

Vitis Unified Software Platform Documentation

Application Acceleration Development

UG1393 (v2019.2) October 1, 2019





Revision History

The following table shows the revision history for this document.

Section	Revision Summary
10/01/2019 Version 2019.2	
Initial Xilinx release for the Xilinx Developer Forum.	N/A

Table of Contents

Revision History	2
Chapter 1: Introduction to the Vitis Unified Software Platform	5
Vitis 2019.2 Software Platform Release Notes.....	5
Introduction to the Vitis Environment for Acceleration.....	8
Vitis Application Methodology.....	15
Chapter 2: Developing Applications	44
Programming Model.....	44
Host Application.....	45
Kernel Requirements.....	62
C/C++ Kernels.....	64
RTL Kernels.....	84
Streaming Connections.....	114
OpenCL Kernels.....	122
Best Practices for Acceleration with Vitis.....	123
Chapter 3: Building and Running the Application	125
Setting up the Vitis Integrated Design Environment.....	125
Build Targets.....	126
Building the Host Program.....	128
Building the FPGA Binary.....	130
Directory Structure.....	143
Running an Application.....	146
Chapter 4: Profiling, Optimizing, and Debugging the Application	148
Profiling the Application.....	148
Optimizing the Performance.....	177
Debugging Applications and Kernels.....	225
Chapter 5: Vitis Environment Reference Materials	273
Vitis Compiler Command.....	274
xrt.ini File.....	304

Platforminfo.....	307
Kernelinfo.....	313
Xclbinutil Utility.....	315
Emconfigutil.....	320
Xilinx Board Utility.....	322
package_xo Command.....	339
HLS Pragmas.....	341
OpenCL Attributes.....	386
Chapter 6: Using the Vitis IDE.....	409
Vitis Command Options.....	409
Creating a Vitis IDE Project.....	410
Building the System.....	423
Vitis IDE Debug Flow.....	426
Configuring the Vitis IDE.....	428
Project Export and Import.....	439
Getting Started with Examples.....	441
Chapter 7: Using the Vitis Analyzer.....	444
Working with Reports.....	445
Creating a Run Configuration.....	447
Configuring the Vitis Analyzer.....	449
Chapter 8: Migrating to a New Target Platform.....	451
Design Migration.....	451
Migrating Releases.....	456
Modifying Kernel Placement.....	457
Address Timing.....	463
Appendix A: Additional Resources and Legal Notices.....	466
Xilinx Resources.....	466
Documentation Navigator and Design Hubs.....	466
Please Read: Important Legal Notices.....	467

Introduction to the Vitis Unified Software Platform

The Vitis™ unified software platform is a new tool that combines all aspects of Xilinx® software development into one unified environment.

Vitis 2019.2 Software Platform Release Notes

This section contains information regarding the features and updates of the Vitis software platform in this release.

What's in the Vitis Software Platform

Hardware-Accelerated Application Development Flow

For FPGA-based acceleration, the Vitis™ core development kit lets you build a software application using the OpenCL™ API to run hardware (HW) kernels on accelerator cards, like the Xilinx® Alveo™ Data Center acceleration cards. The Vitis core development kit also supports running the software application on an embedded processor platform running Linux, such as on Zynq UltraScale+ MPSoC devices. For the embedded processor platform, the Vitis core development kit execution model also uses the OpenCL API and the Linux-based Xilinx Runtime (XRT) to schedule the HW kernels and control data movement.

The Vitis core development kit tools support the Alveo U200, U250, and U280 Data Center accelerator cards, as well as the zcu102_base and zcu104_base embedded processor platforms. In addition to these off-the-shelf platforms, custom platforms are also supported.

The Vitis software platform supports application development for both data center and embedded processor platforms, allowing you to migrate data center applications to embedded platforms. The Vitis core development kit includes the `v++` compiler for the hardware kernel on all platforms, the `g++` compiler for compiling the application to run on an x86 host, and Arm® compiler for cross-compiling the application to run on the embedded processor of a Xilinx device.

Changed Behavior

Migrating from SDAccel

The following table specifies differences between the SDAccel development environment and the Vitis software platform.

Table 1: Migration Summary

Area	SDAccel Behavior	Vitis Behavior
Compilation and Linking	xocc	v++.
Profiling	N/A	For the command line flow, you can use the Vitis analyzer to view the reports generated during the build process.

Supported Platforms

Data Center Accelerator Cards

There are several accelerator cards available for data center use. These must be installed with Xilinx Runtime (XRT), and with individual deployment and development shells for each card. The shell provides the firmware and programming for the accelerator card running in a specific configuration. Shells available for use with the 2019.1 release are listed below.

Note: Only the following target platforms are supported. Any device or platform that is not listed is *not* supported.

Alveo Card	Shell	Version ¹	Notes	2018.3 Tools+XRT	2019.1 Tools+XRT
U200	XDMA	xilinx_u200_xdma_201820_1	-	Superseded	-
		xilinx_u200_xdma_201830_1	SLR Assignments/PLRAM	Production	Production
		xilinx_u200_xdma_201830_2	Bug Fixes/2019.1 Features (64b BAR, DRM)	Beta	Production
	QDMA	xilinx_u200_qdma_201830_1	QDMA (Stream+MM)	Beta	Superseded
		xilinx_u200_qdma_201910_1	QDMA (Stream+MM)	-	Beta
U250	XDMA	xilinx_u250_xdma_201820_1	-	Superseded	-
		xilinx_u250_xdma_201830_1	Profile + Debug	Production	Production
		xilinx_u250_xdma_201830_2	Bug Fixes/2019.1 Features (64b BAR, DRM)	Beta	Production
	QDMA	xilinx_u250_qdma_201910_1	QDMA (Stream+MM)	-	Beta
U280-ES1	XDMA	xilinx_u280-es1_xdma_201830_1	-	Beta	Superseded
		xilinx_u280-es1_xdma_201910_1	Bug Fixes/2019.1 Features (HBM ECC & Temp Mon)	-	Beta

Alveo Card	Shell	Version ¹	Notes	2018.3 Tools+XRT	2019.1 Tools+XRT
U280	XDMA	xilinx_u280_xdma_201910_1	-	-	Production
	QDMA	xilinx_u280_qdma_201910_1	QDMA (Stream+MM)	-	Beta

Notes:

1. Xilinx Runtime and Vitis core development kit releases must be aligned. Older shells can be used with newer tools, but kernels must be recompiled.



TIP: After installation, you can use the `platforminfo` command line utility, which reports platform meta-data including information on interface, clock, valid super logic regions (SLRs) and allocated resources, and memory in a structured format.

Embedded Platforms



IMPORTANT! Artix[®]-7, Kintex[®]-7, Virtex[®]-7, along with baremetal and RTOS platforms, are not supported for acceleration. For non-accelerated based designs, see *Vitis Embedded Software Development Flow: Migration (UG1356)*.

Embedded platforms available for use with the Vitis core development kit as part of the early access program are listed below.

The following platforms are included:

- **zcu102_base:** Based on the ZCU102 Zynq UltraScale+ MPSoC, with XRT and xfOpenCV libraries.
- **zcu104_base:** Based on the ZCU104 Zynq UltraScale+ MPSoC, with XRT and xfOpenCV libraries.

Known Issues

Known Issues for the Vitis software platform are available in [AR#72773](#).

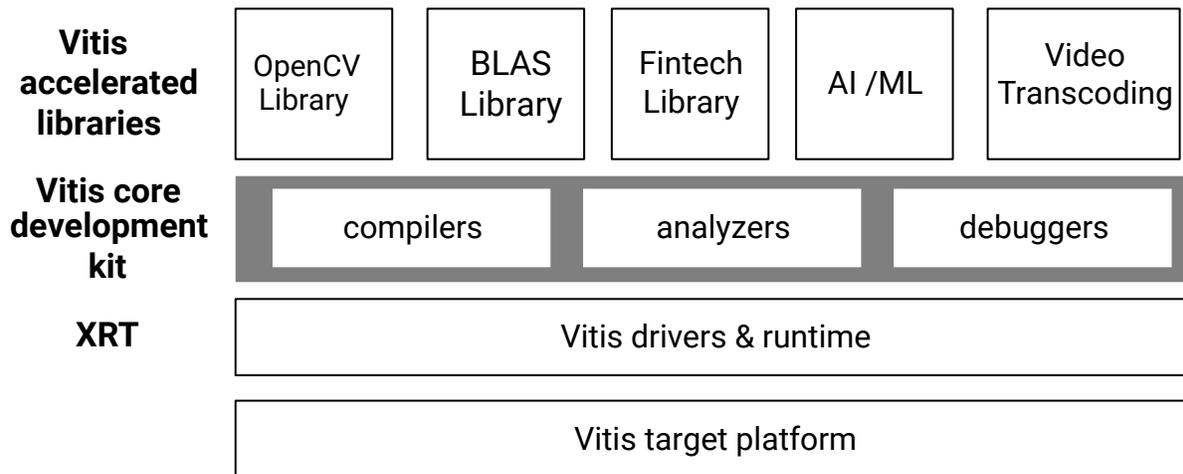
Introduction to the Vitis Environment for Acceleration

Introduction and Overview

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

The Vitis application acceleration development flow provides a framework for developing and delivering FPGA accelerated applications using standard programming languages for both software and hardware components. The software component, or host program, is developed using C/C++ to run on x86 or embedded processors, with OpenCL™ API calls to manage runtime interactions with the accelerator. The hardware component, or kernel, can be developed using C/C++, OpenCL, or RTL. The Vitis software platform accommodates various methodologies, letting you start by developing either the application or the kernel.

Figure 1: Vitis Unified Software Platform



X23292-092619

As shown in the figure above, the Vitis unified software platform consists of the following features and elements:

- Vitis technology targets acceleration hardware platforms, such as the Alveo Data Center accelerator cards, or Zynq UltraScale+ MPSoC and Zynq-7000 based embedded processor platforms.
- Xilinx Runtime (XRT) provides an API and drivers for your host program to connect with the target platform, and handles transactions between your host program and accelerated kernels.
- Vitis core development kit provides the software development tool stack, such as compilers and cross-compilers to build your host program and kernel code, analyzers to let you profile and analyze the performance of your application, and debuggers to help you locate and fix any problems in your application.
- Vitis accelerated libraries provide performance optimized FPGA acceleration with minimal code changes, and without the need to reimplement your algorithms to harness the benefits of Xilinx adaptive computing. Vitis accelerated libraries are available for common functions of math, statistics, linear algebra and DSP, and also for domain specific applications, like vision and image processing, quantitative finance, database, data analytics, and data compression. For more information on Vitis accelerated libraries, refer to https://gitenterprise.xilinx.com/FaaSApps/Vitis_Libraries.

FPGA Acceleration

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

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

For more information on how to architect an application for optimal performance and other recommended design techniques, review the [Vitis Application Methodology](#).

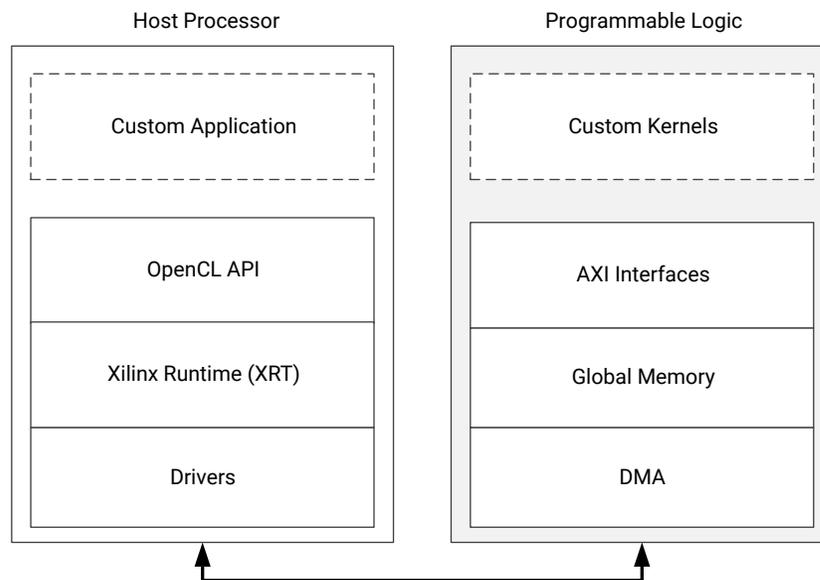
Execution Model

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

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

For instance, in a typical application, the host first transfers data to be operated on by the kernel from host memory into global memory. The kernel subsequently operates on the data, storing results back to the global memory. Upon kernel completion, the host transfers the results back into the host memory. Data transfers between the host and global memory introduce latency, which can be costly to the overall application. To achieve acceleration in a real system, the benefits achieved by the hardware acceleration kernels must outweigh the added latency of the data transfers.

Figure 2: Architecture of a Vitis Core Development Kit Application



X21835-091219

The target platform contains the FPGA accelerated kernels, global memory, and the direct memory access (DMA) for memory transfers. Kernels can have one or more global memory interfaces and are programmable. The Vitis core development kit execution model can be broken down into the following steps:

1. The host program writes the data needed by a kernel into the global memory of the attached device through the PCIe interface on an Alveo Data Center acceleration card, or through the AXI bus on an embedded platform.
2. The host program sets up the kernel with its input parameters.

3. The host program triggers the execution of the kernel function on the FPGA.
4. The kernel performs the required computation while reading data from global memory, as necessary.
5. The kernel writes data back to global memory and notifies the host that it has completed its task.
6. The host program reads data back from global memory into the host memory and continues processing as needed.

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

Build Process



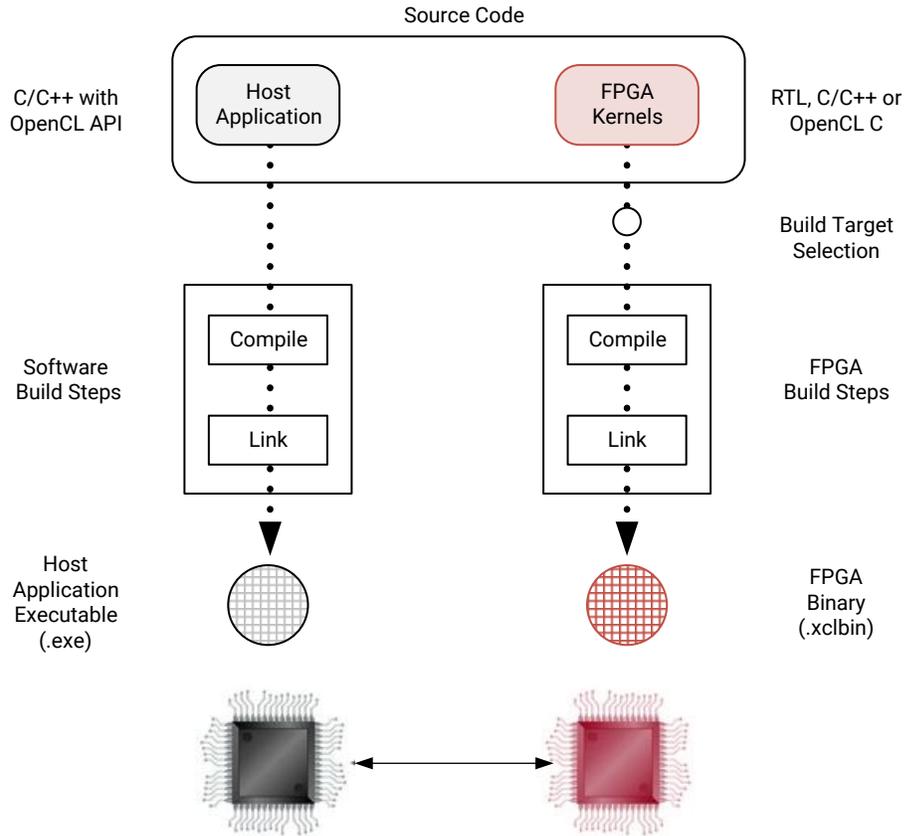
TIP: For a brief tutorial on the actual build process, review the [Vitis Pathway Tutorial](#) on GitHub.

The Vitis core development kit offers all of the features of a standard software development environment:

- Compiler or cross-compiler for host applications running on x86 or Arm® processors.
- Cross-compilers for building the FPGA binary.
- Debugging environment to help identify and resolve issues in the code.
- Performance profilers to identify bottlenecks and help you optimize the application.

The build process follows a standard compilation and linking process for both the host program and the kernel code. As shown in the following figure, the host program is built using the GNU C++ compiler (`g++`) or the GNU C++ Arm cross-compiler for MPSoC-based devices. The FPGA binary is built using the Vitis compiler.

Figure 3: Software/Hardware Build Process



X21399-091019

Host Program Build Process

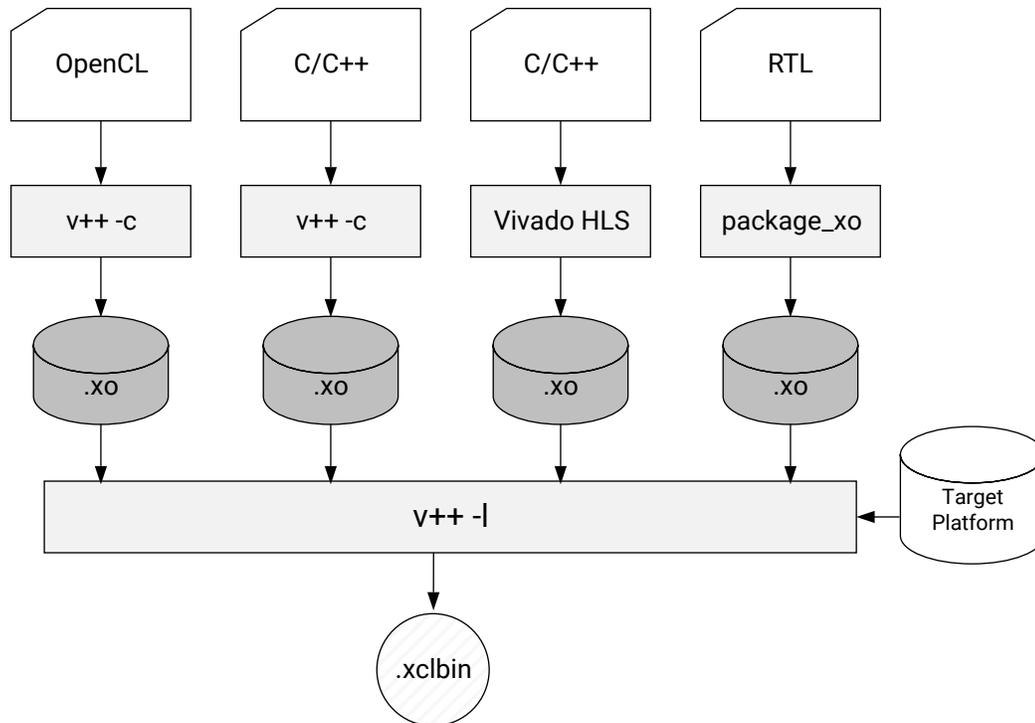
The main application is compiled and linked with the `g++` compiler, using the following two step process:

1. Compile any required code into object files (.o).
2. Link the object files (.o) with the XRT shared library to create the executable.

For details on this topic, refer to [Building the Host Program](#).

FPGA Binary Build Process

Figure 4: FPGA Build Process



X21155-091219

Kernels can be described in C/C++, or OpenCL C code, or can be created from packaged RTL designs. As shown in the figure above, each hardware kernel is independently compiled to a Xilinx object (.xo) file.

Xilinx object (.xo) files are linked with the hardware platform to create an FPGA binary file (.xclbin) that is loaded into the Xilinx device on the target platform.

The key to building the FPGA binary is to determine the build target you are producing. For more information, refer to [Build Targets](#).

For a detailed explanation of the build process, refer to [Building the FPGA Binary](#).

Build Targets

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

- When the build target is software or hardware emulation, the Vitis compiler generates simulation models of the kernels in the FPGA binary. These emulation targets let you build, run, and iterate the design over relatively quick cycles; debugging the application and evaluating performance.
- When the build target is the hardware system, Vitis compiler generates the .xclbin for the hardware accelerator, using the Vivado Design Suite to run synthesis and implementation. It uses these tools with predefined settings proven to provide good quality of results. Using the Vitis core development kit does not require knowledge of these tools; however, hardware-savvy developers can fully leverage these tools and use all the available features to implement kernels.

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

- **Software Emulation** (`sw_emu`): Both the host application code and the kernel code are compiled to run on the host processor. This allows iterative algorithm refinement through fast build-and-run loops. This target is useful for identifying syntax errors, performing source-level debugging of the kernel code running together with application, and verifying the behavior of the system.
- **Hardware Emulation** (`hw_emu`): The kernel code is compiled into a hardware model (RTL), which is run in a dedicated simulator. This build-and-run loop takes longer but provides a detailed, cycle-accurate view of kernel activity. This target is useful for testing the functionality of the logic that will go in the FPGA and getting initial performance estimates.
- **System** (`hw`): The kernel code is compiled into a hardware model (RTL) and then implemented on the FPGA, resulting in a binary that will run on the actual FPGA.

Tutorials and Examples

To help you quickly get started with the Vitis core development kit, you can find tutorials, example applications, and hardware kernels in the following repositories on <http://www.github.com/Xilinx>.

- **Vitis Tutorials:** Provides a number of tutorials that can be worked through to teach specific concepts regarding the tool flow and application development.

The [Vitis Pathway](#) tutorial is an excellent place to start as a new user.

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

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

Vitis Application Methodology

Introduction

This guide is intended for software developers and RTL designers who want to create device-accelerated applications using the Vitis unified software platform. It introduces developers to the fundamental concepts of device-based acceleration and provides steps for accelerating applications with the best possible performance.

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

Acceleration: An Industrial Analogy

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

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

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

A GPU also has workshops and workers, but it has considerably more of them, and the workers are much more specialized. They have access to only specific tools and can do fewer things, but they do them very efficiently. GPU workers function best when they do the same few tasks repeatedly, and when all of them are doing the same thing at the same time. After all, with so many different workers, it is more efficient to give them all the same orders.

Programmable devices take this workshop analogy into the industrial age. If CPUs and GPUs are groups of individual workers taking sequential steps to transform inputs into outputs, programmable devices are factories with assembly lines and conveyer belts. Raw materials are progressively transformed into finished goods by groups of workers dispatched along assembly lines. Each worker performs the same task repeatedly and the partially finished product is transferred from worker to worker on the conveyer belt. This results in a much higher production throughput.

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

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

Methodology Overview

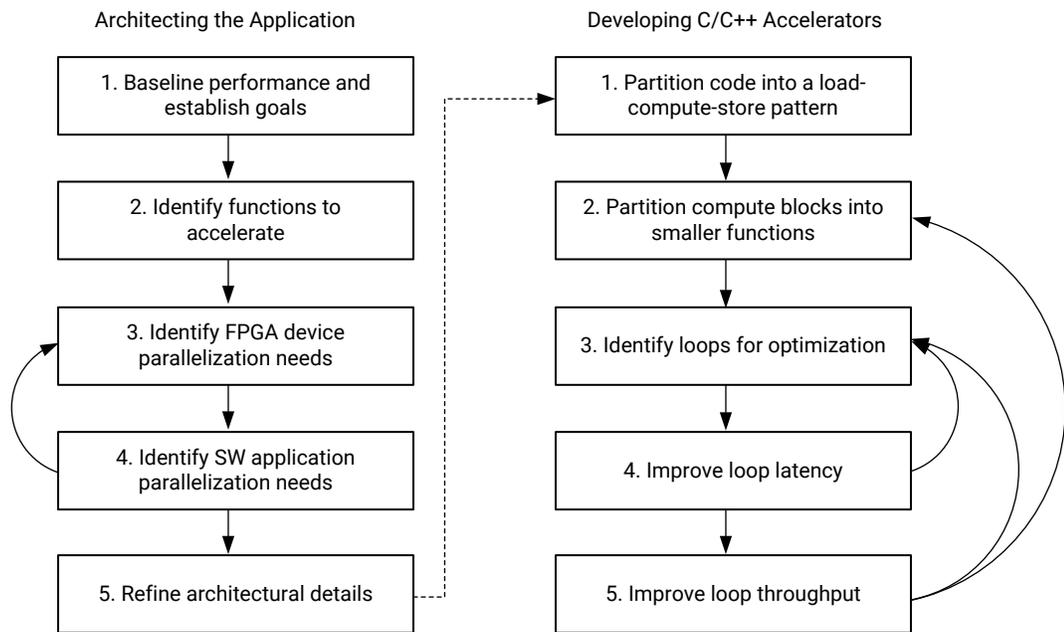
The methodology is comprised of two major phases:

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

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

In the second phase, the developer implements the kernels. This primarily involves structuring source code and applying the desired compiler pragma to create the desired kernel architecture and meet the performance target.

Figure 5: Methodology Overview



X23281-092619

Performance optimization is an iterative process. The initial version of an accelerated application will likely not produce the best possible results. The methodology described in this guide is a process involving constant performance analysis and repeated changes to all aspects of the implementation.

Recommendations

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

- [Chapter 2: Developing Applications](#)
- [Vitis Tutorials](#) on GitHub.

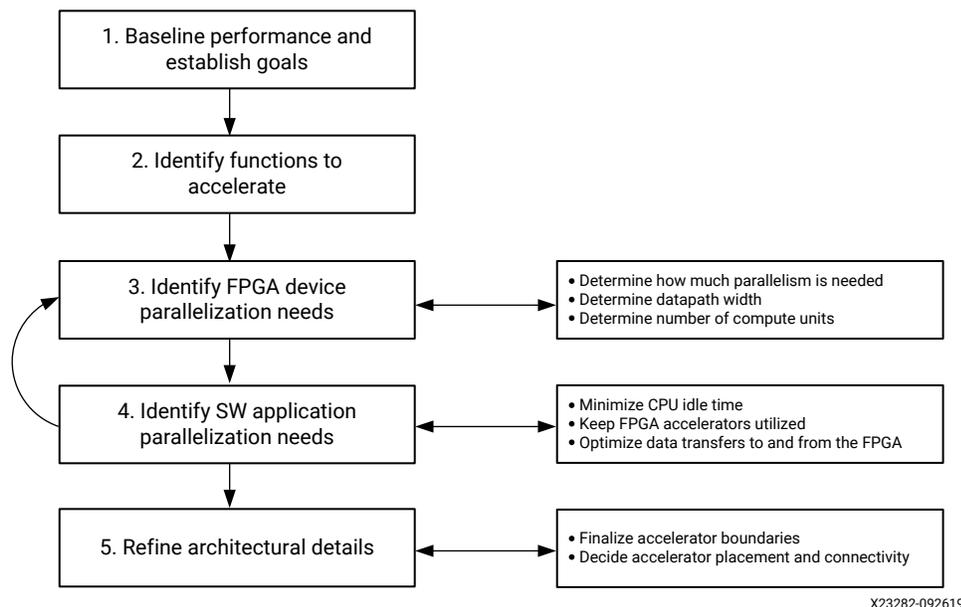
In addition to understanding the key aspects of the Vitis software platform, a good understanding of the following topics will help achieve optimal results with this methodology:

- Application domain
- Software acceleration principles
- Concepts, features and architecture of device
- Features of the targeted device accelerator card and corresponding shell
- Parallelism in hardware implementations (<http://kastner.ucsd.edu/hlsbook/>)

Methodology for Architecting a Device Accelerated Application

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

Figure 6: Methodology for Architecting the Application



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

Step 1: Establish a Baseline Application Performance and Establish Goals

Start by measuring the runtime and throughput performance of the target application. These performance numbers should be generated for the entire application (end-to-end) as well as for each major function in the application. These numbers provide the baseline for most of the subsequent analysis process.

Measure Running Time

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

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

Figure 7: Gprof Output Example

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
70.45	25.54	25.54	256	99.76	99.76	F4(int*, int*, int*)
12.44	30.05	4.51	256	17.61	17.61	F2(int*, int*)
9.91	33.64	3.59	256	14.03	14.03	F1(int*, int*, int*)
7.83	36.48	2.84	256	11.08	11.08	F3(int*, int*)
0.00	36.48	0.00	256	0.00	142.48	F(int*, int*)

Measure Throughput

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

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

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

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

Determine the Maximum Achievable Throughput

In most device-accelerated systems, the maximum achievable throughput is limited by the PCIe® bus. PCIe performance is influenced by many different aspects, such as motherboard, drivers, targeted shell, and transfer sizes. Run DMA tests upfront to measure the effective throughput of PCIe transfers and thereby determine the upper bound of the acceleration potential, such as the xbutil dma test.

Figure 8: Sample Result of `dmatest` on an Alveo U200 Data Center Accelerator Card

```
$ xbutil dmatest
INFO: Found total 1 card(s), 1 are usable
Total DDR size: 65536 MB
Reporting from mem_topology:
Data Validity & DMA Test on bank0
Host -> PCIe -> FPGA write bandwidth = 11381.7 MB/s
Host <- PCIe <- FPGA read bandwidth = 8358.9 MB/s
Data Validity & DMA Test on bank1
Host -> PCIe -> FPGA write bandwidth = 11235.3 MB/s
Host <- PCIe <- FPGA read bandwidth = 7485.3 MB/s
INFO: xbutil dmatest succeeded.
```

An acceleration goal that exceeds this upper bound throughput cannot be met as the system will be I/O bound. Similarly, when defining kernel performance and I/O requirements, keep this upper bound in mind.

Establish Overall Acceleration Goals

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

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

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

Step 2: Identify Functions to Accelerate

After establishing the performance baseline, the next step is to determine which functions should be accelerated in the device. To this extent, there are two aspects to consider:

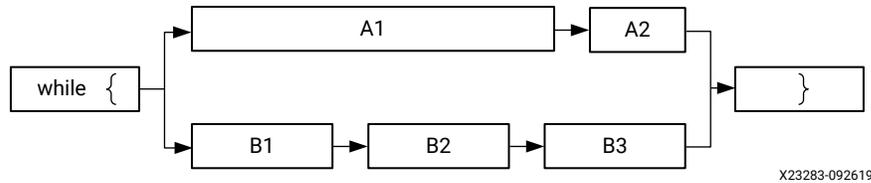
- **Performance bottlenecks:** Which functions are in application hot spots?
- **Acceleration potential:** Do these functions have the potential for acceleration?

Identify Performance Bottlenecks

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

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

Figure 9: Application with Two Parallel Paths



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

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

Identify Acceleration Potential

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

- What is the computational complexity of the function?

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

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

- What is the computational intensity of the function?

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

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

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

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

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

$$\text{Maximum Acceleration Potential} = T_{\text{Min}} / T_{\text{SW}}$$

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

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

Step 3: Identify Device Parallelization Needs

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

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

As described, the assembly line allows the progressive and simultaneous processing of inputs. In hardware, this kind of parallelism is called pipelining. The number of stations on the assembly line corresponds to the number of stages in the hardware pipeline.

Another dimension of parallelism within kernels is the ability to process multiple samples at the same time. This is like putting not just one, but multiple samples on the conveyer belt at the same time. To accommodate this, the assembly line stations are customized to process multiple samples in parallel. This is effectively defining the width of the datapath within the kernel.

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

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

Estimate Hardware Throughput without Parallelization

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

$$T_{HW} = \text{Frequency} / \text{Computational Intensity} = \text{Frequency} * \max(V_{INPUT}, V_{OUTPUT}) / V_{OPS}$$

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

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

Determine How Much Parallelism is Needed

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

$$\text{Speed-up} = T_{HW} / T_{SW} = F_{max} * \text{Running Time} / V_{ops}$$

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

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

$$\text{Parallelism Needed} = T_{Goal} / T_{HW} = T_{Goal} * V_{ops} / F_{max} * \max(V_{INPUT}, V_{OUTPUT})$$

This parallelism can be implemented in various ways: by widening the datapath, by using multiple engines, and by using multiple kernel instances. The developer should then determine the best combination given his needs and the characteristics of his application.

Determine How Many Samples the Datapath Should be Processing in Parallel

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

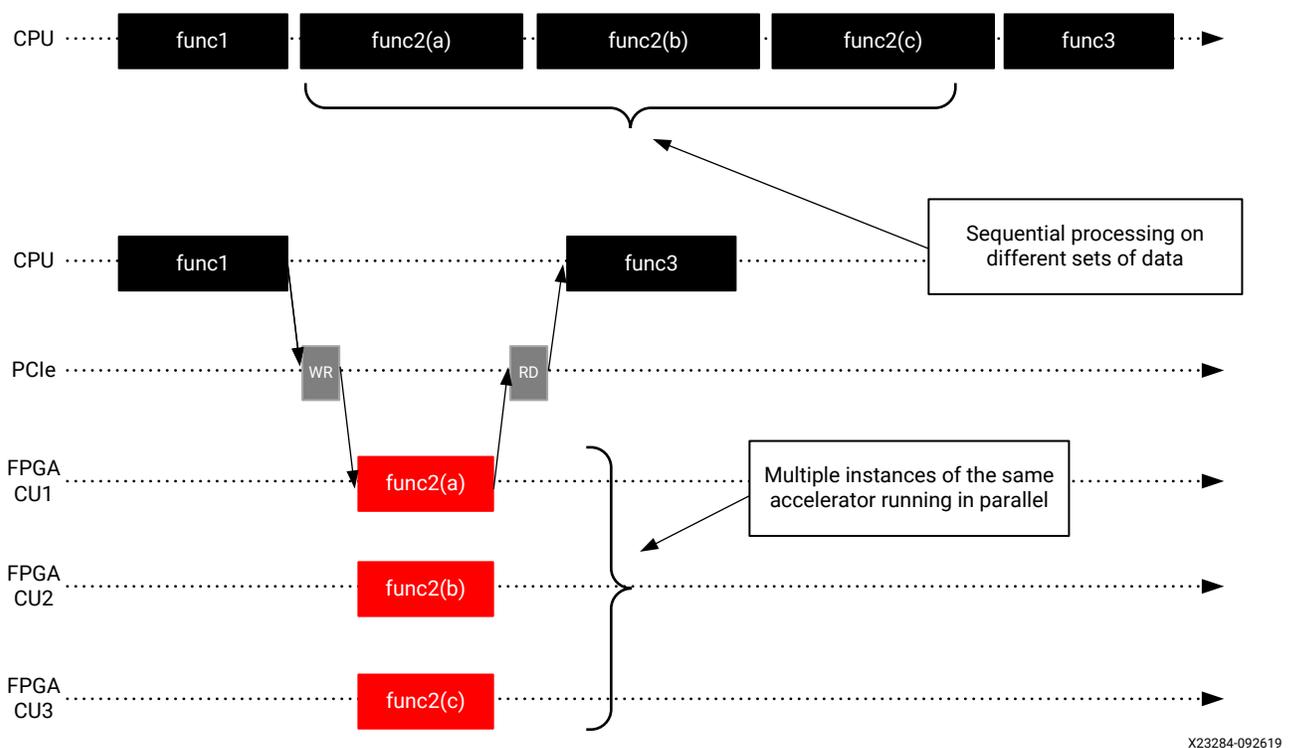
Processing more samples in parallel using a wider datapath improves performance by reducing the latency (running time) of the accelerated function.

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

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

Adding more kernel instances improves the performance of the application by allowing the execution of more invocations of the targeted function in parallel, as shown below. Multiple data sets are processed concurrently by the different instances. Application performance scales linearly with the number of instances, provided that the host application can keep the kernels busy.

Figure 10: Improving Performance with Multiple Compute Units



As illustrated in the [Using Multiple Compute Units](#) tutorial, the Vitis technology makes it easy to scale performance by adding additional instances.

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

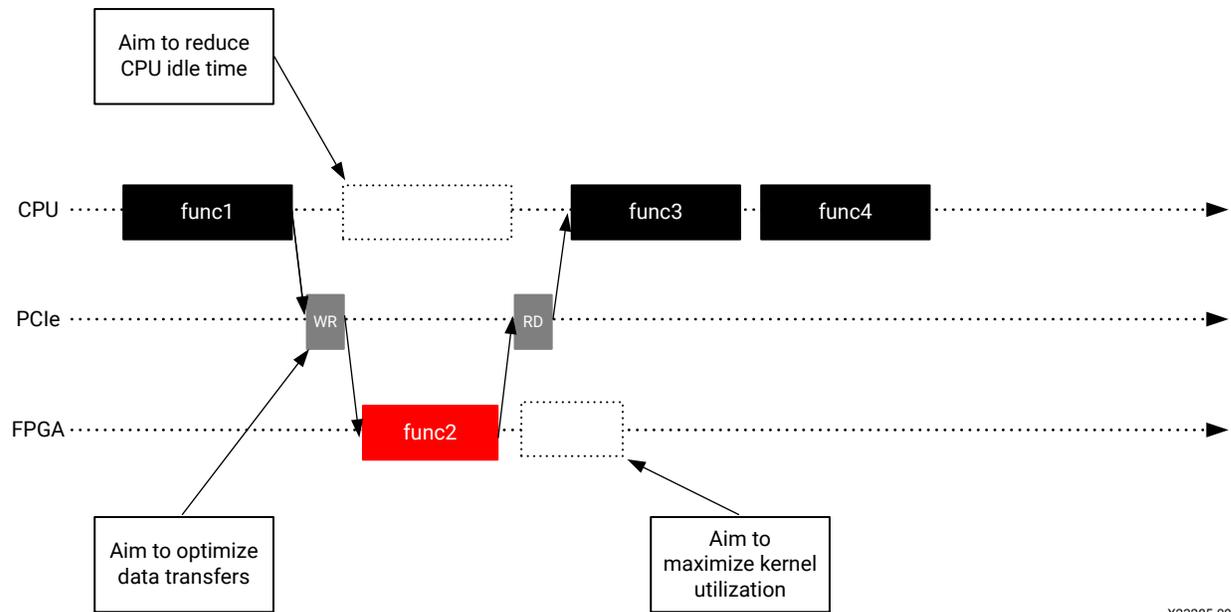
Step 4: Identify Software Application Parallelization Needs

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

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

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

Figure 11: Software Optimization Goals



X23285-092619

In the world of factories and assembly lines, the host application would be the headquarters keeping busy and planning the next generation of products while the factories manufacture the current generation.

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

Minimize CPU Idle Time While the Device Kernels are Running

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

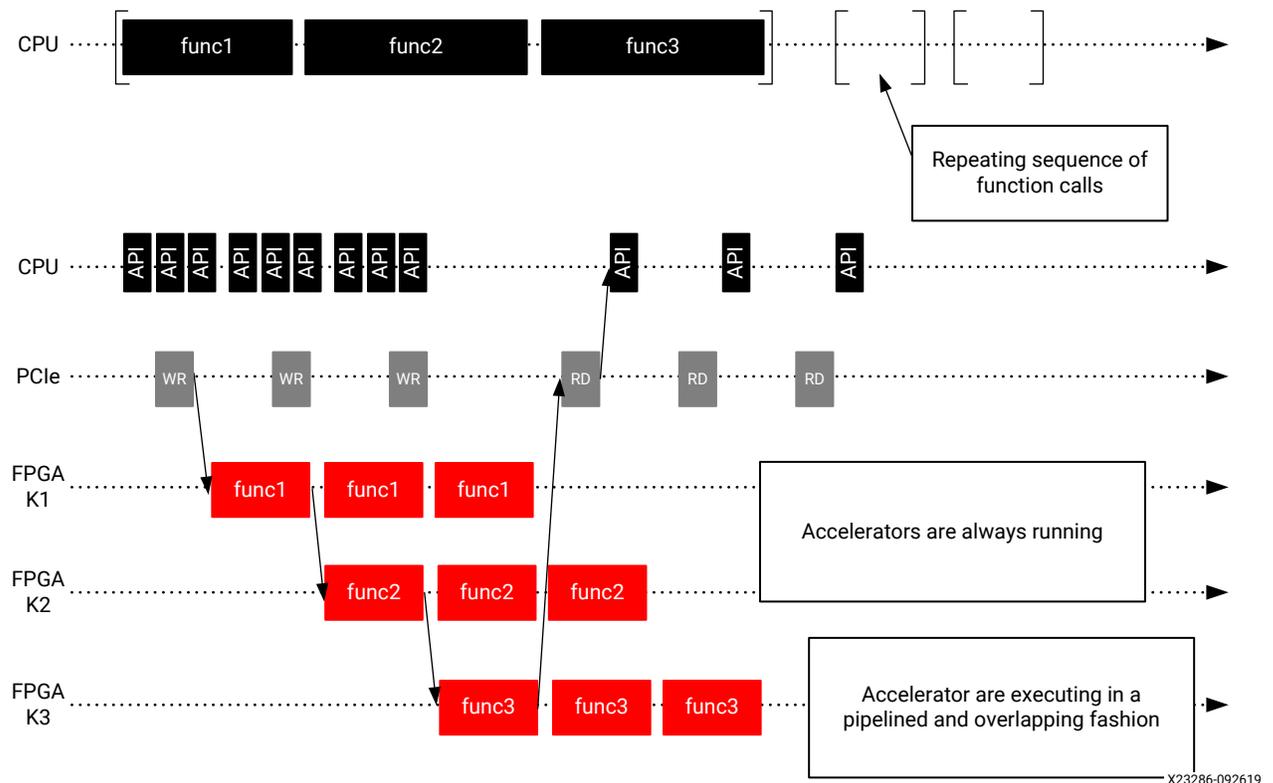
Instead, engineer the software application to avoid such idle cycles. Begin by identifying parts of the application that do not depend on the results of the kernel. Then structure the application so that these functions can be executed on the host in parallel to the kernel running in the device.

Keep the Device Kernels Utilized

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

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

Figure 12: Pipelined Execution of Accelerators



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

More information on software pipelining can be found in the [Concurrent Kernel Execution \(C\) example](#), and the [Host Code Optimization tutorial](#).

Optimize Data Transfers to and from the Device

In an accelerated application, data must be transferred from the host to the device especially in the case of PCIe-based applications. This introduces latency, which can be very costly to the overall performance of the application.

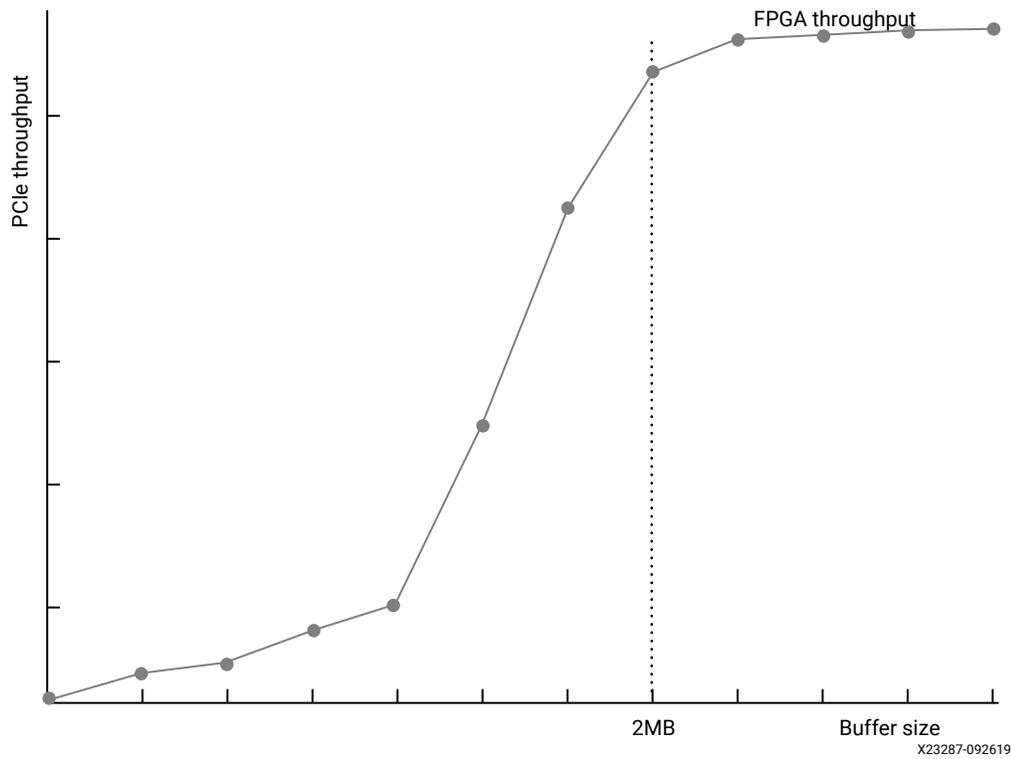
Data needs to be transferred at the right time, otherwise the application performance is negatively impacted if the kernel must wait for data to be available. It is therefore important to transfer data ahead of when the kernel needs it. This is achieved by overlapping data transfers and kernel execution, as described in [Keep the Device Kernels Utilized](#). As shown in the sequence in the previous figure, this technique enables hiding the latency overhead of the data transfers and avoids the kernel having to wait for data to be ready.

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

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

As shown in the following figure, on PCIe-based systems it is usually better to make data transfers of 1MB or more. Running DMA tests in advance using the "xbutil" utility can be useful for finding the optimal buffer sizes. See [dmatest](#) for more information.

Figure 13: Performance of PCIe Transfers as a Function of Buffer Size



It is recommended that you group multiple sets of data in a common buffer to achieve the highest possible throughput.

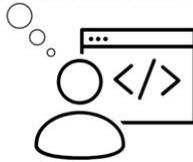
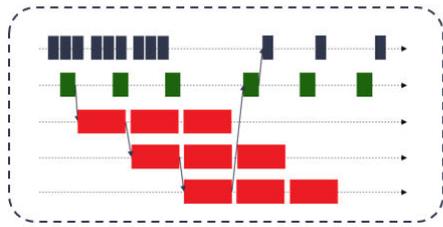
Conceptualize the Desired Application Timeline

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

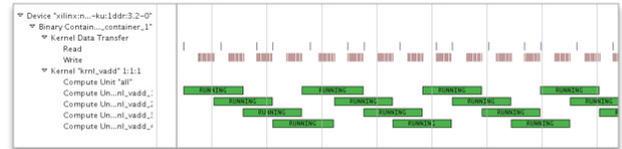
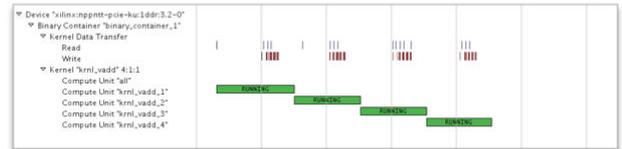
At this point, it is very useful to summarize this information in the form of an expected application timeline. Application timeline sequences, such as the ones shown in the previous chapter, are very effective ways of representing performance and parallelization in action as the application runs. They represent how the potential parallelism built into the architecture is mobilized by the application.

Figure 14: Application Timelines

Developer's Application Timeline View



Vitis Application Timeline View



The Vitis software platform generates timeline views from actual application runs. If the developer has a desired timeline in mind, he can then compare it to the actual results, identify potential issues, and iterate and converge on the optimal results, as shown in the above figure.

Step 5: Refine Architectural Details

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

Finalize Kernel Boundaries

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

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

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

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

Putting multiple engines in a kernel also reduces the number of ports and the number of transactions to global memory that require arbitration, improving the effective bandwidth.

On the other hand, this transformation requires coding explicit I/O multiplexing behavior in the kernel. This is a trade-off the developer needs to make.

Decide Kernel Placement and Connectivity

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

At this point, it is important to understand the features of the targeted platform (shell) and what global memory resources are available. For instance, the Alveo U200 Data Center accelerator card has 4 x 16 GB banks of DDR4 and 3 x 128 KB banks of PLRAM distributed across three super-logic regions (SLRs). For more information, refer to [Vitis 2019.2 Software Platform Release Notes](#).

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

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

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

Methodology for Developing C/C++ Kernels

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

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

- Throughput goal
- Latency goal
- Datapath width
- Number of engines
- Interface bandwidth

These requirements drive the kernel development and optimization process. Achieving the kernel throughput goal is the primary objective, as overall application performance is predicated on each kernel meeting the specified throughput.

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

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

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

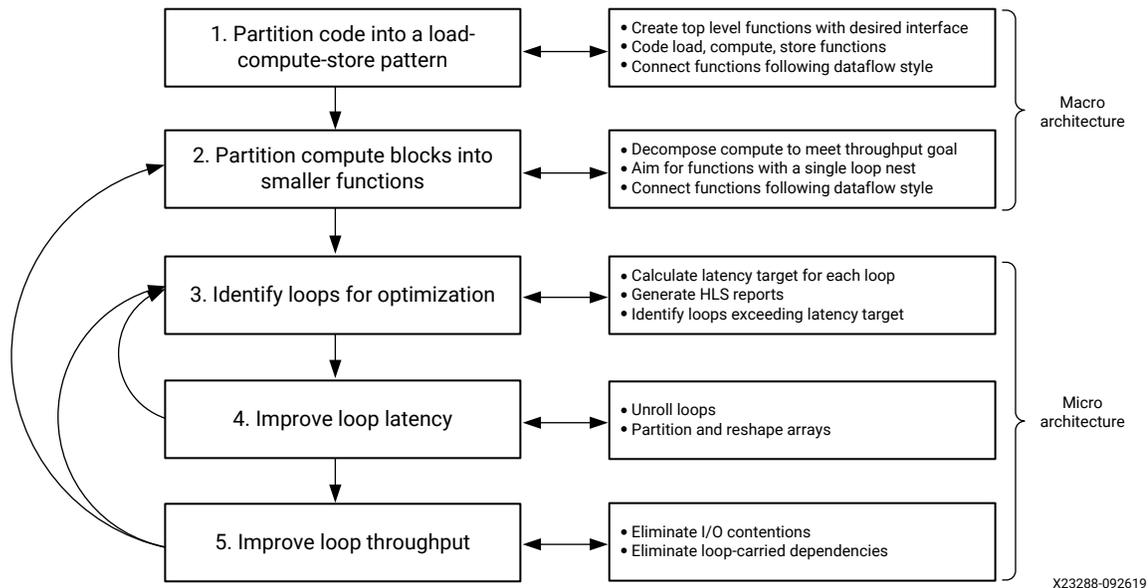
- Functionality is the mathematical relationship between input parameters and output results.
- Algorithm is a series of steps for performing a specific functionality. A given functionality can be performed using a variety of different algorithms. For instance, a sort function can be implemented using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

You must understand that the Vitis compiler (also known as High-Level Synthesis) generates optimized hardware architectures from algorithms written in C/C++. However, it does not transform a particular algorithm into another one.

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

The following methodology assumes that you have identified a suitable algorithm for the functionality that you want to accelerate.

Figure 15: Kernel Development Methodology



About the High-Level Synthesis Compiler

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

The HLS compiler is more restrictive than standard software compilers. For example, there are unsupported constructs including: system function calls, dynamic memory allocation and recursive functions. See "Unsupported C Constructs" in the *Vivado Design Suite User Guide: High-Level Synthesis (UG902)* for more information.

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

Verification Considerations

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

Step 1: Partition the Code into a Load-Compute-Store Pattern

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

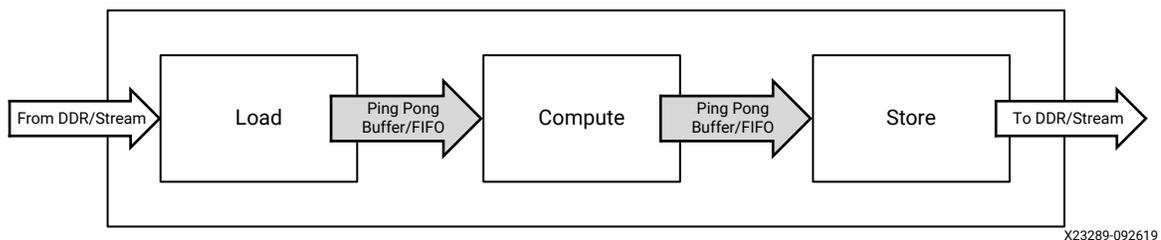
Knowing that kernel accesses to global memory are expensive and that bandwidth is limited, it is very important to carefully plan this aspect of the kernel.

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

This means creating a top-level function with:

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

Figure 16: Load-Compute-Store Pattern



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

The load function is responsible for moving data external to the kernel (i.e. global memory) to the compute function inside the kernel. This function doesn't do any data processing but focuses on efficient data transfers, including buffering and caching if necessary.

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

The store function mirrors the load function. It is responsible for moving data out of the kernel, taking the results of the compute function and transferring them to global memory outside the kernel.

Creating a load-compute-store structure that meets the performance goals starts by engineering the flow of data within the kernel. Some factors to consider are:

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

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

A working example featuring the load-compute-store pattern can be found on the [Vitis Examples](#) GitHub repository.

Create a Top-Level Function with the Desired Interface

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

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

Code the Load and Store Functions

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

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

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

Match Port Width to Datapath Width

In the Vitis software platform, the port of a kernel can be up to 512 bits wide, which means that a kernel can read or write up to 64 bytes per clock cycle per port.

It is recommended to match the width of the kernel ports to width of the datapath in the compute function. For instance, if the datapath needs to process 16 bytes in parallel to meet the desired throughput, then ports should be made 128 bit wide to allow reading and writing 16 bytes in parallel.

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

Use Burst Transfers

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

Atomic accesses to global memory should always be avoided unless absolutely required. The load and store functions should be coded to always infer bursting transaction. This can be done using a `memcpy` operation as shown in the `vadd.cpp` file in the [GitHub example](#), or by creating a tight `for` loop accessing all the required values sequentially, as explained in [Interfaces](#) in [Chapter 2: Developing Applications](#).

Minimize the Number of Data Transfers from Global Memory

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

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

Code the Compute Functions

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

Connect the Load, Compute, and Store Functions

Use standard C/C++ variables and arrays to connect the top-level interfaces and the load, compute and store functions. It can also be useful to use the `hls::stream` class, which models a streaming behavior.

Streaming is a type of data transfer in which data samples are sent in sequential order starting from the first sample. Streaming requires no address management and can be implemented with FIFOs. For more information about the `hls::stream` class, see this "Using HLS Streams" in the *Vivado Design Suite User Guide: High-Level Synthesis (UG902)*.

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

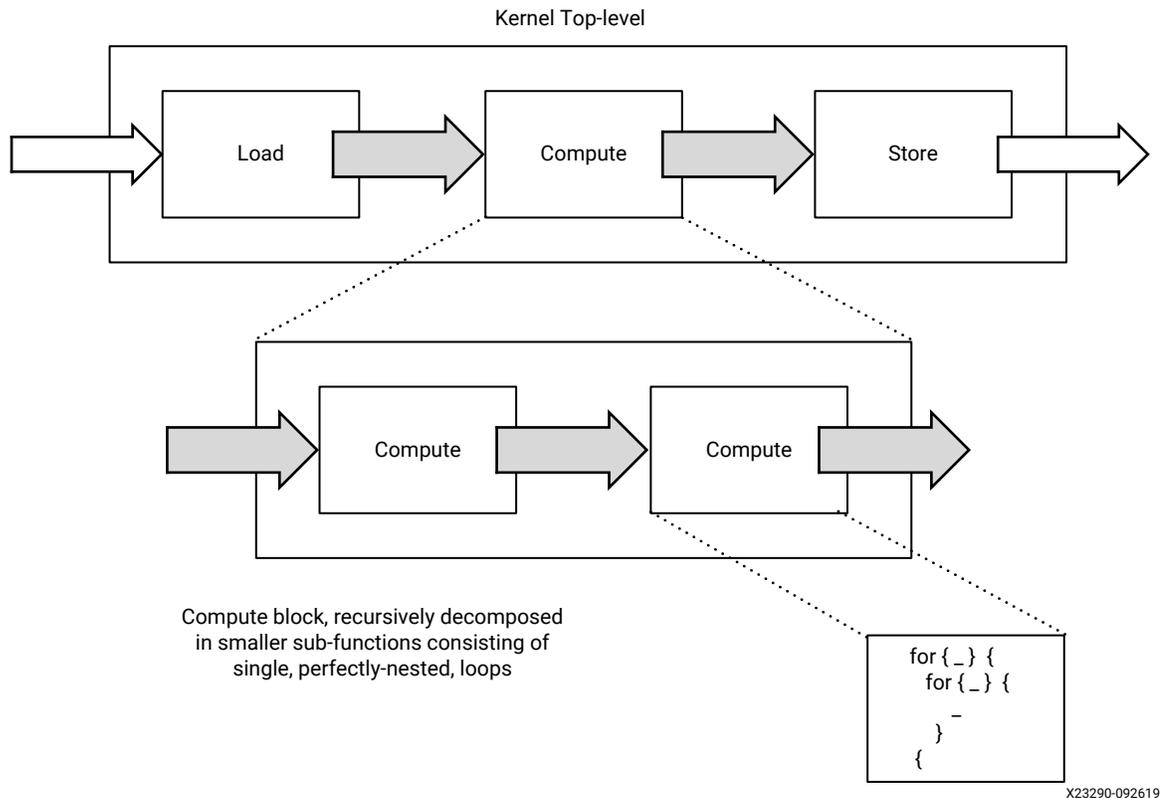
- Data should be transferred in the forward direction only, avoiding feedback whenever possible.
- Each connection should have a single producer and a single consumer.
- Only the load and store functions should access the primary interface of the kernel.

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

Step 2: Partition the Compute Blocks into Smaller Functions

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

Figure 17: Compute Block Sub-Functions



Decompose to Identify Throughput Goals

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

$$\text{Throughput}(\text{Kernel}) = \min(\text{Throughput}(\text{Task}_1), \text{Throughput}(\text{Task}_2), \dots, \text{Throughput}(\text{Task}_N))$$

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

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

Aim for Functions with a Single Loop Nest

As a general rule, if a function has sequential loops in it, these loops execute sequentially in the hardware implementation generated by the HLS compiler. This is usually not desirable, as sequential execution hampers throughput.

However, if these sequential loops are pushed into sequential functions, then the HLS compiler can apply the dataflow optimization and generate an implementation that allows the pipelined and overlapping execution of each task. For more information on the dataflow optimization, see this [link](#) in *Vivado Design Suite User Guide: High-Level Synthesis (UG902)*.

During this partitioning and refining process, put sequential loops into individual functions. Ideally, the lowest-level compute block should only contain a single perfectly-nested loop. For more information on loops, see this [link](#) in *Vivado Design Suite User Guide: High-Level Synthesis (UG902)*

Connect Compute Functions Using the Dataflow ‘Canonical Form’

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

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

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

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

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

Step 3: Identify Loops Requiring Optimization

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

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

$$T = \max(V_{\text{INPUT}}, V_{\text{OUTPUT}}) / \text{Latency}$$

Both the target throughput and the volume of data consumed and produced by the function should be known at this stage of the ‘outside-in’ decomposition process described in this methodology. The developer can therefore easily derive the latency target for each function.

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

The latency of a loop can be calculated as follows:

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

Where:

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

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

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

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

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

Step 4: Improve Loop Latencies

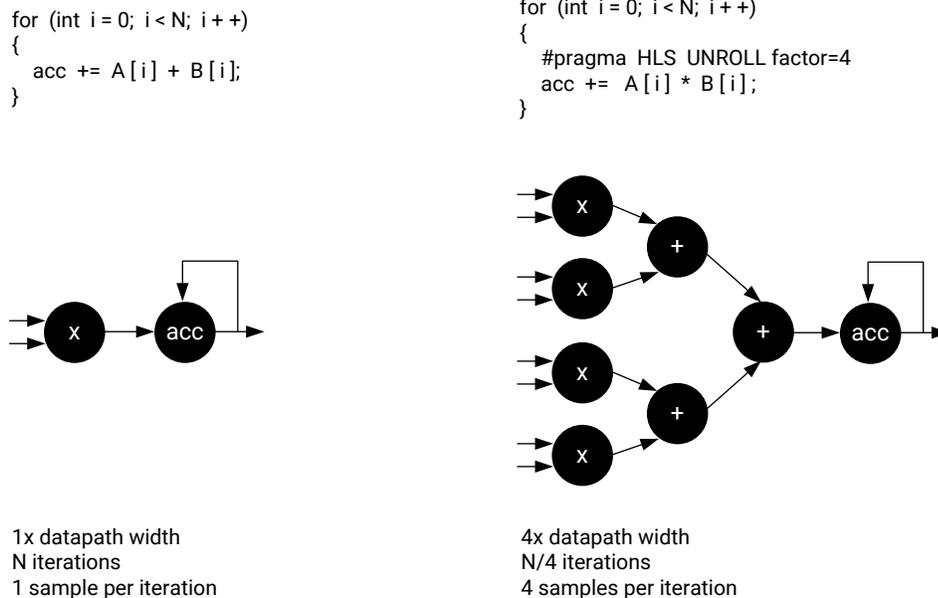
After identifying loops latencies that exceed their target, the first optimization to consider is loop unrolling.

Apply Loop Unrolling

Loop unrolling unwinds the loop, allowing multiple iterations of the loop to be executed together, reducing the loop’s overall trip count.

In the industrial analogy, factories are kernels, assembly lines are dataflow pipelines, and stations are compute functions. Unrolling creates stations which can process multiple objects arriving at the same time on the conveyor belt, which results in higher performance.

Figure 18: Loop Unrolling



X23291-092619

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

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

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

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

Apply Array Partitioning

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

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

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

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

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

Step 5: Improve Loop Throughput

If improving loop latency by reducing the trip count wasn't sufficient, look at ways to reduce the Initiation Interval (II).

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

There are two main factors that can limit the II:

- I/O contentions
- Loop-carried dependencies

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

Eliminate I/O Contentions

I/O contentions appear when a given I/O port of internal memory resources must be accessed more than once per loop iteration. A loop cannot be pipelined with an II lower than the number of times an I/O resource is accessed per loop iteration. If port A must be accessed four times in a loop iteration, then the lowest possible II will be 4.

