# SDAccel Environment User Guide

**UG1023 (v2018.3) January 24, 2019**

See all versions of this document

XILINX®

# Revision History

The following table shows the revision history for this document.

| Section | Revision Summary |
|---|---|
| **01/24/2019 Version 2018.3** | |
| Best Practices for Acceleration with SDAccel | Updated description. |
| XOCC Linking and Compilation Options | Added SLR link. |
| Kernel Language Support | Added SystemVerilog. |
| Setting Up the Runtime | Updated #5 description. |
| Controlling Implementation Results | Updated description. |
| Controlling Report Generation | Updated description. |
| Debugging Features and Techniques | Added notes to table. |
| Chapter 9: RTL Kernels | Updated description. |
| Requirements for Using an RTL Design as an RTL Kernel | Updated description. |
| Kernel Interface Requirements | Updated description. |
| Kernel Software Requirements | Updated description. |
| Interrupt | Updated description. |
| Scalar Arguments | Updated Argument type description. |
| RTL Kernel Type Project Flow | Resized figures. |
| Managing Clocks in an RTL Kernel | Added section. |
| Creating SDAccel Kernels with Vivado HLS | Updated figures and description. |
| Xclbinutil Utility | Updated description. |
| Clocks | Added link and updated code. |
| **12/05/2018 Version 2018.3** | |
| Execution Model of an SDAccel Application | Updated figure. |
| SDAccel Design Methodology | Moved tables to ordered list and updated description. |
| Best Practices for Acceleration with SDAccel | Updated description. |
| Chapter 3: Creating an SDAccel Project | Updated description and figures. |
| Creating an Application Project | Updated section. |
| Understanding the SDx GUI | Updated figure. |
| SDx Assistant | Updated figures. |
| XOCC Linking and Compilation Options | Added section. |
| Exporting an SDx Project | Updated Export Filename figure. |
| Importing an SDx Project | Added Import Type figure. |
| Coding the Host Application | Updated description. |
| Writing C/C++ Kernels | Updated description and split Pointer Arguments and Scalars into sections. |
| Chapter 5: Building the System | Updated Project Editor View figure and updated section headings. |

| Section | Revision Summary |
|---|---|
| Building the FPGA Binary | Cleaned up the sample codes with $ and updated SDAccel Linking, Instantiate Multiple Compute Units, and Assistant XOCC Compile Settings figures. Removed Assistant Top-Level figure. |
| Mapping Kernel Interfaces to Memory Resources | Updated description. |
| Allocating Compute Units to SLRs | Updated description. |
| Controlling Implementation Results | Updated description. |
| Controlling Report Generation | Updated description. |
| System Estimate Report | Updated title and first paragraph. |
| Profile Summary Report | Updated description. |
| Application Timeline | Updated description, tip, and links. |
| Waveform View and Live Waveform Viewer | Updated description. |
| Kernel SLR and DDR Memory Assignments | Added section. |
| Debugging Features and Techniques | Updated tables. |
| System | Updated description. |
| Chapter 8: Command Line Flow | Updated description. |
| Compiling | Updated description. |
| Kernel Code Compilation and Linking | Updated descriptions in the sections. |
| Using the sdaccel.ini File | Split into three tables. |
| Kernel Software Requirements | Updated section. |
| Kernel Interface Requirements | Updated description and S_AXI_CONTROL entry and added registers. |
| Interrupt | Added link. |
| RTL Kernel Wizard | Updated description. |
| Kernel Options | Added description. |
| Streaming Interfaces | Added section. |
| Package RTL Kernel into Xilinx Object File | Updated description. |
| Chapter 10: HLS Kernel Design Integration into SDAccel | Added chapter. |
| Appendix C: Useful Command Line Utilities | Added appendix. |
| SLR Assignments for Kernels | Removed appendix. |
| **10/02/2018 Version 2018.2.xdf** | |
| Chapter 1: SDAccel Introduction and Overview | Updated supported platforms to include the Alveo™ U200 and U250 Data Center accelerator cards. |
| Throughout document. | Replaced `xbsak` and `xbinst` commands with `xbutil` command. |
| Creating an Application Project | Added a note regarding the need to separately install the required platforms. |
| Appendix E: Migrating to a New Target Platform | Added appendix. |
| Mapping Kernel Interfaces to Memory Resources | Added note regarding using the `--sp` option to assign all interfaces/ports of a kernel. |

| Section | Revision Summary |
|---|---|
| **08/13/2018 Version 2018.2** | |
| Appendix F: Installing and Debugging a Board | Removed Board Installation Appendix.<br>Board Installation procedures can be found in the corresponding Board User Guides.<br>See *KCU1500 Board User Guide* (UG1260)and *VCU1525 Reconfigurable Acceleration Platform User Guide* (UG1268). |
| **07/02/2018 Version 2018.2** | |
| Throughout Document | Minor edits. |
| Board Installation and Debug Procedure | Modified content to a command line-based installation flow. |
| Chapter 9: RTL Kernels | Removed redundant topic. |
| Chapter 7: Chapter 7: Debugging Applications and Kernels | Removed references to MicroBlaze Debug. |
| **06/20/2018 Version 2018.2** | |
| Installing, Programming, 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 Building the Host Application and Building the FPGA Binary 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: RTL Kernels | Significant updates to this content. |
| Appendix B: Directory Structure | Describes the directory structure of application projects. |
| SLR Assignments for Kernels | Discusses the use of `--xp` to control kernel placement. |
| Appendix F: 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

XILINX

*Chapter 1*

# 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 gives you 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 runtime 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 Alveo™ U200 and U250 Data Center accelerator cards. 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 available 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, such as 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 integrated into your accelerated application project quickly 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 re-target 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.

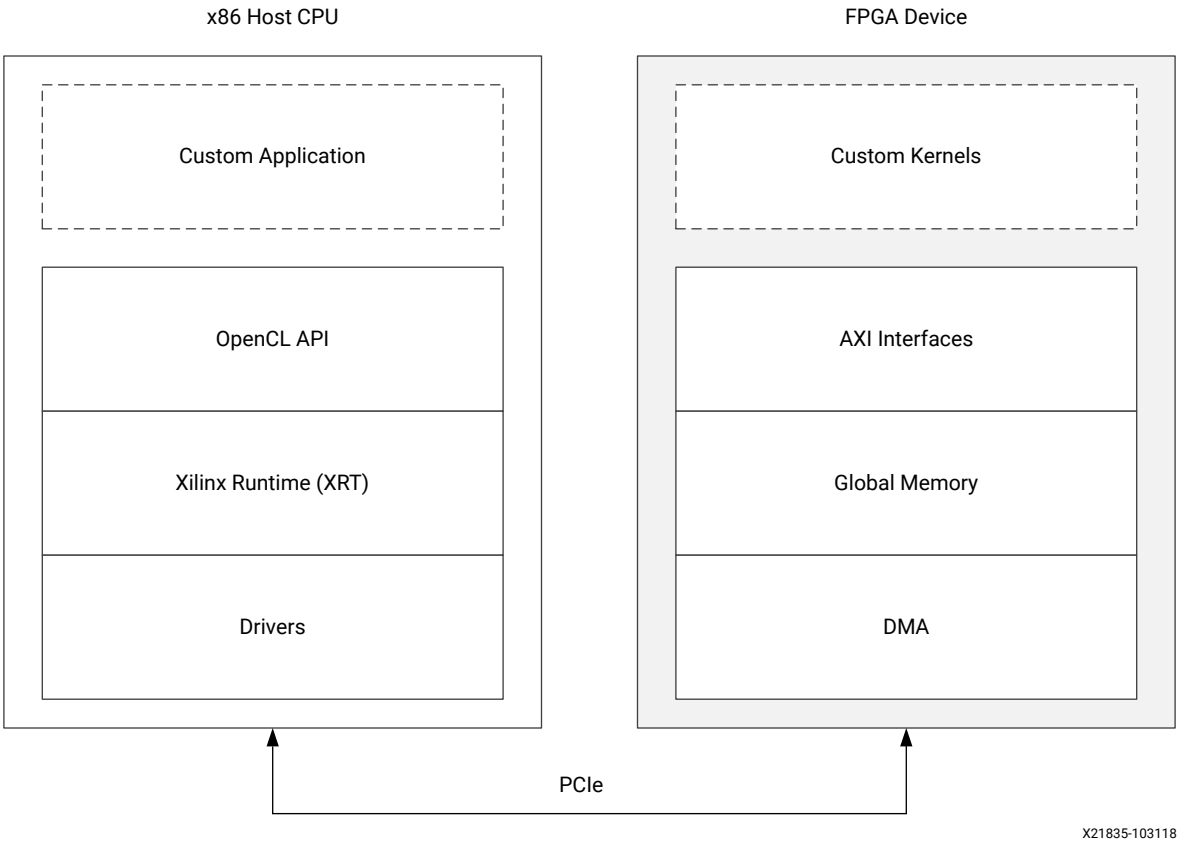
- **Pipelining:** A technique to increase instruction-level parallelism in the hardware implementation of an algorithm by overlapping independent stages of operations or functions. The data dependence in the original software implementation is preserved for functional equivalence, but the required circuit 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 a Xilinx FPGA is the ability to be dynamically reconfigured. For example, loading a compiled program into a processor or reconfiguring the FPGA during runtime can re-purpose the resources of the FPGA to implement additional kernels as the accelerated application runs. This allows 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 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 Xilinx runtime (XRT) manages those interactions. The application is written in C/C++ using OpenCL APIs. The custom kernels are running within a Xilinx FPGA with the XRT managing interactions between the host application and the accelerator. Communication between the host x86 machine and the accelerator board occurs across the PCIe bus.

The SDAccel hardware platform contains global memory banks. The data transfer between the host machine and kernels, in either direction, occurs 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 is 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 attached device through the PCIe interface.
2. The host application 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 while reading and writing data from global memory, as necessary.
5. The 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; this can occur between different types of kernels or multiple instances of the same kernel. The XRT transparently orchestrates the communication between the host application and the kernels in the accelerator. The number of instances of a kernel is determined by compilation options.

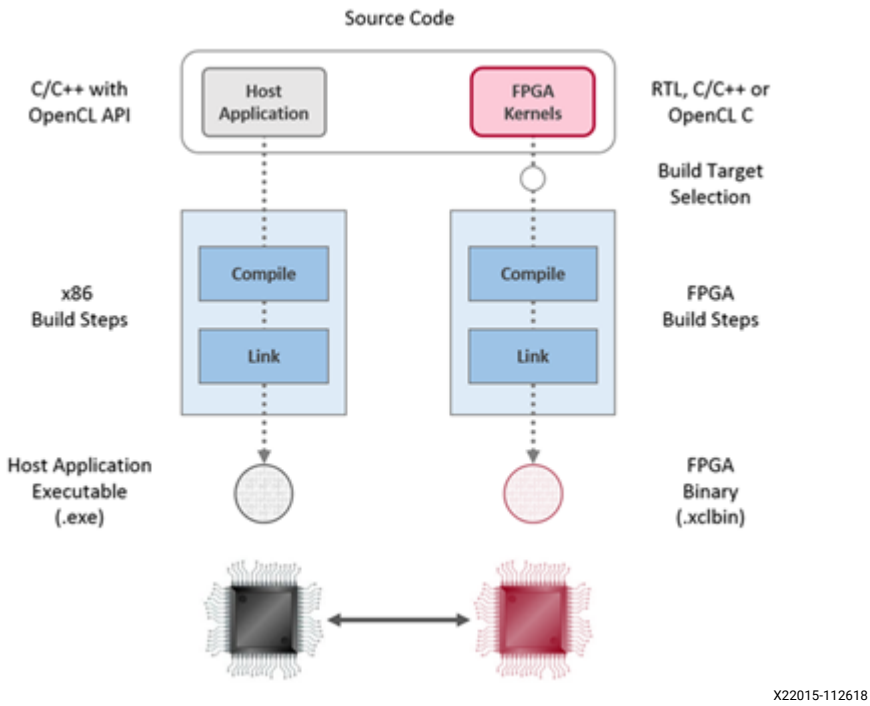
# SDAccel Build Process

The SDAccel environment offers all of the features of a standard software development environment:

- Optimized compiler for host applications
- Cross-compilers for the FPGA
- Robust debugging environment to help identify and resolve issues in the code
- Performance profilers to identify bottlenecks and optimize the code

Within this environment, the build process uses a standard compilation and linking process for both the software elements, and the hardware elements of the project. As shown in the following figure, the host application is built through one process using standard GCC compiler, and the FPGA binary is built through a separate process using the Xilinx `xocc` compiler.

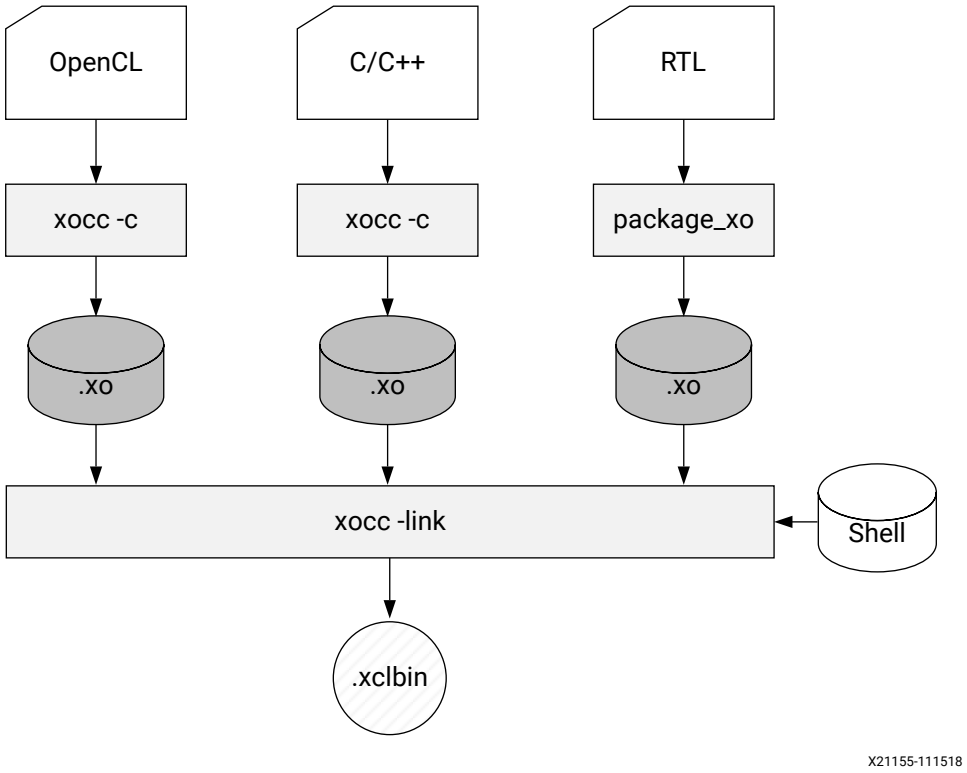
*Figure 2:* **Software/Hardware Build Process**




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 is highlighted in the following figure:
   - 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. 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 (shell) 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 (actual hardware), `xocc` generates the FPGA binary for the device leveraging the Vivado Design Suite to run synthesis and implementation.

*Figure 3:* **FPGA Build Process**




***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 predefined 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 tool 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.

The SDAccel tool 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 for identifying syntax errors, performing source-level debugging of the kernel code running together with application, and verifying the behavior of the system.
- **Hardware Emulation (`hw_emu`):** The kernel code is compiled into a hardware model (RTL) which is run in a dedicated simulator. This build and run loop takes longer but provides a detailed, cycle-accurate, view of kernel activity. This target is useful for testing the functionality of the logic that will go in the FPGA and 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. Kernel languages are not specific to the methodology. A software-centric flow can also use either C/C++, OpenCL, or RTL for kernel. The main differences between the two approaches are the starting point (software application or kernels) and the emphasis that comes with it.

The two use cases can be combined, allowing 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 code written as a standard software program, with some attention to the specific architecture of the code. For more information see the *SDAccel Environment Profiling and Optimization Guide* (UG1207). The software development flow typically uses the following steps:

1. Profile application: Baseline the application in terms of functionalities and performance and isolate functions to be accelerated in hardware.

Functions that consume the most execution time are good candidates to be offloaded and accelerated onto FPGAs.

2. 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.

3. 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
   - Visual timeline showing host calls and kernel start/stop times
4. 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

Hardware-centric flows first focuses on developing and optimizing the kernel(s) and typically leverages advanced FPGA design techniques. For more information, see the *SDAccel Environment Profiling and Optimization Guide* (UG1207). The hardware-centric development flow typically uses the following steps:

1. Baseline the application in terms of functionalities and performance and isolate functions to be accelerated in hardware.
2. Estimate cycle budgets and performance requirements to define accelerator architecture and interfaces.
3. Develop accelerator.
4. Verify functionality and performance. Iterate as needed.
5. Optimize timing and resource utilization. Iterate as needed.
6. Import kernel into SDAccel.
7. Develop sample host code to test with a dummy kernel having the same interfaces as the actual kernel.

8. Verify kernel works correctly with host code using hardware emulation, or running on actual hardware. Iterate as needed.
9. 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

Below are some specific things to keep in mind when developing your application code and hardware function in the SDAccel environment. You can find additional information in the *SDAccel Environment Profiling and Optimization Guide* (UG1207).

- Look to accelerate functions that have a high ratio of compute time to input and output data volume. Compute time can be greatly reduced using FPGA kernels, but data volume adds transfer latency.
- Accelerate functions that have a self-contained control structure and do not require regular synchronization with the host.
- Transfer large blocks of data from host to global device memory. One large transfer is more efficient than several smaller transfers. Run a bandwidth test to find the optimal transfer size.
- Only copy data back to host when necessary. Data written to global memory by a kernel can be directly read by another kernel. Memory resources include PLRAM (small size but fast access with lowest latency), HBM (moderate size and access speed with some latency), and DDR (large size but slow access with high latency).
- Take advantage of the multiple global memory resources to evenly distribute bandwidth across kernels.
- Maximize bandwidth usage between kernel and global memory by performing 512-bit wide bursts.
- Cache data in local memory within the kernels. Accessing local memories is much faster than accessing global memory.
- In the host application, use events and non-blocking transactions to launch multiple requests in a parallel and overlapping manner.
- In the FPGA, use different kernels to take advantage of task-level parallelism and use multiple CUs to take advantage of data-level parallelism to execute multiple tasks in parallel and further increase performance.
- Within the kernels take advantage of tasks-level with dataflow and instruction-level parallelism with loop unrolling and loop pipelining to maximize throughput.
- Some Xilinx FPGAs contain multiple partitions called super logic regions (SLRs). Keep the kernel in the same SLR as the global memory bank that it accesses.

- Use software and hardware emulation to validate your code frequently to make sure it is functionally correct.
- Frequently review the SDAccel Guidance report as it provides clear and actionable feedback regarding deficiencies in your project.

# Getting Started

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

You can find details on downloading and working with design examples from GitHub in Appendix A: Getting Started with Examples. In addition, 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).

***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 SDAccel 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.

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

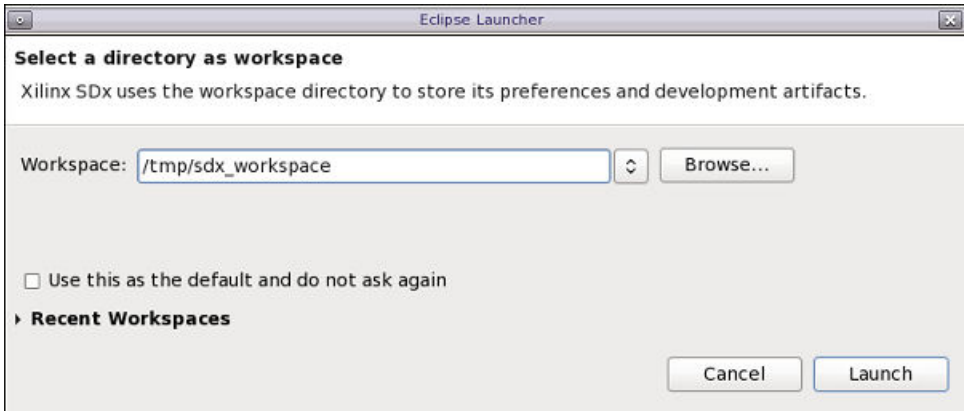
## Using an SDx Workspace

1. Launch the SDx IDE directly from the command line:

```
$ sdx
```

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

*Figure 4:* **Specify the SDx Workspace**




**IMPORTANT!** *When opening a new shell to enter an SDx command, ensure that you first source the* `settings64` *and* `setup` *scripts to set up the tool environment. See the SDAccel 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 for running the SDx tools and the* `xbutil` *board installation utility. Running both tools from the same terminal adversely affects the environment variables and causes tool issues.*

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 **Launch**.

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

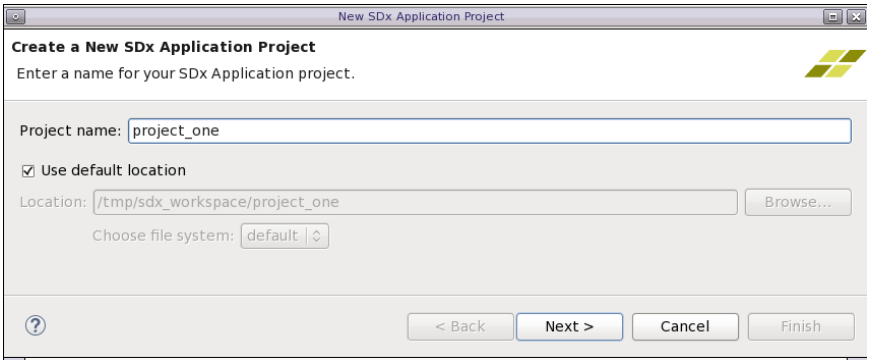
You have now created an SDx workspace and can populate the workspace with projects. Platform and application projects are created to describe the SDx tool flow for creating an SDAccel platform.

# 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.*

1. After launching the SDx IDE you can create a new Project. Select **File → New → SDx Application Project**, or if this is the first time the SDx IDE has been launched, you can select **Create Application Project** on the Welcome screen.
2. The New SDx Project wizard opens.
3. In the Create a New SDx Application Project page, you can specify the project name as shown. Specify the name of the project in the **Project name** field.

*Figure 5:* **Create a New SDx Application Project**




4. 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 project is created in a **Location** of your choice.
5. 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**).

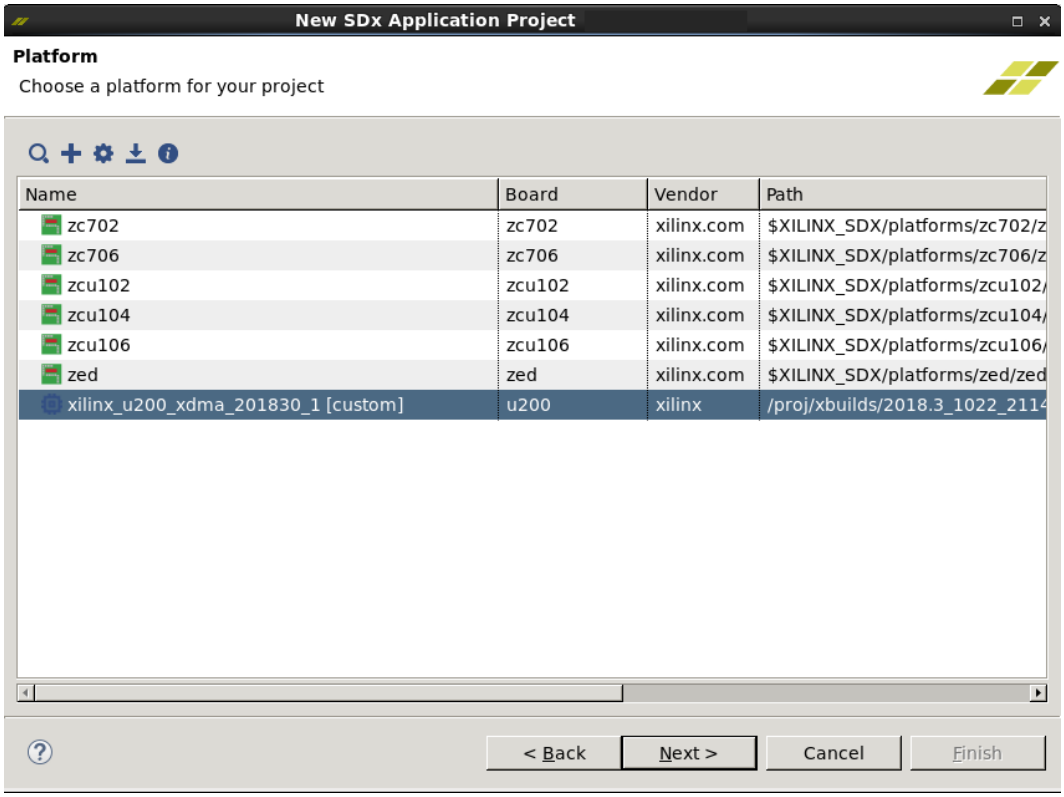
**IMPORTANT!** *The project location cannot be a parent folder of an SDx workspace.*

6. Click **Next**.

The Platform dialog box, similar to the one shown in the following figure, displays the available installed platforms. For installing additional platforms, see the "Installing Platform-Specific Packages" section in *SDAccel Environments Release Notes, Installation, and Licensing Guide* (UG1238).

**IMPORTANT!** *Be sure to select the right platform for your project, as subsequent processes are driven by this choice.*

*Figure 6:* **Specify SDAccel Platform**



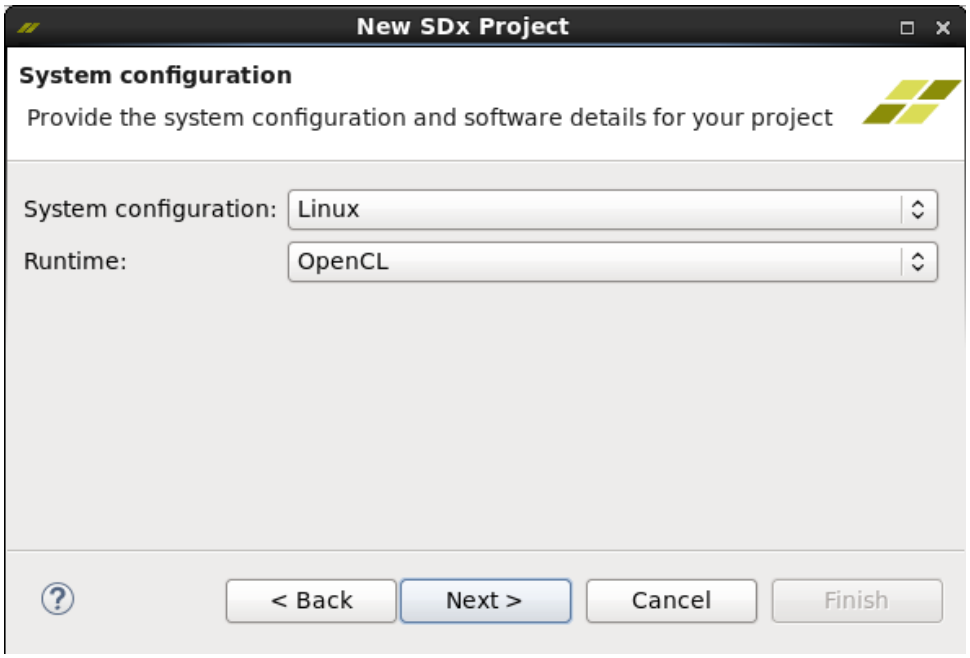

A platform is composed of a shell, which describes the base hardware design and the meta-data used in attaching accelerators to declared interfaces. SDAccel offers platforms for specific boards based on the following:

- Alveo™
  - xilinx_u200
  - xilinx_u250
- Virtex® UltraScale+™
  - vcu1525

You can add custom defined or third-party platforms into a repository. See Appendix D: Managing Platforms and Repositories for more information.

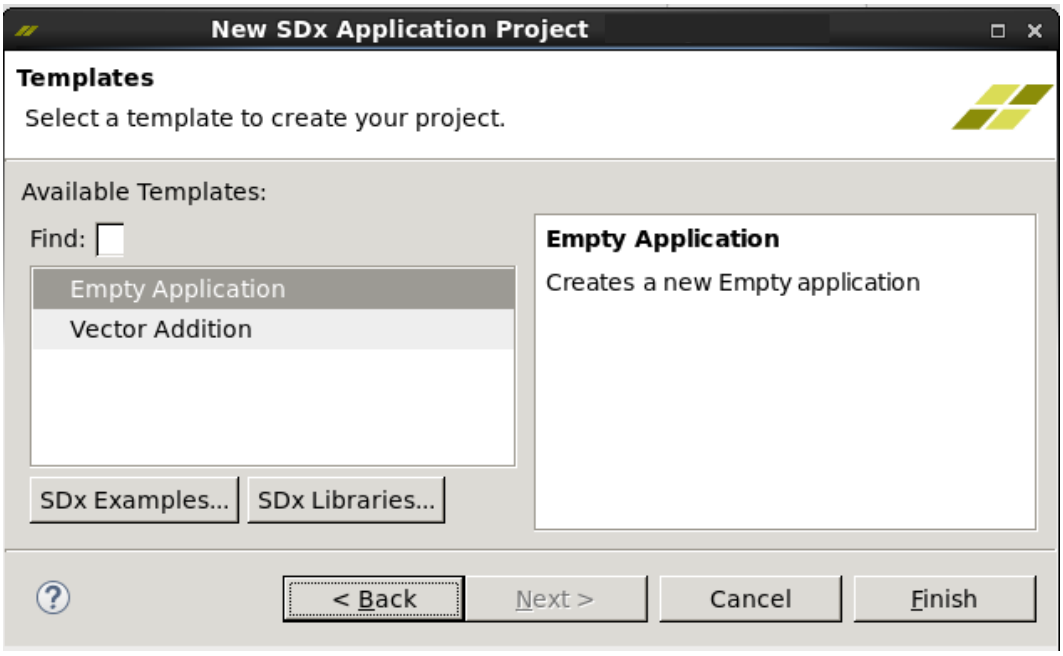
1. To select the target platform for your project, select the desired platform and click **Next**.
2. The System configuration page opens, as shown in the following figure. Select **System configuration** and **Runtime** from a list of those defined for the selected platform. The System Configuration defines the software environment that runs on the hardware platform. It specifies the operating system and the available runtime settings for the processors in the hardware platform.

*Figure 7:* **Specify System Configuration**




3. After selecting the **System Configuration** and clicking **Next**, the Templates page displays, as shown in the following figure. Specify an application template for your new project. The `samples` directory within the SDx tools installation contains multiple source code example templates.
4. Initially, the Template dialog box has an **Empty Application** and a **Vector Addition** application. To access additional SDAccel examples, click the **SDx Examples** button.

*Figure 8:* **Application Templates**




5. 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.

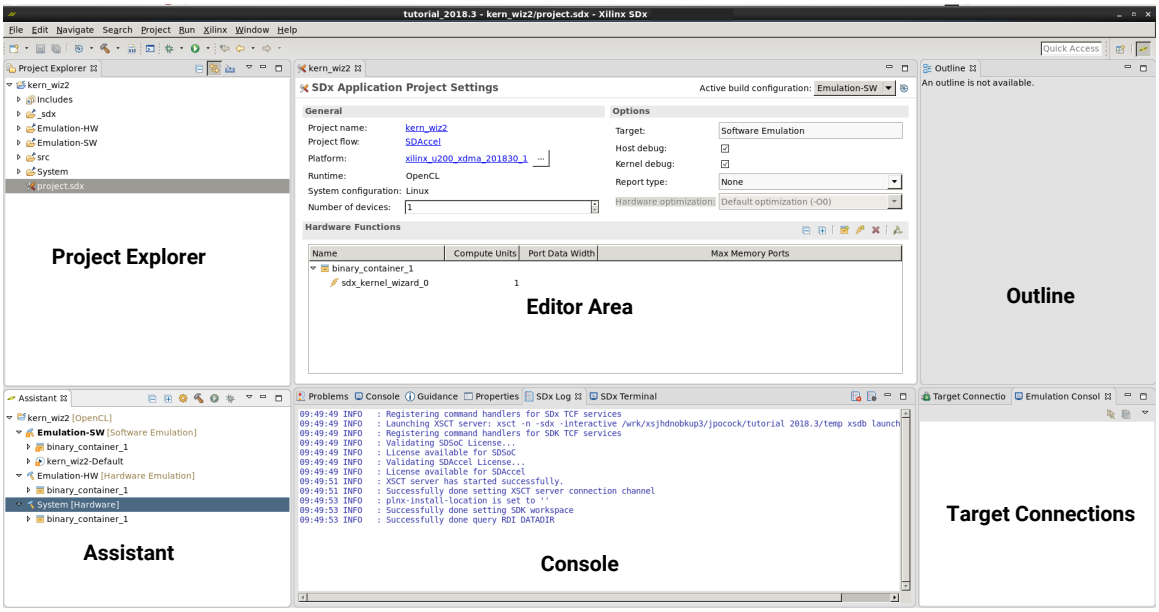
***Note*:** The SDx tool might try to download the examples from a GitHub repo depending on your network configuration. Specific proxy settings might be necessary.

