory naming and structure is not identical to that created by the make flow. The following example shows the generated directory structure for two xocc compile runs (k1 and k2) and one `xocc` link (`design.xclbin`) automatically generated in the GUI flow. The `k1.xo`, `k2.xo` and `design.xclbin` files are located in the working directory. The `_x` directory contains the associated k1 and k2 kernel compile subdirectories. Again, the link, logs, and reports directories contain the respective information on the builds.

Figure 55: **GUI Directory Structure**



The GUI manages the creation of the directory structure using the following `xocc` command specifications which can be found in the makefile:

```
--temp_dir
```

```
--report_dir
```

```
--log_dir
```

See the *SDx Command and Utility Reference Guide* (UG1279) for details on the `xocc` command options.

SLR Assignments for Kernels

Super Logic Region (SLR) and DDR memory bank floor planning is key to meeting quality of results of your design both in terms of frequency and resources. Floor planning involves allocating kernels in different SLRs both to minimize the demand for scarce FPGA resources, but also to connect to the closest memory bank.

IMPORTANT!: *The following features are currently only applicable to KCU1500 and VCU1525 dynamic platforms version 5.0 or 5.1 containing an SDx™ memory subsystem instance.*

Each SLR contains a fixed amount of resources. Placing all kernels in one SLR can impact performance as the design has limited resource usage. Spreading kernels across multiple SLRs reduces congestion and allows your design to more easily be mapped to the FPGA and meet performance.

Second, because the memory banks of a device support archive (DSA) are distributed across SLRs of the platform and 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 bank with which it has the most connections. This reduces competition for SLR-crossing connections and avoids consuming extra logic resources associated with SLR crossing.

***Note*:** See the *SDx Environments Release Notes, Installation, and Licensing Guide* (UG1238) for the allocation of memory channels of the VCU1525 and KCU1500 DSAs).

You can provide directives to the `xocc` compiler to control the hardware placement of kernel logic and the mapping to specific memory bank.

Specifying SLR Assignment Information for a Platform Design

TIP: *When you provide a kernel SLR directive, 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. For complete details on allocating kernels to SLRs see the SDAccel Environment Profiling and Optimization Guide (UG1207).*

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, you 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:

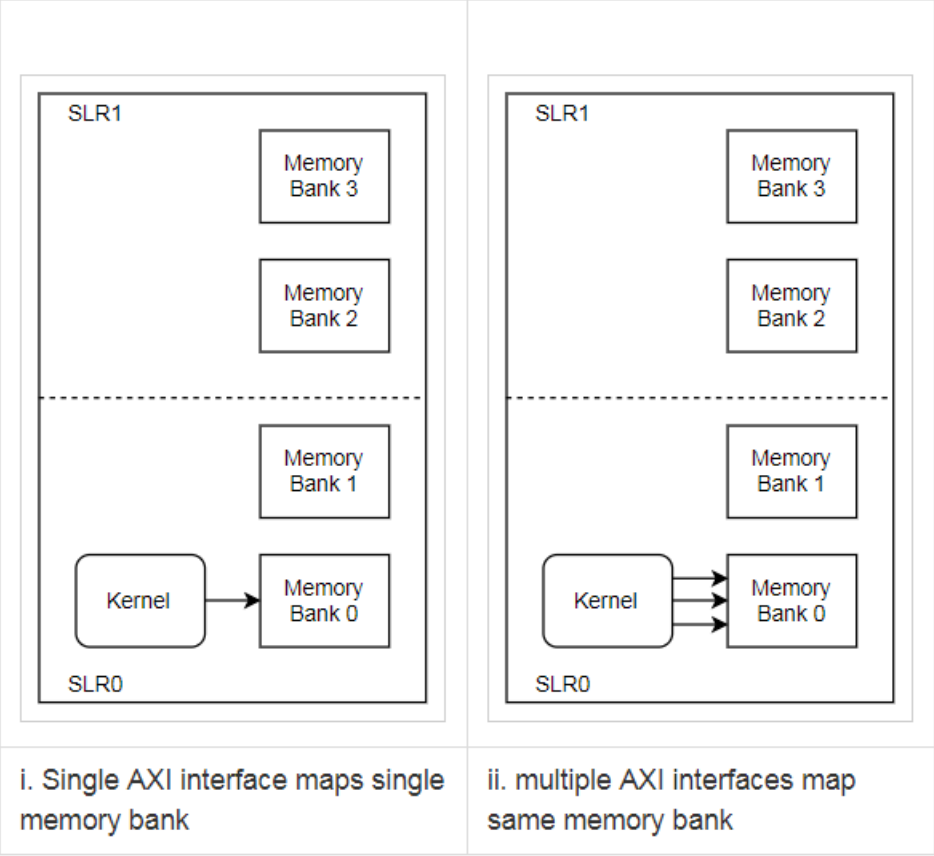
   ```
   --xp param:compiler.userPostSysLinkTcl=<full path to your
   userPostSysLink.tcl file>
   ```

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.*

Guidelines for Kernels that Access Multiple Memory Banks

The memory banks of a DSA are distributed across the super logic regions (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 56: **Kernel and Memory in Same SLR**