The developer needs to assess whether these I/O accesses are necessary or if they can be eliminated. The most common techniques for reducing I/O contentions are:

- Creating internal cache structures

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

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

- Reconfiguring I/Os and memories

As explained earlier in the section about improving latency, the HLS compiler maps arrays to memories, and the default memory configuration can offer sufficient bandwidth for the required throughput. The array partitioning and reshaping pragmas can also be used in this context to create memory structure with higher bandwidth, thereby improving the potential II of the loop.

Eliminate Loop-Carried Dependencies

The most common case for loop-carried dependencies is when a loop iteration relies on a value computed in a prior iteration. There are differences whether the dependencies are on arrays or on scalar variables. For more information, see this [link](#) in the *Vivado Design Suite User Guide: High-Level Synthesis (UG902)*.

- Eliminating dependencies on arrays

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

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

- Eliminating dependencies on scalars

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

Advanced Techniques

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

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

Developing Applications

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

Programming Model

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

Device Topology

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

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

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

Host Application

In the Vitis core development kit, host code is written in C or C++ language using the industry standard OpenCL API. The Vitis core development kit provides an OpenCL 1.2 embedded profile conformant runtime API.

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

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

Note: The Vitis core development kit supports the OpenCL Installable Client Driver (ICD) extension (`cl_khr_icd`). This extension allows multiple implementations of OpenCL to co-exist on the same system.

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

Setting Up the OpenCL Environment

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



TIP: The host code examples and API commands used in this document follow the OpenCL C API. However, XRT also supports the OpenCL C++ wrapper API, and many of the examples in the [GitHub repository](https://github.com/Xilinx/Vitis) are written using the C++ API. For more information on this C++ wrapper API, refer to <https://www.khronos.org/registry/OpenCL/specs/opencv-cplusplus-1.2.pdf>.

Platform

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

```
cl_platform_id platform_id;           // platform id

err = clGetPlatformIDs(16, platforms, &platform_count);

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

```

        NULL);

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

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



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

```

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

Devices

After a Xilinx platform is found, the application needs to identify the corresponding Xilinx devices.

The following code demonstrates finding all the Xilinx devices, with an upper limit of 16, by using API `clGetDeviceIDs`.

```

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

err = clGetDeviceIDs(platform_id, CL_DEVICE_TYPE_ACCELERATOR,
    16, devices, &num_devices);

printf("INFO: Found %d devices\n", num_devices);

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



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

Sub-Devices

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

The following example shows:

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

```

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

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

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

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

// Iterating over sub-devices
std::for_each(devices.begin(), devices.end(), [kernel](cl_device_id sdev) {

    // Context for sub-device
    auto context = clCreateContext(0, 1, &sdev, nullptr, nullptr, &err);

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

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



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

Context

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

```

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

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

Command Queues

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

1. Single out-of-order command queue: Multiple kernel executions can be requested through the same command queue. XRT dispatches kernels as soon as possible, in any order, allowing concurrent kernel execution on the FPGA.
2. Multiple in-order command queue: Each kernel execution will be requested from different in-order command queues. In such cases, XRT dispatches kernels from the different command queues, improving performance by running them concurrently on the device.

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

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

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

Program

As described in [Build Process](#), the host and kernel code are compiled separately to create separate executable files: the host program executable and the FPGA binary (.xclbin). When the host application runs, it must load the .xclbin file using the `clCreateProgramWithBinary` API.

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

```
unsigned char *kernelbinary;
char *xclbin = argv[1];

printf("INFO: loading xclbin %s\n", xclbin);

int size=load_file_to_memory(xclbin, (char **) &kernelbinary);
size_t size_var = size;

cl_program program = clCreateProgramWithBinary(context, 1, &device_id,
    &size_var, (const unsigned char **) &kernelbinary,
    &status, &err);
```

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

The example performs the following steps:

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



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

2. The `load_file_to_memory` function is used to load the file contents in the host machine memory space.
3. The `clCreateProgramWithBinary` API is used to complete the program creation process in the specified context and device.

Executing Commands in the FPGA

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

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

Setting Up Kernels

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

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

```
kernel1 = clCreateKernel(program, "<kernel_name_1>", &err);
kernel2 = clCreateKernel(program, "<kernel_name_2>", &err); // etc
```

Setting Kernel Arguments

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

1. Scalar arguments are used for small data transfer, such as constant or configuration type data. These are write-only arguments from the host application perspective, meaning they are inputs to the kernel.
2. Memory buffer arguments are used for large data transfer. The value is a pointer to a memory object created with the context associated with the program and kernel objects. These can be inputs to, or outputs from the kernel.

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

```
// Create memory buffers
cl_mem dev_buf1 = clCreateBuffer(context, CL_MEM_WRITE_ONLY, size,
&host_mem_ptr1, NULL);
cl_mem dev_buf2 = clCreateBuffer(context, CL_MEM_READ_ONLY, size,
&host_mem_ptr2, NULL);

int err = 0;
// Setup scalar arguments
cl_uint scalar_arg_image_width = 3840;
err |= clSetKernelArg(kernel, 0, sizeof(cl_uint), &scalar_arg_image_width);
cl_uint scalar_arg_image_height = 2160;
err |= clSetKernelArg(kernel, 1, sizeof(cl_uint),
&scalar_arg_image_height);

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



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

Buffer Transfer to/from the FPGA Device

Interactions between the host program and hardware kernels rely on transferring data to and from the global memory in the device. The method to send data back and forth from the FPGA is using `clCreateBuffer`, `clEnqueueWriteBuffer`, and `clEnqueueReadBuffer` commands.



RECOMMENDED: Xilinx recommends using `clEnqueueMigrateMemObjects` instead of `clEnqueueReadBuffer` and `clEnqueueWriteBuffer`.

This method is demonstrated in the following code example.

```
int host_mem_ptr[MAX_LENGTH]; // host memory for input vector
// Fill the memory input
for(int i=0; i<MAX_LENGTH; i++) {
    host_mem_ptr[i] = <... >
}
cl_mem dev_mem_ptr = clCreateBuffer(context, CL_MEM_READ_WRITE,
    sizeof(int) * number_of_words, NULL, NULL);

err = clEnqueueWriteBuffer(commands, dev_mem_ptr, CL_TRUE, 0,
    sizeof(int) * number_of_words, host_mem_ptr, 0, NULL, NULL);
```



IMPORTANT! A single buffer cannot be bigger than 4 GB, yet to maximize throughput from the host to global memory, Xilinx also recommends keeping the buffer size at least 2 MB if possible.

For simple applications, the provided example code would be sufficient to transfer data from the host to the device memory. However, there are a number of coding practices you should adopt to maximize performance and fine-grain control.

Using `clEnqueueMigrateMemObjects`



IMPORTANT! Using `clEnqueueMigrateMemObjects` with `CL_MEM_USE_HOST_PTR` is not applicable to the embedded platform. Embedded platform users should use the `clEnqueueMapBuffer` method, as described in [Using `clEnqueueMapBuffer`](#).

The OpenCL framework provides a number of APIs for transferring data between the host and the device. Typically, data movement APIs, such as `clEnqueueWriteBuffer` and `clEnqueueReadBuffer`, implicitly migrate memory objects to the device after they are enqueued. They do not guarantee when the data is transferred, and this makes it difficult for the host application to synchronize the movement of memory objects with the computation performed on the data.

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

There are two main parts of a `cl_mem` object: host side pointer and device side pointer. Before the kernel starts its operation, the device side pointer is implicitly allocated on the device side memory (for example, on a specific location inside the device global memory) and the buffer becomes a resident on the device. However, by using `clEnqueueMigrateMemObjects` this allocation and data transfer occur upfront, much ahead of the kernel execution. This especially helps to enable *software pipelining* if the host is executing the same kernel multiple times, because data transfer for the next transaction can happen when kernel is still operating on the previous data set, and thus hide the data transfer latency of successive kernel executions.



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

The following code shows the use of `clEnqueueMigrateMemObjects`:

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

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

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

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

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

Allocating Page-Aligned Host Memory



IMPORTANT! This topic is not applicable to the embedded platform. Embedded platform users should use the `clEnqueueMapBuffer` method, as described in [Using clEnqueueMapBuffer](#).

XRT allocates memory space in 4K boundary for internal memory management. If the host memory pointer is not aligned to a page boundary, XRT performs extra `memcpy` to make it aligned. Hence you should align the host memory pointer with the 4K boundary to save the extra memory copy operation.

The following is an example of how `posix_memalign` is used instead of `malloc` for the host memory space pointer.

```
int *host_mem_ptr; // = (int*) malloc(MAX_LENGTH*sizeof(int));
// Aligning memory in 4K boundary
posix_memalign(&host_mem_ptr, 4096, MAX_LENGTH*sizeof(int));

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

```

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

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

Using clEnqueueMapBuffer

Another approach for creating and managing buffers is to use `clEnqueueMapBuffer`. With this approach, it is not necessary to create a host space pointer aligned to the 4K boundary. The `clEnqueueMapBuffer` API maps the specified buffer and returns a pointer created by XRT to this mapped region. Then, fill the host side pointer with your data, followed by `clEnqueueMigrateMemObject` to transfer the data to and from the device. The following code example uses this style.

Note: `CL_MEM_USE_HOST_PTR` is not used for `clCreateBuffer`.

```

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

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

// Setting arguments
clSetKernelArg(kernel, 0, sizeof(cl_mem), &dev_mem_read_ptr);
clSetKernelArg(kernel, 1, sizeof(cl_mem), &dev_mem_write_ptr);

// Get Host side pointer of the cl_mem buffer object
auto host_write_ptr =
clEnqueueMapBuffer(queue, dev_mem_read_ptr, true, CL_MAP_WRITE, 0, bytes, 0, nullptr,
                  nullptr, &err);
auto host_read_ptr =
clEnqueueMapBuffer(queue, dev_mem_write_ptr, true, CL_MAP_READ, 0, bytes, 0, nullptr,
                  nullptr, &err);

// Fill up the host_write_ptr to send the data to the FPGA
for(int i=0; i< MAX; i++) {
    host_write_ptr[i] = <.... >
}

// Migrate
cl_mem mems[2] = {host_write_ptr, host_read_ptr};
clEnqueueMigrateMemObjects(queue, 2, mems, 0, 0, nullptr, &migrate_event));

// Schedule the kernel
clEnqueueTask(queue, kernel, 1, &migrate_event, &enqueue_event);

// Migrate data back to host
clEnqueueMigrateMemObjects(queue, 1, &dev_mem_write_ptr,
                           CL_MIGRATE_MEM_OBJECT_HOST, 1, &enqueue_event,
    
```

```
&data_read_event);
clWaitForEvents(1,&data_read_event);
// Now use the data from the host_read_ptr
```

Buffer Allocation on the Device

By default, all the memory interfaces from all the kernels are connected to a single default global memory bank when kernels are linked. As a result, only one compute unit (CU) can transfer data to and from the global memory bank at a time, limiting the overall performance of the application. If the FPGA contains only one global memory bank, then this is the only option. However, if the device contains multiple global memory banks, you can customize the global memory bank connections by modifying the default connection during kernel linking. This topic is discussed in greater detail in [Mapping Kernel Ports to Global Memory](#). Overall performance is improved by using separate memory banks for different kernels or compute units, enabling multiple kernel memory interfaces to concurrently read and write data.



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

Sub-Buffers

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

Reading a Specific Portion from the Device Buffer

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

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

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

With `clEnqueueMigrateMemObject`, which is recommended over `clEnqueueReadBuffer` or `clEnqueueWriteBuffer`, you can adopt a similar approach by using sub-buffers. This is shown in the following code sample.



TIP: The code sample shows only partial commands to demonstrate the concept.

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

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

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

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

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

// Map the subbuffer into the host space
auto read_mem_host_ptr = clEnqueueMapBuffer(queue, buffer_seg,,,);

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

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

Device Buffer Shared by Multiple Memory Ports or Multiple Kernels

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

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

Kernel Execution

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

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

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

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



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

However, sometimes using a single `clEnqueueTask` to run the kernel is not always feasible due to various reasons. For example, the kernel code can become too big and complex to optimize if it attempts to perform all compute intensive tasks in a single execution. Another possible case is when the host is receiving data over time and not all the data can be processed at one time. Therefore, depending on the situation and application, you may need to break the data and the task of the kernel into multiple `clEnqueueTask` commands as discussed in the next sections.

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

Task Parallelism Using Different Kernels

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

Spatial Data Parallelism: Increase Number of Compute Units

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

Temporal Data Parallelism: Host-to-Kernel Dataflow

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

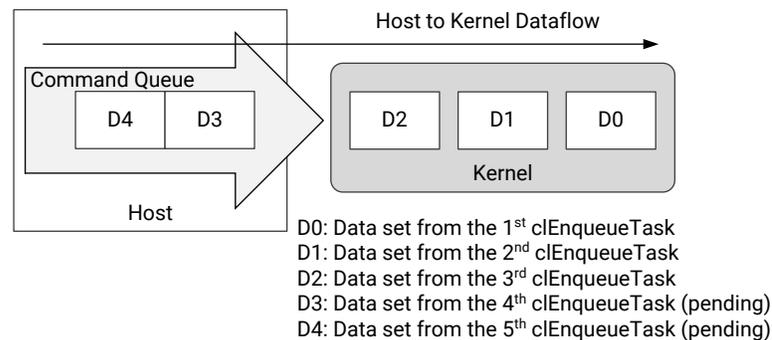
To understand this approach, assume a kernel has only one CU on the FPGA, and the host application enqueues the kernel multiple times with different sets of data. As shown in [Using `clEnqueueMigrateMemObjects`](#), the host application can migrate data to the device global memory ahead of the kernel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

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

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

The following is a conceptual diagram for host-to-kernel dataflow.

Figure 19: Host to Kernel Dataflow



X22774-042519

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

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



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

Symmetrical and Asymmetrical Compute Units

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

- **Symmetrical:** CUs are considered symmetrical when they have exactly the same `--sp` options, and therefore have identical connections to global memory. As a result, the Xilinx Runtime can use them interchangeably. A call to `clEnqueueTask` can result in the invocation of any one instance in a group of symmetrical CUs.
- **Asymmetrical:** CUs are considered asymmetrical when they do not have exactly the same `--sp` options, and therefore do not have identical connections to global memory. Using the same setup of input and output buffers, it is not possible for XRT to execute asymmetrical CUs interchangeably.

Kernel Handle and Compute Units

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

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

Creating Kernel Objects for Specific Compute Units

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

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



IMPORTANT! As discussed in [Creating Multiple Instances of a Kernel](#), the number of CUs is specified by the `connectivity.nk` option in a config file used by the `v++` command during linking. Therefore, whatever is specified in the host program, to create or enqueue kernel objects, must match the options specified by the config file used during linking.

In this case, the Xilinx Runtime identifies the kernel handles (`kernelA`, `kernelB`) for specific CUs, or group of CUs, when the kernel is created. This lets you control which kernel configuration, or specific CU instance is used, when using `clEnqueueTask` from within the host program. This can be useful in the case of asymmetrical CUs, or to perform load and priority management of CUs.

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

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

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

```
// Kernel handle for Symmetrical compute unit group 1: K1,K2,K3
cl_kernel kernelA = clCreateKernel(program, "mykernel:{K1,K2,K3}", &err);

for(i=0; i<3; i++) {
    // Creating buffers for the kernel_handle1
    .....
    // Setting kernel arguments for kernel_handle1
    .....
    // Enqueue buffers for the kernel_handle1
    .....
    // Possible candidates of the executions K1,K2 or K3
    clEnqueueTask(commands, kernelA, 0, NULL, NULL);
    //
}

// Kernel handle for Symmetrical compute unit group 1: K4, K5
cl_kernel kernelB = clCreateKernel(program, "mykernel:{K4,K5}", &err);

for(int i=0; i<2; i++) {
    // Creating buffers for the kernel_handle2
    .....
    // Setting kernel arguments for kernel_handle2
    .....
    // Enqueue buffers for the kernel_handle2
    .....
    // Possible candidates of the executions K4 or K5
    clEnqueueTask(commands, kernelB, 0, NULL, NULL);
}

```

Event Synchronization

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

The following code shows examples for `clFinish` and `clWaitForEvents`.

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

// Create event, read memory from device, wait for read to complete, verify
// results
cl_event readevent;
// host memory for output vector
int host_mem_output_ptr[MAX_LENGTH];
//Enqueue ReadBuffer, with associated event object
clEnqueueReadBuffer(command_queue, dev_mem_ptr, CL_TRUE, 0, sizeof(int) *
number_of_words,

```

```

    host_mem_output_ptr, 0, NULL, &readevent );
    // Wait for clEnqueueReadBuffer event to finish
    clWaitForEvents(1, &readevent);
    // After read is complete, verify results
    ...
  
```

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

1. The `clFinish` API has been explicitly used to block the host execution until the kernel execution is finished. This is necessary otherwise the host can attempt to read back from the FPGA buffer too early and may read garbage data.
2. The data transfer from FPGA memory to the local host machine is done through `clEnqueueReadBuffer`. Here the last argument of `clEnqueueReadBuffer` returns an event object that identifies this particular read command, and can be used to query the event, or wait for this particular command to complete. The `clWaitForEvents` command specifies a single event (the `readevent`), and waits to ensure the data transfer is finished before verifying the data.

Post- Processing and FPGA Cleanup

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

```

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

Summary

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

1. Add error checking after each OpenCL API call for debugging purpose, if required.
2. In the Vitis core development kit, one or more kernels are separately compiled/linked to build the `.xclbin` file. The API `clCreateProgramWithBinary` is used to build the `cl_program` object from the kernel binary.
3. Use buffer for setting the kernel argument (`clSetKernelArg`) before any enqueue operation on the buffer.
4. Transfer data back and forth from the host code to the kernel by using `clEnqueueMigrateMemObjects` or `clEnqueueMapBuffer`.
5. Use `posix_memalign` to align the host memory pointer at 4K boundary (applicable for PCIe-based platforms).

6. Preferably use the out-of-order command queue for concurrent command execution on the FPGA.
7. Execute the whole workload with `clEnqueueTask`, rather than splitting the workload by using `clEnqueueNDRangeKernel`.
8. Use event synchronization commands, `clFinish` and `clWaitForEvents`, to resolve dependencies of the asynchronous OpenCL API calls.
9. Release all OpenCL allocated resources when finished.

Kernel Requirements

The Vitis software platform supports kernels written in C/C++, RTL, or OpenCL. Regardless of source language, all kernels in the Vitis application acceleration development flow have the same properties, and must adhere to the same set of requirements described below.

The key requirements discussed below include the block-level protocol to control the kernel, to start it when ready, and to recognize when it is done; and the interface protocol implemented by the kernel for function arguments, to perform data transfer into and out of the kernel. These requirements are discussed in the following sections.

Kernel Execution Mode

There are three types of kernel execution modes as described in the following table. These modes are mutually exclusive and each kernel can have only one of execution mode.

Sequential Mode	Pipelined Mode	Free Running Mode
<ul style="list-style-type: none"> • Kernel is started by the host program using an API call. • Kernel can only be restarted after the current task is completed. • After the kernel is done, the host program is notified. • Host can enqueue multiple requests using API calls. • Supports streaming, register, and memory mapped interfaces. 	<ul style="list-style-type: none"> • Kernel started by host program using API call. • Kernel can be restarted before its current task is completed. There can be multiple invocations of the same kernel running in parallel. • After the kernel is done, the host application is notified. • Host can enqueue multiple requests using API calls. • Improved performance as multiple invocations of kernel can be overlapped (pipelined). • Supports streaming, register, and memory mapped interfaces. 	<ul style="list-style-type: none"> • The kernel starts as soon as the Xilinx device is programmed with the <code>xclbin</code>. • Kernel is running continuously. It does not stop. • Interacts with host program or other kernels only through streaming interfaces. • Only supports streaming and register interfaces. Memory mapped interfaces are not supported.

When the kernel starts, the application can require the kernel to restart only after it has completed its task, or the application can restart the kernel before it has completed the prior task. Kernels can also run freely on the platform, as described in [Free-running Kernel](#), being capable of continuously receiving data from the host. The application developer will decide what kernel characteristics are appropriate to their application.

The kernel execution mode is controlled by a set of protocols that define how the kernel can start and end a specified task. These are called block-level interface protocols, and they control the kernel independently of any port-level I/O protocol. Block-level handshake signals specify the following:

- When the kernel can start to perform the operation (`ap_start`).
- When the kernel task is complete (`ap_done`).
- When the kernel is idle (`ap_idle`) and ready (`ap_ready`) for new inputs.

Kernel Interface Types

The three types of interfaces supported are described in the following table.

Memory Mapped	Register	Streaming
<ul style="list-style-type: none"> • Designed for bi-directional data transfers with global memory (DDR, PLRAM, HBM). • Access pattern can be random. • Introduces additional latency for memory transfers. • Kernel acts as a master accessing data stored into global memory. • Base Address of data is sent via Register interface. • Host program allocates the buffer for the size of the dataset. 	<ul style="list-style-type: none"> • Designed for scalar values between the host application and the kernel. • Kernel acts as a slave. Register reads and writes are initiated by the host program. 	<ul style="list-style-type: none"> • Designed for uni-directional data transfers between host to card (H2C), card to host (C2H), or between two kernels (K2K). • Access pattern is sequential. • Does not use global memory. • Improves performance. • Data set is unbounded. Sideband signal can be used to indicate the last value in the stream.

Requirements

To satisfy the Vitis application acceleration development flow execution model, a kernel must adhere to following requirements:

- Free running kernels can not have memory ports.
 - Note:** OpenCL kernels do not support continuous mode.
- Only one Register interface is used to access control signals and pass scalar arguments.
- Kernels require at least one of the following interfaces, but can have both:
 - Memory mapped interface to communicate with memory.

- Streaming interface to stream data between the host and kernel, and between kernels.

C/C++ Kernels

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

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



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

```
extern "C" {  
    void kernel_function(int *in, int *out, int size);  
}
```

Data Types

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

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



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

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

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

Arbitrary Precision Integer Types

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

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

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

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

Arbitrary Precision Fixed-Point Data Types

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

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

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

- Header file: `ap_fixed.h`
- Signed fixed point: `ap_fixed<W, I, Q, O, N>`
- Unsigned fixed point: `ap_ufixed<W, I, Q, O, N>`
 - W = Total width < 1024 bits
 - I = Integer bit width. The value of I must be less than or equal to the width (W). The number of bits to represent the fractional part is W minus I. Only a constant integer expression can be used to specify the integer width.
 - Q = Quantization mode. Only predefined enumerated values can be used to specify Q. The accepted values are:
 - `AP_RND`: Rounding to plus infinity.

- AP_RND_ZERO: Rounding to zero.
- AP_RND_MIN_INF: Rounding to minus infinity.
- AP_RND_INF: Rounding to infinity.
- AP_RND_CONV: Convergent rounding.
- AP_TRN: Truncation. This is the default value when Q is not specified.
- AP_TRN_ZERO: Truncation to zero.
- o O = Overflow mode. Only predefined enumerated values can be used to specify O. The accepted values are:
 - AP_SAT: Saturation.
 - AP_SAT_ZERO: Saturation to zero.
 - AP_SAT_SYM: Symmetrical saturation.
 - AP_WRAP: Wrap-around. This is the default value when O is not specified.
 - AP_WRAP_SM: Sign magnitude wrap-around.
- o N = The number of saturation bits in the overflow WRAP modes. Only a constant integer expression can be used as the parameter value. The default value is zero.



TIP: The `ap_fixed` and `ap_ufixed` data types permit shorthand definition, with only `W` and `I` being required, and other parameters assigned default values. However, to define `Q` or `N`, you must also specify the parameters before those, even if you just specify the default values.

In the example code below, the `ap_fixed` type is used to define a signed 18-bit variable with 6 bits representing the integer value above the binary point, and by implication, 12 bits representing the fractional value below the binary point. The quantization mode is set to round to plus infinity (`AP_RND`). Because the overflow mode and saturation bits are not specified, the defaults `AP_WRAP` and `O` are used.

```
#include <ap_fixed.h>
...
    ap_fixed<18,6,AP_RND> my_type;
...
```

When performing calculations where the variables have different numbers of bits (`W`), or different precision (`I`), the binary point is automatically aligned. For more information on using fixed-point data types, see *C++ Arbitrary Precision Fixed-Point Types* in the *Vivado Design Suite User Guide: High-Level Synthesis* ([UG902](#)).

Interfaces

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

Memory Mapped Interfaces

The main data processed by the kernel, often in a large volume, should be transferred through the global memory banks on the FPGA board. The host machine transfers a large chunk of data to one or more global memory bank(s). The kernel accesses the data from those global memory banks, preferably in burst. After the kernel finishes the computation, the resulting data is transferred back to the host machine through the global memory banks.

When writing the kernel interface description, pragmas are used to denote the interfaces coming to and from the global memory banks.

Kernel Interfaces and Memory Banks

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

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

In the example above, there are three large data interfaces. The two inputs are `pixel` and `weights` and one output, `out`. These inputs and outputs connected to the global memory bank are specified in C code by using `HLS INTERFACE m_axi` pragmas as shown above.

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

Sharing ports helps saves FPGA resources, but can limit the performance of the kernel because all the memory transfers have to go through a single port. The bandwidth and throughput of the kernel can be increased by creating multiple ports, using different bundle names, to connect to multiple memory banks.

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

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

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



IMPORTANT! You should specify `bundle=` names using all lowercase characters to be able to assign it to a specific memory bank using the `connectivity.sp` option.

The HLS INTERFACE pragma is used during `v++` compilation, resulting in a compiled kernel object (`.xo`) that has two separate AXI interfaces, `m_axi_gmem` and `m_axi_gmem1`, that can be connected to global memory as needed. During `v++` linking, the separate interfaces can be mapped to different global memory banks using the `connectivity.sp` option in a config file, as described in [Mapping Kernel Ports to Global Memory](#).

Memory Interface Width Considerations

The maximum data width from the global memory to and from the kernel is 512 bits. To maximize the data transfer rate, using this full data width is recommended. The kernel code should be modified to take advantage of the full bit width.

Because a native integer type is used in the prior example, the full data transfer bandwidth is not used. As discussed previously in [Data Types](#), arbitrary precision data types `ap_int` or `ap_uint` can be used to achieve bit-accurate data width for this purpose.

```
void cnn( ap_uint<512> *pixel, // Input pixel
         int *weights, // Input Weight Matrix
         ap_uint<512> *out, // Output pixel
         ... // Other input or output ports

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

The example above shows the output (`out`) interface using the `ap_uint` data type to make use of the full transfer width of 512 bits.

The data width of the memory interfaces should be a power of 2. For data width less than 32, use native C data types. Use `ap_int/ap_uint` for data widths greater than 32 and with power of 2 increment.

Reading and Writing by Burst

Accessing the global memory bank interface from the kernel has a large latency, so global memory transfer should be done in burst. To infer the burst, the following pipelined loop coding style is recommended.

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

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

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

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

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

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

    read_function(m_in, inp1); // Read function contains pipelined for loop
                               // to infer burst

    execute_function(in_var1, out_var1, inp1, inp2); // Core compute function

    write_function(out_var1, m_out); // Write function contains pipelined for
    loop
                                     // to infer burst
}
```

Scalar Inputs

Scalar inputs are typically control variables that are directly loaded from the host machine. They can be thought of as programming data or parameters under which the main kernel computation takes place. These kernel inputs are write-only from the host side. These interfaces are specified in the kernel code as shown below:

```
void process_image(int *input, int *output, int width, int height) {
#pragma HLS INTERFACE s_axilite port=width bundle=control
#pragma HLS INTERFACE s_axilite port=height bundle=control
}
```

In the example, there are two scalar inputs specify the image `width` and `height`. These inputs are specified using the `#pragma HLS INTERFACE s_axilite`. These data inputs come to the kernel directly from the host machine and not using global memory bank.



IMPORTANT! Currently, the Vitis core development kit supports only one control interface bundle for each kernel. Therefore, the `bundle=` name should be same for all scalar data inputs, and the function `return`. In the preceding example, `bundle=control` is used for all scalar inputs.

Enabling Host-to-Kernel Dataflow

As discussed in the [Temporal Data Parallelism: Host-to-Kernel Dataflow](#), if a kernel is capable of accepting more data while it is still operating on data from the previous transactions, XRT can send the next batch of data. The kernel then works on multiple data sets in parallel at different stages of the algorithm, thus improving performance. However, to support host-to-kernel dataflow, the kernel has to implement the `ap_ctrl_chain` protocol using the [HLS INTERFACE](#) pragma for the function return:

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



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

Loops

Loops are an important aspect for a high performance accelerator. Generally, loops are either pipelined or unrolled to take advantage of the highly distributed and parallel FPGA architecture to provide a performance boost compared to running on a CPU.

By default, loops are neither pipelined nor unrolled. Each iteration of the loop takes at least one clock cycle to execute in hardware. Thinking from the hardware perspective, there is an implicit *wait until clock* for the loop body. The next iteration of a loop only starts when the previous iteration is finished.

Loop Pipelining

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

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



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

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

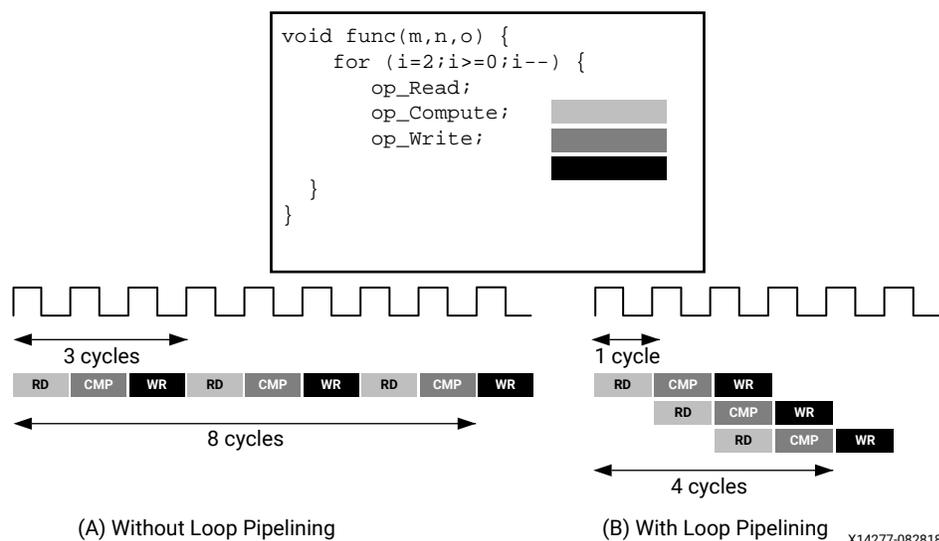
```
vadd: for(int i = 0; i < len; i++) {
    #pragma HLS PIPELINE
    c[i] = a[i] + b[i];
}
```

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

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

The following figure illustrates the difference in execution between pipelined and non-pipelined loops. In this figure, (A) shows the default sequential operation where there are three clock cycles between each input read ($II = 3$), and it requires eight clock cycles before the last output write is performed.

Figure 20: Loop Pipelining



In the pipelined version of the loop shown in (B), a new input sample is read every cycle ($II = 1$) and the final output is written after only four clock cycles: substantially improving both the II and latency while using the same hardware resources.



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

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

Loop Unrolling

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

```
vadd: for(int i = 0; i < 20; i++) {
    #pragma HLS UNROLL
    c[i] = a[i] + b[i];
}
```

In the preceding example, you can see `pragma HLS UNROLL` has been inserted into the body of the loop to instruct the compiler to unroll the loop completely. All 20 iterations of the loop are executed in parallel if that is permitted by any data dependency.



TIP: *Completely unrolling a loop can consume significant device resources, while partially unrolling the loop provides some performance improvement while using fewer hardware resources.*

Partially Unrolled Loop

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

The following code examples illustrates how partially unrolled loops work:

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

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

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

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



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

Loop Dependencies

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

```
Minim_Loop: while (a != b) {
    if (a > b)
        a -= b;
    else
        b -= a;
}
```

This loop cannot be pipelined. The next iteration of the loop cannot begin until the previous iteration ends.

Dealing with various types of dependencies with the Vitis compiler is an extensive topic requiring a detailed understanding of the high-level synthesis procedures underlying the compiler. Refer to the *Vivado Design Suite User Guide: High-Level Synthesis* ([UG902](#)) for more information on "Dependencies with Vivado HLS."

Nested Loops

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

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

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

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

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

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

The above example shows a nested loop structure with two loops that performs some computation on incoming pixel data. In most cases, you want to process a pixel in every cycle, hence `PIPELINE` is applied to the nested loop body structure. The compiler is able to flatten the nested loop structure in the example because it is a perfect nested loop.

The nested loop in the preceding example contains no logic between the two loop declarations. No logic is placed between the `ROW_LOOP` and `COL_LOOP`; all of the processing logic is inside the `COL_LOOP`. Also, both the loops have a fixed number of iterations. These two criteria help the v++ compiler flatten the loops and apply the `PIPELINE` constraint.



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

Sequential Loops

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

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

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

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

This code example is using two separate loops for reading and writing from/to the memory input/output interfaces to infer burst read/write.

By default, these loops are executed sequentially without any overlap. First, the `mem_rd` loop finishes reading all the input data before the `compute` loop starts its operation. Similarly, the `compute` loop finishes processing the data before the `mem_wr` loop starts to write the data. However, the execution of these loops can be overlapped, allowing the `compute` (or `mem_wr`) loop to start as soon as there is enough data available to feed its operation, before the `mem_rd` (or `compute`) loop has finished processing its data.

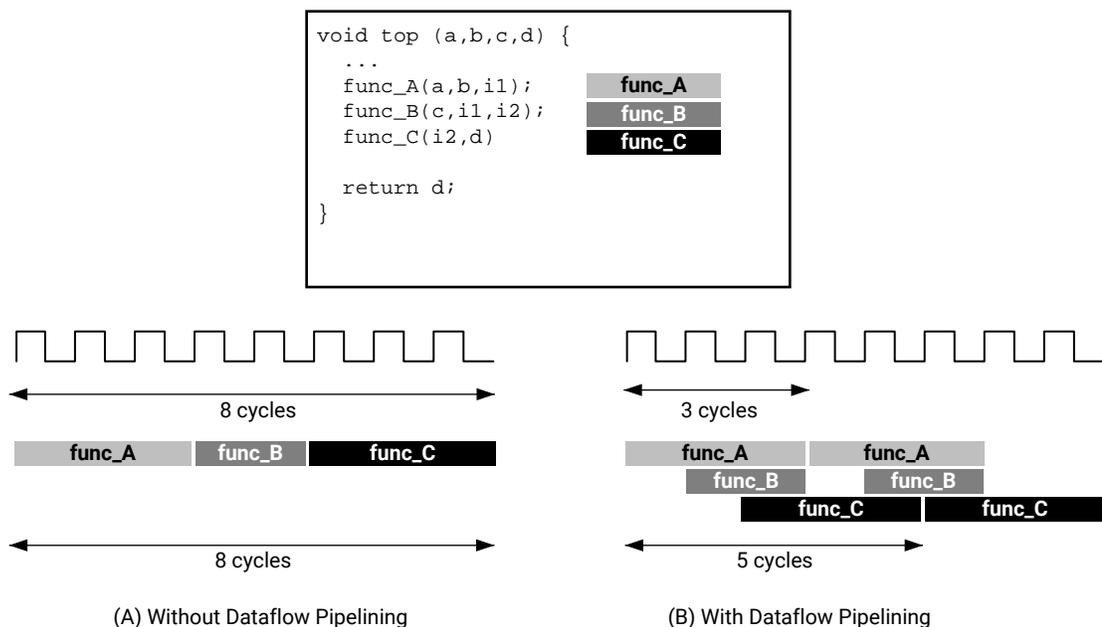
The loop execution can be overlapped using dataflow optimization as described in [Dataflow Optimization](#).

Dataflow Optimization

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

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

Figure 21: Dataflow Optimization



X14266-083118

Dataflow Coding Example

In the dataflow coding example you should notice the following:

1. The `HLS DATAFLOW` pragma is applied to instruct the compiler to enable dataflow optimization. This is not a data mover, which deals with interfacing between the PS and PL, but instead addresses how the data flows through the accelerator.
2. The `stream` class is used as a data transferring channel between each of the functions in the dataflow region.



TIP: The `stream` class infers a first-in first-out (FIFO) memory circuit in the programmable logic. This memory circuit, which acts as a queue in software programming, provides data-level synchronization between the functions and achieves better performance. For additional details on the `hls::stream` class, see the Vivado Design Suite User Guide: High-Level Synthesis (UG902).

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

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

Canonical Forms of Dataflow Optimization

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

- Functions: The canonical form coding guideline for dataflow inside a function specifies:
 1. Use only the following types of variables inside the dataflow region:
 - a. Local non-static scalar/array/pointer variables.
 - b. Local static `hls::stream` variables.
 2. Function calls transfer data only in the forward direction.
 3. Array or `hls::stream` should have only one producer function and one consumer function.
 4. The function arguments (variables coming from outside the dataflow region) should only be read, or written, not both. If performing both read and write on the same function argument then read should happen before write.
 5. The local variables (those that are transferring data in forward direction) should be written before being read.

The following code example illustrates the canonical form for dataflow within a function. Note that the first function (`func1`) reads the inputs and the last function (`func3`) writes the outputs. Also note that one function creates output values that are passed to the next function as input parameters.

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

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

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

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

Troubleshooting Dataflow

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

1. Single producer-consumer violations.
2. Bypassing tasks.
3. Feedback between tasks.
4. Conditional execution of tasks.
5. Loops with multiple exit conditions or conditions defined within the loop.

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

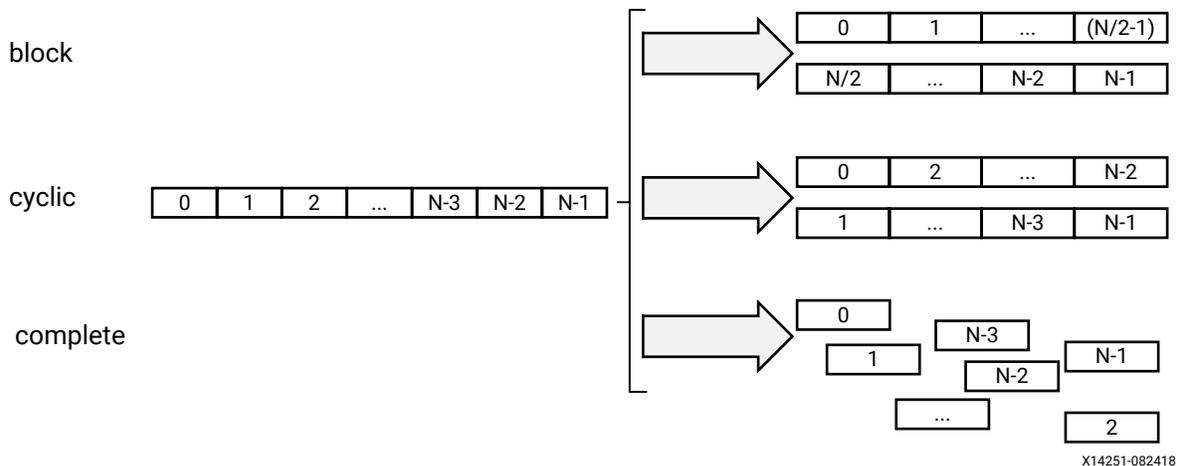
Array Configuration

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

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

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

Figure 22: Partitioning Arrays

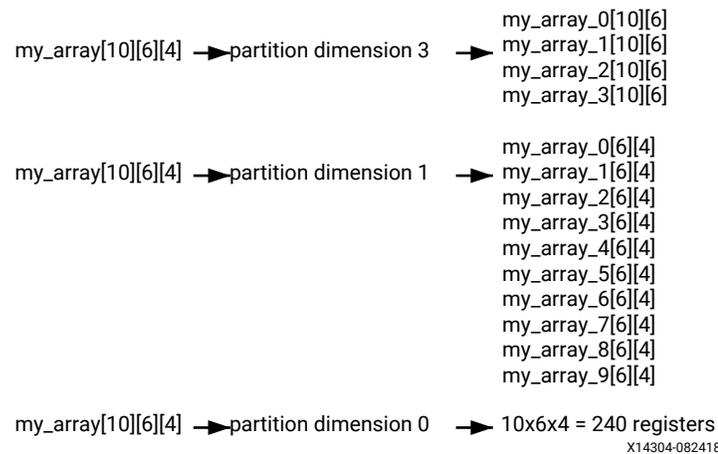


For `block` and `cyclic` partitioning, the `factor` option specifies the number of arrays that are created. In the preceding figure, a `factor` of 2 is used to split the array into two smaller arrays. If the number of elements in the array is not an integer multiple of the `factor`, the later arrays will have fewer elements.

When partitioning multi-dimensional arrays, the `dimension` option is used to specify which dimension is partitioned. The following figure shows how the `dimension` option is used to partition the following example code in three different ways:

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

Figure 23: Partitioning the Dimensions of an Array



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

The Importance of Careful Partitioning

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



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

Choosing a Specific Dimension to Partition

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

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

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

Due to the PIPELINE pragma, the ROW_WISE and COL_WISE loop is flattened together and COMPUTE_LOOP is fully unrolled. To concurrently execute each iteration (k) of the COMPUTE_LOOP, the code must access each column of matrix A and each row of matrix B in parallel. Therefore, the matrix A should be split in the second dimension, and matrix B should be split in the first dimension.

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

Choosing Between Cyclic and Block Partitions

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

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

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

In this version of the code, A and B are now one-dimensional arrays. To access each column of matrix A and each row of matrix B in parallel, cyclic and block partitions are used as shown in the above example. To access each column of matrix A in parallel, `cyclic` partitioning is applied with the `factor` specified as the row size, in this case 64. Similarly, to access each row of matrix B in parallel, `block` partitioning is applied with the `factor` specified as the column size, or 64.

Minimizing Array Accesses with Caching

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

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

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

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

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



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

Function Inlining

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

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

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

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

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

Summary

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

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

9. Minimize the array access by using local cache to improve kernel performance.
10. Consider inlining the function, specifically inside the pipelined region. Functions inside the dataflow should not be inlined.

RTL Kernels

As mentioned in [FPGA Binary Build Process](#), each hardware kernel in the Vitis core development kit is independently compiled to a Xilinx object (.xo) file. These files can be combined into application projects for linking into the FPGA executable (xclbin). This includes the ability to package existing RTL IP from the Vivado Design Suite for use in Vitis applications.

Many hardware engineers have existing RTL IP (including Vivado® IP integrator based designs), or prefer implementing a kernel in RTL and developing it using the Vivado tools. While the Vitis core development kit supports the use of packaged RTL designs, they must adhere to the software and hardware requirements to be used within the accelerated application development flow and runtime library.

Requirements of an RTL Kernel

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

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 Vitis core development kit execution model, an RTL kernel must adhere to the requirements described in [Kernel Requirements](#). The RTL kernel must have at least one clock interface port to supply a clock to the kernel logic. The various interface requirements are summarized in the following table.



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

Table 2: RTL Kernel Interface and Port Requirements

Port or Interface	Description	Comment
ap_clk	Primary clock input port	<ul style="list-style-type: none"> • Name must be exact. • Required port.
ap_clk_2	Secondary optional clock input port	<ul style="list-style-type: none"> • Name must be exact. • Optional port.

Table 2: RTL Kernel Interface and Port Requirements (cont'd)

Port or Interface	Description	Comment
ap_rst_n	Primary active-Low reset input port	<ul style="list-style-type: none"> Name must be exact. Optional port. This signal should be internally pipelined to improve timing. This signal is driven by a synchronous reset in the ap_clk clock domain.
ap_rst_n_2	secondary optional active-Low reset input	<ul style="list-style-type: none"> Name must be exact. Optional port. This signal should be internally pipelined to improve timing. This signal is driven by a synchronous reset in the ap_clk_2 clock domain.
interrupt	Active-High interrupt.	<ul style="list-style-type: none"> Name must be exact. Optional port.
s_axi_control	One (and only one) AXI4-Lite slave control interface	<ul style="list-style-type: none"> Name must be exact; case sensitive. Required port.
AXI4_MASTER	One or more AXI4 master interfaces for global memory access	<ul style="list-style-type: none"> All AXI4 master interfaces must have 64-bit addresses. The RTL kernel developer is responsible for partitioning global memory spaces. Each partition in the global memory becomes a kernel argument. The memory offset for each partition must be set by a control register programmable via the AXI4-Lite slave interface. AXI4 masters must not use Wrap or Fixed burst types and must not use narrow (sub-size) bursts meaning AxSIZE should match the width of the AXI data bus. Any user logic or RTL code that does not conform to the requirements above, must be wrapped or bridged to satisfy these requirements.

Kernel Software Requirements

RTL kernels have the same software interface model as C/C++ and OpenCL kernels. They are seen by the host program as functions with a void return value, pointer arguments, and scalar arguments.

The Vitis core development kit execution model dictates the following:

- Scalar arguments are directly written to the kernel through the AXI4-Lite slave interface.
- Pointer arguments are transferred from the host program to/from memory, and the RTL kernel reads/writes the data in memory through one or more AXI4 memory mapped interfaces.
- Kernels are controlled by the host program 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 will operate in this manner.

The following table outlines the required register map such that a kernel can be used within the Vitis IDE. 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 3: 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 4: Control (0x0)

Bit	Name	Description
0	<code>ap_start</code>	Asserted when kernel can start processing data. Cleared on handshake with <code>ap_done</code> being asserted.
1	<code>ap_done</code>	Asserted when kernel has completed operation. Cleared on read.
2	<code>ap_idle</code>	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 5: 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 6: 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 7: 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.

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

If adding an `interrupt` port to the RTL kernel, the `kernel.xml` file needs to include this information. The `kernel.xml` is generated automatically when using the RTL Kernel Wizard. 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 for the RTL kernel, and the associated `compent.xml` and `kernel.xml` files.

RTL Kernel Development Flow

This section explains the three step process to creating RTL kernels for the Vitis core development kit, which includes:

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

A packaged RTL kernel is delivered as a Xilinx object 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 combined with other kernels, and linked with the target platform and built for hardware or hardware emulation flows.



TIP: An RTL kernel is not suited for software emulation unless you provide a C-model for the kernel.

Packaging the RTL Code as Vivado IP

RTL kernels must be packaged as a Vivado IP suitable for use by the IP integrator. For details on IP packaging in the Vivado tool, see the *Vivado Design Suite User Guide: Creating and Packaging Custom IP* ([UG1118](#)).

The required interfaces for the RTL kernel must be packaged as follows:

- 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_clk_2` is only required when the RTL kernel has two clocks).
- `ap_rst_n` and `ap_rst_n_2` must be packaged as active-Low reset interfaces (when the RTL kernel has a reset).
- `ap_clk` must be 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 IP into a block design in the IP integrator. For information on the tool, refer to *Vivado Design Suite User Guide: Designing IP Subsystems using IP Integrator* (UG994).

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



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 `.tcl` file, add `<: setFileProcessingOrder "late" :>` to the `.tcl` preamble section of the file as follows:

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

2. If constraints are defined in an `.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>tcl/my_ip_constraint.xdc</spirit:name>
  <spirit:userFileType>tcl</spirit:userFileType>
  <spirit:userFileType>USED_IN_implementation</spirit:userFileType>
  <spirit:userFileType>USED_IN_synthesis</spirit:userFileType>
```

```

        <spirit:define>
            <spirit:name>processing_order</spirit:name>
            <spirit:value>late</spirit:value>
        </spirit:define>
    </spirit:file>
    
```

Creating the Kernel Description XML File

An XML kernel description file, called `kernel.xml`, must be created for each RTL kernel so that it can be used in the Vitis application acceleration development flow. The `kernel.xml` file specifies kernel attributes like the register map, and ports that are needed by the runtime and Vitis 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="vitis_kernel_wizard_0" language="ip_c"
    vlnv="mycompany.com:kernel:vitis_kernel_wizard_0:1.0"
    attributes="" preferredWorkGroupSizeMultiple="0" workGroupSize="1"
    interrupt="true">
    <ports>
      <port name="s_axi_control" mode="slave" range="0x1000" dataWidth="32"
        portType="addressable" base="0x0"/>
      <port name="m00_axi" mode="master" range="0xFFFFFFFFFFFFFFFF"
        dataWidth="512" portType="addressable"
        base="0x0"/>
    </ports>
    <args>
      <arg name="axi00_ptr0" addressQualifier="1" id="0" port="m00_axi"
        size="0x8" offset="0x010" type="int*"
        hostOffset="0x0" hostSize="0x8"/>
    </args>
  </kernel>
</root>
    