6. 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.
7. Click **Finish** to close the New SDx Project wizard and open the project.

# Understanding 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 defines 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**.

To open different perspectives, select **Window** → **Perspective** → **Open Perspective**.

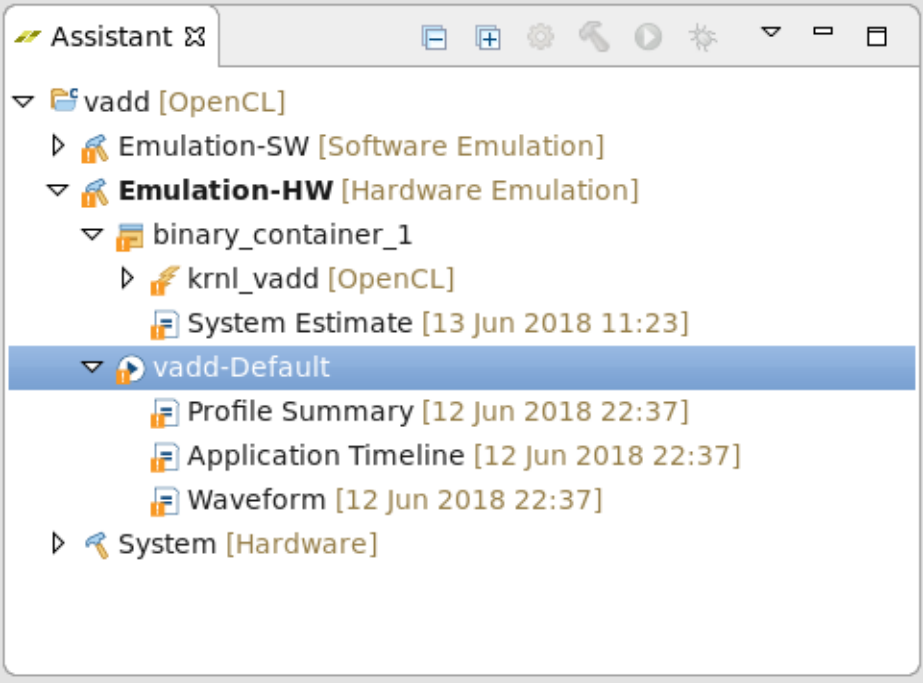
To restore the SDx (default) perspective, click the SDx button on the right side of the main toolbar.

# 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. For the expanded Emulation-HW flow, it shows the binary container contents and the debug reports including the **Profile Summary**, **Application Timeline**, and **Waveform**.

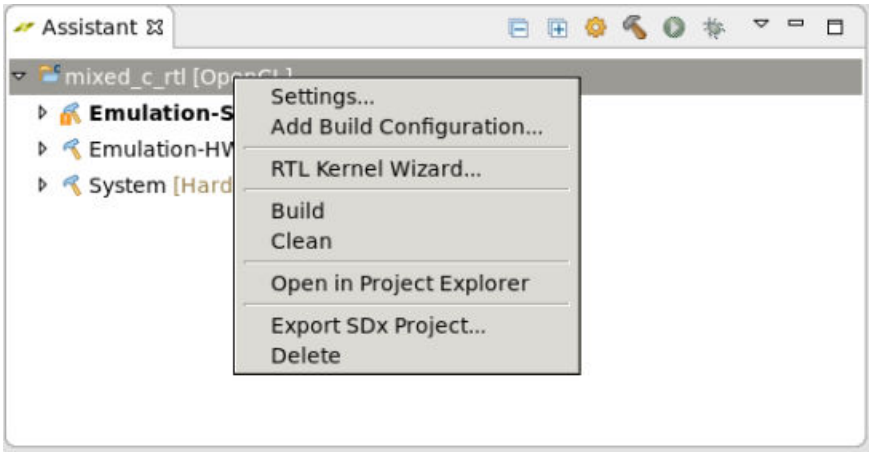
*Figure 10:* **Assistant Tree Structure Example**




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 project (**mixed_c_rtl** in the following example) displays the following menu:

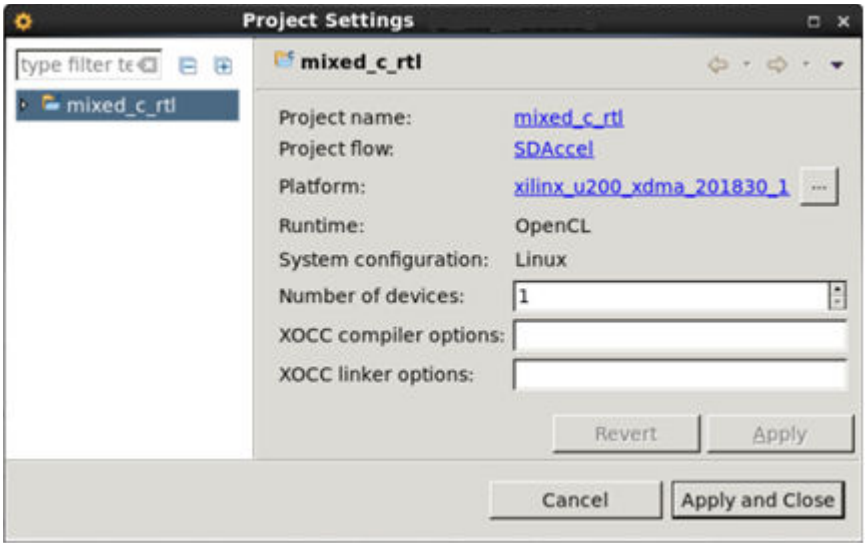
*Figure 11:* **Assistant Right-Click**




**TIP:** *If the actions are disabled/grayed out, this means the project does not have the applicable information.*

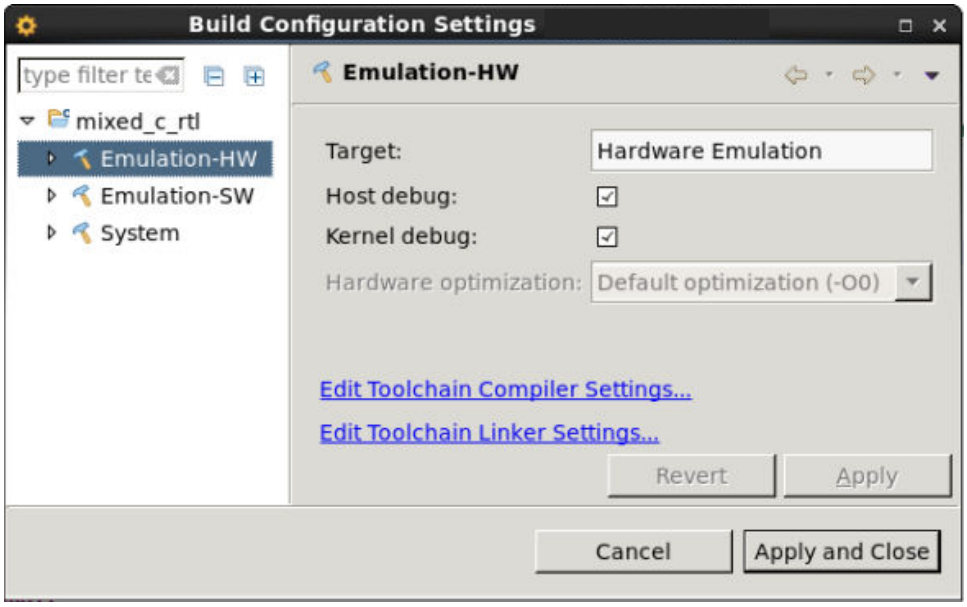
Select **Settings** to open the Project Settings dialog box.

*Figure 12:* **Project Settings**



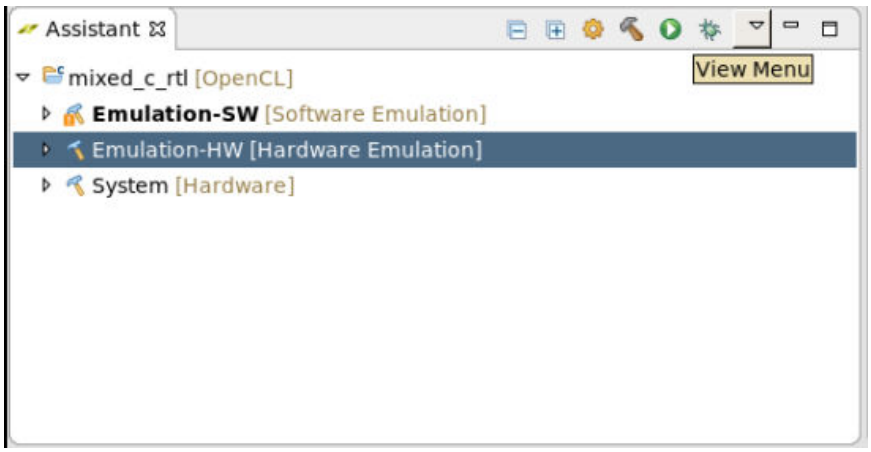

You can select the settings for the various items listed in the tree. For example, if you select the setting for the **Emulation-HW** build configuration, it displays the following. The Assistant makes it easy to navigate through the design objects and view/update their settings.

*Figure 13:* **Emulation-HW**




The **View** menu includes options that affect the Assistant view behavior, but do not affect project data. Select this option 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 open in the tree only when they exist in the project, usually after a project has been built or run, with specific settings.

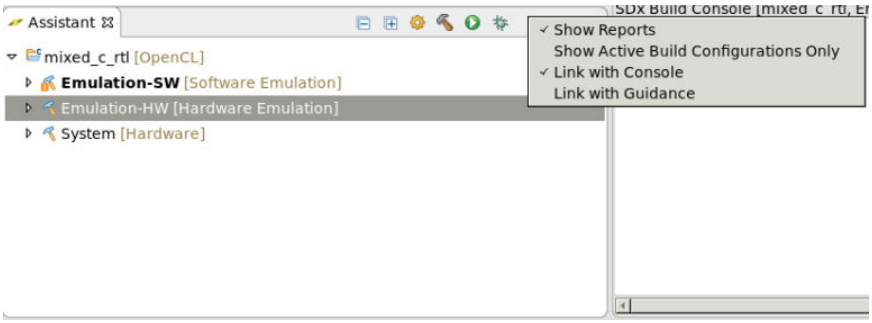
- **Show Active Build Configurations Only:** If checked, the tree will only show the "active" build configurations for each project. The active build configuration, in the assistant, will be the most-recently-built configuration. The active build configuration can also be changed to view the standard 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 switches 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 does 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**

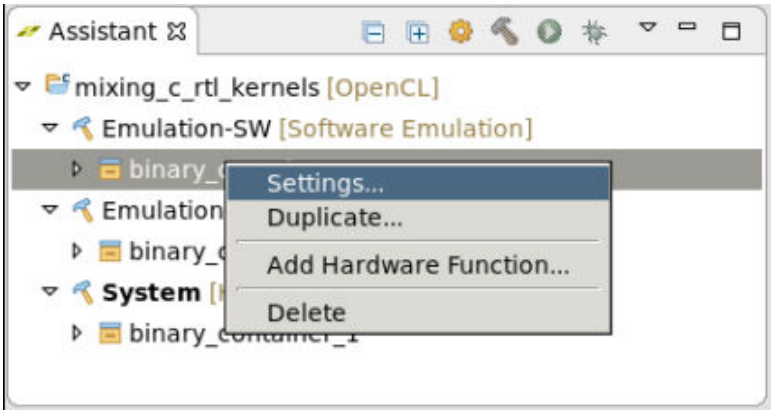



You can see that with a couple of clicks you can access many of the functions and features of the tool using the Assistant.

# XOCC Linking and Compilation Options

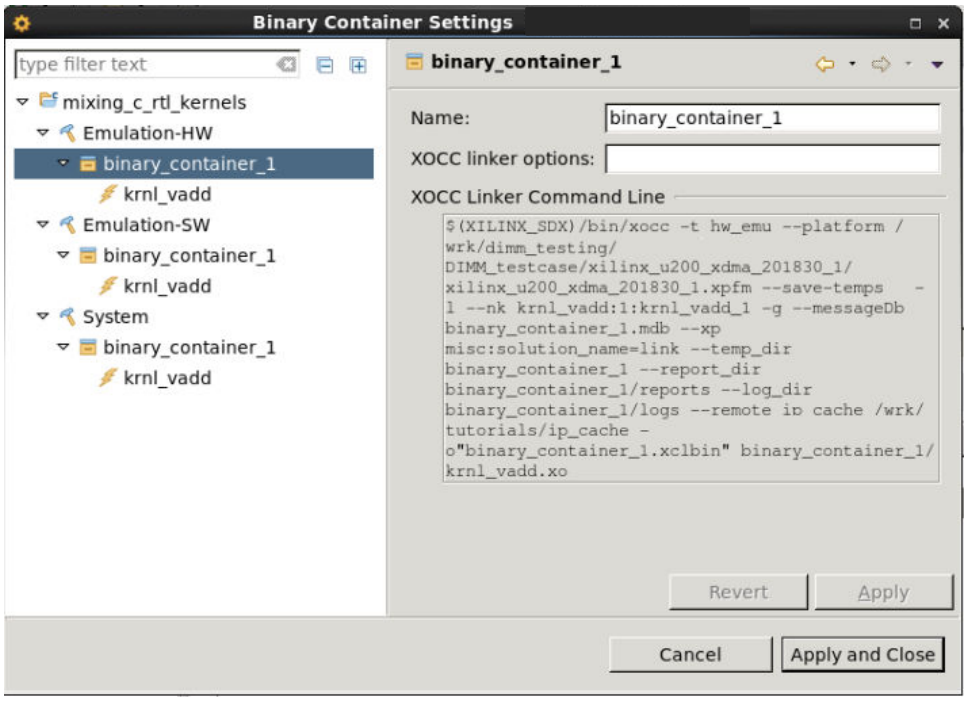
Using the Assistant window, you can update both the `xocc` compile and link options. First, right-click a `binary_container` folder in the Assistant window and select **Settings** as shown below.

*Figure 16:* **XOCC Settings from Assistant**




The Binary Container Settings window is displayed as shown below. To add linker `xocc` options, enter the options directly into the **XOCC linker options** field. Your added options are updated in the `makefile` (located under the **Project Explorer**) and applied during building. In addition, the updated options are also displayed in the **XOCC Linker Command Line** box for convenience.

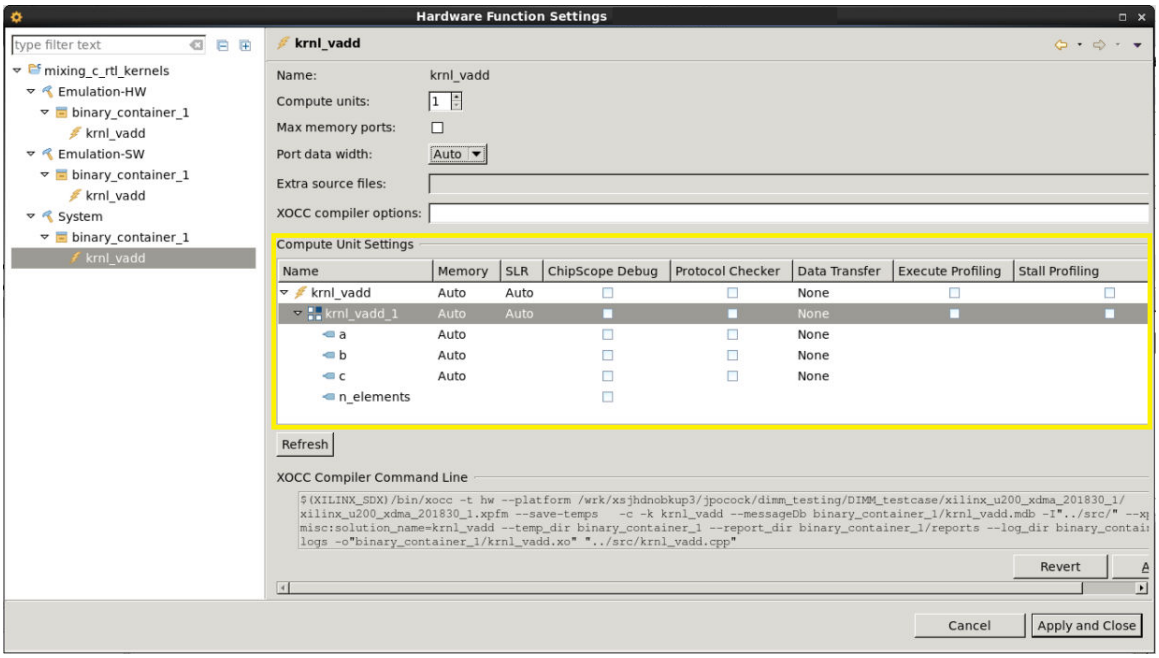
*Figure 17:* **XOCC Linker Options**




The `xocc` linker options differ slightly between the Emulation-HW, Emulation-SW, and System configurations. Specifically, the target (`-t` option) matches the respective configuration and debugging is not enabled in the System configuration.

To add `xocc` compile options to a particular kernel, click the desired kernel under the `binary_container` folder. For instance, in the image below, kernel `krnl_vadd` has been selected under the System configuration. Compile options can be entered directly into the **XOCC compile options** field. Your added options are only applied to that kernel and not shared across kernels. The **XOCC Compiler Command Line** displays the updated options.

*Figure 18:* **XOCC Compiler Options**




There are options to set the number of instances (Compute Units) of the kernel and for setting the port data width. When modifying these options, the associated `xocc` compile option is automatically generated and added to the `xocc` compiler command line for you.

Similar to the `xocc` linker options, the compiler options differ slightly between the Emulation-HW, Emulation-SW, and System configurations. Specifically, the target (`-t` option) matches the respective configuration and debugging is not enabled in the System configuration.

For Emulation-HW and System configurations, a Compute Unit Settings area is also displayed (outlined in yellow in the above image) which allows additional linker and compile options to be set. Since none of the additional options apply to Emulation-SW, the Compute Unit Settings area is not displayed for this configuration.

For Emulation-HW configuration, only Memory and SLR (see Super Logic Region for explanation of SLR) allocation options are displayed. However, for System configuration, additional protocol checker and profiling logic options are displayed. Depending on the options selected, this results in updates to both the `xocc` linker and compile options.

# SDx Project Export and Import

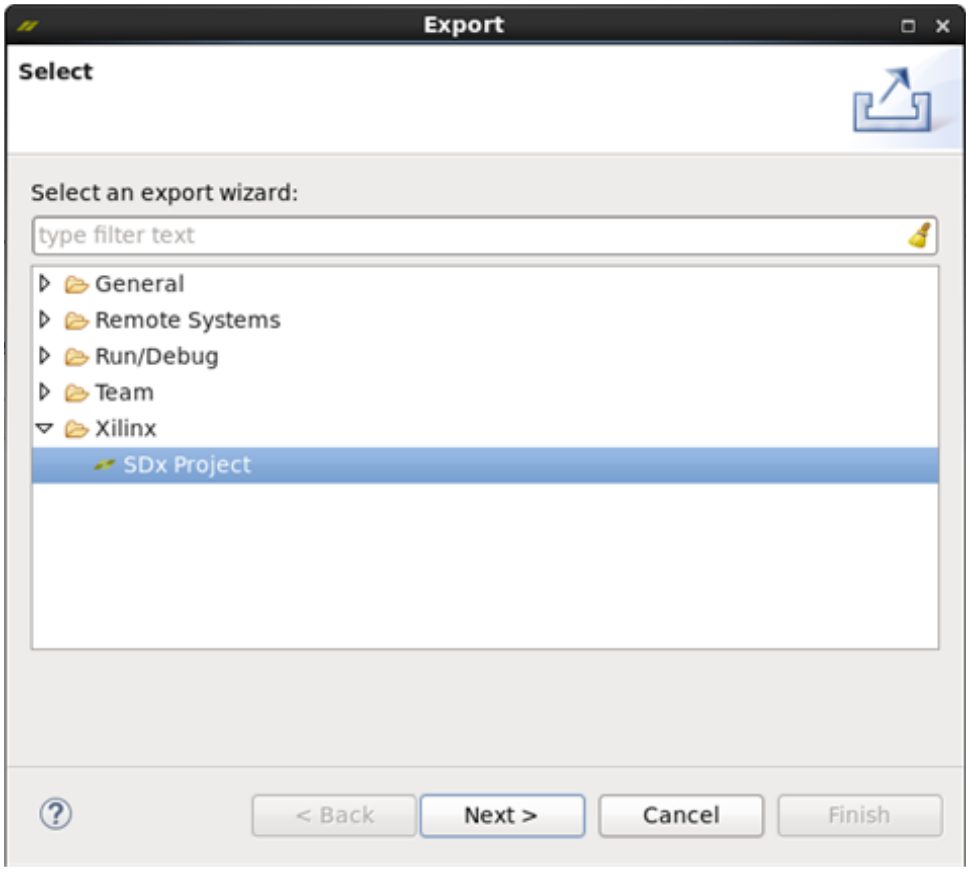
SDx provides a simplified method for exporting/importing one or more SDx projects within your workspace. You can optionally include associated project build folders.

## Exporting 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.

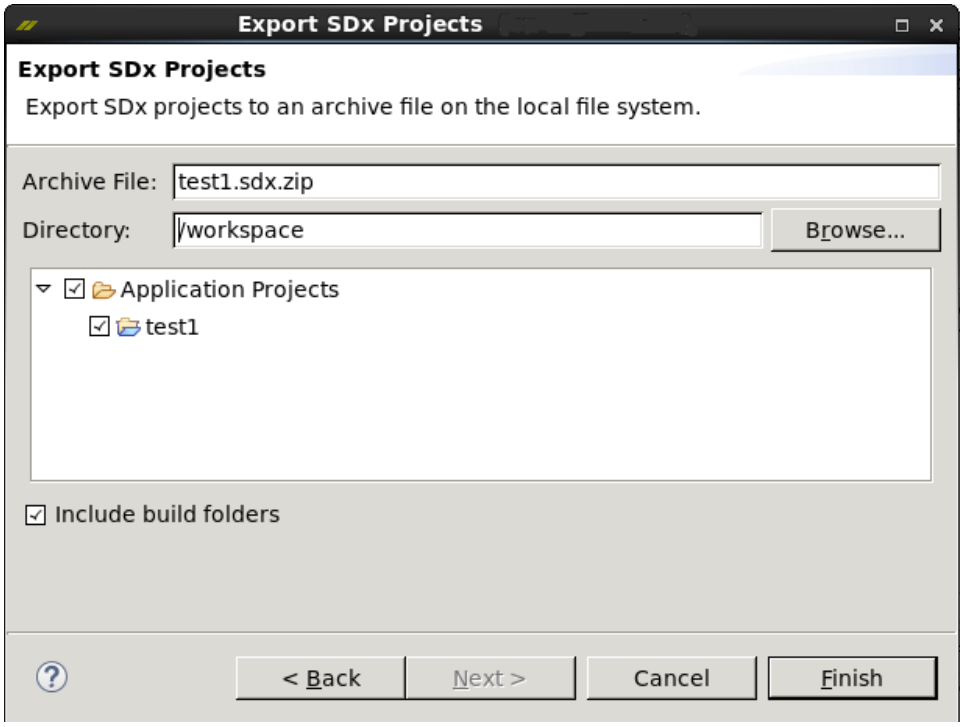
1. To export a project, select **File** → **Export** from the main menu.
2. When the Export wizard opens, select the Project to import under the folder, as shown in the following figure, and click **Next**.

*Figure 19:* **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 **Finish** to save the archive zip file.

*Figure 20:* **Export Filename Build Folder**

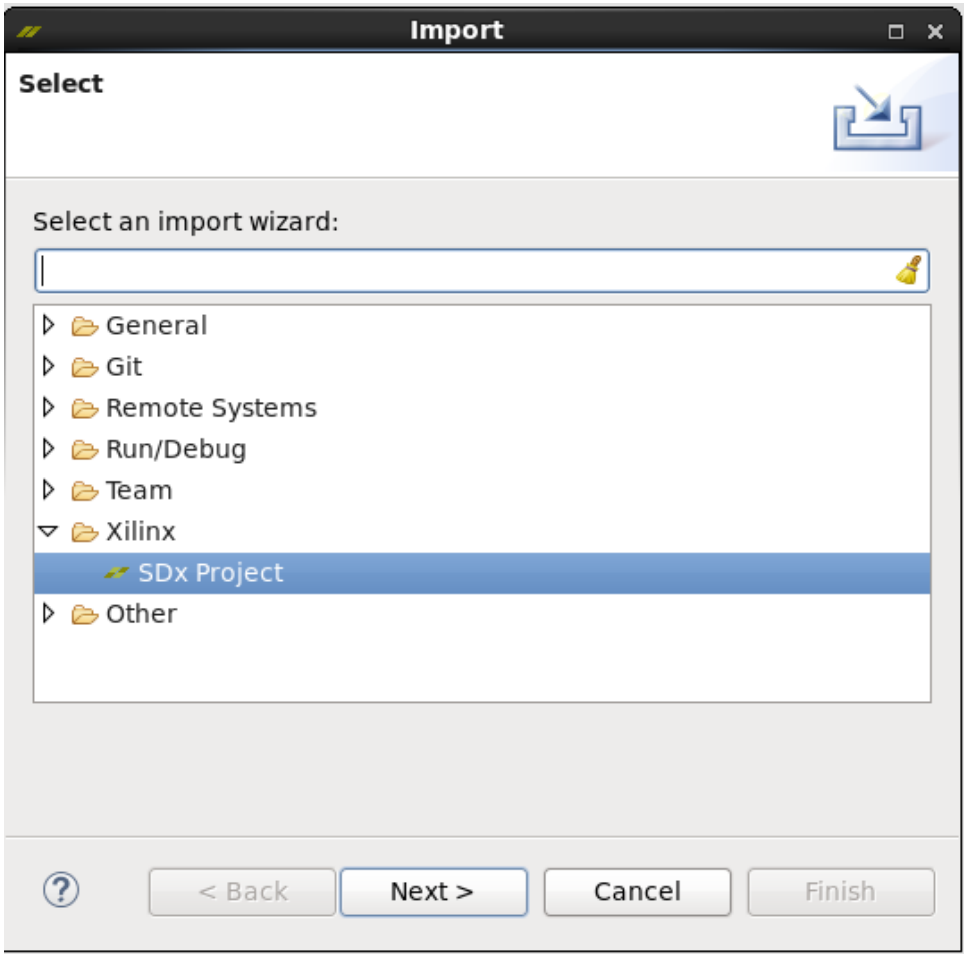



The SDx projects have been successfully archived and can be imported into a different workspace.

# Importing an SDx Project

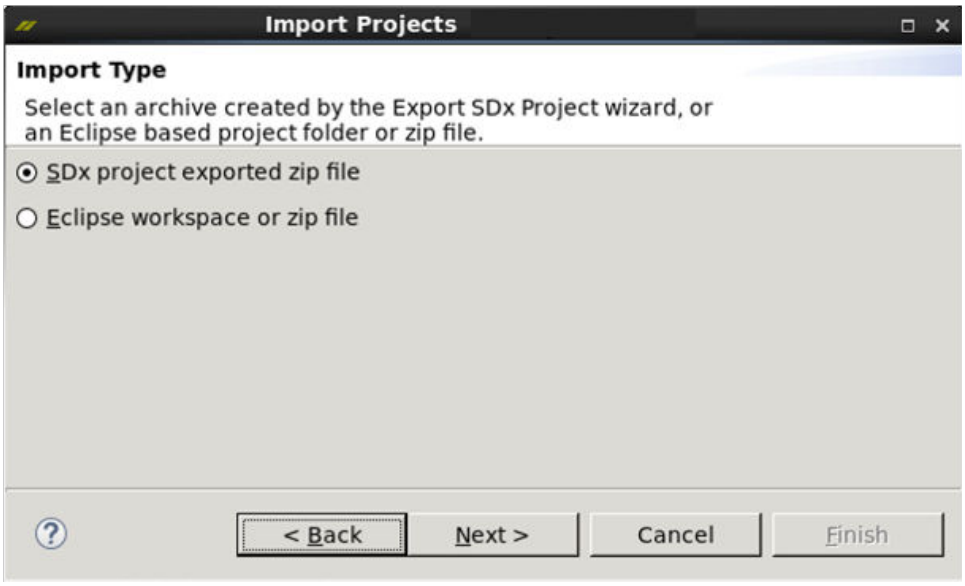
1. To import an SDx project, select **File** → **Import** from the top menu.
2. From the Import window, select the SDx Project import wizard under the Xilinx folder as shown in the following figure and click **Next**.

*Figure 21:* **Import Select an Import Wizard**



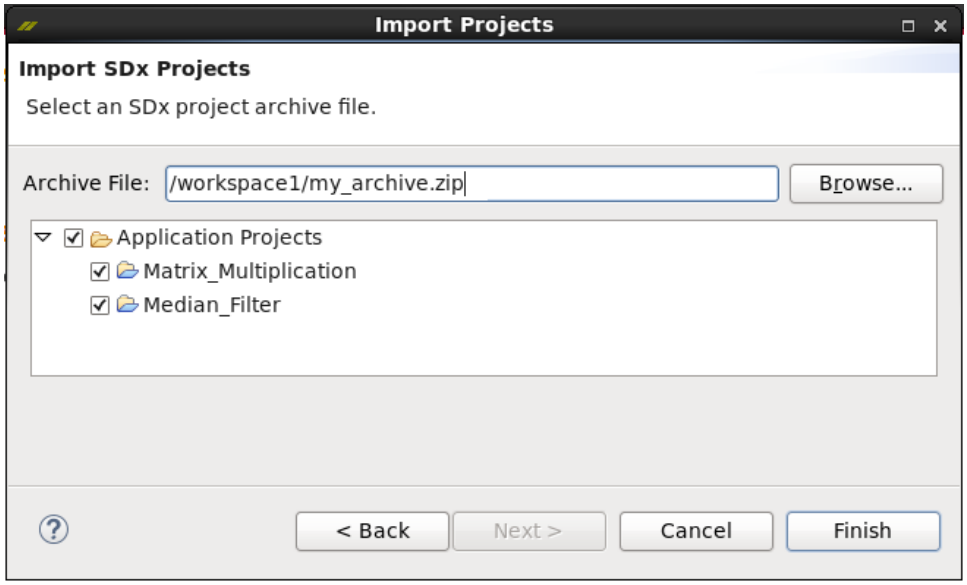

3. This opens the Import Projects window to select the import file type. Select the SDx project exported zip file shown in the following figure and click **Next**.

*Figure 22:* **Import Projects – Import Type**




4. This opens the Import SDx Projects window.
5. Browse and select the desired archive file. It will display the archived projects.
6. Select the projects to import using the check boxes and click **Next**. In the following figure both projects are selected for import.
7. Click **Finish** to import the projects into your workspace.

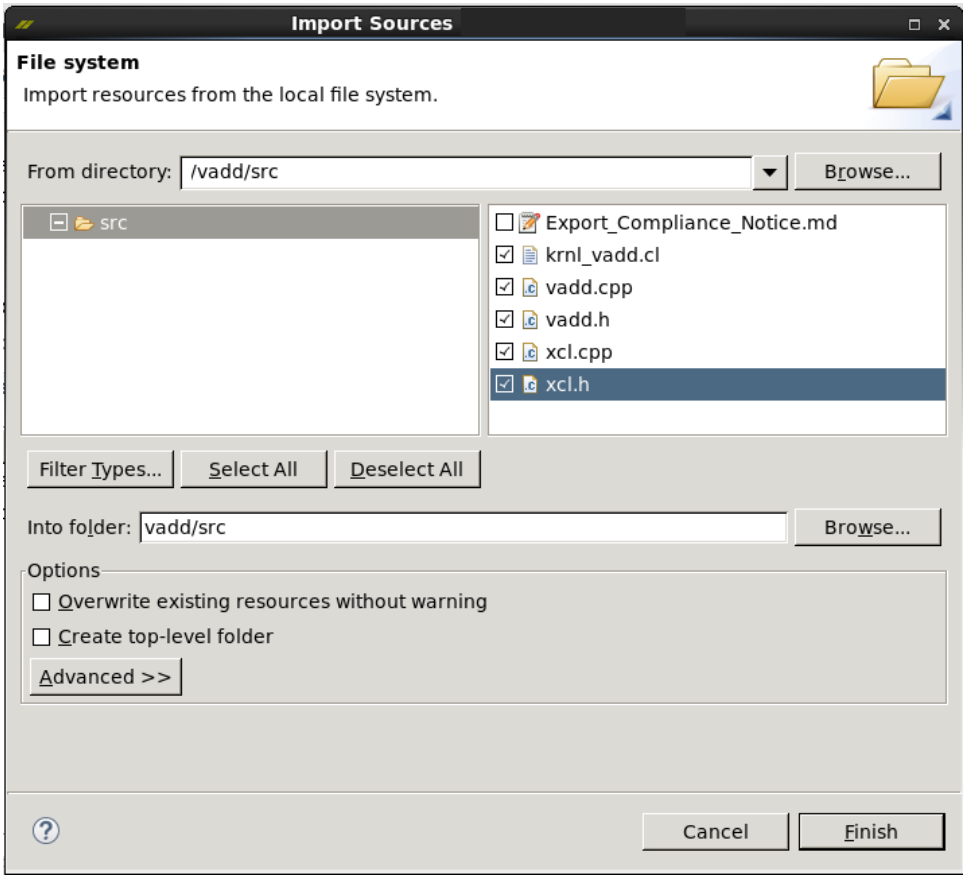
*Figure 23:* **Import Archive Filename**



# 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 left-clicking the `import sources icon` in the Project Explorer. This displays the Import Sources dialog box shown in the following image.