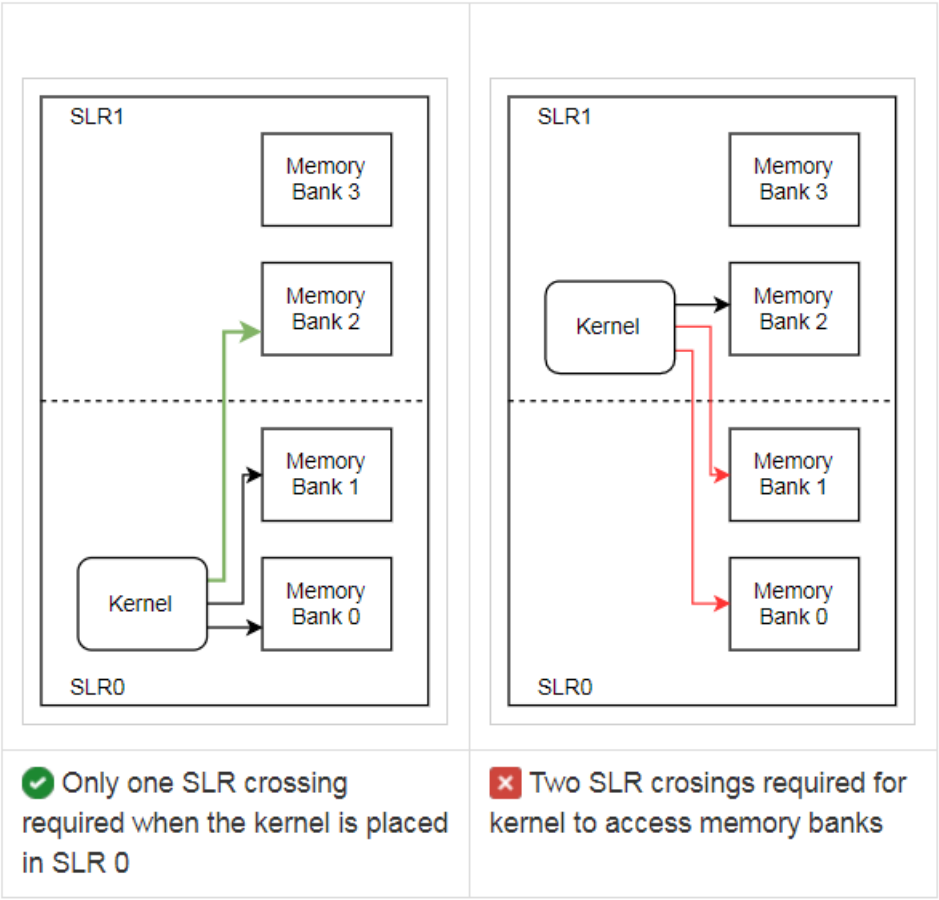


As shown in the previous figure, when a kernel has a single AXI interface that maps only a single memory bank, the DSA release information lists the SLR that is associated with the memory bank of the kernel; therefore, the SLR where the kernel would be best placed. In this simple scenario, the design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

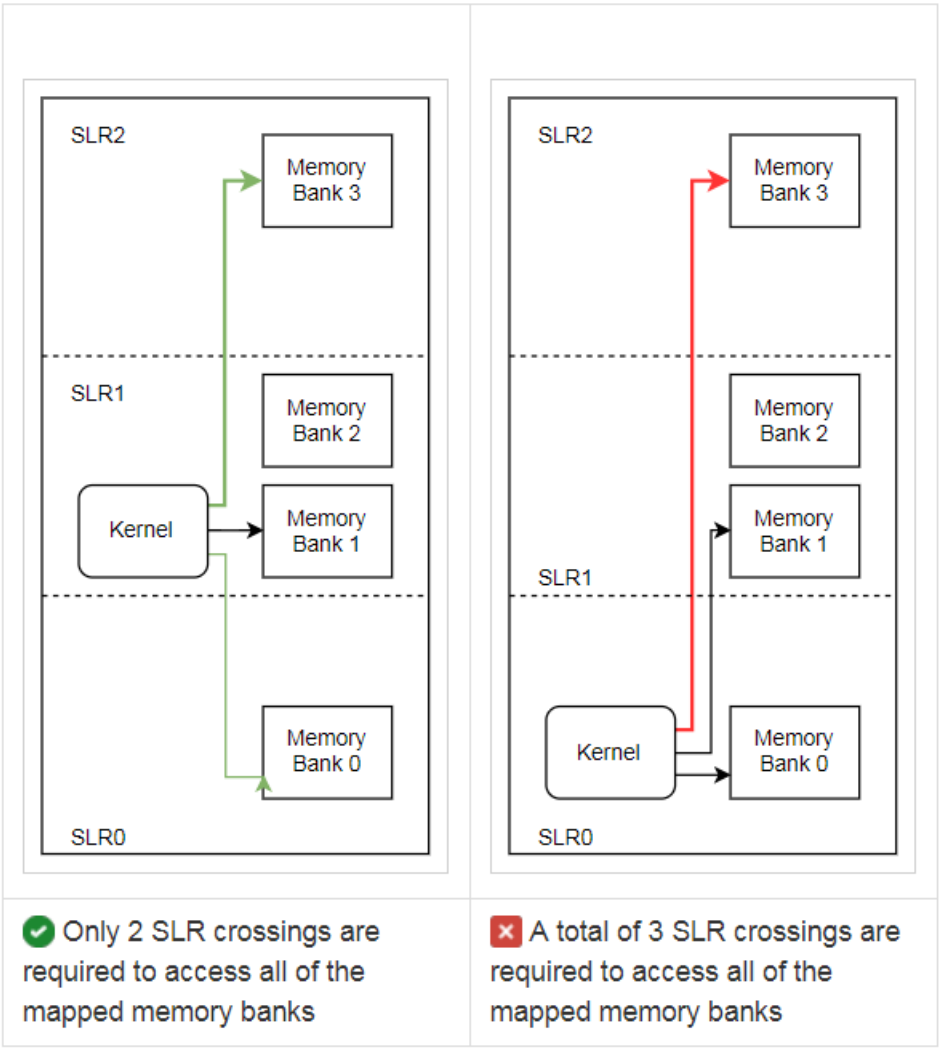
Figure 57: **Memory Bank in Adjoining SLR**



When a kernel has multiple AXI interfaces to multiple memory banks in different SLRs, the recommendation is to place the kernel in the SLR that has the majority of the memory banks accessed by the kernel (shown it the figure above). This minimizes the number of SLR crossings required by this kernel which leaves more SLR crossing resources available for other kernels in 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 58: **Memory Banks Two SLRs Away**



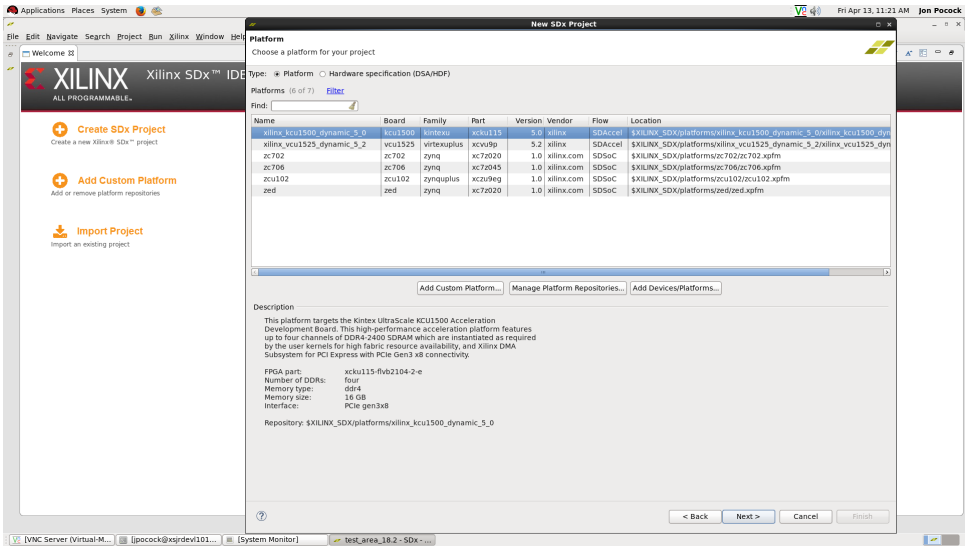
As shown in the previous figure, when a platform contains more than two SLRs, it is possible that the kernel might map a memory bank that is not in the immediately adjacent SLR to its most commonly mapped memory bank. When this scenario arises, memory accesses to the distant memory bank must cross more 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.