```

Note: The `kernel.xml` file can be created automatically using the RTL Kernel Wizard to specify the interface specification of your RTL kernel. Refer to [RTL Kernel Wizard](#) for more information.

The following table describes the format of the `kernel.xml` in detail:

Table 8: Kernel XML File Content

Tag	Attribute	Description
<root>	versionMajor	Set to 1 for the current release of Vitis software platform.
	versionMinor	Set to 6 for the current release of Vitis software platform.

Table 8: Kernel XML File Content (cont'd)

Tag	Attribute	Description
<kernel>	name	Kernel name
	language	Always set it to <code>ip_c</code> for RTL kernels.
	vlnv	Must match the vendor, library, name, and version attributes in the <code>component.xml</code> of an IP. For example, if <code>component.xml</code> has the following tags: <pre><spirit:vendor>xilinx.com</spirit:vendor> <spirit:library>hls</spirit:library> <spirit:name>test_sincos</spirit:name> <spirit:version>1.0</spirit:version></pre> The <code>vlnv</code> attribute in kernel XML must be set to: <code>xilinx.com:hls:test_sincos:1.0</code>
	attributes	Reserved. Set it to empty string: ""
	preferredWorkGroupSizeMultiple	Reserved. Set it to 0.
	workGroupSize	Reserved. Set it to 1.
	interrupt	Set to "true" (<code>interrupt="true"</code>) if the RTL kernel has an interrupt, otherwise omit.
	hwControlProtocol	Specifies the control protocol for the RTL kernel: <ul style="list-style-type: none"> <code>ap_ctrl_hs</code>: The default control protocol for RTL kernels. <code>ap_ctrl_chain</code>: The control protocol for chained kernels that support dataflow. This adds <code>ap_continue</code> to the control registers to enable <code>ap_done/ap_continue</code> completion acknowledgment. <code>ap_ctrl_none</code>: Control protocol (none) applied for continuously operating kernels that have no need for start or done. Refer to Free-running Kernel for details.

Table 8: Kernel XML File Content (cont'd)

Tag	Attribute	Description
<port>	name	Specifies the port name. ★ IMPORTANT! The AXI4-Lite interface must be named <i>S_AXI_CONTROL</i> .
	mode	At least one AXI4 master port and one AXI4-Lite slave control port are required. AXI4-Stream ports can be specified to stream data between kernels. <ul style="list-style-type: none"> For AXI4 master port, set it to "master." For AXI4 slave port, set it to "slave." For AXI4-Stream master port, set it to "write_only." For AXI4-Stream slave port, set it "read_only."
	range	The range of the address space for the port.
	dataWidth	The width of the data that goes through the port, default is 32-bits.
	portType	Indicate whether or not the port is addressable or streaming. <ul style="list-style-type: none"> For AXI4 master and slave ports, set it to "addressable." For AXI4-Stream ports, set it to "stream."
	base	For AXI4 master and slave ports, set to 0x0. This tag is not applicable to AXI4-Stream ports.

Table 8: Kernel XML File Content (cont'd)

Tag	Attribute	Description
<arg>	name	Specifies the kernel software argument name.
	addressQualifier	Valid values: 0: Scalar kernel input argument 1: global memory 2: local memory 3: constant memory 4: pipe
	id	Only applicable for AXI4 master and slave ports. The ID needs to be sequential. It is used to determine the order of kernel arguments. Not applicable for AXI4-Stream ports.
	port	Specifies the <port> name to which the <code>arg</code> is connected.
	size	Size of the argument in bytes. The default is 4 bytes.
	offset	Indicates the register memory address.
	type	The C data type of the argument. For example <code>uint*</code> , <code>int*</code> , <code>float*</code> .
	hostOffset	Reserved. Set to <code>0x0</code> .
	hostSize	Size of the argument. The default is 4 bytes.
	memSize	For AXI4-Stream ports, <code>memSize</code> sets the depth of the created FIFO.  TIP: Not applicable to AXI4 ports.
The following tags specify additional tags for AXI4-Stream ports. They do not apply to AXI4 ports.		
<pipe>	For each pipe in the compute unit, the compiler inserts a FIFO for buffering the data. The pipe tag describes configuration of the FIFO.	
	name	This specifies the name for the FIFO inserted for the AXI4-Stream port. This name must be unique among all pipes used in the same compute unit.
	width	This specifies the width of FIFO in bytes. For example, <code>0x4</code> 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.

Creating the XO File from the RTL Kernel

The final step is to package the RTL IP and its `kernel.xml` file together into a Xilinx object file (`.xo`) so the kernel can be used in the Vitis core development kit. This is done using the `package_xo` Tcl command in the Vivado Design Suite.

The following example packages an RTL kernel IP named `test_sincos` and the associated `kernel.xml` file into an object file named `test.xo`. The `package_xo` command is run from within the Vivado tool:

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

The output of the `package_xo` command is the `test.xo` file, that can be added as a source file to the `v++` command as discussed in [Chapter 3: Building and Running the Application](#), or added to an application project as discussed in [Chapter 6: Using the Vitis IDE](#).

RTL Kernel Wizard

The RTL kernel wizard automates some of the steps you need to take to ensure that the RTL IP is packaged into a kernel object (`.xo`) that can be used by the Vitis compiler. The RTL Kernel wizard:

- Steps you through the process of specifying the interface requirements for your RTL kernel, and generates a top-level RTL wrapper based on the provided information.
- Automatically generates an AXI4-Lite interface module including the control logic and register file, included in the top level wrapper.
- Includes an example kernel IP module in the top-level wrapper that you can replace with your own RTL IP design, after ensuring correct connectivity between your RTL IP and the wrapper.
- Automatically generates a `kernel.xml` file to match the kernel specification from the wizard.
- Generates a simple simulation test bench for the generated RTL kernel wrapper.
- Generates an example host program to run and debug the RTL kernel.

The RTL Kernel wizard can be accessed from the Vitis IDE, or from the Vivado IP catalog. In either case it creates a Vivado project containing an example design to act as a template for defining your own RTL kernel.

The example design consists of a simple RTL IP adder, called VADD, that you can use to guide you through the process of mapping your own RTL IP into the generated top-level wrapper. The connections include clock(s), reset(s), `s_axilite` control interface, `m_axi` interfaces, and optionally `axis` streaming interfaces.

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

Launch the RTL Kernel Wizard

The RTL Kernel Wizard can be launched from the Vitis IDE, or from the Vivado IDE.



TIP: Running the wizard from the Vitis IDE automatically imports the generated RTL kernel, and example host code, into the current application project when the process is complete.

To launch the RTL Kernel Wizard from within the Vitis IDE, select the **Xilinx → RTL Kernel Wizard** menu item from an open application project. For details on working with the GUI, refer to [Chapter 6: Using the Vitis IDE](#).

To launch the RTL Kernel Wizard from the Vivado IDE:

1. Create a new Vivado project, select the target platform when choosing a board for the project.
2. In the Flow Navigator, click the **IP catalog** command.
3. Type `RTL Kernel` in the IP catalog search box.
4. Double-click **RTL Kernel Wizard** to launch the wizard.

Using the RTL Kernel Wizard

The RTL Kernel wizard is organized into multiple pages that break down the process of defining an RTL kernel. The pages of the wizard include:

1. [General Settings](#)
2. [Scalars](#)
3. [Global Memory](#)
4. [Streaming Interfaces](#)
5. [Summary](#)

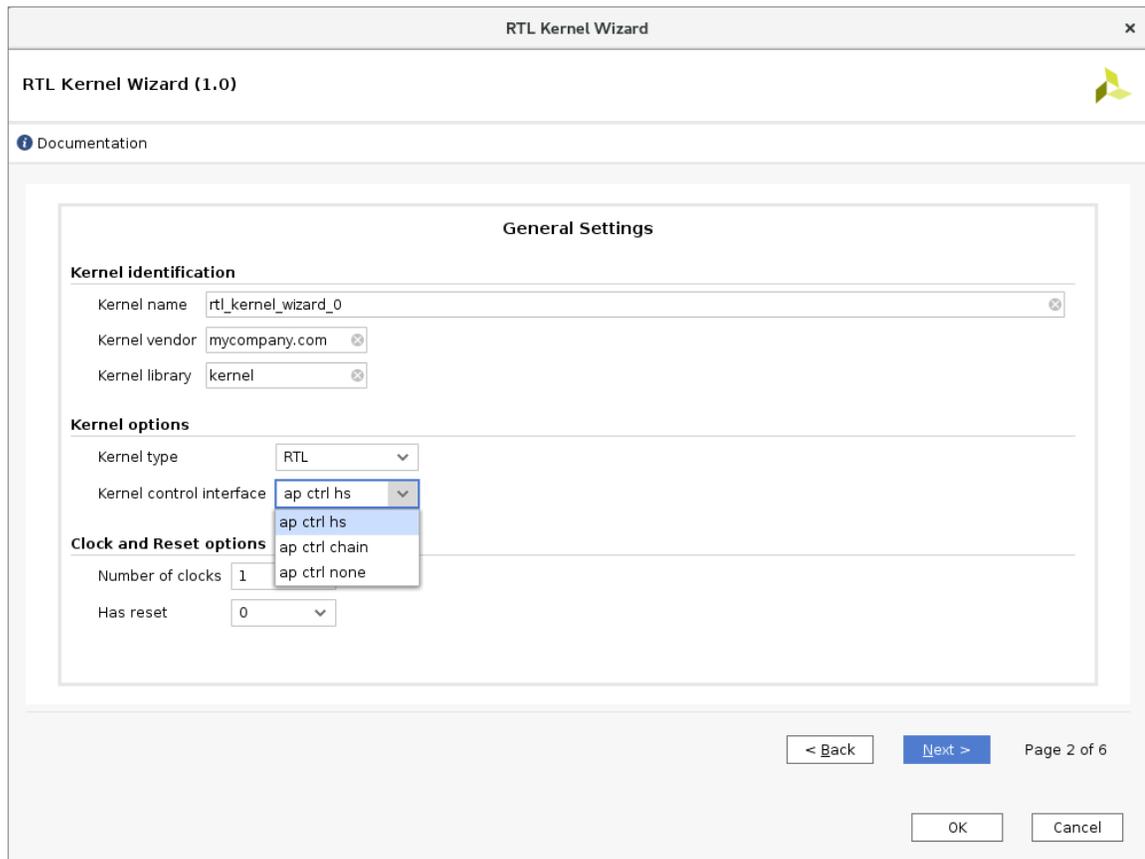
To navigate between pages, click **Next** and **Back** as needed.

To finalize the kernel and build a project based on the kernel specification, click **OK** on the Summary page.

General Settings

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

Figure 24: RTL Kernel Wizard General Settings



The following are three settings in the General Settings tab.

Kernel Identification

- **Kernel name:** The kernel name. This will be the name of the IP, top-level module name, kernel, and C/C++ functional model. This identifier shall conform to C and Verilog identifier naming rules. It must also conform to Vivado IP integrator naming rules, which prohibits underscores except when placed in between alphanumeric characters.
- **Kernel vendor:** The name of the vendor. Used in the Vendor/Library/Name/Version (VLNV) format described in the *Vivado Design Suite User Guide: Designing with IP (UG896)*.
- **Kernel library:** The name of the library. Used in the VLNV. Must conform to the same identifier rules.

Kernel options

- **Kernel type:** The RTL Kernel wizard currently supports two types of kernels: RTL, and Block Design.

- **RTL:** The RTL type kernel consists of a Verilog RTL top-level module with a Verilog control register module and a Verilog kernel example inside the top-level module.
- **Block Design:** The block design type kernel also delivers a Verilog top-level module, but instead it instantiates an IP integrator block diagram inside of the top-level. The block design consists of a MicroBlaze™ subsystem that uses a block RAM exchange memory to emulate the control registers. Example MicroBlaze software is delivered with the project to demonstrate using the MicroBlaze to control the kernel.
- **Kernel control interface:** There are three types of control interfaces available for the RTL kernel. `ap_ctrl_hs`, `ap_ctrl_chain`, and `ap_ctrl_none`. This defines the `hwControlProtocol` for the `<kernel>` tag as described in [Creating the Kernel Description XML File](#).

Clock and Reset Options

- **Number of clocks:** Sets the number of clocks used by the kernel. Every RTL kernel has one primary clock called `ap_clk` and an optional reset called `ap_rst_n`. All AXI interfaces on the kernel are driven with this clock.

When setting **Number of clocks** to 2, a secondary clock and optional reset are provided to be used by the kernel internally. The secondary clock and reset are called `ap_clk_2` and `ap_rst_n_2`. This secondary clock supports independent frequency scaling and is independent from the primary clock. The secondary clock is useful if the kernel clock needs to run at a faster or slower rate than the AXI4 interfaces, which must be clocked on the primary clock.



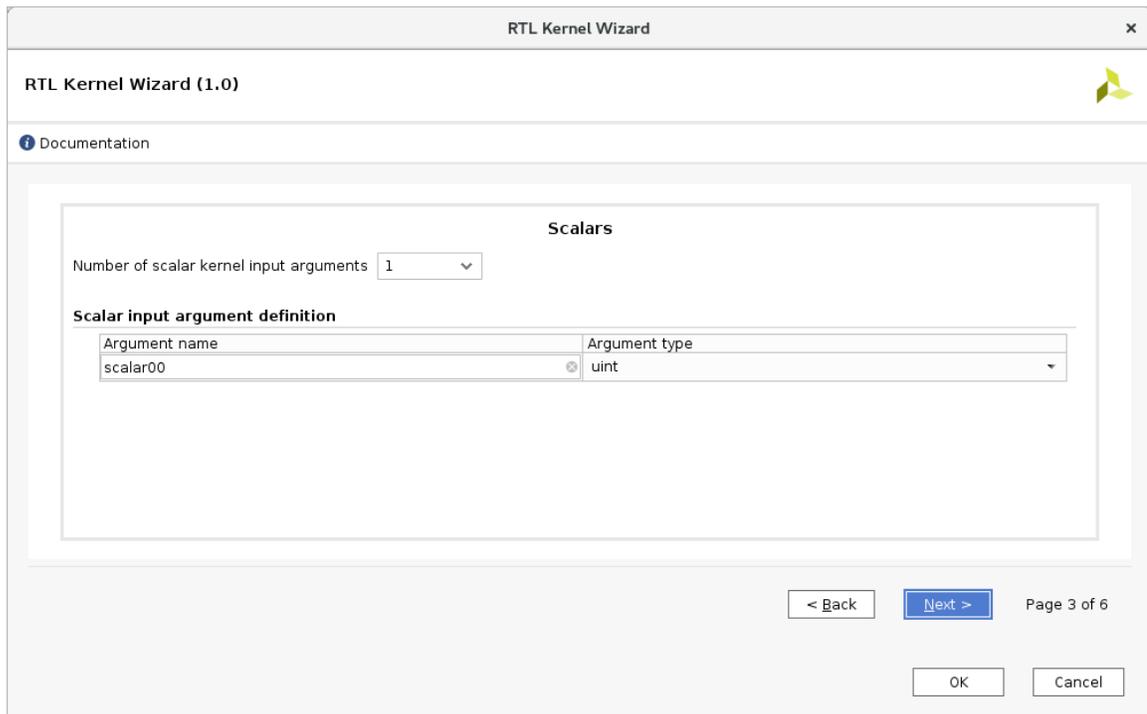
IMPORTANT! *When designing with multiple clocks, proper clock domain crossing techniques must be used to ensure data integrity across all clock frequency scenarios. Refer to [UltraFast Design Methodology Guide for the Vivado Design Suite \(UG949\)](#) for more information.*

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

Scalars

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

Figure 25: Kernel Wizard Scalars



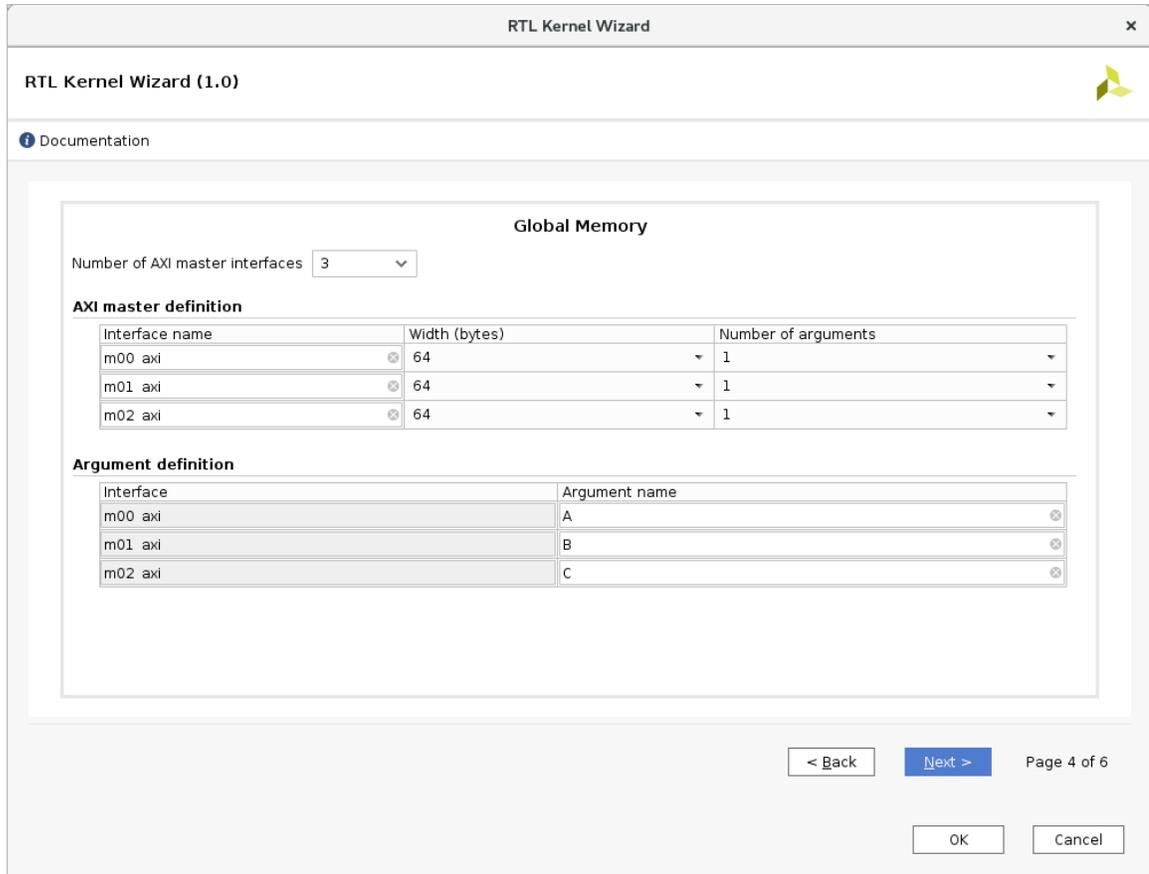
- **Number of scalar kernel input arguments:** Specifies the number of scalar input arguments to pass to the kernel. For each number specified, a table row is generated that allows customization of the argument name and argument type. There is no required minimum number of scalars and the maximum allowed 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 RTL kernel module. The data types available are limited to the ones specified by the [OpenCL C Specification Version 2.0](#) in "6.1.1 Built-in Scalar Data Types" section. The specification provides the associated bit-widths for each data type. The RTL wizard reserves 64 bits for all scalars in the register map regardless of their argument type. If the argument type is 32 bits or less, the RTL Wizard sets the upper 32 bits (of the 64 bits allocated) as a reserved address location. Data types that represent a bit width greater than 32 bits require two write operations to the control registers.

Global Memory

Figure 26: Global Memory



RTL Kernel Wizard (1.0)

Documentation

Global Memory

Number of AXI master interfaces: 3

AXI master definition

Interface name	Width (bytes)	Number of arguments
m00_axi	64	1
m01_axi	64	1
m02_axi	64	1

Argument definition

Interface	Argument name
m00_axi	A
m01_axi	B
m02_axi	C

< Back Next > Page 4 of 6

OK Cancel

Global memory is accessed by the kernel through AXI4 master interfaces. Each AXI4 interface operates independently of each other, and each AXI4 interface can be connected to one or more memory controllers to off-chip memory such as DDR4. Global memory is primarily used to pass large data sets to and from the kernel from the host. It can also be used to pass data between kernels. For recommendations on how to design these interfaces for optimal performance, see [Memory Performance Optimizations for AXI4 Interface](#).



TIP: For each interface, the RTL Kernel wizard generates example AXI master logic in the top-level wrapper to provide a starting point that can be discarded if not needed.

- Number of AXI master interfaces:** Specify the number of interfaces present on the kernel. The maximum is 16 interfaces. For each interface, you can customize an interface name, data width, and the number of associated arguments. Each interface contains all read and write channels. The default names proposed by the RTL kernel wizard are `m00_axi` and `m01_axi`. If not changed, these names will have to be used when assigning an interface to global memory as described in [Mapping Kernel Ports to Global Memory](#).

AXI master definition (table columns)

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

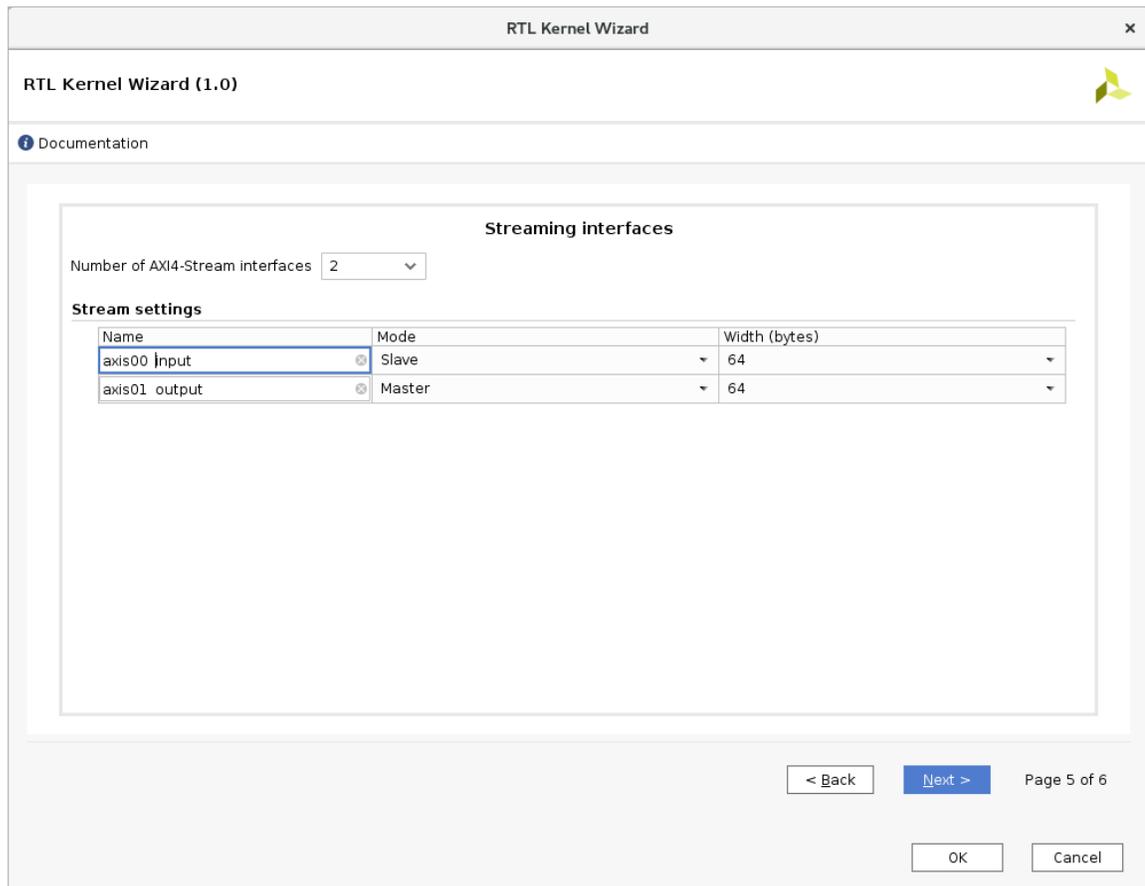
Argument definition

- **Interface:** Specifies the name of the AXI Interface. This value is copied from the interface name defined in the table, and cannot be modified here.
- **Argument name:** Specifies the name of the pointer argument as it appears on the function prototype signature. Each argument is assigned an ID value. This ID value is used to access the argument from the host software as described in [Host Application](#). The ID value assignments can be found on the summary page of this wizard. To ensure maximum compatibility, the argument name follows the same identifier rules as the kernel name. The argument name is used in the generated RTL kernel control register module as an output signal.

Streaming Interfaces

The streaming interfaces page allows configuration of AXI4-Stream interfaces on the kernel. Streaming interfaces are only available on select platforms and if the chosen platform does not support streaming, then the page does not appear. Streaming interfaces are used for direct host-to-kernel and kernel-to-host communication, as well as continuously operating kernels as described in [Streaming Connections](#).

Figure 27: 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 whether the interface is a master or slave interface. An AXI4-Stream slave interface is a read-only interface, and the RTL kernel can be sent data with the `clWriteStream` API from the host program. An AXI4-Stream master interface is a write-only interface, and the host program can receive data through the interface with the `clReadStream` API.
- Width (bytes):** Specifies the `TDATA` width (in bytes) of the AXI4-Stream interface. This interface width is limited to 1 to 64 bytes in powers of 2.

The streaming interface uses the `TDATA/TKEEP/TLAST` signals of the AXI4-Stream protocol. Stream transactions consists of a series of transfers where the final transfer is terminated with the assertion of the `TLAST` signal. Stream transfers must adhere to the following:

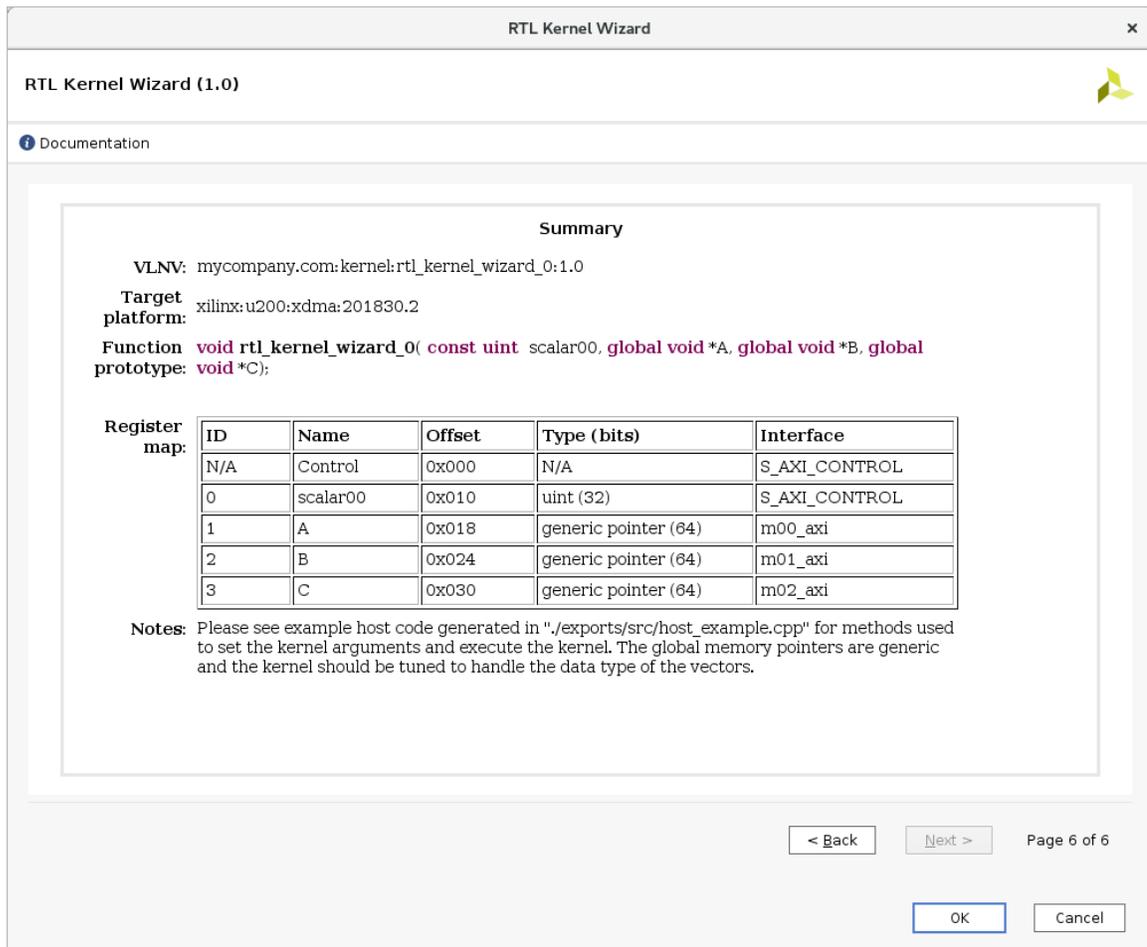
- AXI4-Stream transfer occurs when `TVALID/TREADY` are both asserted.

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

Summary

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

Figure 28: Kernel Wizard Summary



Click **OK** to generate the top-level wrapper for the RTL kernel, the VADD temporary RTL kernel IP, the `kernel.xml` file, the simulation test bench, and the example `host.cpp` code. After these files are created, the RTL Kernel wizard opens a project in the Vivado Design Suite to let you complete kernel development.

Using the RTL Kernel Project in Vivado IDE

If you launched the RTL Kernel wizard from the Vitis IDE, after clicking **OK** on the Summary page, the Vivado Design Suite open with an example IP project to let you complete your RTL kernel code.

If you launched the RTL Kernel wizard from within the Vivado IP catalog, after clicking **OK** on the Summary page, an RTL Kernel Wizard IP is instantiated into your current project. From there you must take the following steps:

1. When the Generate Output Products dialog box appears, click **Skip** to close it.

2. Right-click the `<kernel_name>.xci` file that is added to the Sources view, and select **Open IP Example Design**.
3. In the Open Example Design dialog box, specify the **Example project directory**, or accept the default value, and click **OK**.



TIP: An example project is created for the RTL kernel IP. This example IP project is the same as the example project created if you launch the RTL Kernel wizard from the Vitis IDE, and is where you will complete the development work for your kernel.

4. You can now close the original Vivado project from which you launched the RTL Kernel wizard.

Depending on the **Kernel Type** you selected for the kernel options, the example IP project is populated with a top-level RTL kernel file that contains either a Verilog example and control registers as described in [RTL Type Kernel Project](#), or an instantiated IP integrator block design as described in [Block Design Type Kernel Project](#). The top-level Verilog file contains the expected input/output signals and parameters. These top-level ports are matched to the kernel specification file (`kernel.xml`) and can be combined with your RTL code, or /block design, to complete the RTL kernel.

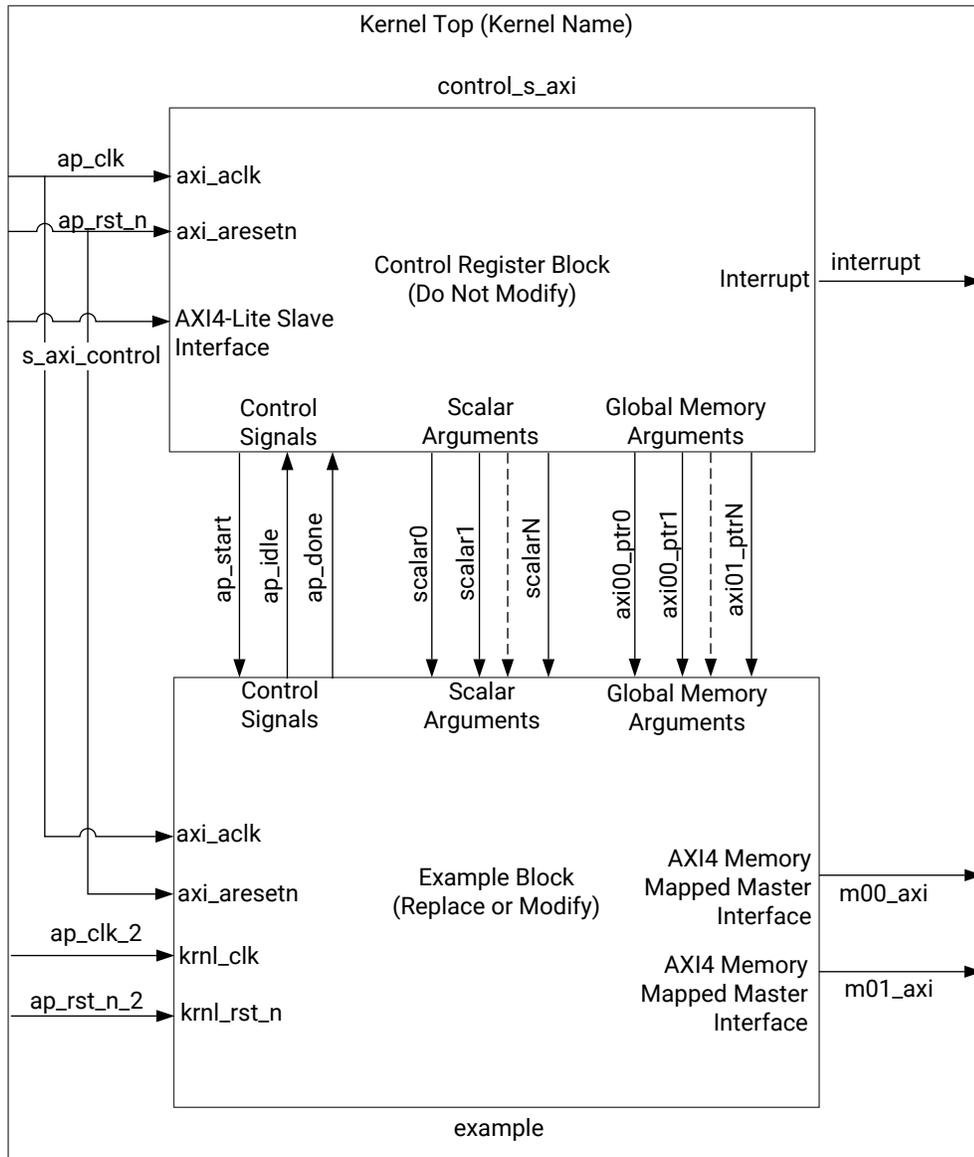
The AXI4 interfaces defined in the top-level file contain a minimum subset of AXI4 signals required to generate an efficient, high throughput interface. Signals that are not present inherit optimized defaults when connected to the rest of the AXI system. These optimized defaults allow the system to omit AXI features that are not required, saving area and reducing complexity. If your RTL code or block design contains AXI signals that were omitted, you can add these signals to the ports in the top-level RTL kernel file, and the IP packager will adapt to them appropriately.

The next step in the process customizes the contents of the kernel and then packages those contents into a Xilinx Object (`xo`) file.

RTL Type Kernel Project

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

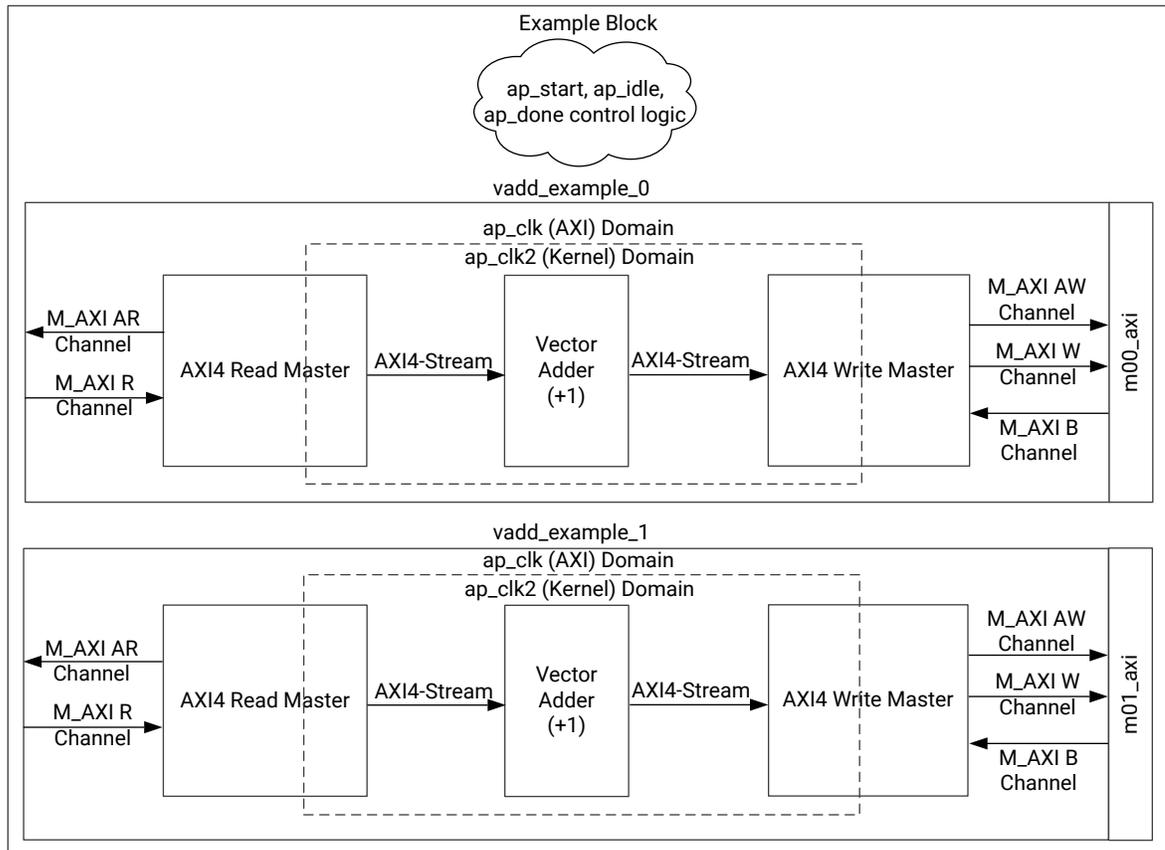
Figure 29: Kernel Type RTL Top



X22079-011019

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

Figure 30: Kernel Type RTL Example



The following table describes some important files in the example IP project, relative to the root of the Vivado project for the kernel, where `<kernel_name>` is the name of the kernel you specified in the RTL Kernel wizard.

Table 9: RTL Kernel Wizard Source and Test Bench File

Filename	Description	Delivered with Kernel Type
<code><kernel_name>_ex.xpr</code>	Vivado project file	All
imports directory		
<code><kernel_name>.v</code>	Kernel top-level module	All
<code><kernel_name>_control_s_axi.v</code>	RTL control register module	RTL
<code><kernel_name>_example.sv</code>	RTL example block	RTL
<code><kernel_name>_example_vadd.sv</code>	RTL example AXI4 vector add block	RTL
<code><kernel_name>_example_axi_read_master.sv</code>	RTL example AXI4 read master	RTL
<code><kernel_name>_example_axi_write_master.sv</code>	RTL example AXI4 write master	RTL
<code><kernel_name>_example_adder.sv</code>	RTL example AXI4-Stream adder block	RTL
<code><kernel_name>_example_counter.sv</code>	RTL example counter	RTL

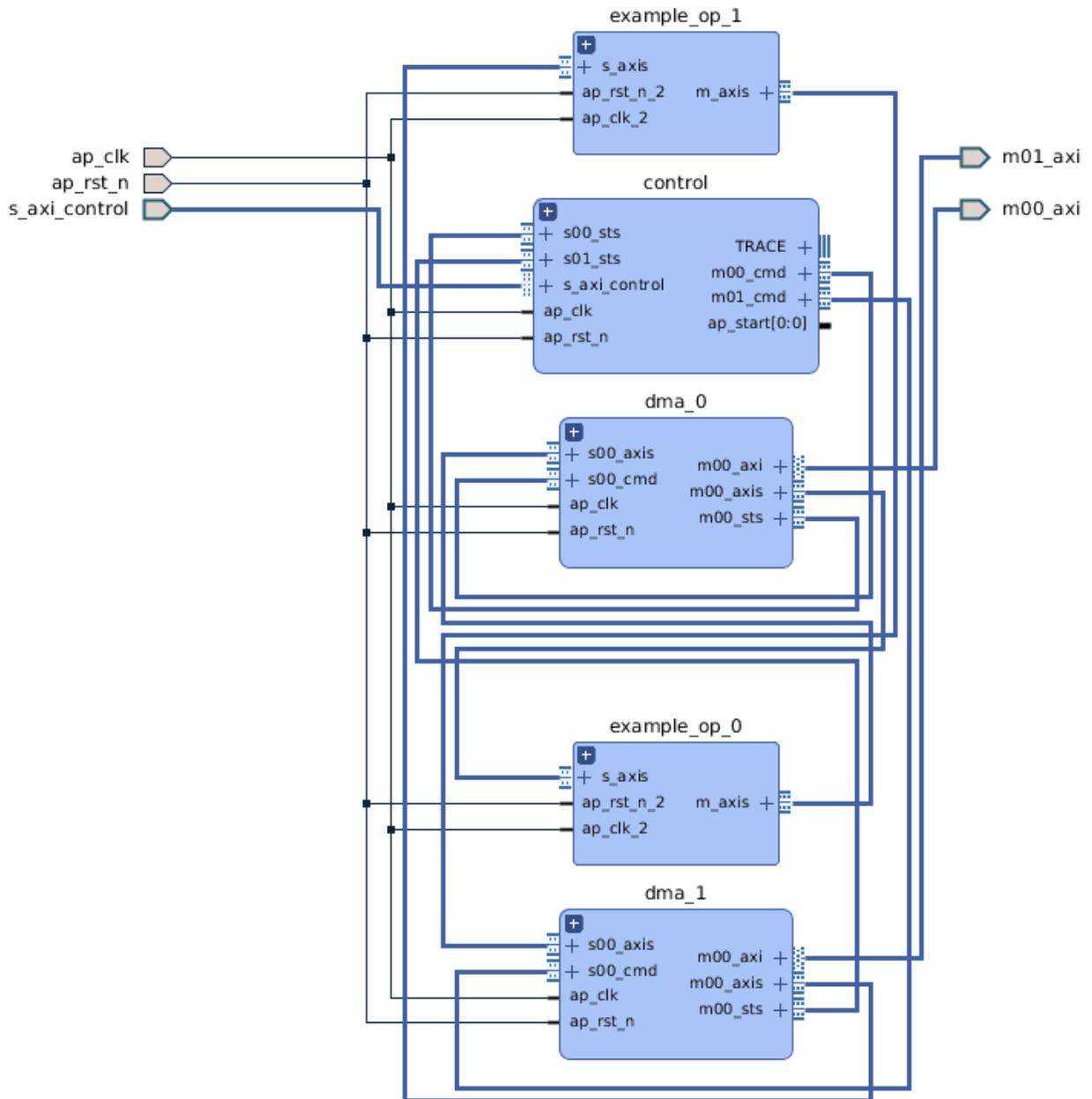
Table 9: RTL Kernel Wizard Source and Test Bench File (cont'd)

Filename	Description	Delivered with Kernel Type
<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
exports directory		
src/host_example.cpp	Host code example	All
makefile	Makefile example	All

Block Design Type Kernel Project

The block design type kernel delivers an IP integrator block design (.bd) at the top-level of the example project. A MicroBlaze processor subsystem is used to sample the control registers and to control the flow of the kernel. The MicroBlaze processor system uses a block RAM as an exchange memory between the host and the kernel instead of a register file.

Figure 31: Block Design Type Kernel



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 a Vitis IDE project to compile and link an ELF file for the MicroBlaze core. This ELF file is loaded into the Vivado kernel project and initialized directly into the MicroBlaze instruction memory.

The following steps can be used to modify the MicroBlaze processor program:

1. If the design has been updated, you might need to run the Export Hardware option. The option can be found in the **File → Export → Export Hardware** menu location. When the export Hardware dialog opens, click **OK**.
2. The core development kit application can now be invoked. Select **Tools → Launch Vitis** from the main menu.
3. When the Vitis IDE opens, click **X** just to the right of the text on the Welcome tab to close the welcome dialog box. This shows an already loaded Vitis IDE project underneath.
4. From the Project Explorer, the source files are under the `<Kernel Name>_control/src` section. Modify these as appropriate.
5. When updates are complete, compile the source by selecting the menu option **Project → Build All → Check for errors/warnings and resolve if necessary**. The ELF file is automatically updated in the IDE.
6. Run simulation to test the updated program and debug if necessary.

Simulation Test Bench

A SystemVerilog test bench is generated for simulating the example IP project. This test bench exercises the RTL kernel to ensure its operation is correct. It is populated with the checker function to verify the `add_one` operation.

This generated test bench can be used as a starting point in verifying the kernel functionality. It writes/reads from the control registers and executes the kernel multiple times while also including a simple reset test. It is also useful for debugging AXI issues, reset issues, bugs during multiple iterations, and kernel functionality. Compared to hardware emulation, it executes a more rigorous test of the hardware corner cases, but does not test the interaction between host code and kernel.

To run a simulation, click **Vivado Flow Navigator → Run Simulation** located on the left hand side of the GUI and select **Run Behavioral Simulation**. If behavioral simulation is working as expected, a post-synthesis functional simulation can be run to ensure that synthesis results are matched with the behavioral model.

Out-of-Context Synthesis

The Vivado kernel project is configured to run synthesis and implementation in out-of-context (OOC) mode. A Xilinx Design Constraints (XDC) file is populated in the design to provide default clock frequencies for this purpose.

You should always synthesize the RTL kernel before packaging it with the `package_xo` command. Running synthesis is useful to determine whether the kernel synthesizes without errors. It also provides estimates of resource utilization and operating frequency. Without pre-synthesizing the RTL kernel you could encounter errors during the `v++` linking process, and it could be much harder to debug the cause.

To run OOC synthesis, click **Run Synthesis** from the **Vivado Flow Navigator → Synthesis** menu.

The synthesized outputs can also be used to package the RTL kernel with a netlist source, instead of RTL source.



IMPORTANT! A block design type kernel must be packaged as a netlist using the `package_xo` command.

Software Model and Host Code Example

A C++ software model of the `add_one` example operation, `<kernel_name>_cmodel.cpp`, is provided in the `./imports` directory. This software model can also be modified to model the function of your kernel. When running `package_xo`, this model can be included with the kernel source files to enable software emulation for the kernel. The hardware emulation and system builds always use the RTL description of the kernel.

In the `./exports/src` directory, an example host program is provided and is called `host_example.cpp`. The host program takes the binary container as an argument to the program. The host code loads the binary as part of the `init` function. The host code instantiates the kernel, allocates the buffers, sets the kernel arguments, executes the kernel, and then collects and checks the results for the example `add_one` function.

For information on using the host program and kernel code in an application, refer to [Creating a Vitis IDE Project](#).

Generate RTL Kernel

After the kernel is designed and tested in the example IP project in the Vivado IDE, the final step is to generate the RTL kernel object file (`.xo`) for use by the Vitis compiler.

Click the **Generate RTL Kernel** command from the **Vivado Flow Navigator** → **Project Manager** menu. The Generate RTL Kernel dialog box opens with three main packaging options:

- 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 `exports` directory of the Vivado kernel project.

At this point, you can close the Vivado kernel project. If the Vivado kernel project was invoked from the Vitis IDE, the example host code called `host_example.cpp` and kernel Xilinx Object (`.xo`) files are automatically imported into the `./src` folder of the application project in the Vitis IDE.

Modifying an Existing RTL Kernel Generated from the Wizard

From the Vitis IDE, you can modify an existing RTL kernel by selecting it from the `./src` folder of an application project where it is in use. Right-click the `.xo` file in the Project Explorer view, and select **RTL Kernel Wizard**. The Vitis IDE attempts to open the Vivado project for the selected RTL kernel.



TIP: If the Vitis IDE is unable to find the Vivado project, it returns an error and does not let you edit the RTL kernel.

A dialog box opens displaying two options to edit an existing RTL kernel. Selecting **Edit Existing Kernel Contents** re-opens the Vivado Project, letting you modify and regenerate the kernel contents. Selecting **Re-customize Existing Kernel Interfaces** opens the RTL Kernel wizard. Options other than the **Kernel Name** can be modified, and the previous Vivado project is replaced.



IMPORTANT! All files and changes in the previous Vivado project are lost when the updated RTL kernel project is created.

Design Recommendations for RTL Kernels

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

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

- [Memory Performance Optimizations for AXI4 Interface](#)
- [Managing Clocks in an RTL Kernel](#)
- [Quality of Results Considerations](#)
- [Debug and Verification Considerations](#)

Memory Performance Optimizations for AXI4 Interface

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



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

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

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

Managing Clocks in an RTL Kernel

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

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

- Single input clock: `ap_clk`
 - External interfaces and internal kernel logic run at the same frequency.

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

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

1. RTL external interfaces are clocked at `ap_clk`.
2. The frequency synthesizer can have multiple output clocks that are used as internal clocks to the RTL kernel.
3. You must provide a Tcl script to downgrade DRCs related to clock resource placement in Vivado placement to prevent a DRC error from occurring. Refer to `CLOCK_DEDICATED_ROUTE` in the *Vivado Design Suite Properties Reference Guide (UG912)* for more information. The following is an example of the needed Tcl command that you will add to your Tcl script:

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

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

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

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

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

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

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

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



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

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

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

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

Quality of Results Considerations

The following recommendations help improve results for timing and area:

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

Debug and Verification Considerations

- RTL kernels should be verified in their own test bench using advanced verification techniques including verification components, randomization, and protocol checkers. The AXI Verification IP (VIP) is available in the Vivado IP catalog and can help with the verification of AXI interfaces. The RTL kernel example designs contain an AXI VIP-based test bench with sample stimulus files.

- 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.

Streaming Connections

Streaming Data Between the Host and Kernel (H2K)

The Vitis core development kit provides a programming model that supports the direct streaming of data from host-to-kernel and kernel-to-host, without the need to migrate data through global memory as an intermediate step. This programming model uses minimal storage compared to the larger and slower global memory bank, and thus significantly improves both performance and power.

By using data streams, you can realize some of the following advantages:

- The host application does not need to know the size of the data coming from the kernel.
- Data residing in host memory can be transferred to the kernel as soon as it is needed.
- Processed data can be transferred from the kernel back to the host program when it is required.

Host-to-kernel and kernel-to-host streaming are only supported in PCIe-based platforms, such as the Alveo Data Center accelerator cards. However, kernel-to-kernel streaming data transfer is supported for both PCIe-based and embedded platforms. In addition, this feature is only available on specific platform shells, such as the QDMA shell for the Alveo Data Center accelerator cards. If your platform is not configured to support streaming, your application will not run.

Host Coding Guidelines

Xilinx provides new OpenCL™ APIs for streaming operation as extension APIs.

- `clCreateStream()`: Creates a read or write stream.
- `clReleaseStream()`: Frees the created stream and its associated memory.
- `clWriteStream()`: Writes data to stream.
- `clReadStream()`: Gets data from stream.
- `clPollStreams()`: Polls for any stream on the device to finish. Required only for non-blocking stream operation.

The typical API flow is described below:

- Create the required number of the read/write streams by `clCreateStream`.
 - Streams should be directly attached to the OpenCL device object because it does not use any command queue. A stream itself is a command queue that only passes the data in a particular direction, either from host to kernel or from kernel to host.
 - An appropriate flag should be used to denote the stream as `CL_STREAM_READ_ONLY` or `CL_STREAM_WRITE_ONLY` from the perspective of the host program).
 - To specify how the stream is connected to the device, a Xilinx extension pointer object (`cl_mem_ext_ptr_t`) is used to identify the kernel, and the kernel argument the stream is associated with.



IMPORTANT! *If the streaming kernel has multiple compute units, the host code needs to use a unique `cl_kernel` object for each compute unit. The host code must use `clCreateKernel` with `<kernel_name>:{compute_unit_name}` to get each compute unit, creating streams for them, and enqueueing them individually.*

In the following code example, a `read_stream` and a `write_stream` are created, and associated with a `cl_kernel` object, and specified kernel arguments.

```
#include <CL/cl_ext_xilinx.h> // Required for Xilinx extension pointer

// Device connection specification of the stream through extension
// pointer
cl_mem_ext_ptr_t ext; // Extension pointer
ext.param = kernel; // The .param should be set to kernel
                    // (cl_kernel type)
ext.obj = nullptr;

// The .flag should be used to denote the kernel argument
// Create write stream for argument 3 of kernel
ext.flags = 3;
cl_stream write_stream = clCreateStream(device_id,
CL_STREAM_WRITE_ONLY, CL_STREAM, &ext, &ret);

// Create read stream for argument 4 of kernel
ext.flags = 4;
cl_stream read_stream = clCreateStream(device_id, CL_STREAM_READ_ONLY,
CL_STREAM, &ext,&ret);
```

- Set the remaining non-streaming kernel arguments and enqueue the kernel. The following code block shows setting typical kernel argument (non-stream arguments, such as buffer and/or scalar) and kernel enqueueing:

```
// Set kernel non-stream argument (if any)
clSetKernelArg(kernel, 0,.....);
clSetKernelArg(kernel, 1,.....);
clSetKernelArg(kernel, 2,.....);
// Argument 3 and 4 are not set as those are already specified during
// the clCreateStream through the extension pointer

// Schedule kernel enqueue
clEnqueueTask(commands, kernel, . . . . );
```

- Initiate Read and Write transfers by `clReadStream` and `clWriteStream` commands.
 - Note the usage of attribute `CL_STREAM_XFER_REQ` associated with read and write request.
 - The `.flag` is used to denote transfer mechanism.
 - **CL_STREAM_EOT:** Currently, successful stream transfer mechanism depends on identifying the end of the transfer by an *End of Transfer* signal. This flag is mandatory in the current release.
 - **CL_STREAM_NONBLOCKING:** By default the Read and Write transfers are blocking. For non-blocking transfer, `CL_STREAM_NONBLOCKING` has to be set.
 - The `.priv_data` is used to specify a string (as a name for tagging purpose) associated with the transfer. This will help identify specific transfer completion when polling the stream completion. It is required when using the non-blocking version of the API.

In the following code block, the stream read and write transfers are executed with the non-blocking approach.

```
// Initiate the READ transfer
cl_stream_xfer_req rd_req {0};

rd_req.flags = CL_STREAM_EOT | CL_STREAM_NONBLOCKING;
rd_req.priv_data = (void*)"read"; // You can think this as tagging the
                                   transfer with a name

clReadStream(read_stream, host_read_ptr, max_read_size, &rd_req, &ret);

// Initiating the WRITE transfer
cl_stream_xfer_req wr_req {0};

wr_req.flags = CL_STREAM_EOT | CL_STREAM_NONBLOCKING;
wr_req.priv_data = (void*)"write";

clWriteStream(write_stream, host_write_ptr, write_size, &wr_req , &ret);
```

- Poll all the streams for completion. For the non-blocking transfer, a polling API is provided to ensure the read/write transfers are completed. For the blocking version of the API, polling is not required.
 - The polling results are stored in the `cl_streams_poll_req_completions` array, which can be used in verifying and checking the stream events result.
 - The `clPollStreams` is a blocking API. It returns the execution to the host code as soon as it receives the notification that all stream requests have been completed, or until you specify the timeout.

```
// Checking the request completion
cl_streams_poll_req_completions poll_req[2] {0, 0}; // 2 Requests

auto num_compl = 2;
clPollStreams(device_id, poll_req, 2, 2, &num_compl, 5000, &ret);
// Blocking API, waits for 2 poll request completion or 5000ms,
  whichever occurs first
```

- Read and use the stream data in host.
 - After the successful poll request is completed, the host can read the data from the host pointer.
 - Also, the host can check the size of the data transferred to the host. For this purpose, the host needs to find the correct poll request by matching `priv_data` and then fetching `nbytes` (the number of bytes transferred) from the `cl_streams_poll_req_completions` structure.

```

for (auto i=0; i<2; ++i) {
    if(rd_req.priv_data == poll_req[i].priv_data) { // Identifying the
                                                    read transfer
        // Getting read size, data size from kernel is unknown
        ssize_t result_size=poll_req[i].nbytes;
    }
}
    
```

The header file containing function prototype and argument description is available in the [Xilinx Runtime GitHub repository](#).

Kernel Coding Guidelines

The basic guidelines to develop stream-based C kernel are as follows:

- Use `hls::stream` with the `qdma_axis<D,0,0,0>` data type. The `qdma_axis` data type needs the header file `ap_axi_sdata.h`.
- The `qdma_axis<D,0,0,0>` is a special class used for data transfer between host and kernel when using the streaming platform. This is only used in the streaming kernel interface interacting with the host, not with another kernel. The template parameter `<D>` denotes data width. The remaining three parameters should be set to 0 (not to be used in the current release).
- The following code block shows a simple kernel interface with one input stream and one output stream.

```

#include "ap_axi_sdata.h"
#include "hls_stream.h"

//qdma_axis is the HLS class for stream data transfer between host and
kernel for streaming platform
//It contains "data" and two sideband signals (last and keep) exposed to
the user via class member function.
typedef qdma_axis<64,0,0,0> datap;

void kernel_top (
    hls::stream<datap> &input,
    hls::stream<datap> &output,
    .... , // Other Inputs/Outputs if any
)
{
    #pragma HLS INTERFACE axis port=input
    #pragma HLS INTERFACE axis port=output
}
    
```

- The `qdma_axis` data type contains three variables which should be used inside the kernel code:
 - **data:** Internally `qdma_axis` contains an `ap_uint<D>` that should be accessed by the `.get_data()` and `.set_data()` method.
 - The `D` must be 8, 16, 32, 64, 128, 256, or 512 bits wide.
 - **last:** The `last` variable is used to indicate the last value of an incoming and outgoing stream. When reading from the input stream, `last` is used to detect the end of the stream. Similarly when kernel writes to an output stream transferred to the host, the `last` must be set to indicate the end of stream.
 - `get_last/set_last:` Accesses and sets the `last` variable used to denote the last data in the stream.
 - **keep:** In some special situations, the `keep` signal can be used to truncate the last data to the fewer number of bytes. However, `keep` should not be used to any data other than the last data from the stream. So, in most of the cases, you should set `keep` to -1 for all the outgoing data from the kernel.
 - `get_keep/set_keep:` Accesses/sets the `keep` variable.
 - For all the data before the last data, `keep` must be set to -1 to denote all bytes of the data are valid.
 - For the last data, the kernel has the flexibility to send fewer bytes. For example, for the four bytes data transfer, the kernel can truncate the last data by sending one byte, two bytes, or three bytes by using the following `set_keep()` function.
 - If the last data is one byte \geq `.set_keep(1)`
 - If the last data is two bytes \geq `.set_keep(3)`
 - If the last data is three bytes \geq `.set_keep(7)`
 - If the last data is all four bytes (similar to all non-last data) \geq `.set_keep(-1)`
- The following code block shows how the stream `input` is read. Note the usage of `.last` to determine the last data.

```
// Stream Read
// Using "last" flag to determine the end of input-stream
// when kernel does not know the length of the input data
hls::stream<ap_uint<64> > internal_stream;
while(true) {
    datap temp = input.read(); // "input" -> Input stream
    internal_stream << temp.get_data(); // Getting data from the
    stream
    if(temp.get_last()) // Getting last signal to determine the
    EOT (end of transfer).
        break;
}
```

- The following code block shows how the stream `output` is written. The `set_keep` is setting -1 for all data (general case). Also, the kernel uses the `set_last()` to specify the last data of the stream.



IMPORTANT! For the proper functionality of the host and kernel system, it is very important to set the `last` bit setting.

```
// Stream Write
for(int j = 0; j <....; j++) {
    datap t;
    t.set_data(...);
    t.set_keep(-1);           // keep flag -1 , all bytes are valid
    if(... )                 // check if this is last data to be write
        t.set_last(1);      // Setting last data of the stream
    else
        t.set_last(0);
    output.write(t);        // output stream from the kernel
}
```

Streaming Data Transfers Between Kernels (K2K)

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



IMPORTANT! This feature is only available on specific platform shells, such as the QDMA shell for the Alveo Data Center accelerator cards. If your platform is not configured to support streaming, your application will not run.

Host Coding Guidelines

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

Streaming Kernel Coding Guidelines

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



IMPORTANT! Host-to-kernel and kernel-to-host streaming, as described in [Streaming Data Between the Host and Kernel \(H2K\)](#), requires the use of the `qdma_axis` data type. Both the `ap_axiu` and `qdma_axis` data types are defined inside the `ap_axi_sdata.h` header file that is distributed with the Vitis software platform installation.

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

```
// Producer kernel - provides output as a data stream
// The example kernel code does not show any other inputs or outputs.

void kernel1 (.... , hls::stream<ap_axiu<32, 0, 0, 0> >& stream_out) {
#pragma HLS interface axis port=stream_out

    for(int i = 0; i < ...; i++) {
        int a = ..... ;           // Internally generated data
        ap_axiu<32, 0, 0, 0> v;    // temporary storage for ap_axiu
        v.data = a;               // Writing the data
        stream_out.write(v);      // Writing to the output stream.
    }
}

// Consumer kernel - reads data stream as input
// The example kernel code does not show any other inputs or outputs.

void kernel2 (hls::stream<ap_axiu<32, 0, 0, 0> >& stream_in, .... ) {
#pragma HLS interface axis port=stream_in

    for(int i = 0; i < ...; i++) {
        ap_axiu<32, 0, 0, 0> v = stream_in.read(); // Reading the input stream
        int a = v.data; // Extract the data

        // Do further processing
    }
}
```



TIP: The example kernels above show the definition of the streaming input/output ports in the kernel signature, and the handling of the input/output stream in the kernel code. The connection of *kernel1* to *kernel2* must be defined during the kernel linking process as described in [Specify Streaming Connections Between Compute Units](#).

Free-running Kernel

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

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

Host Coding for Free Running Kernels

If the free-running kernel interacts with the host, the host code should manage the stream operation by `clCreateStream/clReadStream/clWriteStream` as discussed in [Host Coding Guidelines](#) of [Streaming Data Between the Host and Kernel \(H2K\)](#). As the free-running kernel has no other types of inputs or outputs, such as memory ports or control ports, there is no need to specify `clSetKernelArg`. The `clEnqueueTask` is not used because the kernel works on the stream data as soon as it starts receiving from the host or other kernels, and it stalls when the data is not available.

Coding Guidelines for Free Running Kernels

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

- Using `hls::stream<ap_axiu<D,0,0,0> >` if the port is interacting with another stream port from the kernel.
- Using `hls::stream<qdma_axis<D,0,0,0> >` if the port is interacting with the host.

The guidelines for using a pragma include:

- The kernel interface should not have any `#pragma HLS interface s_axilite` or `#pragma HLS interface m_axi` (as there should not be any memory or control port).
- The kernel interface must contain this special pragma:

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

The following code example shows a free-running kernel with one input and one output communicating with another kernel.

```
void kernel_top(hls::stream<ap_axiu<32, 0, 0, 0> >& input,
               hls::stream<ap_axiu<32, 0, 0, 0> >& output) {
    #pragma HLS interface axis port=input
    #pragma HLS interface axis port=output
    #pragma HLS interface ap_ctrl_none port=return // Special pragma for free-
    running kernel

    #pragma HLS DATAFLOW // The kernel is using DATAFLOW optimization
    ...
}
```



TIP: The example shows the definition of the streaming input/output ports in a free running kernel. However, the streaming connection from the free running kernel to or from another kernel must be defined during the kernel linking process as described in [Specify Streaming Connections Between Compute Units](#).

OpenCL Kernels

This OpenCL kernel discussion is based on the information provided in the [C/C++ Kernels](#) topic. The same programming techniques for accelerating the performance of a kernel apply to both C/C++ and OpenCL kernels. However, the OpenCL kernel uses the `__attribute` syntax in place of pragmas. For details of the available attributes, refer to [OpenCL Attributes](#).

The following code examples show some of the elements of an OpenCL kernel for the Vitis application acceleration development flow. This is not intended to be a primer on OpenCL or kernel development, but to merely highlight some of the key difference between OpenCL and C/C++ kernels.

Kernel Signature

In C/C++ kernels, the kernel is identified on the command line of the Vitis compiler by the use of the `v++ --kernel` option. However, in OpenCL code, the `__kernel` keyword identifies a kernel in the code. You can have multiple kernels defined in a single `.cl` file, and the Vitis compiler will compile all of the kernels, unless you specify the `--kernel` option to identify which kernel to compile.

```
__kernel __attribute__((reqd_work_group_size(1, 1, 1)))
void apply_watermark(__global const TYPE * __restrict input,
    __global TYPE * __restrict output, int width, int height) {
{
    ...
}
```



TIP: The complete code for the kernel function above, `apply_watermark`, can be found in the [Global Memory Two Banks \(CL\)](#) example in the [Vitis Examples repository on GitHub](#).

In the example above, you can see the watermark kernel has two pointer type arguments: `input` and `output`, and has two scalar type `int` arguments: `width` and `height`.

In C/C++ kernels these arguments would need to be identified with the `HLS INTERFACE` pragmas. However, in the OpenCL kernel the Vitis compiler, and Vivado HLS recognize the kernel arguments, and compile them as needed: pointer arguments into `m_axi` interfaces, and scalar arguments into `s_axilite` interfaces.

Kernel Optimizations

Because the kernel is running in programmable logic on the target platform, optimizing your task to the environment is an important element of application design. Most of the optimization techniques discussed in [C/C++ Kernels](#) can be applied to OpenCL kernels. Instead of applying the HLS pragmas used for C/C++ kernels, you will use the `__attribute__` keyword described in [OpenCL Attributes](#). Following is an example:

```
// Process the whole image
__attribute__((xcl_pipeline_loop))
image_traverse: for (uint idx = 0, x = 0, y = 0 ; idx < size ; ++idx, x+=
DATA_SIZE)
{
    ...
}
```

The example above specifies that the `for` loop, `image_traverse`, should be pipelined to improve the performance of the kernel. The target II in this case is 1. Refer to [xcl_pipeline_loop](#) for more information.

In the following code example, the watermark function uses the `opencl_unroll_hint` attribute to let the Vitis compiler unroll the loop to reduce latency and improve performance. However, in this case the `__attribute__` is only a suggestion that the compiler can ignore if needed. Refer to [opencl_unroll_hint](#) for details.

```
//Unrolling below loop to process all 16 pixels concurrently
__attribute__((opencl_unroll_hint))
watermark: for ( int i = 0 ; i < DATA_SIZE ; i++)
{
    ...
}
```

For more information, you can review the [OpenCL Attributes](#) topics to see what specific optimizations are supported for OpenCL kernels, and review the [C/C++ Kernels](#) content to see how these optimizations can be applied in your kernel design.

Best Practices for Acceleration with Vitis

Below are some specific things to keep in mind when developing your application code and hardware function in the Vitis™ core development kit.

- Look to accelerate functions that have a high ratio of compute time to input and output data volume. Compute time can be greatly reduced using FPGA kernels, but data volume adds transfer latency.
- Accelerate functions that have a self-contained control structure and do not require regular synchronization with the host.

- Transfer large blocks of data from host to global device memory. One large transfer is more efficient than several smaller transfers. Run a bandwidth test to find the optimal transfer size.
- Only copy data back to host when necessary. Data written to global memory by a kernel can be directly read by another kernel. Memory resources include PLRAM (small size but fast access with lowest latency), HBM (moderate size and access speed with some latency), and DDR (large size but slow access with high latency).
- Take advantage of the multiple global memory resources to evenly distribute bandwidth across kernels.
- Maximize bandwidth usage between kernel and global memory by performing 512-bit wide bursts.
- Cache data in local memory within the kernels. Accessing local memories is much faster than accessing global memory.
- In the host application, use events and non-blocking transactions to launch multiple requests in a parallel and overlapping manner.
- In the FPGA, use different kernels to take advantage of task-level parallelism and use multiple CUs to take advantage of data-level parallelism to execute multiple tasks in parallel and further increase performance.
- Within the kernels take advantage of tasks-level with dataflow and instruction-level parallelism with loop unrolling and loop pipelining to maximize throughput.
- Some Xilinx FPGAs contain multiple partitions called super logic regions (SLRs). Keep the kernel in the same SLR as the global memory bank that it accesses.
- Use software and hardware emulation to validate your code frequently to make sure it is functionally correct.
- Frequently review the Vitis Guidance report as it provides clear and actionable feedback regarding deficiencies in your project.

Building and Running the Application

After the host program and the kernel code is written, you can build the application, which includes building the host program and platform file (`xclbin`). The build process follows a standard compilation and linking process for both the host program and the kernel code. However, the first step in building the application is to identify the build target, indicating if you are building for test or simulation of the application, or building for the target hardware. After building, both the host program and the FPGA binary, you will be ready to run the application.

Setting up the Vitis Integrated Design Environment

The Vitis integrated design environment (IDE) includes three elements that must be installed and configured to work together properly: the Vitis core tools, the Xilinx Runtime (XRT), and an accelerator card such as the Alveo Data Center accelerator card.

If you have the elements of the Vitis IDE installed, you need to setup the environment to run in a specific command shell by running the following shell scripts (`.csh` scripts are also provided):

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

You can also specify the location of the available platforms for use with your Vitis IDE by setting the following environment variable:

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



TIP: The `PLATFORM_REPO_PATHS` environment variable points to directories containing platform files (`.xpfm`).

Build Targets

The build target of the Vitis IDE defines the nature and contents of the FPGA binary (.xclbin) created during compilation and linking. There are three different build targets: two emulation targets used for validation and debugging purposes: software emulation and hardware emulation, and the default system hardware target used to generate the actual binary (.xclbin) loaded into the Xilinx device.

Compiling for an emulation target is significantly faster than compiling for the real hardware. The emulation run is performed in a simulation environment, which offers enhanced debug visibility and does not require an actual accelerator card.

Table 10: Comparison of Emulation Flows with Hardware Execution

Software Emulation	Hardware Emulation	Hardware Execution
Host application runs with a C/C++ or OpenCL model of the kernels.	Host application runs with a simulated RTL model of the kernels.	Host application runs with actual hardware implementation of the kernels.
Used to confirm functional correctness of the system.	Test the host / kernel integration, get performance estimates.	Confirm that the system runs correctly and with desired performance.
Fastest build time supports quick design iterations.	Best debug capabilities, moderate compilation time with increased visibility of the kernels.	Final FPGA implementation, long build time with accurate (actual) performance results.

Software Emulation

The main goal of software emulation is to ensure functional correctness of the host program and kernels. For software emulation, both the host code and the kernel code are compiled to run on the host x86 processor. The v++ compiler does the minimum transformation of the kernel code to create the FPGA binary, in order to run the host program and kernel code together. The software emulation flow can be used for algorithm refinement, debugging functional issues, and letting developers iterate quickly through the code to make improvements. The programming model of development through fast compile and run iterations is preserved.

In the context of the Vitis unified software platform, 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.



TIP: 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.

As discussed in [Vitis Compiler Command](#), the software emulation target is specified in the `v++` command with the `-t` option:

```
v++ -t sw_emu ...
```

Hardware Emulation

The hardware emulation flow enables the programmer to check the functional correctness of the RTL description of the FPGA binary synthesized from the C, C++, or OpenCL kernel code.

Each kernel is compiled to a hardware model (RTL). During hardware emulation, kernels are run in the Vivado logic simulator, with a waveform viewer to examine the kernel design. In addition, hardware emulation provides performance and resource estimates for the hardware implementation.

In hardware emulation, compile and execution times are longer than for software emulation, but it provides a detailed, cycle-accurate, view of kernel activity. Xilinx recommends that you use small data sets for validation during hardware emulation to keep run times manageable.



IMPORTANT! *The DDR memory model and the memory interface generator (MIG) model used in hardware emulation are high-level simulation models. These models provide good simulation performance, but only approximate latency values and are not cycle-accurate like the kernels. Therefore, performance numbers shown in the profile summary report are approximate, and should be used for guidance and for comparing relative performance between different kernel implementations.*

As discussed in [Vitis Compiler Command](#), the hardware emulation target is specified in the `v++` command with the `-t` option:

```
v++ -t hw_emu ...
```

System Hardware Target

When the build target is the system hardware, `v++` builds the FPGA binary for the Xilinx device by running synthesis and implementation on the design. Therefore, it is normal for this build target to take a longer period of time than generating either the software or hardware emulation targets in the Vitis IDE. However, the final FPGA binary can be loaded into the hardware of the accelerator card, or embedded processor platform, and the application can be run in its actual operating environment.

As discussed in [Vitis Compiler Command](#), the system hardware target is specified in the `v++` command with the `-t` option:

```
v++ -t hw ...
```

Building the Host Program

The host program, written in C/C++ using OpenCL™ API calls, is built using the GNU C++ compiler (`g++`) which is based on GNU compiler collection (GCC). Each source file is compiled to an object file (`.o`) and linked with the Xilinx runtime (XRT) shared library to create the executable which runs on the host CPU.



TIP: `g++` supports many standard GCC options which are not documented here. For information refer to the [GCC Option Summary](#).

Compiling and Linking for x86



TIP: Setup the command shell or window as described in [Setting up the Vitis Integrated Design Environment](#) prior to running the tools.

Each source file of the host application is compiled into an object file (`.o`) using the `g++` compiler.

```
g++ ... -c <source_file1> <source_file2> ... <source_fileN>
```

The generated object files (`.o`) are linked with the Xilinx Runtime (XRT) shared library to create the executable host program. Linking is performed using the `-l` option.

```
g++ ... -l <object_file1.o> ... <object_fileN.o>
```

Compiling and linking for x86 follows the standard `g++` flow. The only requirement is to include the XRT header files and link the XRT shared libraries.

When compiling the source code, the following `g++` options are required:

- `-I$XILINX_XRT/include/`: XRT include directory.
- `-I$XILINX_VIVADO/include`: Vivado tools include directory.
- `-std=c++11`: Define the C++ language standard (instead of the include directory).

When linking the executable, the following `g++` options are required:

- `-L$XILINX_XRT/lib/`: Look in XRT library.
- `-lOpenCL`: Search the named library during linking.
- `-lpthread`: Search the named library during linking.
- `-lrt`: Search the named library during linking.
- `-lstdc++`: Search the named library during linking.

Note: In the [Vitis Examples](#) you may see the addition of `xc12.cpp` source file, and the `-I../libs/xc12` include statement. These additions to the host program and `g++` command provide access to helper utilities used by the example code, but are generally not required for your own code.

Compiling and Linking for Arm



TIP: Setup the command shell or window as described in [Setting up the Vitis Integrated Design Environment](#) prior to running the tools.

The host program (`host.exe`), is cross-compiled for an Arm processor, and linked using the following two step process:

1. Compile the `host.cpp` into an object file (`.o`) using the GNU Arm cross-compiler version of `g++`:

```
$XILINX_VITIS/gnu/aarch64/lin/aarch64-linux/bin/aarch64-linux-gnu-g++ \
-D__USE_XOPEN2K8 -I$SYSROOT/usr/include/xrt -I$XILINX_VIVADO/include \
-I$SYSROOT/usr/include -c -fmessage-length=0 -std=c++14 \
--sysroot=$SYSROOT -o src/host.o ../src/host.cpp
```

2. Link the object file with required libraries to build the executable application.