*Figure 24:* **Import Sources Dialog Box**

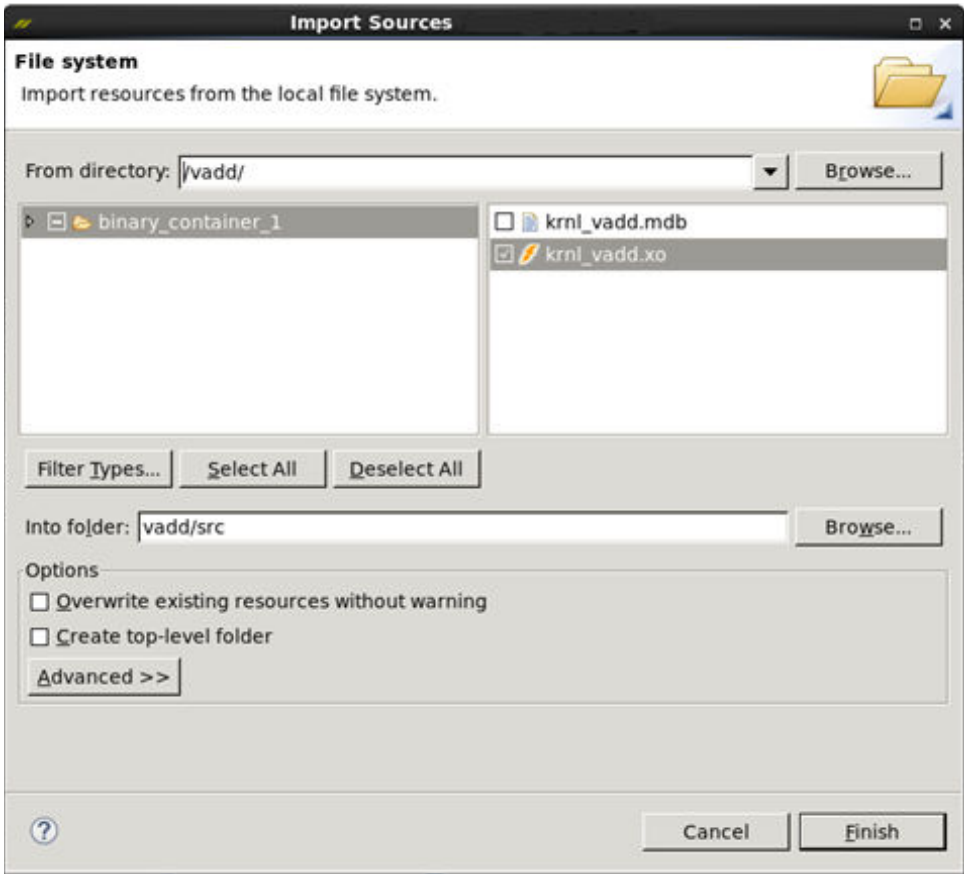



1. In the dialog box, click **Browse** to select the directory from which you want to import sources. Select the desired sources and click **Finish**.
2. The source files in that directory will be displayed. Select the desired sources to import by selecting the appropriate check boxes and click **Finish**. 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.*

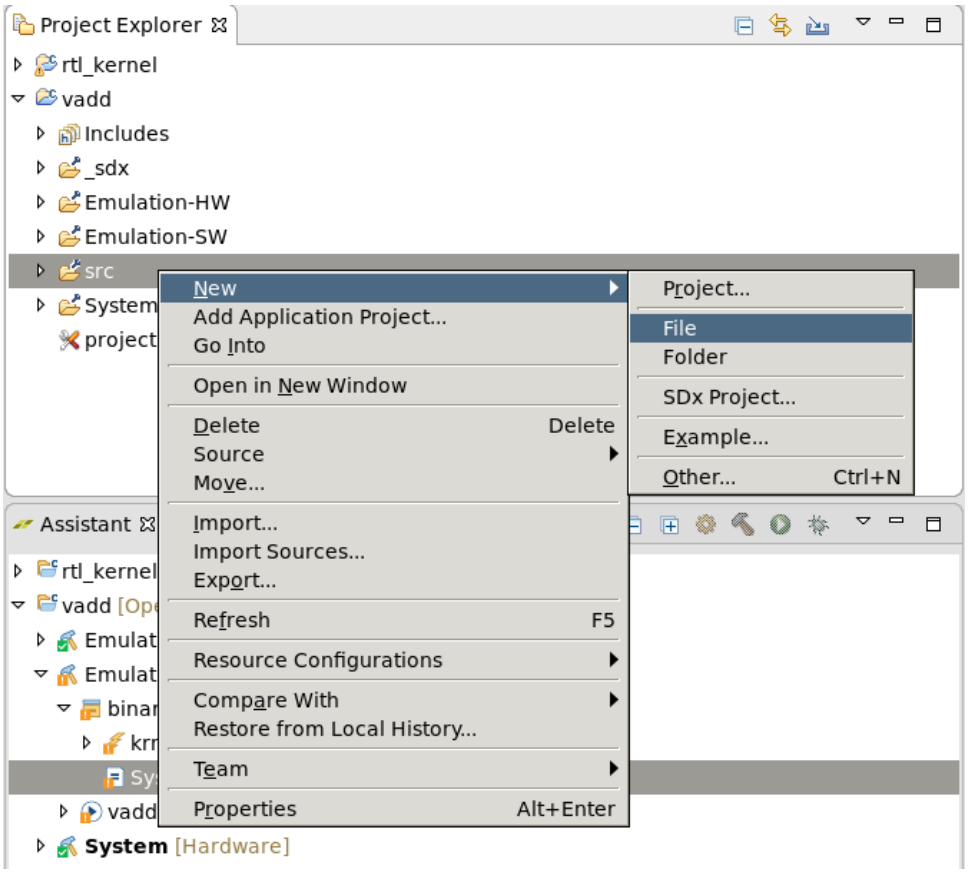
3. Similarly you can import compiled kernels (`.xo` files) into the project through the **Import Sources** selection. In the following image, the `krnl_vadd.xo` file will be imported into the project.

*Figure 25:* **Import .xo Sources**



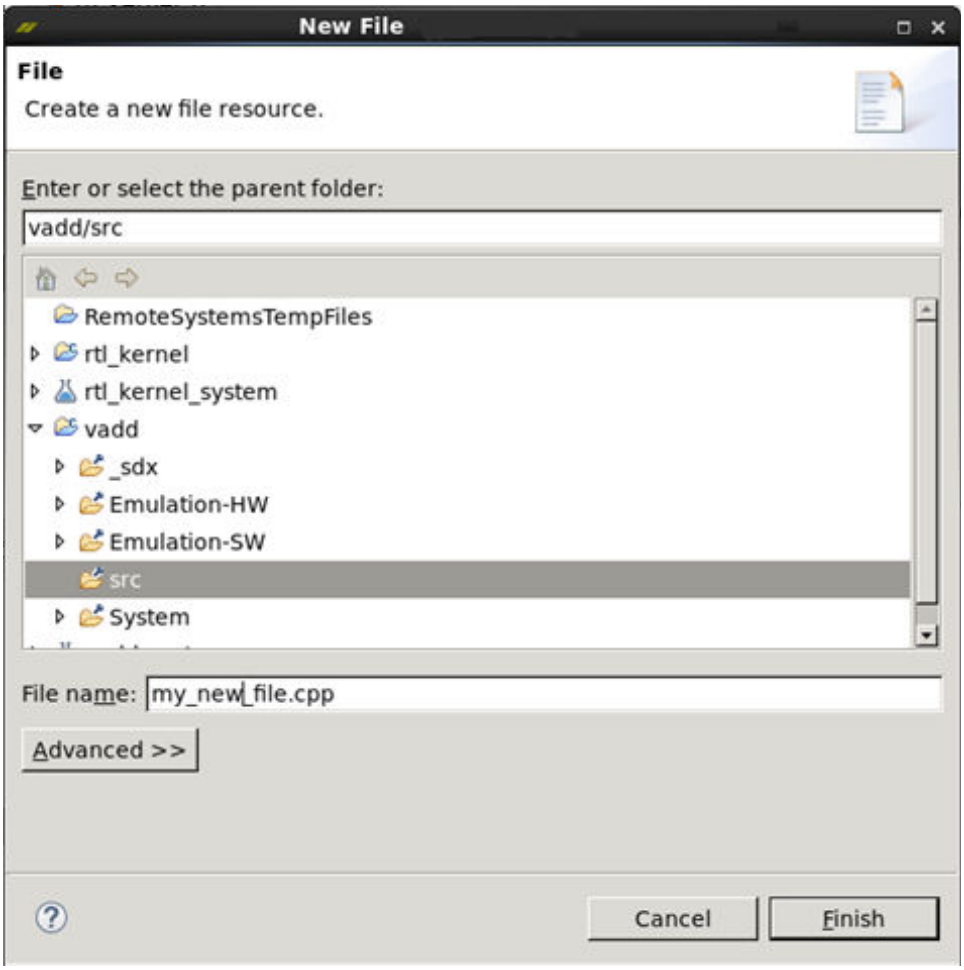

4. In addition to importing sources, you can also create and edit new source files in the GUI. With the project open in the SDx IDE, right-click the `src` folder and select **New** → **File** as shown in the following image.

*Figure 26:* **New File Select**




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

*Figure 27:* **New File Name**




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

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

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

The following topics discuss how to write code for the host application to setup the Xilinx Runtime (XRT), 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 set up 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 below.

**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 waiting to be executed to a device. You can setup the command queue to handle commands in the order submitted, or to be out-of-order so that a command can be executed as soon as possible. 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,
CL_QUEUE_PROFILING_ENABLE | CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE
, &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 `load_file_to_memory` function is used to load the file contents in the host machine memory space. The `clCreateProgramWithBinary` command downloads the FPGA binary (`.xclbin`) to the device and returns a `cl_program` handle. The following example shows the creation of the program using these API calls:

```
char *fpga_bin;
size_t fpga_bin_size;
fpga_bin_size = load_file_to_memory(binaryName, &fpga_bin);

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

# Transferring Data to/from the FPGA Device

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. You can customize the global memory bank connections by modifying the default connection. This improves kernel performance by enabling multiple kernels to concurrently read and write data from separate global memory banks. See* Mapping Kernel Interfaces to Memory Resources *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 set the arguments for 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 scheduled to run on the FPGA by using the `clEnqueueTask`. The request to execute 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` (and `clEnqueueMigrateMemObjects`) command is asynchronous in nature, and will return immediately after the command is enqueued in the command queue, you might need to manage the scheduling of events within the host application. 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 (SystemVerilog, 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.

All kernels require the following:

- 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):
  - AXI4 master interface to communicate with memory.
  - AXI4-Stream interface for transferring data between kernels.

In the case of OpenCL kernels, the AXI4-Lite interface is generated automatically while the AXI4-Lite memory map interfaces are generated based on the `__global` directive in the function definition. For C/C++ kernels, use interface pragmas to map to AXI4-Lite and AXI4 memory map interface. While for RTL kernels, you are responsible for adding these interfaces.

# 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

Kernels written in C/C++ are supported by the SDAccel environment. The above matrix multiplication kernel can be expressed in C/C++ code as shown below. 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.

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, each function argument should have a valid HLS interface pragma.

```
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)
```

## Pointer Arguments

All pointers are mapped to global memory. The data is accessed through AXI interfaces which can be mapped to different banks. The memory interface specification needs the following two pragmas:

1. The first is to define which argument the AXI memory map interface is accessed. An offset is always required. The `offset=slave` means that the offset of the array `<variable_name>` will be made available through the AXI slave interface of the kernel.

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

2. The second pragma for the AXI Slave interface. Scalars (and pointer offsets) are mapped to one AXI Slave control interface which must be named `control`.

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

***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_`.

Using current platforms (version 5.x 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_`.

### *Scalars*

Scalars are considered constant inputs and should also be mapped to `s_axilite`. The control interface specification is generated by the following command:

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

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.

---

**RECOMMENDED:** *When using `structs`, Xilinx recommends 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 Xilinx recommends 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.

## Writing RTL Kernels

RTL kernels have both software and hardware requirements for it to be used in the SDAccel environment framework. On the software side, the RTL kernel must operate and adhere to the register definitions described in Kernel Software Requirements.

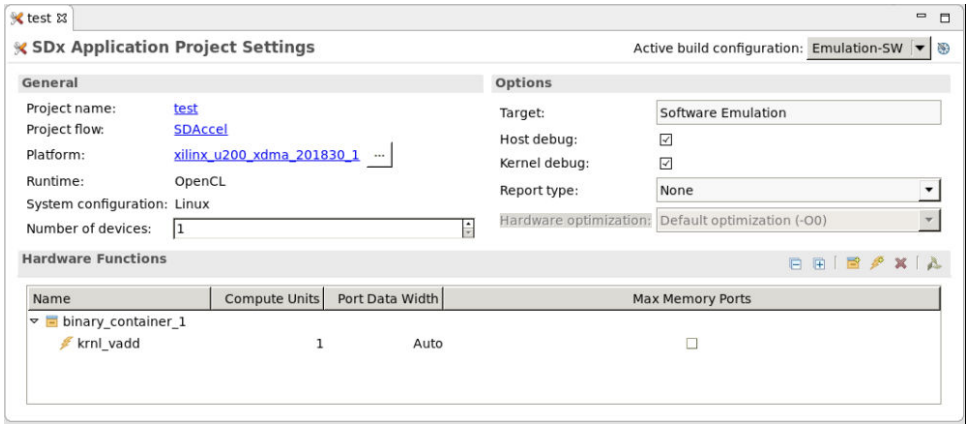
On the hardware side, it requires the interfaces outlined in the Kernel Interface Requirements.

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

# Building the System

Building the system requires building both the hardware (kernels) and the software (host code) side of the system. The Project Editor view, 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.

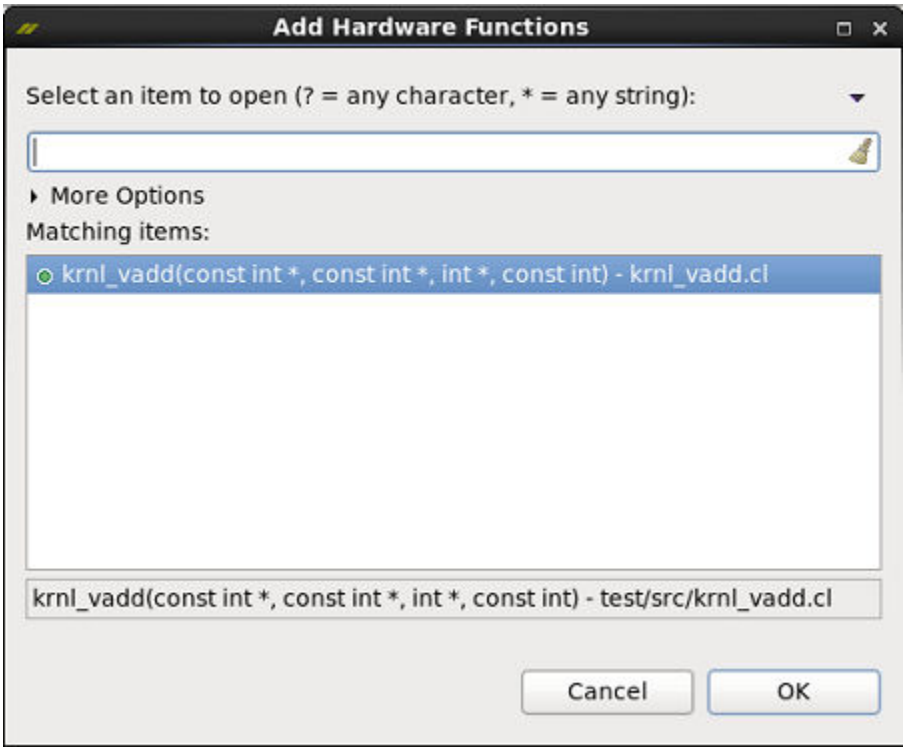
*Figure 28:* **Project Editor View**




The bottom portion of the Editor view 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 clicking 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 displays a list of kernels defined in the project. Select the kernel from the Add Hardware Functions dialog box as shown in the following figure.

*Figure 29:* **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 Creating Multiple Instances of a Kernel.

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
- 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.

# Building the Host Application

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.*

## Compiling the Host Application

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
```

## Linking the Host Application

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.

# Building the FPGA Binary

The kernel code is written in C, C++, OpenCL C, or RTL and is built by the `xocc` compiler; 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.

## Compiling the Kernels

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: RTL Kernels for more information.

## Build Target

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.
- For system build (`hw`), `xocc` generates the FPGA binary and the system can be run on hardware.

## Linking the Kernels

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

The `xocc` command to link files is:

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

## Creating Multiple Instances of a Kernel

During the linking stage, you can specify the number of instances of a kernel, referred to as a compute unit, through the `--nk xocc` switch. 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.

***Note*:** For additional information on the `--nk` options, see *SDAccel Environment Programmers Guide* (UG1277) and *SDx Command and Utility Reference Guide* (UG1279).

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

```
$ xocc -nk <kernel name>:<no of instances>:<name1>.<name2>…<nameN>
```

For example, the kernel `foo` is instantiated three times with compute unit names `fooA`, `fooB`, and `fooC`:

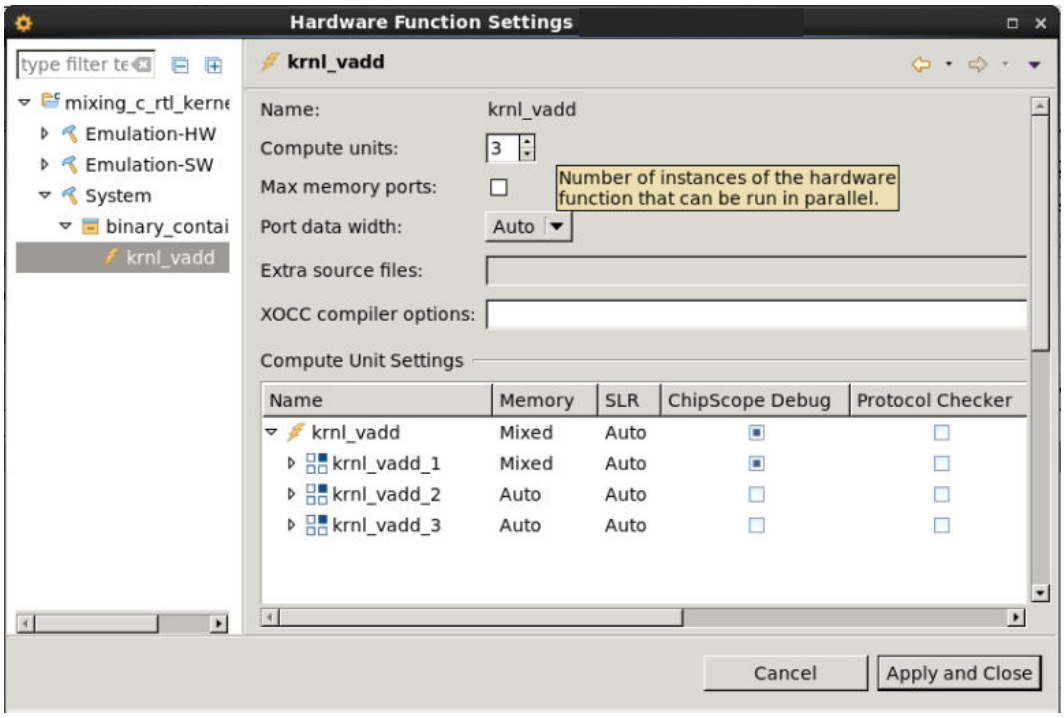
```
$ xocc --nk foo:3:fooA.fooB.fooC
```

**TIP:** *While the kernel instance name is optional, it is highly recommended to specify one as it 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**.

From within the Project Settings dialog box, select the desired kernel to instantiate and update the Compute units value. In the following figure, the kernel, `krnl_vadd`, will be instantiated three times (that is, three CUs).

*Figure 30:* **Instantiate Multiple Compute Units**




In the figure above, three compute units of the `krnl_vadd` kernel will be linked into the FPGA binary (`.xclbin`), addressable as `krnl_vadd_1`, `krnl_vadd_2`, and `krnl_vadd_3`.

# Mapping Kernel Interfaces to Memory Resources

The link phase is when the memory ports of the kernels are connected to memory resources which include PLRAM and DDR. By default, all kernel memory ports are connected to the same DDR bank. As a result, only one memory interface can transfer data to and from the DDR bank at one time, limiting overall performance. 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. For additional information, see *SDAccel Environment Programmers Guide* (UG1277) and *SDx Command and Utility Reference Guide* (UG1279).

Global memory is the DDR memory accessible by a platform. SDAccel platforms can have access to multiple global memory banks. In applications with multiple kernel instances running concurrently, this can result in significant performance gains. Even if there is only one compute unit in the device, by mapping its input and output ports to different banks you can improve overall performance by enabling simultaneous accesses to input and output data.

Specifying the desired kernel port to memory bank mapping requires taking the following steps:

1. In the host application, allocate buffers using a vendor extension pointer.
2. During `xocc` linking, use the `--sp` option to map the kernel interface to the desired 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 create buffers using a `cl_mem_ext_ptr_t` vendor extension pointer. The vendor extension pointer is used to indicate which kernel argument this buffer maps to. The runtime uses this information in conjunction with data in the FPGA binary to determine in which memory bank the buffer should be allocated.

During `xocc` linking, the `xocc --sp` option specifies the assignment of kernel ports to available memory resources, overriding the default assignments.

The directive to assign a compute unit's memory interface to a memory resource is:

```
--sp <COMPUTE_UNIT>.<MEM_INTERFACE>:<MEMORY>
```

Where

- `COMPUTE_UNIT` is the name of the compute unit (CU)
- `MEM_INTERFACE` is the name of one of the compute unit's memory interface or function argument
- `MEMORY` is the memory resource

It is necessary to have a separate directive for each memory interface connection.

**TIP:** *To obtain kernel information including kernel, port, and argument names use the command line tool `kernelinfo` if you have the `.xo` file or the `platforminfo` if you have the `.xclbin` file. For more information on the tool, see the* Appendix C: Useful Command Line Utilities.

For example, `xocc ... --sp vadd_1.m_axi_gmem:DDR[3]` assigns the memory interface called `m_axi_gmem` from a CU named `vadd` to DDR[3] memory.

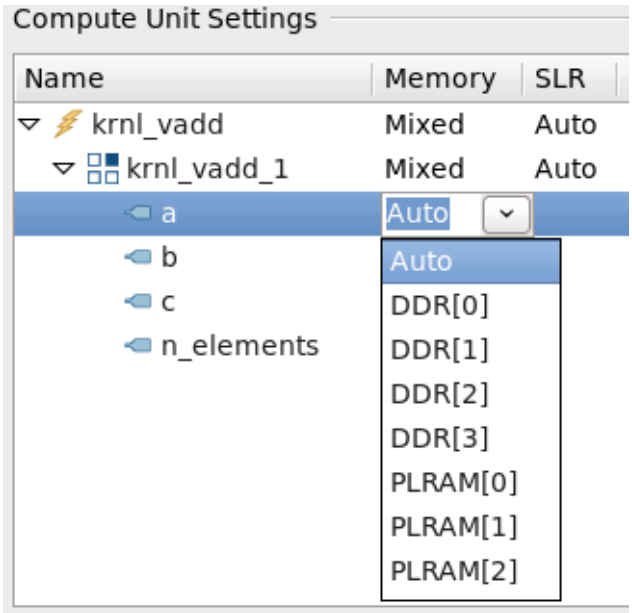
***Note:*** Starting in release 2018.3 and moving forward, memory resource are specified using vector formatting with the resource item enclosed in square brackets (that is, [..] ). For example, the DDR memory resource names of a device with four (4) DDR banks are specified at `DDR[0]`, `DDR[1]`, `DDR[2]`, and `DDR[3]`. PLRAM is specified in a similar fashion. While release 2018.3 supports the legacy `sptag` names (that is, *bank<n>*) for platforms available in 2018.2.xdf and any associated updates in 2018.3, this support will be deprecated in subsequent releases. All new platforms in 2018.3, however, do not support legacy `sptag` names and require the vector syntax format.

The `--sp` switch can be added through the SDx GUI similar to the process outlined in Creating Multiple Instances of a Kernel. 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.

To add directives to the `xocc` compilation through the GUI, from within the **Assistant**, right-click the desired kernel under **System** and select Settings.

This displays the hardware function settings dialog window where you can change the memory interface mapping under the Compute Unit Settings area. To change the memory resource mapping of a CU for a particular argument, click the **Memory** setting of the respective argument and change to the desired memory resource. The following figure shows the *a* argument being selected.

*Figure 31:* **Compute Unit Memory Setting**



To select the identical memory resource for all CU arguments, click the memory resource for the CU (that is, `krnl_vadd_1` in the example above) and select the desired memory resource.

**IMPORTANT!** *When using the `--sp` option to assign kernel interfaces to memory banks, you must specify the `--sp` option for all interfaces of the kernel. Refer to "Customization of DDR Bank to Kernel Connection" in the SDAccel Environment Programmers Guide (UG1277) for more information.*

## Allocating Compute Units to SLRs

A Compute Unit (CU) is allocated to a super logic region (SLR) during `xocc` linking using the `--slr` directive. The syntax of the command line directive is:

```
--slr <COMPUTE_UNIT>:<SLR_NUM>
```

where `COMPUTE_UNIT` is the name of the CU and `SLR_NUM` is the SLR number to which the CU is assigned.

For example, `xocc ... --slr vadd_1:SLR2` assigns the CU named `vadd_1` to SLR2.

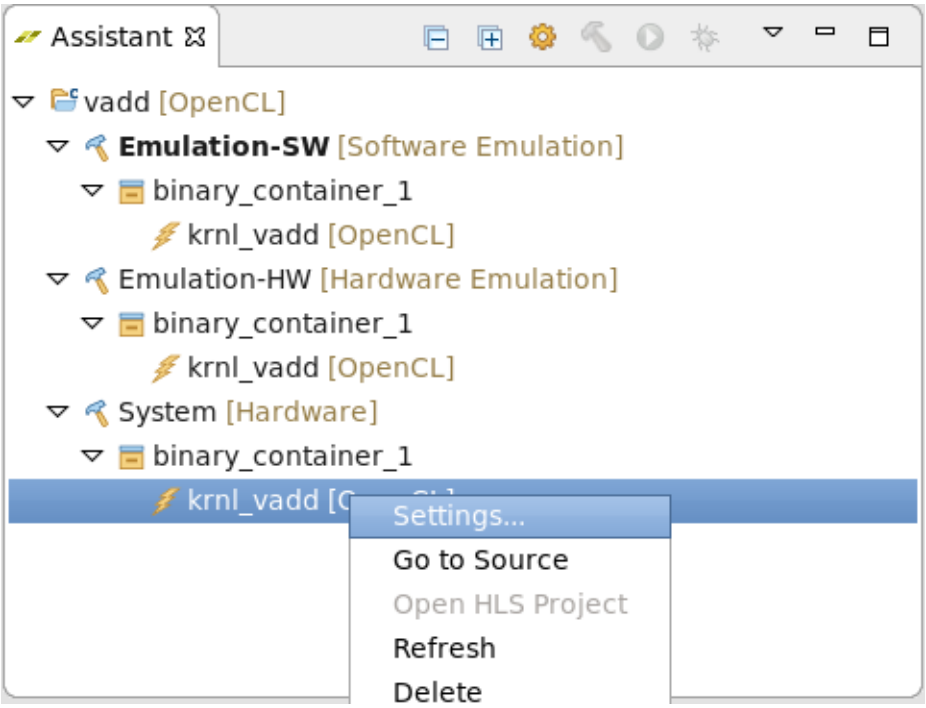
The `--slr` directive must be applied separately for each CU in the design. For instance, in the following example, three invocations of the `--slr` directive are used to assign all three CUs to SLRs; `krnl_vadd_1` and `krnl_vadd_2` are assigned to SLR1 while `krnl_vadd_3` is assigned to SLR2.

```
--slr krnl_vadd_1:SLR1 --slr krnl_vadd_2:SLR1 --slr krnl_vadd_3:SLR2
```

In the absence of an `--slr` directive for a CU, the tools are free to place the CU in any SLR.

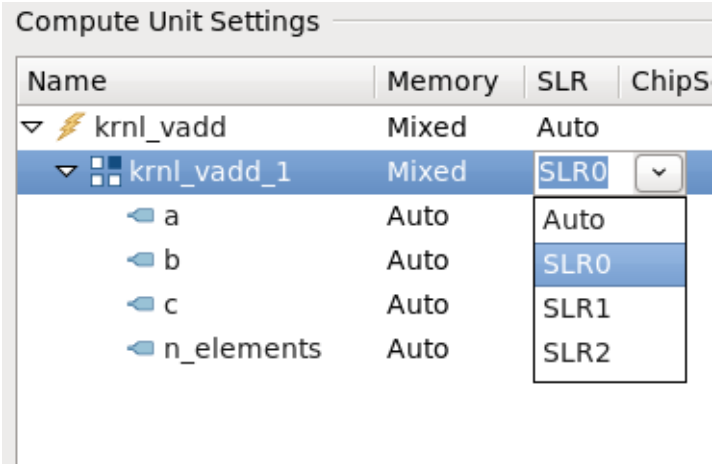
To allocate a CU to an SLR in the GUI flow, right-click the desired kernel under **System** or **Emulation-HW** configurations and select Settings as shown in the following figure.

*Figure 32:* **xocc Link Settings**




This displays the hardware function settings dialog window. Under the Compute Unit Settings area, you can change the SLR where the CU is allocated to by clicking the **SLR** setting of the respective CU and selecting the desired SLR from the menu as shown. Selecting **Auto** allows the tools the freedom to place the CU in any SLR.

*Figure 33:* **Compute Unit SLR Setting**



# Controlling Implementation 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 parameters and valid values can be found in the *SDx Command and Utility Reference Guide* (UG1279).

**TIP:** *Familiarity with the Vivado 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, parameters are specified as `param:<param_name>=<value>`, where:

- `param`: Required keyword.
- `param_name`: Name of a parameter to apply.
- `value`: Appropriate value for the parameter.

**IMPORTANT!** *The `xocc` linker does not check the validity of the parameter or value. Be careful to apply valid values or the downstream tools might 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 as shown below:

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

You can specify `param` values in an `xocc.ini` file with each option specified on a separate line (without the `--xp` switch).

An `xocc.ini` is an initialization file that contains `--xp` settings. Locate the file in the same directory as the build configuration.

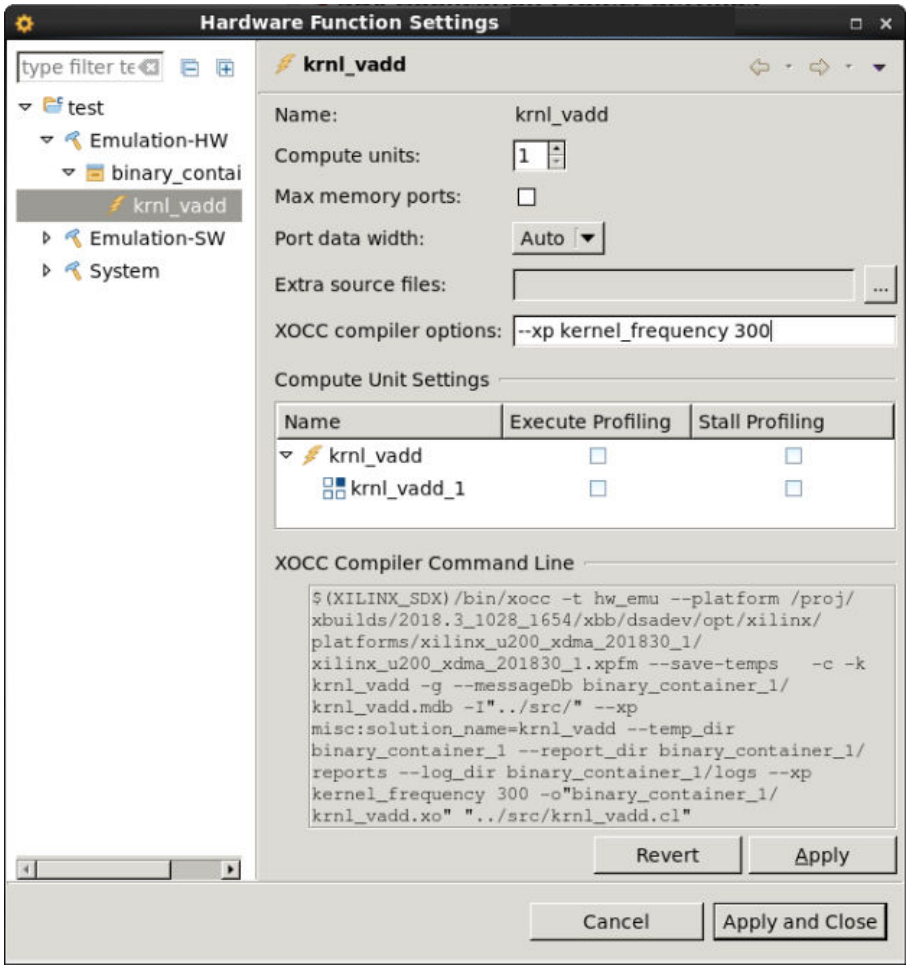
```
param:compiler.enableDSAIntegrityCheck=true
param:prop:kernel.foo.kernel_flags="-std=c++0x"
```

Under the GUI flow, if no `xocc.ini` is present, the application uses 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 similar to that outlined in Creating Multiple Instances of a Kernel. 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 demonstrates the `--xp` setting for the `krnl_vadd` kernel.