Managing Platforms and Repositories

SDx™ comes with built-in platforms. If you need to use a custom platform for your project, you need to make the platform available to SDAccel™ for application implementation.

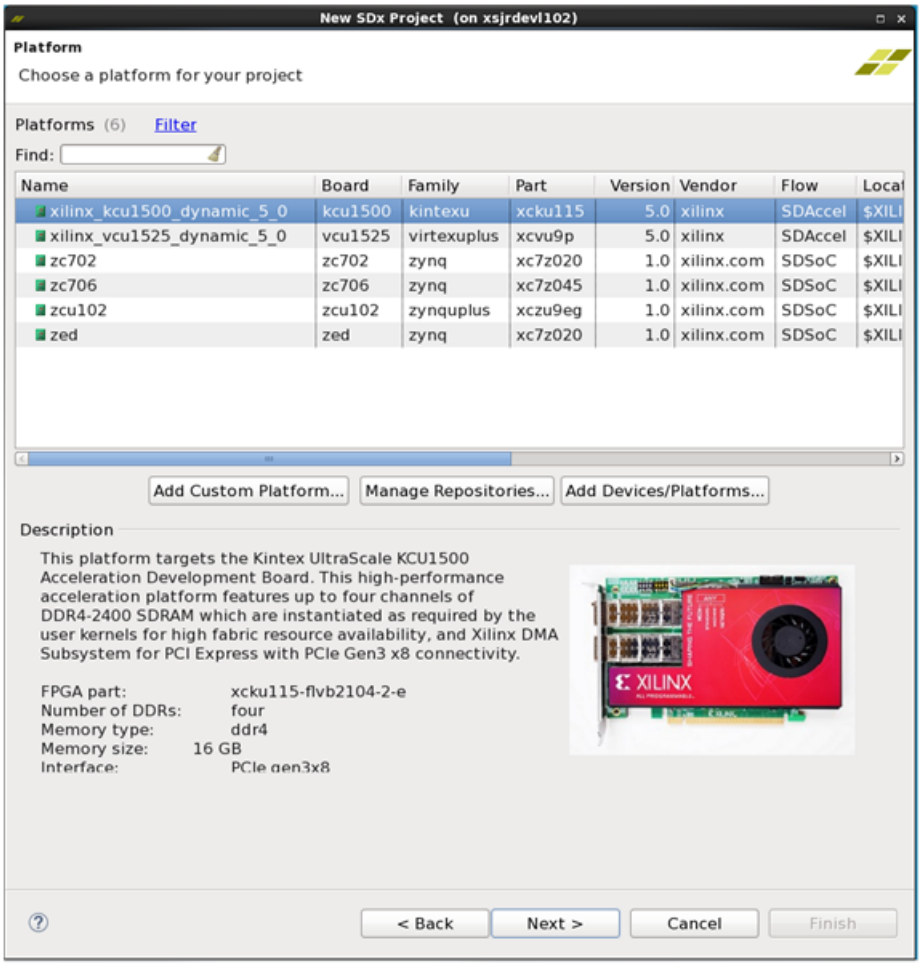
When you are creating a project, you can manage the platforms that are available for use in SDAccel application projects, from the Platform Selection page of the SDx New Project wizard. This lets you add a new platform for a project as it is being created.

Figure 59: **SDAccel Platform Browse**



This opens the New SDx Project dialog box, which lets you manage the available platforms and platform repositories

Figure 60: **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. Select **Xilinx → Add Custom Platform** 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. Select **Help → Add Devices/Platforms** to directly add custom platforms to the tool.

XILINX

Appendix E

JTAG Fallback for Private Debug Network

RTL kernel and platform debug in a data center environment typically uses the XVC-over-PCIe® connection due to the typical inaccessibility of the physical JTAG connector of the board. While XVC-over-PCIe allows you to remotely debug your systems, certain debug scenarios such as AXI interconnect system hangs can prevent you from accessing the design debug functionality that depends on these PCIe/AXI features. Being able to debug these kinds of scenarios is especially important for platform designers.

The "JTAG Fallback" feature is designed to provide access to debug networks that were previously only accessible via XVC-over-PCIe. The JTAG Fallback feature can be enabled without having to change the XVC-over-PCIe-based debug network in the platform design.

On the host side, when the Vivado® user connects via `hw_server` to a JTAG cable that is connected to the physical JTAG pins of the device under test (DUT), `hw_server` disables the XVC-over-PCIe pathway to the DUT. When you disconnect from the JTAG cable, hw_server re-enables the XVC-over-PCIe pathway to the DUT.

JTAG Fallback Steps

Here are the steps required to enable JTAG Fallback:

1. Enable the JTAG Fallback feature of the Debug Bridge (AXI-to-BSCAN mode) master of the debug network to which you want to provide JTAG access. This step enables a BSCAN slave interface on this Debug Bridge instance.
2. Instantiate another Debug Bridge (BSCAN Primitive mode) in the static logic partition of the platform design.
3. Connect the BSCAN master port of the Debug Bridge (BSCAN Primitive mode) from step 2 to the BSCAN slave interface of the Debug Bridge (AXI-to-BSCAN mode) from step 1.

Appendix F

Installing, Programming, and Debugging Boards

Introduction

This appendix contains the installation, programming, and debugging instructions are for installing and deploying a Xilinx® accelerator board to execute applications created with the SDAccel™environment.

The SDAccel application executes in hardware using one of the listed FPGA boards.

You can develop and run applications using the following:

- 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.
- A single computer to perform both development and deployment roles: both SDAccel and the accelerator board must be installed.

This procedure provides instructions installing a hardware accelerator board on a host computer. It also includes instructions for using the `xbinst` board installation utility and the Vivado® Design Suite to install the board. When you install SDAccelthe `xbinst` utility, the Vivado Design Suite, and any required cable drivers are included.

The process of installing a board requires that you use the following SDAcceltools:

- `xbsak` and `xbinst`: The use of these utilities for that purpose and how to use those utilities along with useful Operating System commands are described in the Debugging and Troubleshooting a Board in *SDx Command and Utility Reference Guide* (UG1279).

Programming the Board

The following steps are for progamming a board using the `xbsak flash` command directly from the deployment computer and its Linux OS using the command line prompt to program the configuration memory (flash memory device) on the FPGA board with specified configuration files from which the the FPGA can boot.

Xilinx boards are already programmed and are visible when you type the following command:

```
lspci -d 10ee: :
```

The command returns the following:

```
$ lspci -d 10ee:
03:00.0 Serial controller: Xilinx Corporation Device 6a90
03:00.1 Serial controller: Xilinx Corporation Device 6a8f
```

This confirms that the program is using the `xbsak` flash.

***Note*:** thePCI™ device names, here 03:00.0 and 03:00.1, are used in the following commands.

Before programming the FPGA flash, ensure that the `xclmgmt` and `xocl` drivers are unloaded, and that the device has been removed from the kernel before using the `xbsak flash` command.