```
$XILINX_VITIS/gnu/aarch64/lin/aarch64-linux/bin/aarch64-linux-gnu-g++ \
-o host.exe src/host.o -lxilinxopencl -lpthread -lrt -lstdc++ -lgmp -
lxrt_core \
-L$SYSROOT/usr/lib/ --sysroot=$SYSROOT
```

When compiling the application for use with an embedded process, you must specify the `sysroot` for the application. The `sysroot` is part of the platform where the basic system root file structure is defined.



IMPORTANT! The above examples rely on the use of `$SYSROOT` environment variable that must be used to specify the location of the `sysroot` for your embedded platform.

The following are key elements to compiling the host code for an edge platform:

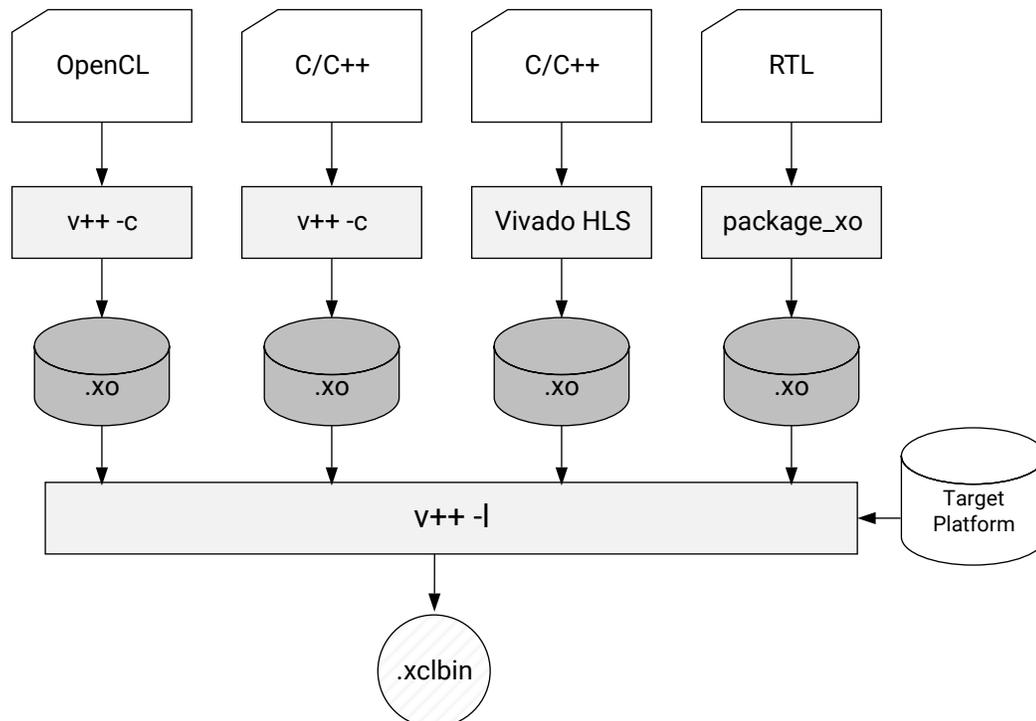
- **Compilation:**
 - The cross compiler needed is the `aarch64-linux-gnu-g++` found in the Vitis installation hierarchy.
 - The required include paths are:
 - `$SYSROOT/usr/include/xrt`
 - `$SYSROOT/usr/include`
 - `$XILINX_VIVADO/include`
- **Linking:**
 - `$SYSROOT/usr/lib`: Library paths location.

- `$SYSROOT/usr/lib/xrt`: XRT library path.
- `xilinxopencl`: XRT required library.
- `pthread`: XRT required library.
- `rt`: XRT required library.
- `stdc++`: XRT required library.
- `gmp`: XRT required library.
- `xrt_core`: XRT required library.

Building the FPGA Binary

The kernel code is written in C, C++, OpenCL C, or RTL, and is built by compiling the kernel code into a Xilinx object file (`.xo`), and linking the `.xo` files into an FPGA binary file (`.xclbin`), as shown in the following figure.

Figure 32: FPGA Build Process



X21155-091219

The process, as outlined above, has two steps:

1. Build the Xilinx object files from the kernel source code.
 - For C, C++, or OpenCL kernels, the `v++ -c` command compiles the source code into Xilinx object (`.xo`) files. Multiple kernels are compiled into separate `.xo` files.
 - For RTL kernels, the `package_xo` command produces the `.xo` file to be used for linking. Refer to [RTL Kernels](#) for more information.
 - You can also create kernel object files (`.xo`) working directly in the Vivado HLS tool. Refer to [Compiling Kernels Directly in Vivado HLS](#) for more information.
2. After compilation, the `v++ -l` command links one or multiple kernel objects (`.xo`), together with the hardware platform (`.xsa`), to produce the FPGA binary (`.xclbin`).



TIP: The `v++` command can be used from the command-line, in scripts, or a build system like `make`, and can also be used through the GUI as discussed in [Chapter 6: Using the Vitis IDE](#).

Compiling Kernels with Vitis Compiler



IMPORTANT! Setup the command shell or window as described in [Setting up the Vitis Integrated Design Environment](#) prior to running the tools.

The first stage in building the `xclbin` file is to compile the kernel code using the Xilinx Vitis compiler. There are multiple `v++` options that need to be used to correctly compile your kernel. The following is an example command line to compile the `vadd` kernel:

```
v++ -t sw_emu --platform xilinx_u200_xdma_201830_2 -c -k vadd \
-I'./src' -o'vadd.sw_emu.xo' ./src/vadd.cpp
```

The various arguments used are described below. Note that some of the arguments are required.

- `-t sw_emu`: Specifies the build target as software emulation, as discussed in [Build Targets](#). This is a required argument.
- `--platform xilinx_u200_xdma_201830_2`: Specifies the accelerator platform for the build. This is required because runtime features, and the platform shell are linked as part of the FPGA binary. To compile a kernel for an embedded processor application, you simply specify an embedded processor platform: `--platform $PLATFORM_REPO_PATHS/zcu102_base/zcu102_base.xpfm`.
- `-c`: Compile the kernel. Required. The kernel must be compiled (`-c`) and linked (`-l`) in two separate steps.
- `-k vadd`: Name of the kernel associated with the source files.
- `./src/vadd.cpp`: Specify source files for the kernel. Multiple source files can be specified. Required.
- `-o'vadd.sw_emu.xo'`: Specify the shared object file output by the compiler. Required.

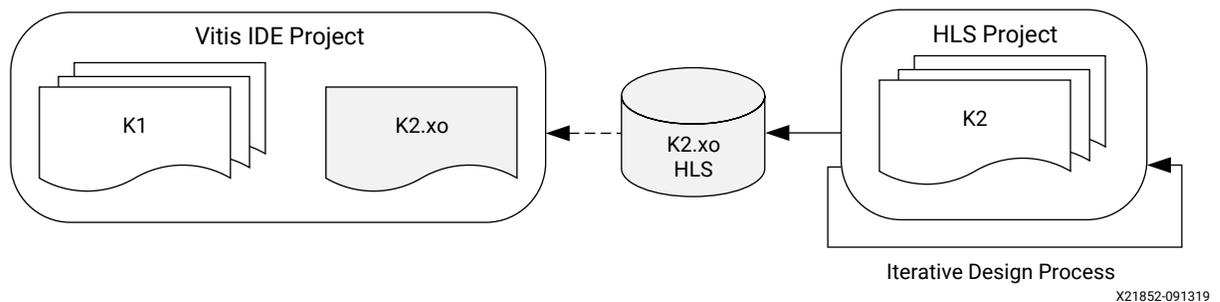
Refer to [Vitis Compiler Command](#) for details of the various command line options. Refer to [Output Directories from the v++ Command](#) to get an understanding of the location of various output files.

Compiling Kernels Directly in Vivado HLS

The use model described for the Vitis core development kit is a top-down approach, starting with C/C++ or OpenCL code, and working toward compiled kernels. However, you can also directly develop the kernel to produce a Xilinx® object (.xo) file to be paired for linking using v++ to produce the .xclbin. This approach can be used for C/C++ kernels using the Vivado® HLS tool, which is the focus of this section, or RTL kernels using the Vivado Design Suite. Refer to [RTL Kernels](#) for more information.

The approach of developing the kernel directly, either in RTL or C/C++, to produce an .xo file, is sometimes referred to as the bottom-up flow. This allows you to validate kernel performance and perform optimizations within the Vivado HLS tool, and export the Xilinx® object file for use in the Vitis application acceleration development flow.

Figure 33: Vivado HLS Bottom-Up Flow



The benefits of the Vivado HLS bottom-up flow can include:

- Design, validate, and optimize the kernel separately from the main application.
- Enables a team approach to design, with collaboration on host program and kernel development.
- Specific kernel optimizations are preserved in the .xo file.
- A collection of .xo files can be used and reused like a library.

Creating Kernels in Vivado HLS

Generating kernels from C/C++ code for use in the Vitis core development kit follows the standard Vivado HLS process as described in *Vivado Design Suite User Guide: High-Level Synthesis (UG902)*. However, since the kernel is required to operate in the Vitis software platform, the standard kernel requirements must be satisfied (see [Kernel Requirements](#)). Most importantly, the interfaces must be modeled as AXI memory interfaces, except for scalar parameters which are mapped to an AXI4-Lite interface.

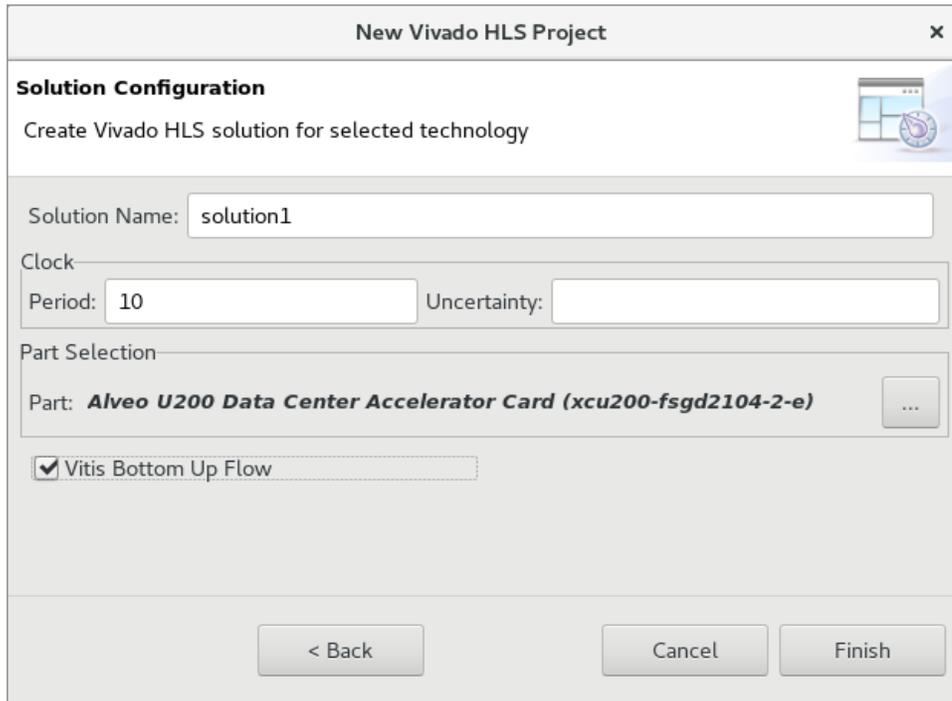
The following example defines the function signature, and provides HLS INTERFACE pragmas to define the interface ports for the function arguments:

```
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
}
```

The following describes the process for creating and compiling your HLS kernel:

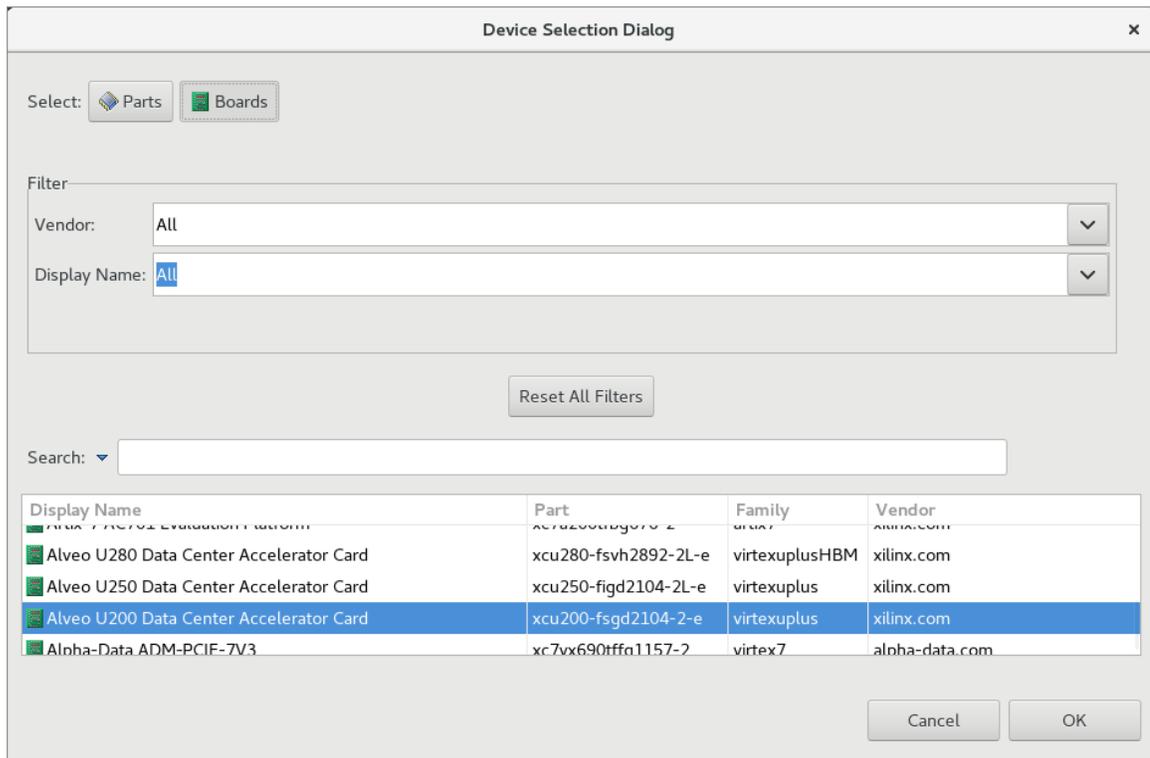
1. Launch Vivado HLS to open the GUI, and specify **File** → **New Project**.
2. In the New Vivado HLS Project dialog box, specify the **Project name**, define the **Location** for the project, and click **Next**.
3. In the Add/Remove files dialog box, click **Add Files** to the kernel source code to the project, select **Top Function** to define the kernel function, and click **Next**.
4. You can specify a C-based simulation testbench if you have one available, by clicking **Add Files**, or skip this by clicking **Next**.
5. In the Solution Configuration dialog box, you must specify the **Clock Period** for the kernel.

Figure 34: New Vivado HLS Project



6. Choose the target platform by clicking the **Browse** button in the Part Selection field to open the Device Selection dialog box. Select the **Boards** command, and select the target platform for your compiled kernel, as shown below. Click OK to select the platform and return to the Solution Configuration dialog box.

Figure 35: Device Selection



- In the Solution Configuration dialog box, enable the **Vitis Bottom Up Flow** checkbox, and click **Finish** to complete the process and create your HLS kernel project.

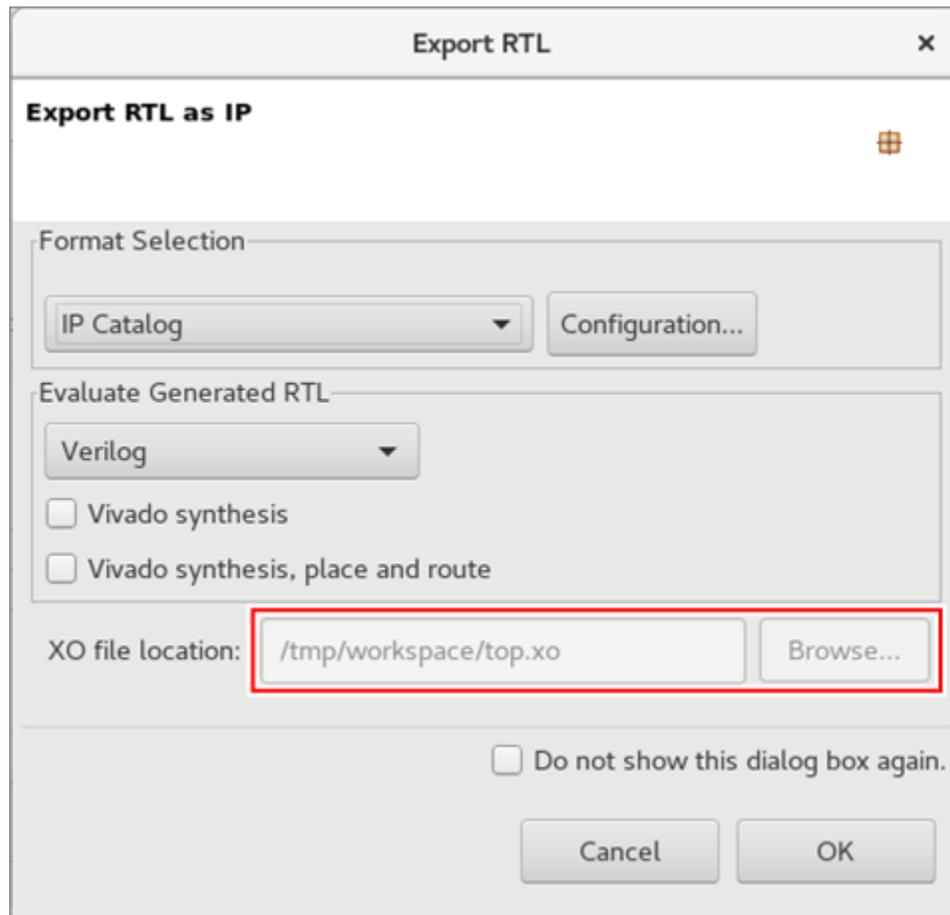


IMPORTANT! You must enable the **Vitis Bottom Up Flow** in order to generate the Xilinx object (.xo) file from the project.

When the HLS project has been created you can **Run C-Synthesis** to compile the kernel code. Refer to *Vivado Design Suite User Guide: High-Level Synthesis (UG902)* for a complete description of the HLS tool flow.

After synthesis is completed, the kernel can be exported as an .xo file for use in the Vitis core development kit. The export command is available through the **Solution** → **Export RTL** command from the main menu.

Figure 36: Export RTL as IP



Specify the file location, and the kernel is exported as a Xilinx object `.xo` file.

The `(.xo)` file can be used as an input file during the `v++` linking process. Refer to [Linking the Kernels](#) for more information. You can also add it to an application project in the Vitis integrated design environment (IDE), as discussed in [Creating a Vitis IDE Project](#).

However, keep in mind that HLS kernels, created in the bottom-up flow described here, have certain limitations when used in the Vitis application acceleration development flow. Software emulation is not supported for applications using HLS kernels, because duplicated header file dependencies can create problems. GDB debug is not supported in the hardware emulation flow for HLS kernels, or RTL kernels.

Vivado HLS Script for Creating Kernels

If you run HLS synthesis through Tcl scripts, you can edit the following script to create HLS kernels as previously described:

```
# Define variables for your HLS kernel:
set projName <proj_name>
set krnlName <kernel_name>
set krnlFile <kernel_source_code>
#set krnlTB <kernel_test_bench>
set krnlPlatform <target_part>
set path <path_to_project>

#Script to create and output HLS kernel
open_project $projName
set_top $krnlName
add_files $krnlFile
#add_files -tb $krnlTB
open_solution "solution1"
set_part $krnlPlatform
create_clock -period 10 -name default
config_sdx -optimization_level none -target xocc
config_export -vivado_optimization_level 0 -vivado_phys_opt none
config_compile -name_max_length 256 -pipeline_loops 64
config_schedule -enable_dsp_full_reg
#source "./hlsKernel/solution1/directives.tcl"
#csim_design
csynth_design
#cosim_design
export_design -rtl verilog -format ip_catalog -xo $path/$krnlName.xo
```

Run the HLS kernel script by using the following command after setting up your environment as discussed in [Setting up the Vitis Integrated Design Environment](#).

```
vivado_hls -f <hls_kernel_script>.tcl
```

Packaging RTL Kernels with package_xo

Kernels written in RTL are compiled in the Vivado tool using the `package_xo` command line utility which generates a Xilinx object file (`.xo`) which can subsequently be used by the `v++` command, during the linking stage. (See [package_xo Command](#).) The process for creating RTL kernels, and using the `package_xo` command to generate an `.xo` file is described in [RTL Kernels](#).

Linking the Kernels



TIP: Setup the command shell or window as described in [Setting up the Vitis Integrated Design Environment](#) prior to running the tools.

The kernel compilation process results in a Xilinx object file (`.xo`) whether the kernel is written in C/C++, OpenCL C, or RTL. During the linking stage, `.xo` files from different kernels are linked with the platform to create the FPGA binary container file (`.xclbin`) used by the host program.

Similar to compiling, linking requires several options. The following is an example command line to link the `vadd` kernel binary:

```
v++ -t sw_emu --platform xilinx_u200_xdma_201830_2 -l vadd.sw_emu.xo \
-o'vadd.sw_emu.xclbin' --config ./connectivity.cfg
```

This command contains the following arguments:

- `-t sw_emu`: Specifies the build target. When linking, you must use the same `-t` and `--platform` arguments as specified when the input file (`.xo`) was compiled.
- `--platform xilinx_u200_xdma_201830_2`: Specifies the platform to link the kernels with. To link the kernels for an embedded processor application, you simply specify an embedded processor platform: `--platform $PLATFORM_REPO_PATHS/zcu102_revmin/zcu102_revmin.xpfm`
- `-l`: Link the kernels and platform into an FPGA binary file (`xclbin`).
- `vadd.sw_emu.xo`: Input object file. Multiple object files can be specified to build into the `.xclbin`.
- `-o'vadd.sw_emu.xclbin'`: Specify the output file name. The output file in the link stage will be an `.xclbin` file. The default output name is `a.xclbin`
- `--config ./connectivity.cfg`: Specify a configuration file that is used to provide `v++` command options for a variety of uses. Refer to [Vitis Compiler Command](#) for more information on the `--config` option.



TIP: Refer to [Output Directories from the v++ Command](#) to get an understanding of the location of various output files.

Beyond simply linking the Xilinx object files (`.xo`), the linking process is also where important architectural details are determined. In particular, this is where the number of compute unit (CUs) to instantiate into hardware is specified, connections from kernel ports to global memory are assigned, and CUs are assigned to SLRs. The following sections discuss some of these build options.

Creating Multiple Instances of a Kernel

By default, the linker builds a single hardware instance from a kernel. If the host program will execute the same kernel multiple times, due to data processing requirements for instance, then it must execute the kernel on the hardware accelerator in a sequential manner. This can impact overall application performance. However, you can customize the kernel linking stage to instantiate multiple hardware compute units (CUs) from a single kernel. This can improve performance as the host program can now make multiple overlapping kernel calls, executing kernels concurrently by running separate compute units.

Multiple CUs of a kernel can be created by using the `connectivity.nk` option in the `v++` config file during linking. Edit a config file to include the needed options, and specify it in the `v++` command line with the `--config` option, as described in [Vitis Compiler Command](#).

For example, for the `vadd` kernel, two hardware instances can be implemented in the config file as follows:

```
[connectivity]
#nk=<kernel name>:<number>:<cu_name>.<cu_name>...
nk=vadd:2
```

Where:

- `<kernel_name>`: Specifies the name of the kernel to instantiate multiple times.
- `<number>`: The number of kernel instances, or CUs, to implement in hardware.
- `<cu_name>.<cu_name>...`: Specifies the instance names for the specified number of instances. This is optional, and the CU name will default to `kernel_1` when it is not specified.

Then the config file is specified on the `v++` command line:

```
v++ --config vadd_config.txt ...
```

In the `vadd` example above, the result is two instances of the `vadd` kernel, named `vadd_1` and `vadd_2`.



TIP: You can check the results by using the `xclbinutil` command to examine the contents of the `xclbin` file. Refer to [Xclbinutil Utility](#).

The following example results in three CUs of the `vadd` kernel, named `vadd_X`, `vadd_Y`, and `vadd_Z` in the `xclbin` binary file:

```
[connectivity]
nk=vadd:3:vadd_X.vadd_Y.vadd_Z
```

Mapping Kernel Ports to Global Memory

The link phase is when the memory ports of the kernels are connected to memory resources which include DDR, HBM, and PLRAM. By default, when the `xclbin` file is produced during the `v++` linking process, all kernel memory interfaces are connected to the same global memory bank. As a result, only one kernel interface can transfer data to/from the memory bank at one time, limiting the performance of the application due to memory access.

If the FPGA contains only one global memory bank, this is the only available approach. However, all of the Alveo Data Center accelerator cards contain multiple global memory banks. During the linking stage, you can specify which global memory bank each kernel port (or interface) is connected to. Proper configuration of kernel to memory connectivity is important to maximize bandwidth, optimize data transfers, and improve overall performance. Even if there is only one compute unit in the device, mapping its input and output ports to different global memory banks can improve performance by enabling simultaneous accesses to input and output data.

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

1. Specify the kernel interface with different `bundle` names as discussed in [Kernel Interfaces and Memory Banks](#).
2. During `v++` linking, use the `connectivity.sp` option in a config file to map the kernel port to the desired memory bank.

To map the kernel ports to global memory banks using the `connectivity.sp` option of the `v++` config file, use the following steps.

1. Starting with the kernel code example from [Kernel Interfaces and Memory Banks](#):

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

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

Note that the memory interface inputs `pixel` and `weights` are assigned different bundle names in the example above. This creates two separate ports that can be assigned to separate global memory banks.



IMPORTANT! You must specify `bundle=` names using all lowercase characters to be able to assign it to a specific memory bank using the `connectivity.sp` option.

2. During `v++` linking, the separate ports can be mapped to different global memory banks. Edit a config file to include the `connectivity.sp` option, and specify it in the `v++` command line with the `--config` option, as described in [Vitis Compiler Command](#).

For example, for the `cnn` kernel shown above, the `connectivity.sp` option in the config file would be as follows:

```
[connectivity]
#sp=<compute_unit_name>.<interface_name>:<bank name>
sp=cnn_1.m_axi_gmem:DDR[0]
sp=cnn_1.m_axi_gmem1:DDR[1]
```

Where:

- `<compute_unit_name>` is an instance name of the CU as determined by the `connectivity.nk` option, described in [Creating Multiple Instances of a Kernel](#), or is simply `<kernel_name>_1` if multiple CUs are not specified.
- `<interface_name>` is the name of the kernel port as defined by the HLS INTERFACE pragma, including `m_axi_` and the bundle name. In the `cnn` kernel above, the ports would be `m_axi_gmem` and `m_axi_gmem1`.



TIP: If the port is not specified as part of a bundle, then the `<interface_name>` is simply the specified `port = name`, without the `m_axi_` prefix.

- `<bank_name>` is denoted as `DDR[0]`, `DDR[1]`, `DDR[2]`, and `DDR[3]` for a platform with four DDR banks. Some platforms also provide support for PLRAM and HBM memory banks, in which case you would use `PLRM[0]` or `HBM[0]` for example.



TIP: You can use the `platforminfo` utility to get information on the global memory banks available in a specified platform. Refer to [Platforminfo](#) for more information.

The customized bank connection needs to be reflected in the host code as well. This was previously discussed in [Buffer Allocation on the Device](#).

Note: Up to 15 kernel ports can be connected to a single global memory bank. Therefore, if there are more than 15 memory interfaces, then you must explicitly perform the memory mapping as described here, using the `connectivity.sp` option to distribute connections across different memory banks.

Specify Streaming Connections Between Compute Units

The Vitis core development kit supports streaming data transfer between two kernels, allowing data to move directly from one kernel to another without having to transmit back through global memory. However, the process has to be implemented in the kernel code itself, as described in [Streaming Data Transfers Between Kernels \(K2K\)](#), and also specified during the kernel build process.

Note: This also supports streaming connections to/from free running kernels as described in [Free-running Kernel](#).

The streaming data ports of kernels can be connected during `v++` linking using the `connectivity.stream_connect` option in a config file that is specified using the `--config` option, as described in [Vitis Compiler Command](#).

To connect the streaming output port of a producer kernel to the streaming input port of a consumer kernel, setup the connection in the `v++` config file using the `connectivity.stream_connect` option as follows:

```
[connectivity]
#stream_connect=<cu_name>.<output_port>:<cu_name>.<input_port>
stream_connect=vadd_1.stream_out:vadd_2.stream_in
```

Where:

- `<cu_name>` is an instance name of the CU as determined by the `connectivity.nk` option, described in [Creating Multiple Instances of a Kernel](#).
- `<output_port>` or `<input_port>` is the streaming port defined in the producer or consumer kernel as described in [Streaming Kernel Coding Guidelines](#), or as described in [Coding Guidelines for Free Running Kernels](#).

The config file is specified on the `v++` command line during the link process:

```
v++ -l --config vadd_config.txt ...
```

Assigning Compute Units to SLRs

Currently, the Xilinx devices on Alveo Data Center accelerator cards use stacked silicon devices with several Super Logic Regions (SLRs) to provide device resources, including global memory. When assigning ports to global memory banks, as described in [Mapping Kernel Ports to Global Memory](#), it is best that the CU instance is assigned to the same SLR as the global memory it is connected to. In this case, you will want to manually assign the kernel instance, or CU into the same SLR as the global memory to ensure the best performance.

A CU can be assigned into an SLR during the `v++` linking process using the `connectivity.slr` option in a config file, and specified with the `--config` option in the `v++` command line. The syntax of the `connectivity.slr` option in the config file is as follows:

```
[connectivity]
#slr=<compute_unit_name>:<slr_ID>
slr=vadd_1:SLR2
slr=vadd_2:SLR3
```

where:

- `<compute_unit_name>` is an instance name of the CU as determined by the `connectivity.nk` option, described in [Creating Multiple Instances of a Kernel](#), or is simply `<kernel_name>_1` if multiple CUs are not specified.
- `<slr_ID>` is the SLR number to which the CU is assigned, in the form SLR0, SLR1,...

The assignment of a CU to an SLR must be specified for each CU separately, but is not required. In the absence of an SLR assignment, the `v++` linker is free to assign the CU to any SLR.

After editing the config file to include the SLR assignments, you can use it during the `v++` linking process by specifying the config file using the `--config` option:

```
v++ -l --config config_slr.txt ...
```

Controlling Report Generation

The `v++ -R` option (or `--report_level`) controls the level of information to report during compilation or linking for hardware emulation and system targets. Builds that generate fewer reports will typically run more quickly.

The command line option is as follows:

```
$ v++ -R <report_level>
```

Where `<report_level>` is one of the following options:

- `-R0`: Minimal reports and no intermediate design checkpoints (DCP).
- `-R1`: Includes R0 reports plus:
 - Identifies design characteristics to review for each kernel (`report_failfast`).
 - Identifies design characteristics to review for the full post-optimization design.
 - Saves post-optimization design checkpoint (DCP) file for later examination or use in the Vivado Design Suite.
- `-R2` : Includes R1 reports plus:
 - Includes all standard reports from the Vivado tools, including saved DCPs after each implementation step.
 - Design characteristics to review for each SLR after placement.



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

Directory Structure

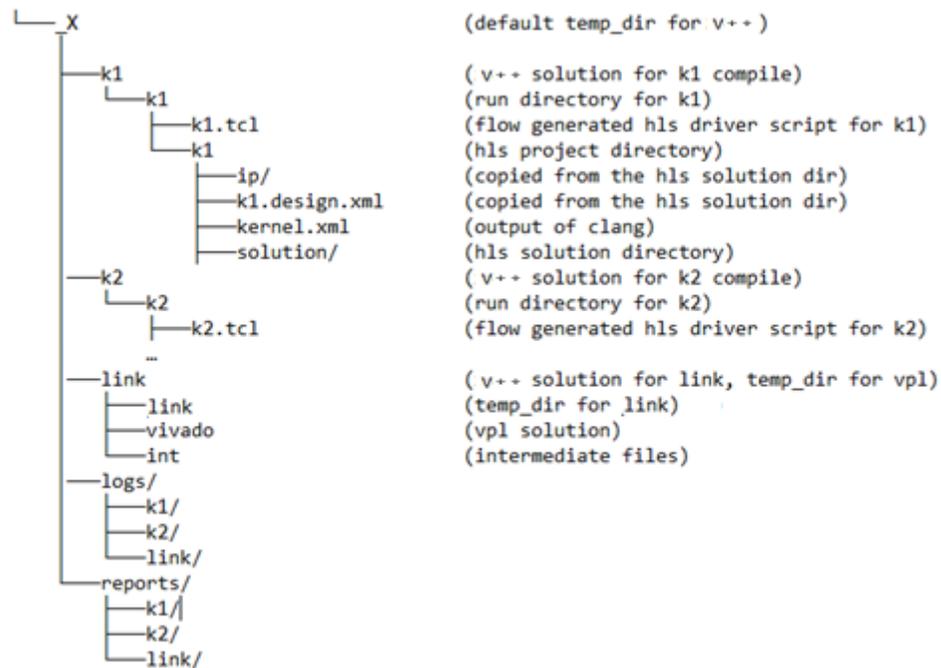
The directory structure generated by the command-line flow, and the IDE flow, has been organized to allow you to easily find and access files. By navigating the various `compile`, `link`, `logs`, and `reports` directories, you can easily find generated files. Similarly, each kernel will also have a directory structure created.

Output Directories from the v++ Command

When using `v++` 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 v++ compile runs (k1 and k2) and one v++ 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 37: Command Line Directory Structure



You can optionally change the directory structure using the following v++ options:

```
--log_dir <dir_name>
```

```
--report_dir <dir_name>
```

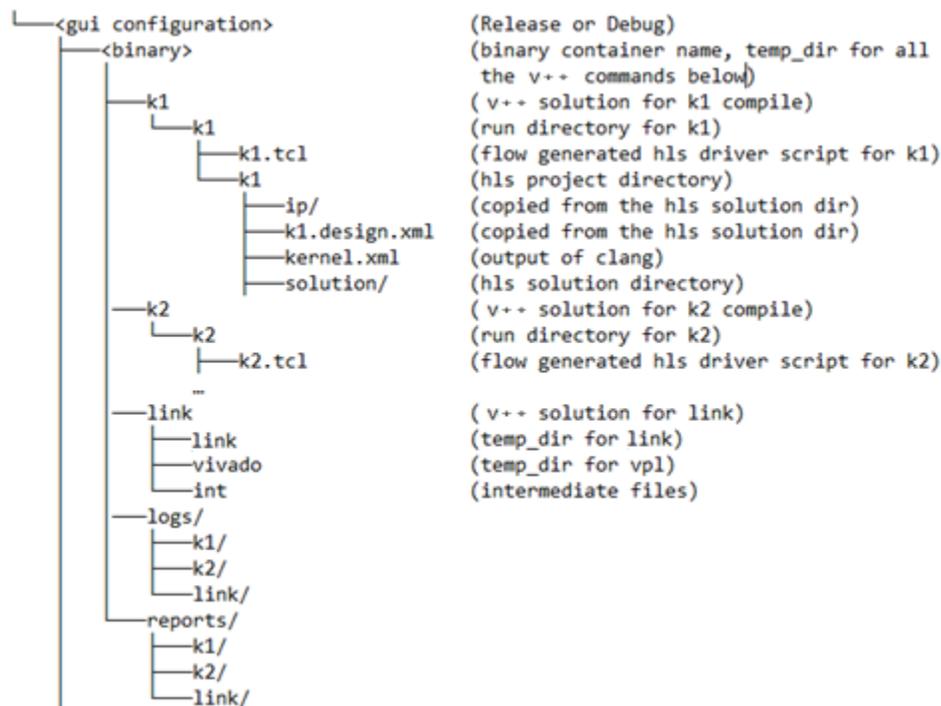
```
--temp_dir <dir_name>
```

See [Vitis Compiler Command](#) for additional details.

Output Directories from the Vitis IDE

The default directory structure of the output files from the Vitis IDE flow, though similar, is not identical to that created by the command-line flow. The following example shows the directory structure automatically generated by the Vitis IDE for two kernels, k1 and k2, compiled and linked by the `v++` command. 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 38: GUI Directory Structure



The Vitis IDE manages the creation of the directory structure using the following `v++` command options, which can be specified through the GUI:

```
--temp_dir
```

```
--report_dir
```

```
--log_dir
```

Refer to [Chapter 6: Using the Vitis IDE](#) for more information.

Running an Application



TIP: Setup the command shell or window as described in [Setting up the Vitis Integrated Design Environment](#) prior to running the builds.

As explained in [Build Targets](#), there are three different types of builds you can target in the Vitis core development kit: software emulation, hardware emulation, and the system hardware build. Each of these build targets offers advantages, and limitations, and it is recommended that you work your way through building and running each of these targets.

- **Software emulation:** The software emulation build compiles and links quickly, runs on the x86 system as a compiled C-model, and lets you quickly iterate on both the host code and kernel logic.
- **Hardware emulation:** The host program runs as before, but the kernel code is compiled into an RTL behavioral model which is run in the Vivado simulator. This build and run loop takes longer but provides a cycle-accurate view of the kernel logic.
- **System hardware:** The host program runs as before, but this time connected with the actual accelerator card, running the FPGA binary produced for your application. The performance data and results you capture here are the actual performance of your accelerated application. Yet this run might still reveal opportunities to optimize your design.

Running Emulation Builds

1. Edit the `xrt.ini` file required by XRT. Optional, but recommended.

As described in [xrt.ini File](#), the `xrt.ini` file specifies various parameters to control debugging, profiling, and message logging in XRT when running the host application and kernel execution. This enables the runtime to capture debugging and profile information as the application is running. The `Emulation` group in the `xrt.ini` provides features that affect your emulation run.



TIP: Be sure to use the `g++ -g` option when compiling your kernel code for emulation mode.

2. Create the `emconfig.json` file required for running emulation as described in [Emconfigutil](#). Required.

The emulation configuration file, `emconfig.json`, is generated from the specified platform using the `emconfigutil` command, and provides information used by the Xilinx runtime library during emulation. The following example creates the `emconfig.json` file for the specified target platform:

```
emconfigutil --platform xilinx_u200_xdma_201830_2
```



TIP: It is mandatory to have an up-to-date `.json` file for running emulation on your target platform.

3. Set the `XCL_EMULATION_MODE` environment variable to `sw_emu` (software emulation) or `hw_emu` (hardware emulation) as appropriate. This changes the application execution to emulation mode. In emulation mode, the runtime looks for the `emconfig.json` file in the same directory as the host executable, and reads in the target configuration for the emulation runs. Required.

Use the following syntax to set the environment variable for C shell:

```
setenv XCL_EMULATION_MODE sw_emu
```

Bash shell:

```
export XCL_EMULATION_MODE=sw_emu
```



IMPORTANT! *The emulation targets will not run if the `XCL_EMULATION_MODE` environment variable is not properly set.*

4. Run the application.

With the runtime initialization (`xrt.ini`), emulation configuration file (`emconfig.json`), and the `XCL_EMULATION_MODE` environment set, run the host executable with the desired command line argument, as you normally would run it if it was not an emulation run.. For example:

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



TIP: *This command line assumes that the host program is written to take the name of the `xclbin` file as an argument, as most Vitis examples and tutorials do. However, your application may have the name of the `xclbin` file hard-coded into the host program, or may require a different approach to running the application.*

Running the System Hardware Build

Running the system hardware build will let you see the actual application running on an accelerator card, such as the Alveo Data Center accelerator card, or perhaps a prototype of an embedded processor platform. In order to use the accelerator card, you must have it installed as described in *Getting Started with Alveo Data Center Accelerator Cards* ([UG1301](#)).

Beyond the installation of the accelerator and XRT, you must unset the `XCL_EMULATION_MODE` environment variable prior to running the system hardware build. If you had it set for running emulation, you will need to unset it now.

With the runtime initialization (`xrt.ini`), and the `XCL_EMULATION_MODE` environment unset, run the application on hardware. For example:

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

Profiling, Optimizing, and Debugging the Application

Running the system, either in emulation or on the system hardware, presents a series of potential challenges and opportunities. Running the system for the first time, you can profile the application to identify bottlenecks, or performance issues that offer opportunities to optimize the design, as discussed in the sections below. Of course, running the application can also reveal coding errors, or design errors that need to be debugged to get the system running as expected.

Profiling the Application

The Vitis™ core development kit 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 Vitis: System resource usage and performance, kernel optimization, host optimization, and PCIe® bandwidth optimization. The following Vitis reports and graphical tools support your efforts to profile and optimize these areas:

- [Design Guidance](#)
- [System Estimate Report](#)

- [HLS Report](#)
- [Profile Summary Report](#)
- [Application Timeline](#)
- [Waveform View and Live Waveform Viewer](#)

Reports are automatically generated after running the active build, either from the command line as described in [Running an Application](#), or from the Vitis integrated design environment (IDE). Separate sets of reports are generated for all three build targets and can be found in the respective report directories. Refer to [Directory Structure](#) for more information on locating these reports.

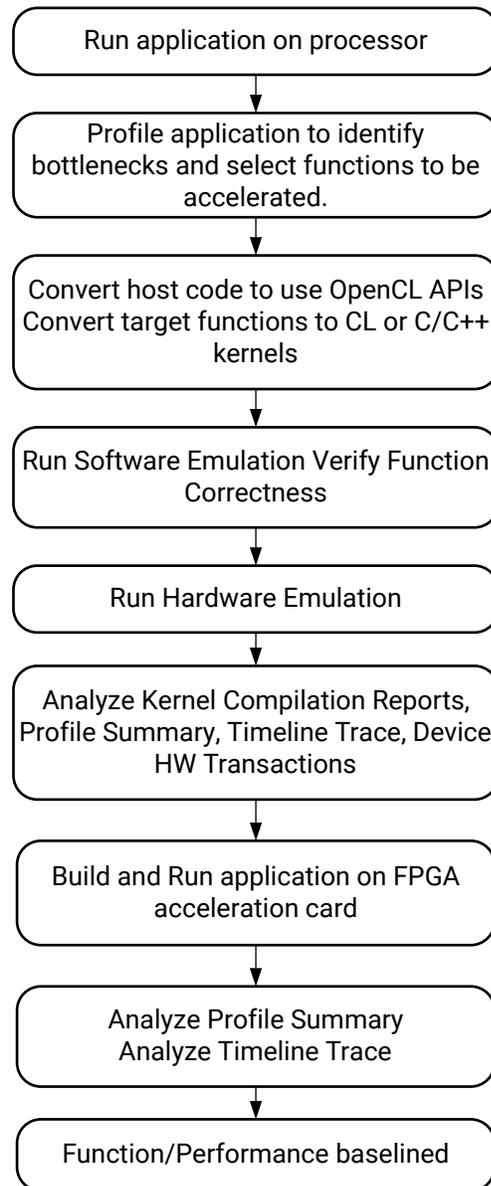
Reports can be viewed in a web browser, a spreadsheet viewer, or from the Vitis IDE. To access these reports from the Vitis IDE, ensure the Assistant view is visible and double-click the desired report.

The following topics briefly describe the various reports and graphical visualization tools and how they can be used to profile your design.

Baselining Functionalities and Performance

It is very important to understand the performance of your application before you start any optimization effort. This is achieved by establishing a baseline for the application in terms of functionalities and performance.

Figure 39: **Baselining Functionalities and Performance Flow**



X22238-082719

Identify Bottlenecks

The first step is to identify the bottlenecks of the current application running on your existing platform. The most effective way is to run the application with profiling tools, like `valgrind`, `callgrind`, and `GNU gprof`. The profiling data generated by these tools show the call graph with the number of calls to all functions and their execution time. The functions that consume the most execution time are good candidates to be offloaded and accelerated onto FPGAs.

Convert Target Functions

After the target functions are selected, convert them to OpenCL C kernels or C/C++ kernels without any optimization. Refer to [C/C++ Kernels](#) for more information. The application code calling these kernels will also need to be converted to use OpenCL APIs for data movement and task scheduling. Refer to [Host Application](#) for more information.



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

Run Software and Hardware Emulation

Run software and hardware emulation on the accelerated application as described in [Running an Application](#), to verify functional correctness, and to generate profiling data on the host code and the kernels. Analyze the kernel compilation reports, profile summary, timeline trace, and device hardware transactions to understand the baseline performance estimate for timing interval, and latency and resource utilization, such as DSP and block RAM.

Build and Run the Application

The last step in baselining is building and running the application on an FPGA acceleration card, like one of the Alveo™ Data Center accelerator cards. Analyze the reports from the system compilation, and the profiling data from application execution to see the actual performance and resource utilization on hardware.



TIP: Save all the reports during the baseline process, so that you can refer back to them and compare results during optimization.

Design Guidance

The Vitis core development kit has a comprehensive design guidance tool that provides immediate, actionable guidance to the software developer for issues detected in their designs. These issues might be related to the source code, or due to missed tool optimizations. Also, the rules are generic rules based on an extensive set of reference designs. Nevertheless, these rules might not be applicable for your specific design. Therefore, it is up to you to understand the specific guidance rules and take appropriate action based on your specific algorithm and requirements.

Guidance is generated from the Vivado HLS, Vitis profiler, and Vivado Design Suite when invoked by the `v++` compiler. 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 profile design guidance helps you interpret the profiling results which allows you to focus on improving performance.

The guidance includes hyperlinks, examples, and links to documentation. This helps improve productivity by quickly highlighting issues and directing you to additional information in using the Vitis technology.

There is one HTML guidance report for each run of the `v++` command, including compile and link. The report files are located in the `--report_dir` under the specific output name. For example:

- `v++_compile_<output>_guidance.html` for `v++` compilation
- `v++_link_<output>_guidance.html` for `v++` linking

Design guidance is automatically generated after building or running a design in the Vitis IDE.

You can open the Guidance report as discussed in [Chapter 7: Using the Vitis Analyzer](#). To access the Guidance report, open the Compile summary, the Link summary, or the Run summary, and open the Guidance report. With the Guidance report open, hovering over the guidance highlights recommended resolutions.

The following image shows an example of the Guidance report displayed in the Vitis analyzer. Clicking a link displays an expanded view of the actionable guidance.

Figure 40: Design Guidance Example

Name	Threshold	Actual	Details	Resolution
Host Data Transfer (3)				
HOST_MIGRATE_MEM (1)	> 0			
HOST_MIGRATE_MEM	> 0	2	Migrate Memory OpenCL APIs were used 2 time(s).	
HOST_READ_TRANSFER_SIZE (1)	> 4.096			
HOST_READ_TRANSFER_SIZE	> 4.096	1.024	Host read average size was 1.024 KB across 1 transfers.	Improve efficiency by consolidating read transfers. Click here .
HOST_WRITE_TRANSFER_SIZE (1)	> 4.096			
HOST_WRITE_TRANSFER_SIZE	> 4.096	2.048	Host write average size was 2.048 KB across 1 transfers.	Improve efficiency by consolidating write transfers. Click here .
Kernel Data Transfer (21)				
KERNEL_PORT_DATA_WIDTH (6)	= 512			
KERNEL_PORT_DATA_WIDTH	= 512	32	Port mmult_A/m_axi_gmem1 has a data width of 32.	
KERNEL_PORT_DATA_WIDTH	= 512	32	Port mmult_B/m_axi_gmem has a data width of 32.	
KERNEL_PORT_DATA_WIDTH	= 512	32	Port mmult_B/m_axi_gmem1 has a data width of 32.	
KERNEL_PORT_DATA_WIDTH	= 512	32	Port mmult_C/m_axi_gmem has a data width of 32.	
KERNEL_PORT_DATA_WIDTH	= 512	32	Port mmult_C/m_axi_gmem1 has a data width of 32.	
KERNEL_PORT_DATA_WIDTH	= 512	32	Port mmult_A/m_axi_gmem has a data width of 32.	
KERNEL_READ_TRANSFER_AMOUNT_MAX (2)	< 2.000			
KERNEL_READ_TRANSFER_AMOUNT_MAX	< 2.000	0.000	Total kernel read of 0.000000 MB on xilinx_u200_qdma_201910_1-0 was 0.0000% of host	
KERNEL_READ_TRANSFER_AMOUNT_MAX	< 2.000	1.000	Total kernel read of 0.002048 MB on xilinx_u200_qdma_201910_1-0 was 100.000% of host data.	
KERNEL_READ_TRANSFER_AMOUNT_MIN (2)	> 0.250			
KERNEL_READ_TRANSFER_AMOUNT_MIN	> 0.250	0.000	Total kernel read of 0.000000 MB on xilinx_u200_qdma_201910_1-0 was 0.0000% of host data.	Ensure compute units read data written by host. Click here .

Resolution

You can maximize the performance of your host write transfers by writing the required amount of data in less transfers.

Not recommended:

```
for (int i=0; i < 10; i++)
  clEnqueueWriteBuffer(queue, buffer+i*size,
    1, 0, size, hostPtr, ...);
```

Recommended:

```
clEnqueueWriteBuffer(queue, buffer,
  1, 0, 10*size, hostPtr, ...);
```

Command Line

When kernels are compiled and when the FPGA binary is linked, guidance reports are generated automatically by the `v++` command. You can view these reports in the Vitis analyzer by opening the `<output_filename>.compile.summary` or the `<output_filename>.link.summary` for the application project. The `<output_filename>` is the output of the `v++` command.

As an example, launch the Vitis analyzer and open the report using this command:

```
vitis_analyzer <output_filename>.link.summary
```

When the Vitis analyzer opens, it displays the link summary report, as well as the compile summary, and a collection of reports generated during the compile and link processes. Both the compile and link steps generate Guidance reports to view by clicking the **Build** heading on the left-hand side. Refer to [Chapter 7: Using the Vitis Analyzer](#) for more information.

Data Interpretation

The Guidance view places each entry in a separate row. Each row might contain the name of the guidance rule, threshold value, actual value, and a brief but specific description of the rule. The last field provides a link to reference material intended to assist in understanding and resolving any of the rule violations.

In the GUI Guidance view, guidance rules are grouped by categories and unique IDs in the Name column and annotated with symbols representing the severity. These are listed individually in the HTML report. In addition, as the HTML report does not show tooltips, a full Name column is included in the HTML report as well.

The following list describes all fields and their purpose as included in the HTML guidance reports.

- **Id:** Each guidance rule is assigned a unique ID. Use this id to uniquely identify a specific message from the guidance report.
- **Name:** The Name column displays a mnemonic name uniquely identifying the guidance rule. These names are designed to assist in memorizing specific guidance rules in the view.
- **Severity:** The Severity column allows the easy identification of the importance of a guidance rule.
- **Full Name:** The Full Name provides a less cryptic name compared to the mnemonic name in the Name column.
- **Categories:** Most messages are grouped within different categories. This allows the GUI to display groups of messages within logical categories under common tree nodes in the Guidance view.

- **Threshold:** The Threshold column displays an expected threshold value, which determines whether or not a rule is met. The threshold values are determined from many applications that follow good design and coding practices.
- **Actual:** The Actual column displays the values actually encountered on the specific design. This value is compared against the expected value to see if the rule is met.
- **Details:** The Details column provides a brief but specific message describing the specifics of the current rule.
- **Resolution:** The Resolution column provides a pointer to common ways the model source code or tool transformations can be modified to meet the current rule. Clicking the link brings up a pop-up window or the documentation with tips and code snippets that you can apply to the specific issue.

System Estimate Report

The process step with the longest execution time includes building the hardware system and the FPGA binary to run on Xilinx devices. Build time is also affected by the target device and the number of compute units instantiated onto the FPGA fabric. Therefore, it is useful to estimate the performance of an application without needing to build it for the system hardware.

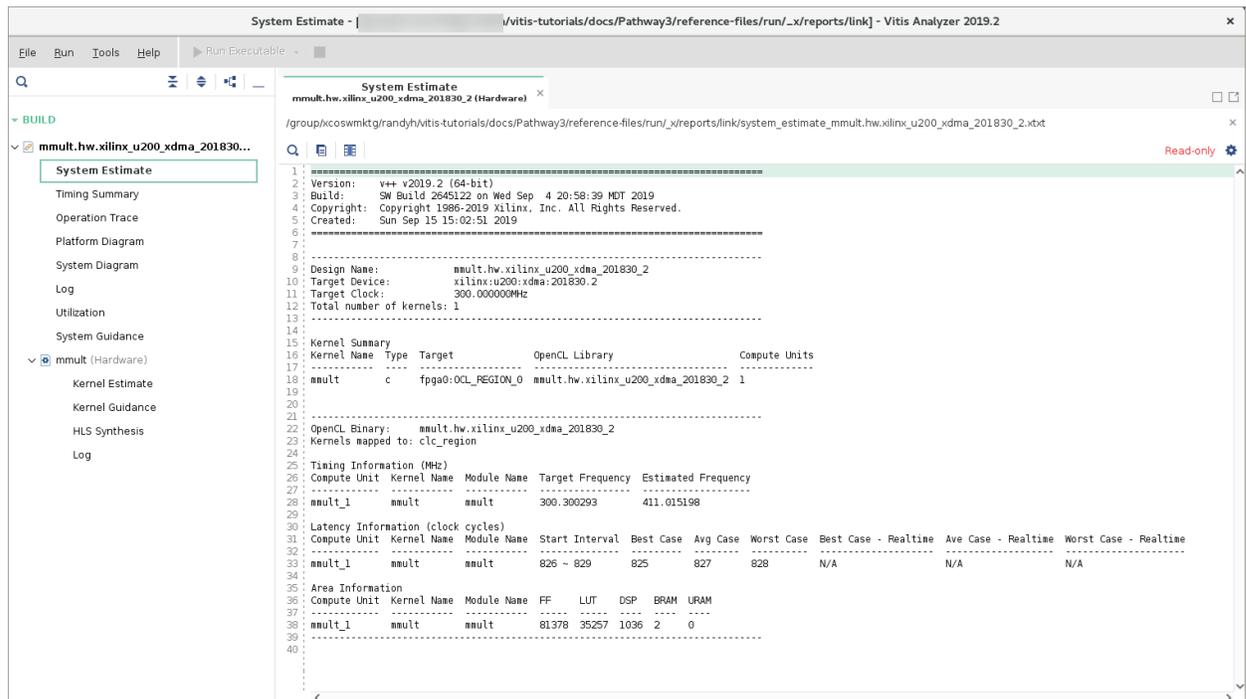
The System Estimate report provides estimates of FPGA resource usage and the frequency at which the hardware accelerated kernels can operate. The report is automatically generated for hardware emulation and system hardware builds. The report contains high-level details of the user kernels, including resource usage and estimated frequency. This report can be used to guide design optimization.

You can also force the generation of the System Estimate report with the following option:

```
v++ .. --report_level estimate
```

An example report is shown in the figure:

Figure 41: System Estimate



Command Line

The System Estimate report generated by the `v++` command provides information on every binary container in the application, as well as every compute unit in the design. The report is structured as follows:

- Target device information
- Summary of every kernel in the application
- Detailed information on every binary container in the solution

The System Estimate report can be opened in the Vitis analyzer tool, intended for viewing reports from the Vitis compiler when the application is built, and the XRT runtime when the application is run. You can launch the Vitis analyzer and open the report using the following command:

```
vitis_analyzer <output_filename>.link.summary
```

The `<output_filename>` is the output of the `v++` command. This opens the Link Summary for the application project in the Vitis analyzer tool. Then, select the System Estimate report. Refer to [Chapter 7: Using the Vitis Analyzer](#) for more information.



TIP: Because the System Estimate report is a text file, you can also view it in a text editor or target platform.

Data Interpretation

The following example report file represents the information generated for the estimate report:

```

-----
---
Design Name:                mmult.hw_emu.xilinx_u200_xdma_201830_2
Target Device:              xilinx:u200:xdma:201830.2
Target Clock:               300.000000MHz
Total number of kernels: 1
-----
---

Kernel Summary
Kernel Name  Type  Target                OpenCL
Library                                Compute Units
-----
mmult        c    fpga0:OCL_REGION_0
mmult.hw_emu.xilinx_u200_xdma_201830_2  1
-----

OpenCL Binary:             mmult.hw_emu.xilinx_u200_xdma_201830_2
Kernels mapped to:        clc_region

Timing Information (MHz)
Compute Unit  Kernel Name  Module Name  Target Frequency  Estimated
Frequency
-----
mmult_1      mmult        mmult        300.300293        411.015198

Latency Information (clock cycles)
Compute Unit  Kernel Name  Module Name  Start Interval  Best Case  Avg
Case  Worst Case
-----
mmult_1      mmult        mmult        826 ~ 829        825
827          828

Area Information
Compute Unit  Kernel Name  Module Name  FF      LUT      DSP      BRAM      URAM
-----
mmult_1      mmult        mmult        81378   35257   1036     2         0
-----
---

```

Design and Target Device Summary

All design estimate reports begin with an application summary and information about the target device. The device information is provided in the following section of the report:

```

-----
---
Design Name:          mmult.hw_emu.xilinx_u200_xdma_201830_2
Target Device:       xilinx:u200:xdma:201830.2
Target Clock:        300.000000MHz
Total number of kernels: 1
-----
---
```

For the design summary, the information provided includes the following:

- **Target Device:** Name of the Xilinx device on the target platform that runs the FPGA binary built by the Vitis compiler.
- **Target Clock:** Specifies the target operating frequency for the compute units (CUs) mapped to the FPGA fabric.

Kernel Summary

This section lists all of the kernels defined for the application project. The following example shows the kernel summary:

```

Kernel Summary
Kernel Name  Type  Target                OpenCL
Library                                Compute Units
-----
mmult        c      fpga0:OCL_REGION_0
mmult.hw_emu.xilinx_u200_xdma_201830_2  1
```

In addition to the kernel name, the summary also provides the execution target and type of the input source. Because there is a difference in compilation and optimization methodology for OpenCL™, C, and C/C++ source files, the type of kernel source file is specified.

The Kernel Summary section is the last summary information in the report. From here, detailed information on each compute unit binary container is presented.

Timing Information

For each binary container, the detail section begins with the execution target of all compute units (CUs). It also provides timing information for every CU. As a general rule, if the estimated frequency for the FPGA binary is higher than the target frequency, the CU will be able to run in the device. If the estimated frequency is below the target frequency, the kernel code for the CU needs to be further optimized to run correctly on the FPGA fabric. This information is shown in the following example:

```
OpenCL Binary:      mmult.hw_emu.xilinx_u200_xdma_201830_2
Kernels mapped to: clc_region

Timing Information (MHz)
Compute Unit  Kernel Name  Module Name  Target Frequency  Estimated
Frequency
-----
mmult_1      mmult        mmult        300.300293        411.015198
```

It is important to understand the difference between the target and estimated frequencies. CUs are not placed in isolation into the FPGA fabric. CUs are placed as part of a valid FPGA design that can include other components defined by the device developer to support a class of applications.

Because the CU custom logic is generated one kernel at a time, an estimated frequency that is higher than the target frequency indicates that the CU can run at the higher estimated frequency. Therefore, CU should meet timing at the target frequency during implementation of the FPGA binary.

Latency Information

The latency information presents the execution profile of each CU in the binary container. When analyzing this data, it is important to recognize that all values are measured from the CU boundary through the custom logic. In-system latencies associated with data transfers to global memory are not reported as part of these values. Also, the latency numbers reported are only for CUs targeted at the FPGA fabric. The following is an example of the latency report:

```
Latency Information (clock cycles)
Compute Unit  Kernel Name  Module Name  Start Interval  Best Case  Avg
Case Worst Case
-----
mmult_1      mmult        mmult        826 ~ 829        825
827          828
```

The latency report is divided into the following fields:

- Start interval
- Best case latency
- Average case latency

- Worst case latency

The start interval defines the amount of time that has to pass between invocations of a CU for a given kernel.

The best, average, and worst case latency numbers refer to how much time it takes the CU to generate the results of one ND Range data tile for the kernel. For cases where the kernel does not have data dependent computation loops, the latency values will be the same. Data dependent execution of loops introduces data specific latency variation that is captured by the latency report.

The interval or latency numbers will be reported as "undef" for kernels with one or more conditions listed below:

- OpenCL kernels that do not have explicit `reqd_work_group_size(x, y, z)`
- Kernels that have loops with variable bounds

Note: The latency information reflects estimates based on the analysis of the loop transformations and exploited parallelism of the model. These advanced transformations such as pipelining and data flow can heavily change the actual throughput numbers. Therefore, latency can only be used as relative guides between different runs.

Area Information

Although the FPGA can be thought of as a blank computational canvas, there are a limited number of fundamental building blocks available in each FPGA. These fundamental blocks (FF, LUT, DSP, block RAM) are used by the Vitis compiler to generate the custom logic for each CU in the design. The quantity of fundamental resources needed to implement the custom logic for a single CU determines how many CUs can be simultaneously loaded into the FPGA fabric. The following example shows the area information reported for a single CU:

```
Area Information
Compute Unit  Kernel Name  Module Name  FF      LUT      DSP      BRAM     URAM
-----
mmult_1      mmult        mmult        81378   35257    1036     2        0
-----
---
```

HLS Report



IMPORTANT! When the `--save-temps` option is specified, the HLS report and HLS guidance are only generated for hardware emulation and system builds for C and OpenCL kernels. It is not generated for software emulation or RTL kernels.

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 the hardware description language used for implementing the kernel logic on the FPGA. The report provides estimated FPGA resource usage, operating frequency, latency, and interface signals of the custom-generated hardware logic. These details provide many insights to guide kernel optimization.

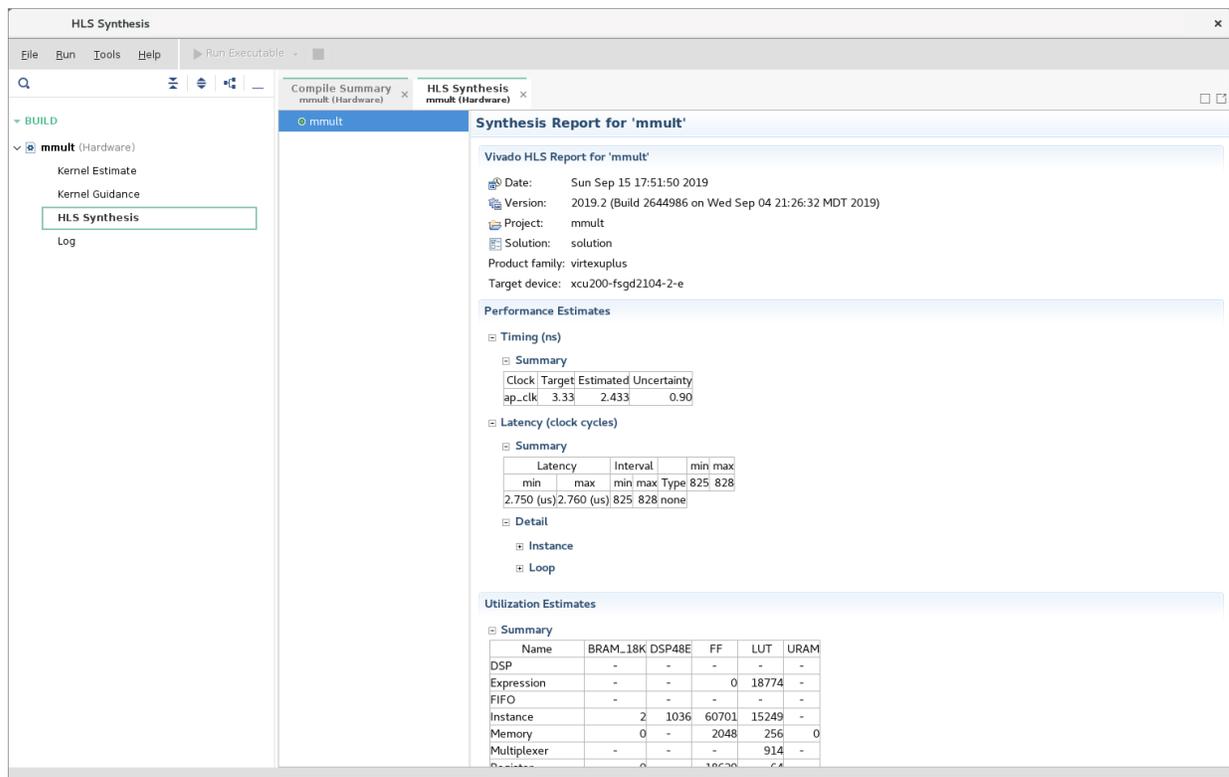
When running from the command line, this report can be found in the following directory:

```

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

The HLS report can be opened from the Vitis analyzer by opening the Compile Summary, or the Link Summary as described in [Chapter 7: Using the Vitis Analyzer](#). An example of the HLS report is shown.