*Figure 34:* **Assistant XOCC Compile Settings**



## Controlling 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 `<report_level>` is one of the following `report_level` options:

- `-R0`: Minimal reports and no intermediate design checkpoints (DCP)
- `-R1`: Includes R0 reports plus:
  - Identifies design characteristics to review for each kernel (`report_failfast`)
  - Identifies design characteristics to review for full design post-opt (`report_failfast`)
  - Saves post-opt DCP
- `-R2` : Includes R1 reports plus:
  - The Vivado default reporting including DCP after each implementation step
  - Design characteristics to review for each SLR after placement (`report_failfast`)

**TIP:** *The `report_failfast` is a utility that 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 described in Creating Multiple Instances of a Kernel:

- 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 software 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. Software 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. 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. Xilinx recommends explicitly defining optimal connections. See* Kernel SLR and DDR Memory Assignments *for details.*

## Specifying a Target

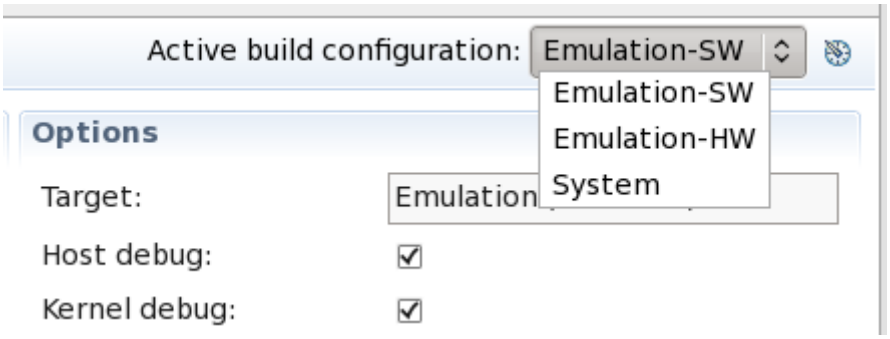
You can specify the target build from the command-line with the following command:

```
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 (see the following figure):

- Emulation-SW
- Emulation-HW
- System

*Figure 35:* **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

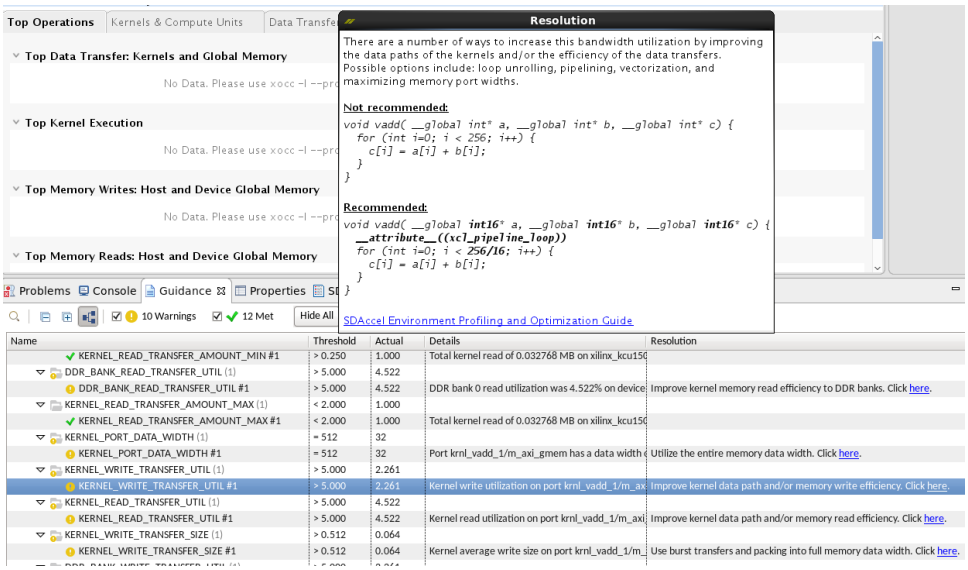
The SDAccel environment has a comprehensive design guidance tool that provides immediate actionable guidance to the software application developers for detected issues in their designs. Guidance is generated from HLS, the SDx Profiler and the Vivado® Design Suite when invoked from `xocc`. The generated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

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 highlights 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 bandwidth.

*Figure 36:* **Design Guidance Example**




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

There is 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 `<output>` is the `.xo` name:

- `xocc_compile_<output>_guidance.html` for `xocc` compilation
- `xocc_link_t_guidance.html` for `xocc` linking

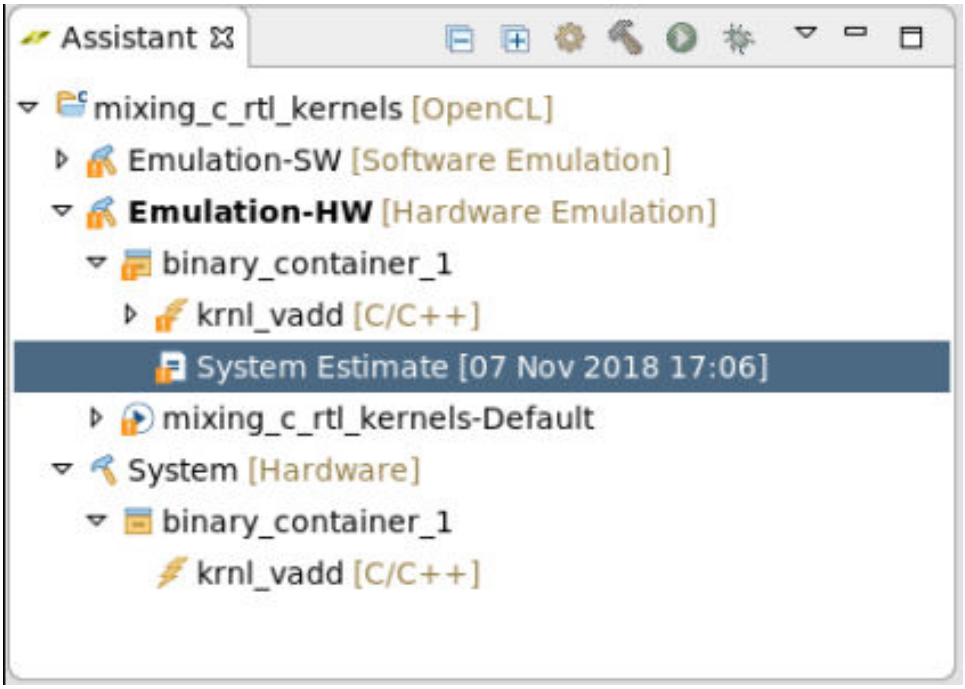
The profile design guidance helps you interpret the profiling results and know exactly where to focus on to improve performance. Specific details of the reports and additional design guidance details can be found in *SDAccel Environment Profiling and Optimization Guide* (UG1207).

# System Estimate Report

The SDAccel HLS generates the System Estimate 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 shown below.

**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. Xilinx® recommends iterating in Hardware Emulation and optimizing before performing a System build.*

*Figure 37:* **System Estimate Assistant View**

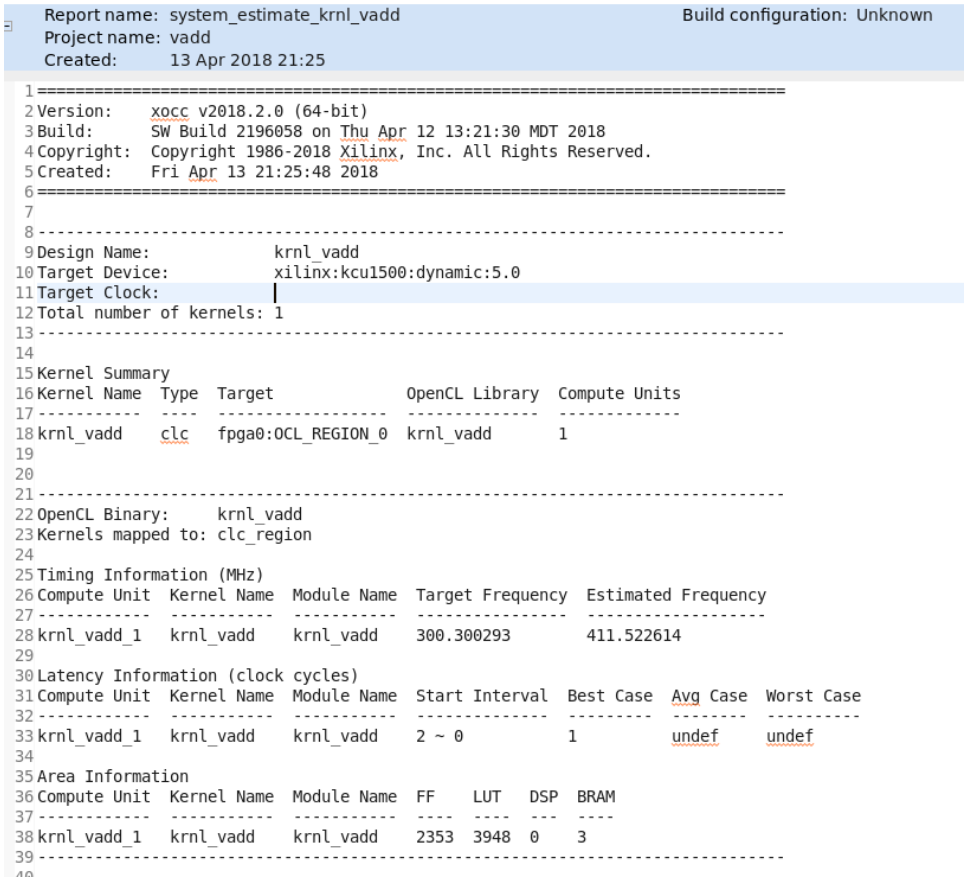



The report contains high-level details of the user kernels including resource usage and estimated frequency. The results can be used to guide the design optimization. For instance, if the target frequency is not met, it might be necessary to revisit the source code.

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 three BRAMs.

*Figure 38:* **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
```

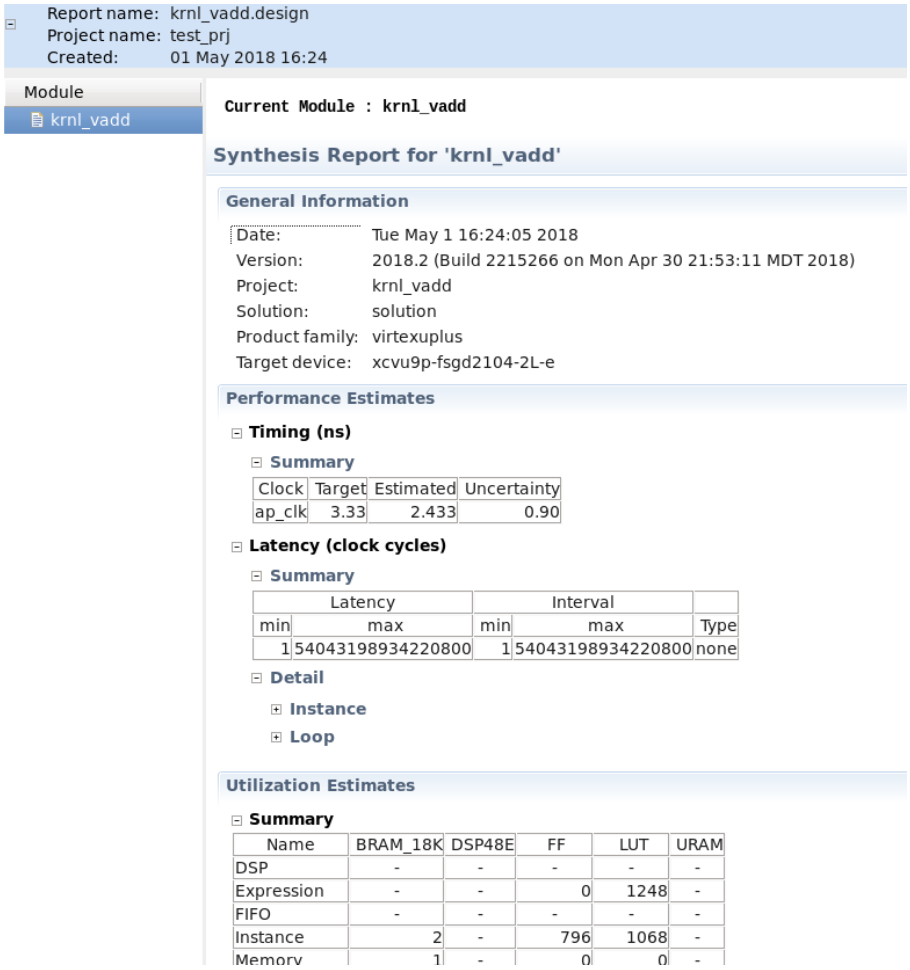
For additional details on the System Estimate report see the *SDAccel Environment Profiling and Optimization Guide* (UG1207).

# HLS Report

The HLS Report provides details about the high-level synthesis (HLS) process of a user kernel and is generated in Hardware emulation and System builds. This process translates the C/C++ and OpenCL 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 39:* **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).

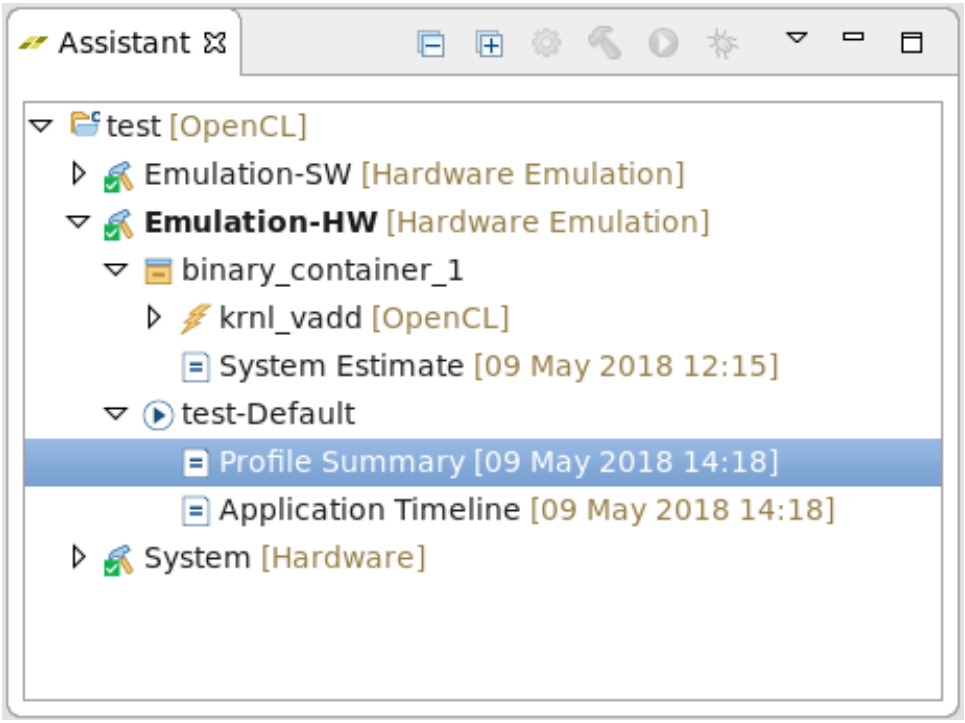
# Profile Summary Report

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.*

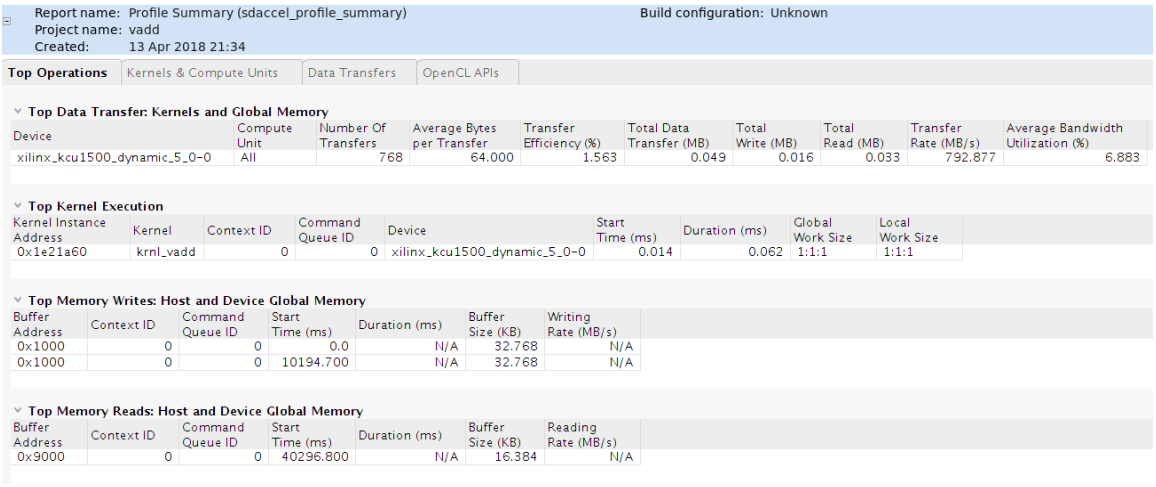
To open the Profile Summary report in the SDx IDE, double-click the Profile Summary report under the Assistant as shown in the following image.

*Figure 40:* **Opening Profile Summary Report**




An example of the Profile Summary report is shown here.

*Figure 41:* **Profile Summary**




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

*Table 1:* **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 through 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, the profile summary data is generated by using the `--profile_kernel` option during the linking stage. The `--profile_kernel` syntax is given below:

```
--profile_kernel <[data]:<[kernel_name|all]:[compute_unit_name|all]:
[interface_name|all]:[counters|all]>
```

See the *SDAccel Environment Profiling and Optimization Guide* (UG1207) for complete details.

# 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.

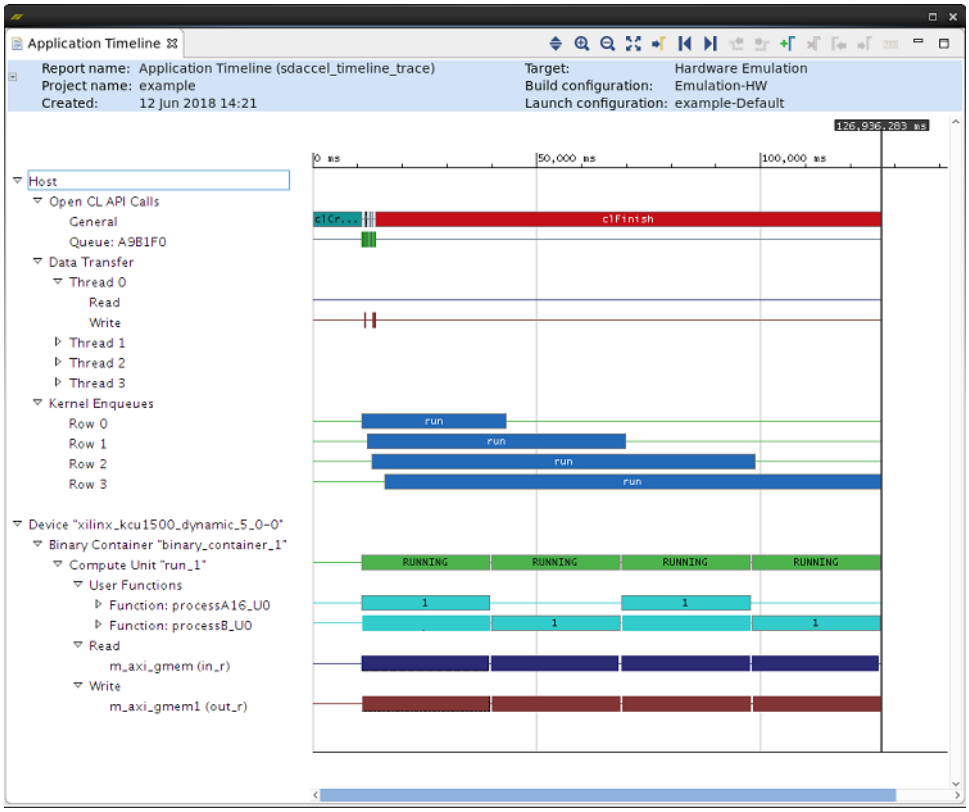
This graphical representation enables the programmer to identify issues regarding kernel synchronization and efficient concurrent execution.

**TIP:** *By default, timeline and device trace data are only collected during hardware emulation and not System build. Turning on device profiling for System build 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.

The following is a snapshot of the Application Timeline window which displays host and device events on a common timeline. Host activity is displayed at the top of the image and kernel activity is shown on the bottom of the image. Host activities include creating the program, running the kernel and data transfers between global memory and the host. The kernel activities include read/write accesses and transfers between global memory and the kernel(s). This information helps you understand details of application execution and identify potential areas for improvements.

*Figure 42:* **Application Timeline**




Timeline data can be enabled and collected through the command line flow, however, viewing must be done through the GUI. Complete instructions for enabling and displaying timeline data collection through both the command and GUI flows are given in *SDAccel Environment Profiling and Optimization Guide* (UG1207).

# Waveform View and Live Waveform Viewer

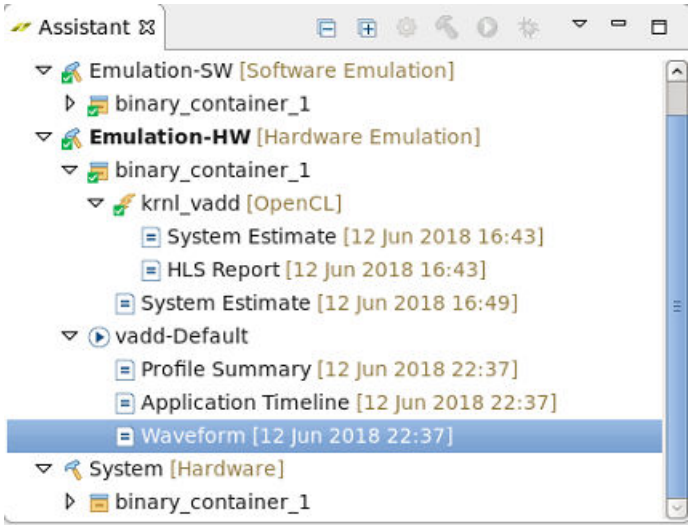
The SDx Development Environment can generate a Waveform View when running hardware emulation. It displays in-depth details on the emulation results at the system level, compute unit (CU) level, and at the function level. The details include data transfers between the kernel and global memory and data flow through 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 runtime 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.

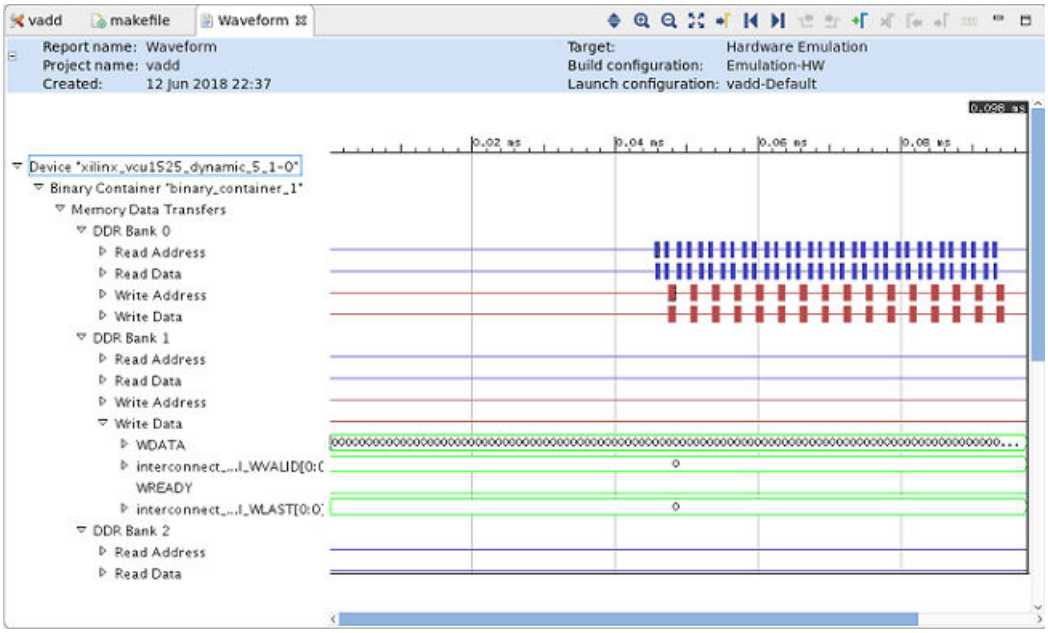
Double-click the **Waveform** in the Assistant view (shown in the following image) to open the Waveform View window.

*Figure 43:* **Opening Waveform View**




An example of the Waveform View is shown here.

*Figure 44:* **Waveform View Example**

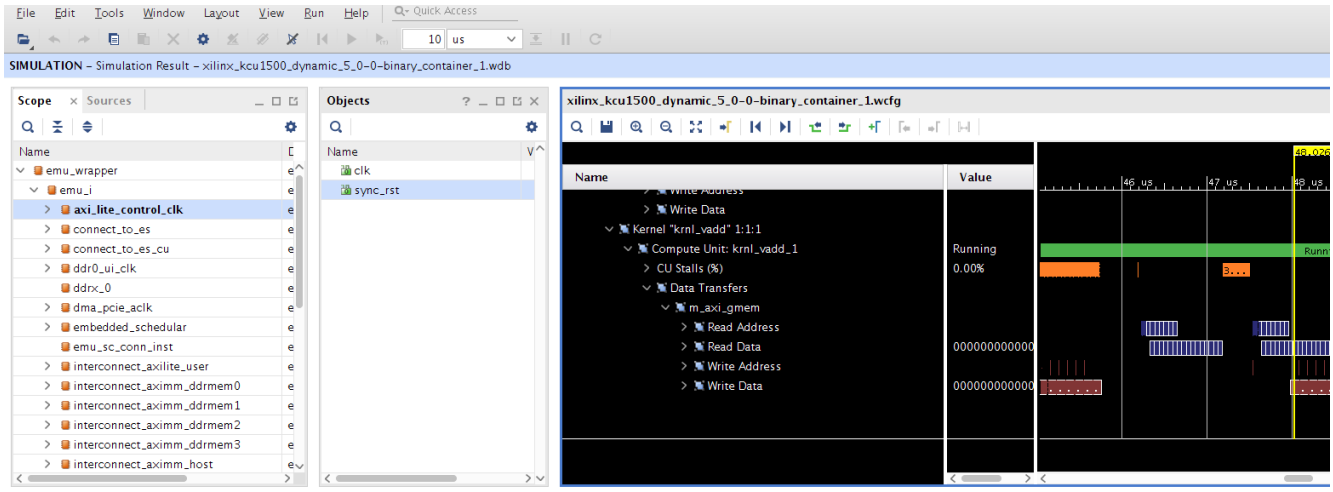



The Live Waveform Viewer can be viewed if you select **Launch Live Waveform** in the Run Configuration Main tab. Or, if the Launch Live Waveform is not selected, you can open the waveform (`.wdb`) with `xsim` through the Linux command line. The `.wdb` file is located in the sub-directory, `Emulation-HW/<kernel_name>-Default`, within the project directory. Use the following Linux line command to open `xsim`:

```
xsim -gui <filename.wdb> &
```

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

*Figure 45:* **Live Waveform Viewer Example**



# Kernel SLR and DDR Memory Assignments

Kernel compute unit (CU) instance and DDR memory resource floorplanning are keys to meeting quality of results of your design in terms of frequency and resources. Floorplanning involves explicitly allocating CUs (a kernel instance) to SLRs and mapping CUs to DDR memory resources. When floorplanning, both CU resource usage and DDR memory bandwidth requirements need to be considered.

The largest Xilinx FPGAs are made up of multiple stacked silicon dies. Each stack is referred to as a super logic region (SLR) and has a fixed amount of resources and memory including DDR interfaces. Available device SLR resources which can be used for custom logic can be found in *SDAccel Environments Release Notes, Installation, and Licensing Guide* (UG1238) or can be displayed using the `platforminfo` utility described in Platforminfo Utility.

You can use the actual kernel resource utilization values to help distribute CUs across SLRs to reduce congestion in any one SLR. The system estimate report lists the number of resources (LUTs, Flip-Flops, BRAMs, etc.) used by the kernels early in the design cycle. The report can be generated during hardware emulation and system compilation through the command line or GUI and is described in System Estimate Report.

Use this information along with the available SLR resources to help assign CUs to SLRs such that no one SLR is over-utilized. The less congestion in an SLR, the better the tools can map the design to the FPGA resources and meet your performance target. For mapping memory resources and CUs, see Mapping Kernel Interfaces to Memory Resources and Allocating Compute Units to SLRs, respectively.

***Note*:** While compute units can be connected to any available DDR memory resource, it is also necessary to account for the bandwidth requirements of the kernels when assigning to SLRs. *SDAccel Environment Profiling and Optimization Guide* (UG1207) provides details on allocating and optimizing DDR bandwidth.

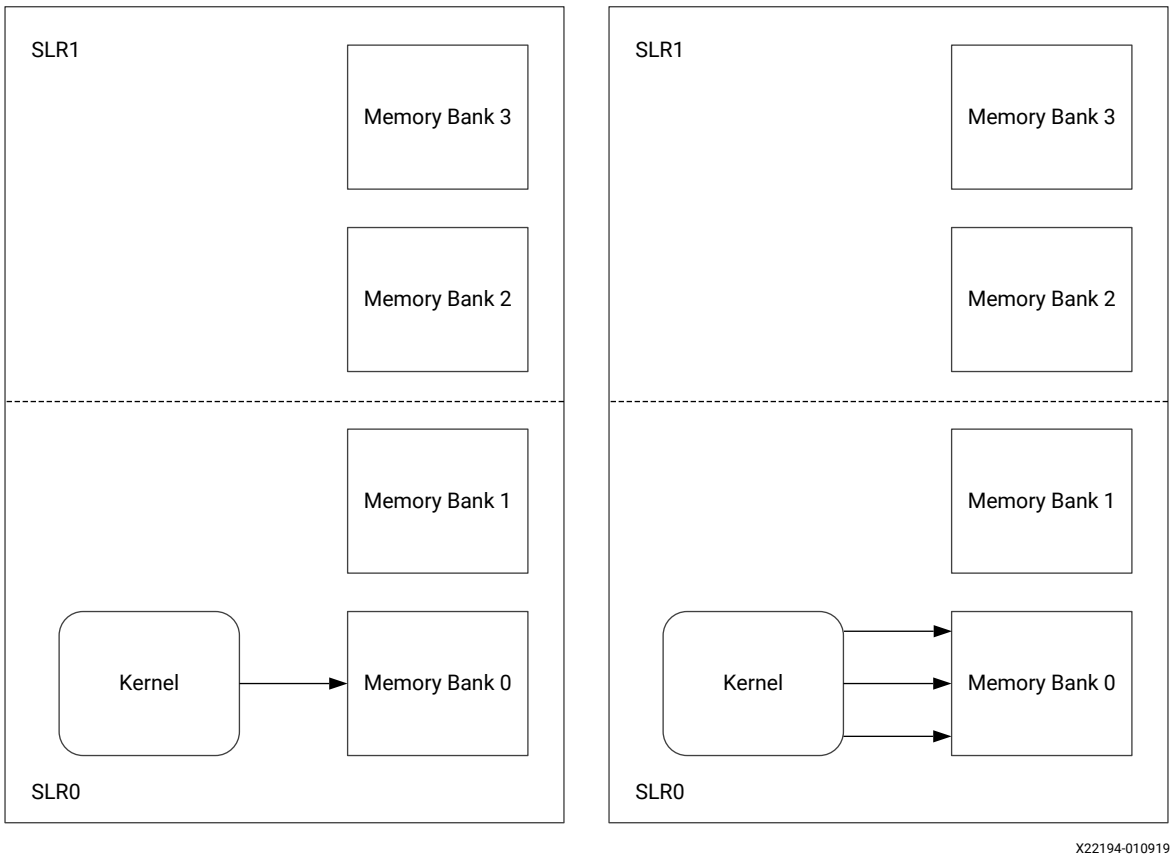
After allocating your CUs to SLRs, map any CU master AXI port(s) to DDR memory resources. Xilinx recommends connecting to a DDR memory resource in the same SLR as the CU. This reduces competition for the limited SLR-crossing connection resources. In addition, connections between SLRs use super long line (SLL) routing resources, which incurs a greater delay than a standard intra-SLR routing.