1. Enter the following commands:

   ```
   $ sudo -i
   # rmmod xocl
   # rmmod xclmgmt
   # echo 1 > /sys/bus/pci/devices/0000\:03\:00.0/remove
   # echo 1 > /sys/bus/pci/devices/0000\:03\:00.1/remove
   # exit$
   ```

   Now, you can use `xbsak` flash.

2. Go to the `xbinst` directory and run the `install.sh` script:

   ```
   $ cd xbinst
   $ ./install.sh -k no
   ```

3. Source the `setup.{sh,csh}` to use `xbsak` flash.

   ```
   $ source xbinst/setup.{sh|csh}
   ```

4. Find the necessary MCS files and run `xbsak` flash:

   ```
   $ find xbinst -name '*.mcs'
   ```

5. Program the flash memory using the `xbsak flash` command and the `-m <`*`mcs_file`*`>` as follows:

- For a KCU board, the command returns two MCS files: the primary and the secondary MCS file.

```
$ xbsak flash -m <primary mcs file> -n <secondary mcs file>
```

- For a VCU board, the command returns the MCS file in the `firmware` directory.

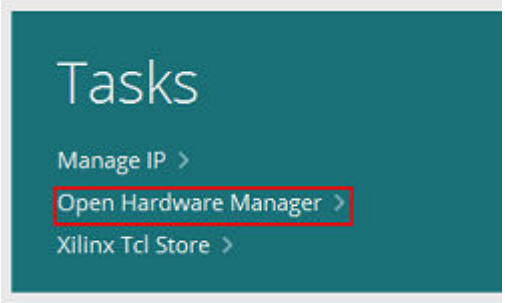
```
$ xbsak flash -m <mcs file>
```

Programming the Base Platform

The SDx™ system compiler targets a specific FPGA. 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 you load a user application.

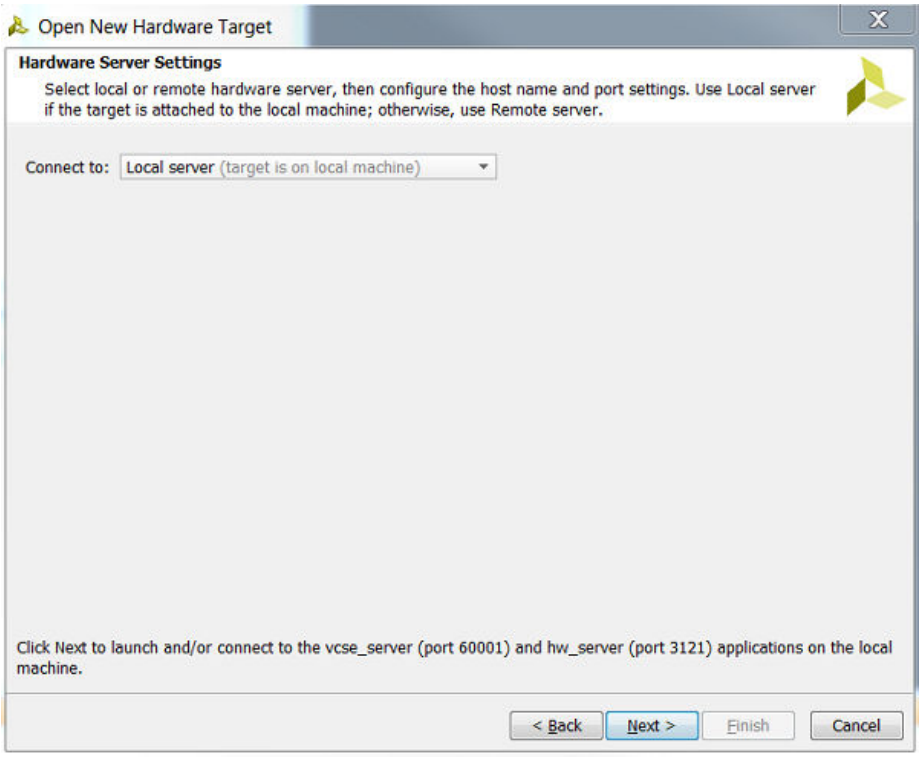
To program the firmware program:

1. Connect the board to the programming computer with an installation of Vivado® Design Suite using a USB cable.
2. Connect the Xilinx board and the programming computer with an installation of the Vivado Design Suite. On the programming computer, start the **Open Hardware Manager**.

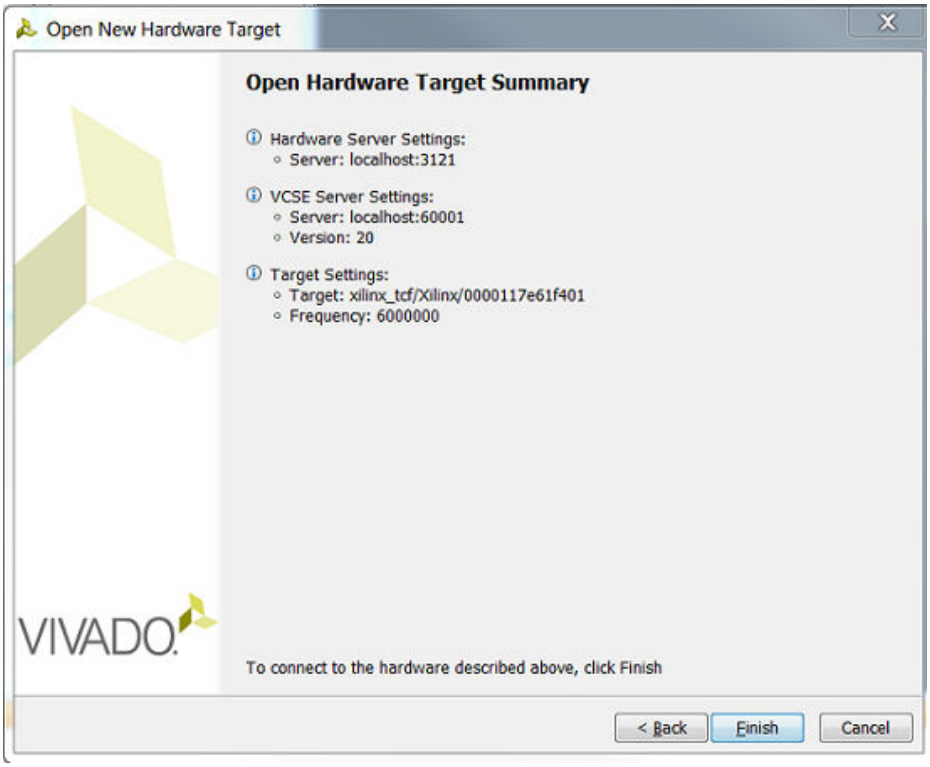


3. Select **Open a New Hardware Target.**

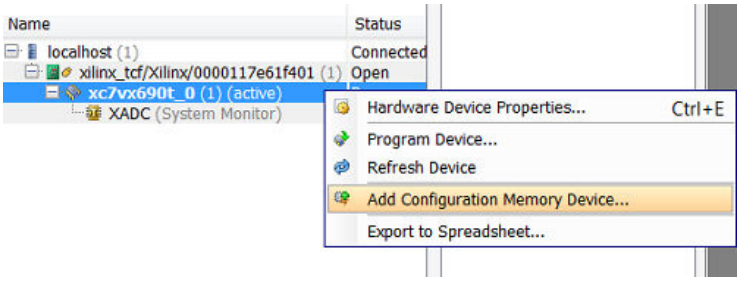
4. Click **Next** in the Open Hardware Target window.