Figure 42: HLS Report



Command Line

The HLS report can be viewed through the Vitis analyzer by opening the `<output_filename>.compile.summary` or the `<output_filename>.link.summary` for the application project. The `<output_filename>` is the output of the `v++` command.

You can launch the Vitis analyzer and open the report using the following command:

```
vitis_analyzer <output_filename>.compile.summary
```

When the Vitis analyzer opens, it displays the compile summary and a collection of reports generated during the compile process. The HLS report can be viewed by clicking the **Build** heading on the left-hand side. Refer to [Chapter 7: Using the Vitis Analyzer](#) for more information.

Data Interpretation

The left pane of the HLS report shows the module hierarchy. Each module generated as part of the HLS run is represented in this hierarchy. You can select any of these modules to present the synthesis details of the module in the right-side of the Synthesis Report window. The Synthesis Report is separated into several sections:

- General Information
- Performance Estimates (timing and latency)
- Utilization Estimates
- Interface Information

If this information is part of a hierarchical block, it will sum up the information of the blocks contained in the hierarchy. Therefore, the hierarchy can also be navigated from within the report when it is clear which instance contributes to the overall design.



CAUTION! *Regarding the absolute counts of cycles and latency, these numbers are based on estimates identified during HLS synthesis, especially with advanced transformations, such as pipelining and dataflow; these numbers might not accurately reflect the final results. If you encounter question marks in the report, this might be due to variable bound loops, and you are encouraged to set trip counts for such loops to have some relative estimates presented in this report.*

Profile Summary Report

When properly configured, the Vitis runtime collects profiling data on host applications and kernels. After the application finishes execution, the Profile Summary report is saved as a `.csv` file in the solution report directory (`--report_dir`), or the working directory where the Vitis compiler was launched.

The Profile Summary provides annotated details regarding the overall application performance. All data generated during the execution of the application is grouped into categories. The Profile Summary lets you examine kernel execution and data transfer statistics.



TIP: *The Profile Summary report can be generated for all build configurations. However, with the software emulation build, the report will not include any data transfer details under kernel execution efficiency and data transfer efficiency. This information is only generated in hardware emulation or system builds.*

An example of the Profile Summary report is shown below.

Figure 43: Profile Summary

Run Summary		Profile Summary							
mmult.hw_emu.xclbin		mmult.hw_emu.xclbin							
Top Operations		Kernels & Compute Units		Data Transfers		OpenCL APIs			
▼ Top Data Transfer: Kernels to Global Memory									
Device	Compute Unit	Number Of Transfers	Average Bytes per Transfer	Transfer Efficiency (%)	Total Data Transfer (MB)	Total Write (MB)	Total Read (MB)	Total Transfer Rate (MB/s)	
xilinx_u200_qdma_201910_1-0	mmult_B	48	64.000	1.563	0.003	0.001	0.002	1058.090	
xilinx_u200_qdma_201910_1-0	mmult_A	0	0.0	0.0	0.0	0.0	0.0	0.0	
▼ Top Kernel Execution									
Kernel Instance Address	Kernel	Context ID	Command Queue ID	Device	Start Time (ms)	Duration (ms)	Global Work Size	Local Work Size	
0xc74020	mmult	0	0	xilinx_u200_qdma_201910_1-0	0.033	0.006	1:1:1	1:1:1	
▼ Top Memory Writes: Host to Global Memory									
Buffer Address	Context ID	Command Queue ID	Start Time (ms)	Duration (ms)	Buffer Size (KB)	Writing Rate (MB/s)			
0x5000000000	0	0	281785.000	N/A	2.048	N/A			
▼ Top Memory Reads: Host to Global Memory									
Buffer Address	Context ID	Command Queue ID	Start Time (ms)	Duration (ms)	Buffer Size (KB)	Reading Rate (MB/s)			
0x5000002000	0	0	283889.000	N/A	1.024	N/A			

The report has multiple tabs that can be selected. A description of each tab is given here:

- **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 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 details on the profile summary, see [Data Interpretation](#).

Command Line

Capturing the data required for the Profile Summary requires two steps prior to actually running the application.

1. The FPGA binary (`xclbin`) file is configured for capturing profiling data by using the `v++ --profile_kernel` option during the linking process. See the [Vitis Compiler Command](#) for complete details. The following shows an example command:

```
v++ -g -l --profile_kernel data:all:all:all ...
```

Note: The `--profile_kernel` option is additive and can be used multiple times on the command line to specify individual kernels, CUs, and interfaces.

- The runtime requires the presence of an `xrt.ini` file, as described in [xrt.ini File](#), that includes the keyword for capturing profiling data:

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

With the two steps done in advance, the runtime creates the `profile_summary.csv` report file when running the application.

The CSV report can be viewed in a spreadsheet tool or utility, or can be opened in the Vitis analyzer tool, intended for viewing reports from the Vitis compiler when the application is built, and the XRT runtime when the application is run. You can launch the Vitis analyzer and open the report using the following command:

```
vitis_analyzer profile_summary.csv
```

Related Information

- [Running an Application](#)
- [Directory Structure](#)
- [Chapter 7: Using the Vitis Analyzer](#)

Data Interpretation

The Profile Summary includes a number of useful statistics for your OpenCL application. This provides a general idea of the functional bottlenecks in your application. The Profile Summary consists of four sections with the following information:

1. Top Operations

- **Top Data Transfer: Kernels and Global Memory:** This table displays the profile data for top data transfers between FPGA and device memory.
 - **Device:** Name of device
 - **Compute Unit:** Name of compute unit
 - **Number of Transfers:** Sum of write and read AXI transactions monitored on device
 - **Average Bytes per Transfer:** (Total read bytes + Total write bytes) / (Total read AXI transactions + Total write AXI transactions)
 - **Transfer Efficiency (%):** (Average bytes per transfer) / min (4K, (Memory bit width / 8 * 256))
 AXI4 specification limits the maximum burst length to 256 and maximum burst size to 4K bytes.
 - **Total Data Transfer (MB):** (Total read bytes + Total write bytes) / 1.0e6
 - **Total Write (MB):** (Total write bytes) / 1.0e6

- **Total Read (MB):** (Total read bytes) / 1.0e6
- **Transfer Rate (MB/s):** (Total data transfer) / (Compute unit total time)
- **Top Kernel Execution**
 - **Kernel Instance Address:** Host address of kernel instance (in hex)
 - **Kernel:** Name of kernel
 - **Context ID:** Context ID on host
 - **Command Queue ID:** Command queue ID on host
 - **Device:** Name of device where kernel was executed (format: <device>-<ID>)
 - **Start Time (ms):** Start time of execution (in ms)
 - **Duration (ms):** Duration of execution (in ms)
 - **Global Work Size:** NDRange of kernel
 - **Local Work Size:** Work group size of kernel
- **Top Memory Writes: Host and Device Global Memory**
 - **Buffer Address:** Host address of buffer (in hex)
 - **Context ID:** Context ID on host
 - **Command Queue ID:** Command queue ID on host
 - **Start Time (ms) :** Start time of write transfer (in ms)
 - **Duration (ms):** Duration of write transfer (in ms)
 - **Buffer Size (KB):** Size of write transfer (in KB)
 - **Writing Rate (MB/s):** Writing rate = (Buffer size) / (Duration)
- **Top Memory Reads: Host and Device Global Memory**
 - **Buffer Address:** Host address of buffer (in hex)
 - **Context ID:** Context ID on host
 - **Command Queue ID:** Command queue ID on host
 - **Start Time (ms):** Start time of read transfer (in ms)
 - **Duration (ms):** Duration of read transfer (in ms)
 - **Buffer Size (KB):** Size of read transfer (in KB)
 - **Reading Rate (MB/s):** Reading rate = (Buffer size) / (Duration)

2. Kernels & Compute Units

- **Kernel Execution (includes estimated device times):** This displays the profile data summary for all kernel functions scheduled and executed.
 - **Kernel:** Name of kernel
 - **Number of Enqueues:** Number of times kernel is enqueued
 - **Total Time (ms):** Sum of runtimes of all enqueues (measured from START to END in OpenCL execution model)
 - **Minimum Time (ms):** Minimum runtime of all enqueues
 - **Average Time (ms):** (Total time) / (Number of enqueues)
 - **Maximum Time (ms):** Maximum runtime of all enqueues
- **Compute Unit Utilization (includes estimated device times):** This displays the summary profile data for all compute units on the FPGA.
 - **Device:** Name of device (format: <device>-<ID>)
 - **Compute Unit:** Name of compute unit
 - **Kernel:** Kernel this compute unit is associated with
 - **Global Work Size:** NDRange of kernel (format is x:y:z)
 - **Local Work Size:** Local work group size (format is x:y:z)
 - **Number of Calls:** Number of times the compute unit is called
 - **Total Time (ms):** Sum of runtimes of all calls
 - **Minimum Time (ms):** Minimum runtime of all calls
 - **Average Time (ms):** (Total time) / (Number of work groups)
 - **Maximum Time (ms):** Maximum runtime of all calls
 - **Clock Frequency (MHz):** Clock frequency used for a given accelerator (in MHz)

3. Data Transfers

- **Data Transfer: Host and Global Memory:** This displays the profile data for all read and write transfers between the host and device memory via PCI Express® link.
 - **Context: Number of Devices:** Context ID and number of devices in context
 - **Transfer Type:** READ or WRITE
 - **Number of Transfers:** Number of host data transfers
 - Note:* Can contain `printf` transfers
 - **Transfer Rate (MB/s):** (Total bytes sent) / (Total time in μ s)
where Total time includes software overhead

- **Average Bandwidth Utilization (%)**: $(\text{Transfer rate}) / (\text{Maximum transfer rate})$
 where Maximum transfer rate = $(256 / 8 \text{ bytes}) * (300 \text{ MHz}) = 9.6 \text{ GB/s}$
- **Average Size (KB)**: $(\text{Total KB sent}) / (\text{Number of transfers})$
- **Total Time (ms)**: Sum of transfer times
- **Average Time (ms)**: $(\text{Total time}) / (\text{Number of transfers})$
- **Data Transfer: Kernels and Global Memory**: This displays the profile data for all read and write transfers between the FPGA and device memory.
 - **Device**: Name of device
 - **Compute Unit/Port Name**: <Name of compute unit>/<Name of port>
 - **Kernel Arguments**: List of arguments connected to this port
 - **DDR Bank**: DDR bank number this port is connected to
 - **Transfer Type**: READ or WRITE
 - **Number of Transfers**: Number of AXI transactions monitored on device
Note: Might contain `printf` transfers)
 - **Transfer Rate (MB/s)**: $(\text{Total bytes sent}) / (\text{Compute unit total time})$
 - Compute unit total time = Total execution time of compute unit
 - Total bytes sent = sum of bytes across all transactions
 - **Average Bandwidth Utilization (%)**: $(\text{Transfer rate}) / (0.6 * \text{Maximum transfer rate})$
 where Maximum transfer rate = $(512 / 8 \text{ bytes}) * (300 \text{ MHz}) = 19200 \text{ MB/s}$
 - **Average Size (KB)**: $(\text{Total KB sent}) / (\text{number of AXI transactions})$
 - **Average Latency (ns)**: $(\text{Total latency of all transaction}) / (\text{Number of AXI transactions})$
- 4. **OpenCL API Calls**: This displays the profile data for all OpenCL host API function calls executed in the host application.
 - **API Name**: Name of API function (for example, `clCreateProgramWithBinary`, `clEnqueueNDRangeKernel`)
 - **Number of Calls**: Number of calls to this API
 - **Total Time (ms)**: Sum of runtimes of all calls
 - **Minimum Time (ms)**: Minimum runtime of all calls
 - **Average Time (ms)**: $(\text{Total time}) / (\text{Number of calls})$
 - **Maximum Time (ms)**: Maximum runtime of all calls

Application Timeline

The Application Timeline collects and displays host and kernel events on a common timeline to help you understand and visualize the overall health and performance of your systems. The graphical representation lets you see issues regarding kernel synchronization and efficient concurrent execution. The displayed events include:

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

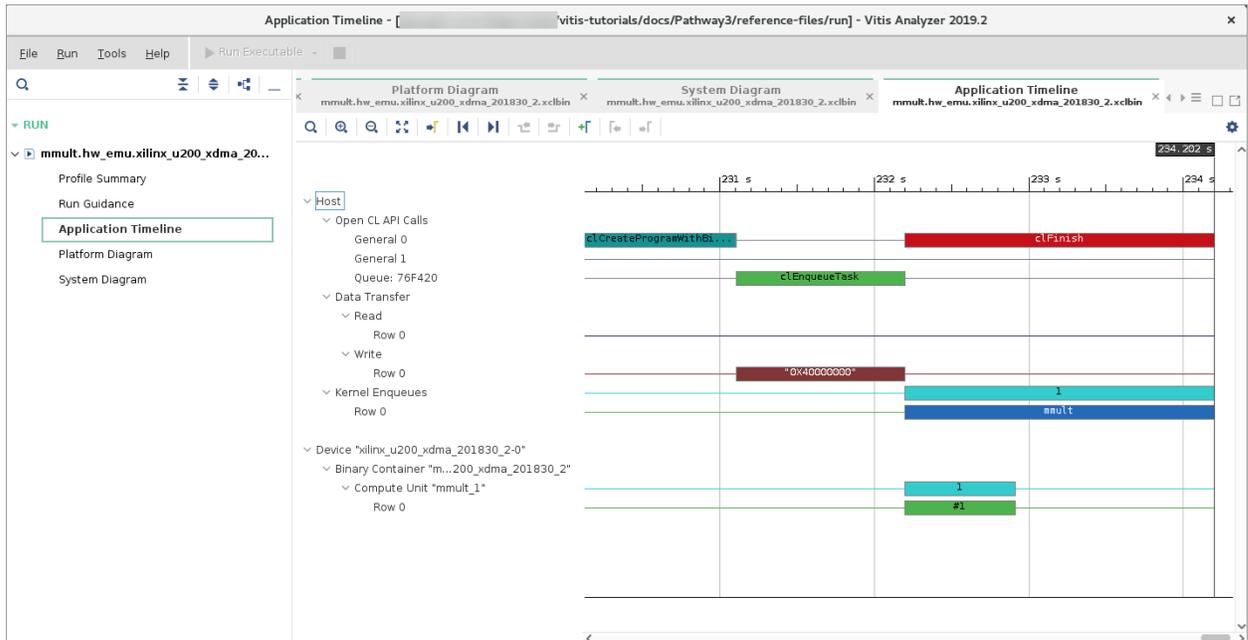
While this is useful for debugging and profiling the application, the timeline and device trace data are not collected by default because the runtime needs to periodically unload collected trace data from the FPGA, which can affect performance by adding time to the application execution. However, the trace data is collected with dedicated resources in the kernel, and does not affect kernel functionality.



TIP: *Turning on device profiling for the system build can negatively affect overall performance. This feature should be used for system performance debugging only. To collect data during system testing, update the runtime configuration as described in [xrt.ini File](#).*

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 44: Application Timeline



Timeline data can be enabled and collected through the command line flow. However, viewing must be done in the Vitis analyzer as described in [Chapter 7: Using the Vitis Analyzer](#).

Command Line

To generate the Application Timeline report, you must complete the following steps to enable timeline and device trace data collection in the command line flow:

1. Instrument the FPGA binary during linking, by adding Acceleration Monitors and AXI Performance Monitors to kernels using the `v++ --profile_kernel` option. This option has three distinct instrumentation options (`data`, `stall`, and `exec`), as described in the [Vitis Compiler Command](#). As an example, add `--profile_kernel` to the `v++` linking command line:

```
v++ -g -l --profile_kernel data:all:all:all ...
```

2. After the kernels are instrumented during the build process, data gathering must also be enabled during the application runtime execution by editing the `xrt.ini` file. Refer to [xrt.ini File](#) for more information.

The following `xrt.ini` file will enable maximum information gathering when the application is run:

```
[Debug]
profile=true
timeline_trace=true
data_transfer_trace=coarse
stall_trace=all
```



TIP: If you are collecting a large amount of trace data, you may need to use the `--trace_memory` with the `v++` command, and the `trace_buffer_size` keyword in the `xrt.ini`.

After running the application, the Application Timeline data is captured in a CSV file called `timeline_trace.csv`.

- The CSV report can be viewed in a spreadsheet tool or utility, or can be opened in the Vitis analyzer tool, intended for viewing reports from the Vitis compiler when the application is built, and the XRT when the application is run. You can launch the Vitis analyzer and open the report using the following command:

```
vitis_analyzer timeline_trace.csv
```

Data Interpretation

The Application Timeline window displays host and device events on a common timeline. This information helps you understand details of application execution and identify potential areas for improvements. The Application Timeline report has two main sections: Host and Device. The Host section shows the trace of all the activity originating from the host side. The Device section shows the activity of the CUs on the FPGA.

The report has the following structure:

- Host**
 - OpenCL API Calls:** All OpenCL API calls are traced here. The activity time is measured from the host perspective.
 - General:** All general OpenCL API calls such as `clCreateProgramWithBinary`, `clCreateContext`, and `clCreateCommandQueue`, are traced here.
 - Queue:** OpenCL API calls that are associated with a specific command queue are traced here. This includes commands such as `clEnqueueMigrateMemObjects`, and `clEnqueueNDRangeKernel`. If the user application creates multiple command queues, then this section shows all the queues and activities.
 - Data Transfer:** In this section the DMA transfers from the host to the device memory are traced. There are multiple DMA threads implemented in the OpenCL runtime and there is typically an equal number of DMA channels. The DMA transfer is initiated by the user application by calling OpenCL APIs such as `clEnqueueMigrateMemObjects`. These DMA requests are forwarded to the runtime which delegates to one of the threads. The data transfer from the host to the device appear under **Write** as they are written by the host, and the transfers from device to host appear under **Read**.

- **Kernel Enqueues:** The kernels enqueued by the host program are shown here. The kernels here should not be confused with the kernels/CUs on the device. Here kernel refers to the `NDRangeKernels` and tasks created by the OpenCL commands `clEnqueueNDRangeKernels` and `clEnqueueTask`. These are plotted against the time measured from the host's perspective. Multiple kernels can be scheduled to be executed at the same time, and they are traced from the point they are scheduled to run until the end of the kernel execution. This is the reason for multiple entries. The number of rows depend on the number of overlapping kernel executions.

Note: Overlapping of the kernels should not be mistaken for actual parallel execution on the device as the process might not be ready to execute right away.
- **Device "name"**
 - **Binary Container "name":** Binary container name.
 - **Accelerator "name":** Name of the compute unit (a.k.a., Accelerator) on the FPGA.
 - **User Functions:** In the case of the Vivado HLS tool kernels, functions that are implemented as data flow processes are traced here. The trace for these functions show the number of active instances of these functions that are currently executing in parallel. These names are generated in hardware emulation when waveform is enabled.

Note: Function level activity is only possible in hardware emulation.

 - **Function:** "name a"
 - **Function:** "name b"
 - **Read:** A CU reads from the DDR over AXI-MM ports. The trace of a data read by a CU is shown here. The activity is shown as transaction and the tool-tip for each transaction shows more details of the AXI transaction. These names are generated when `--profile_kernel data` is used where `(m_axi_<bundle name>(port))`.
 - **Write:** A compute unit writes to the DDR over AXI-MM ports. The trace of data written by a CU is shown here. The activity is shown as transactions and the tool-tip for each transaction shows more details of the AXI transaction. This is generated when `--profile_kernel data` is used where `m_axi_<bundle name>(port)`.

Waveform View and Live Waveform Viewer

The Vitis core development kit can generate a Waveform view when running hardware emulation. It displays in-depth details at the system-level, CU level, and at the function level. The details include data transfers between the kernel and global memory and data flow through inter-kernel pipes. These details provide many insights into performance bottlenecks from the system-level down to individual function calls to help optimize your application.

The Live Waveform Viewer is similar to the Waveform view, however, it provides even lower-level details with some degree of interactivity. The Live Waveform Viewer can also be opened using the Vivado logic simulator, `xsim`.

Note: The Waveform view allows you to examine the device transactions from within the Vitis analyzer, as described in [Chapter 7: Using the Vitis Analyzer](#). In contrast, the Live Waveform Viewer generates the Vivado simulation waveform viewer to examine the hardware transactions in addition to user selected signals.

Waveform View and Live Waveform Viewer data are not collected by default because it requires the runtime to generate simulation waveforms during hardware emulation, which consumes more time and disk space. Refer to [Command Line](#) for instructions on enabling these features.

Figure 45: Waveform View



The Live Waveform Viewer can be enabled in the `xrt.ini` file for use from the command line, as described in [xrt.ini File](#). You can also open the waveform database (`.wdb`) file with the Vivado logic simulator through the Linux command line:

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



TIP: Refer to [Directory Structure](#) for information on locating the `.wdb` file.

Command Line

Follow these instructions to enable waveform data collection from the command line during hardware emulation and open the viewer:

1. Enable debug code generation during compilation and linking using the `-g` option.

```
v++ -c -g -t hw_emu ...
```

2. Create an `xrt.ini` file in the same directory as the host executable with the following contents (see [xrt.ini File](#) for more information):

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

[Emulation]
launch_waveform=batch
```



IMPORTANT! For Live Waveform Viewer, add the following to the `xrt.ini`:

```
[Emulation]
launch_waveform=gui
```

3. Run the hardware emulation build of the application as described in [Running an Application](#). The hardware transaction data is collected in the waveform database file, `<hardware_platform>-<device_id>-<xclbin_name>.wdb`. Refer to [Directory Structure](#) for the location of this file.
4. Open the Waveform view in the Vitis analyzer as described in [Waveform View and Live Waveform Viewer](#).

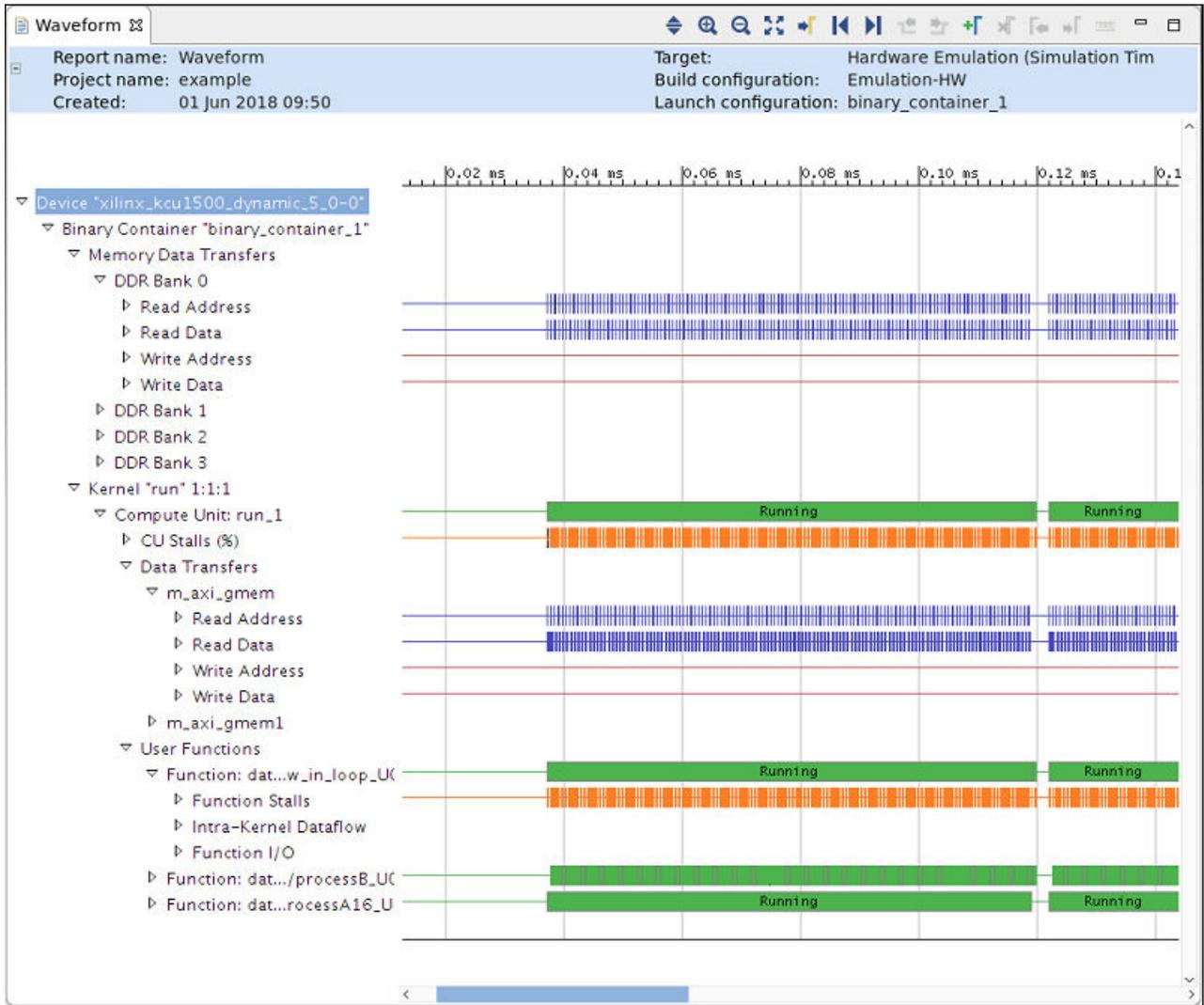


TIP: If Live Waveform Viewer is enabled, the simulation waveform opens during the hardware emulation run.

Data Interpretation Waveform View

The following image shows the Waveform view:

Figure 46: Waveform View



The waveform view is organized hierarchically for easy navigation.

Note: This viewer is based on the actual waveforms generated during hardware emulation (Kernel Trace). This allows the viewer to descend all the way down to the individual signals responsible for the abstracted data. However, as it is post processing the data, no additional signals can be added, and some of the runtime analysis such as DATAFLOW transactions cannot be visualized.

The hierarchy tree and descriptions are:

- **Device “name”:** Target device name.
- **Binary Container “name”:** Binary container name.
- **Memory Data Transfers:** For each DDR Bank, this shows the trace of all the read and write request transactions arriving at the bank from the host.

- **Kernel “name” 1:1:1:** For each kernel and for each compute unit of that kernel, this section breaks down the activities originating from the compute unit.
- **Compute Unit: “name”:** Compute unit name.
- **CU Stalls (%):** Stall signals are provided by the HLS tool to inform you when a portion of their circuit is stalling because of external memory accesses, internal streams (that is, dataflow), or external streams (that is, OpenCL pipes). The stall bus, shown in detailed kernel trace, compiles all of the lowest level stall signals and reports the percentage that are stalling at any point in time. This provides a factor of how much of the kernel is stalling at any point in the simulation.

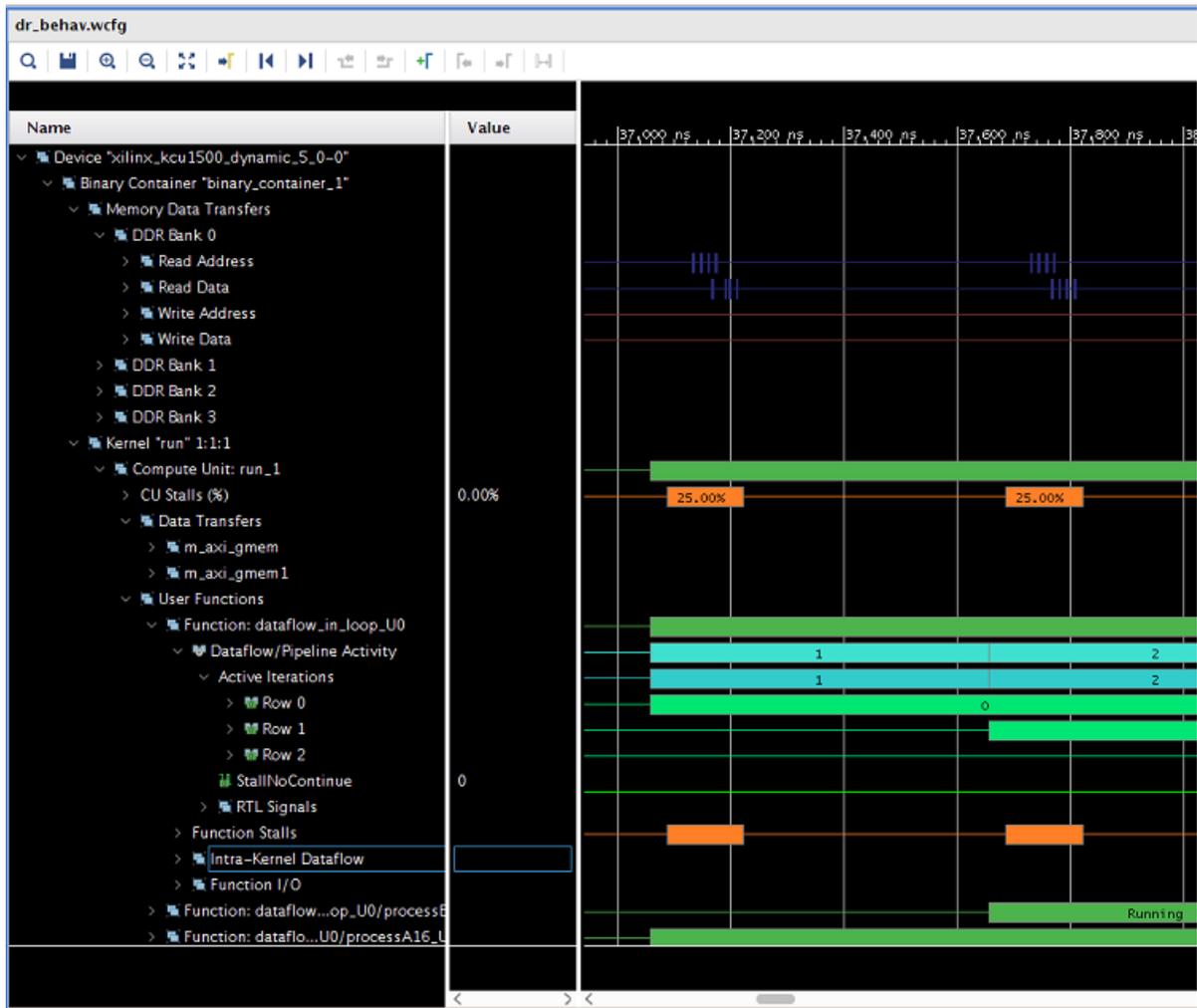
For example, if there are 100 lowest level stall signals and 10 are active on a given clock cycle, then the CU Stall percentage is 10%. If one goes inactive, then it is 9%.

- **Data Transfers:** This shows the read/write data transfer accesses originating from each Master AXI port of the compute unit to the DDR.
- **User Functions:** This information is available for the HLS tool kernels and shows the user functions. Function: <name>
- **Function Stalls:** Shows the different type stalls experienced by the process. It contains External Memory and Internal-Kernel Pipe stalls. The number of rows is dynamically incremented to accommodate the visualization of any concurrent execution.
- **Intra-Kernel Dataflow:** FIFO activity internal to the kernel.
- **Function I/O:** Actual interface signals.

Data Interpretation Live Waveform

The following figure shows the Live Waveform Viewer while running hardware emulation.

Figure 47: Live Waveform Viewer



The Live Waveform Viewer is organized hierarchically for easy navigation. Below are the hierarchy tree and descriptions.

Note: As the live waveform viewer is presented as part of the Vivado logic simulator (`xsim`) run, you can add extra signals and internals of the register transfer (RTL) design to the same view. Refer to the *Vivado Design Suite User Guide: Logic Simulation (UG900)* for more information.

- **Device "name":** Target device name.
- **Binary Container "name":** Binary container name.
- **Memory Data Transfers:** For each DDR Bank, this shows the trace of all the read and write request transactions arriving at the bank from the host.
- **Kernel "name" 1:1:1:** For each kernel and for each compute unit of that kernel, this section breaks down the activities originating from the compute unit.

- **Compute Unit: "name":** Compute unit name.
- **CU Stalls (%):** Stall signals are provided by the Vivado HLS tool to inform you when a portion of the circuit is stalling because of external memory accesses, internal streams (that is, dataflow), or external streams (that is, OpenCL pipes). The stall bus shown in detailed kernel trace compiles all of the lowest level stall signals and reports the percentage that are stalling at any point in time. This provides a factor of how much of the kernel is stalling at any point in the simulation.

For example, if there are 100 lowest level stall signals and 10 are active on a given clock cycle, then the CU Stall percentage is 10%. If one goes inactive, then it is 9%.

- **Data Transfers:** This shows the read/write data transfer accesses originating from each Master AXI port of the compute unit to the DDR.
- **User Functions:** This information is available for the HLS kernels and shows the user functions.
 - **Function: "name":** Function name.
 - **Dataflow/Pipeline Activity:** This shows the number of parallel executions of the function if the function is implemented as a dataflow process.
 - **Active Iterations:** This shows the currently active iterations of the dataflow. The number of rows is dynamically incremented to accommodate the visualization of any concurrent execution.
 - **StallNoContinue:** This is a stall signal that tells if there were any output stalls experienced by the dataflow processes (function is done, but it has not received a continue from the adjacent dataflow process).
 - **RTL Signals:** These are the underlying RTL control signals that were used to interpret the above transaction view of the dataflow process.
 - **Function Stalls:** Shows the different types of stalls experienced by the process.
 - **External Memory:** Stalls experienced while accessing the DDR memory.
 - **Internal-Kernel Pipe:** If the compute units communicated between each other through pipes, then this will show the related stalls.
 - **Intra-Kernel Dataflow:** FIFO activity internal to the kernel.
 - **Function I/O:** Actual interface signals.
 - **Function: "name":** Function name.
 - **Function: "name":** Function name.

Optimizing the Performance

Host Optimization

This section focuses on optimization of the host program, which uses the OpenCL™ API to schedule the individual compute unit executions, and data transfers to and from the FPGA board. As a result, you need to think about concurrent execution of tasks through the OpenCL command queue(s). This section discusses common pitfalls, and how to recognize and address them.

Reducing Overhead of Kernel Enqueuing

The OpenCL API execution model supports data parallel and task parallel programming models. Kernels are usually enqueued by the OpenCL runtime multiple times and then scheduled to be executed on the device. You must send the command to start the kernel in one of two ways:

- Using `clEnqueueNDRange` API for the data parallel case
- Using `clEnqueueTask` for the task parallel case

The dispatching process is executed on the host processor, and the kernel commands and arguments need to be sent to the accelerator, over the PCIe® bus in the case of the Alveo card for instance. In the Xilinx Runtime (XRT), the overhead of dispatching the command and arguments to the accelerator can be between 30 μ s and 60 μ s, depending the number of arguments on the kernel. You can reduce the impact of this overhead by minimizing the number of times the kernel needs to be executed.

For the data parallel case, Xilinx recommends that you carefully choose the global and local work sizes for your host code and kernel so that the global work size is a small multiple of the local work size. Ideally, the global work size is the same as the local work size as shown in the following code snippet:

```
size_t global = 1;
size_t local = 1;
clEnqueueNDRangeKernel(world.command_queue, kernel, 1, nullptr,
                        &global, &local, 2, write_events.data(),
                        &kernel_events[0]);
```

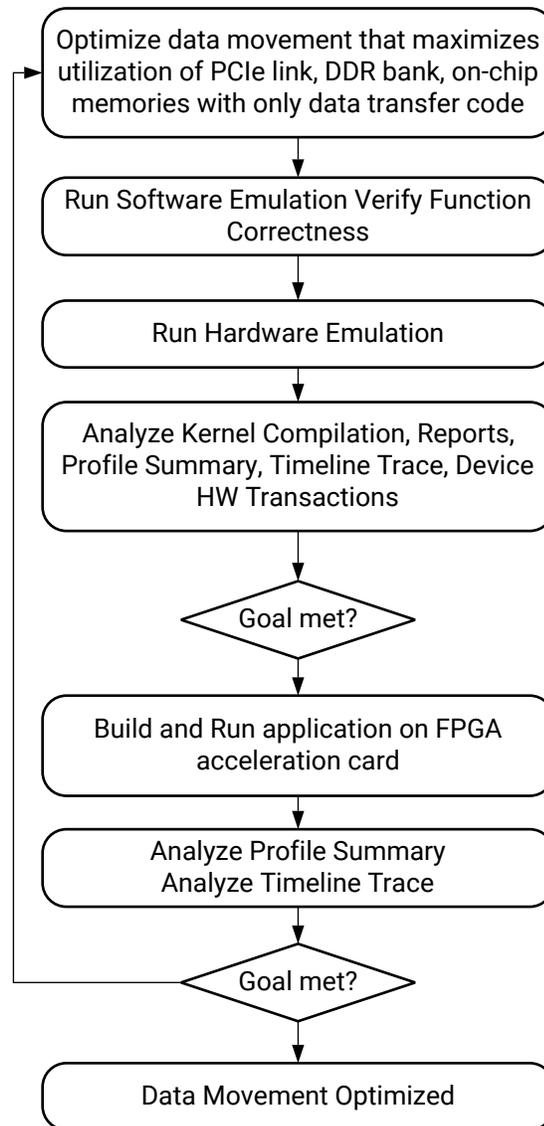


RECOMMENDED: For the task parallel case, Xilinx recommends minimizing the calls to `clEnqueueTask`. Ideally, you should finish all the workload in a single call to `clEnqueueTask`.

For more information on reducing overhead on kernel execution, see [Kernel Execution](#).

Optimizing Data Movement

Figure 48: Optimizing Data Movement Flow



X22239-082719

In the OpenCL API, all data is transferred from the host memory to the global memory on the device first and then from the global memory to the kernel for computation. The computation results are written back from the kernel to the global memory and lastly from the global memory to the host memory. A key factor in determining strategies for kernel optimization is understanding how data can be efficiently moved around.



RECOMMENDED: Optimize the data movement in the application before optimizing computation.

During data movement optimization, it is important to isolate data transfer code from computation code because inefficiency in computation might cause stalls in data movement. Xilinx recommends that you modify the host code and kernels with data transfer code only for this optimization step. The goal is to maximize the system level data throughput by maximizing PCIe bandwidth usage and DDR bandwidth usage. It usually takes multiple iterations of running software emulation, hardware emulation, as well as execution on FPGAs to achieve this goal.

Overlapping Data Transfers with Kernel Computation

Applications, such as database analytics, have a much larger data set than can be stored in the available memory on the acceleration device. They require the complete data to be transferred and processed in blocks. Techniques that overlap the data transfers with the computation are critical to achieve high performance for these applications.

Below is the `vadd` kernel from the [overlap_c](#) example in the [host](#) category of [Vitis Examples: Getting Started](#) on GitHub.

```
#define BUFFER_SIZE 256
#define DATA_SIZE 1024

//TRIPCOUNT identifier
const unsigned int c_len = DATA_SIZE / BUFFER_SIZE;
const unsigned int c_size = BUFFER_SIZE;

extern "C" {
void vadd(int *c, int *a, int *b, const int elements) {
#pragma HLS INTERFACE m_axi port = c offset = slave bundle = gmem
#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 s_axilite port = c bundle = control
#pragma HLS INTERFACE s_axilite port = a bundle = control
#pragma HLS INTERFACE s_axilite port = b bundle = control
#pragma HLS INTERFACE s_axilite port = elements bundle = control
#pragma HLS INTERFACE s_axilite port = return bundle = control

    int arrayA[BUFFER_SIZE];
    int arrayB[BUFFER_SIZE];
    for (int i = 0; i < elements; i += BUFFER_SIZE) {
        #pragma HLS LOOP_TRIPCOUNT min=c_len max=c_len
        int size = BUFFER_SIZE;
        if (i + size > elements)
            size = elements - i;
    readA:
        for (int j = 0; j < size; j++) {
            #pragma HLS PIPELINE II=1
            #pragma HLS LOOP_TRIPCOUNT min=c_size max=c_size
            arrayA[j] = a[i + j];
        }

    readB:
        for (int j = 0; j < size; j++) {
            #pragma HLS PIPELINE II=1
            #pragma HLS LOOP_TRIPCOUNT min=c_size max=c_size
            arrayB[j] = b[i + j];
        }
    }
}
```

```

vadd_writeC:
  for (int j = 0; j < size; j++) {
    #pragma HLS PIPELINE II=1
    #pragma HLS LOOP_TRIPCOUNT min=c_size max=c_size
    c[i + j] = arrayA[j] + arrayB[j];
  }
}
}

```

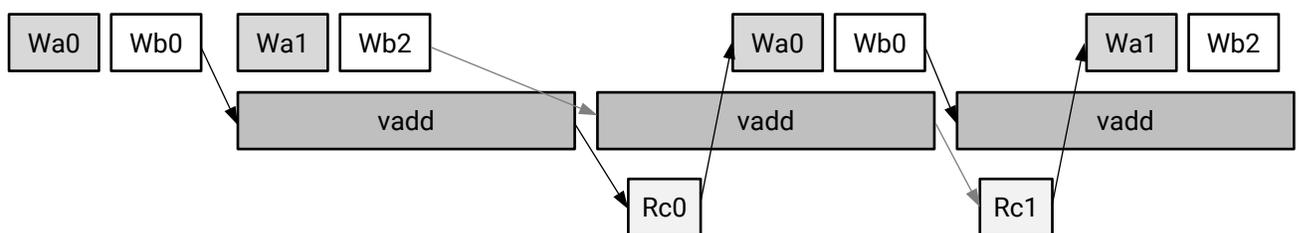
From the host perspective, there are four tasks to perform in this example:

1. Write buffer a (Wa)
2. Write buffer b (Wb)
3. Execute vadd kernel
4. Read buffer c (Rc)

Using an out-of-order command queue, data transfer and kernel execution can overlap as illustrated in the figure below. In the host code for this example, double buffering is used for all buffers so that the kernel can process one set of buffers while the host can operate on the other set of buffers.

The OpenCL `event` object provides an easy method to set up complex operation dependencies and synchronize host threads and device operations. Events are OpenCL objects that track the status of operations. Event objects are created by kernel execution commands, read, write, copy commands on memory objects or user events created using `clCreateUserEvent`. You can ensure an operation has completed by querying events returned by these commands. The arrows in the figure below show how event triggering can be set up to achieve optimal performance.

Figure 49: Event Triggering Set Up



X22780-042519

The host code enqueues the four tasks in a loop to process the complete data set. It also sets up event synchronization between different tasks to ensure that data dependencies are met for each task. The double buffering is set up by passing different memory objects values to `clEnqueueMigrateMemObjects` API. The event synchronization is achieved by having each API call wait for other event as well as trigger its own event when the API completes.

```
// THIS PAIR OF EVENTS WILL BE USED TO TRACK WHEN A KERNEL IS FINISHED WITH
// THE INPUT BUFFERS. ONCE THE KERNEL IS FINISHED PROCESSING THE DATA, A NEW
// SET OF ELEMENTS WILL BE WRITTEN INTO THE BUFFER.
vector<cl::Event> kernel_events(2);
vector<cl::Event> read_events(2);
cl::Buffer buffer_a[2], buffer_b[2], buffer_c[2];

for (size_t iteration_idx = 0; iteration_idx < num_iterations; iteration_idx
++) {
    int flag = iteration_idx % 2;

    if (iteration_idx >= 2) {
        OCL_CHECK(err, err = read_events[flag].wait());
    }

    // Allocate Buffer in Global Memory
    // Buffers are allocated using CL_MEM_USE_HOST_PTR for efficient memory
and
    // Device-to-host communication
    std::cout << "Creating Buffers..." << std::endl;
    OCL_CHECK(err,
        buffer_a[flag] =
            cl::Buffer(context,
                CL_MEM_READ_ONLY | CL_MEM_USE_HOST_PTR,
                bytes_per_iteration,
                &A[iteration_idx * elements_per_iteration],
                &err));
    OCL_CHECK(err,
        buffer_b[flag] =
            cl::Buffer(context,
                CL_MEM_READ_ONLY | CL_MEM_USE_HOST_PTR,
                bytes_per_iteration,
                &B[iteration_idx * elements_per_iteration],
                &err));
    OCL_CHECK(err,
        buffer_c[flag] = cl::Buffer(
            context,
            CL_MEM_WRITE_ONLY | CL_MEM_USE_HOST_PTR,
            bytes_per_iteration,
            &device_result[iteration_idx * elements_per_iteration],
            &err));

    vector<cl::Event> write_event(1);

    OCL_CHECK(err, err = krnl_vadd.setArg(0, buffer_c[flag]));
    OCL_CHECK(err, err = krnl_vadd.setArg(1, buffer_a[flag]));
    OCL_CHECK(err, err = krnl_vadd.setArg(2, buffer_b[flag]));
    OCL_CHECK(err, err = krnl_vadd.setArg(3, int(elements_per_iteration)));

    // Copy input data to device global memory
    std::cout << "Copying data (Host to Device)..." << std::endl;
    // Because we are passing the write_event, it returns an event object
    // that identifies this particular command and can be used to query
    // or queue a wait for this particular command to complete.
    OCL_CHECK(
```

```

err,
err = q.enqueueMigrateMemObjects({buffer_a[flag], buffer_b[flag]},
                                0 /*0 means from host*/,
                                NULL,
                                &write_event[0]);
set_callback(write_event[0], "ooo_queue");

printf("Enqueueing NDRange kernel.\n");
// This event needs to wait for the write buffer operations to complete
// before executing. We are sending the write_events into its wait list
to
// ensure that the order of operations is correct.
//Launch the Kernel
std::vector<cl::Event> waitList;
waitList.push_back(write_event[0]);
OCL_CHECK(err,
err = q.enqueueNDRangeKernel(
krnl_vadd, 0, 1, 1, &waitList, &kernel_events[flag]));
set_callback(kernel_events[flag], "ooo_queue");

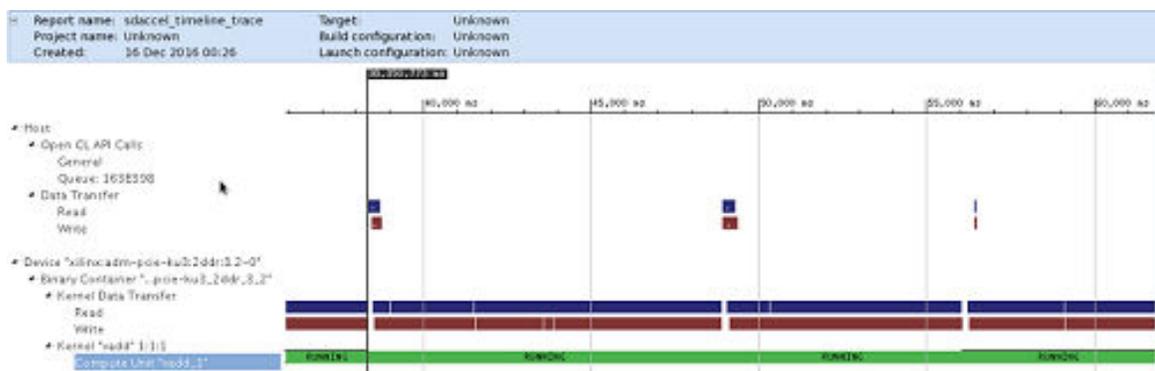
// Copy Result from Device Global Memory to Host Local Memory
std::cout << "Getting Results (Device to Host)..." << std::endl;
std::vector<cl::Event> eventList;
eventList.push_back(kernel_events[flag]);
// This operation only needs to wait for the kernel call. This call will
// potentially overlap the next kernel call as well as the next read
// operations
OCL_CHECK(err,
err = q.enqueueMigrateMemObjects({buffer_c[flag]},
                                CL_MIGRATE_MEM_OBJECT_HOST,
                                &eventList,
                                &read_events[flag]));
set_callback(read_events[flag], "ooo_queue");

OCL_CHECK(err, err = read_events[flag].wait());
}

```

The Application Timeline view below clearly shows that the data transfer time is completely hidden, while the compute unit `vadd_1` is running constantly.

Figure 50: Data Transfer Time Hidden in Application Timeline View



Buffer Memory Segmentation

Allocation and deallocation of memory buffers can lead to memory segmentation in the DDR controllers. This might result in sub-optimal performance of compute units, even if they could theoretically execute in parallel.

This issue occurs most often when multiple pthreads for different compute units are used and the threads allocate and release many device buffers with different sizes every time they enqueue the kernels. In this case, the timeline trace will exhibit gaps between kernel executions and it might seem the processes are sleeping.

Each buffer allocated by runtime should be continuous in hardware. For large memory, it might take some time to wait for that space to be freed, when many buffers are allocated and deallocated. This can be resolved by allocating device buffer and reusing it between different enqueues of a kernel.

For more details on optimizing memory performance, see [Reading and Writing by Burst](#).

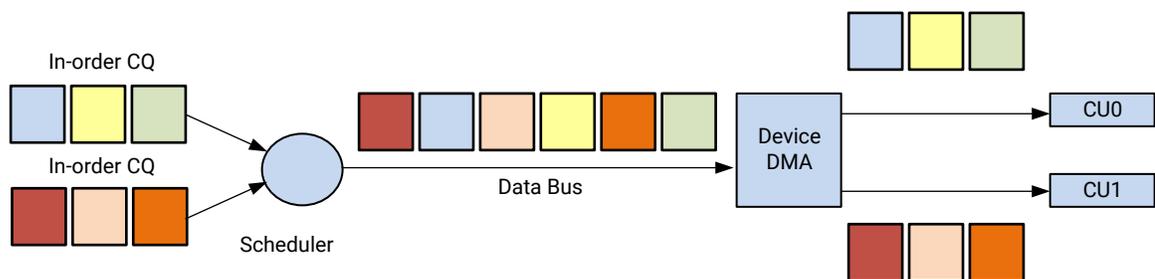
Compute Unit Scheduling

Scheduling kernel operations is key to overall system performance. This becomes even more important when implementing multiple compute units (of the same kernel or of different kernels). This section examines the different command queues responsible for scheduling the kernels.

Multiple In-Order Command Queues

The following figure shows an example with two in-order command queues, CQ0 and CQ1. The scheduler dispatches commands from each queue in order, but commands from CQ0 and CQ1 can be pulled out by the scheduler in any order. You must manage synchronization between CQ0 and CQ1 if required.

Figure 51: Example with Two In-Order Command Queues



X22781-091219

The following is code extracted from `host.cpp` of the [concurrent_kernel_execution_c](#) example that sets up multiple in-order command queues and enqueues commands into each queue:

```

OCL_CHECK(err,
    cl::CommandQueue ordered_queue1(
        context, device, CL_QUEUE_PROFILING_ENABLE, &err));
OCL_CHECK(err,
    cl::CommandQueue ordered_queue2(
        context, device, CL_QUEUE_PROFILING_ENABLE, &err));
...

printf("[Ordered Queue 1]: Enqueueing scale kernel\n");
OCL_CHECK(
    err,
    err = ordered_queue1.enqueueNDRangeKernel(
        kernel_mscale, offset, global, local, nullptr,
&kernel_events[0]));

set_callback(kernel_events[0], "scale");
...
printf("[Ordered Queue 1]: Enqueueing addition kernel\n");
OCL_CHECK(
    err,
    err = ordered_queue1.enqueueNDRangeKernel(
        kernel_madd, offset, global, local, nullptr,
&kernel_events[1]));

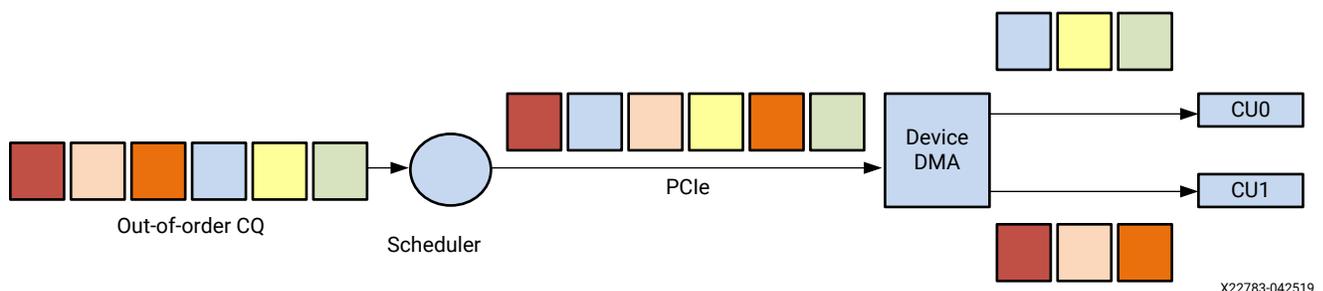
set_callback(kernel_events[1], "addition");
...
printf("[Ordered Queue 2]: Enqueueing matrix multiplication kernel\n");
OCL_CHECK(
    err,
    err = ordered_queue2.enqueueNDRangeKernel(
        kernel_mmult, offset, global, local, nullptr,
&kernel_events[2]));
set_callback(kernel_events[2], "matrix multiplication");

```

Single Out-of-Order Command Queue

The following figure shows an example with a single out-of-order command queue. The scheduler can dispatch commands from the queue in any order. You must manually define event dependencies and synchronizations as required.

Figure 52: Example with Single Out-of-Order Command Queue



X22783-042519

The following is code extracted from `host.cpp` of the `concurrent_kernel_execution_c` example that sets up a single out-of-order command queue and enqueues commands as needed:

```

    OCL_CHECK(
        err,
        cl::CommandQueue ooo_queue(context,
                                    device,
                                    CL_QUEUE_PROFILING_ENABLE |

CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE,
                                    &err));

...
    printf("[OOO Queue]: Enqueueing scale kernel\n");
    OCL_CHECK(
        err,
        err = ooo_queue.enqueueNDRangeKernel(
            kernel_mscale, offset, global, local, nullptr, &ooo_events[0]));
    set_callback(ooo_events[0], "scale");
...
    // This is an out of order queue, events can be executed in any order.
    Since
    // this call depends on the results of the previous call we must pass
    the
    // event object from the previous call to this kernel's event wait list.
    printf("[OOO Queue]: Enqueueing addition kernel (Depends on scale)\n");

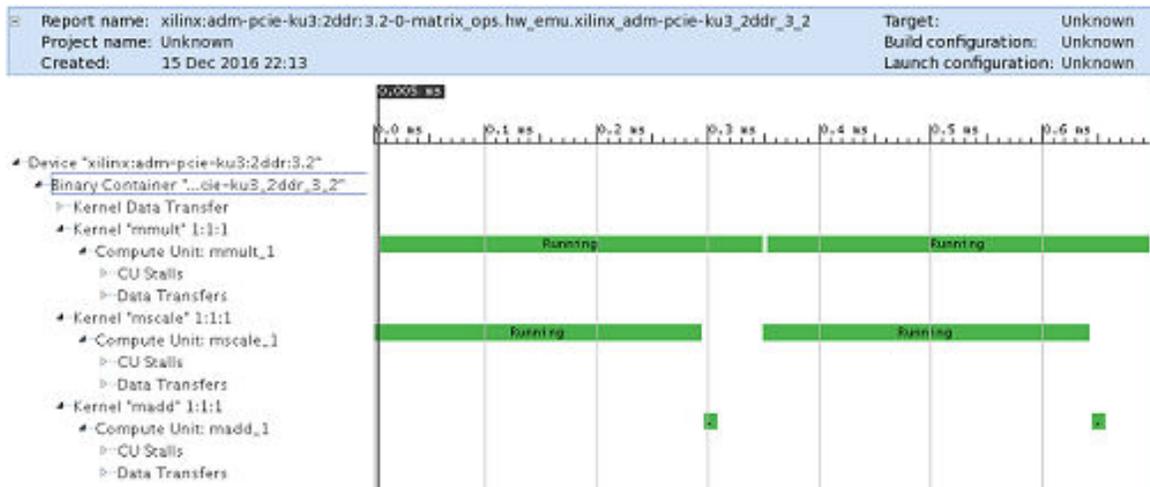
    kernel_wait_events.resize(0);
    kernel_wait_events.push_back(ooo_events[0]);