It might be necessary to cross an SLR region to connect to a DDR resource in a different SLR. However, if both the `--sp` and the `--slr` directives are explicitly defined, the tools automatically add additional crossing logic to minimize the effect of the SLL delay, and facilitates better timing closure.

# Guidelines for Kernels that Access Multiple Memory Banks

The DDR memory resources are distributed across the super logic regions (SLRs) of the platform. Since the number of connections available for crossing between SLRs is limited, the general guidance is to place a kernel in the same SLR as the DDR memory resource with which it has the most connections. This reduces competition for SLR-crossing connections and avoids consuming extra logic resources associated with SLR crossing.

*Figure 46:* **Kernel and Memory in Same SLR**




***Note*:** The image on the left shows a single AXI interface mapped to a single memory bank. The image on the right shows multiple AXI interfaces mapped to the same memory bank.

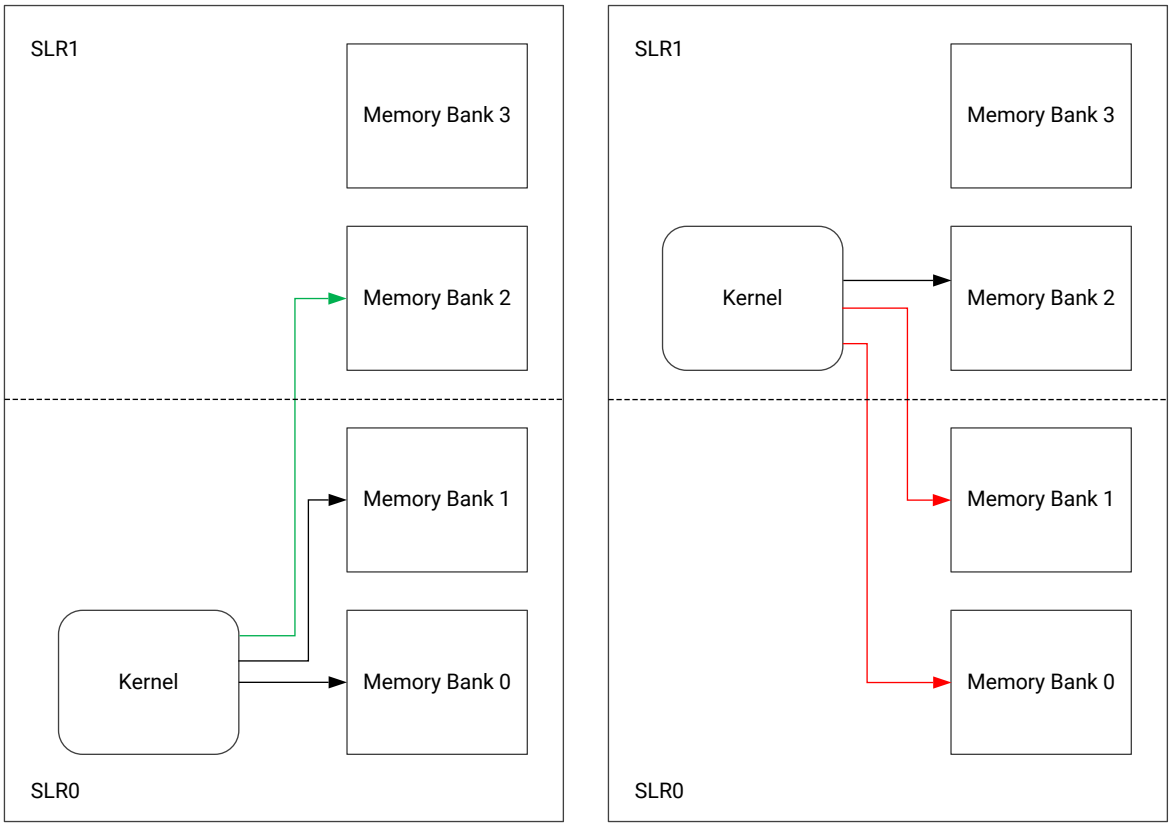
As shown in the previous figure, when a kernel has a single AXI interface that maps only a single memory bank, the `platforminfo` utility described in Platforminfo Utility lists the SLR that is associated with the memory bank of the kernel; therefore, the SLR where the kernel would be best placed. In this scenario, the design tools might automatically place the kernel in that SLR without need for extra input; however, you might need to provide an explicit SLR assignment for some of the kernels under the following conditions:

- If the design contains a large number of kernels accessing the same memory bank.

- A kernel requires some specialized logic resources that are not available in the SLR of the memory bank.

When a kernel has multiple AXI interfaces and all of the interfaces of the kernel access the same memory bank, it can be treated in a very similar way to the kernel with a single AXI interface, and the kernel should reside in the same SLR as the memory bank that its AXI interfaces are mapping.

*Figure 47:* **Memory Bank in Adjoining SLR**

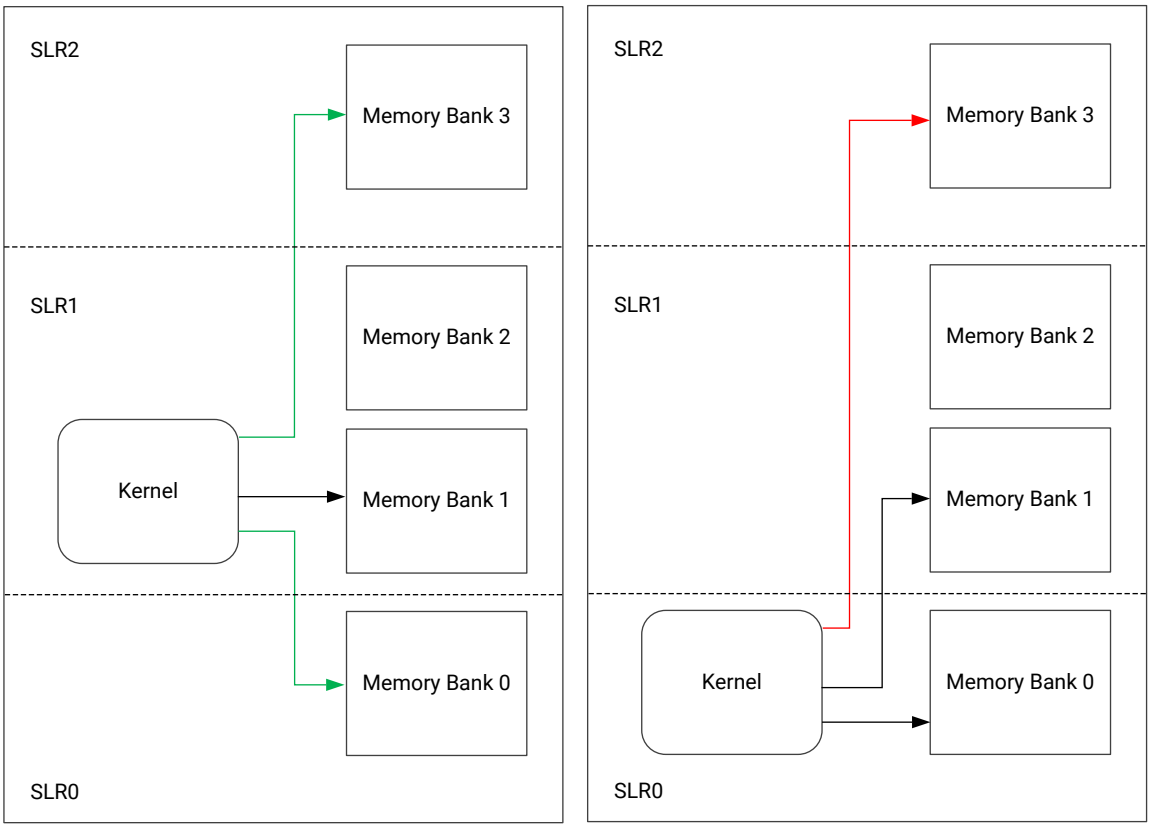



***Note*:** The image on the left shows one SLR crossing is required when the kernel is placed in SLR0. The image on the right shows two SLR crossings are required for kernel to access memory banks.

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

When the kernel is mapping memory banks from different SLRs, explicitly specify the SLR assignment as described in Kernel SLR and DDR Memory Assignments.

*Figure 48:* **Memory Banks Two SLRs Away**




***Note*:** The image on the left shows two SLR crossings are required to access all of the mapped memory banks. The image on the right shows three SLR crossings are required to access all of the mapped memory banks.

As shown in the previous figure, when a platform contains more than two SLRs, it is possible that the kernel might map a memory bank that is not in the immediately adjacent SLR to its most commonly mapped memory bank. When this scenario arises, memory accesses to the distant memory bank must cross more than one SLR boundary and incur additional SLR-crossing resource costs. To avoid such costs it might be better to place the kernel in an intermediate SLR where it only requires less expensive crossings into the adjacent SLRs.

XILINX

Chapter 7

# Debugging Applications and Kernels

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

In addition, for hardware-centric debugging, designs running on hardware can be debugged with both PCIe® using Xilinx® virtual cable (XVC) and JTAG using 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 in the three build configurations. Each feature and technique is described in more detail in the *SDAccel Environment Debugging Guide* (UG1281).

*Table 2:* **Features and Techniques for Debugging Different Build Configurations**

| Feature/ Technique | OS | Host | Kernel | FPGA (Platform) |
|---|---|---|---|---|
| Software Emulation | dmesg | GDB | GDB | xbutil |
| Hardware Emulation | dmesg | GDB | GDB<br>Kernel Waveform Viewer | xbutil |
| System | dmesg | GDB | Kernel Waveform Viewer<br>ILA | xbutil |

**Notes:**

1. dmesg is a Linux command.
2. GDB is the GNU Debugger.
3. xbutil is a Xilinx provided utility.

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

*Table 3:* **Software and Hardware Debugging Features and Techniques**

| Software-centric | Hardware-centric |
|---|---|
| GNU Debugger (GDB) | Kernel waveform viewer |
| Xilinx utility `xbutil` | Integrated Logic Analyzer (ILA) |
| Linux `dmesg` | N/A |

Using both software-centric, and hardware-centric debugging features, you can isolate and identify functional issues, protocol problems, as well as troubleshoot 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 correctness of the logic generated 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 runtime 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 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 and system performance is met. SDAccel provides specific hardware debug capabilities which include waveform analysis, kernel activity reports, and 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.

## GNU Debugging

For the GNU debugging (GDB), you can add breakpoints, inspect variables, and debug the kernel or host code. This familiar software debug flow allows quick debugging to validate the functionality. SDAccel also provides special GDB extensions to examine the content of the OpenCL runtime 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 using 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 should 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 using 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 should be familiar to 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 through the Xilinx Virtual Cable (XVC) using the Vivado tools.

# 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 Chapter 5: 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 cards. It supports kernels expressed in OpenCL™ C, C++ and RTL (SystemVerilog, Verilog, or VHDL).

This chapter walks through the command-line flow and shows how to build the software and hardware components from the command-line, or a script. 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 the 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/${XILINX_XRT}/include/
-I/${XILINX_VIVADO}/include/ -g -Wall -c -o vadd.o vadd.cpp
```

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

*Table 4:* **Host Code Compilation**

| Code | Description |
|---|---|
| xcpp | |
| -DSDX_PLATFORM=xilinx_vcu1525_dynamic_5_1 | (define macro) |
| -I/${XILINX_XRT}/include/ | (include directory of header files) |
| -I/${XILINX_VIVADO}/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 | (specify output file, vadd.o) |
| 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/${XILINX_XRT}/lib/
-rpath,/lnx64/lib/csim vadd.o
```

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

*Table 5:* **Linking the Host Code**

| Code | Description |
|---|---|
| xcpp | |
| -o vadd.exe | (create output file, vadd.exe) |
| -Wl | (specify warning level) |
| -lxilinxopencl -lpthread -lrt -lstdcc++ | (link with specified library files) |
| -L/${XILINX_XRT}/lib/ | (search directories for library files) |
| -rpath,/lnx64/lib/csim | (designate runtime search path) |
| vadd.o | (object code 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 here.

- 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_u200`.

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

- Specify the build target with the `-t / --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 generates 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_u200 vadd.cl vadd.h \
-o krnl_vadd.xo --report_level 2 --report_dir reports
```

This performs the following:

- Compile the kernel
- Name the Kernel `krnl_vadd`
- Use `xilinx_u200` platform
- Use `vadd_file1.cpp` and `vadd_file2.cpp` source files
- Generate an 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 Creating Multiple Instances of a Kernel, 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, Xilinx recommends adding one.

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

- As described in Mapping Kernel Interfaces to Memory Resources, 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_instance_name>.<interface_name>:<bank name>
```

- Linking is also done using the `-l`/`--link` option.

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

Putting it all together in an example:

```
xocc -l --platform xilinx_u200 -nk krnl_vadd:1:krnl_vadd1 \
--sp krnl_vadd1.m_axi_gmem:DDR[3] -o vadd.xclbin
```

This performs the following:

- Link the kernel
- Use the `xilinx_u200` 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 checks 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.

### Runtime 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 runtime 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.

*Table 6:* **Debug Group**

| Key | Valid Values | Descriptions |
|---|---|---|
| `debug` | `[true\|false]` | Enable or disable kernel debug.<br>• true: enable<br>• false: disable<br>• Default: false |
| `profile` | `[true\|false]` | Enable or disable OpenCL code profiling.<br>• true: enable<br>• false: disable<br>• Default: false |
| `timeline_trace` | `[true\|false]` | Enable or disable profile timeline trace<br>• true: enable<br>• false: disable<br>• Default: false |
| `device_profile` | `[true\|false]` | Enable or disable device profiling.<br>• true: enable<br>• false: disable<br>• Default: false |

*Table 7:* **Runtime Group**

| Key | Valid Values | Descriptions |
|---|---|---|
| `api_checks` | `[true\|false]` | Enable or disable OpenCL API checks.<br>• true: enable<br>• false: disable<br>• Default: true |
| `runtime_log` | `null console syslog filename` | Specify where the runtime logs are printed<br>• null: Do not print any logs.<br>• console: Print logs to `stdout`<br>• syslog: Print logs to Linux syslog<br>• filename: Print logs to the specified file. For example, runtime_log=my_run.log<br>• Default: null |

*Table 7:* **Runtime Group** *(cont'd)*

| Key | Valid Values | Descriptions |
|---|---|---|
| `polling_throttle` | An integer | Specify the time interval in microseconds that the runtime library polls the device status. Default: 0 |

*Table 8:* **Emulation Group**

| Key | Valid Values | Descriptions |
|---|---|---|
| `aliveness_message_interval` | Any integer | Specify the interval in seconds that aliveness messages need to be printed<br>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`<br>• true = print in users console<br>• false = do not print in user console<br>• Default: true |
| `print_warnings_in_console` | `[true\|false]` | Controls the printing emulation warning messages to users console. Emulation warning messages are always logged into a file called `emulation_debug.log`.<br>• true = print in users console<br>• false = do not print in user console<br>• 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`.<br>• true = print in users console<br>• false = do not print in user console<br>• 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.<br>• true: enable<br>• false: disable<br>• Default: false |

*Table 8:* **Emulation Group** *(cont'd)*

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

# emconfigutil Settings

The emconfigutil command and options can be provided in the Command field under emconfigutil to create an emulation configuration file.

For more information on emconfigutil and its options, refer to the "emconfigutil (Emulation Configuration) Utility" section in *SDx Command and Utility Reference Guide* (UG1279).

*Figure 49:* **emconfigutil Settings**

# 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. SDAccel™ allows RTL designs to be used, however they must adhere to the software and hardware requirements to be used within the tool flow and runtime 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

An RTL design must meet both interface and software requirements to be used as an RTL kernel within the SDAccel framework.

It might be necessary to add or modify the original RTL design to meet these requirements, which are outlined in the following sections.

### Kernel Interface Requirements

To satisfy the SDAccel execution model, an RTL kernel must adhere to the following interface requirements:

- A single slave AXI4-Lite interface used to access control registers (to pass scalar arguments and to start/stop the kernel) is required.
- At least one of the following interfaces (can have both interfaces):
  - AXI4 master interface to communicate with memory.
  - AXI4-Stream interface for transferring data between kernels.
- At least one clock interface port.

The various interface requirements are summarized in the following table.

***Note*:** In some instances the port names must be written exactly.

*Table 9:* **RTL Kernel Interface and Port Requirements**

| Port or Interface | Description | Comment |
|---|---|---|
| ap_clk | Primary clock input port | • Name must be exact.<br>• Required port. |
| ap_clk_2 | Secondary optional clock input port | • Name must be exact.<br>• Optional port. |
| ap_rst_n | Primary active-Low reset input port | • Name must be exact.<br>• Optional port.<br>• This signal should be internally pipelined to improve timing.<br>• 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.<br>• Optional port.<br>• This signal should be internally pipelined to improve timing.<br>• This signal is driven by a synchronous reset in the ap_clk_2 clock domain. |
| interrupt | Active-High interrupt. | • Name must be exact.<br>• Optional port. |
| s_axi_control | One and only one AXI4-Lite slave control interface | • Name must be exact; case sensitive.<br>• Required port. |
| AXI4_MASTER | One or more AXI4 master interfaces for global memory access | • All AXI4 master interfaces must have 64-bit addresses.<br>• 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.<br>• 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.<br>• Any user logic or RTL code that does not conform to the requirements above, must be wrapped or bridged to satisfy these requirements. |

# Kernel Software Requirements

RTL kernels have the same software interface model as OpenCL™ and C/C++ kernels. That is, they are seen by the host application as functions with a void return value, scalar arguments, and pointer arguments. For instance:

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

The SDAccel execution model dictates the following:

- Scalar arguments are directly written to the kernel through an AXI4-Lite slave interface.

- Pointer arguments are transferred to/from memory.
- Kernels are expected to read/write data in memory though one or more AXI4 memory map interface.
- Kernels are controlled by the host application through the control register (shown below) through the AXI4-Lite slave interface.

If the RTL design has a different execution model, it must be adapted to ensure that it can be completed in this manner.

The following table outlines the required register map such that a kernel can be used within the SDAccel environment. The control register is required by all kernels while the interrupt related registers are only required for designs with interrupts. All user-defined registers must begin at location `0x10`; locations below this are reserved.

*Table 10:* **Address Map**

| Address | Name | Description |
|---|---|---|
| 0x0 | Control | Controls and provides kernel status. |
| 0x4 | Global Interrupt Enable | Used to enable interrupt to the host. |
| 0x8 | IP Interrupt Enable | Used to control which IP generated signal are used to generate an interrupt. |
| 0xC | IP Interrupt Status | Provides interrupt status. |
| 0x10 | Kernel arguments | This would include scalars and global memory arguments for instance. |

*Table 11:* **Control (0x0)**

| Bit | Name | Description |
|---|---|---|
| 0 | ap_start | Asserted when kernel can start processing data. Cleared on handshake with ap_done being asserted. |
| 1 | ap_done | Asserted when kernel has completed operation. Cleared on read. |
| 2 | ap_idle | Asserted when kernel is idle. |
| 31:3 | Reserved | Reserved |

***Note*:** 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.

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

*Table 12:* **Global Interrupt Enable (0x4)**

| Bit | Name | Description |
|---|---|---|
| 0 | Global Interrupt Enable | When asserted, along with the IP Interrupt Enable bit, the interrupt is enabled. |
| 31:1 | Reserved | Reserved |

*Table 13:* **IP Interrupt Enable (0x8)**

| Bit | Name | Description |
|---|---|---|
| 0 | Interrupt Enable | When asserted, along with the Global Interrupt Enable bit, the interrupt is enabled. |
| 31:1 | Reserved | Reserved |

*Table 14:* **IP Interrupt Status (0xC)**

| Bit | Name | Description |
|---|---|---|
| 0 | Interrupt Status | Toggle on write. |
| 31:1 | Reserved | Reserved |

## Interrupt

RTL kernels can optionally have an interrupt port containing a single interrupt. The port name must be called `interrupt` and be active-High. It is enabled when both the Global Interrupt Enable (GIE) and Interrupt Enable Register (IER) bits are asserted. Further, the interrupt is cleared only when writing a one to bit-0 of the IP Interrupt Status Register.

If adding an `interrupt` port to the kernel, the `kernel.xml` file needs be updated with this information. The `kernel.xml` is generated automatically when using the RTL Kernel Wizard. For details on updating the file, see Create Kernel Description XML File.

# 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. You can use this wrapper to wrap your RTL IP into an RTL kernel accessible by theSDAccel framework.

***Note*:** It is not required to use the code generated by the Wizard. You can completely generate your own RTL kernel as long as it meets the software and interface requirements outline above.

If you do use the generated wrapper, you need to replace the generated RTL IP (VADD) with your RTL IP and connect 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, perform the following:

1. Launch the SDx Development Environment.
2. Create an SDx Project (Application Project Type).
3. Click **Xilinx → RTL Kernel Wizard**.

To launch the RTL Kernel Wizard from Vivado IDE, perform 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.

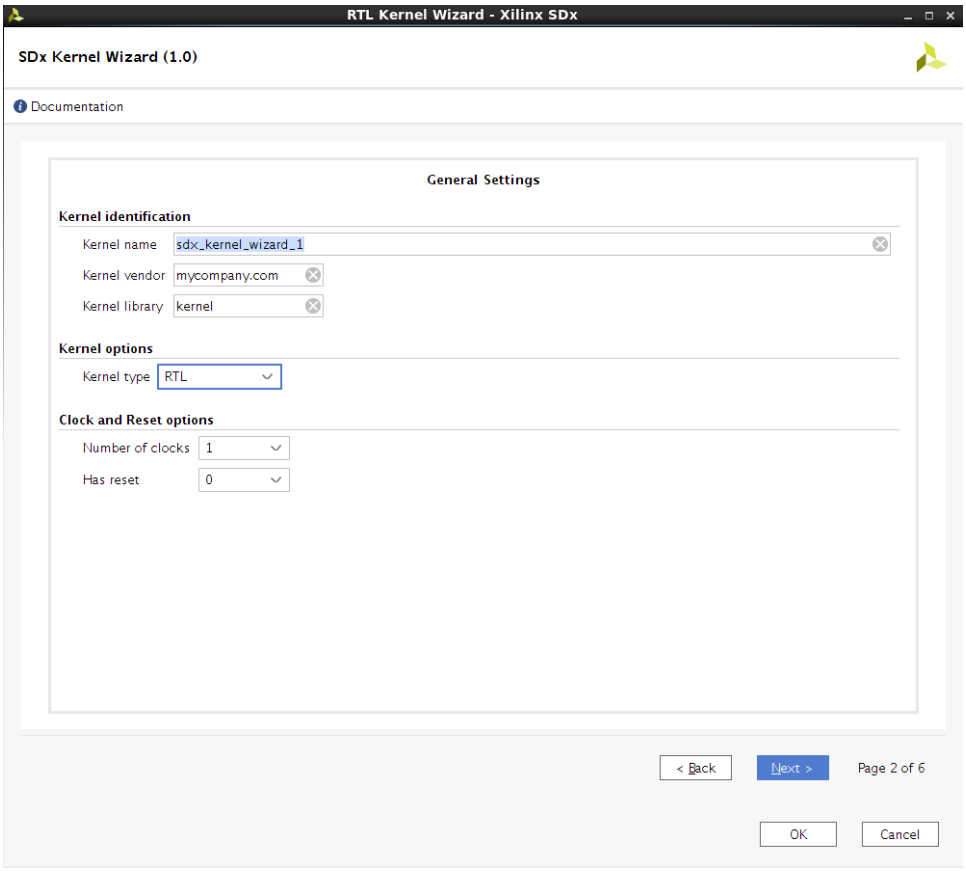
# Using the RTL Kernel Wizard

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 inputs of the wizard, 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 50:* **RTL Kernel Wizard General Settings**




## Kernel Identification

The following are 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 Vendor/Library/Name/Version (VLNV) format described in the *Vivado Design Suite User Guide: Designing with IP* (UG896).
- **Kernel library:** The name of the library. Used in the VLNV. Must conform to the same identifier rules.

### Kernel Options

- **Kernel type:** 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.
- **Enable MicroBlaze debug (only available on select configurations):** Adds a MicroBlaze Debug Module (MDM) to a Block Design Kernel type example. The boundary scan interface of the MDM module is connected to the top-level of the kernel. The debug interface is connected to the MicroBlaze instance. This option is only available for platforms that support system debug over the Xilinx® Virtual Cable and if the Kernel type is chosen as Block Design.

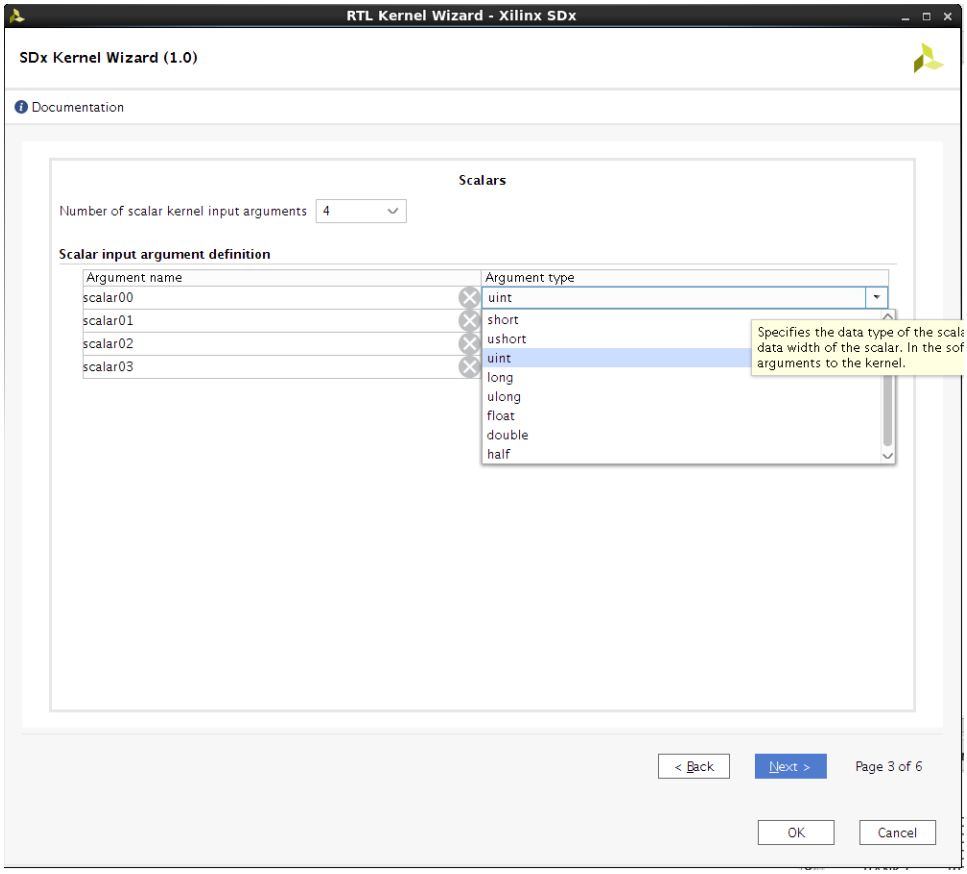
### Clock and Reset Options

- **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_n`. 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 is a reset port included with each clock. Block Design type kernels must have a reset input.

## *Scalar 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 register is created to facilitate passing the argument from software to hardware. See the following figure.

*Figure 51:* **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.

The following is the scalar input argument definition:

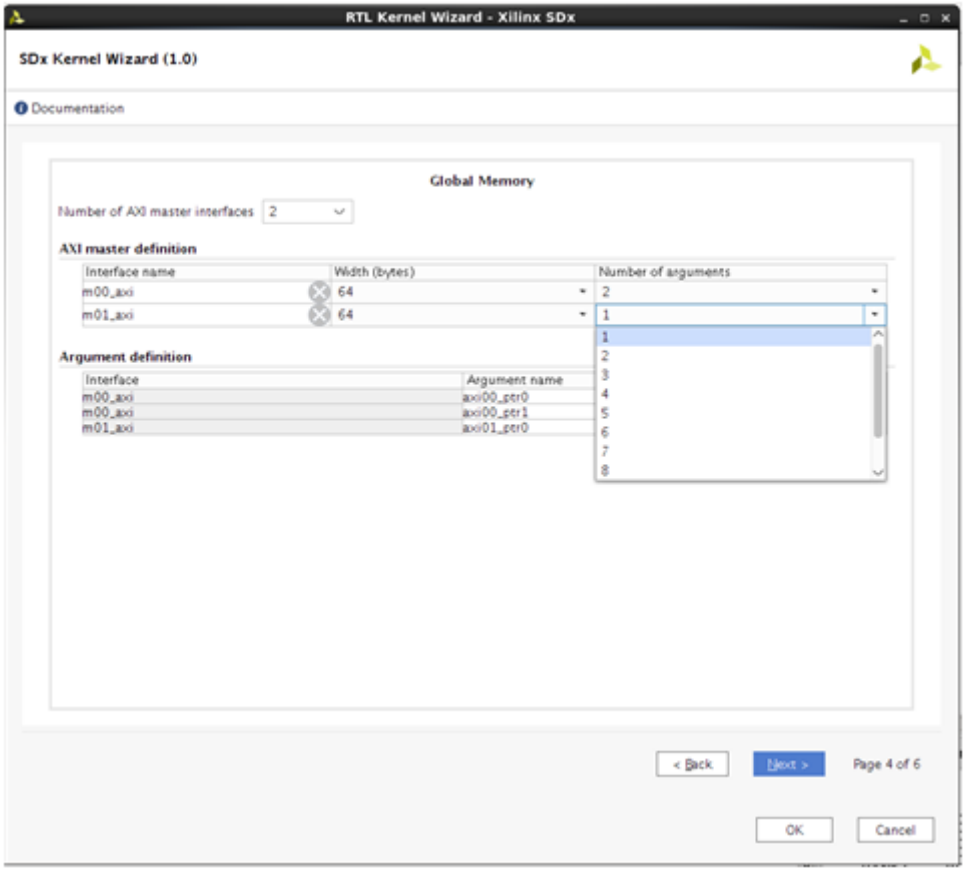
- **Argument name:** The argument name is used in the generated Verilog control register module as an output signal. Each argument is assigned an ID value. This ID value is used to access the argument from the host software. The ID value assignments can be found on the summary page of this wizard. To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

- **Argument type:** Specifies the data type, and hence bit-width, of the argument. This affects the register width in the generated Verilog module. The data types available are limited to the ones specified by the OpenCL C Specification Version 2.0 in "6.1.1 Built-in Scalar Data Types" section. The specification provides the associated bit-widths for each data type. The RTL wizard reserves 64 bits for all scalars in the register map regardless of their argument type. If the argument type is 32 bits or less, the RTL Wizard sets the upper 32 bits (of the 64 bits allocated) as a reserved address location. Data types that represent a bit width greater than 32 bits require two write operations to the control registers.

## Global Memory

Global memory is accessed by the kernel through AXI4 master interfaces (see the following figure).

*Figure 52:* **Global Memory**




Each AXI4 interface operates independently of each other, and each AXI4 interface can be connected to one or more memory controllers to off-chip memory such as DDR4. Global memory is primarily used to pass large data sets to and from the kernel from the host. It can also be used to pass data between kernels. 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.

In the Global Memory dialog box, you can specify the **Number of AXI master interfaces** present on the kernel. The maximum is 16 interfaces. For each interface, you can customize an interface name, data width, and the number of associated arguments. Each interface contains all read and write channels. The default names proposed by the RTL kernel wizard are `m00_axi` and `m01_axi`. If not changed, these names will have to be used when assigning a DDR bank through the `--sp` option.

### AXI Master Definition (Table Columns)

- **Interface name:** Specifies the name of the interface. To ensure maximum compatibility, the argument name follows the same identifier rules as the kernel name.
- **Width (in bytes):** Specifies the data width of the AXI data channels. Xilinx recommends matching to the native data width of the memory controller AXI4 slave interface. The memory controller slave interface is typically 64 bytes (512 bits) wide.
- **Number of arguments:** Specifies the number of arguments to associate with this interface. Each argument represents a data pointer to global memory that the kernel can access.