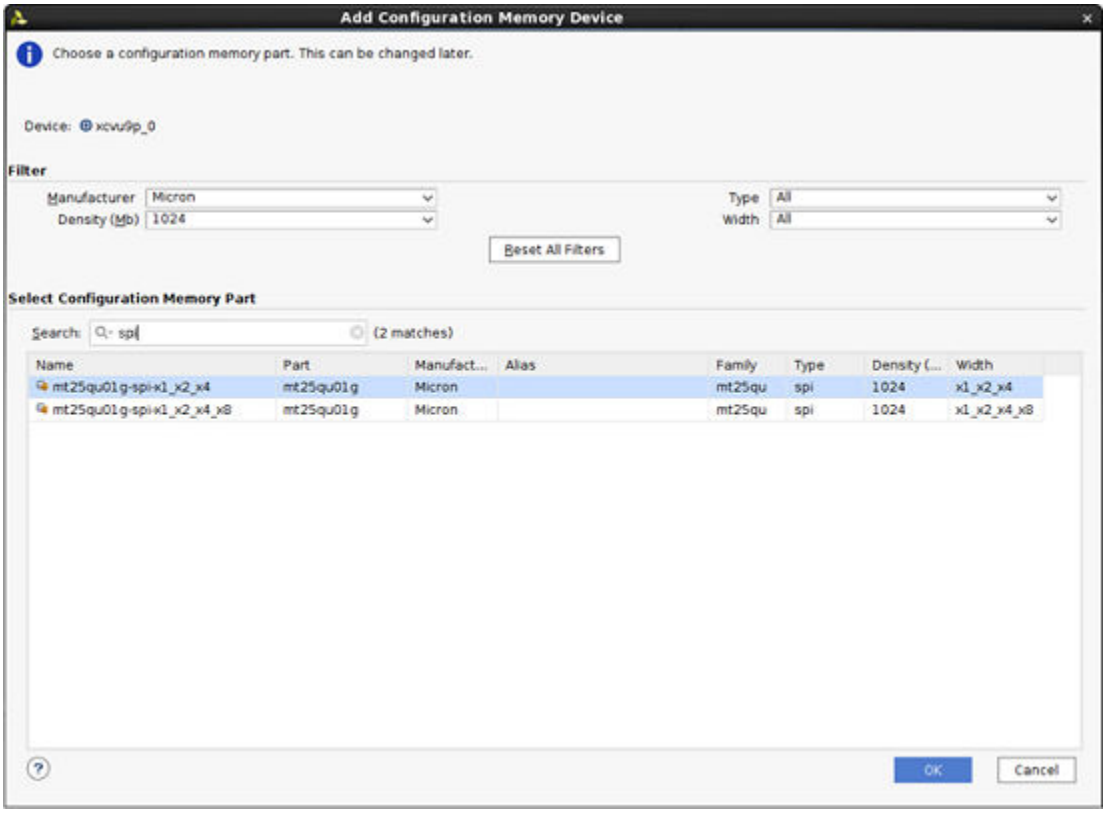
5. In the Open Hardware Target Summary window, click **Finish**.



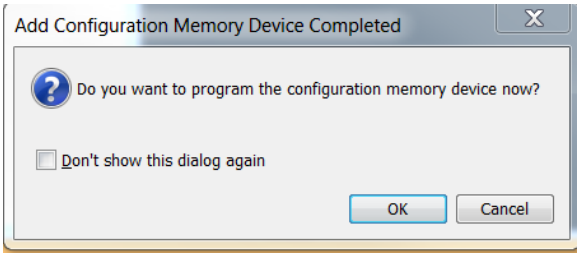
6. Right-click the FPGA (**vu9p_0**) and select **Add Configuration Memory Device**.



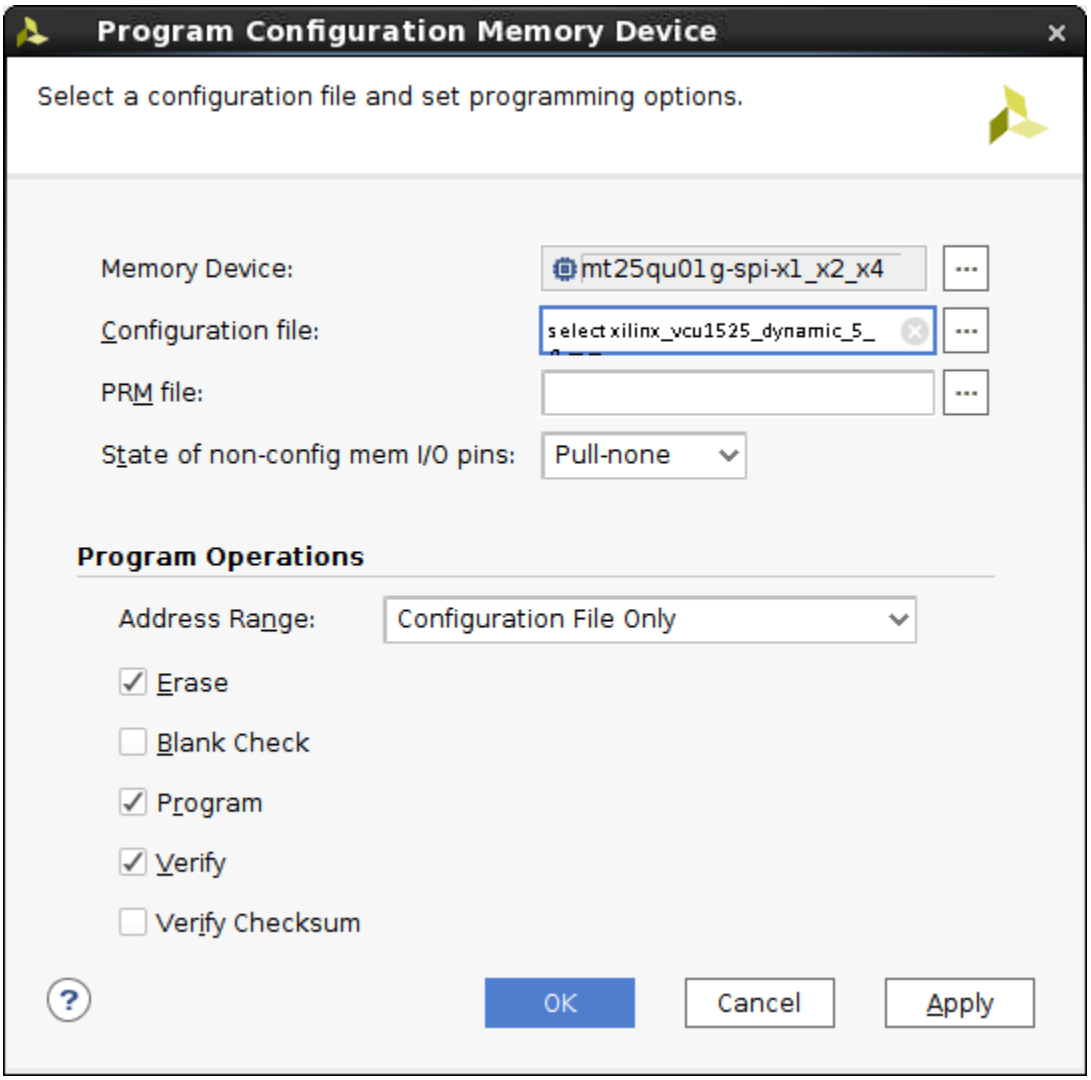
7. Select **mt25qu01g-spi-x1_x2_x4** as the configuration memory.