    OCL_CHECK(err,
        err = ooo_queue.enqueueNDRangeKernel(
            kernel_madd,
            offset,
            global,
            local,
            &kernel_wait_events, // Event from previous call
            &ooo_events[1]));
    set_callback(ooo_events[1], "addition");
...
    // This call does not depend on previous calls so we are passing nullptr
    // into the event wait list. The runtime should schedule this kernel in
    // parallel to the previous calls.
    printf("[OOO Queue]: Enqueueing matrix multiplication kernel\n");
    OCL_CHECK(err,
        err = ooo_queue.enqueueNDRangeKernel(
            kernel_mmult,
            offset,
            global,
            local,
            nullptr, // Does not depend on previous call
            &ooo_events[2]));
    set_callback(ooo_events[2], "matrix multiplication");

```

The Application Timeline view (as shown in the following figure) that the compute unit `mmult_1` is running in parallel with the compute units `mscale_1` and `madd_1`, using both multiple in-order queues and single out-of-order queue methods.

Figure 53: Application Timeline View Showing mult_1 Running with mscale_1 and madd_1

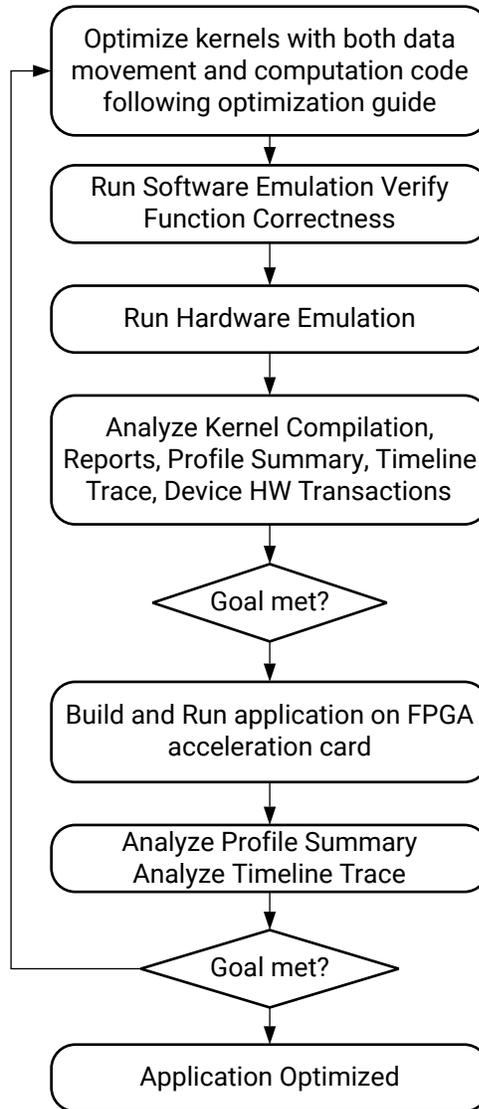


Kernel Optimization

One of the key advantages of an FPGA is its flexibility and capacity to create customized designs specifically for your algorithm. This enables various implementation choices to trade off algorithm throughput versus power consumption. The following guidelines help manage the design complexity and achieve the desired design goals.

Optimizing Kernel Computation

Figure 54: Optimizing Kernel Computation Flow



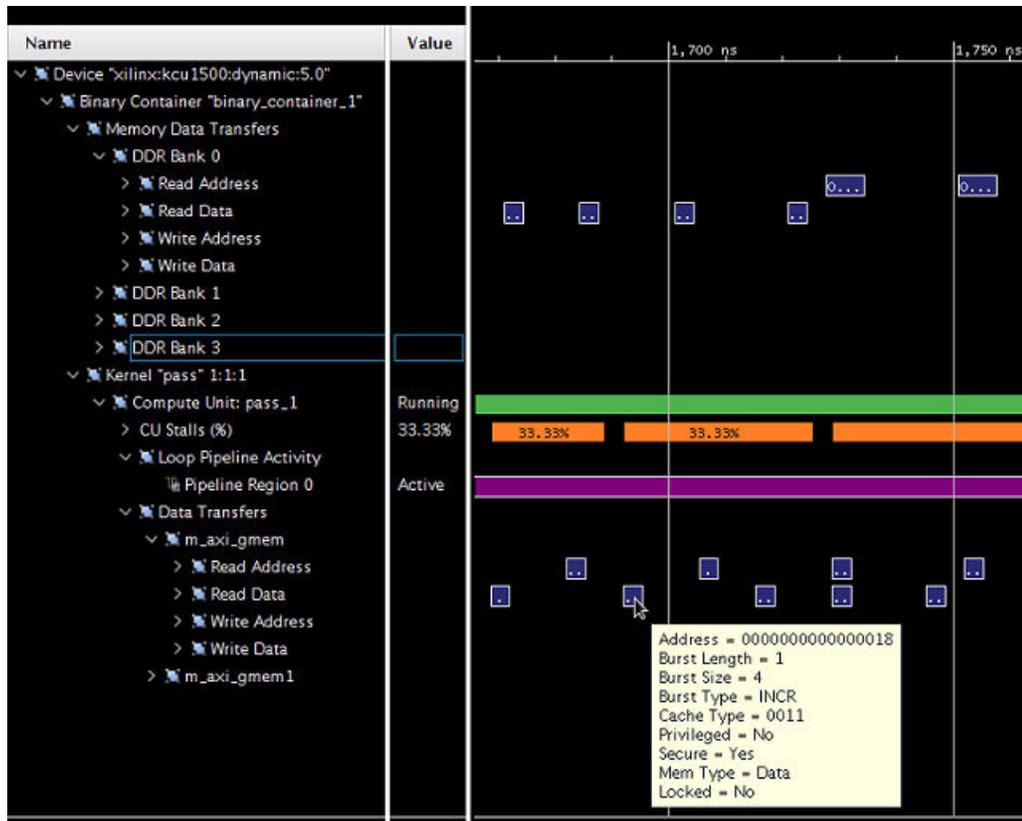
X22240-082719

The goal of kernel optimization is to create processing logic that can consume all the data as soon as it arrives at the kernel interfaces. The key metric is the initiation interval (II), or the number of clock cycles before the kernel can accept new input data. Optimizing the II is generally achieved by expanding the processing code to match the data path with techniques such as function pipelining, loop unrolling, array partitioning, data flowing, etc. For more information on kernel optimization, see [Linking the Kernels](#).

Interface Attributes (Detailed Kernel Trace)

The detailed kernel trace provides easy access to the AXI transactions and their properties. The AXI transactions are presented for the global memory, as well as the Kernel side (Kernel "pass" 1:1:1) of the AXI interconnect. The following figure illustrates a typical kernel trace of a newly accelerated algorithm.

Figure 55: Accelerated Algorithm Kernel Trace



Most interesting with respect to performance are the fields:

- **Burst Length:** Describes how many packages are sent within one transaction.
- **Burst Size:** Describes the number of bytes being transferred as part of one package.

Given a burst length of 1 and just 4 bytes per package, it will require many individual AXI transactions to transfer any reasonable amount of data.

Note: The Vitis core development kit never creates burst sizes less than 4 bytes, even if smaller data is transmitted. In this case, if consecutive items are accessed without AXI bursts enabled, it is possible to observe multiple AXI reads to the same address.

Small burst lengths, as well as burst sizes, considerably less than 512 bits are therefore good opportunities to optimize interface performance.

Using Burst Data Transfers

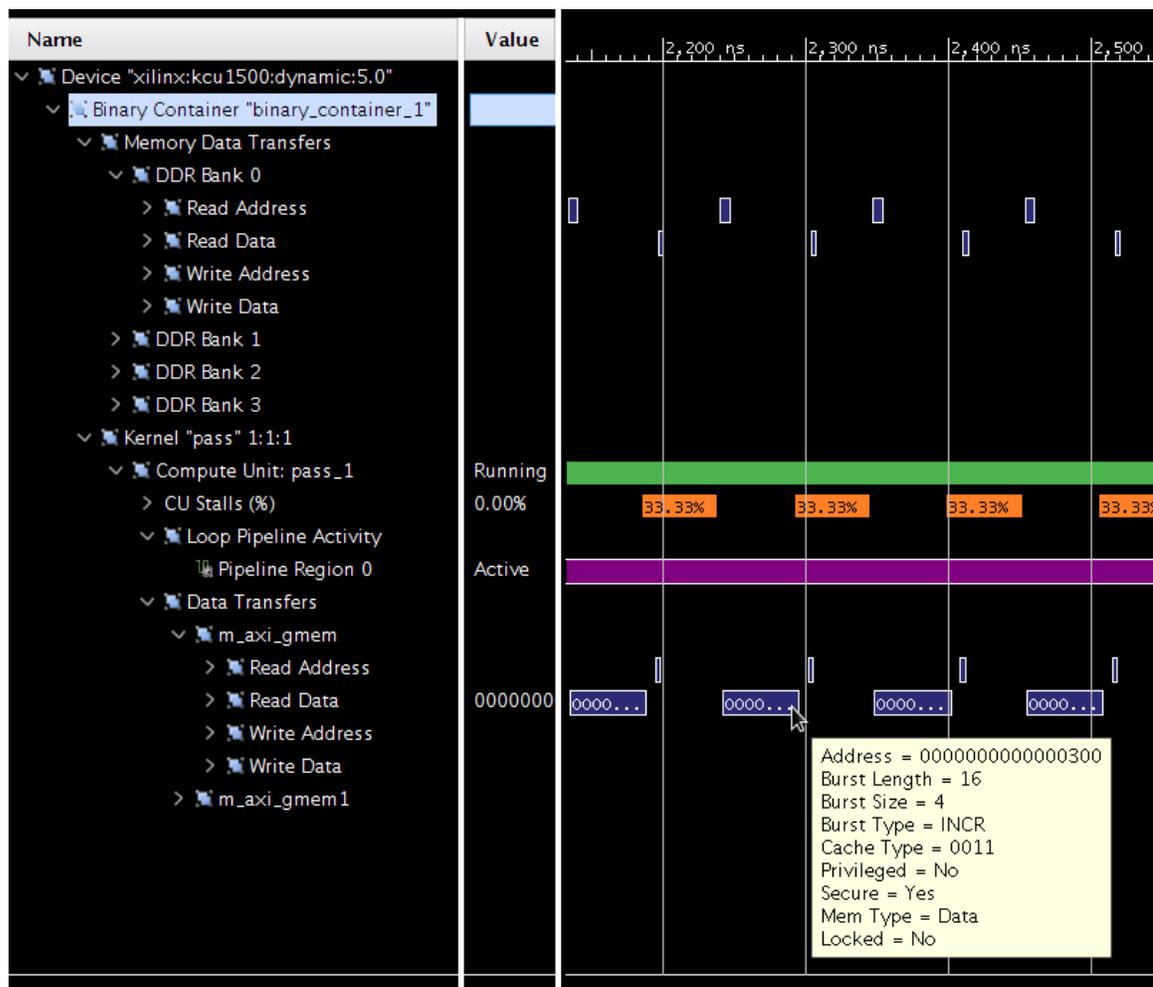
Transferring data in bursts hides the memory access latency and improves bandwidth usage and efficiency of the memory controller.



RECOMMENDED: *Infer burst transfers from successive requests of data from consecutive address locations. Refer to [Reading and Writing by Burst](#) for more details.*

If burst data transfers occur, the detailed kernel trace will reflect the higher burst rate as a larger burst length number:

Figure 56: Burst Data Transfer with Detailed Kernel Trace



In the previous figure, it is also possible to observe that the memory data transfers following the AXI interconnect are actually implemented rather differently (shorter transaction time). Hover over these transactions, you would see that the AXI interconnect has packed the 16x4 byte transaction into a single package transaction of 1x64 bytes. This effectively uses the AXI4 bandwidth which is even more favorable. The next section focuses on this optimization technique in more detail.

Burst inference is heavily dependent on coding style and access pattern. However, you can ease burst detection and improve performance by isolating data transfer and computation, as shown in the following code snippet:

```
void kernel(T in[1024], T out[1024]) {
    T tmpIn[1024];
    T tmpOu[1024];
    read(in, tmpIn);
    process(tmpIn, tmpOut);
    write(tmpOut, out);
}
```

In short, the function `read` is responsible for reading from the AXI input to an internal variable (`tmpIn`). The computation is implemented by the function `process` working on the internal variables `tmpIn` and `tmpOut`. The function `write` takes the produced output and writes to the AXI output.

The isolation of the read and write function from the computation results in:

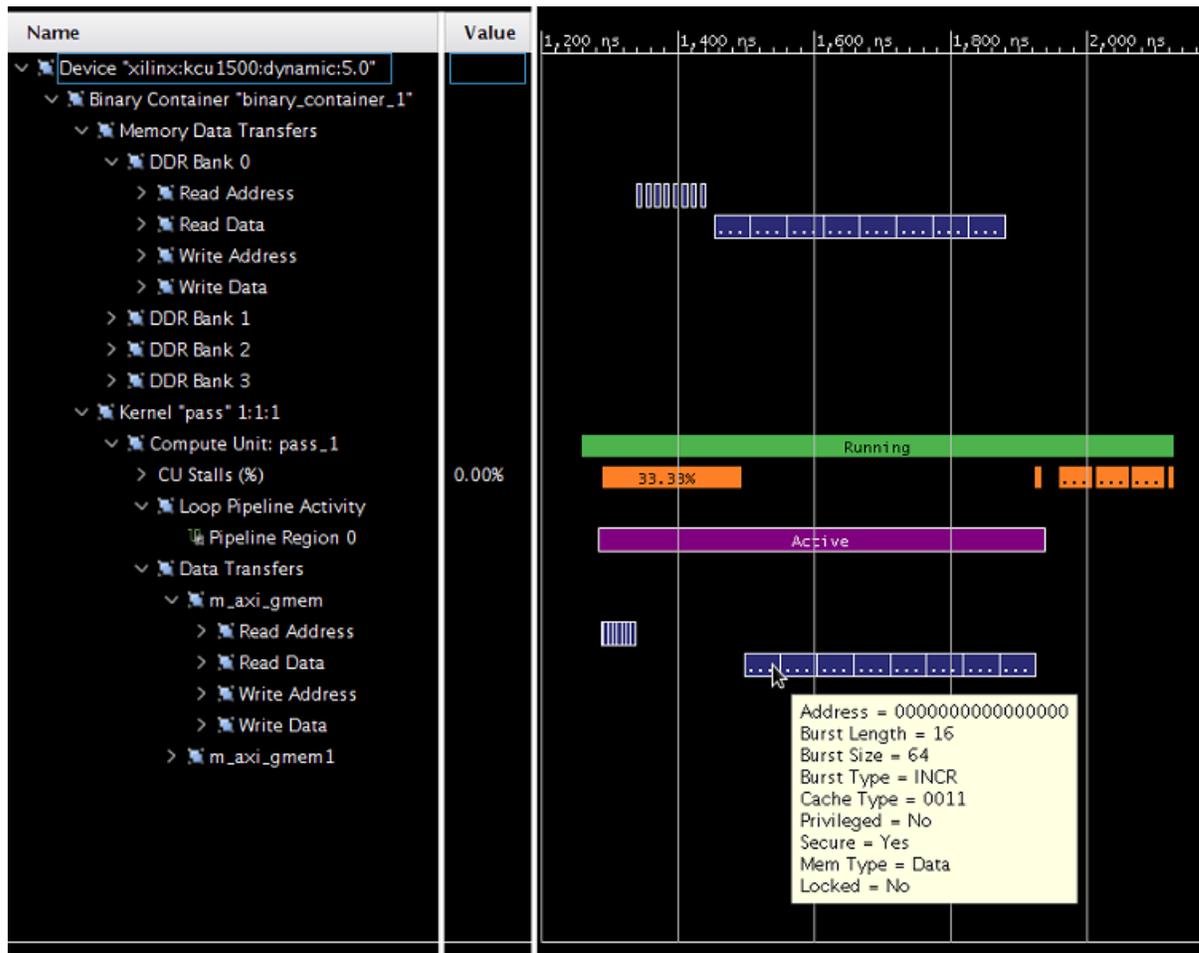
- Simple control structures (loops) in the read/write function which makes burst detection simpler.
- The isolation of the computational function away from the AXI interfaces, simplifies potential kernel optimization. See [Kernel Optimization](#) for more information.
- The internal variables are mapped to on-chip memory, which allow faster access compared to AXI transactions. Acceleration platforms supported in the Vitis core development kit can have as much as 10 MB on-chip memories that can be used as pipes, local memories, and private memories. Using these resources effectively can greatly improve the efficiency and performance of your applications.

Using Full AXI Data Width

The user data width between the kernel and the memory controller can be configured by the Vitis compiler based on the data types of the kernel arguments. To maximize the data throughput, Xilinx recommends that you choose data types map to the full data width on the memory controller. The memory controller in all supported acceleration cards supports 512-bit user interface, which can be mapped to OpenCL vector data types, such as `int16` or C/C++ arbitrary precision data type `ap_int<512>`.

As shown on the following figure, you can observe burst AXI transactions (Burst Length 16) and a 512-bit package size (Burst Size 64 bytes).

Figure 57: Burst AXI Transactions



This example shows good interface configuration as it maximizes AXI data width as well as actual burst transactions.

Complex structs or classes, used to declare interfaces, can lead to very complex hardware interfaces due to memory layout and data packing differences. This can introduce potential issues that are very difficult to debug in a complex system.



RECOMMENDED: Use simple structs for kernel arguments that can be packed to 32-bit boundary. Refer to the Custom Data Type Example in kernel_to_gmem category at [Xilinx Getting Started Example](#) on GitHub for the recommended method to use structs.

Setting Data Width Using OpenCL Attributes

The OpenCL API provides attributes to support a more automatic approach to incrementing AXI data width usage. The change of the interface data types, as stated above is supported in the API as well but will require the same code changes as C/C++ to the algorithm to accommodate the larger input vector.

To eliminate manual code modifications, the following OpenCL attributes are interpreted to perform data path widening and vectorization of the algorithm:

- `vec_type_hint`
- `reqd_work_group_size`
- `xcl_zero_global_work_offset`

Examine the combined functionality on the following case:

```
__attribute__((reqd_work_group_size(64, 1, 1)))
__attribute__((vec_type_hint(int)))
__attribute__((xcl_zero_global_work_offset))
__kernel void vector_add(__global int* c, __global const int* a, __global
const int* b) {
    size_t idx = get_global_id(0);
    c[idx] = a[idx] + b[idx];
}
```

In this case, the hard coded interface is a 32-bit wide data path (`int *c`, `int* a`, `int *b`), which drastically limits the memory throughput if implemented directly. However, the automatic widening and transformation is applied, based on the values of the three attributes.

- **`__attribute__((vec_type_hint(int)))`**: Declares that `int` is the main type used for computation and memory transfer (32-bit). This knowledge is used to calculate the vectorization/widening factor based on the target bandwidth of the AXI interface (512 bits). In this example the factor would be $16 = 512 \text{ bits} / 32\text{-bit}$. This implies that in theory, 16 values could be processed if vectorization can be applied.
- **`__attribute__((reqd_work_group_size(X, Y, Z)))`**: Defines the total number of work items (where `X`, `Y`, and `Z` are positive constants). $X * Y * Z$ is the maximum number of work items therefore defining the maximum possible vectorization factor which would saturate the memory bandwidth. In this example, the total number of work items is $64 * 1 * 1 = 64$.

The actual vectorization factor to be applied will be the greatest common divider of the vectorization factor defined by the actual coded type or the `vec_type_hint`, and the maximum possible vectorization factor defined through `reqd_work_group_size`.

The quotient of maximum possible vectorization factor divided by the actual vectorization factor provides the remaining loop count of the OpenCL description. As this loop is pipelined, it can be advantageous to have several remaining loop iterations to take advantage of a pipelined implementation. This is especially true if the vectorized OpenCL code has long latency.

- **`__attribute__((xcl_zero_global_work_offset))`**: The `__attribute__((xcl_zero_global_work_offset))` instructs the compiler that no global offset parameter is used at runtime, and all accesses are aligned. This gives the compiler valuable information with regard to alignment of the work groups, which in turn usually propagates to the alignment of the memory accesses (less hardware).

It should be noted, that the application of these transformations changes the actual design to be synthesized. Partially unrolled loops require reshaping of local arrays in which data is stored. This usually behaves nicely, but can interact poorly in rare situations.

For example:

- For partitioned arrays, when the partition factor is not divisible by the unrolling/vectorization factor.
 - The resulting access requires a lot of multiplexers and will create a difficult issue for the scheduler (might severely increase memory usage and compilation time). Xilinx recommends using partitioning factors that are powers of two (as the vectorization factor is always a power of two).
- If the loop being vectorized has an unrelated resource constraint, the scheduler complains about II not being met.
 - This is not necessarily correlated with a loss of performance (usually it is still performing better) because the II is computed on the unrolled loop (which has therefore a multiplied throughput for each iteration).
 - The scheduler informs you of the possible resources constraints and resolving those will further improve the performance.
 - Note that a common occurrence is that a local array does not get automatically reshaped (usually because it is accessed in a later section of the code in non-vectorizable method).

Reducing Kernel to Kernel Communication Latency with OpenCL Pipes

The OpenCL API 2.0 specification introduces a new memory object called a pipe. A pipe stores data organized as a FIFO. Pipe objects can only be accessed using built-in functions that read from and write to a pipe. Pipe objects are not accessible from the host. Pipes can be used to stream data from one kernel to another inside the FPGA without having to use the external memory, which greatly improves the overall system latency. For more information, see [Pipe Functions](#) on Version 2.0 of the OpenCL C Specification from Khronos Group.

In the Vitis IDE, pipes must be statically defined outside of all kernel functions. Dynamic pipe allocation using the OpenCL 2.x `clCreatePipe` API is not supported. The depth of a pipe must be specified by using the OpenCL attribute `xcl_reqd_pipe_depth` in the pipe declaration. For more information, see [xcl_reqd_pipe_depth](#).

As specified in `xcl_reqd_pipe_depth`, the valid depth values are as follows: 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768.

A given pipe can have one and only one producer and consumer in different kernels.

```
pipe int p0 __attribute__((xcl_reqd_pipe_depth(32)));
```

Pipes can be accessed using standard OpenCL `read_pipe()` and `write_pipe()` built-in functions in non-blocking mode or using the Xilinx extended `read_pipe_block()` and `write_pipe_block()` functions in blocking mode.

The status of pipes can be queried using OpenCL `get_pipe_num_packets()` and `get_pipe_max_packets()` built-in functions.

The following function signatures are the currently supported pipe functions, where `gentype` indicates the built-in OpenCL C scalar integer or floating-point data types.

```
int read_pipe_block (pipe gentype p, gentype *ptr)
int write_pipe_block (pipe gentype p, const gentype *ptr)
```

The following “[dataflow/dataflow_pipes_ocl](#)” from [Xilinx Getting Started Examples](#) on GitHub uses pipes to pass data from one processing stage to the next using blocking `read_pipe_block()` and `write_pipe_block()` functions:

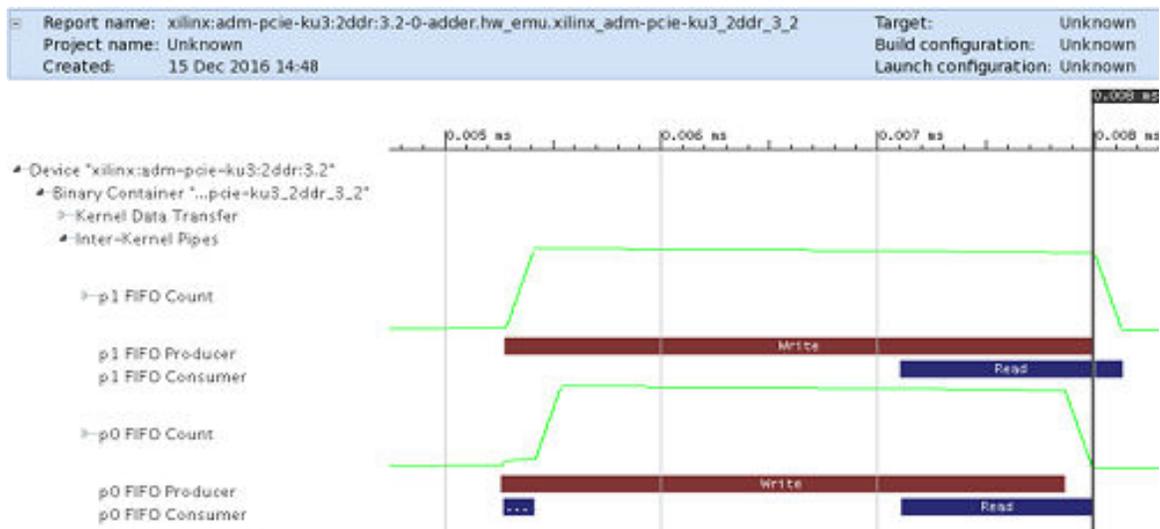
```
pipe int p0 __attribute__((xcl_reqd_pipe_depth(32)));
pipe int p1 __attribute__((xcl_reqd_pipe_depth(32)));
// Input Stage Kernel : Read Data from Global Memory and write into Pipe P0
kernel __attribute__((reqd_work_group_size(1, 1, 1)))
void input_stage(__global int *input, int size)
{
    __attribute__((xcl_pipeline_loop))
    mem_rd: for (int i = 0 ; i < size ; i++)
    {
        //blocking Write command to pipe P0
        write_pipe_block(p0, &input[i]);
    }
}
// Adder Stage Kernel: Read Input data from Pipe P0 and write the result
// into Pipe P1
kernel __attribute__((reqd_work_group_size(1, 1, 1)))
void adder_stage(int inc, int size)
{
    __attribute__((xcl_pipeline_loop))
    execute: for(int i = 0 ; i < size ; i++)
    {
        int input_data, output_data;
        //blocking read command to Pipe P0
        read_pipe_block(p0, &input_data);
        output_data = input_data + inc;
        //blocking write command to Pipe P1
        write_pipe_block(p1, &output_data);
    }
}
// Output Stage Kernel: Read result from Pipe P1 and write the result to
// Global
// Memory
kernel __attribute__((reqd_work_group_size(1, 1, 1)))
void output_stage(__global int *output, int size)
{
    __attribute__((xcl_pipeline_loop))
    mem_wr: for (int i = 0 ; i < size ; i++)
```

```

    {
        //blocking read command to Pipe P1
        read_pipe_block(p1, &output[i]);
    }
}
    
```

The Device Traceline view shows the detailed activities and stalls on the OpenCL pipes after hardware emulation is run. This information can be used to choose the correct FIFO sizes to achieve the optimal application area and performance.

Figure 58: Device Traceline View



Optimizing Computational Parallelism

By default, C/C++ does not model computational parallelism, as it always executes any algorithm sequentially. On the other hand, the OpenCL API does model computational parallelism with respect to work groups, but it does not use any additional parallelism within the algorithm description. However, fully configurable computational engines like FPGAs allow more freedom to exploit computational parallelism.

Coding Data Parallelism

To leverage computational parallelism during the implementation of an algorithm on the FPGA, it should be mentioned that the synthesis tool will need to be able to recognize computational parallelism from the source code first. Loops and functions are prime candidates for reflecting computational parallelism and compute units in the source description. However, even in this case, it is key to verify that the implementation takes advantage of the computational parallelism as in some cases the Vitis technology might not be able to apply the desired transformation due to the structure of the source code.

It is quite common, that some computational parallelism might not be reflected in the source code to begin with. In this case, it will need to be added. A typical example is a kernel that might be described to operate on a single input value, while the FPGA implementation might execute computations more efficiently in parallel on multiple values. This kind of parallel modeling is described in [Using Full AXI Data Width](#). A 512-bit interface can be created using OpenCL vector data types such as `int16` or C/C++ arbitrary precision data type `ap_int<512>`.

Note: These vector types can also be used as a powerful way to model data parallelism within a kernel, with up to 16 data paths operating in parallel in case of `int16`. Refer to the *Median Filter Example* in the *vision* category at [Xilinx Getting Started Example](#) on GitHub for the recommended method to use vectors.

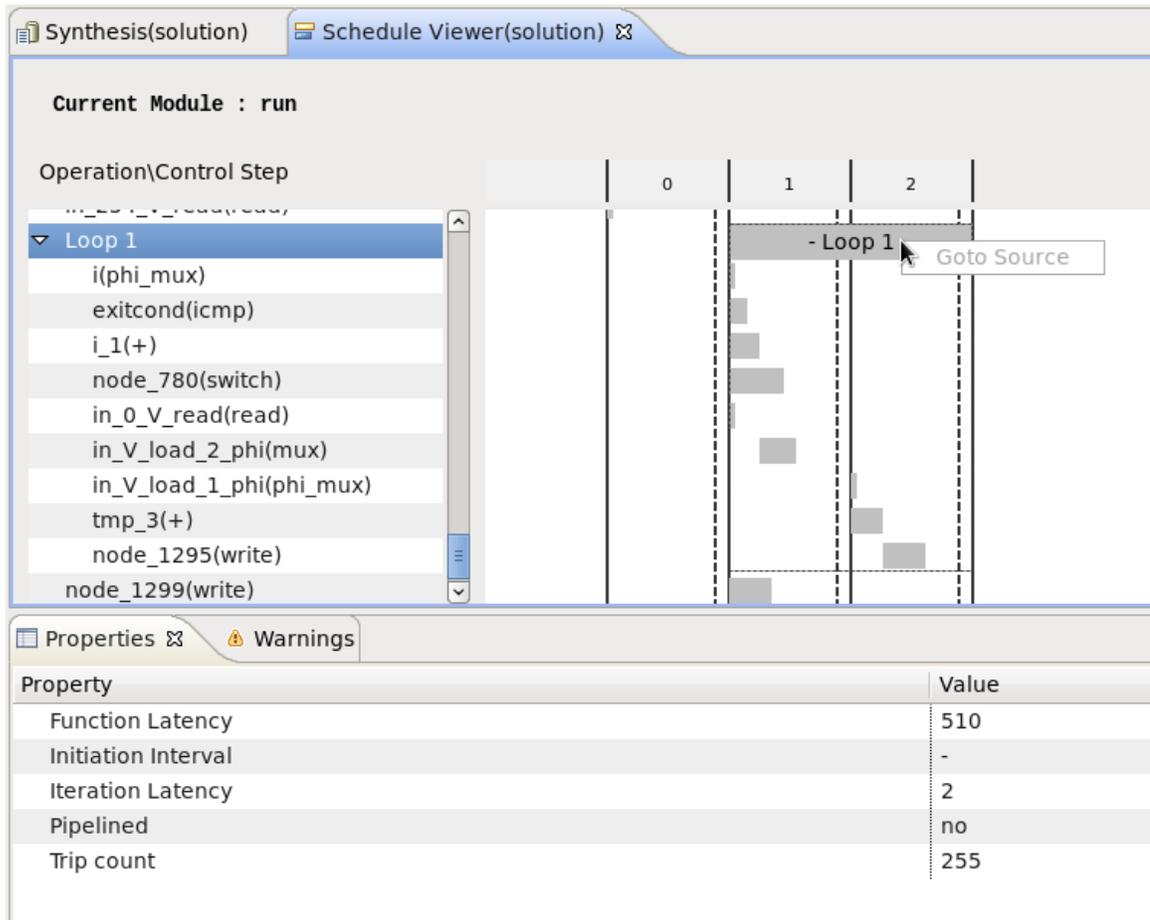
Loop Parallelism

Loops are the basic C/C++/OpenCL API method of representing repetitive algorithmic code. The following example illustrates various implementation aspects of a loop structure:

```
for(int i = 0; i<255; i++) {
    out[i] = in[i]+in[i+1];
}
out[255] = in[255];
```

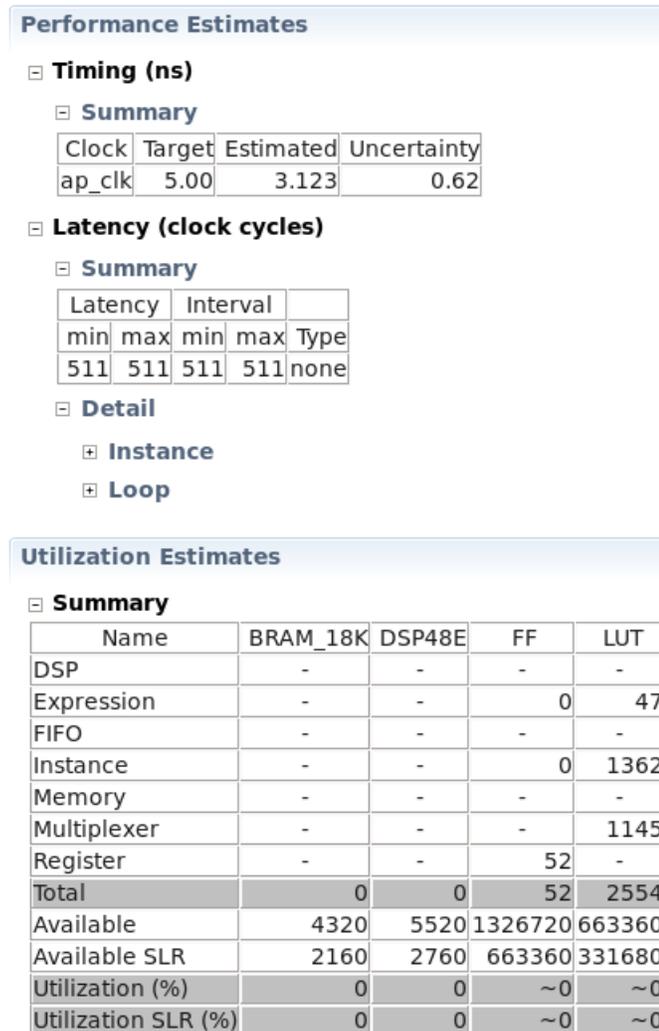
This code iterates over an array of values and adds consecutive values, except the last value. If this loop is implemented as written, each loop iteration requires two cycles for implementation, which results in a total of 510 cycles for implementation. This can be analyzed in detail through the Schedule Viewer in the HLS Project:

Figure 59: Implemented Loop Structure in Schedule Viewer



This can also be analyzed in terms of total numbers and latency through the Vivado synthesis results:

Figure 60: Synthesis Results Performance Estimates



The key numbers here are the latency numbers and total LUT usage. For example, depending on the configuration, you could get latency of 511 and total LUT usage of 47. As a result, these values can vary based on the implementation choices. While this implementation will require very little area, it results in significant latency.

Unrolling Loops

Unrolling a loop enables the full parallelism of the model to be used. To perform this, mark a loop to be unrolled and the tool will create the implementation with the most parallelism possible. To mark a loop to unroll, an OpenCL loop can be marked with the UNROLL attribute:

```
__attribute__((opencl_unroll_hint))
```

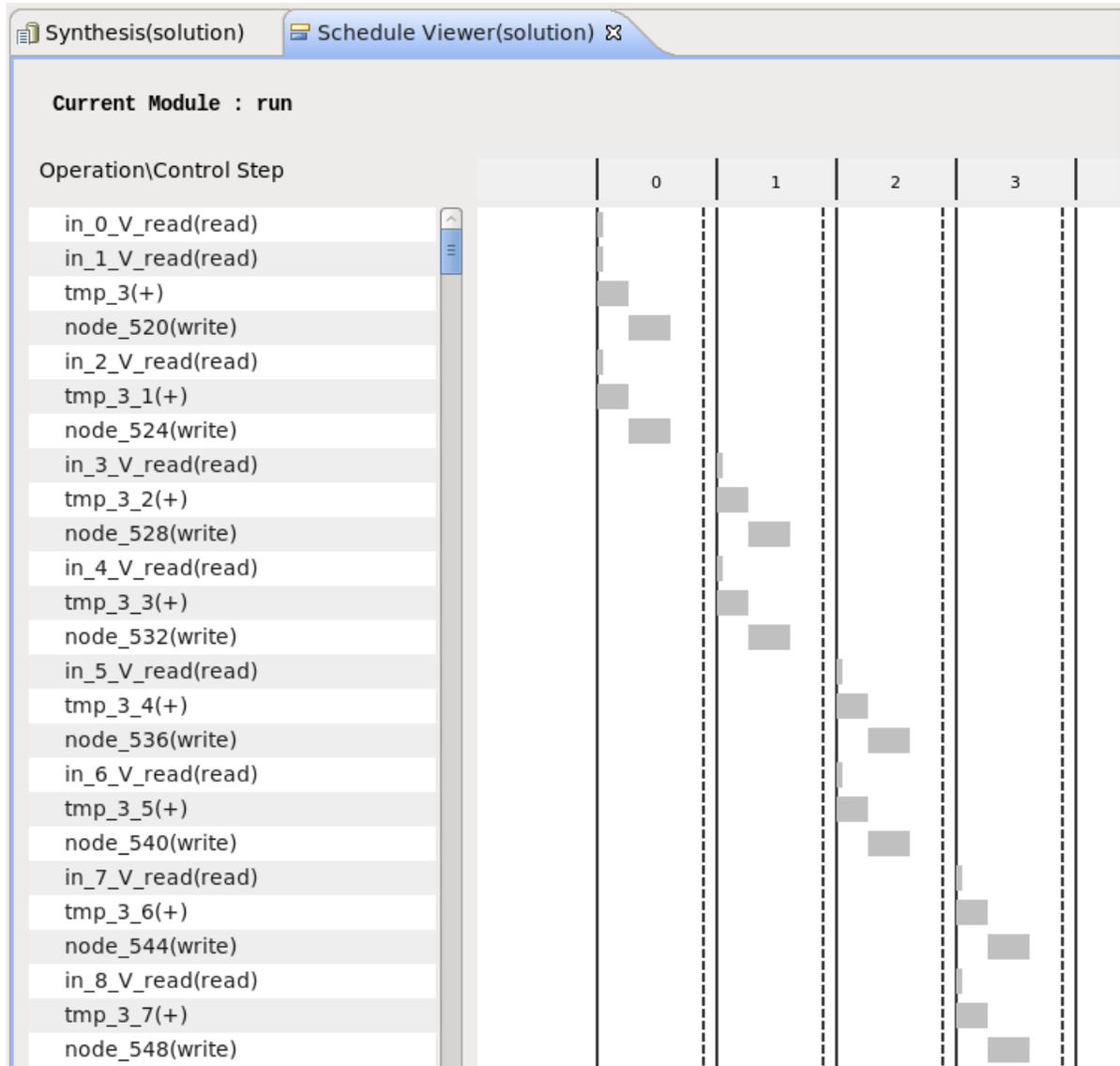
Or a C/C++ loop can use the unroll pragma:

```
#pragma HLS UNROLL
```

For more information, see [Loop Unrolling](#).

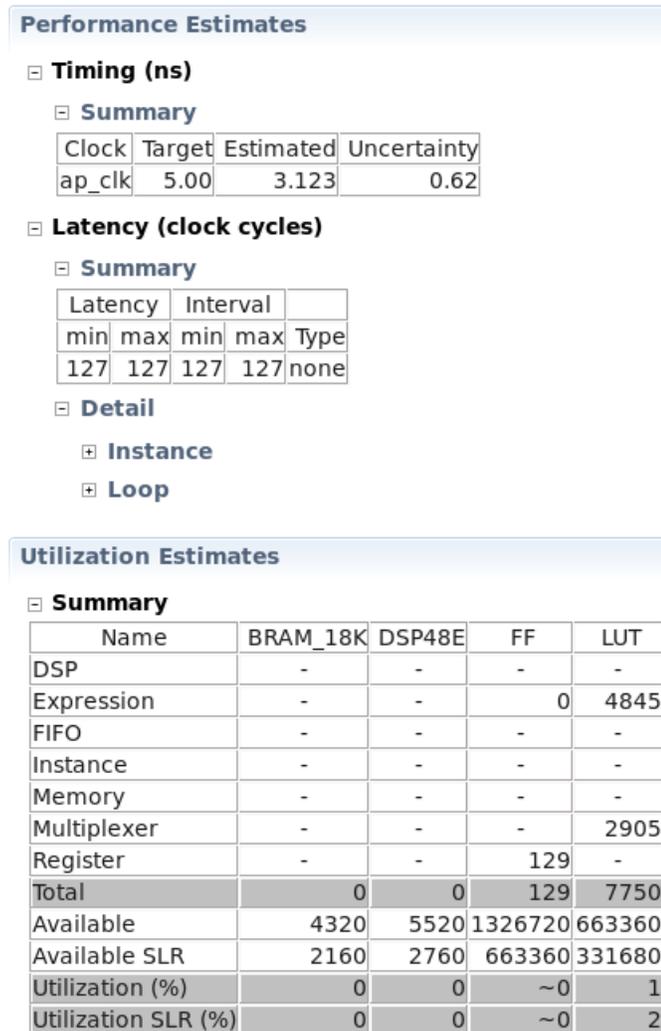
When applied to this specific example, the Schedule Viewer in the HLS Project will be:

Figure 61: Schedule Viewer



The following figure shows the estimated performance:

Figure 62: Performance Estimates



Therefore, the total latency was considerably improved to be 127 cycles and as expected the computational hardware was increased to 4845 LUTs, to perform the same computation in parallel.

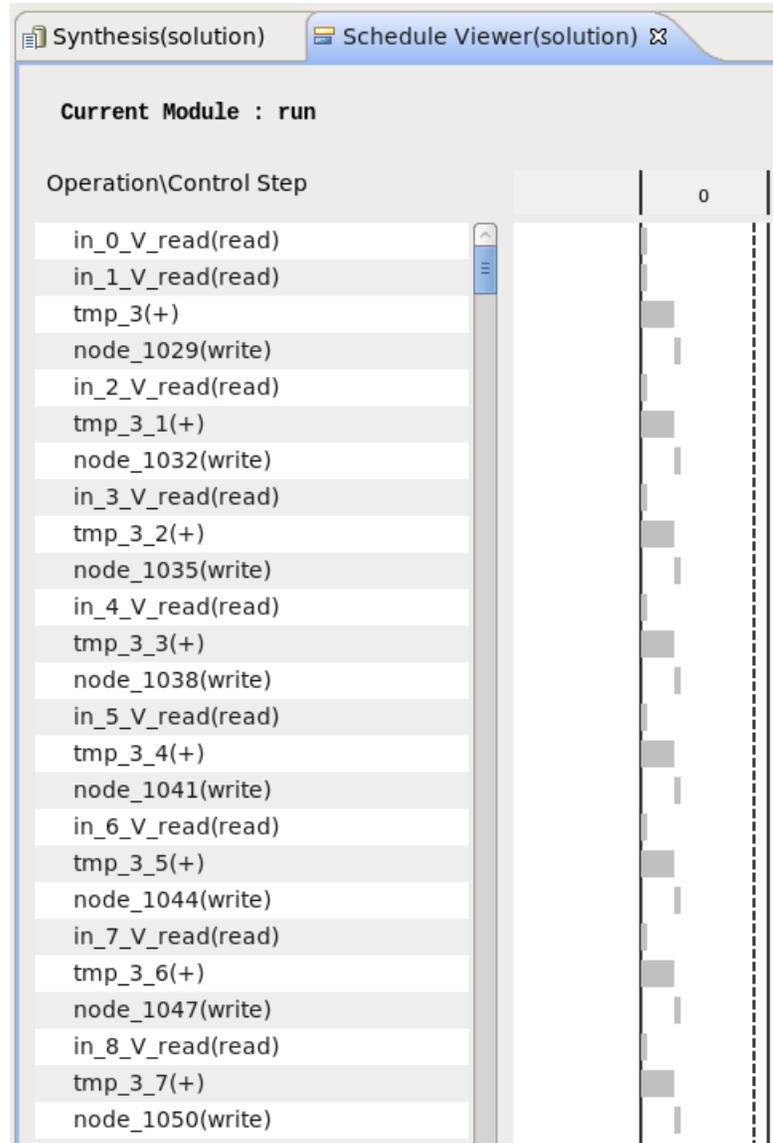
However, if you analyze the for-loop, you might ask why this algorithm cannot be implemented in a single cycle, as each addition is completely independent of the previous loop iteration. The reason is the memory interface is used for the variable `out`. The Vitis core development kit uses dual port memory by default for an array. However, this implies that at most two values can be written to the memory per cycle. Thus to see a fully parallel implementation, you must specify that the variable `out` should be kept in registers as in this example:

```
#pragma HLS array_partition variable= out complete dim= 0
```

For more information, see [pragma HLS array_partition](#).

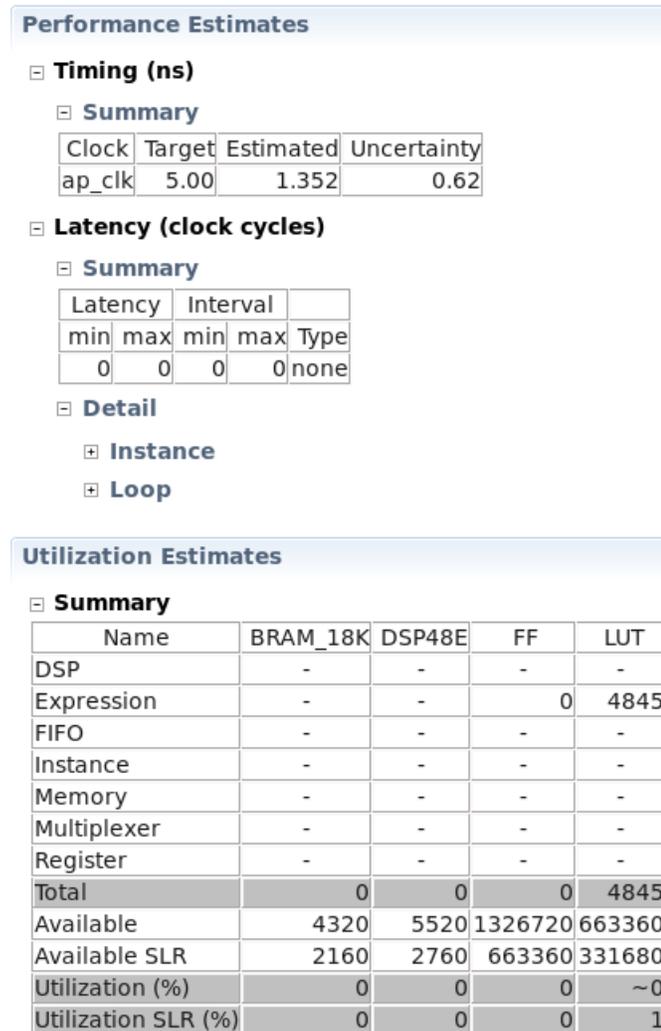
The results of this transformation can be observed in the following Schedule Viewer:

Figure 63: Transformation Results in Schedule Viewer



The associated estimates are:

Figure 64: Transformation Results Performance Estimates



Accordingly, this code can be implemented as a combinatorial function requiring only a fraction of the cycle to complete.

Pipelining Loops

Pipelining loops allow you to overlap iterations of a loop in time, as discussed in [Loop Pipelining](#). Allowing loop iterations to operate concurrently is often a good approach, as resources can be shared between iterations (less resource utilization), while requiring less execution time compared to loops that are not unrolled.

Pipelining is enabled in C/C++ through the [pragma HLS pipeline](#):

```
#pragma HLS PIPELINE
```

While the OpenCL API uses the `xcl_pipeline_loop` attribute:

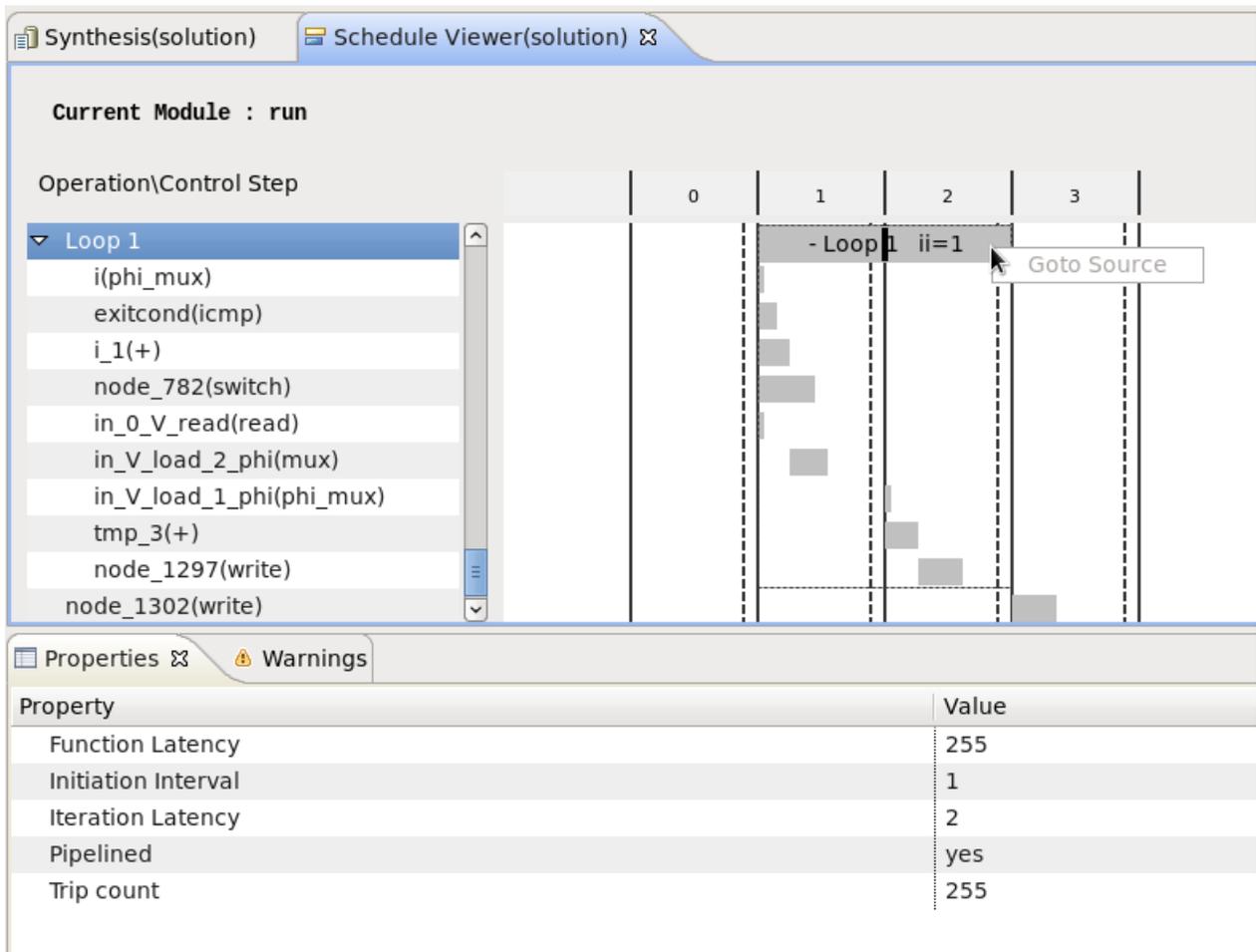
```
__attribute__((xcl_pipeline_loop))
```

Note: The OpenCL API has an additional method of specifying loop pipelining, see `xcl_pipeline_workitems`. The reason is the work item loops are not explicitly stated and pipelining these loops require this attribute:

```
__attribute__((xcl_pipeline_workitems))
```

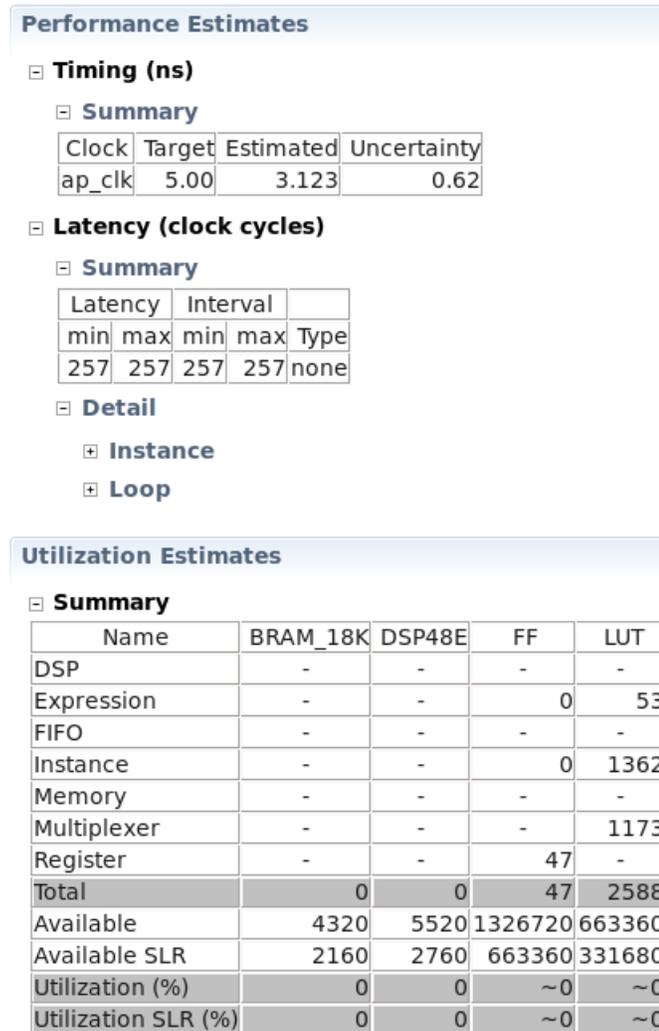
In this example, the Schedule Viewer in the HLS Project produces the following information:

Figure 65: Pipelining Loops in Schedule Viewer



With the overall estimates being:

Figure 66: Performance Estimates



Because each iteration of a loop consumes only two cycles of latency, there can only be a single iteration overlap. This enables the total latency to be cut into half compared to the original, resulting in 257 cycles of total latency. However, this reduction in latency was achieved using fewer resources when compared to unrolling.

In most cases, loop pipelining by itself can improve overall performance. Yet, the effectiveness of the pipelining depends on the structure of the loop. Some common limitations are:

- Resources with limited availability such as memory ports or process channels can limit the overlap of the iterations (Initiation Interval).
- Loop-carry dependencies, such as those created by variable conditions computed in one iteration affecting the next, might increase the II of the pipeline.

These are reported by the tool during high-level synthesis and can be observed and examined in the Schedule Viewer. For the best possible performance, the code might have to be modified to remove these limiting factors, or the tool needs to be instructed to eliminate some dependency by restructuring the memory implementation of an array, or breaking the dependencies all together.

Task Parallelism

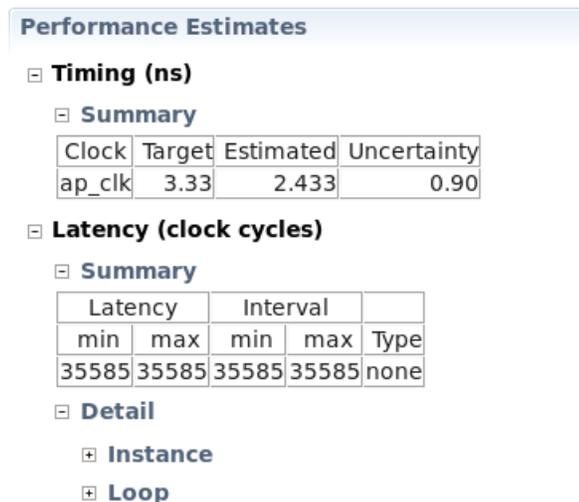
Task parallelism allows you to take advantage of dataflow parallelism. In contrast to loop parallelism, when task parallelism is deployed, full execution units (tasks) are allowed to operate in parallel taking advantage of extra buffering introduced between the tasks.

See the following example:

```
void run (ap_uint<16> in[1024],
         ap_uint<16> out[1024]
        ) {
    ap_uint<16> tmp[128];
    for(int i = 0; i<8; i++) {
        processA(&(in[i*128]), tmp);
        processB(tmp, &(out[i*128]));
    }
}
```

When this code is executed, the function `processA` and `processB` are executed sequentially 128 times in a row. Given the combined latency for `processA` and `processB`, the loop is 278 and the total latency can be estimated as:

Figure 67: Performance Estimates



The extra cycle is due to loop setup and can be observed in the Schedule Viewer.

For C/C++ code, task parallelism is performed by adding the `DATAFLOW` pragma into the for-loop:

```
#pragma HLS DATAFLOW
```

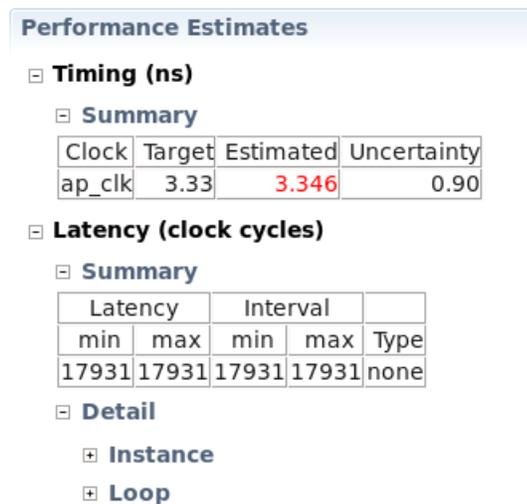
For OpenCL API code, add the attribute before the for-loop:

```
__attribute__((xcl_dataflow))
```

Refer to [Dataflow Optimization](#), [HLS Pragmas](#), and [OpenCL Attributes](#) for more details on this topic.

As illustrated by the estimates in the HLS report, applying the transformation will considerably improve the overall performance effectively using a double (ping pong) buffer scheme between the tasks:

Figure 68: Performances Estimates

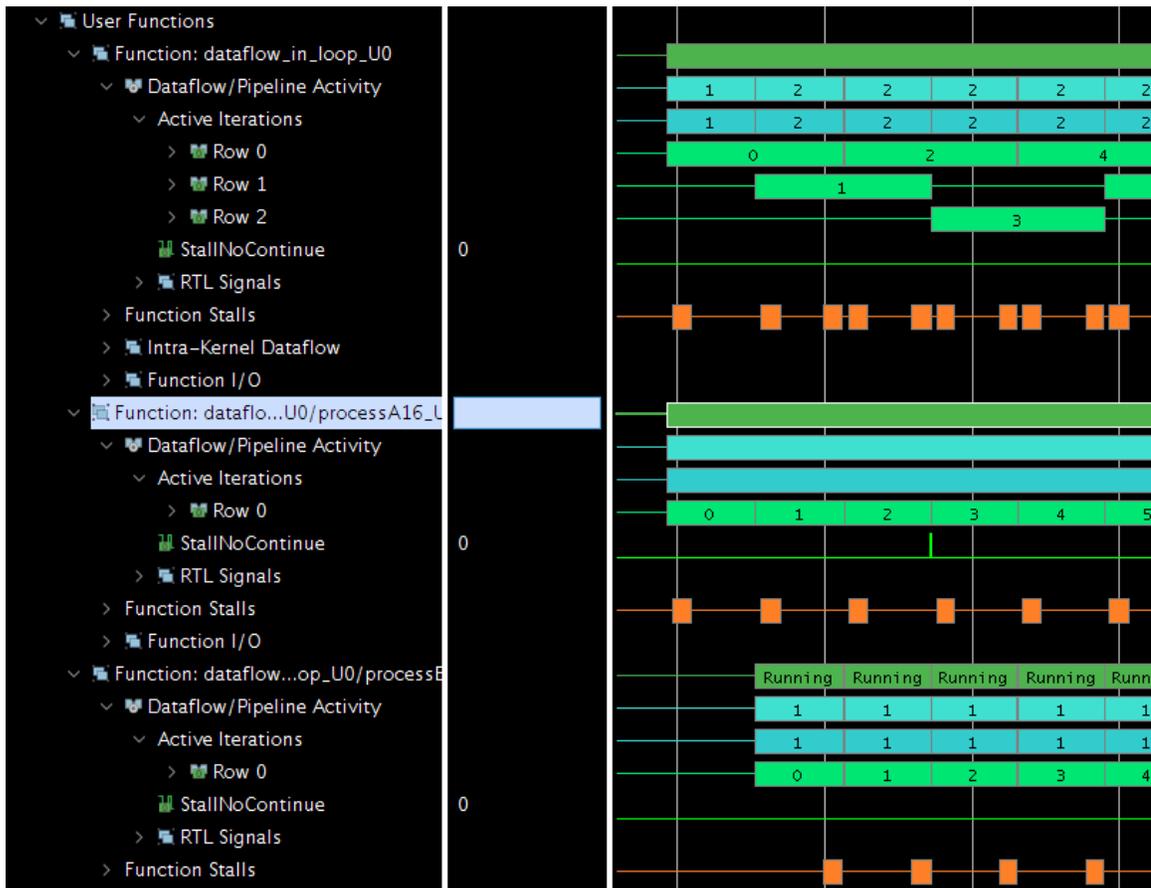


The overall latency of the design has almost halved in this case due to concurrent execution of the different tasks of the different iterations. Given the 139 cycles per processing function and the full overlap of the 128 iterations, this allows the total latency to be:

```
(1x only processA + 127x both processes + 1x only processB) * 139 cycles = 17931 cycles
```

Using task parallelism is a powerful method to improve performance when it comes to implementation. However, the effectiveness of applying the `DATAFLOW` pragma to a specific and arbitrary piece of code might vary vastly. It is often necessary to look at the execution pattern of the individual tasks to understand the final implementation of the `DATAFLOW` pragma. Finally, the Vitis core development kit provides the Detailed Kernel Trace, which illustrates concurrent execution.

Figure 69: Detailed Kernel Trace



For this Detailed Kernel Trace, the tool displays the start of the dataflow loop, as shown in the previous figure. It illustrates how processA is starting up right away with the beginning of the loop, while processB waits until the completion of the processA before it can start up its first iteration. However, while processB completes the first iteration of the loop, processA begins operating on the second iteration, etc.

A more abstract representation of this information is presented in [Application Timeline](#) for the host and device activity.

Optimizing Compute Units

Data Width

One, if not the most important aspect for performance is the data width required for the implementation. The tool propagates port widths throughout the algorithm. In some cases, especially when starting out with an algorithmic description, the C/C++/OpenCL API code might only use large data types such as integers even at the ports of the design. However, as the algorithm is mapped to a fully configurable implementation, smaller data types such as 10-/12-bit might often suffice. It is beneficial to check the size of basic operations in the HLS Synthesis report during optimization.

In general, when the Vitis core development kit maps an algorithm onto the FPGA, more processing is required to comprehend the C/C++/OpenCL API structure and extract operational dependencies. Therefore, to perform this mapping the Vitis core development kit generally partitions the source code into operational units which are then mapped onto the FPGA. Several aspects influence the number and size of these operational units (ops) as seen by the tool.

In the figure, the basic operations and their bit-width are reported.

Figure 70: Operations Utilization Estimates

Utilization Estimates						
<input type="checkbox"/> Summary						
Name	BRAM_18K	DSP48E	FF	LUT		
DSP	-	-	-	-		
Expression	-	-	0	102		
FIFO	-	-	-	-		
Instance	-	-	-	-		
Memory	0	-	24	12		
Multiplexer	-	-	-	80		
Register	-	-	51	-		
Total	0	0	75	194		
Available	4320	5520	1326720	663360		
Available SLR	2160	2760	663360	331680		
Utilization (%)	0	0	~0	~0		
Utilization SLR (%)	0	0	~0	~0		
<input type="checkbox"/> Detail						
<input type="checkbox"/> Instance						
<input type="checkbox"/> DSP48						
<input type="checkbox"/> Memory						
<input type="checkbox"/> FIFO						
<input type="checkbox"/> Expression						
Variable Name	Operation	DSP48E	FF	LUT	Bitwidth P0	Bitwidth P1
i_1_fu_124_p2	+	0	0	11	3	1
i_2_fu_148_p2	+	0	0	15	7	1
i_3_fu_179_p2	+	0	0	15	7	1
sum_i9_fu_194_p2	+	0	0	15	8	8
sum_i_fu_158_p2	+	0	0	15	8	8
exitcond_fu_118_p2	icmp	0	0	9	3	4
exitcond_i6_fu_173_p2	icmp	0	0	11	7	8
exitcond_i_fu_142_p2	icmp	0	0	11	7	8
Total		8	0	0	102	39
<input type="checkbox"/> Multiplexer						
<input type="checkbox"/> Register						

Look for bit widths of 16, 32, and 64 bits commonly used in algorithmic descriptions and verify that the associated operation from the C/C++/OpenCL API source actually requires the bit width to be this large. This can considerably improve the implementation of the algorithm, as smaller operations require less computation time.

Fixed Point Arithmetic

Some applications use floating point computation only because they are optimized for other hardware architecture. As discussed in [Arbitrary Precision Fixed-Point Data Types](#), using fixed point arithmetic for applications like deep learning can save the power efficiency and area significantly while keeping the same level of accuracy.



RECOMMENDED: Xilinx recommends exploring fixed point arithmetic for your application before committing to using floating point operations.

Macro Operations

It is sometimes advantageous to think about larger computational elements. The tool will operate on the source code independently of the remaining source code, effectively mapping the algorithm without consideration of surrounding operations onto the FPGA. When applied, the Vitis technology keeps operational boundaries, effectively creating macro operations for specific code. This uses the following principles:

- Operational locality to the mapping process
- Reduction in complexity for the heuristics

This might create vastly different results when applied. In C/C++, macro operations are created with the help of `#pragma HLS inline off`. While in the OpenCL API, the same kind of macro operation can be generated by *not* specifying the following attribute when defining a function:

```
__attribute__((always_inline))
```

For more information, see [pragma HLS inline](#).

Using Optimized Libraries

The OpenCL specification provides many math built-in functions. All math built-in functions with the `native_` prefix are mapped to one or more native device instructions and will typically have better performance compared to the corresponding functions (without the `native_` prefix). The accuracy and in some cases the input ranges of these functions is implementation-defined. In the Vitis technology, these `native_` built-in functions use the equivalent functions in the Vivado HLS tool Math library, which are already optimized for Xilinx FPGAs in terms of area and performance.



RECOMMENDED: Xilinx recommends using `native_` built-in functions or the HLS tool Math library if the accuracy meets the application requirement.

Optimizing Memory Architecture

Memory architecture is a key aspect of implementation. Due to the limited access bandwidth, it can heavily impact the overall performance, as shown in the following example:

```
void run (ap_uint<16> in[256][4],
         ap_uint<16> out[256]
        ) {
    ...
    ap_uint<16> inMem[256][4];
    ap_uint<16> outMem[256];
```

```

... Preprocess input to local memory

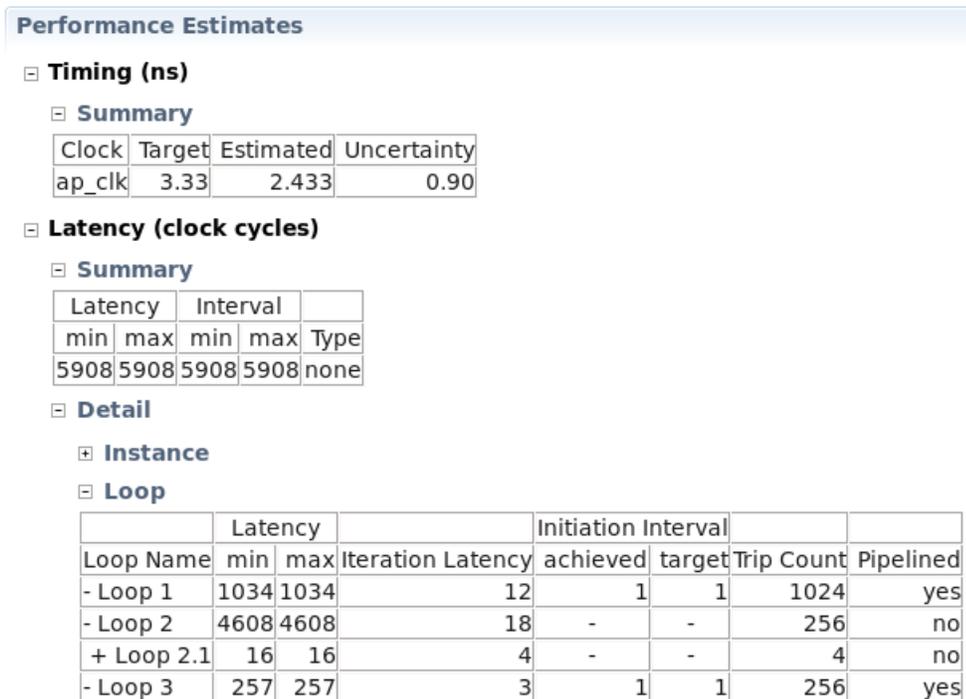
for( int j=0; j<256; j++) {
    #pragma HLS PIPELINE OFF
    ap_uint<16> sum = 0;
    for( int i = 0; i<4; i++) {

        sum += inMem[j][i];
    }
    outMem[j] = sum;
}

... Postprocess write local memory to output
}
    
```

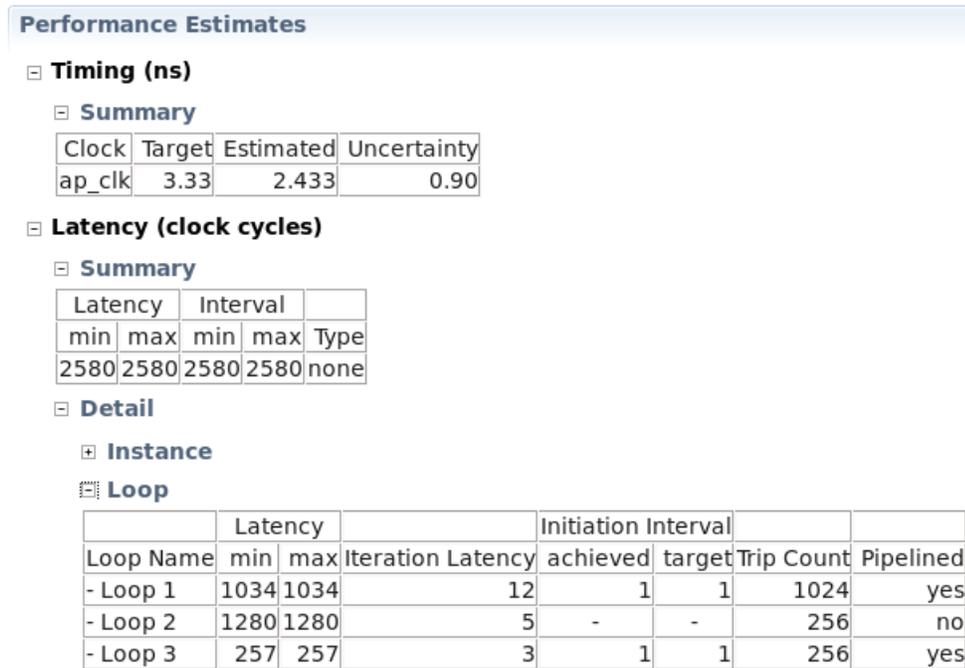
This code adds the four values associated with the inner dimension of the two dimensional input array. If implemented without any additional modifications, it results in the following estimates:

Figure 71: Performance Estimates



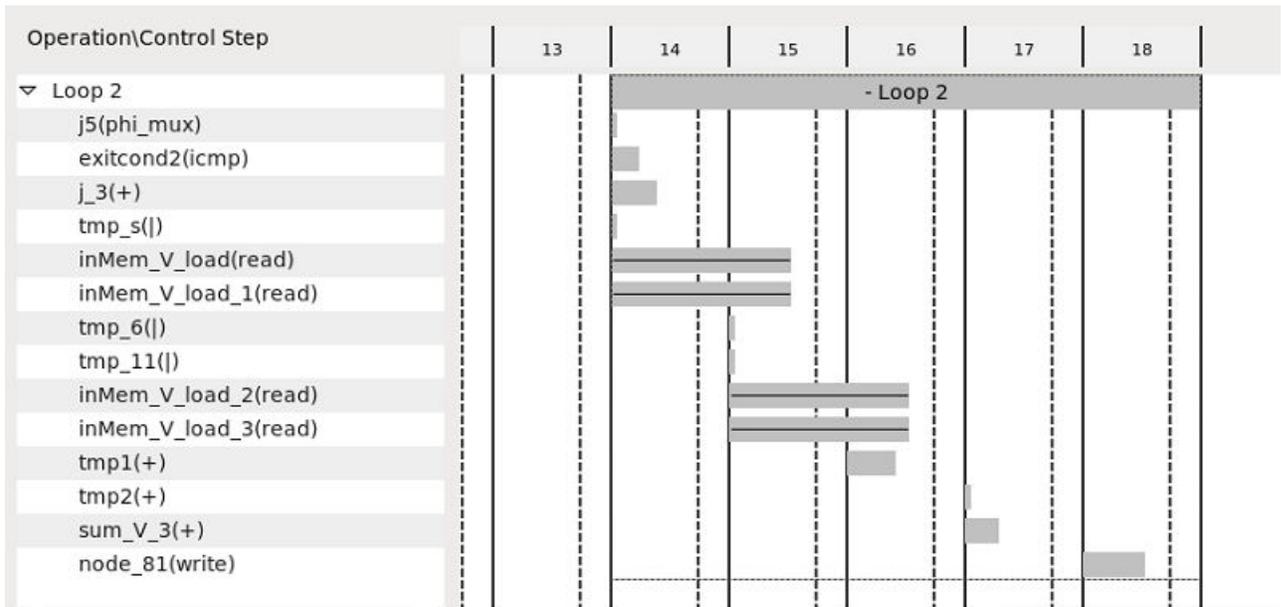
The overall latency of 4608 (Loop 2) is due to 256 iterations of 18 cycles (16 cycles spent in the inner loop, plus the reset of sum, plus the output being written). This is observed in the Schedule Viewer in the HLS Project. The estimates become considerably better when unrolling the inner loop.

Figure 72: Performance Estimates



However, this improvement is largely because of the process using both ports of a dual port memory. This can be seen from the Schedule Viewer in the HLS Project:

Figure 73: Schedule Viewer



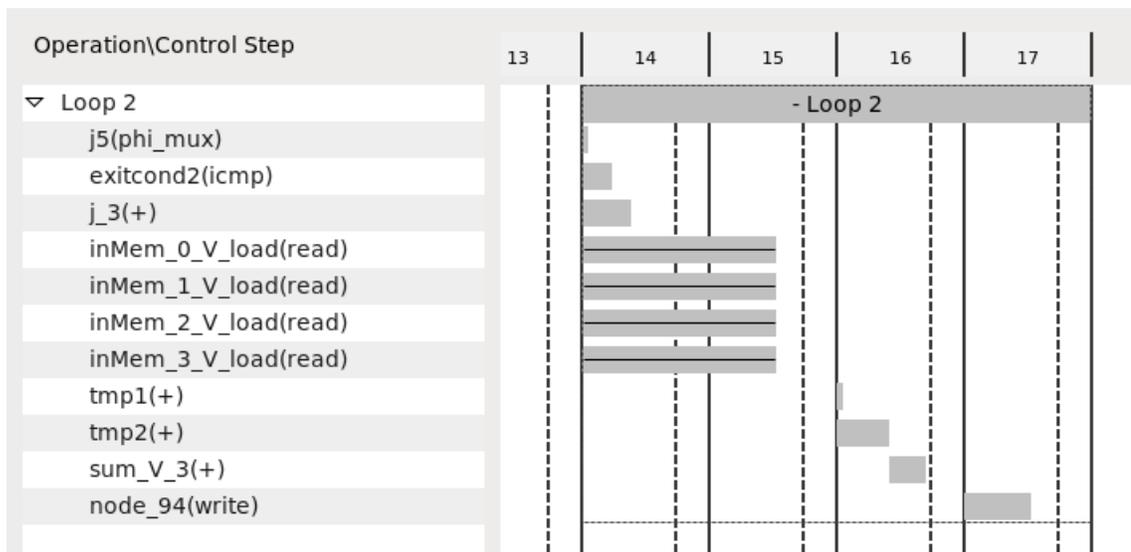
Two read operations are performed per cycle to access all the values from the memory to calculate the sum. This is often an undesired result as this completely blocks the access to the memory. To further improve the results, the memory can be split into four smaller memories along the second dimension:

```
#pragma HLS ARRAY_PARTITION variable=inMem complete dim=2
```

For more information, see [pragma HLS array_partition](#).

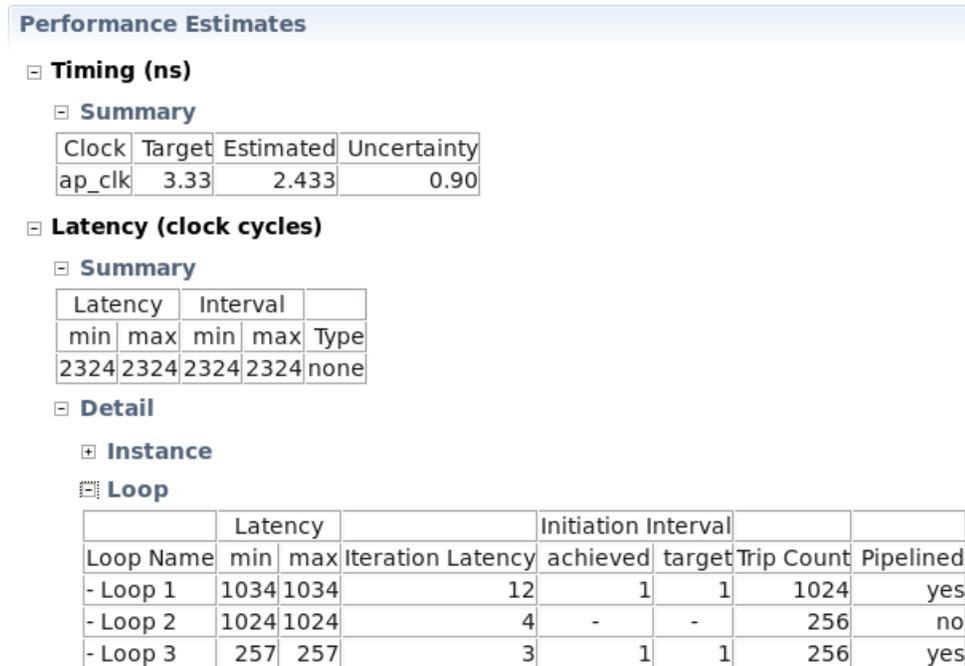
This results in four array reads, all executed on different memories using a single port:

Figure 74: Executed Four Arrays Results



Using a total of $256 * 4$ cycles = 1024 cycles for loop 2.

Figure 75: Performance Estimates



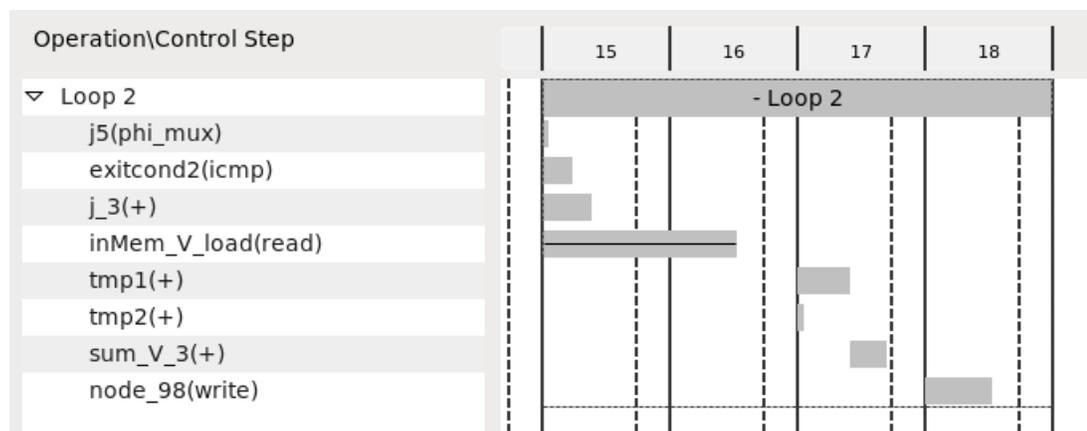
Alternatively, the memory can be reshaped into to a single memory with four words in parallel. This is performed through the pragma:

```
#pragma HLS array_reshape variable=inMem complete dim=2
```

For more information, see [pragma HLS array_reshape](#).

This results in the same latency as when the array partitioning, but with a single memory using a single port:

Figure 76: Latency Result



Although, either solution creates comparable results with respect to overall latency and utilization, reshaping the array results in cleaner interfaces and less routing congestion making this the preferred solution.

Note: This completes array optimization, in a real design the latency could be further improved by employing loop parallelism (see [Loop Parallelism](#)).

```
void run (ap_uint<16> in[256][4],
         ap_uint<16> out[256]
        ) {
    ...

    ap_uint<16> inMem[256][4];
    ap_uint<16> outMem[256];
    #pragma HLS array_reshape variable=inMem complete dim=2

    ... Preprocess input to local memory

    for( int j=0; j<256; j++) {
        #pragma HLS PIPELINE OFF
        ap_uint<16> sum = 0;
        for( int i = 0; i<4; i++) {
            #pragma HLS UNROLL
            sum += inMem[j][i];
        }
        outMem[j] = sum;
    }

    ... Postprocess write local memory to output
}
```

Kernel SLR and DDR Memory Assignments

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

The largest Xilinx FPGAs are made up of multiple stacked silicon dies. Each stack is referred to as a super logic region (SLR) and has a fixed amount of resources and memory including DDR interfaces. Available device SLR resources which can be used for custom logic can be found in the [Vitis 2019.2 Software Platform Release Notes](#), or can be displayed using the `platforminfo` utility described in [Platforminfo](#).

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

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

Note: While compute units can be connected to any available DDR memory resource, it is also necessary to account for the bandwidth requirements of the kernels when assigning to SLRs.

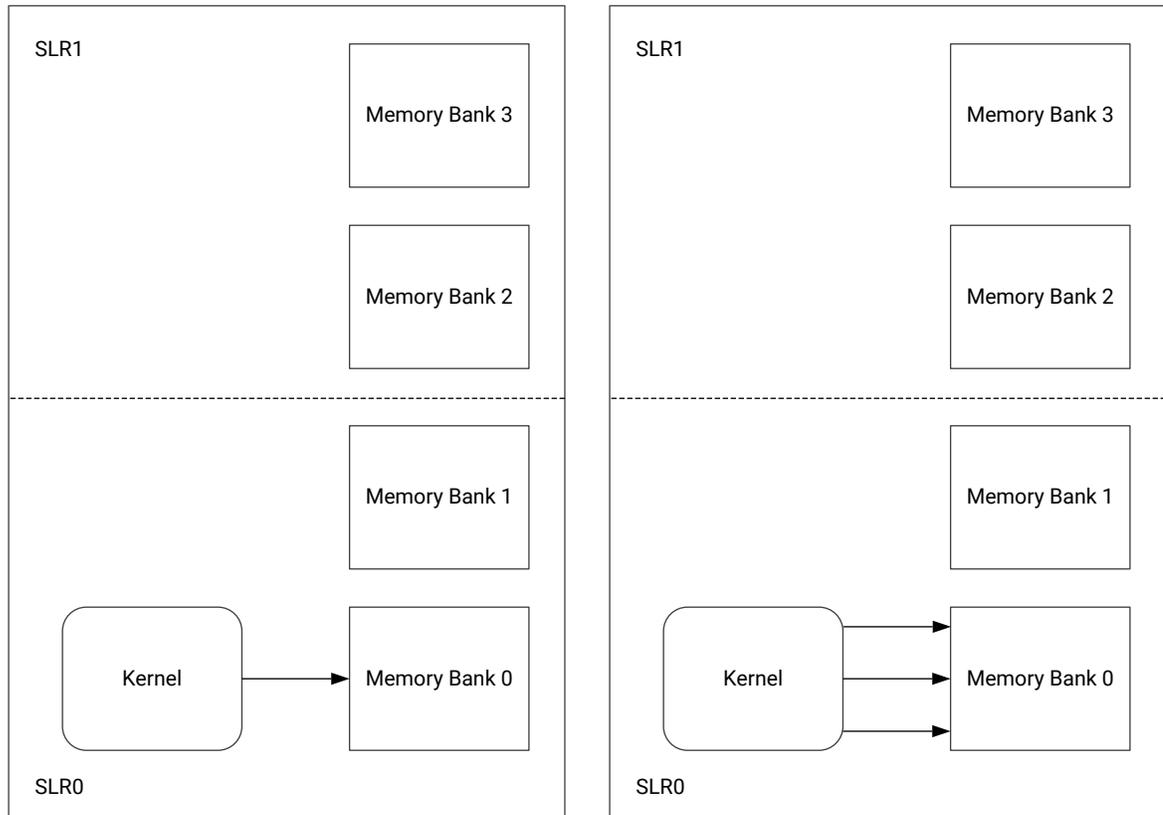
After allocating your CUs to SLRs, map any CU master AXI port(s) to DDR memory resources. Xilinx recommends connecting to a DDR memory resource in the same SLR as the CU. This reduces competition for the limited SLR-crossing connection resources. In addition, connections between SLRs use super long line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

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

Guidelines for Kernels that Access Multiple Memory Banks

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

Figure 77: Kernel and Memory in Same SLR



X22194-010919

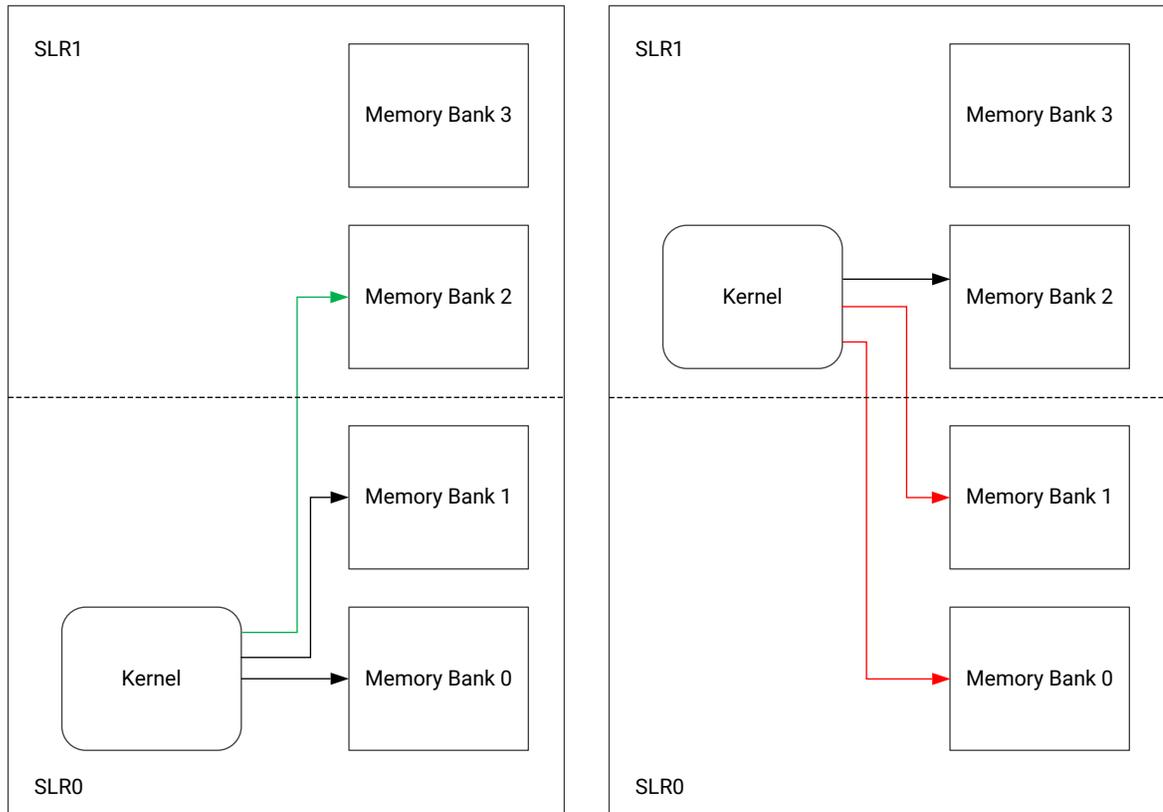
Note: The image on the left shows a single AXI interface mapped to a single memory bank. The image on the right shows multiple AXI interfaces mapped to the same memory bank.

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](#) 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 78: Memory Bank in Adjoining SLR



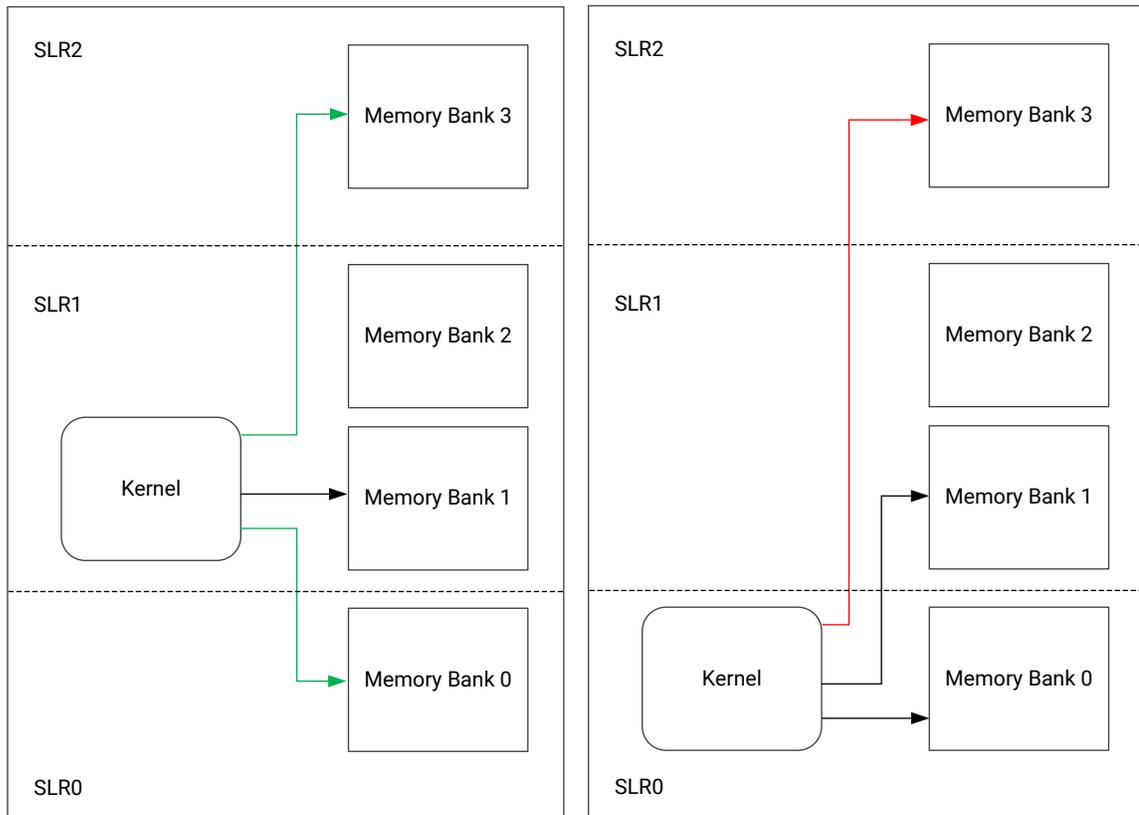
X22195-010919

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 in 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 79: Memory Banks Two SLRs Away



X22196-010919

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.

Exploring Kernel Optimizations Using Vivado HLS

All kernel optimizations using OpenCL or C/C++ can be performed from within the Vitis core development kit. The primary performance optimizations, such as those discussed in this section (pipelining function and loops, applying dataflow to enable greater concurrency between functions and loops, unrolling loops, etc.), are performed by the Vivado HLS tool.

The Vitis core development kit automatically calls the HLS tool. However, to use the GUI analysis capabilities, you must launch the HLS tool directly from within the Vitis technology. Using the HLS tool in standalone mode, as discussed in [Compiling Kernels Directly in Vivado HLS](#), enables the following enhancements to the optimization methodology:

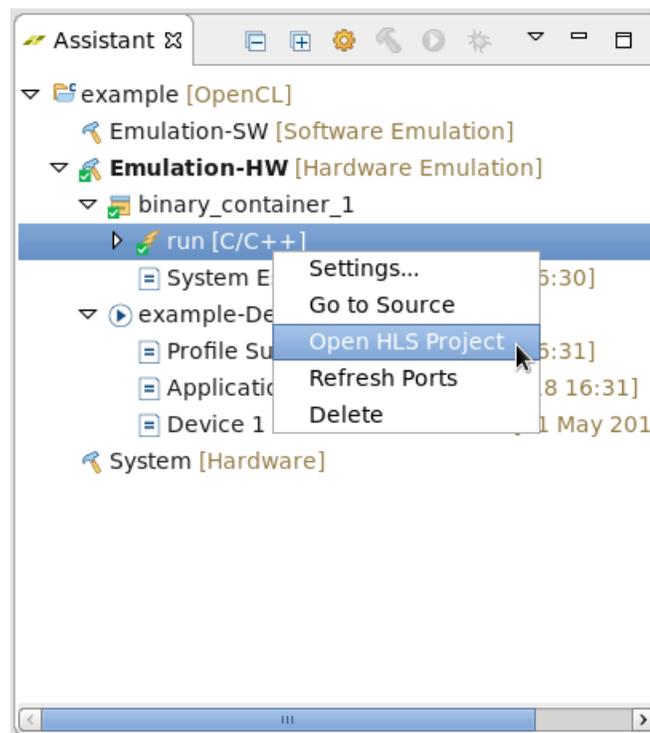
- The ability to focus solely on the kernel optimization because there is no requirement to execute emulation.
- The skill to create multiple solutions, compare their results, and explore the solution space to find the most optimum design.
- The competence to use the interactive Analysis Perspective to analyze the design performance.



IMPORTANT! Only the kernel source code is incorporated back into the Vitis core development kit. After exploring the optimization space, ensure that all optimizations are applied to the kernel source code as OpenCL attributes or C/C++ pragmas.

To open the HLS tool in standalone mode, from the Assistant window, right-click the hardware function object, and select **Open HLS Project**, as shown in the following figure.

Figure 80: Open HLS Project



Topological Optimization

This section focuses on the topological optimization. It looks at the attributes related to the rough layout and implementation of multiple compute units and their impact on performance.

Multiple Compute Units

Depending on available resources on the target device, multiple compute units of the same kernel (or different kernels) can be created to run in parallel, which improves the system processing time and throughput. For more details, see [Creating Multiple Instances of a Kernel](#).

Using Multiple DDR Banks

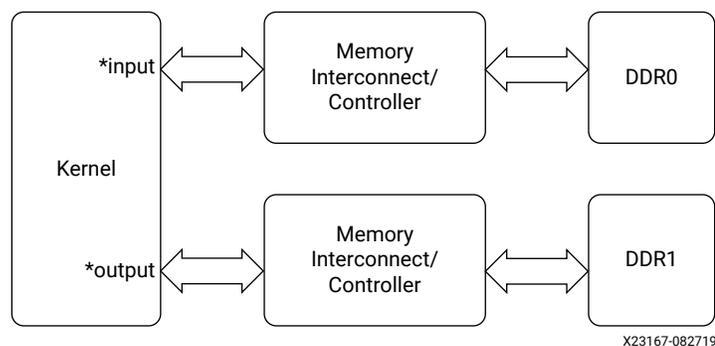
Acceleration cards supported in Vitis technology provide one, two, or four DDR banks, and up to 80 GB/s raw DDR bandwidth. For kernels moving large amount of data between the FPGA and the DDR, Xilinx® recommends that you direct the Vitis compiler and runtime library to use multiple DDR banks.

In addition to DDR banks, the host application can access PLRAM to transfer data directly to a kernel. This feature is enabled using the `connectivity.sp` option in a configuration file specified with the `v++ --config` option. Refer to [Mapping Kernel Ports to Global Memory](#) for more information on implementing this optimization and [Memory Mapped Interfaces](#) on data transfer to the global memory banks.

To take advantage of multiple DDR banks, you need to assign CL memory buffers to different banks in the host code as well as configure the `xclbin` file to match the bank assignment in `v++` command line.

The following block diagram shows the [Global Memory Two Banks \(C\)](#) example in [Vitis Examples](#) on GitHub. This example connects the input pointer interface of the kernel to DDR bank 0, and the output pointer interface to DDR bank 1.

Figure 81: Global Memory Two Banks Example



Assigning DDR Bank in Host Code

Bank assignment in host code is supported by Xilinx vendor extension. The following code snippet shows the header file required, as well as assigning input and output buffers to DDR bank 0 and bank 1, respectively:

```
#include <CL/cl_ext.h>
...
int main(int argc, char** argv)
{
    ...
    cl_mem_ext_ptr_t inExt, outExt; // Declaring two extensions for both
    buffers
    inExt.flags = 0|XCL_MEM_TOPOLOGY; // Specify Bank0 Memory for input
    memory
    outExt.flags = 1|XCL_MEM_TOPOLOGY; // Specify Bank1 Memory for output
    Memory
    inExt.obj = 0 ; outExt.obj = 0; // Setting Obj and Param to Zero
    inExt.param = 0 ; outExt.param = 0;

    int err;
    //Allocate Buffer in Bank0 of Global Memory for Input Image using
    Xilinx Extension
    cl_mem buffer_inImage = clCreateBuffer(world.context, CL_MEM_READ_ONLY
    | CL_MEM_EXT_PTR_XILINX,
        image_size_bytes, &inExt, &err);
    if (err != CL_SUCCESS){
        std::cout << "Error: Failed to allocate device Memory" << std::endl;
        return EXIT_FAILURE;
    }
    //Allocate Buffer in Bank1 of Global Memory for Input Image using
    Xilinx Extension
    cl_mem buffer_outImage = clCreateBuffer(world.context,
    CL_MEM_WRITE_ONLY | CL_MEM_EXT_PTR_XILINX,
        image_size_bytes, &outExt, NULL);
    if (err != CL_SUCCESS){
        std::cout << "Error: Failed to allocate device Memory" << std::endl;
        return EXIT_FAILURE;
    }
    ...
}
```

`cl_mem_ext_ptr_t` is a struct as defined below:

```
typedef struct{
    unsigned flags;
    void *obj;
    void *param;
} cl_mem_ext_ptr_t;
```

- Valid values for `flags` are:
 - `XCL_MEM_DDR_BANK0`
 - `XCL_MEM_DDR_BANK1`
 - `XCL_MEM_DDR_BANK2`
 - `XCL_MEM_DDR_BANK3`

- `<id> | XCL_MEM_TOPOLOGY`

Note: The `<id>` is determined by looking at the Memory Configuration section in the `xxx.xclbin.info` file generated next to the `xxx.xclbin` file. In the `xxx.xclbin.info` file, the global memory (DDR, PLRAM, etc.) is listed with an index representing the `<id>`.

- `obj` is the pointer to the associated host memory allocated for the CL memory buffer only if `CL_MEM_USE_HOST_PTR` flag is passed to `clCreateBuffer` API, otherwise set it to `NULL`.
- `param` is reserved for future use. Always assign it to 0 or `NULL`.

Assigning Global Memory for Kernel Code

Creating Multiple AXI Interfaces

OpenCL kernels, C/C++ kernels, and RTL kernels have different methods for assigning function parameters to AXI interfaces.

- For OpenCL kernels, the `--max_memory_ports` option is required to generate one AXI4 interface for each global pointer on the kernel argument. The AXI4 interface name is based on the order of the global pointers on the argument list.

The following code is taken from the example `gmem_2banks_ocl` in the `kernel_to_gmem` category from the [Vitis Getting Started Examples](#) on GitHub:

```
__kernel __attribute__((reqd_work_group_size(1, 1, 1)))
void apply_watermark(__global const TYPE * __restrict input,
__global TYPE * __restrict output, int width, int height) {
    ...
}
```

In this example, the first global pointer `input` is assigned an AXI4 name `M_AXI_GMEM0`, and the second global pointer `output` is assigned a name `M_AXI_GMEM1`.

- For C/C++ kernels, multiple AXI4 interfaces are generated by specifying different “bundle” names in the HLS INTERFACE pragma for different global pointers. Refer to [Kernel Interfaces and Memory Banks](#) for more information.

The following is a code snippet from the `gmem_2banks_c` example that assigns the `input` pointer to the bundle `gmem0` and the `output` pointer to the bundle `gmem1`. The bundle name can be any valid C string, and the AXI4 interface name generated will be `M_AXI_<bundle_name>`. For this example, the input pointer will have AXI4 interface name as `M_AXI_gmem0`, and the output pointer will have `M_AXI_gmem1`. Refer to [pragma HLS interface](#) for more information.

```
#pragma HLS INTERFACE m_axi port=input offset=slave bundle=gmem0
#pragma HLS INTERFACE m_axi port=output offset=slave bundle=gmem1
```

- For RTL kernels, the port names are generated during the import process by the RTL kernel wizard. The default names proposed by the RTL kernel wizard are `m00_axi` and `m01_axi`. If not changed, these names have to be used when assigning a DDR bank through the `connectivity.sp` option in the configuration file. Refer to [Mapping Kernel Ports to Global Memory](#) for more information.

Assigning AXI Interfaces to DDR Banks



IMPORTANT! When using more than one DDR interface, Xilinx requires you to specify the DDR memory bank for each kernel/CU, and specify the SLR to place the kernel into. For more information, see [Mapping Kernel Ports to Global Memory](#) and [Assigning Compute Units to SLRs](#).

The following is an example configuration file that specifies the `connectivity.sp` option, and the `v++` command line that connects the input pointer (`M_AXI_GMEM0`) to DDR bank 0 and the output pointer (`M_AXI_GMEM1`) to DDR bank 1:

The `config_sp.txt` file:

```
[connectivity]
sp=apply_watermark_1.m_axi_gmem0:DDR[0]
sp=apply_watermark_1.m_axi_gmem1:DDR[1]
```

The `v++` command line:

```
v++ apply_watermark --config config_sp.txt
```

You can use the Device Hardware Transaction view to observe the actual DDR Bank communication, and to analyze DDR usage.

Figure 82: Device Hardware Transaction View Transactions on DDR Bank



Assigning AXI Interfaces to PLRAM

Some platforms support PLRAMs. In these cases, use the same `--sp` option as described in [Assigning AXI Interfaces to DDR Banks](#), but use the name, `PLRAM[id]`. Valid names supported by specific platforms can be found in the Memory Configuration section of the `xclibin.info` file generated alongside `xclbin`.

Assigning Kernels to SLR Regions

Assigning ports to global memory banks requires the kernel to be physically routed on the FPGA, to connect to the assigned DDR, HBM, or block RAM. Currently, large FPGAs use stacked silicon devices with several super logic regions (SLRs). By default, the Vitis core development kit will place the compute units in the same SLR as the target platform. This is not always desirable, especially when the kernel connects to specific memory banks in a different SLR region. In this case, you will want to manually assign the kernel instance, or CU into the same SLR as the global memory. For more information, see [Mapping Kernel Ports to Global Memory](#).

You can assign the CU instance to an SLR using the `connectivity.slr` option described in [Assigning Compute Units to SLRs](#).



TIP: To better understand the platform attributes, such as the number of DDRs and SLR regions, you can detail the target platform using the `platforminfo` command described in [Platforminfo](#).

Debugging Applications and Kernels

The Vitis™ core development kit 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 debugging and hardware-centric debugging flows.

For hardware-centric debugging, kernels running on hardware can be debugged using Xilinx® virtual cable (XVC) running over the PCIe® bus, for Alveo™ Data Center accelerator cards, and debugged using USB-JTAG cables for both Alveo cards and embedded processor platforms.

Debugging Features and Techniques

The following table lists the features or techniques that can be used for debugging your application in the different build configurations.

Table 11: Features and Techniques for Debugging Different Build Configurations

Feature/ Technique	OS	Host	Kernel	FPGA (Platform)
Software Emulation	-	GDB	GDB	-
Hardware Emulation	-	GDB	GDB Kernel Waveform Viewer	-

Table 11: Features and Techniques for Debugging Different Build Configurations
(cont'd)

Feature/ Technique	OS	Host	Kernel	FPGA (Platform)
System	dmesg	GDB	Kernel Waveform Viewer ChipScope	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 12: Software and Hardware Debugging Features and Techniques

Software-centric	Hardware-centric
GNU Debugger (GDB)	Kernel waveform viewer
Xilinx utility <code>xbutil</code>	ChipScope
Linux <code>dmesg</code>	-

Using both software-centric and hardware-centric debugging features, you can isolate and identify functional issues, protocol issues, and troubleshoot board hangs.

Debugging in Hardware Emulation

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

GDB-Based Debugging

Debugging using a software-based GDB flow is fully supported during hardware emulation. Except for the execution of the actual RTL code representing the kernel code, there is no difference to you because GDB maps the RTL back into the source code description. This limits the breakpoints and observability of the variables in some cases, because during the RTL generation by Vivado HLS, some variables and loops from the kernel source might have been dissolved.

Waveform-Based Kernel Debugging

The C/C++ and OpenCL kernel code is synthesized using Vivado HLS to transform it into a Hardware Description Language (HDL) and later implement it in the Vivado Design Suite to produce the FPGA binary (`xclbin`).

Another debugging approach is based on simulation waveforms. Hardware-centric designers are likely to be familiar with this approach. This waveform-based HDL debugging is supported by the Vitis core development kit using both the command line flow, or through the IDE flow during hardware emulation.



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

Enable Waveform Debugging with the Vitis Compiler Command

The waveform debugging process can be enabled through the `v++` command using the following steps:

1. Turn on debug code generation during kernel compilation.

```
v++ -g ...
```

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

```
[Emulation]
launch_waveform=batch

[Debug]
profile=true
timeline_trace=true
data_transfer_trace=fine
```

3. Execute hardware emulation. The hardware transaction data is collected in the file named `<hardware_platform>-<device_id>-<xclbin_name>.wdb`. This file can directly be opened through the Vitis IDE.



TIP: *If `launch_waveform` is set to `gui` in the `xrt.ini`, a live waveform viewer is spawned during the execution of the hardware emulation.*

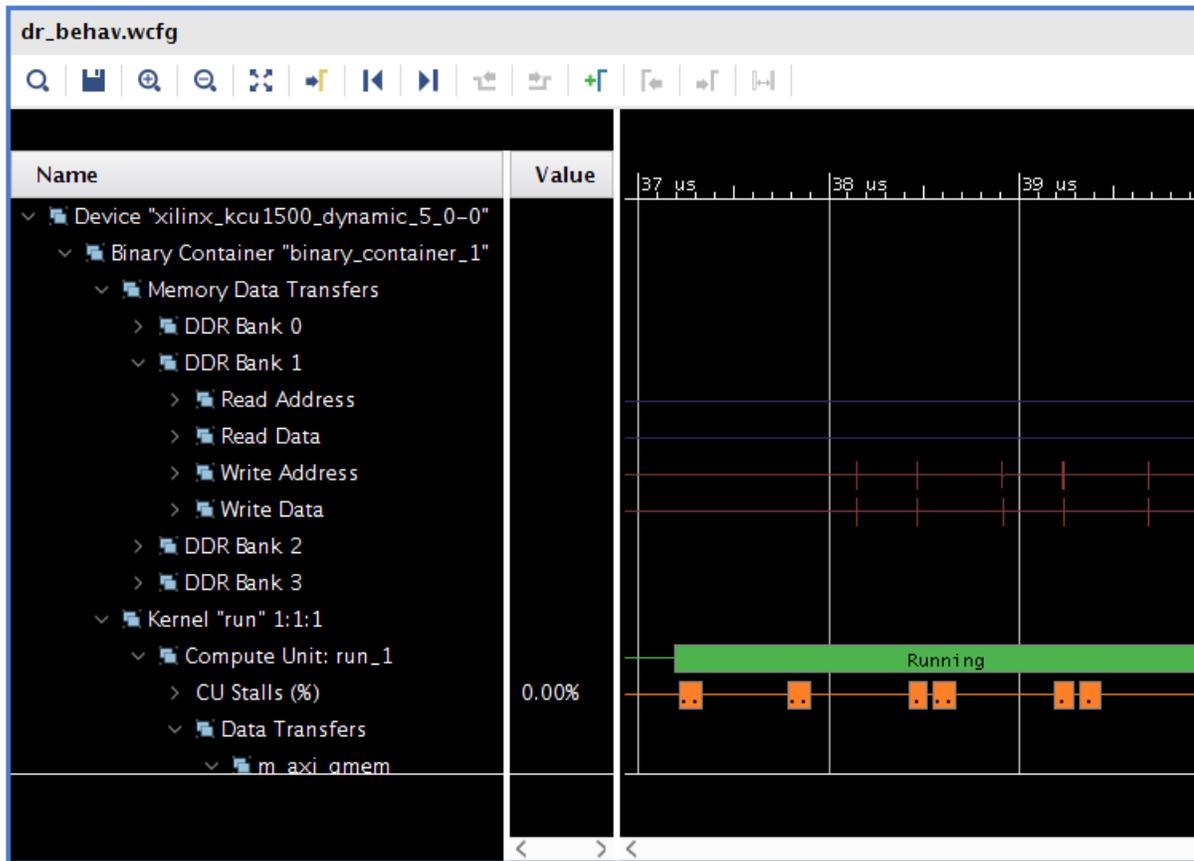
Run the Waveform-Based Kernel Debugging Flow

The Vitis IDE provides waveform-based HDL debugging in the hardware emulation mode. The waveform is opened in the Vivado waveform viewer which should be familiar to Vivado logic simulation users. The Vitis IDE lets you display kernel interfaces, internal signals, and includes debug controls such as restart, HDL breakpoints, as well as HDL code lookup and waveform markers. In addition, it provides top-level DDR data transfers (per bank) along with kernel-specific details including compute unit stalls, loop pipeline activity, and data transfers.

For details, see [Waveform View and Live Waveform Viewer](#).

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

Figure 83: Waveform Viewer



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



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

- **Kernel <kernel name><workgroup size> Compute Unit<CU name>:** Kernel name, workgroup size, and compute unit name.
- **CU Stalls (%):** This shows a summary of stalls for the entire CU. A bus of all lowest-level stall signals is created, and the bus is represented in the waveform as a percentage (%) of those signals that are active at any point in time.
- **Data Transfers:** This shows the data transfers for all AXI masters on the CU.
- **User Functions:** This lists all of the functions within the hierarchy of the CU.
- **Function:** <function name>

- **Dataflow/Pipeline Activity:** This shows the function-level loop dataflow/pipeline signals for a CU.
- **Function Stalls:** This lists the three stall signals within this function.
- **Function I/O:** This lists the I/O for the function. These I/O are of protocol `-m_axi`, `ap_fifo`, `ap_memory`, or `ap_none`.



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

Hardware Debugging Using ChipScope

After the final system image (`xclbin`) is generated and executed on the Vitis target platform, the entire system including the host application running on the CPU, and the Vitis accelerated kernels on the Xilinx FPGA, can be confirmed to be executing correctly on the actual hardware. At this stage, you can validate the functioning of the host code and kernel in the target hardware and debug any issues found. Some of the conditions that can be reviewed or analyzed are listed as follows:

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

ILA

The Vitis core development kit provides insertion of the System Integrated Logic Analyzer (ILA) into a design to capture and view AXI transaction level activity by probing the signals between kernel interfaces and global memory. The ILA provides custom event triggering on one or more signals to allow waveform capture at system speeds. The waveforms can be analyzed in a viewer and used to debug hardware, finding protocol violations or performance issues for example, and can be crucial for debugging difficult situation like application hangs.

Captured data can be accessed through the Xilinx virtual cable (XVC) using the Vivado tools. See the *Vivado Design Suite User Guide: Programming and Debugging* (UG908) for complete details.

Note: ILA debug cores require system resources, including logic and local memory to capture and store the signal data. Therefore they provide excellent visibility into your kernel, but they can affect both performance and resource utilization.

System ILAs can be inserted into the design using the `v++ --dk` option as shown below:

```
$ v++ --dk chipscope:<compute_unit_name>:<interface_name>
```

Refer to the [Vitis Compiler Command](#) for more information.

Checking the FPGA Board for Hardware Debug Support

Supporting hardware debugging requires the platform to support several IP components, most notably the Debug Bridge. Talk to your platform designer to determine if these components are included in the target platform. If a Xilinx platform is used, debug availability can be verified using the `platforminfo` utility to query the platform. Debug capabilities are listed under the `chipscope_debug` objects.

For example, to query the a platform for hardware debug support, the following `platforminfo` command can be used. A response can be seen showing that the platform contains a user and management debug network, and also supports debugging a MicroBlaze™ processor.

```
$ platforminfo --json="hardwarePlatform.extensions.chipscope_debug" --
platform xilinx_u200_xdma_201830_1
{
  "debug_networks": {
    "user": {
      "name": "User Debug Network",
      "pcie_pf": "1",
      "bar_number": "0",
      "axi_baseaddr": "0x000C0000",
      "supports_jtag_fallback": "false",
      "supports_microblaze_debug": "true",
      "is_user_visible": "true"
    },
    "mgmt": {
      "name": "Management Debug Network",
      "pcie_pf": "0",
      "bar_number": "0",
      "axi_baseaddr": "0x001C0000",
      "supports_jtag_fallback": "true",
```

```

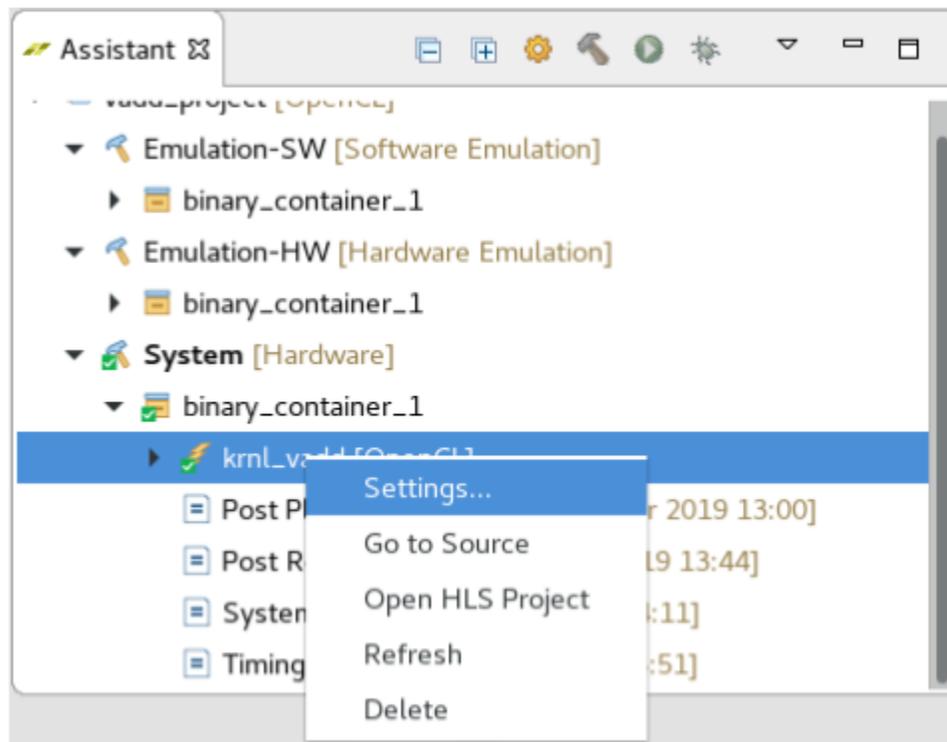
        "supports_microblaze_debug": "true",
        "is_user_visible": "false"
    }
}
}

```

Enabling ChipScope from the Vitis IDE

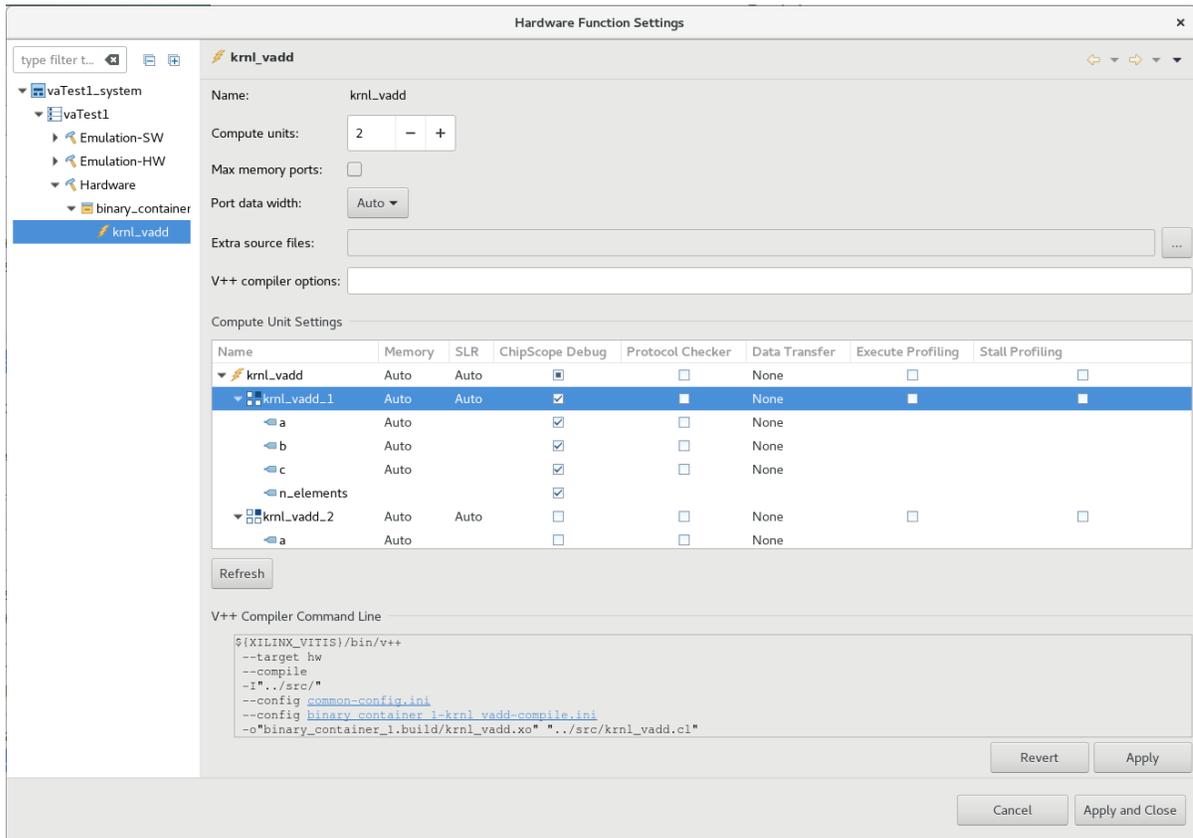
The Vitis IDE provides options to enable the ChipScope debug feature on all the interface ports of the compute units in the design. When enabling this option on a compute unit, the Vitis compiler adds a System ILA debug core to monitor the interface ports of the compute unit. This ensures that you can debug the interface signals on the Vitis target platform hardware while the kernel is running. You can access this through the Settings command by right-clicking on a kernel in the system build configuration in the Assistant window as shown below.

Figure 84: Vitis Assistant View



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

Figure 85: Vitis Hardware Function Settings



TIP: If you enable the **ChipScope Debug** option on larger designs with multiple kernels and/or compute units can result in overuse of the FPGA resources. Xilinx recommends using the `v++ --dk list_ports` option on the command line to determine the number and type of interfaces on the compute units. If you know which ports need to be monitored for debug as the design runs in hardware, the recommended methodology is to use the `--dk` option documented in the following topic.

Command Line Flow

The following section covers the `v++` linker options that can be used to list the available kernel ports, as well as enable the ILA core on the selected ports.

The ILA core provides transaction-level visibility into an instance of a compute unit (CU) running on hardware. AXI traffic of interest can also be captured and viewed using the ILA core. The ILA core can be added to an existing RTL kernel to enable debugging features within that design, or it can be inserted automatically by the `v++` compiler during the linking stage. The `v++` command provides the `--dk` option to attach System ILA cores at the interfaces to the kernels for debugging and performance monitoring purposes.

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

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

In general, the `<interface_name>` is optional. If not specified, all ports are expected to be analyzed. The `chipscope` option requires the explicit name of the compute unit to be provided for the `<compute_unit_name>` and `<interface_name>`. The `list_ports` option generates a list of valid compute units and port combinations in the current design and must be used after the kernel has been compiled.

First, you must compile the kernel source files into an `.xo` file:

```
v++ -c -k <kernel_name> --platform <platform> -o <kernel_xo_file>.xo
<kernel_source_files>
```

After the kernel has been compiled into an `.xo` file, `--dk list_ports` can be added to the command line options used during the `v++` linking process. This causes the `v++` compiler to print the list of valid compute units and port combinations for the kernel. See the following example:

```
v++ -l --platform <platform> --nk
<kernel_name>:<compute_units>:<kernel_nameN> --dk list_ports
<kernel_xo_file>.xo
```

Finally, ChipScope debug can be enabled on the desired ports by replacing `list_ports` with the appropriate `--dk chipscope` command syntax:

```
v++ -l --platform <platform> --nk
<kernel_name>:<compute_units>:<kernel_nameN> --dk
chipscope:<compute_unit_name>:<interface_name> <kernel_xo_file>.xo
```

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

When the design is built, you can debug the design using the Vivado® hardware manager as described in [Debugging with ChipScope](#).

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.

Debugging Flows

The Vitis core development kit provides application-level debug features which allow the host code, the kernel code, and the interactions between them to be efficiently debugged. The recommended application-level debugging flow consists of three levels of debugging:

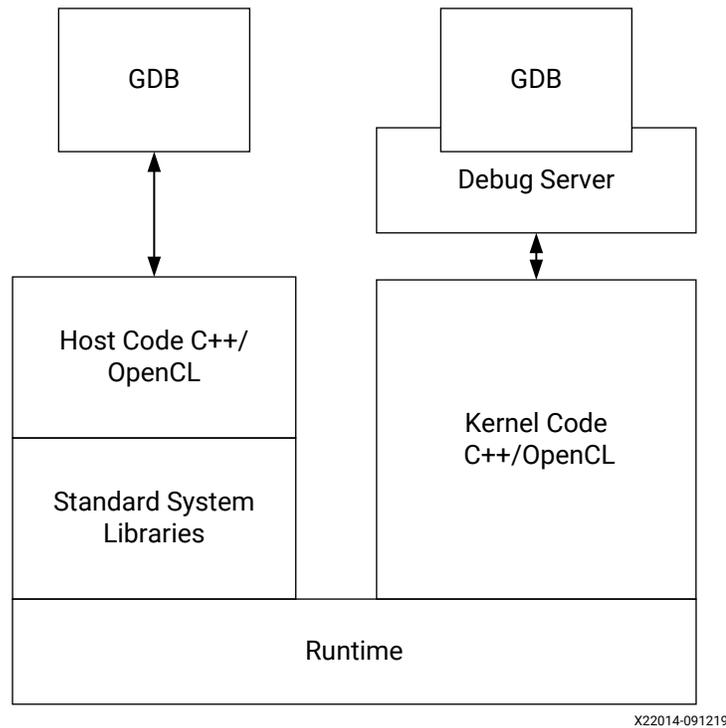
- Perform software emulation (`sw_emu`) to confirm the algorithm functionality.
- Perform hardware emulation (`hw_emu`) to compile the kernel for hardware, and confirm the correctness of the generated logic, and evaluate its simulated performance.
- Perform a system build hardware (`hw`) to implement and test the application running in hardware.

This three-tiered approach allows debugging the host and kernel code, and the interactions between them at different levels of abstraction. Each provides specific insights into the design and makes debugging easier. All flows are supported through an integrated GUI flow as well as through a batch flow using basic compile time and runtime setup options. A brief description of each flow follows.

Software Emulation

The following figure shows the software emulation flow diagram for host and kernel code (written in C/C++ or OpenCL™). The GNU debugging (GDB) can be used to debug both the host and kernel code. Xilinx recommends iterating the design as much as possible in Software Emulation, which takes little compile time and executes quickly. For more detailed information on software emulation, see [Software Emulation](#).

Figure 86: Software Emulation



Vitis Emulation Debug

The Vitis core development kit supports typical software-like debugging for the host as well as kernel code. This flow is supported during software and hardware emulation and allows the use of breakpoints and the analysis of variables as commonly done during software debugging.

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

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. Vitis 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 Vitis core development kit 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 `v++` command line executable.

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 information, see [GDB-Based Debugging](#).

Command Line Debug Flow

The command line debug flow in the Vitis core development kit provides tools to debug the host and kernel application running in all modes: software emulation, hardware emulation, or hardware execution.

Note: You can debug host code using this feature only when running in hardware execution mode.

There are four steps to debugging in the Vitis core development kit using the command line flow:

1. General environment setup, as discussed in [Running an Application](#).
2. Prepare the host code for debug.
3. Prepare the kernel code for debug.
4. Launch GDB Standalone to debug.



IMPORTANT! *The Vitis core development kit supports host program debugging in all modes, but kernel debugging is only supported in the emulation flows with `gdb`. In addition, more hardware-centric debugging support, such as waveform analysis, is provided for the kernels.*

General Environment Setup

Running software or hardware emulation requires first the tool setup followed by the selection of the emulation mode. To set up the environment, see [Setting up the Vitis Integrated Design Environment](#).

Preparing the Host Code

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

```
g++ -g ...
```

Preparing the Kernel

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

```
v++ -g -t [sw_emu | hw_emu | hw] ...
```

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

- Checking whether the kernel makes out-of-bounds accesses to the interface buffers (`fsanitize=address`).
- Checking whether the kernel makes accesses to uninitialized local memory (`fsanitize=memory`).

These are compiler options enabled through a `v++` configuration file using the `advanced.param` syntax, and the `param:compiler.fsanitize` directive, as follows:

```
[advanced]
#param=<param_type>:<param_name>.<value>
param=compiler.fsanitize=address,memory
```

Then the config file is specified on the `v++` command line:

```
v++ -l -t sw_emu --config ./advanced.txt -o bin_kernel.xclbin
```

Refer to the [Vitis Compiler Command](#) for more information on the `--config` option.

When applied, the emulation run produces a debug log with emulation diagnostic messages that are written to `<project_dir>/Emulation-SW/<proj_name>-Default/<emulation_debug.log`.

Launching GDB Host Code Debug

You can launch GDB standalone to debug the host program if the code is built with debug information (built with the `-g` flag). This flow should also work while using a graphical front-end for GDB, such as the data display debugger (DDD) available from GNU. The following steps are the instructions for launching GDB.

1. Ensure that the environment variable `XCL_EMULATION_MODE` is set to the correct mode.
2. The application debug feature must be enabled at runtime using an attribute in the `xrt.ini` file. Create an `xrt.ini` file in the same directory as your host executable, and include the following lines:

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

This informs the runtime library that the kernel is debug enabled. Refer to [xrt.ini File](#) for more information.

3. Start `gdb` through the Xilinx wrapper:

```
xgdb --args host.exe test.xclbin
```

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

- Launches GDB on the host program:

```
gdb --args host.exe test.xclbin
```

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

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

Launching Host and Kernel Debug

In software emulation, to better mimic the hardware being emulated, kernels are spawned off as separate processes. If you are using GDB to debug the host code, breakpoints set on kernel lines are not hit because the kernel code is not run within that process. To support the concurrent debugging of the host code and the kernel code, Vitis technology provides a mechanism to attach to spawned kernels through the use of `xrt_server`.

1. You must start three different terminals in the command line flow. In the first terminal, start the `xrt_server` using the following command:

```
${XILINX_VITIS}/bin/xrt_server --sdx-url
```

2. In a second terminal, run the host code in `xgdb` as described in [Launching GDB Host Code Debug](#).

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



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

3. In a third terminal, run the `xgdb` command, and at the GDB prompt, run the following commands:

- For software emulation:

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

- For hardware emulation:

1. Locate the `xrt_server` temporary directory: `/tmp/sdx/$uid`.
2. Find the `xrt_server` process id (PID) containing the DWARF file of this debug session.
3. At the `gdb` command line, run: `file /tmp/sdx/$uid/$pid/NUM.DWARF`.

- In either case, connect to the kernel process:

```
target remote :NUM
```

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



TIP: When debugging software/hardware emulation kernels in the Vitis IDE, these steps are handled automatically and the kernel process is automatically attached, providing multiple contexts to debug both the host code and kernel code simultaneously.

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

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

GDB-Based Debugging

This section shows how host and kernel debugging can be performed with the help of GDB. Because this flow should be familiar to software developers, this section focuses on the extensions of host code debugging capabilities specifically for FPGAs, and the current status of kernel-based hardware emulation support.

Xilinx Runtime GDB Extensions

The Xilinx Runtime (XRT) debug environment introduces new GDB commands that give visibility from the host application into the XRT library.

Note: If you run GDB outside of the Vitis technology, these commands need to be enabled as described in [Launching GDB Host Code Debug](#).

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

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

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

- Commands that give visibility into the IP on the Vitis target platform. This functionality is only available in the system flow (hardware execution) and not in any of the emulation flows.

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

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

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

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

GDB Kernel-Based Debugging

GDB kernel debugging is supported for the software emulation and hardware emulation flows. When the GDB executable is connected to the kernel in the IDE or command line flows, you can set breakpoints and query the content of variables in the kernel, similar to normal host code debugging. This is fully supported in the software emulation flow because the kernel GDB processes attach to the spawned software processes.

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

Using printf() to Debug Kernels

The simplest approach to debugging algorithms is to verify key data values throughout the execution of the program. For application developers, printing checkpoint values in the code is a tried and trusted method of identifying issues within the execution of a program. Because part of the algorithm is now running on an FPGA, even this debugging technique requires additional support.

C/C++ Kernel

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

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

OpenCL Kernel

The Vitis IDE supports the OpenCL™ `printf()` built-in function within the kernels in all development flows: software emulation, hardware emulation, and running the kernel in actual hardware. The following is an example of using `printf()` in the kernel, and the output when the kernel is executed with `global` size of 8:

```
__kernel __attribute__((reqd_work_group_size(1, 1, 1)))
void hello_world(__global int *a)
{
    int idx = get_global_id(0);

    printf("Hello world from work item %d\n", idx);
    a[idx] = idx;
}
```

The output is as follows:

```
Hello world from work item 0
Hello world from work item 1
Hello world from work item 2
Hello world from work item 3
Hello world from work item 4
Hello world from work item 5
Hello world from work item 6
Hello world from work item 7
```



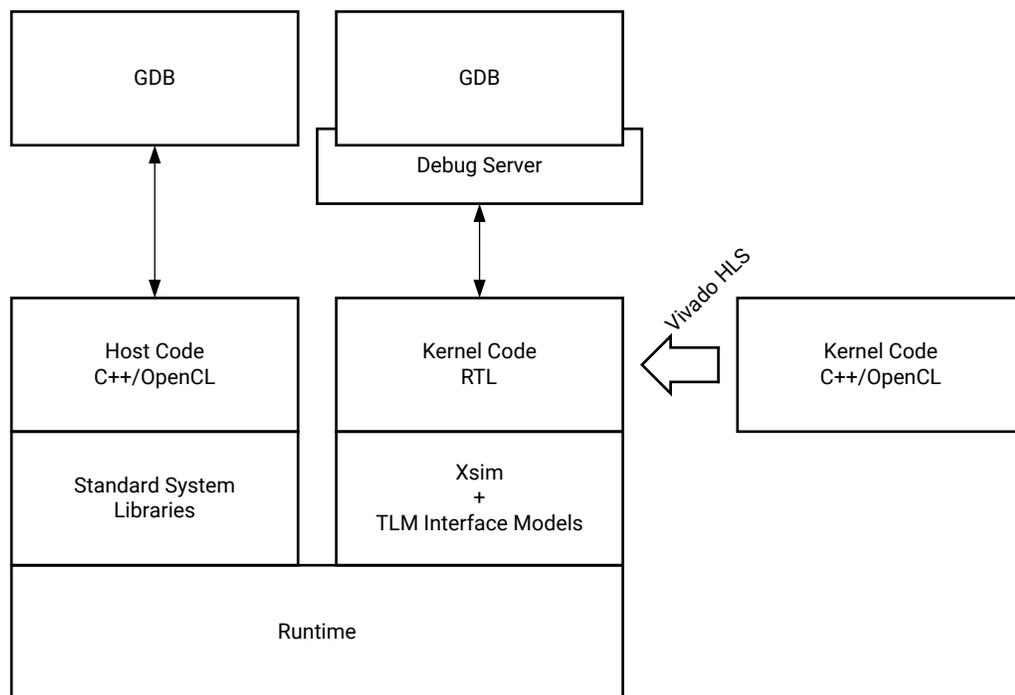
IMPORTANT! *`printf()` messages are buffered in the global memory and unloaded when kernel execution is completed. If `printf()` is used in multiple kernels, the order of the messages from each kernel display on the host terminal is not certain. Please note, especially when running in hardware emulation and hardware, the hardware buffer size might limit `printf` output capturing.*

Note: This feature is only supported for OpenCL kernels in all development flows.

Hardware Emulation

The following figure shows the hardware emulation flow diagram which can be used to validate the host code, profile host and kernel performance, give estimated FPGA resource usage, and verify the kernel using an accurate model of the hardware (RTL). The GDB can be used to debug both the host and kernel code. During hardware emulation the host code is executed concurrently with a simulation of the RTL model of the kernel, directly imported, or created through Vivado HLS from the C/C++/OpenCL kernel code. For more information, see [Hardware Emulation](#).

Figure 87: Hardware Emulation



X21159-111418

Verify the host code and the kernel hardware implementation is correct by running hardware emulation on a data set. Hardware emulation performs detailed verification using an accurate model of the hardware (RTL) together with the host code C/OpenCL model. The hardware emulation flow invokes the Vivado logic simulator in the Vitis core development kit to test the kernel logic that is to be executed on the FPGA compute fabric. The interface between the models is represented by a transaction-level model (TLM) to limit impact of interface model on the overall execution time. The execution time for hardware emulation is longer than software emulation.



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

During the hardware emulation stage, you can optionally modify the kernel code to improve performance. Iterate in hardware emulation until the functionality is correct and the estimated kernel performance is sufficient.

Debugging in Hardware Emulation

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

GDB-Based Debugging

Debugging using a software-based GDB flow is fully supported during hardware emulation. Except for the execution of the actual RTL code representing the kernel code, there is no difference to you because GDB maps the RTL back into the source code description. This limits the breakpoints and observability of the variables in some cases, because during the RTL generation by Vivado HLS, some variables and loops from the kernel source might have been dissolved.

Waveform-Based Kernel Debugging

The C/C++ and OpenCL kernel code is synthesized using Vivado HLS to transform it into a Hardware Description Language (HDL) and later implement it in the Vivado Design Suite to produce the FPGA binary (`xclbin`).

Another debugging approach is based on simulation waveforms. Hardware-centric designers are likely to be familiar with this approach. This waveform-based HDL debugging is supported by the Vitis core development kit using both the command line flow, or through the IDE flow during hardware emulation.



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

Enable Waveform Debugging with the Vitis Compiler Command

The waveform debugging process can be enabled through the `v++` command using the following steps:

1. Turn on debug code generation during kernel compilation.