### Argument Definition

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

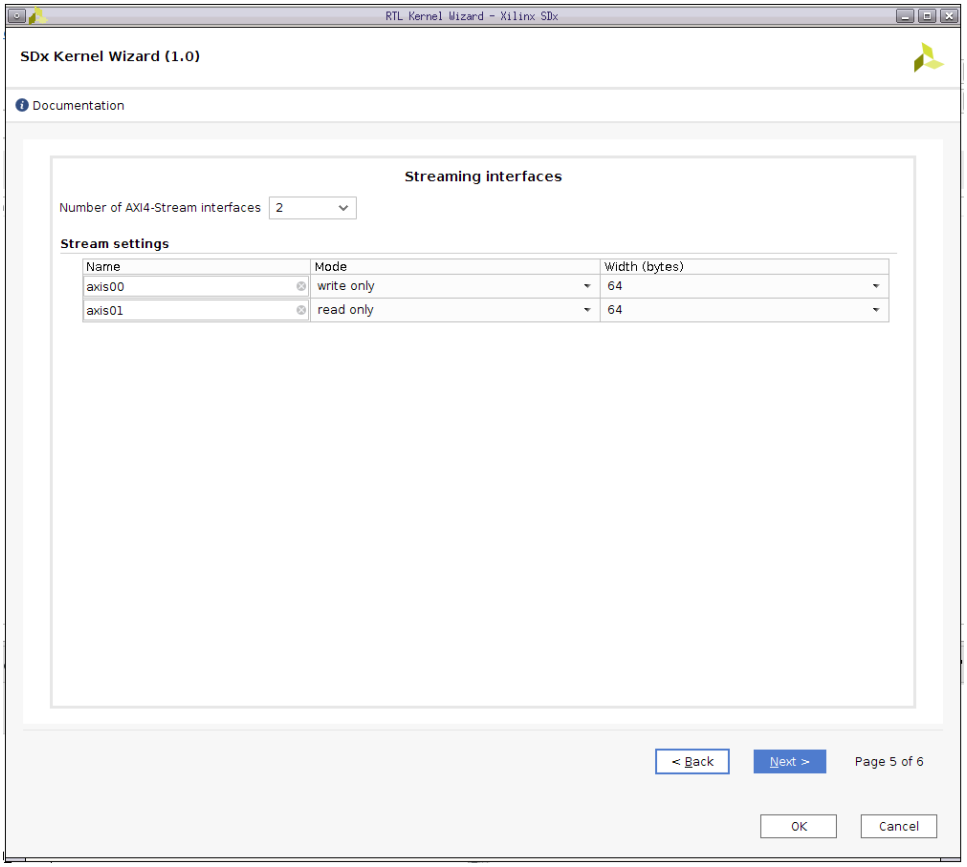
### Streaming Interfaces

The streaming interfaces page allows configuration of AXI4-Stream interfaces on the kernel. Streaming interfaces are only available on select platforms and if the chosen platform does not support streaming, then the page does not appear. Streaming interfaces are used for direct host-to-kernel and kernel-to-host communication. Each interface automatically maps to the host over the PCIe® interface when the kernel is linked to the platform. The stream protocol uses the `TDATA/TKEEP/TLAST` signals of the AXI4-Stream protocol. Stream transactions consists of a series of transfers where the final transfer is terminated with the assertion of the `TLAST` signal. The following figure shows the configuration options. Stream transfers must adhere to the following:

- AXI4-Stream transfer occurs when `TVALID/TREADY` are both asserted.
- `TDATA` must be 8, 16, 32, 64, 128, 256, or 512 bits wide.
- `TKEEP` (per byte) must be all 1s when `TLAST` is 0.
- `TKEEP` can be used to signal a ragged tail when `TLAST` is 1. For example, on a 4-byte interface, `TKEEP` can only be `0b0001`, `0b0011`, `0b0111`, or `0b1111` to specify the last transfer is 1-byte, 2 bytes, 3 bytes, or 4 bytes in size, respectively.
- `TKEEP` cannot be all zeros (even if `TLAST` is 1).
- `TLAST` must be asserted at the end of a packet.

- `TREADY` input/`TVALID` output should be low if kernel is not started to avoid lost transfers.

*Figure 53:* **Streaming Interfaces**

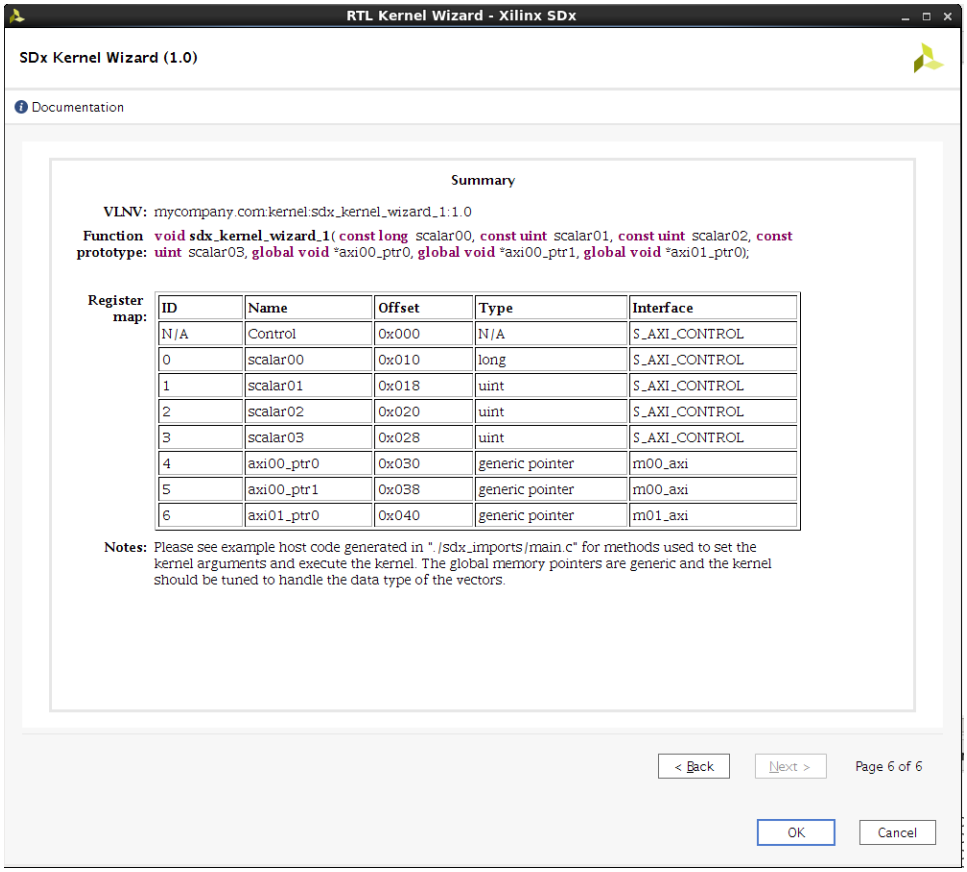



- **Number of AXI4-Stream interfaces:** Specifies the number of AXI4-Stream interfaces that exist on the kernel. A maximum of 32 interfaces can be enabled per kernel. Xilinx recommends keeping the number of interfaces as low as possible to reduce the amount of area consumed.
- **Name:** Specifies the name of the interface. To ensure maximum compatibility, the argument name follows the same identifier rules as the kernel name.
- **Mode:** Specifies the direction of the interface. A read only interface is an AXI4-Stream slave interface and can be sent data with the `clWriteStream` API. A write only interface is an AXI4-Stream master interface and the host can receive data from the interface with the `clReadStream` API.
- **Width (bytes):** Specifies the `TDATA` width (in bytes) of the AXI4-Stream interface. This interface width is limited to 1 to 64 bytes in powers of 2.

## 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 54:* **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 the `.xci` file in the Design Sources View in Vivado, and select **Open IP Example Design**.
3. In the open example design window, select an output directory (or accept default) and click **OK**. This opens a new Vivado project with the example design in it.
4. You can now close the current Vivado project from which the RTL Kernel Wizard was invoked.

### *Interrupt*

By default, the RTL Kernel Wizard creates a single interrupt port, named `interrupt`, along with the interrupt logic in the Control Register block. This is reflected in the generated Verilog code and the associated `compent.xml` and `kernel.xml` files.

The interrupt is active-High and is enabled by setting both the Global Interrupt Enable (GIE) and Interrupt Enable (IER) registers. By default, the IER uses the internal `ap_done` signal to trigger an interrupt.

An interrupt is only cleared by writing one to bit 0 of the ISR.

## 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.

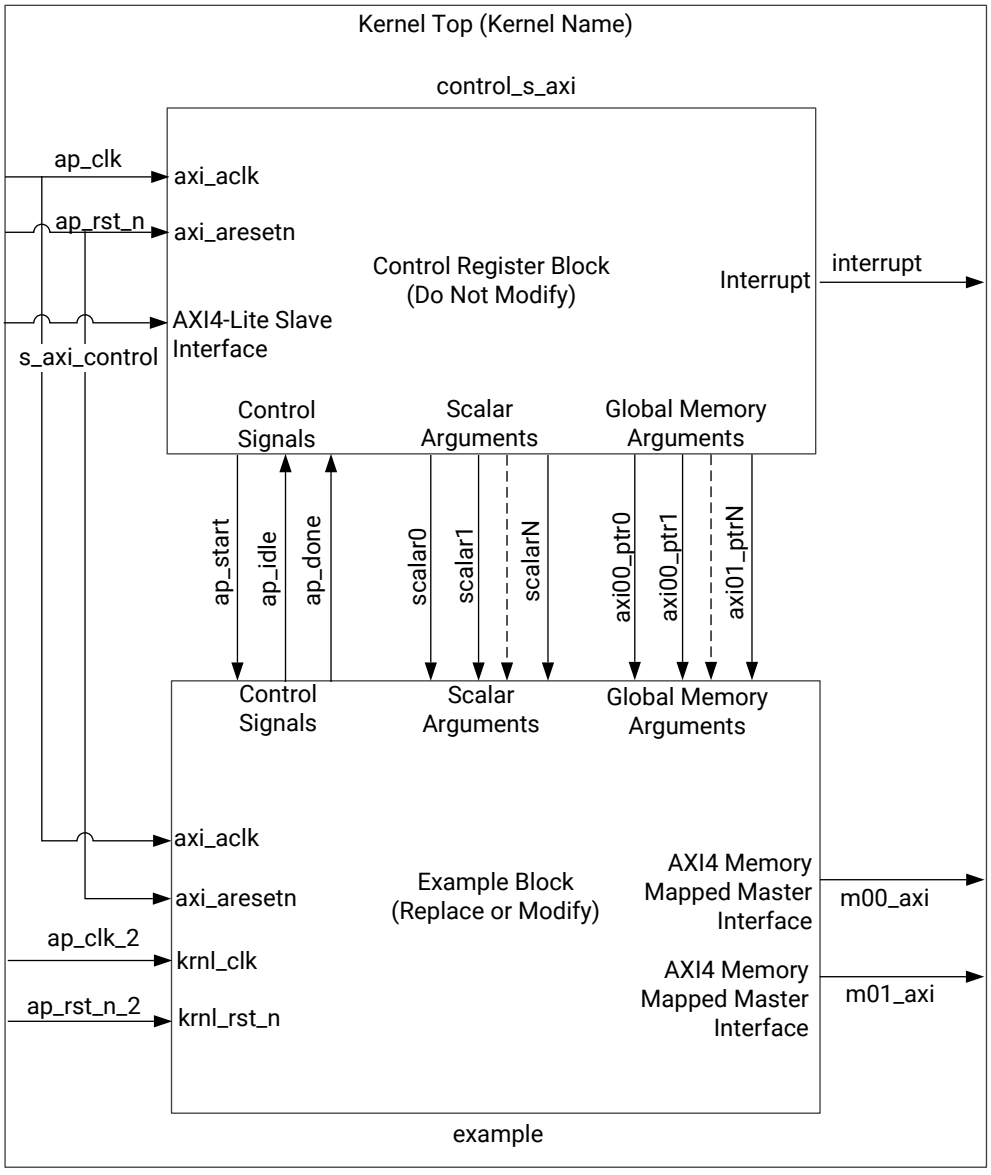
The top-level Verilog file contains the expected input/output signals and parameters. These top-level ports are matched to the kernel specification file (`kernel.xml`) and when combined with the rest of the RTL/block design becomes the acceleration kernel. The AXI4 interfaces defined at the top-level file contain a minimum subset of AXI4 signals required to generate an efficient, high throughput interface. Signals omitted 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 selected **Kernel Type**, 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 example design. The following figure illustrates the top-level design configured with two AXI4-master interfaces. Care should be taken if the Control Register module is modified to ensure that it still aligns with the `kernel.xml` file located in the imports directory of the Vivado kernel project. The example sub-module can be replaced with your custom logic or used as a starting point for your design.

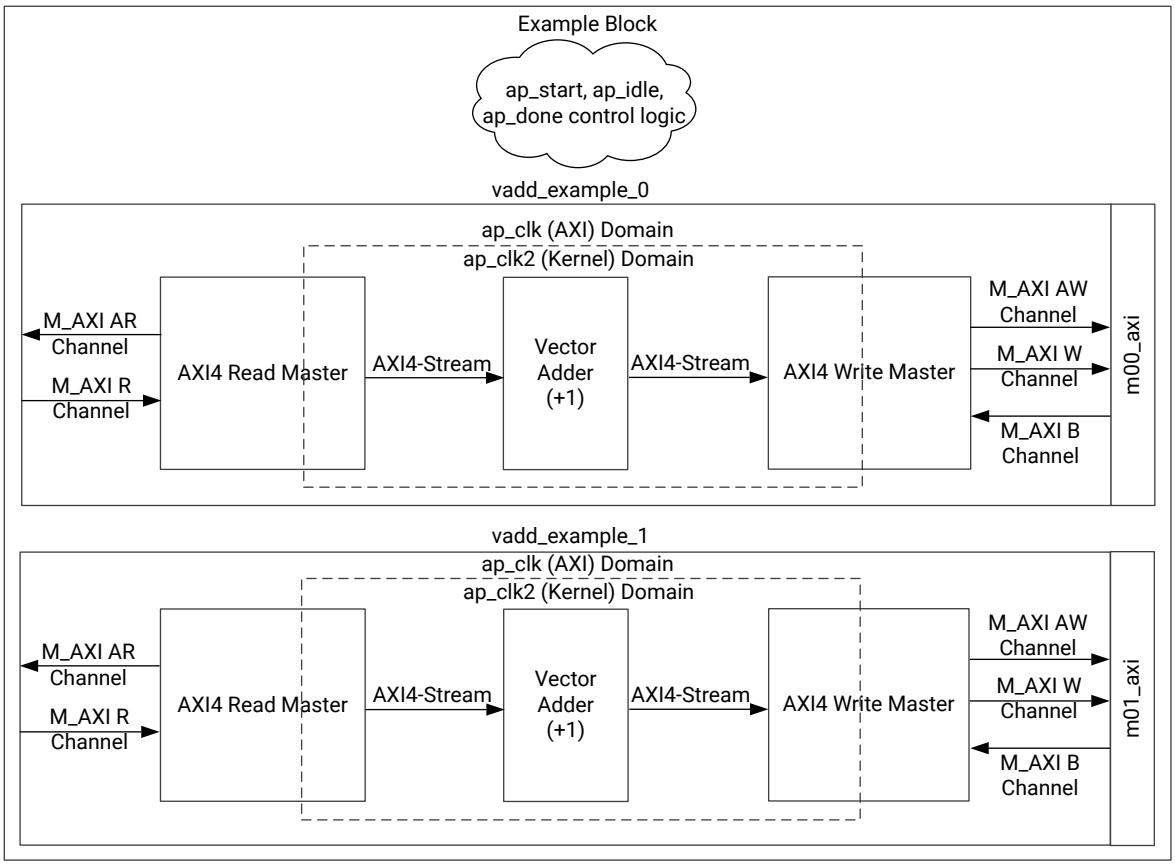
*Figure 55:* **Kernel Type RTL Top**




X22079-011019

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 56:* **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 15:* **RTL Kernel 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 |

*Table 15:* **RTL Kernel Wizard Source and Test Bench File** *(cont'd)*

| Filename | Description | Delivered with Kernel Type |
|---|---|---|
| <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>_cmodel.cpp | Software C-Model example for software emulation. | All |
| <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** | | |
| src/host_example.cpp | 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 opens, 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 opens, click **X** just to the right of the text on the Welcome tab to close the welcome dialog box. This shows an already loaded SDK project underneath.
4. From the Project Explorer, the source files are under the `<Kernel Name>_control/src` section. Modify these as appropriate.
5. When updates are complete, compile the source by selecting the menu option **Project → Build All → Check for errors/warnings and resolve if necessary**. The ELF file is automatically updated in the 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 exercises 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 writes/reads from the control registers and executes the kernel multiple times while also including a simple reset test. It is also useful for debugging AXI issues, reset issues, bugs during multiple iterations, and kernel functionality. Compared to hardware emulation, it executes a more rigorous test of the hardware corner cases, but does not test the interaction between host code and kernel.

To run a simulation, click **Vivado Flow Navigator → Run Simulation** located on the left hand side of the GUI and select **Run Behavioral Simulation**. If behavioral simulation is working as expected, a post-synthesis functional simulation can be run to ensure that synthesis 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 synthesizes without errors. It also provides 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 OOC 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 has 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 expects 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 instantiates the kernel, allocates the buffers, sets the kernel arguments, executes the kernel, and then collects and checks the results for the example *add one* function.

## *Package 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 re-target the new device and it must be regenerated from the source. If the design contains a block design, the netlist (DCP) based kernel is the only packaging option available.

Optionally, 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 uses 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** revisits 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.

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.

## 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 User Guide: Creating and Packaging Custom IP* (UG1118) for details on IP packaging in Vivado.

The following interface packaging is required for the RTL Kernel:

- The AXI4-Lite interface name must be packaged as `S_AXI_CONTROL`, but the underlying AXI ports can be named differently.

- The AXI4 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` is set to read-only or write-only.

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

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

1. *If the constraints are given in a* `.ttcl` *file, add* `<: setFileProcessingOrder "late" :>` *to the* `.ttcl` *preamble section of the file as shown below:*

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

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

```
<spirit:file>
        <spirit:name>ttcl/my_ip_constraint.xdc</spirit:name>
        <spirit:userFileType>ttcl</spirit:userFileType>
        <spirit:userFileType>USED_IN_implementation</spirit:userFileType>
        <spirit:userFileType>USED_IN_synthesis</spirit:userFileType>
        <spirit:define>
             <spirit:name>processing_order</spirit:name>
             <spirit:value>late</spirit:value>
        </spirit:define>
</spirit:file>
```

# Create Kernel Description XML File

A kernel description XML file needs to be created for each RTL kernel such that it can be used in the SDAccel environment. The file must be called `kernel.xml`. The XML file specifies kernel attributes like the register map and ports which are needed by the runtime and SDAccel flows. The following is an example of a `kernel.xml` file.

```
<?xml version="1.0" encoding="UTF-8"?>
<root versionMajor="1" versionMinor="6">
  <kernel name="sdx_kernel_wizard_0" language="ip_c"
vlnv="mycompany.com:kernel:sdx_kernel_wizard_0:1.0" attributes=""
preferredWorkGroupSizeMultiple="0" workGroupSize="1" interrupt="true">
    <ports>
      <port name="s_axi_control" mode="slave" range="0x1000" dataWidth="32"
portType="addressable" base="0x0"/>
      <port name="m00_axi" mode="master" range="0xFFFFFFFFFFFFFFFF"
dataWidth="512" portType="addressable" base="0x0"/>
    </ports>
    <args>
      <arg name="axi00_ptr0" addressQualifier="1" id="0" port="m00_axi"
size="0x8" offset="0x010" type="int*" hostOffset="0x0" hostSize="0x8"/>
    </args>
  </kernel>
</root>
```

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

*Table 16:* **Kernel XML Format**

| Tag | Attribute | Description |
|---|---|---|
| <root> | versionMajor | Set to 1 for the current release of SDAccel. |
| | versionMinor | Set to 6 for the current release of SDAccel. |
| <kernel> | name | Kernel name |
| | language | Always set it to `ip_c` 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:<br>`<spirit:vendor>xilinx.com</spirit:vendor>`<br>`<spirit:library>hls</spirit:library>`<br>`<spirit:name>test_sincos</spirit:name>`<br>`<spirit:version>1.0</spirit:version>`<br>The vlnv attribute in kernel XML must be set to:<br>`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. |
| | interrupt | Set equal to "true" (that is,. interrupt="true") if interrupt present else omit. |

*Table 16:* **Kernel XML Format** *(cont'd)*

| Tag | Attribute | Description |
|---|---|---|
| <port> | name | Port name. At least an AXI4 master port and an AXI4-Lite slave port are required. 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."<br>• For AXI4 slave port, set it to "slave."<br>• For AXI4-Stream master port, set it to "write_only."<br>• 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.<br>• For AXI4 master and slave ports, set it to "addressable."<br>• For AXI4-Stream ports, set it to "stream." |
| | base | For AXI4 master and slave ports, set to `0x0`. This tag is not applicable to AXI4-Stream ports. |
| <arg> | name | Kernel argument name. |
| | addressQualifier | Valid values:<br>0: Scalar kernel input argument<br>1: global memory<br>2: local memory<br>3: constant memory<br>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.<br>Not applicable for AXI4-Stream ports. |
| | port | Indicates the port to which the `arg` is connected. |
| | 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 AXI4 master and slave ports.<br>For AXI4-Stream ports, `memSize` sets the depth of the created FIFO. |
| The following tags specify additional information for AXI4-Stream ports. They are not applicable to AXI4 master or slave ports. | | |

*Table 16:* **Kernel XML Format** *(cont'd)*

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

## Package RTL Kernel into Xilinx Object File

The final step is to package the RTL IP and the associated kernel XML file together into a Xilinx object file (`.xo`) so it can be used by the SDAccel compiler. The following example command line packages `test_sincos` RTL IP and `kernel.xml` into object file named `test.xo`.

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

# Designing RTL Recommendations

While the RTL Kernel 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).

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

These topics are described in the following subsections.

# Memory Performance Optimizations for AXI4 Interface

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

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

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

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

# Managing Clocks in an RTL Kernel

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

If you require additional clocks within the RTL kernel, a frequency synthesizer such as the Clocking Wizard IP or MMCM/PLL primitive can be instantiated within the RTL kernel.

Thus your RTL kernel can use just the primary clock, both primary and secondary clock, or primary and secondary clock along with an internal frequency synthesizer. The following shows the advantages and disadvantages of using these three RTL kernel clocking methods:

- Single input clock: `ap_clk`
  - External interfaces and internal kernel logic run at the same frequency.
  - No clock domain crossing (CDC) issues.
  - Frequency of `ap_clk` can automatically be scaled to allow kernel to meet timing.
- Two input clocks: `ap_clk` and `ap_clk_2`
  - Kernel logic can run at either clock frequency.
  - Need proper CDC technique to move from one frequency to another.
  - Both `ap_clk` and `ap_clk_2` can automatically scale their frequencies independently to allow the kernel to meet timing.
- Using a frequency synthesizer inside the kernel:
  - Additional device resources required to generate clocks.
  - Must have `ap_clk` and optionally `ap_clk_2` interfaces.
  - Generated clocks can have different frequencies for different CUs.
  - Kernel logic can run at any available clock frequency.
  - Need proper CDC technique to move from one frequency to another.

When using a frequency synthesizer in the RTL kernel there are some constraints you should be aware of:

1. RTL external interfaces are clocked at `ap_clk`.
2. The frequency synthesizer can have multiple output clocks that are used as internal clocks to the RTL kernel.
3. You must provide a Tcl script to downgrade the clock resource placement DRCs in Vivado placement to stop a Vivado DRC error from occurring. An example of the Tcl command follows:

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

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

4. Use the `xocc --xp` option to specify the above Tcl script for use by Vivado implementation, after optimization. For example:

```
--xp vivado_prop:run.impl_1.STEPS.OPT_DESIGN.TCL.POST={<PATH>/<TCL
Script>}
```

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

   ```
   xocc --kernel_frequency 250
   ```

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

   ```
   xocc --kernel_frequency 0:250|1:500
   ```

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

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

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

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

# Quality of Results Considerations

The following recommendations help improve results for timing and area:

- Pipeline all reset inputs and internally distribute resets avoiding high fanout nets.
- Reset only essential control logic flip-flops (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 (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.

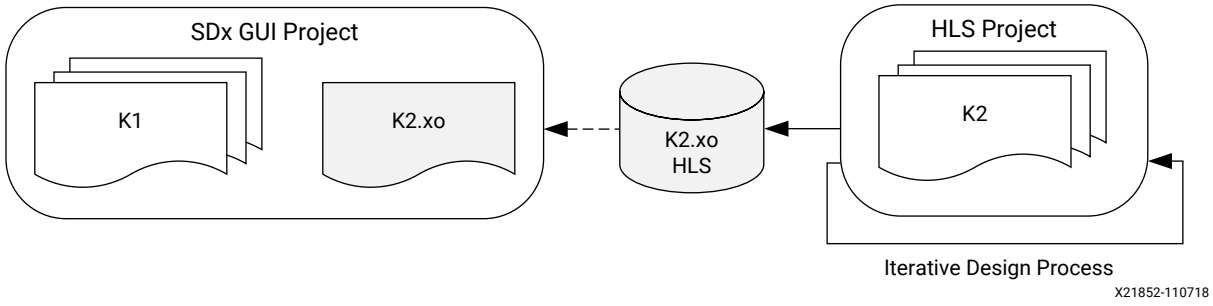
Chapter 10

# HLS Kernel Design Integration into SDAccel

The major flow described in the application-centric methodology of this guide is concerned with accelerator kernels being developed and integrated in the host application of a project in a top-down model. This implies, all source code is presented to SDAccel™ and sections of it are dedicated to being synthesized into accelerator modules. This flow calls the Vivado® High-Level Synthesis (HLS) tool to translate the function into hardware implementable accelerator code.

Alternatively, SDAccel provides a bottom-up flow, where HLS-based hardware kernels are created directly by importing from a Vivado HLS project. This allows you to perform optimizations and to validate kernel performance within the Vivado HLS project. When your kernel meets performance and resource requirements, the resulting Xilinx® object file (`.xo`) is handed off for inclusion into the SDx™ project. During hand-off, all kernel Vivado HLS optimization is maintained.

*Figure 57:* **Vivado HLS Design Flow**




The benefits of the bottom-up flow include:

- Designer can design, validate, and optimize the kernel prior to integration into the complete SDAccel project.
- Specific kernel optimizations are maintained for each kernel.
- Independent Vivado HLS and project locations allow separation of application and kernels.
- VHLS project can be used by multiple different projects, like a library instantiation.
- Allow teams to collaborate for increased productivity.

# Creating SDAccel Kernels with Vivado HLS

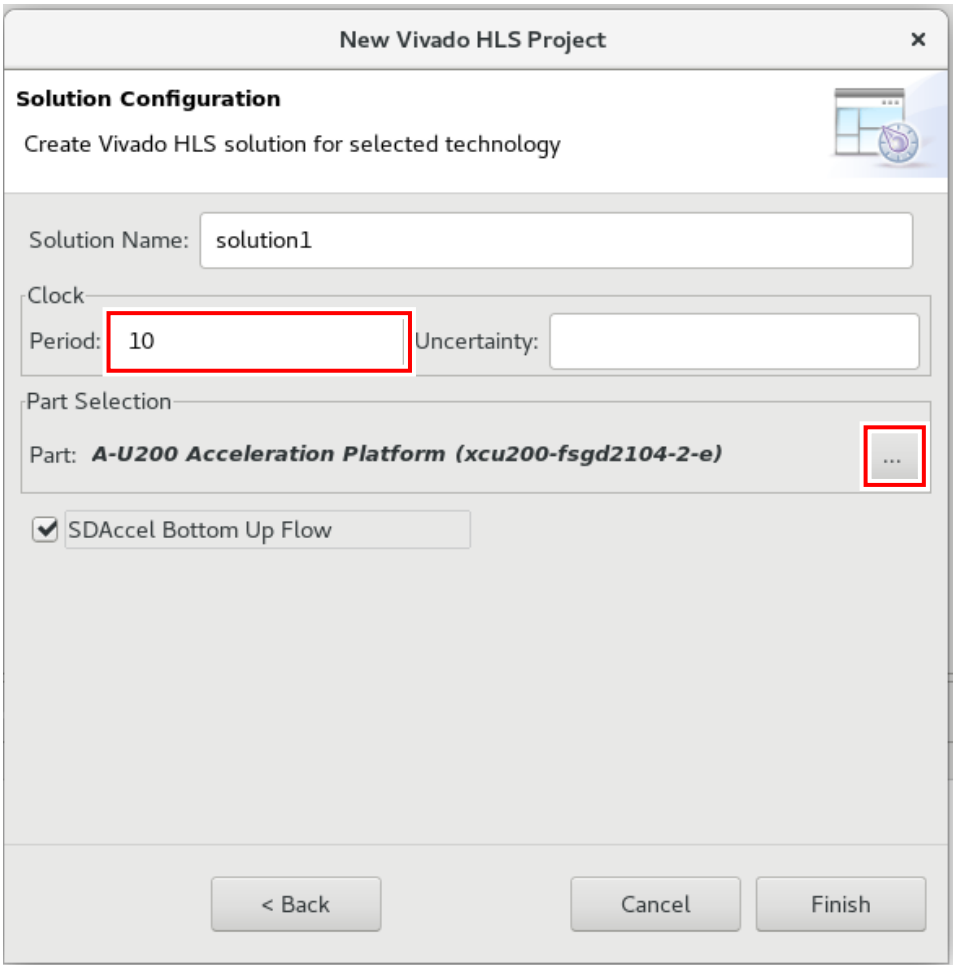
Running Vivado HLS to generate kernels from C/C++ for SDAccel follows the regular Vivado HLS flow. However, since the kernel is supposed to operate as an accelerator in an SDAccel, the SDAccel kernel modeling guidelines need to be followed (see C/C++ modeling guide). Most importantly, the interfaces need to be modeled as AXI memory interfaces except for scalar parameters called by the value, which are mapped to an AXI4-Lite interface. This is illustrated in the following example:

```
void krnl_idct(const ap_int<512> *block,
               const ap_uint<512> *q,
               ap_int<512> *voutp,
               int ignore_dc,
               unsigned int blocks) {
  #pragma HLS INTERFACE m_axi     port=block     offset=slave
bundle=p0       depth=512
  #pragma HLS INTERFACE s_axilite port=block                  bundle=control
  #pragma HLS INTERFACE m_axi     port=q         offset=slave
bundle=p1       depth=2
  #pragma HLS INTERFACE s_axilite port=q                      bundle=control
  #pragma HLS INTERFACE m_axi     port=voutp     offset=slave
bundle=p2       depth=512
  #pragma HLS INTERFACE s_axilite port=voutp                  bundle=control
  #pragma HLS INTERFACE s_axilite port=ignore_dc              bundle=control
  #pragma HLS INTERFACE s_axilite port=blocks                 bundle=control
  #pragma HLS INTERFACE s_axilite port=return                 bundle=control
```

**RECOMMENDED:** *The use of `ap-datatypes` in the interfaces require the use of `ap-datatypes` in the test bench for HLS. This might result in slower C/C++ simulation speeds and mapping to native C/C++ should be considered. As most host code is based on native data types, using them in the kernel interfaces is recommended.*

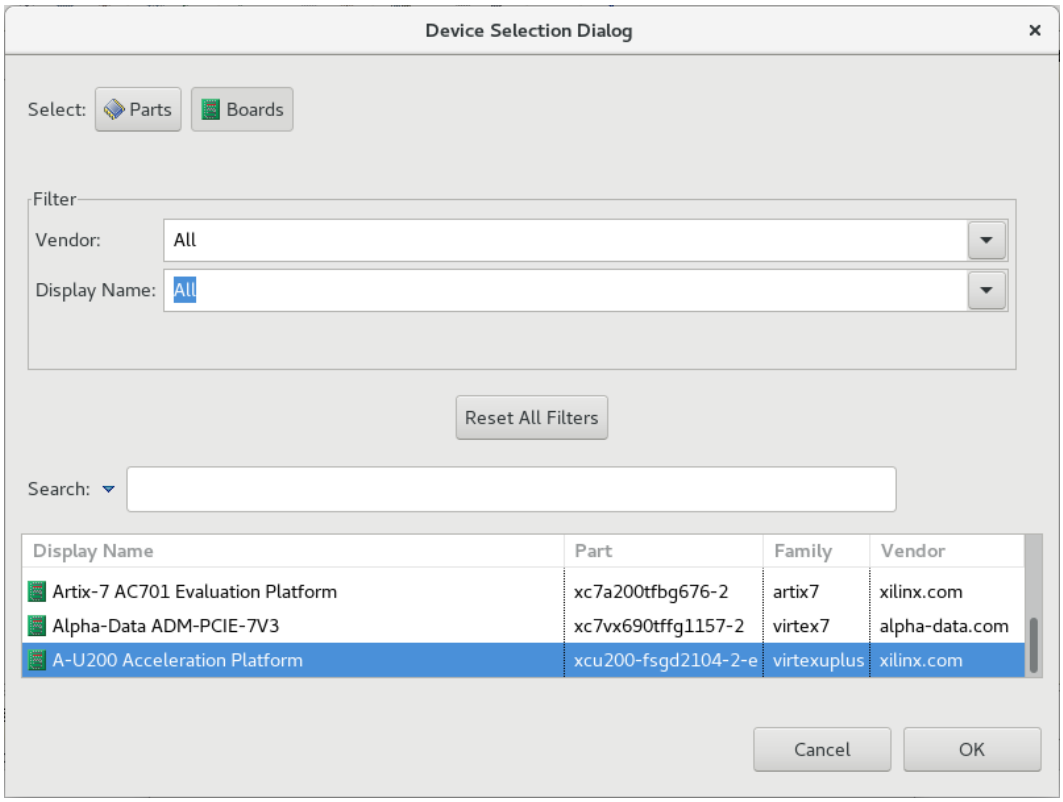
It is important to specify the **Clock Period** and **Part Selection**. This is done through the New Vivado HLS Project dialog in the Vivado HLS GUI flow:

*Figure 58:* **New Vivado HLS Project**




Choose the platform by clicking the **Browse** button to open the Device Selection Dialog and select the accelerator board from the Device list.