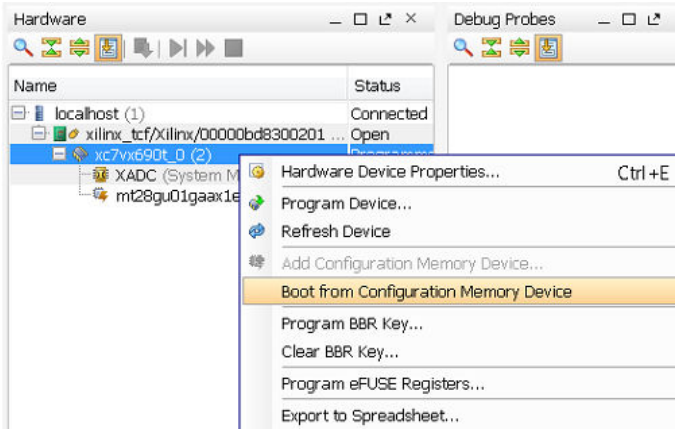
8. Click **OK** to program the configuration memory.



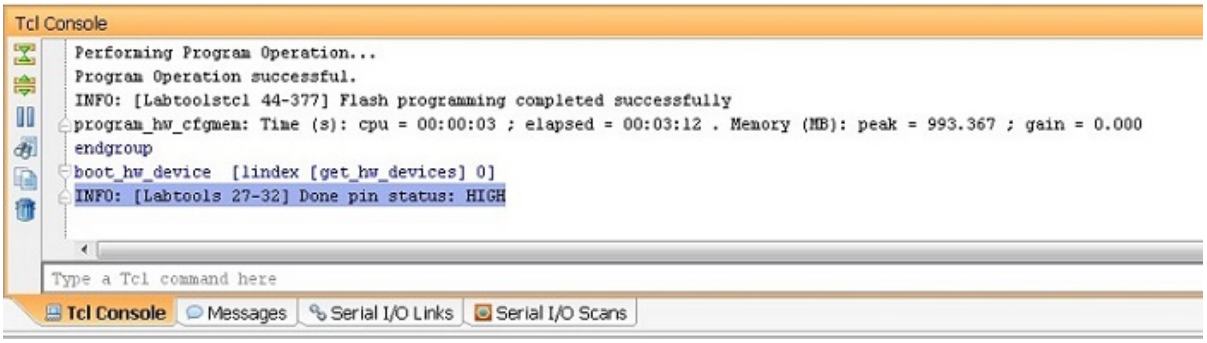
9. In the **Programming Configuration Memory Device** window, go to the **Configuration File** entry box, browse to, and select the primary MCS file (`xilinx_*.*.mcs`). Verify the settings, as shown in the Program Configuration Memory Device window, and click **OK** to start programming the configuration memory.



10. After the memory has been configured, right-click the FPGA (**xcku1500**) and select **Boot From Configuration Device**.



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



11. Reboot the host computer.

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

Debugging and Troubleshooting a Board

***Note*:** You can find newly-discovered information related to board installation in Answer Record ##.

This section covers debugging and troubleshooting. For software support and related software debug techniques, see *SDx Command and Utility Reference Guide* (UG1279).

This section lists the commands to run to query and investigate situations, such as:

- You receive a new board and want to do sanity checks.
- Your host application is abrubtly interrupted, then the board or the driver might be left in a stale or unknown mode.
- The deployment machine needs to be rebooted to set everything back to a working state.

The following commands are used for debug: The first two commands are Xilinx board utility tools that are further detailed in *SDx Command and Utility Reference Guide* (UG1279). The other command you might need to use are Linux command line tools: `lspci` and `dmesg`.

The Xilinx utilities for board installation tool are called:

- `xbsak`
- `xbinst`

Two common scenarios that require debugging are, the following:

- You receive a new board, but the board is not showing as a PCI™ device using the `lspci` command, then the board is probably not programmed, and you need to install its firmware as listed in the ???, after that, you can do sanity checks before using you application.

- You had a working board, and it is not behaving properly anymore after an accelerator function design was running on the board, it is probably an OS or driver issue that you need to resolve.

SDx Debug Command Options

`xbsak scan`: Scans for Xilinx PCIe device(s) and associated drivers for host machine. A normal output looks like the following:

```
$ xbsak scan
Linux:4.4.0-116-generic:#140-Ubuntu SMP Mon Feb 12 21:23:04 UTC 2018:x86_64
Distribution: Ubuntu 16.04.3 LTS
GLIBC: 2.23
---
XILINX_OPENCL="/opt/dsa/<board_name>/xbinst"
LD_LIBRARY_PATH="/opt/dsa/<board_name>/xbinst/runtime/lib/x86_64:"
---
[0]mgmt:0x6a8f:0x4351:[xclmgmt:2017.4.4:0]
[0]user:0x6a90:0x4351:[xocl:2017.4.5:129]
```

Determining if Output is Showing an Error

When you use the `xbsak scan` command, and get results like the following:

```
[0]mgmt:0x6a8f:0x4351:[???:???:???]
[0]user:0x6a90:0x4351:[xocl:2017.4.5:129]
```

Such output would indicate that the management driver, `xclmgmt` is not loaded; it could either have been unloaded using the Linux command, `$ sudo rmmod xclmgmt`, or there was an error at the OS level.

You can fix an unloaded management driver with the following command:

```
$ sudo modprobe xclmgmt
```

Otherwise, if you use `xbsak scan`, and you get results like the following:

```
[0]mgmt:0x6a8f:0x4351:[xclmgmt:2017.4.4:0]
[0]user:0x6a90:0x4351:[???:???:???]
```

This would indicate that the user function driver `xocl` is not loaded; it could have either have been unloaded with the `xbsak list $ sudo rmmod xocl` command or there was an error at the OS level.

You can fix this with the following command:

```
$ sudo modprobe xocl
```

dmatext Command

Use the `xbsak dmatest` to confirm that there is no issues accessing the DDR device memory from the host, or to ensure that the path: **Host → PCIe → Device → DDR** and corresponding reverse path **DDR Device → PCIe → Host**are still working and functioning as expected.

***Note*:** The `xbsak dmatest` also provides a PCIe bandwidth test/benchmark and confirmation that the board and host computer are operating at the optimum speed. The maximum performance is achieved with Gen3 by 16 lanes and you see write speeds of around 7GB/s and read speeds of around 11GB/s.

Output for VCU1525

A typical output for vcu1525 is:

```
$ xbsak dmatest
Linux:3.10.0-327.el7.x86_64:#1 SMP Thu Nov 19 22:10:57 UTC 2015:x86_64
Distribution: CentOS Linux release 7.2.1511 (Core)
GLIBC: 2.17
---
XILINX_OPENCL="/opt/dsa/<board_name>/xbinst"
LD_LIBRARY_PATH="/opt/dsa/<board_name>/xbinst/runtime/lib/x86_64:"
---
INFO: Found 1 device(s)
Total DDR size: 65536 MB
Reporting from mem_topology:
Data Validity & DMA Test on DDR[0]
INFO: Host -> PCIe -> MIG write bandwidth = 7284.55 MB/s
INFO: Host <- PCIe <- MIG read bandwidth = 11192 MB/s
Data Validity & DMA Test on DDR[1]
INFO: Host -> PCIe -> MIG write bandwidth = 7514.58 MB/s
INFO: Host <- PCIe <- MIG read bandwidth = 11523.6 MB/s
Data Validity & DMA Test on DDR[2]
INFO: Host -> PCIe -> MIG write bandwidth = 7539.88 MB/s
INFO: Host <- PCIe <- MIG read bandwidth = 11238.2 MB/s
Data Validity & DMA Test on DDR[3]
INFO: Host -> PCIe -> MIG write bandwidth = 7840.42 MB/s
INFO: Host <- PCIe <- MIG read bandwidth = 11264.1 MB/s
INFO: xbsak dmatest successful.
```

***Note*:** In the previous output, the memory topology of the programmed design is used the four memory banks on the card. Other designs, like the verify design only, use one DDR memory bank so the output would show only one channel.

You can use the `xbsak query` command to check the general status of the board. A typical output is:

```
---
INFO: Found 1 device(s)
DSA name:       xilinx_vcu1525_dynamic_5_0
Vendor:         10ee
Device:         6a8f
SDevice:        4350
SVendor:        10ee
DDR size:       0x4000000 KB
DDR count:      4
```

```
OnChip Temp:     35 C
Power(Beta):     **Unable to estimate power**
OCL Frequency:
        0:       300 MHz
        1:       500 MHz
PCIe:            GEN3 x 16
DMA bi-directional threads:    2
MIG Calibrated: true

Device DDR Usage:
  Bank[0].mem:  0x0 KB
  Bank[0].bo:   0
  Bank[1].mem:  0x0 KB
  Bank[1].bo:   0
  Bank[2].mem:  0x0 KB
  Bank[2].bo:   0
  Bank[3].mem:  0x0 KB
  Bank[3].bo:   0

Total DMA Transfer Metrics:
  Chan[0].h2c:  0x2d1ea5a KB
  Chan[0].c2h:  0x2c3383b KB
  Chan[1].h2c:  0x24a7a51 KB
  Chan[1].c2h:  0x232b000 KB

Firewall Last Error Status:
        0:       0x0 (GOOD)
        1:       0x0 (GOOD)
        2:       0x0 (GOOD)

Xclbin ID:  0x5ade7395

Mem Topology:
     Bank       Type        Base Address     Size (KB)
 [0] bank0      MEM_DDR4             0x0     0x1000000
 [1] bank2      MEM_DDR4    0x800000000     0x1000000
 [2] bank3      MEM_DDR4    0xc00000000     0x1000000
 [3] bank1      MEM_DDR4    0x400000000     0x1000000

Compute Unit Status:
  CU[0]: sdx_kernel_wizard_0@0x1800000 (IDLE)
INFO: xbsak query successful.
```

Failure to Create a Compute Program

You can check potential issues using the `clCreateProgramWithBinary()` function. This function can identify issues that occur when programming the FPGA with the provided `xclbin` file. One scenario is that the program is not be compatible with the currently programmed bitstream, which derived from the programmed firmware DSA on the configuration flash memory of the card.

There are several ways to assert the <what are we asserting?>, but they all need to check the timestamp programmed into the bitstream of the programmed DSA against the `xclbin`, by checking the output of `dmesg`:

```
xclbin: TimeStamp:5a27f562 VBNV:
ROM: TimeStamp:5aace1cf VBNV:<board_name>
TimeStamp of ROM did not match Xclbin
```

If you have access to the `xbinst` installation area used to program the device configuration memory, you can check the `dsabin` in the`firmware`directory: `/opt/dsa/<board_name>/xbinst/firmware/10ee-6a90-4350-00000000M<#######>.dsabin` shows the timestamp programmed into the device configuration memory.

If you have access to the `xclbin` file, you can use the `xclbinsplit` option to split the `xclbin` into several files:

On the deployment machine (after sourcing `setup.sh|csh`), type the following:

```
$ $XILINX_OPENCL/runtime/bin/xclbinsplit verify.xclbin
```

OR:

```
$ xclbinsplit verify.xclbin
```

When the SDAccel™ tool is setup on the development machine, type the following :

```
$ $XILINX_SDX/runtime/bin/xclbinsplit verify.xclbin
```

This produces a set of files for which the `split-xclbin.xml` lists the timestamp of the DSA that was used to create it:

```
 <platform vendor="xilinx" boardid="vcu1525" name="dynamic"
featureRomTime="#####">
```

The output is in decimal representation. You can convert to a hexadecimal representation, with the following:

```
$ printf "%x\n" 1512568162
5a27f562
```

Also, you can look directly into the DSA archive file for the same information by using the following command:

```
$ unzip $XILINX_SDX/platforms/<board_name>/hw/<board_name>.dsa dsa.xml -d
unzip_dsa
[...]  inflating: unzip_dsa/dsa.xml
$ grep -i time unzip_dsa/dsa.xml
< ... FeatureRomTimestamp="#####">
```

Useful Debig Operating System Debug Commands

The following are some typical outputs for a VCU1525 board when running debug commands. Only the relevant output relating to accelerator boards are shown. Some numbers would change depending on the deployment machine.