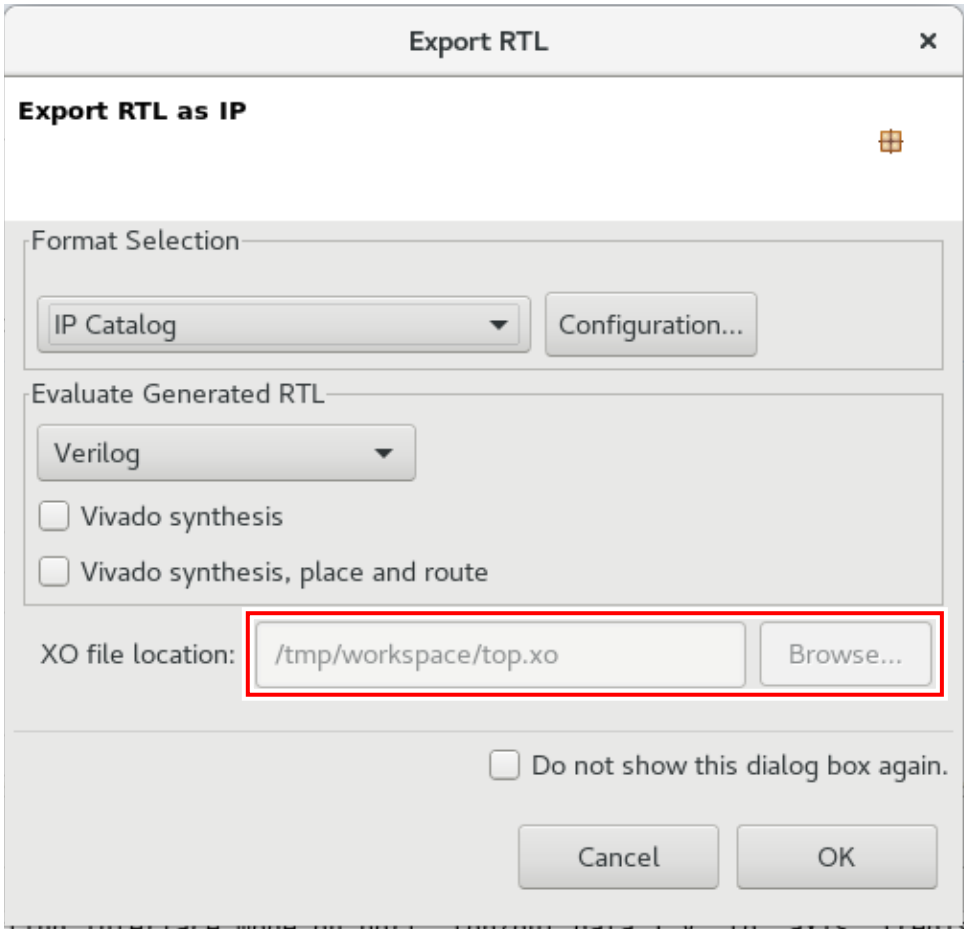
*Figure 59:* **Device Selection**




When completed, the iterative optimization process can resume until the best possible implementation results are achieved. For more information, see *Vivado Design Suite User Guide: High-Level Synthesis* (UG902).

After synthesis is completed for the optimized design, it needs to be exported to the SDAccel tool chain. The export command is available through the **Main Toolbar → Solution → Export RTL** menu item.

*Figure 60:* **Export RTL as IP**




It is only necessary to confirm the XO file location, which names the generated `XO-File` that is imported in the next section back into SDAccel.

***Note*:** Most of the options shown in the previous section can also be set and changed from a running project through the **Main Menu → Solution → Solution Settings**. The Synthesis and Export sections have the same content as previously shown in this documentation.

This completes the HLS synthesis part for SDAccel. In the following section, some required details for the command line flow are shown.

## Typical Vivado HLS Script for SDAccel Synthesis

If you run your HLS synthesis through command line scripts, the following Tcl code is equivalent to the GUI flow shown before:

```
open_project guiProj
set_top krnl_idct
add_files src/krnl_idct.cpp
add_files -tb src/idct.cpp
open_solution "solution1"
set_part {xcu200-fsgd2104-2-e} -tool vivado
create_clock -period 10 -name default
config_sdx -optimization_level none -target xocc
config_schedule -effort medium -enable_dsp_full_reg
config_compile -name_max_length 256 -pipeline_loops 64
#source "./guiProj/solution1/directives.tcl"
csim_design
csynth_design
cosim_design
export_design -rtl verilog -format ip_catalog -xo \
    /wrk/bugs/xoFlow/idct_hls/krnl_idct.xo
```

# Incorporating Vivado HLS Kernel Projects into SDAccel

The Vivado HLS output is the kernel code exported as a Xilinx object file (`xo`). This file can be seamlessly integrated into SDAccel by selecting the object file as input. The SDAccel automatically extracts from the `xo` kernel name, so the host code can start applying the accelerator.

During SDAccel compilation, it is possible to create multiple compute units from the kernels, but the implementation remains the same as designed during the Vivado HLS run.

In SDAccel, the regular debug and analysis features are fully supported for this flow. It is possible to build the hardware emulation flow to test and debug in detail the implementation and tune the system build host code performance.

***Note:*** The pure software emulation mode is currently not supported as duplicated header file dependencies can create an issue.

# Known Limitations

This flow has certain limitations not present in top-down flow:

- No software emulation support for projects with `xo` files (potential missing and duplicated header files).
- GDB Kernel debug in hardware emulation flow is not supported.
- HLS analysis functionality is only available in the Vivado HLS project and not from SDAccel.

XILINX

*Appendix A*

# Getting Started with Examples

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

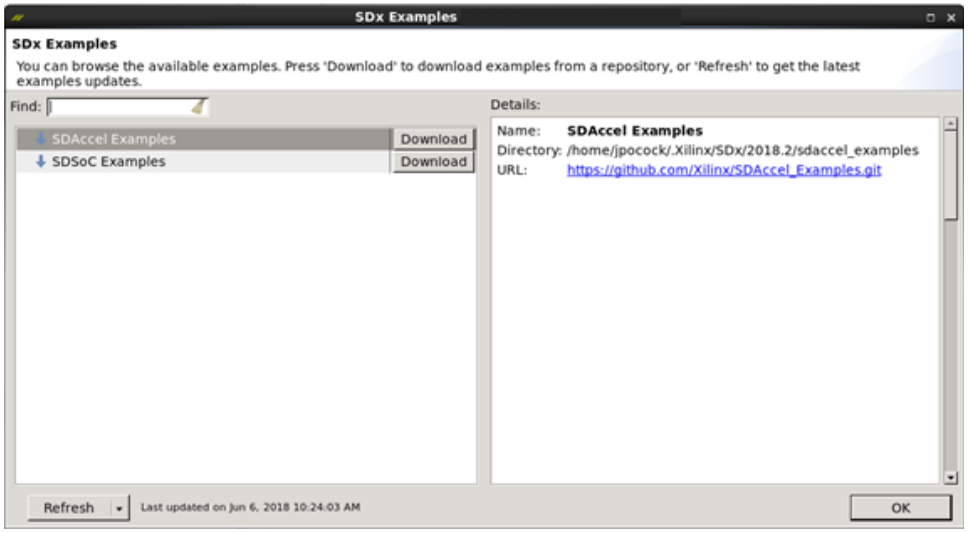
- Be a useful learning tool for both the SDx IDE and compilation flows such as makefile flows.
- Help you quickly get started in creating a new application project.
- Demonstrate useful coding styles.
- Highlight important optimization techniques.

Every platform provided within the SDx environment contains sample designs to get you started, and are accessible through the project creation flow 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

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**.

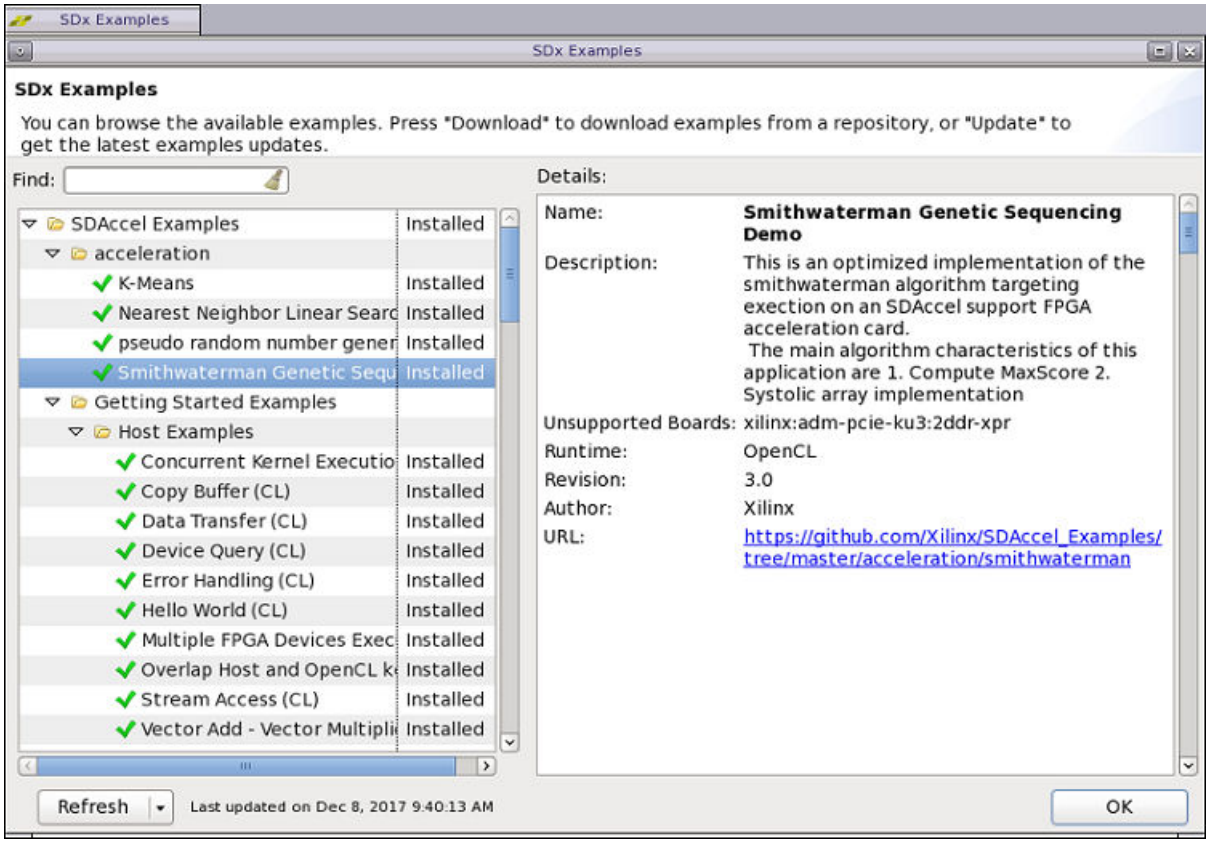
*Figure 61:* **SDAccel Examples – Empty**




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 62:* **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.

***Note*:** Corporate firewalls can restrict outbound connections. Specific proxy settings might be necessary.

# 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 Linux: `~/.Xilinx/SDx/<version>`

The download directory cannot be changed from the SDx Examples dialog box. You might want to download the example files to a different location from the `.Xilinx` folder. To perform this, use the `git` command from a command shell to specify a new destination folder for the downloaded examples:

```
git clone https://github.com/Xilinx/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. To make the files local, locate the files and manually make them local to your project.

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

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

XILINX

*Appendix B*

# Directory Structure

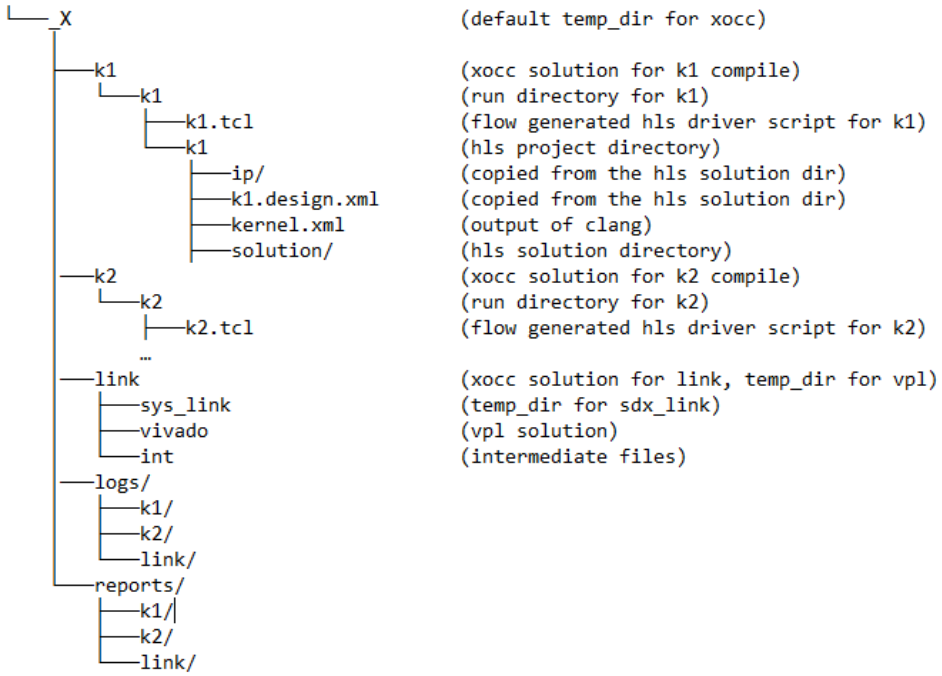
The directory structure generated by the GUI and make flows has been organized to allow you to easily find and access files. By navigating each `compile`, `link`, `logs`, and `reports` directories, you can easily reach generated files. Similarly, each kernel will also have a directory structure created.

## Command Line

When using `xocc` on the command line, by default it creates a 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 sub-directories. The `link`, `logs`, and `reports` directories contain the respective information on the builds.

*Figure 63:* **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)>
```

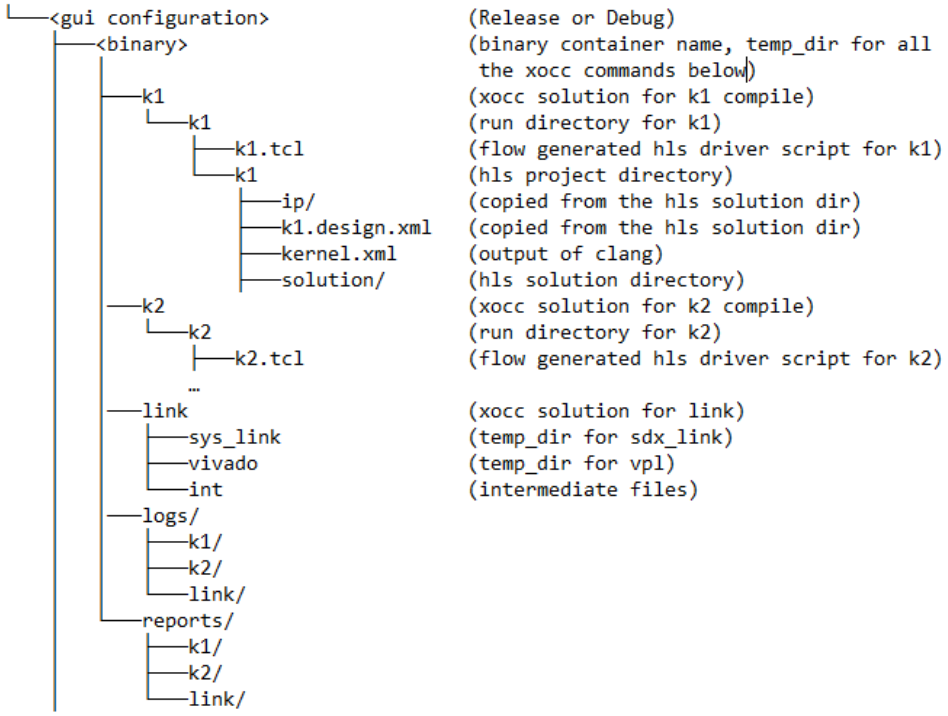
```
--temp_dir <dir_name (full or relative path)>
```

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 makefile 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 sub-directories. Again, the link, logs, and reports directories contain the respective information on the builds.

*Figure 64:* **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.

XILINX

*Appendix C*

# Useful Command Line Utilities

There are several Xilinx® command line utilities that provide detailed information to help construct `xocc` command line and give data about the platform, including SLR resource availability. These include `platforminfo`, `kernelinfo`, and `xclbininfo`.

## Platforminfo Utility

The `platforminfo` command line utility reports platform meta-data including information on interface, clock, valid SLRs and allocated resources, and memory in a structured format. This information can be referenced when allocating kernels to SLRs or memory resources for instance.

The following command will report information on the `xilinx_u200_xdma_201830_1.xpfm` platform.

```
platforminfo -p $PLATFORM_REPO_PATHS/xilinx_u200_xdma_201830_1.xpfm
```

Condensed output examples, using the above command, are given below.

For more information, including command options, see the *SDx Command and Utility Reference Guide* (UG1279).

### Basic Platform Information

Platform information and high-level description are reported. The `xdma` performs memory transfers between the host and Xilinx accelerator card over a PCIe®.

```
Platform:      xdma
File:          /opt/xilinx/platforms/xilinx_u200_xdma_201830_1/
               xilinx_u200_xdma_201830_1.xpfm
Description:   This platform targets the Alveo U200 Data Center Accelerator
               Card. This high-performance acceleration platform features
               up to four channels of DDR4-2400 SDRAM which are
               instantiated as required by the user kernels for high fabric
               resource availability, and Xilinx DMA Subsystem for PCI
               Express with PCIe Gen3 x16 connectivity.
Platform Type: SDAccel
```

## Hardware Platform Information

General information on the hardware platform is reported. For the Software Emulation and Hardware Emulation field, a "1" indicates this platform is suitable for these configurations. The **Maximum Number of Compute Units** field gives the maximum number of compute units allowable in this platform.

```
Vendor:                              xilinx
Board:                               U200 (xdma)
Name:                                xdma
Version:                             201830.1
Generated Version:                   2018.3
Software Emulation:                  1
Hardware Emulation:                  1
FPGA Family:                         virtexuplus
FPGA Device:                         xcu200
Board Vendor:                        xilinx.com
Board Name:                          xilinx.com:au200:1.0
Board Part:                          xcu200-fsgd2104-2-e
Maximum Number of Compute Units:     60
```

## Interface Information

The following shows the reported PCIe interface information.

```
  Interface Name: PCIe
  Interface Type: gen3x16
  PCIe Vendor Id:    0x10EE
  PCIe Device Id:    0x5000
  PCIe Subsystem Id: 0x000E
```

## Clock Information

Reports the maximum kernel clock frequencies available. The Clock Index is the reference used in the `--kernel_frequency xocc` directive when overriding the default value. For specific details, see *SDx Command and Utility Reference Guide* (UG1279).

```
  Default Clock Index: 0
  Clock Index:         1
    Frequency:         500.000000
  Clock Index:         0
    Frequency:         300.000000
```

## Valid SLRs

Reports the valid SLRs in the platform.

```
  SLR0, SLR1, SLR2
```

## Resource Availability

The total available resources and resources available per SLR are reported. This information can be used to assess applicability of the platform for the design and help guide allocation of compute unit to available SLRs.

```
Total...
  LUTs:  1051996
  FFs:   2197301
  BRAMs: 1896
  DSPs:  6833
Per SLR...
  SLR0:
    LUTs:  354825
    FFs:   723370
    BRAMs: 638
    DSPs:  2265
  SLR1:
    LUTs:  159108
    FFs:   329166
    BRAMs: 326
    DSPs:  1317
  SLR2:
    LUTs:  354966
    FFs:   723413
    BRAMs: 638
    DSPs:  2265
```

## Memory Information

Reports the available DDR and PLRAM memory connections per SLR as shown in the example output below.

```
Type: ddr4
Bus SP Tag: DDR
  Segment Index: 0
    Consumption: automatic
    SP Tag:      bank0
    SLR:         SLR0
    Max Masters: 15
  Segment Index: 1
    Consumption: default
    SP Tag:      bank1
    SLR:         SLR1
    Max Masters: 15
  Segment Index: 2
    Consumption: automatic
    SP Tag:      bank2
    SLR:         SLR1
    Max Masters: 15
  Segment Index: 3
    Consumption: automatic
    SP Tag:      bank3
    SLR:         SLR2
    Max Masters: 15
Bus SP Tag: PLRAM
  Segment Index: 0
    Consumption: explicit
```

```
      SLR:          SLR0
      Max Masters: 15
    Segment Index: 1
      Consumption: explicit
      SLR:          SLR1
      Max Masters: 15
    Segment Index: 2
      Consumption: explicit
      SLR:          SLR2
      Max Masters: 15
```

The `Bus SP Tag` heading can be DDR or PLRAM and gives associated information below.

The `Segment Index` field is used in association with the `SP Tag` to generate the associated memory resource index as shown below.

```
Bus SP Tag[Segment Index]
```

For example, if `Segment Index` is 0, then the associated DDR resource index would be DDR[0].

This memory index is used when specifying memory resources in the `xocc` command as shown below:

```
xocc … --sp vadd.m_axi_gmem:DDR[3]
```

There can be more than one `Segment Index` associated with an SLR. For instance, in the output above, SLR1 has both `Segment Index` 1 and 2.

The `Consumption` field indicates how a memory resource is used when building the design:

- **default:** If an `--sp` directive is not specified, it uses this memory resource by default during `xocc` build. For example in the report below, DDR with `Segment Index` 1 is used by default.
- **automatic:** When the maximum number of memory interfaces have been used under `Consumption: default` have been fully applied, then the interfaces under `automatic` is used. The maximum number of interfaces per memory resource are given in the **Max Masters** field.
- **explicit:** For PLRAM, consumption is set to `explicit` which indicates this memory resource is only used when explicitly indicated through the `xocc` command line.

## Feature ROM Information

The feature ROM information provides build related information on ROM platform and can be requested by Xilinx Support when debugging system issues.

```
  ROM Major Version:       10
  ROM Minor Version:       1
  ROM Vivado Build ID:     2388429
  ROM DDR Channel Count:   5
  ROM DDR Channel Size:    16
  ROM Feature Bit Map:     655885
  ROM UUID:                00194bb3-707b-49c4-911e-a66899000b6b
  ROM CDMA Base Address 0: 620756992
  ROM CDMA Base Address 1: 0
  ROM CDMA Base Address 2: 0
  ROM CDMA Base Address 3: 0
```

## Software Platform Information

Although software platform information is reported, it is only useful for SDSoC™ users, which have an OS running on the device, and not applicable to SDAccel™ users which use a host machine.

```
Number of Runtimes:            1
Linux root file system path:   tbd
Default System Configuration:  config0_0
System Configurations:
  System Config Name:                        config0_0
  System Config Description:                 config0_0 Linux OS on x86_0
  System Config Default Processor Group:     x86_0
  System Config Default Boot Image:
  System Config Is QEMU Supported:           0
  System Config Processor Groups:
    Processor Group Name:      x86_0
    Processor Group CPU Type:  x86
    Processor Group OS Name:   Linux OS
  System Config Boot Images:
Supported Runtimes:
  Runtime: OpenCL
```

# Kernelinfo Utility

The `kernelinfo` utility extracts information from `.xo` files which can be used during host code development. This information includes hardware function names, arguments, offsets, and port data.

To run the `kernelinfo` utility, enter the following in a Linux terminal:

```
kernelinfo <filename.xo>
```

The output is divided into three categories:

- Kernel Definitions
- Arguments
- Ports

Examples of each are shown below based on the following command:

```
kernelinfo krnl_vadd.xo
```

Where `krnl_vadd.xo` is a packaged kernel.

For more information, including command options, see *SDx Command and Utility Reference Guide* (UG1279).

## Kernel Definition

Reports high-level kernel definition information. Importantly, for the host code development, the kernel name is given in the `name` field. In this example, the kernel name is `krnl_vadd`.

```
=== Kernel Definition ===
name: krnl_vadd
language: c
vlnv: xilinx.com:hls:krnl_vadd:1.0
preferredWorkGroupSizeMultiple: 1
workGroupSize: 1
debug: true
containsDebugDir: 1
sourceFile: krnl_vadd/cpu_sources/krnl_vadd.cpp
```

## Arguments

Reports kernel function arguments. For each argument, it reports the following:

- `name`
- `addressQualifier`
- `id`
- `port`
- `size`
- `offset`
- `hostOffset`
- `hostSize`
- `type`

In the following example, there are four arguments:

- `a`
- `b`
- `c`
- `n_elements`

```
=== Arg ===
name: a
addressQualifier: 1
id: 0
port: M_AXI_GMEM
size: 0x8
offset: 0x10
hostOffset: 0x0
hostSize: 0x8
type: int*

=== Arg ===
name: b
addressQualifier: 1
id: 1
port: M_AXI_GMEM
size: 0x8
offset: 0x1C
hostOffset: 0x0
hostSize: 0x8
type: int*

=== Arg ===
name: c
addressQualifier: 1
id: 2
port: M_AXI_GMEM1
size: 0x8
offset: 0x28
hostOffset: 0x0
hostSize: 0x8
type: int*

=== Arg ===
name: n_elements
addressQualifier: 0
id: 3
port: S_AXI_CONTROL
size: 0x4
offset: 0x34
hostOffset: 0x0
hostSize: 0x4
type: int const
```

## Ports

Reports the memory and control ports used by the kernel.

```
=== Port ===
name: M_AXI_GMEM
mode: master
range: 0xFFFFFFFF
dataWidth: 32
portType: addressable
base: 0x0

=== Port ===
name: M_AXI_GMEM1
mode: master
range: 0xFFFFFFFF
dataWidth: 32
portType: addressable
base: 0x0

=== Port ===
name: S_AXI_CONTROL
mode: slave
range: 0x1000
dataWidth: 32
portType: addressable
base: 0x0
```

# Xclbinutil Utility

The `xclbinutil` utility can create, modify, and report `xclbin` content information. Detailed command options can be found in "xclbinutil Utility" section in *SDx Command and Utility Reference Guide* (UG1279). For most users, the contents and how the `xclbin` was created is desired. This information can be obtained through the `--info` option and reports information on the `xclbin`, hardware platform, clocks, memory configuration, kernel, and how the `xclbin` was generated.

An example of the `xclbinutil` command using the `--info` option is shown below. In addition, an example of the generated output, divided into sections, based on the command is also given.

```
xclbinutil -i binary_container_1.xclbin --info
```

## xclbin Information

```
Generated by:           xocc (2018.3) on Tue Nov 20 19:42:42 MST 2018
Version:                2.1.1660
Kernels:                krnl_vadd
Signature:              Not Present
Content:                HW Emulation Binary
```

```
UUID:                    979eb04c-b99c-4cbe-9a67-ad07b89f303b
Sections:                BITSTREAM, MEM_TOPOLOGY, IP_LAYOUT, CONNECTIVITY,
                         DEBUG_IP_LAYOUT, CLOCK_FREQ_TOPOLOGY,
BUILD_METADATA,
                         EMBEDDED_METADATA, DEBUG_DATA
```

## Hardware Platform (Shell) Information

```
Vendor:                  xilinx
Board:                   u200
Name:                    xdma
Version:                 201830.1
Generated Version:       Vivado 2018.3 (SW Build: 2388429)
Created:                 Wed Nov 14 20:06:10 2018
FPGA Device:             xcu200
Board Vendor:            xilinx.com
Board Name:              xilinx.com:au200:1.0
Board Part:              xilinx.com:au200:part0:1.0
Platform VBNV:           xilinx_u200_xdma_201830_1
Static UUID:             00194bb3-707b-49c4-911e-a66899000b6b
Feature ROM TimeStamp:   1542252769
```

## Clocks

Reports the maximum kernel clock frequencies available. Both the clock names and clock indexes are provided. The clock indexes are identical as reported in the Platforminfo Utility.

```
Name:       DATA_CLK
Index:      0
Type:       DATA
Frequency:  300 MHz

Name:       KERNEL_CLK
Index:      1
Type:       KERNEL
Frequency:  500 MHz
```

## Memory Configuration

```
Name:          bank0
Index:         0
Type:          MEM_DDR4
Base Address:  0x0
Address Size:  0x400000000
Bank Used:     No

Name:          bank1
Index:         1
Type:          MEM_DDR4
Base Address:  0x400000000
Address Size:  0x400000000
Bank Used:     Yes

Name:          bank2
Index:         2
Type:          MEM_DDR4
```

```
Base Address:  0x800000000
Address Size:  0x400000000
Bank Used:     No

Name:          bank3
Index:         3
Type:          MEM_DDR4
Base Address:  0xc00000000
Address Size:  0x400000000
Bank Used:     No

Name:          PLRAM[0]
Index:         4
Type:          MEM_DDR4
Base Address:  0x1000000000
Address Size:  0x20000
Bank Used:     No

Name:          PLRAM[1]
Index:         5
Type:          MEM_DRAM
Base Address:  0x1000020000
Address Size:  0x20000
Bank Used:     No

Name:          PLRAM[2]
Index:         6
Type:          MEM_DRAM
Base Address:  0x1000040000
Address Size:  0x20000
Bank Used:     No
```

# Kernel Information

For each kernel in the `xclbin` the function definition, ports, and instance information is reported.