- `lspci`: Is a command on Unix-like operating systems that prints ("lists") detailed information about all PCIe® buses and devices in the system. It is based on a common portable library, `libpci`, that provides access to the PCIe configuration space on a variety of operating systems. With this command, you can check if the accelerator board has booted up correctly, has been recognized as a PCIe device, and has been enumerated. The `lspci` command lists the PCIecontroller addresses, as follows:

```
$ lspci
...
03:00.0 Serial controller: Xilinx Corporation Device 6a90
03:00.1 Serial controller: Xilinx Corporation Device 6a8f
...
```

***Note*:** Output lines for this command have been omitted for brevity here.

If you are a Xilinx® board (as opposed to a board in a cloud environment), you can query using only the *`vendor_ID`* (`10ee` for Xilinx). The comand looks like the following:

```
$ lspci -d 10ee:
03:00.0 Serial controller: Xilinx Corporation Device 6a90
03:00.1 Serial controller: Xilinx Corporation Device 6a8f
```

Where:

- `-d` is the device.
- `10ee` is the XilinxID.

you can get the more verbose output in a similar way:

```
$ lspci -vv -d 10ee:
03:00.0 Serial controller: Xilinx Corporation Device 6a90 (prog-if 01
[16450])
    Subsystem: Xilinx Corporation Device 4351
    Control: I/O+ Mem+ BusMaster+ SpecCycle- MemWINV- VGASnoop- ParErr-
```

```
Stepping- SERR- FastB2B- DisINTx-
    Status: Cap+ 66MHz- UDF- FastB2B- ParErr- DEVSEL=fast >TAbort-
<TAbort- <MAbort- >SERR- <PERR- INTx-
    Latency: 0, Cache Line Size: 64 bytes
    Interrupt: pin A routed to IRQ 46
    Region 0: Memory at f6000000 (32-bit, non-prefetchable) [size=32M]
    Region 1: Memory at f8040000 (32-bit, non-prefetchable) [size=64K]
    Capabilities: <access denied>
    Kernel driver in use: xocl
    Kernel modules: xocl

03:00.1 Serial controller: Xilinx Corporation Device 6a8f (prog-if 01
[16450])
    Subsystem: Xilinx Corporation Device 4351
    Control: I/O+ Mem+ BusMaster- SpecCycle- MemWINV- VGASnoop- ParErr-
Stepping- SERR- FastB2B- DisINTx-
    Status: Cap+ 66MHz- UDF- FastB2B- ParErr- DEVSEL=fast >TAbort-
<TAbort- <MAbort- >SERR- <PERR- INTx-
    Region 0: Memory at f4000000 (32-bit, non-prefetchable) [size=32M]
    Region 2: Memory at f8020000 (32-bit, non-prefetchable) [size=128K]
    Region 4: Memory at f8000000 (32-bit, non-prefetchable) [size=128K]
    Capabilities: <access denied>
    Kernel driver in use: xclmgmt
    Kernel modules: xclmgmt
```

```
//Using the verbose option shows:
$ lspci -v
03:00.0 Serial controller: Xilinx Corporation Device 6a90 (prog-if 01
[16450])
    Subsystem: Xilinx Corporation Device 4351
    Flags: bus master, fast devsel, latency 0, IRQ 46
    Memory at f6000000 (32-bit, non-prefetchable) [size=32M]
    Memory at f8040000 (32-bit, non-prefetchable) [size=64K]
    Capabilities: <access denied>
    Kernel driver in use: xocl
    Kernel modules: xocl

03:00.1 Serial controller: Xilinx Corporation Device 6a8f (prog-if 01
[16450])
    Subsystem: Xilinx Corporation Device 4351
    Flags: fast devsel
    Memory at f4000000 (32-bit, non-prefetchable) [size=32M]
    Memory at f8020000 (32-bit, non-prefetchable) [size=128K]
    Memory at f8000000 (32-bit, non-prefetchable) [size=128K]
    Capabilities: <access denied>
    Kernel driver in use: xclmgmt
    Kernel modules: xclmgmt
```

When required, you can run the `lspci` command in super-user (`sudo`) mode to show the information hidden under the access-denied entries. The output would be as follows:

```
$ sudo lspci -v
...
03:00.0 Serial controller: Xilinx Corporation Device 6a90 (prog-if 01
[16450])
    Subsystem: Xilinx Corporation Device 4351
    Flags: bus master, fast devsel, latency 0, IRQ 46
    Memory at f6000000 (32-bit, non-prefetchable) [size=32M]
    Memory at f8040000 (32-bit, non-prefetchable) [size=64K]
    Capabilities: [40] Power Management version 3
```

```
    Capabilities: [60] MSI-X: Enable+ Count=33 Masked-
    Capabilities: [70] Express Endpoint, MSI 00
    Capabilities: [100] Advanced Error Reporting
    Capabilities: [1c0] #19
    Capabilities: [350] Vendor Specific Information: ID=0001 Rev=1 Len=02c
<?>
    Capabilities: [400] Access Control Services
    Kernel driver in use: xocl
    Kernel modules: xocl

03:00.1 Serial controller: Xilinx Corporation Device 6a8f (prog-if 01
[16450])
    Subsystem: Xilinx Corporation Device 4351
    Flags: fast devsel
    Memory at f4000000 (32-bit, non-prefetchable) [size=32M]
    Memory at f8020000 (32-bit, non-prefetchable) [size=128K]
    Memory at f8000000 (32-bit, non-prefetchable) [size=128K]
    Capabilities: [40] Power Management version 3
    Capabilities: [70] Express Endpoint, MSI 00
    Capabilities: [100] Advanced Error Reporting
    Capabilities: [400] Access Control Services
    Kernel driver in use: xclmgmt
    Kernel modules: xclmgmt
```

- `dmesg`: Lets you view the messages from the drivers. The command line looks like the following:

```
$ dmesg
```

Another `dmesg` option is `dmesg -T`. This option provides a timestamp that is human-readable instead of using the time in seconds since the machine has booted.

If the `dmesg` command returns too much information, you can use the following variation, which clears the buffer before the next use:

```
$ sudo dmesg -C
```

- `lsmod`: Lets you check if driver modules are loaded in the OS, as follows:

```
lsmod | grep -E "^xocl|^xclmgmt"
```

The output would be similar to the following:

```
xocl     94208  0
```

```
xclmgmt  69632  0
```

- The first column shows the `xocl` and `xclmgmt` that are driver modules.
- The second column is the size in memory of the drivers.

- The third column with the 0 indicates the drivers are currently not in use, also indicating that no application is running on the host and using or accessing the accelerator board.

Other OS Commands

You can insert or remove drivers as shown before with the `modprobe, rmmod` functions.

Appendix G

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 provides access to Xilinx documents, videos, and support resources, which you can filter and search to find information. To open the Xilinx Documentation Navigator (DocNav):

- From the SDAccel™ or SDSoC™ IDE, select **Help** → **Documentation and Tutorials**.
- On Windows, select **Start** → **All Programs** → **Xilinx Design Tools** → **DocNav**.
- At the Linux command prompt, enter `docnav`.

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

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

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

References

1. *SDx 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 Getting Started Tutorial* (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