Below is an example of the reported function definition.

```
Definition
----------
   Signature: krnl_vadd (int* a, int* b, int* c,
              int const  n_elements)
```

Below is an example of the reported ports.

```
Ports
-----
   Port:           M_AXI_GMEM
   Mode:           master
   Range (bytes):  0xFFFFFFFF
   Data Width:     32 bits
   Port Type:      addressable

   Port:           M_AXI_GMEM1
   Mode:           master
   Range (bytes):  0xFFFFFFFF
   Data Width:     32 bits
   Port Type:      addressable
```

```
   Port:           S_AXI_CONTROL
   Mode:           slave
   Range (bytes): 0x1000
   Data Width:     32 bits
   Port Type:      addressable
```

Below is an example of the reported instance(s) of the kernel.

```
Instance:        krnl_vadd_1
   Base Address: 0x0

   Argument:          a
   Register Offset:   0x10
   Port:              M_AXI_GMEM
   Memory:            bank1 (MEM_DDR4)

   Argument:          b
   Register Offset:   0x1C
   Port:              M_AXI_GMEM
   Memory:            bank1 (MEM_DDR4)

   Argument:          c
   Register Offset:   0x28
   Port:              M_AXI_GMEM1
   Memory:            bank1 (MEM_DDR4)

   Argument:          n_elements
   Register Offset:   0x34
   Port:              S_AXI_CONTROL
   Memory:            <not applicable>
```

# Tool Generation Information

The utility also reports the `xocc` command line used to generate the `xclbin`. The Command Line section gives the actual `xocc` command line used, while the Options section displays each option used in the command line, but in a more readable format with one option per line.

```
Generated By
------------
Command:       xocc
Version:       2018.3 - Tue Nov 20 19:42:42 MST 2018 (SW BUILD: 2394611)
Command Line:  xocc -t hw_emu --platform /opt/xilinx/platforms/
xilinx_u200_xdma_201830_1/xilinx_
               u200_xdma_201830_1.xpfm --save-temps -l --nk krnl_vadd:1 -g
               --messageDb binary_container_1.mdb
               --xp misc:solution_name=link --temp_dir binary_container_1
               --report_dir binary_container_1/reports --log_dir binary_
               container_1/logs --remote_ip_cache
               /wrk/tutorials/ip_cache -obinary_container_1.xclbin binary_
               container_1/krnl_vadd.xo
Options:       -t hw_emu
               --platform /opt/xilinx/platforms/xilinx_u200_xdma_201830_1/
xilinx_u200_xdma_201830_1.xpfm
               --save-temps
               -l
               --nk krnl_vadd:1
               -g
```

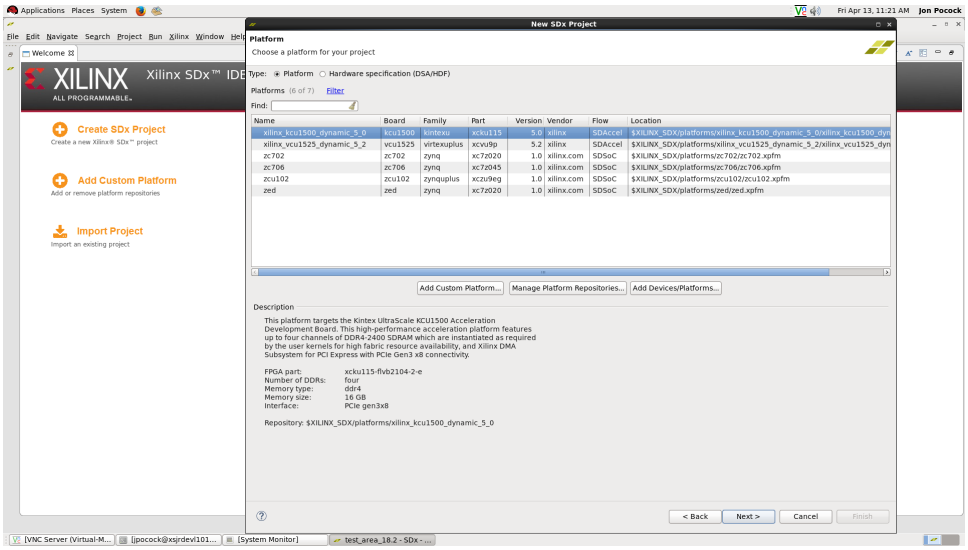
```
                --messageDb binary_container_1.mdb
                --xp misc:solution_name=link
                --temp_dir binary_container_1
                --report_dir binary_container_1/reports
                --log_dir binary_container_1/logs
                --remote_ip_cache /wrk/tutorials/ip_cache
                -obinary_container_1.xclbin binary_container_1/krnl_vadd.xo
================================================================================
==
User Added Key Value Pairs
---------------------------
   <empty>
================================================================================
==
```

# Managing Platforms and Repositories

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

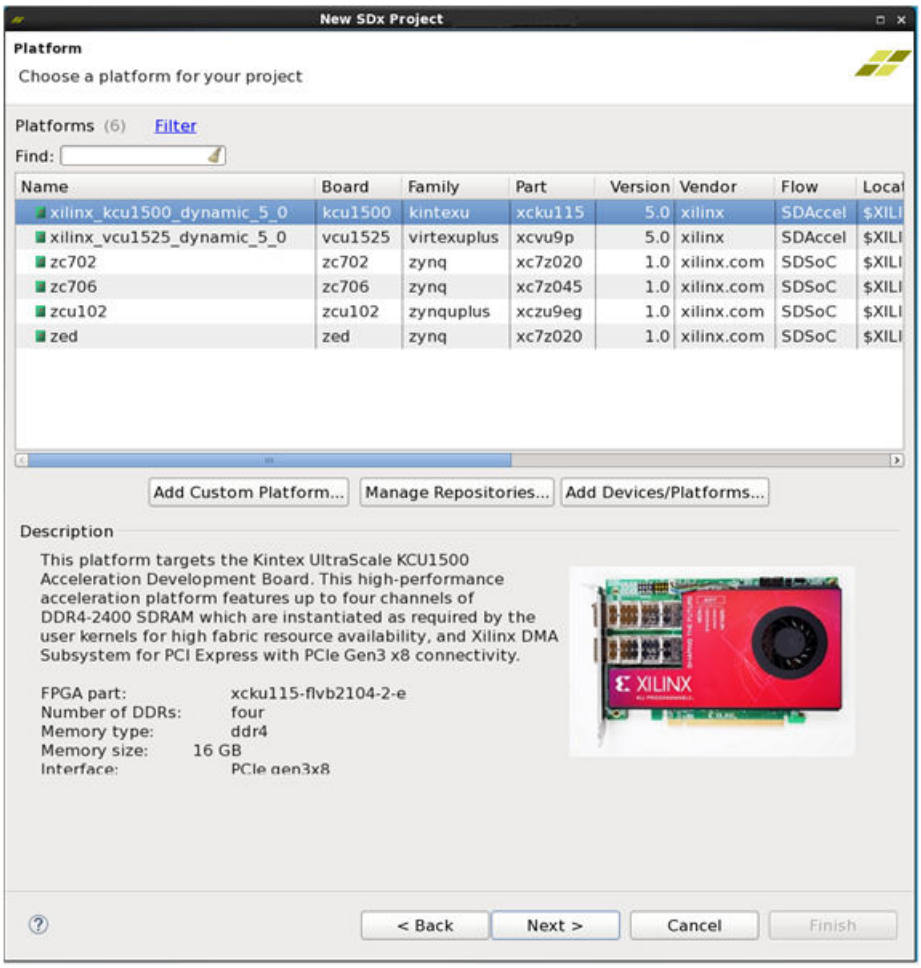
When you are creating a project, you can manage the platforms that are available for use in SDAccel™ application projects, from the Platform Selection page of the SDx New Project wizard. This lets you add a new platform for a project as it is being created.

*Figure 65:* **SDAccel Platform Browse**




This opens the New SDx Project dialog box, where you can manage the available platforms and platform repositories.

*Figure 66:* **Specify SDAccel Platform**




- **Add Custom Platform:** 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:** 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:** 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*

# Migrating to a New Target Platform

This migration guide is intended for users who need to migrate their accelerated SDAccel™ environment application from one target platform to another. For example, moving an application from a Virtex® UltraScale+™ VCU1525 Acceleration Development Board to a U200 Acceleration Development Board.

The following topics are addressed as part of this:

- An overview of the Design Migration Process including the physical aspects of FPGA devices.
- Any changes to the host code and design constraints if a new release is used.
- Controlling kernel placements and DDR interface connections.
- Timing issues in the new shell which might require additional options to achieve performance.

## Design Migration

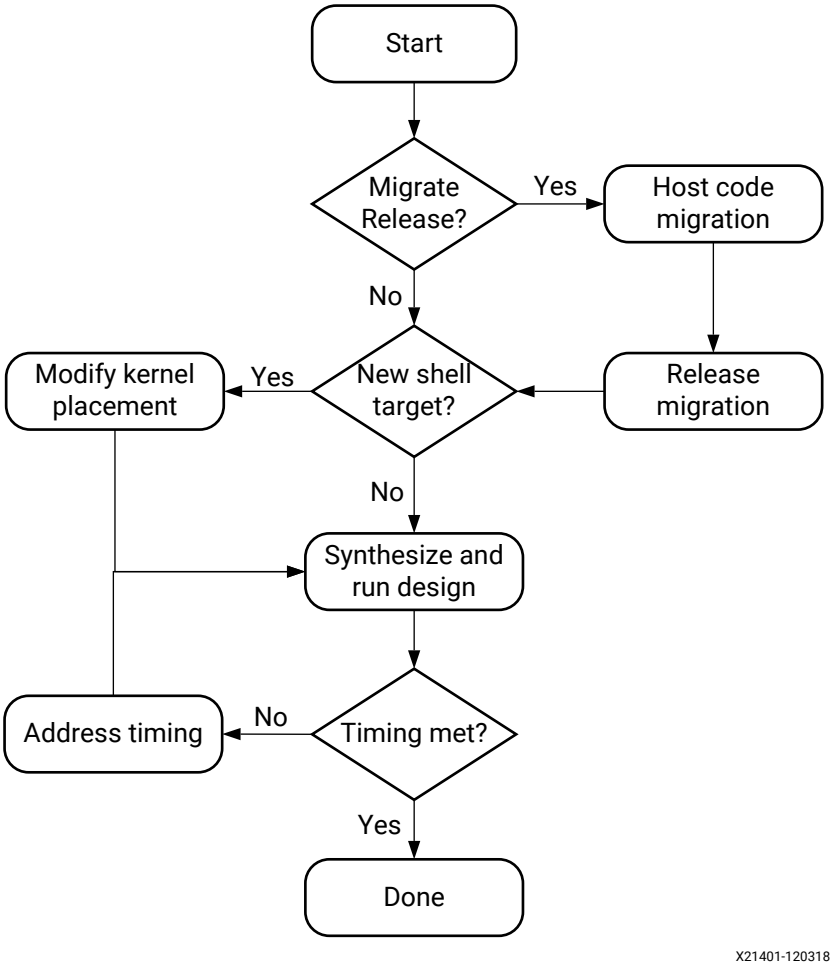
When migrating an application implemented in one target platform to another, it is important to understand the differences between the target platforms, and the impact those differences have on the design.

Key considerations:

- Is there a change in the release?
- Does the new target platform contain a different shell?
- Do the kernels need to be redistributed across the Super Logic Regions (SLRs)?
- Does the design meet the required frequency (timing) performance in the new platform?

The following diagram summarizes the migration flow described in this guide, and the topics to consider during the migration process.

*Figure 67:* **Shell Migration Flowchart**



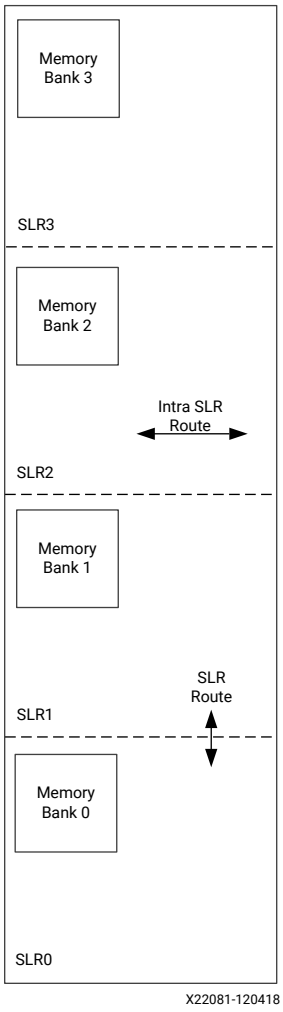

**IMPORTANT!** *Before starting to migrate a design it is important to understand the architecture of an FPGA and the shell.*

# Understanding an FPGA Architecture

Before migrating any design to a new target platform, you should have a fundamental understanding of the FPGA architecture. The following diagram shows the floorplan of a Xilinx® FPGA device. The concepts to understand are:

- SSI Devices
- SLRs
- SLR routing resources
- Memory interfaces

*Figure 68:* **Physical View of Xilinx FPGA with Four SLR Regions**




**TIP:** *The FPGA floorplan shown above is for a SSI device with four SLRs where each SLR contains a DDR Memory interface.*

### Stacked Silicon Interconnect Devices

A SSI device is one in which multiple silicon dies are connected together via silicon interconnect, and packaged into a single device. An SSI device enables high-bandwidth connectivity between multiple die by providing a much greater number of connections. It also imposes much lower latency and consumes dramatically lower power than either a multiple FPGA or a multi-chip module approach, while enabling the integration of massive quantities of interconnect logic, transceivers, and on-chip resources within a single package. The advantages of SSI devices are detailed in *Xilinx Stacked Silicon Interconnect Technology Delivers Breakthrough FPGA Capacity, Bandwidth, and Power Efficiency*.

### Super Logic Region

An SLR is a single FPGA die slice contained in an SSI device. Multiple SLR components are assembled to make up an SSI device. Each SLR contains the active circuitry common to most Xilinx FPGA devices. This circuitry includes large numbers of:

- LUTs
- Registers
- I/O Components
- Gigabit Transceivers
- Block Memory
- DSP Blocks

One or more kernels may be implemented within an SLR. A single kernel may not be implemented across multiple SLRs.

### SLR Routing Resources

The custom hardware implemented on the FPGA is connected via on-chip routing resources. There are two types of routing resources in an SSI device:

- **Intra-SLR Resources:** Intra-SLR routing resource are the fast resources used to connect the hardware logic. The SDAccel environment automatically uses the most optimal resources to connect the hardware elements when implementing kernels.
- **Super Long Line (SLL) Resources:** SLLs are routing resources running between SLRs, used to connect logic from one region to the next. These routing resources are slower than intra-SLR routes. However, when a kernel is placed in one SLR, and the DDR it connects to is in another, the SDAccel environment automatically implements dedicated hardware to use SLL routing resources without any impact to performance. More details on managing placement are provided in Modifying Kernel Placement.

### Memory Interfaces

Each SLR contains one or more memory interfaces. These memory interfaces are used to connect to the DDR memory where the data in the host buffers is copied before kernel execution. Each kernel will read data from the DDR memory and write the results back to the same DDR memory. The memory interface connects to the pins on the FPGA and includes the memory controller logic.

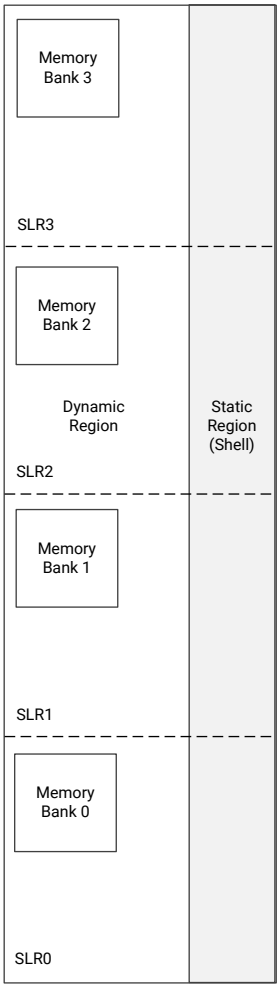
## Understanding Shells

In the SDAccel development environment, a shell is the hardware design that is implemented onto the FPGA before any custom logic, or accelerators are added. The shell defines the attributes of the FPGA used in the target platform and is composed of two regions:

- Static region which contains kernel and device management logic.
- Dynamic region where the custom logic of the accelerated kernels is placed.

The figure below shows an FPGA with the shell applied.

*Figure 69:* **Shell on an FPGA with Four SLR Regions**




The shell, which is a static region that cannot be modified by the user, contains the logic required to operate the FPGA, and transfer data to and from the dynamic region. The static region, shown above in gray, might exist within a single SLR, or as in the above example, might span multiple SLRs. The static region contains:

- DDR memory interface controllers
- PCIe® interface logic
- XDMA logic
- Firewall logic, etc.

The dynamic region is the area shown in white above. This region contains all the reconfigurable components of the shell and is the region where all the accelerator kernels are placed.

Because the static region consumes some of the hardware resources available on the device, the custom logic to be implemented in the dynamic region can only use the remaining resources. In the example shown above, the shell defines that all four DDR memory interfaces on the FPGA can be used. This will require resources for the memory controller used in the DDR interface.

Details on how much logic may be implemented in the dynamic region of each shell is provided in the *SDAccel Environments Release Notes, Installation, and Licensing Guide* (UG1238). This topic is also addressed in Modifying Kernel Placement, later in this guide.

# Migrating Releases

Before migrating to a new target platform, you should also determine if you will need to target the new platform to a different release of the SDAccel environment. If you do intend to target a new release, it is highly recommended to first target the existing platform using the new software release to confirm there are no changes required, and then migrate to a new target platform.

There are two steps to follow when targeting a new release with an existing platform:

- Host Code Migration
- Release Migration

**IMPORTANT!** *Before migrating to a new release, it is recommended that you review the SDAccel Environments Release Notes, Installation, and Licensing Guide (UG1238).*

## Host Code Migration

In the 2018.3 release of the SDAccel environment there are some fundamental changes to how the Xilinx runtime (XRT) environment and shell(s) are installed. In previous releases, both the XRT environment and shell(s) were automatically installed with the SDAccel environment. This has implications on the setup required to compile the host code.

Refer to the *SDAccel Environments Release Notes, Installation, and Licensing Guide* (UG1238) for details on the 2018.3 installation.

The `XILINX_XRT` environment variable is used to specify the location of the XRT environment and must be set before you compile the host code. When the XRT environment has been installed, the `XILINX_XRT` environment variable can be set by sourcing the `/opt/xilinx/xrt/setup.csh`, or `/opt/xilinx/xrt/setup.sh` file as appropriate. Secondly, ensure that your `LD_LIBRARY_PATH` variable also points to the XRT installation area.

To compile, and run the host code, make sure you source the `<SDX_INSTALL_DIR>/settings64.csh`, or `<SDX_INSTALL_DIR>/settings64.sh` file from the SDAccel installation.

If you are using the GUI, it will automatically incorporate the new XRT location and generate the `makefile` when you build your project.

However, if you are using your own custom `makefile`, you need to make the following changes:

- In your `makefile`, do not use the `XILINX_SDX` environment variable which was used in prior releases.
- The `XILINX_SDX` variables and paths must be updated to the `XILINX_XRT` environment variable:
  - Include directories are now specified as: `-I${XILINX_XRT}/include` and `-I${XILINX_XRT}/include/CL`
  - Library path is now: `-L${XILINX_XRT}/lib`
  - OpenCL™ library will be: `libxilinxopencl.so`. So, use `-lxilinxopencl` in your `makefile`

# Release Migration

After migrating the host code, build the code on the existing target platform using the new release of the SDAccel development environment. Verify that you can run the project in the SDAccel environment using the new release, and make sure it completes successfully, and meets the timing requirements.

Issues which can occur when using a new release are:

- Changes to C libraries or library files.
- Changes to kernel path names.
- Changes to the HLS pragmas or pragma options embedded in the kernel code.
- Changes to C/C++/OpenCL compiler support.
- Changes to the performance of kernels: this may require adjustments to the pragmas in the existing kernel code.

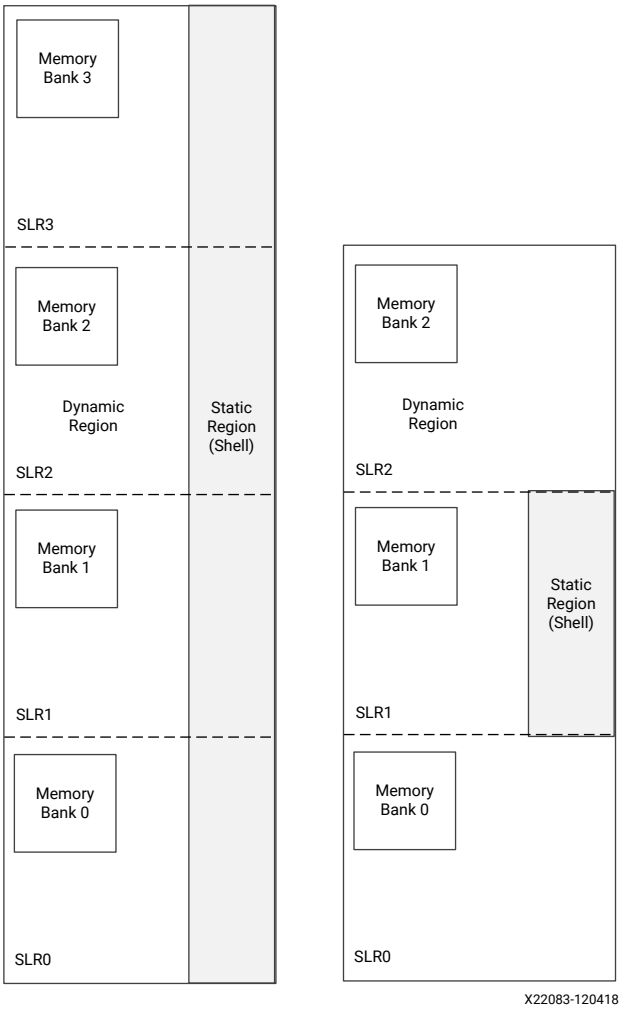
Address these issues using the same techniques you would use during the development of any kernel. At this stage, ensure the throughput performance of the target platform using the new release meets your requirements. If there are changes to the final timing (the maximum clock frequency), you can address these when you have moved to the new target platform. This is covered in Address Timing.

# Modifying Kernel Placement

The primary issue when targeting a new platform is ensuring that an existing kernel placement will work in the new target platform. Each target platform has an FPGA defined by a shell. As shown in the figure below, the shell(s) can be different.

- The shell of the original platform on the left has four SLRs, and the static region is spread across all four SLRs.
- The shell of the target platform on the right has only three SLRs, and the static region is fully-contained in SLR1.

*Figure 70:* **Comparison of Shells of the Hardware Platform**



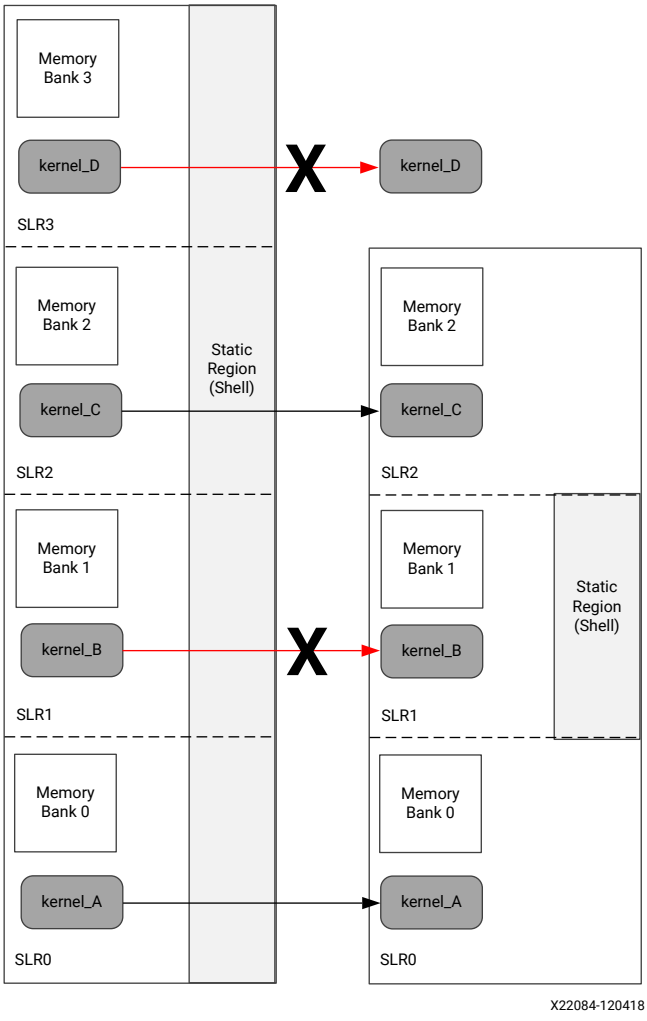

This section explains how to modify the placement of the kernels.

# Implications of a New Hardware Platform

The figure below highlights the issue of kernel placement when migrating to a new target platform, or shell. In the example below:

- Existing kernel, kernel_B, is too large to fit into SLR2 of the new target platform because most of the SLR is consumed by the static region.
- The existing kernel, kernel_D, must be relocated to a new SLR because the new target platform does not have four SLRs like the existing platform.

*Figure 71:* **Migrating Platforms – Kernel Placement**




When migrating to a new platform, you need to take the following actions:

- Understand the resources available in each SLR of the new target platform, as documented in the *SDAccel Environments Release Notes, Installation, and Licensing Guide* (UG1238).
- Understand the resources required by each kernel in the design.

- Use the `xocc` linker options (`--slr` and `--sp`) to specify which SLR each kernel is placed in, and which DDR bank each kernel connects to.

These items are addressed in the remainder of this section.

# Determining Where to Place the Kernels

To determine where to place kernels, two pieces of information are required:

- Resources available in each SLR of the shell of the hardware platform (`.dsa`).
- Resources required for each kernel.

With these two pieces of information you will then determine which kernel or kernels can be placed in each SLR of the shell.

Keep in mind when performing these calculation that 10% of the available resources can be used by system infrastructure:

- Infrastructure logic can be used to connect a kernel to a DDR interface if it has to cross an SLR boundary.
- In an FPGA, resources are also used for signal routing. It is never possible to use 100% of all available resources in an FPGA because signal routing also requires resources.

### Available SLR Resources

The resources available in each SLR provided by Xilinx can be found in the *SDAccel Environments Release Notes, Installation, and Licensing Guide* (UG1238). The figure below shows an example shell. In this example you can see:

- The SLR description indicates which SLR contains static and/or dynamic regions.
- The resources available in each SLR (LUTs, Registers, RAM, etc.) are listed.

This allows you to determine what resources are available in each SLR.

*Table 17:* **SLR Resources of a Hardware Platform**

| Area | SLR 0 | SLR 1 | SLR 2 |
|---|---|---|---|
| SLR description | Bottom of device; dedicated to dynamic region. | Middle of device; shared by dynamic and static region resources. | Top of device; dedicated to dynamic region. |
| Dynamic region pblock name | pfa_top_i_dynamic_region_pblock_dynamic_SLR0 | pfa_top_i_dynamic_region_pblock_dynamic_SLR1 | pfa_top_i_dynamic_region_pblock_dynamic_SLR2 |
| Compute unit placement syntax | set_property CONFIG.SLR_ASSIGNMENTS SLR0[get_bd_cells<cu_name>] | set_property CONFIG.SLR_ASSIGNMENTS SLR1[get_bd_cells<cu_name>] | set_property CONFIG.SLR_ASSIGNMENTS SLR2[get_bd_cells<cu_name>] |

*Table 17:* **SLR Resources of a Hardware Platform** *(cont'd)*

| Area | SLR 0 | SLR 1 | SLR 2 |
|---|---|---|---|
| **Global memory resources available in dynamic region** | | | |
| Memory channels; system port name | bank0 (16 GB DDR4) | bank1 (16 GB DDR4, in static region)<br>bank2 (16 GB DDR4, in dynamic region) | bank3 (16 GB DDR4) |
| **Approximate available fabric resources in dynamic region** | | | |
| CLB LUT | 388K | 199K | 388K |
| CLB Register | 776K | 399K | 776K |
| Block RAM Tile | 720 | 420 | 720 |
| UltraRAM | 320 | 160 | 320 |
| DSP | 2280 | 1320 | 2280 |

### Kernel Resources

The resources for each kernel can be obtained from the **System Estimate** report.

The **System Estimate** report is available in the **Assistant** view after either the Hardware Emulation or System run are complete. An example of this report is shown below.

*Figure 72:* **System Estimate Report**

```
Area Information
Compute Unit     Kernel Name    Module Name    FF    LUT   DSP  BRAM
---------------  -------------  -------------  ----  ----  ---  ----
smithwaterman_1  smithwaterman  smithwaterman  2925  4304  1    10
-------------------------------------------------------------------------
```

- FF refers to the CLB Registers noted in the platform resources for each SLR.
- LUT refers to the CLB LUTs noted in the platform resources for each SLR.
- DSP refers to the DSPs noted in the platform resources for each SLR.
- BRAM refers to the block RAM Tile noted in the platform resources for each SLR.

This information can help you determine the proper SLR assignments for each kernel.

## Assigning Kernels to SLRs

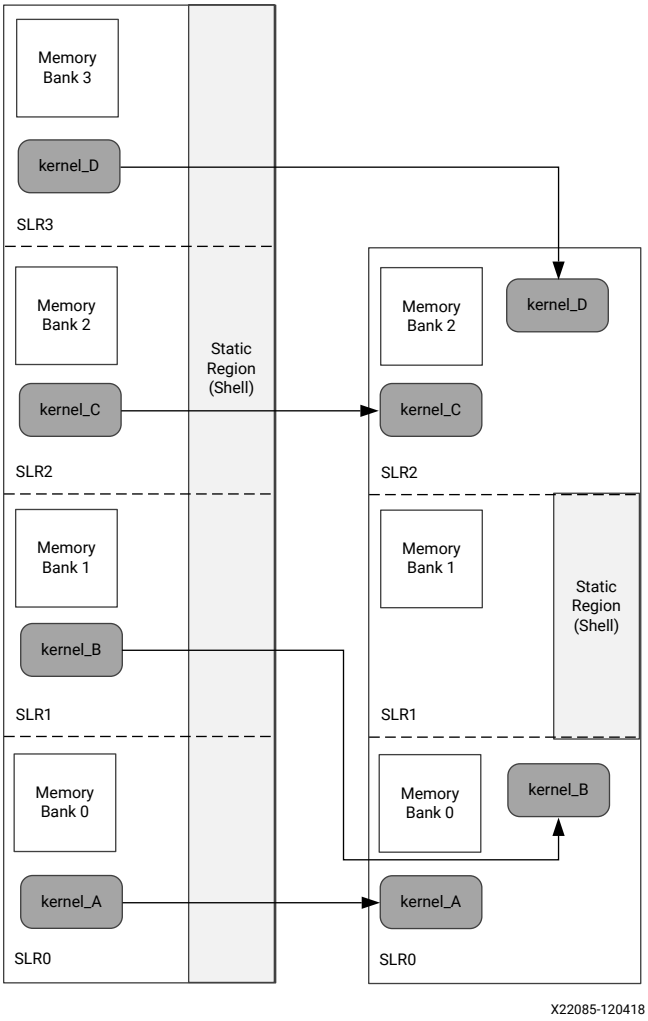
Each kernel in a design can be assigned to a SLR region using the `xocc --slr` command line option to specify a placement file. When placing kernels, it is recommended to also assign the specific DDR memory bank that the kernel will connect to using the `xocc --sp` command line option. An example can be used to demonstrate these two command line options.

The figure below shows an example where the existing target platform shell has four SLRs, and the new target platform has a shell with three SLRs, and the static region is also structured differently between the target platforms. In this migration example:

- Kernel_A is mapped to SLR0.
- Kernel_B, which no longer fits in SLR1, is remapped to SLR0, where there are available resources.
- Kernel_C is mapped to SLR2.
- Kernel_D, is remapped to SLR2, where there are available resources.

The kernel mappings are illustrated in the figure below.

*Figure 73:* **Mapping of Kernels Across SLRs**



### Specifying Kernel Placement

For the above example, the kernels are placed using the following `xocc` command option.

```
xocc --slr kernel_A:SLR0 \
  --slr kernel_B:SLR0 \
  --slr kernel_C:SLR2 \
  --slr kernel_D:SLR2
```

With these command line options, each of the kernels is placed as shown in the figure above.

### Specifying Kernel DDR Interfaces

You should also specify the kernel DDR memory interface when specifying kernel placements. Specifying the DDR interface ensures the automatic pipelining of kernel connections to a DDR interface in a different SLR. This ensures there is no degradation in timing which can reduce the maximum clock frequency.

In this example, using the kernel placements in the above figure:

- Kernel_A is connected to Memory Bank 0.
- Kernel_B is connected to Memory Bank 1.
- Kernel_C is connected to Memory Bank 2.
- Kernel_D is connected to Memory Bank 1.

The following `xocc` command line performs these connections:

```
xocc --sp kernel_A.arg1:bank0 \
  --sp kernel_B.arg1:bank1 \
  --sp kernel_C.arg1:bank2 \
  --sp kernel_D.arg1:bank1
```

**IMPORTANT!** *When using the `--sp` option to assign kernel ports to memory banks, you must specify the `--sp` option for all interfaces/ports of the kernel. Refer to "Customization of DDR Bank to Kernel Connection" in the SDAccel Environment Programmers Guide (UG1277) for more information.*

# Address Timing

Perform a system run and if it completes with no violations, then the migration is successful.

If timing has not been met you may need to specify some custom constraints to help meet timing. Refer to *UltraFast Design Methodology Guide for the Vivado Design Suite* (UG949) for more information on meeting timing.

## Custom Constraints

Custom constraints are passed to the Vivado® tools using the `xocc -xp` option for custom placement and timing constraints. Custom Tcl constraints for floorplanning of the kernels will need to be reviewed in the context of the new target platform (`.dsa`). For example, if a kernel was moved to a different SLR in the new shell, the corresponding placement constraints for that kernel will also need to be modified.

In general, timing is expected to be comparable between different target platforms that are based on the 9P Virtex UltraScale device. Any custom Tcl constraints for timing closure will need to be evaluated and might need to be modified for the new platform.

Additionally, any non-default options that are passed to `xocc` or to the Vivado tools using the `xocc --xp` switch will need to be updated for the new shell.

## Timing Closure Considerations

Design performance and timing closure can vary when moving across SDx™ releases or shell(s), especially when one of the following conditions is true:

- Floorplan constraints were needed to close timing.
- Device or SLR resource utilization was higher than the typical guideline:
  - LUT utilization was higher than 70%
  - DSP, RAMB, and UltraRAM utilization was higher than 80%
  - FD utilization was higher than 50%
- High effort compilation strategies were needed to close timing.

The utilization guidelines provide a threshold above which the compilation of the design can take longer, or performance can be lower than initially estimated. For larger designs which usually require using more than one SLR, specify the kernel/DDR association on the `xocc` command line while verifying that any floorplan constraint ensures the following:

- The utilization of each SLR is below the recommended guidelines.
- The utilization is balanced across SLRs if one type of hardware resource needs to be higher than the guideline.

For designs with overall high utilization, increasing the amount of pipelining in the kernels, at the cost of higher latency, can greatly help timing closure and achieving higher performance.

For quickly reviewing all aspects listed above, use the fail-fast reports generated throughout the SDx flow when using one of the following two options:

- `xocc -R 1`

    - `report_failfast` is run at the end of each kernel synthesis step
    - `report_fafailst` is run after `opt_design` on the entire design
    - `opt_design` DCP is saved
- `xocc -R 2`
    - Same reports as with `-R 1`, plus:
    - `report_failfast` is post-placement for each SLR
    - Additional reports and intermediate DCPs are generated

All reports and DCPs can be found in the implementation directory, including kernel synthesis reports:

```
<runDir>/_x/link/vivado/prj/prj.runs/impl_1
```

For more information about timing closure and the fail-fast report, see the *UltraFast Design Methodology Timing Closure Quick Reference Guide* (UG1292).

XILINX

*Appendix F*

# 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 through XVC-over-PCIe. The *JTAG Fallback* feature can be enabled without having to change the XVC-over-PCIe-based debug network in the platform design.

On the host side, when the Vivado® user connects through `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.

XILINX

*Appendix G*

# Additional Resources and Legal Notices

## Xilinx Resources

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

## Documentation Navigator and Design Hubs

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

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

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

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

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

# References

1. *SDAccel Environments Release Notes, Installation, and Licensing Guide* (UG1238)
2. *SDAccel Environment User Guide* (UG1023)
3. *SDAccel Environment Profiling and Optimization Guide* (UG1207)
4. *SDAccel Environment 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 User Guide: 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

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**