```
v++ -g . . .
```

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

```
[Emulation]
launch_waveform=batch

[Debug]
profile=true
timeline_trace=true
data_transfer_trace=fine
```

3. Execute hardware emulation. The hardware transaction data is collected in the file named `<hardware_platform>-<device_id>-<xclbin_name>.wdb`. This file can directly be opened through the Vitis IDE.



TIP: If `launch_waveform` is set to `gui` in the `xrt.ini`, a live waveform viewer is spawned during the execution of the hardware emulation.

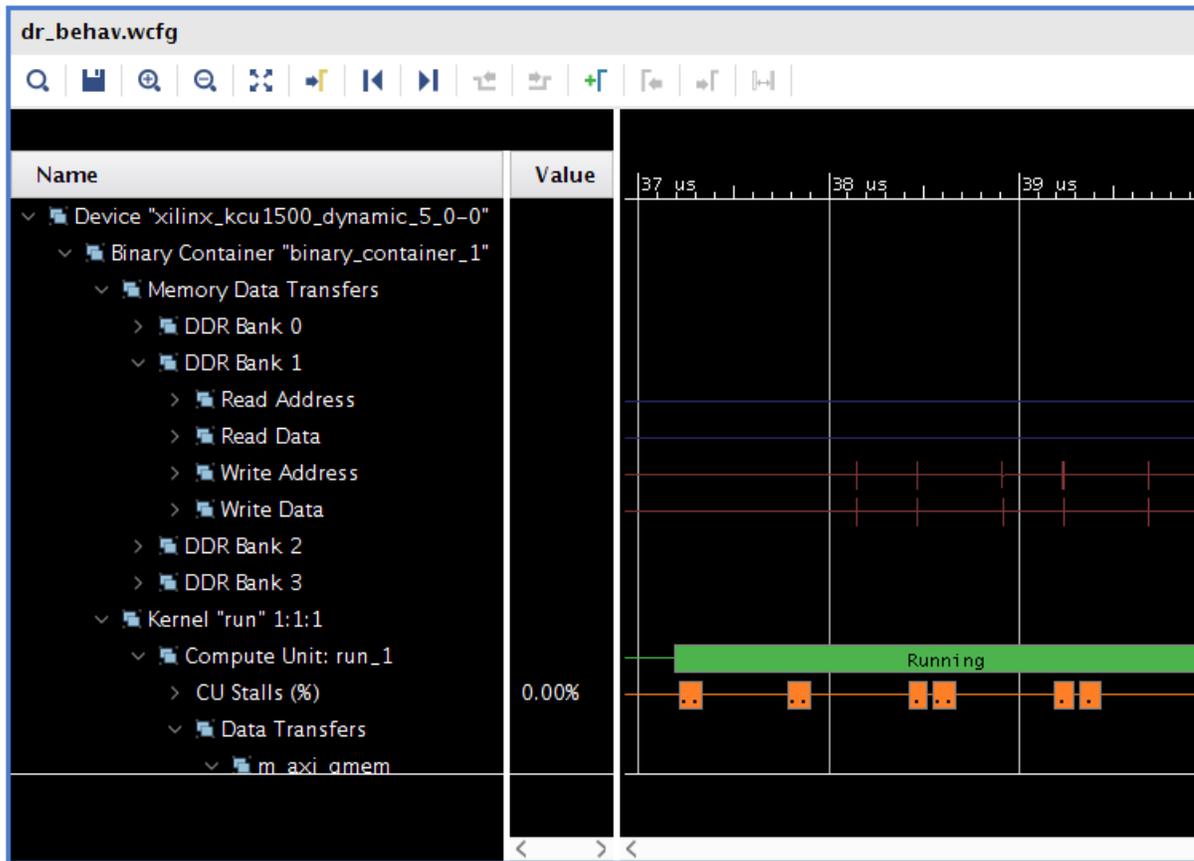
Run the Waveform-Based Kernel Debugging Flow

The Vitis IDE provides waveform-based HDL debugging in the hardware emulation mode. The waveform is opened in the Vivado waveform viewer which should be familiar to Vivado logic simulation users. The Vitis IDE lets you display kernel interfaces, internal signals, and includes debug controls such as restart, HDL breakpoints, as well as HDL code lookup and waveform markers. In addition, it provides top-level DDR data transfers (per bank) along with kernel-specific details including compute unit stalls, loop pipeline activity, and data transfers.

For details, see [Waveform View and Live Waveform Viewer](#).

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

Figure 88: Waveform Viewer



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



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

- **Kernel <kernel name><workgroup size> Compute Unit<CU name>:** Kernel name, workgroup size, and compute unit name.
- **CU Stalls (%):** This shows a summary of stalls for the entire CU. A bus of all lowest-level stall signals is created, and the bus is represented in the waveform as a percentage (%) of those signals that are active at any point in time.
- **Data Transfers:** This shows the data transfers for all AXI masters on the CU.
- **User Functions:** This lists all of the functions within the hierarchy of the CU.
- **Function:** <function name>

- **Dataflow/Pipeline Activity:** This shows the function-level loop dataflow/pipeline signals for a CU.
- **Function Stalls:** This lists the three stall signals within this function.
- **Function I/O:** This lists the I/O for the function. These I/O are of protocol `-m_axi`, `ap_fifo`, `ap_memory`, or `ap_none`.

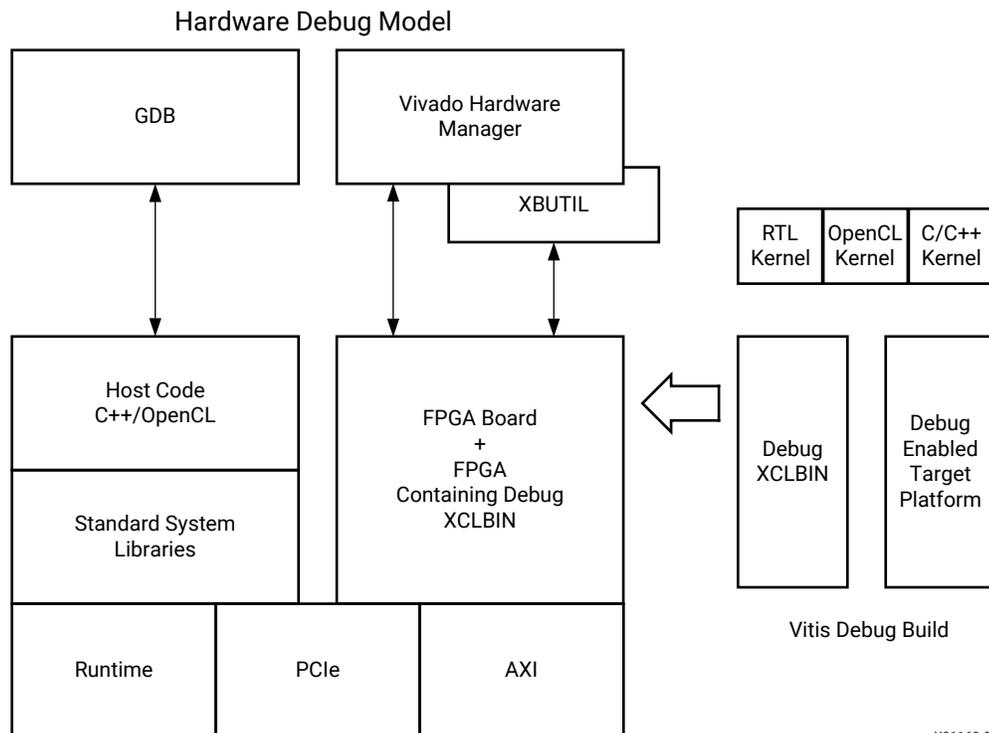


TIP: As with any waveform debugger, additional debug data of internal signals can be added by selecting the instance of interest from the scope menu and the signals of interest from the object menu. Similarly, debug controls such as HDL breakpoints, as well as HDL code lookup and waveform markers are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

Hardware Execution

During hardware execution, the actual hardware platform is used to execute the kernels; however, hardware debugging requires additional logic to be incorporated into the overall hardware model, and will impact FPGA resources and performance. The difference between the debug and final configurations of the kernel is the inclusion of special hardware logic in the platform, such as ILA and VIO debug cores and AXI performance monitors for debug purposes. This additional logic should be removed in the final compilation for production designs.

Figure 89: Hardware Execution



X21160-092519

The Vitis core development kit provides specific hardware debug capabilities that include ChipScope™ debug cores (such as System ILAs), which can be viewed in Vivado hardware manager, with waveform analysis, kernel activity reports, and memory access analysis to localize these critical hardware issues.



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

System Verification and Hardware Debug

Application Hangs

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

If the process of hardware debugging does not resolve the problem, it is necessary to perform hardware debugging using the ChipScope™ feature.

AXI Firewall Trips

The AXI firewall should prevent host hangs. This is why Xilinx recommends the AXI Protocol Firewall IP to be included in Vitis target platforms. When the firewall trips, one of the first checks to perform is confirming if the host code and kernels are set up to use the same memory banks. The following steps detail how to perform this check.

1. Use `xbutil` to program the FPGA:

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



TIP: Refer to [Xilinx Board Utility](#) for more information on `xbutil`.

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

```
xbutil query
```

In the following example, there are no kernels associated with memory banks:

```

#####
Mem Topology
Tag          Type          Temp          Size          Device Memory Usage
[0] bank0    MEM_DDR4      Not Supp      16 GB         0 Byte         0
[1] bank1    MEM_DDR4      Not Supp      16 GB         0 Byte         0
[2] bank2    **UNUSED**    Not Supp      16 GB         0 Byte         0
[3] bank3    **UNUSED**    Not Supp      16 GB         0 Byte         0
[4] PLRAM[0] MEM_DRAM      Not Supp      128 KB        0 Byte         0
[5] PLRAM[1] **UNUSED**    Not Supp      128 KB        0 Byte         0
[6] PLRAM[2] **UNUSED**    Not Supp      128 KB        0 Byte         0

Total DMA Transfer Metrics:
Chan[0].h2c: 416 MB
Chan[0].c2h: 328 MB
Chan[1].h2c: 96 MB
Chan[1].c2h: 184 MB
    
```

3. If the host code expects any DDR banks/PLRAMs to be used, this report should indicate an issue. In this case, it is necessary to check kernel and host code expectations. If the host code is using the Xilinx OpenCL extensions, it is necessary to check which DDR banks should be used by the kernel. These should match the `connectivity.sp` options specified as discussed in [Mapping Kernel Ports to Global Memory](#).

Kernel Hangs Due to AXI Violations

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

1. The Vitis core development kit provides two options for instrumentation to be applied during `v++` linking (`-l`). Both of these options add hardware to your implementation, and based on resource utilization it might be necessary to limit instrumentation.
 - a. Add Lightweight AXI Protocol Checkers (`lapc`). These protocol checkers are added using the `--dk` option. The following syntax is used:

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

In general, the `<interface_name>` is optional. If not specified, all ports are expected to be analyzed. The `protocol` option is used to define the protocol checkers to be inserted. This option can accept a special keyword, `all`, for `<compute_unit_name>` and/or `<interface_name>`. The `list_ports` option generates a list of valid compute units and port combinations in the current design.

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

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

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

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

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

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

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

2. When the application is rebuilt, rerun the host application using the `xclbin` with the added SPM IP and LAPC IP.
3. When the application hangs, you can use `xbutil status` to check for any errors or anomalies.
4. Check the SPM output:
 - Run `xbutil status --spm` a couple of times to check if any counters are moving. If they are moving then the kernels are active.



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

- If the counters are stagnant, the outstanding counts greater than zero might mean some AXI transactions are hung.

5. Check the LAPC output:

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



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

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

Host Application Hangs When Accessing Memory

Application hangs can also be caused by incomplete DMA transfers initiated from the host code. This does not necessarily mean that the host code is wrong; it might also be that the kernels have issued illegal transactions and locked up the AXI.

1. If the platform has an AXI firewall, such as in the Vitis target platforms, it is likely to trip. The driver issues a `SIGBUS` error, kills the application, and resets the device. You can check this by running `xbutil query`. The following figure shows such an error in the firewall status:

```

Firewall Last Error Status:
 0:      0x0 (GOOD)
 1:      0x0 (GOOD)
 2:      0x80000 (RECS_WRITE_TO_BVALID_MAX_WAIT). Error occurred on Tue 2017-12-19 11:39:13 PST
Xclbin ID: 0x5a39da87
    
```



TIP: If the firewall has not tripped, the Linux tool, `dmesg`, can provide additional insight.

2. When you know that the firewall has tripped, it is important to determine the cause of the DMA timeout. The issue could be an illegal DMA transfer, or kernel misbehavior. However, a side effect of the AXI firewall tripping is that the health check functionality in the driver resets the board after killing the application; any information on the device that might help with debugging the root cause is lost. To debug this issue, disable the health check thread in the `xclmgmt` kernel module to capture the error. This uses common Unix kernel tools in the following sequence:
 - a. `sudo modinfo xclmgmt`: This command lists the current configuration of the module and indicates if the `health_check` parameter is ON or OFF. It also returns the path to the `xclmgmt` module.
 - b. `sudo rmmod xclmgmt`: This removes and disables the `xclmgmt` kernel module.
 - c. `sudo insmod <path to module>/xclmgmt.ko health_check=0`: This re-installs the `xclmgmt` kernel module with the health check disabled.



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

3. With the health check disabled, rerun the application. You can use the kernel instrumentation to isolate this issue as previously described.

Typical Errors Leading to Application Hangs

The user errors that typically create application hangs are listed below:

- Read-before-write in 5.0+ target platforms causes a Memory Interface Generator error correction code (MIG ECC) error. This is typically a user error. For example, this error might occur when a kernel is expected to write 4 KB of data in DDR, but it produces only 1 KB of data, and then try to transfer the full 4 KB of data to the host. It can also happen if you supply a 1 KB buffer to a kernel, but the kernel tries to read 4 KB of data.

- An ECC read-before-write error also occurs if no data has been written to a memory location as the last bitstream download which results in MIG initialization, but a read request is made for that same memory location. ECC errors stall the affected MIG because kernels are usually not able to handle this error. This can manifest in two different ways:
 1. The CU might hang or stall because it cannot handle this error while reading or writing to or from the affected MIG. The `xbutil` query shows that the CU is stuck in a `BUSY` state and is not making progress.
 2. The AXI Firewall might trip if a PCIe® DMA request is made to the affected MIG, because the DMA engine is unable to complete the request. AXI Firewall trips result in the Linux kernel driver killing all processes which have opened the device node with the `SIGBUS` signal. The `xbutil` query shows if an AXI Firewall has indeed tripped and includes a timestamp.

If the above hang does not occur, the host code might not read back the correct data. This incorrect data is typically 0s and is located in the last part of the data. It is important to review the host code carefully. One common example is compression, where the size of the compressed data is not known up front, and an application might try to migrate more data to the host than was produced by the kernel.

Defensive Programming

The Vitis IDE is capable of creating very efficient implementations. In some cases, however, implementation issues can occur. One such case is if a write request is emitted before there is enough data available in the process to complete the write transaction. This can cause deadlock conditions when multiple concurrent kernels are affected by this issue and the write request of a kernel depends on the input read being completed.

To avoid such situations, a conservative mode is available on the adapter. In principle, it delays the write request until it has all of the data necessary to complete the write. This mode is enabled during compilation by applying the following `--advanced.param` option to the `v++` compiler:

```
--advanced.param:compiler.axiDeadLockFree=yes
```

Because enabling this mode can impact performance, you might prefer to use this as a defensive programming technique where this option is inserted during development and testing and then removed during optimization. You might also want to add this option when the accelerator hangs repeatedly.

Debugging with ChipScope

You can use the ChipScope debugging environment and the Vivado hardware manager to help you debug your host application and kernels quickly and more effectively. To achieve this, at least one of the following must be true:

- Your Vitis application project has been instrumented with debug cores, using the `--dk` compiler switch (as described in [Hardware Debugging Using ChipScope](#)).

- The RTL kernels used in your project must have been instantiated with debug cores (as described in [Adding Debug IP to RTL Kernels](#)).

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

Note: Debugging on the kernel platform requires additional logic to be incorporated into the overall hardware model, which might have an impact on resource use and kernel performance.

Running XVC and HW Servers

The following steps are required to run the Xilinx virtual cable (XVC) and HW servers, host applications, and finally trigger and arm the debug cores in Vivado hardware manager.

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

Adding Debug IP to RTL Kernels



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

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



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

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

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

The following is an example of an ILA debug core instantiated into the RTL kernel source file of the [RTL Kernel Debug](#) example design on GitHub. The ILA monitors the output of the combinatorial adder as specified in the `src/hdl/krn1_vadd_rtl_int.sv` file.

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

After the RTL kernel has been instrumented for debug with the appropriate debug cores, you can analyze the hardware in the Vivado hardware manager features as described in the previous topic.

Debugging through the Host Application

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

Pausing a C++ Host Application

The following code example is from the `src/host.cpp` code from the [RTL Kernel](#) example on GitHub:

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

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

```

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

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

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

        //Set the Kernel Arguments
        ...

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

The addition of the conditional `if (interactive)` test and the use of the `wait_for_enter` function pause the host application to give the ILA time to set up the required triggers and prepare to capture data from the kernel. After the Vivado hardware manager is set up and configured properly, press `Enter` to continue running the host application.

Pausing the Host Application Using GDB

Instead of making changes to the host application to pause before a kernel execution, you can run a GDB session from the Vitis IDE. You can then set a breakpoint prior to the kernel execution in the host application. When the breakpoint is reached, you can set up the debug ILA triggers in Vivado hardware manager, arm the trigger, and then resume the kernel execution in GDB.

Automated Setup for Hardware Debug

Note: A full Vitis core development kit install is required to complete the following task. See the [Vitis 2019.2 Software Platform Release Notes](#) for more information about installation.

1. Set up your Vitis core development kit as described in [Setting up the Vitis Integrated Design Environment](#).
2. Start `xvc_pcie` and `hw_server` apps using the `debug_hw` script.

```

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

Note: The `/dev/xvc_*` character device will differ depending on the platform. In this example, the character device is `/dev/xvc_pub.m1025`, though on your system it is likely to differ.

3. In the Vitis IDE, modify the host code to include a pause statement *after* the kernel has been created/downloaded and *before* the kernel execution is started, then recompile the host program.
 - For C++ host code, add a pause after the creation of the `cl::Kernel` object. The following snippet is from the Vector Add template design C++ host code:

```

134 // This call will get the kernel object from program. A kernel is an
135 // OpenCL function that is executed on the FPGA.
136 cl::Kernel krnl_vector_add(program,"krnl_vadd");
137
138 // Add a pause here to prompt user to arm ILA trigger
139 std::cout << "Pausing to allow you to arm ILA trigger. Hit enter here to resume host program.." << std::endl;
140 std::cin.get();
141
142 // These commands will allocate memory on the Device. The cl::Buffer objects can
143 // be used to reference the memory locations on the device.
144 cl::Buffer buffer_a(context, CL_MEM_USE_HOST_PTR | CL_MEM_READ_ONLY,
145 size_in_bytes, source_a.data());
146 cl::Buffer buffer_b(context, CL_MEM_USE_HOST_PTR | CL_MEM_READ_ONLY,
147 size_in_bytes, source_b.data());
148 cl::Buffer buffer_result(context, CL_MEM_USE_HOST_PTR | CL_MEM_WRITE_ONLY,
149 size_in_bytes, source_results.data());
150
151
    
```

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

```

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

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

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

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

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

4. Run your modified host program.

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

5. Launch Vivado Design Suite using the debug_hw script located in \$XILINX_VITIS/bin.

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

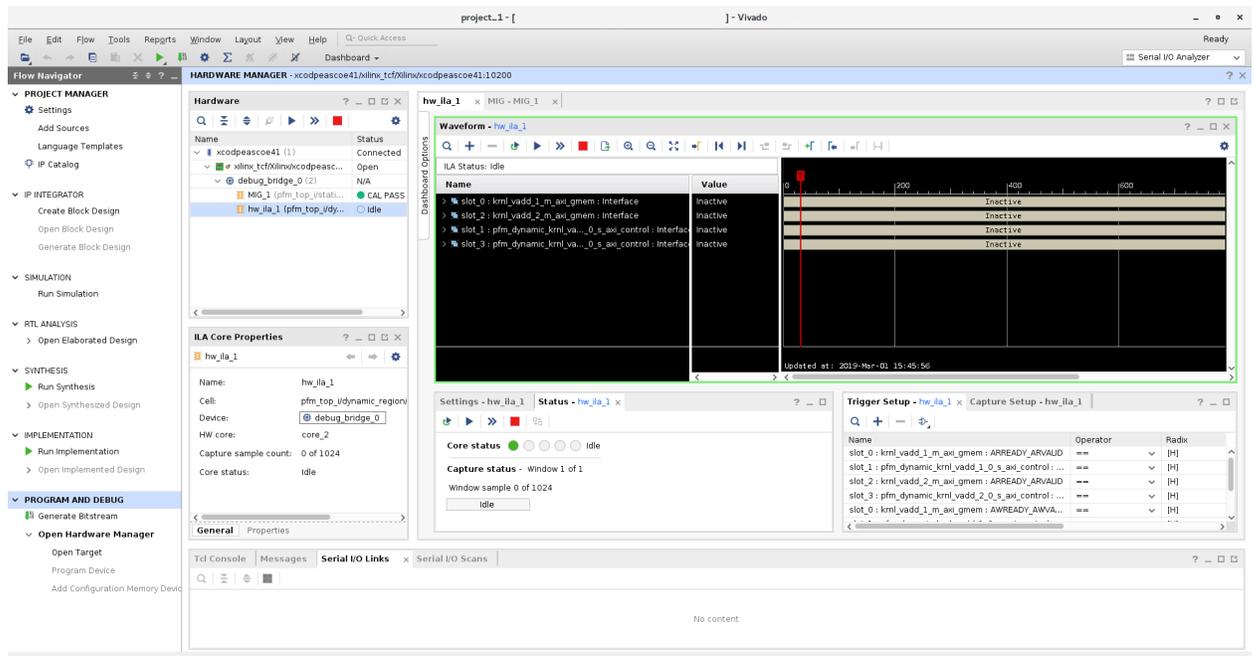
The command window displays the following example:

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

***** Vivado v2019.2 (64-bit)
***** SW Build 2245749 on Date Time
***** IP Build 2245576 on Date Time
** Copyright 1986-2019 Xilinx, Inc. All Rights Reserved.

start_gui
```

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

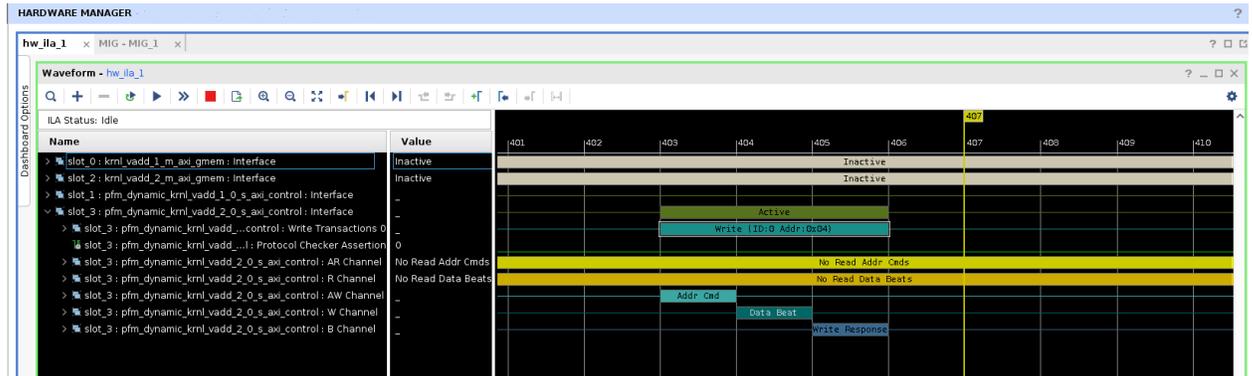


7. Press Enter to un-pause the host program.

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

TEST PASSED
```

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



Manual Setup for Hardware Debug

There are two methods to set up your hardware debug servers based on your platform.

Manually Starting Debug Servers

Note: The following steps are also applicable when using Nimble and other cloud platforms.

There are two steps required to start the debug servers prior to debugging the design in Vivado hardware manager.

- Source the Vitis core development kit setup script, `settings64.csh` or `settings64.sh`, and launch the `xvc_pcie` server. The filename passed to `xvc_pcie` must match the character driver file installed with the kernel device driver.

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

Note: The `xvc_pcie` server has many useful command line options. You can issue `xvc_pcie -help` to obtain the full list of available options.

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

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

Starting Debug Servers on an Amazon F1 Instance

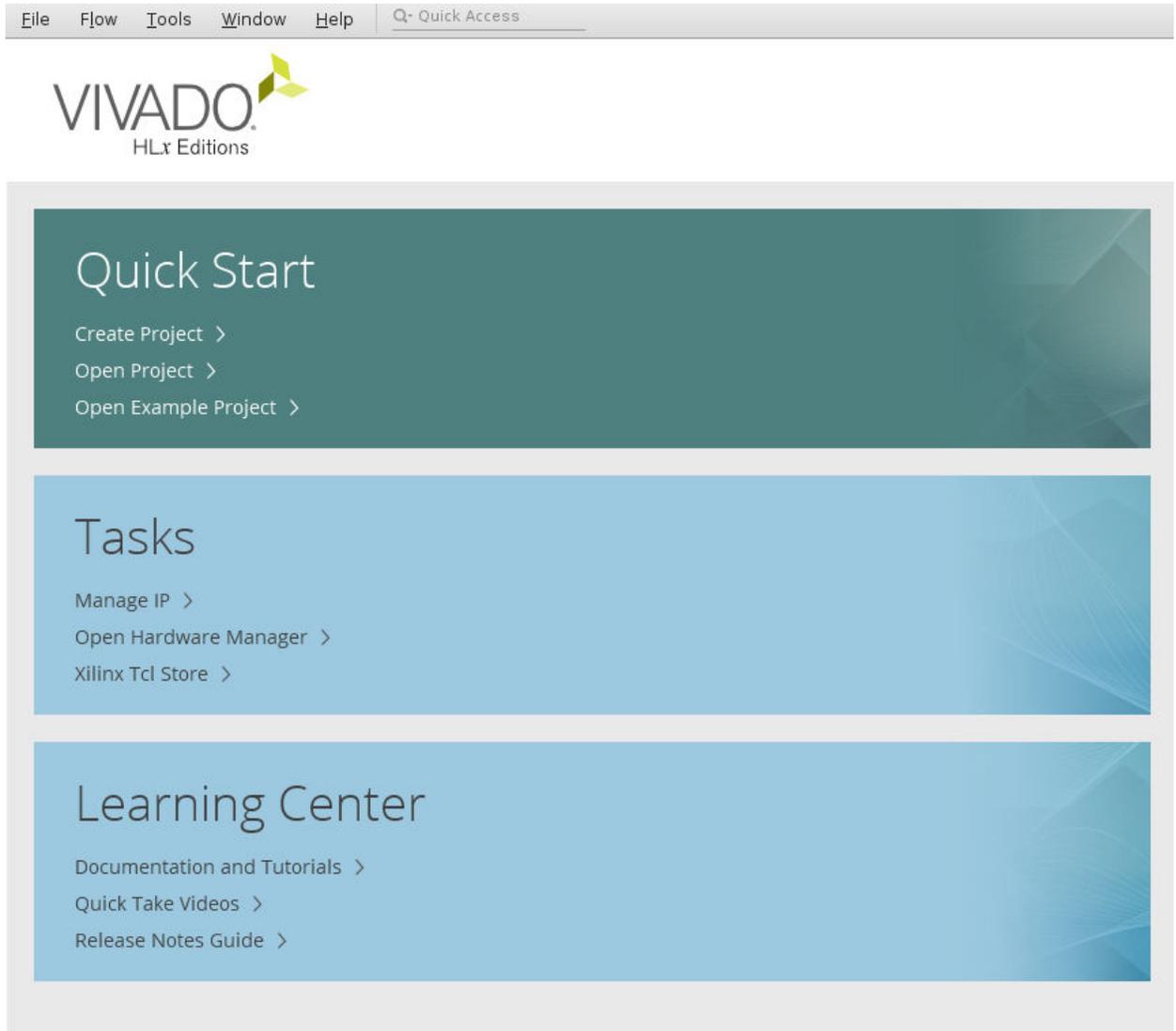
Instructions to start the debug servers on an Amazon F1 instance can be found here: https://github.com/aws/aws-fpga/blob/master/hdk/docs/Virtual_JTAG_XVC.md

Debugging Designs Using Vivado Hardware Manager

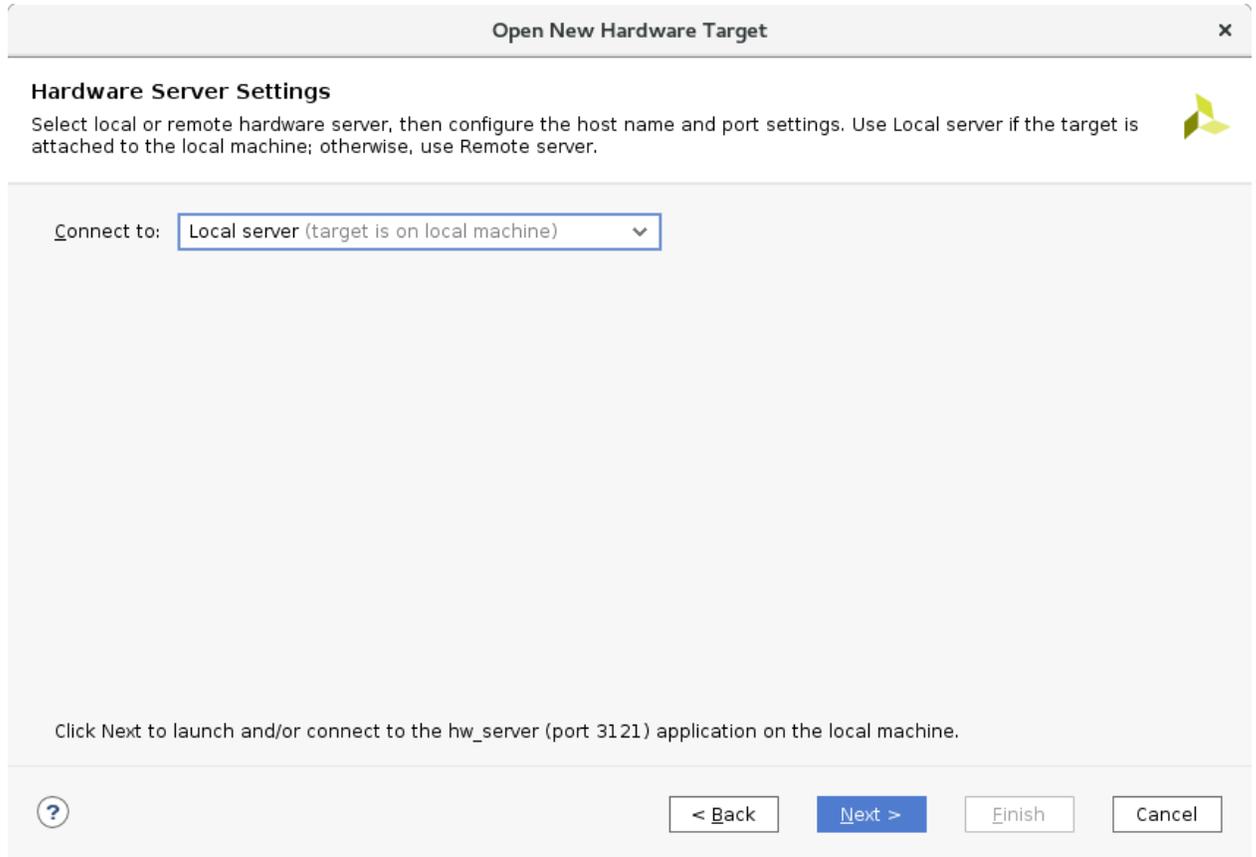
Traditionally, a physical JTAG connection is used to debug FPGAs. The Vitis unified software platforms have leveraged Xilinx Virtual Cable (XVC) for a debug flow that enables debug in the cloud. To take advantage of this capability, Vitis enables running the XVC server. The XVC server is an implementation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

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

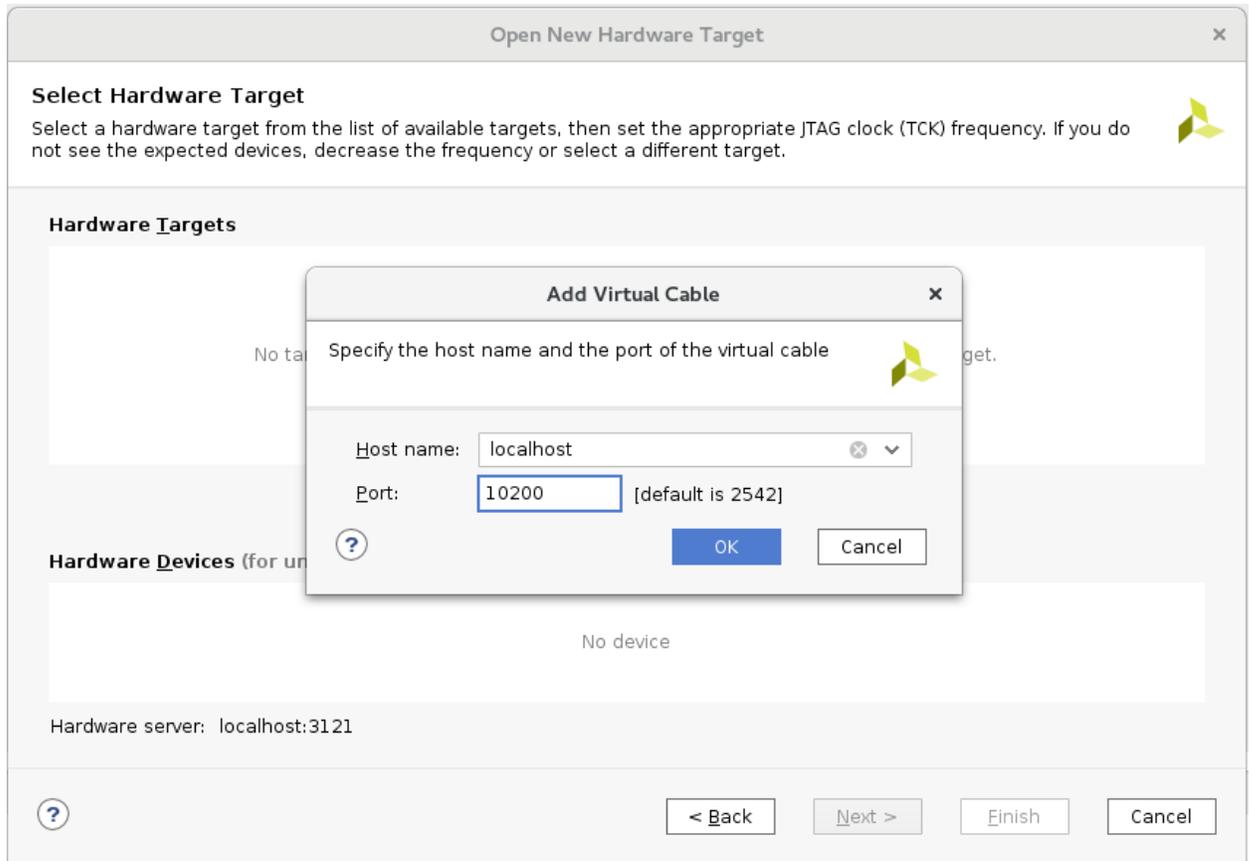
1. Launch the Vivado Lab Edition, or the full Vivado Design Suite.
2. Select **Open Hardware Manager** from the Tasks menu, as shown in the following figure.



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



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

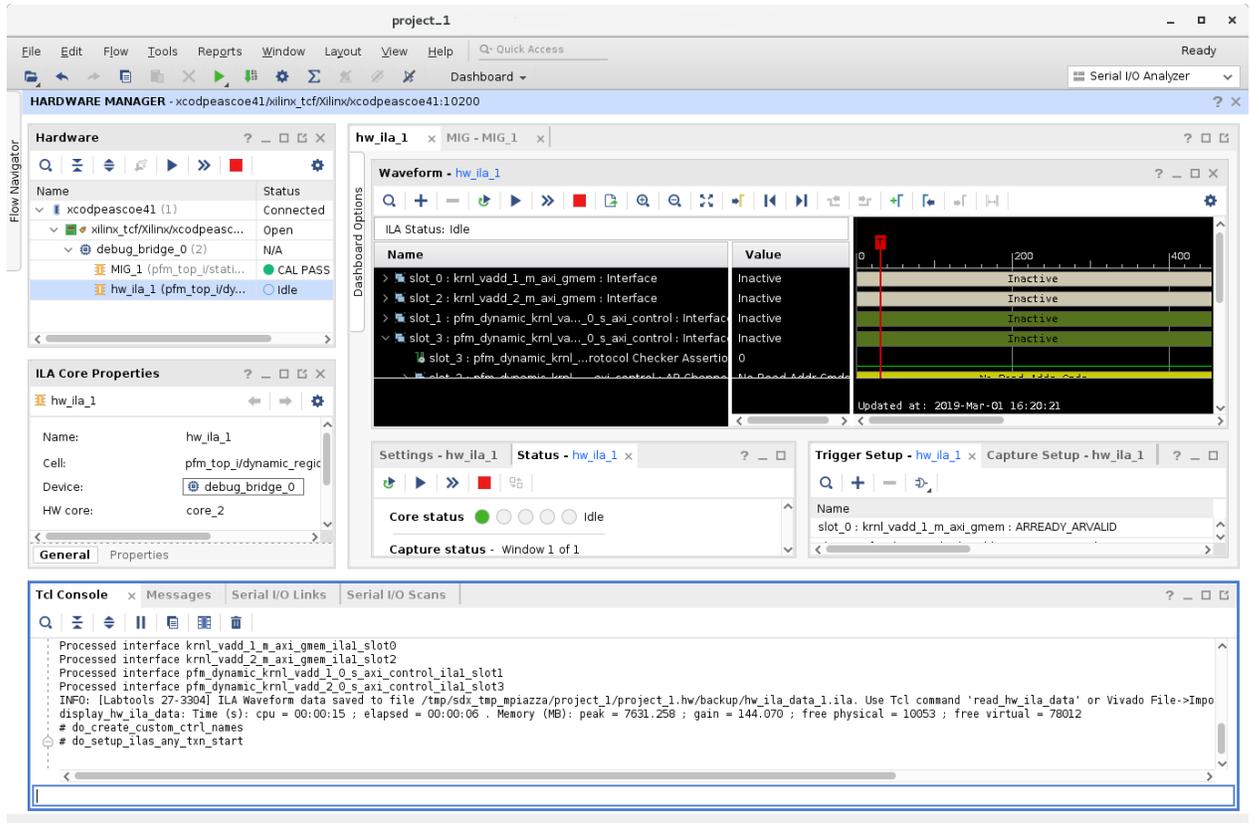


5. Select the debug bridge instance from the Hardware window of the Vivado hardware manager.
6. In the Hardware Device Properties window select the appropriate probes file for your design by clicking the icon next to the Probes file entry, selecting the file, and clicking **OK**. This refreshes the hardware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



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

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



Hardware Debugging Using ChipScope

After the final system image (xclbin) is generated and executed on the Vitis target platform, the entire system including the host application running on the CPU, and the Vitis accelerated kernels on the Xilinx FPGA, can be confirmed to be executing correctly on the actual hardware. At this stage, you can validate the functioning of the host code and kernel in the target hardware and debug any issues found. Some of the conditions that can be reviewed or analyzed are listed as follows:

- System hangs that could be due to protocol violations:
 - These violations can take down the entire system.
 - These violations can cause the kernel to get invalid data or to hang.
 - It is hard to determine where or when these violations originated.
 - To debug this condition, you should use an ILA triggered off of the AXI protocol checker, which needs to be configured on the Vitis target platform in use.
- Issues inside the RTL kernel:
 - These problems are sometimes caused by the implementation: timing issues, race condition, and bad design constraint.

- Functional bugs that hardware emulation did not show.
- Performance issues:
 - For example, the frames per second processing is not what you expect.
 - You can examine data beats and pipelining.
 - Using an ILA with trigger sequencer, you can examine the burst size, pipelining, and data width to locate the bottleneck.

ILA

The Vitis core development kit provides insertion of the System Integrated Logic Analyzer (ILA) into a design to capture and view AXI transaction level activity by probing the signals between kernel interfaces and global memory. The ILA provides custom event triggering on one or more signals to allow waveform capture at system speeds. The waveforms can be analyzed in a viewer and used to debug hardware, finding protocol violations or performance issues for example, and can be crucial for debugging difficult situation like application hangs.

Captured data can be accessed through the Xilinx virtual cable (XVC) using the Vivado tools. See the *Vivado Design Suite User Guide: Programming and Debugging* (UG908) for complete details.

Note: ILA debug cores require system resources, including logic and local memory to capture and store the signal data. Therefore they provide excellent visibility into your kernel, but they can affect both performance and resource utilization.

System ILAs can be inserted into the design using the `v++ --dk` option as shown below:

```
$ v++ --dk chipscope:<compute_unit_name>:<interface_name>
```

Refer to the [Vitis Compiler Command](#) for more information.

Checking the FPGA Board for Hardware Debug Support

Supporting hardware debugging requires the platform to support several IP components, most notably the Debug Bridge. Talk to your platform designer to determine if these components are included in the target platform. If a Xilinx platform is used, debug availability can be verified using the `platforminfo` utility to query the platform. Debug capabilities are listed under the `chipscope_debug` objects.

For example, to query the a platform for hardware debug support, the following `platforminfo` command can be used. A response can be seen showing that the platform contains a user and management debug network, and also supports debugging a MicroBlaze™ processor.

```
$ platforminfo --json="hardwarePlatform.extensions.chipscope_debug" --
platform xilinx_u200_xdma_201830_1
{
  "debug_networks": {
    "user": {
      "name": "User Debug Network",

```

```

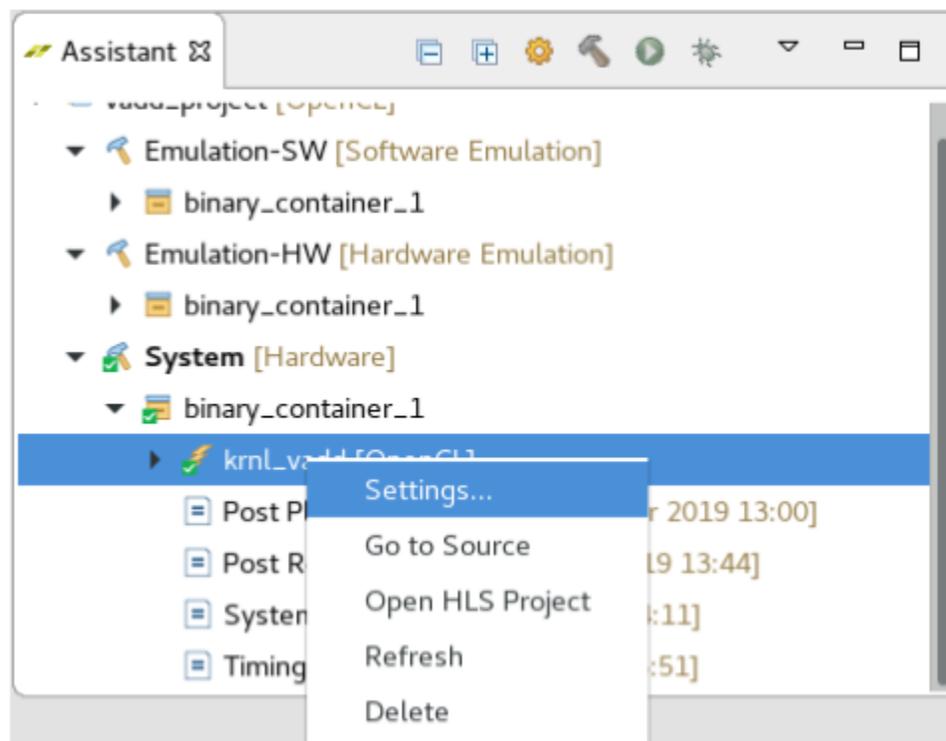
        "pcie_pf": "1",
        "bar_number": "0",
        "axi_baseaddr": "0x000C0000",
        "supports_jtag_fallback": "false",
        "supports_microblaze_debug": "true",
        "is_user_visible": "true"
    },
    "mgmt": {
        "name": "Management Debug Network",
        "pcie_pf": "0",
        "bar_number": "0",
        "axi_baseaddr": "0x001C0000",
        "supports_jtag_fallback": "true",
        "supports_microblaze_debug": "true",
        "is_user_visible": "false"
    }
}
}
}

```

Enabling ChipScope from the Vitis IDE

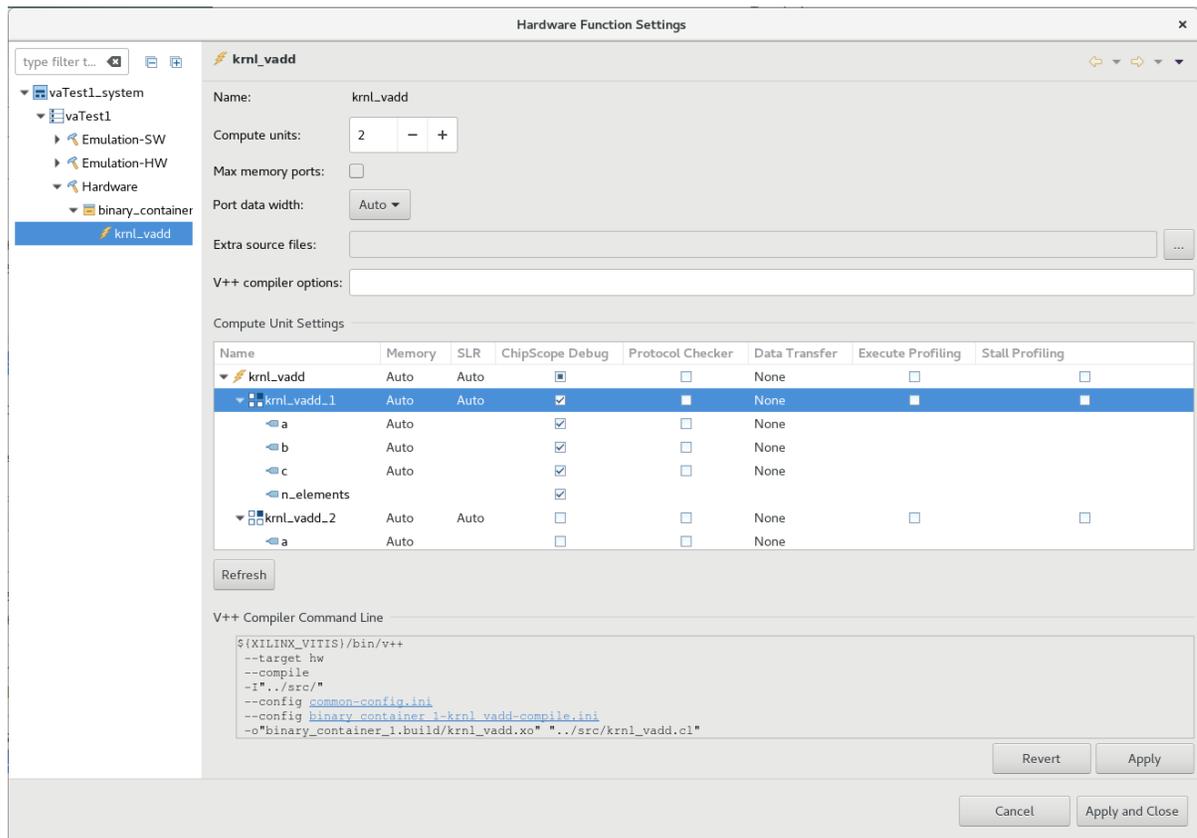
The Vitis IDE provides options to enable the ChipScope debug feature on all the interface ports of the compute units in the design. When enabling this option on a compute unit, the Vitis compiler adds a System ILA debug core to monitor the interface ports of the compute unit. This ensures that you can debug the interface signals on the Vitis target platform hardware while the kernel is running. You can access this through the Settings command by right-clicking on a kernel in the system build configuration in the Assistant window as shown below.

Figure 90: Vitis Assistant View



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

Figure 91: Vitis Hardware Function Settings



TIP: If you enable the **ChipScope Debug** option on larger designs with multiple kernels and/or compute units can result in overuse of the FPGA resources. Xilinx recommends using the `v++ --dk list_ports` option on the command line to determine the number and type of interfaces on the compute units. If you know which ports need to be monitored for debug as the design runs in hardware, the recommended methodology is to use the `--dk` option documented in the following topic.

Command Line Flow

The following section covers the `v++` linker options that can be used to list the available kernel ports, as well as enable the ILA core on the selected ports.

The ILA core provides transaction-level visibility into an instance of a compute unit (CU) running on hardware. AXI traffic of interest can also be captured and viewed using the ILA core. The ILA core can be added to an existing RTL kernel to enable debugging features within that design, or it can be inserted automatically by the `v++` compiler during the linking stage. The `v++` command provides the `--dk` option to attach System ILA cores at the interfaces to the kernels for debugging and performance monitoring purposes.

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

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

In general, the `<interface_name>` is optional. If not specified, all ports are expected to be analyzed. The `chipscope` option requires the explicit name of the compute unit to be provided for the `<compute_unit_name>` and `<interface_name>`. The `list_ports` option generates a list of valid compute units and port combinations in the current design and must be used after the kernel has been compiled.

First, you must compile the kernel source files into an `.xo` file:

```
v++ -c -k <kernel_name> --platform <platform> -o <kernel_xo_file>.xo
<kernel_source_files>
```

After the kernel has been compiled into an `.xo` file, `--dk list_ports` can be added to the command line options used during the `v++` linking process. This causes the `v++` compiler to print the list of valid compute units and port combinations for the kernel. See the following example:

```
v++ -l --platform <platform> --nk
<kernel_name>:<compute_units>:<kernel_nameN> --dk list_ports
<kernel_xo_file>.xo
```

Finally, ChipScope debug can be enabled on the desired ports by replacing `list_ports` with the appropriate `--dk chipscope` command syntax:

```
v++ -l --platform <platform> --nk
<kernel_name>:<compute_units>:<kernel_nameN> --dk
chipscope:<compute_unit_name>:<interface_name> <kernel_xo_file>.xo
```

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

When the design is built, you can debug the design using the Vivado® hardware manager as described in [Debugging with ChipScope](#).

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.

Utilities for Hardware Debugging

In some cases, the normal Vitis IDE and command line debug features are limited in their ability to isolate an issue. This is especially true when the software or hardware appears not to make any progress (hangs). These kinds of system issues are best analyzed with the help of the utilities mentioned in this section.

Using the Linux `dmesg` Utility

Well-designed kernels and modules report issues through the kernel ring buffer. This is also true for Vitis technology modules that allow you to debug the interaction with the accelerator board on the lowest Linux level.

Note: This utility is intended for use in hardware debug only.



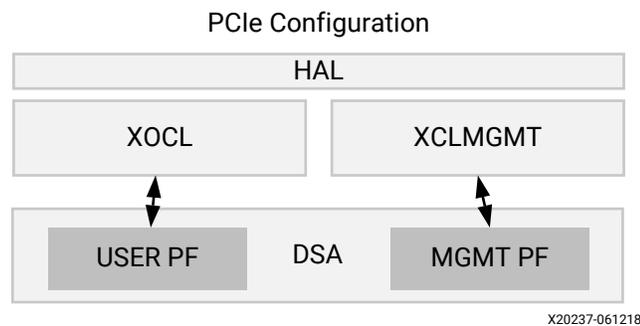
TIP: In most cases, it is sufficient to work with the less verbose `xbutil` feature to localize an issue. Refer to [Xilinx Board Utility](#) for more information on the `xbutil` command.

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

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

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

Figure 92: Software Platform Layers



X20237-061218

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

```
sudo dmesg -c
```

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

```
sudo dmesg
```

The `dmesg` utility prints a record shown in the following example:

```
[ 9902.316729] xclmgmt: AXI Firewall 2 has tripped. Status: 0x00000
[ 9902.316874] xclmgmt: xclmgmt_killall_processes
[ 9902.317007] xclmgmt: Killing pid: 19891
[ 9902.317501] xocl:xdma_xfer_submit: xfer 0xfffff8001c1be1018,268435456, s 0x1 timed out, ep 0x10000000.
[ 9902.317911] xocl:engine_reg_dump: 0-H2CO-MM: ioread32(0xffffc900064e0000) = 0x1fc00006 (id).
[ 9902.318410] xocl:engine_reg_dump: 0-H2CO-MM: ioread32(0xffffc900064e0040) = 0x00000001 (status).
[ 9902.318895] xocl:engine_reg_dump: 0-H2CO-MM: ioread32(0xffffc900064e0064) = 0x00f83e1f (control)
[ 9902.319370] xocl:engine_reg_dump: 0-H2CO-MM: ioread32(0xffffc900064e0080) = 0xa7a30000 (first_desc_lo)
[ 9902.319848] xocl:engine_reg_dump: 0-H2CO-MM: ioread32(0xffffc900064e0084) = 0x00000000 (first_desc_hi)
[ 9902.320336] xocl:engine_reg_dump: 0-H2CO-MM: ioread32(0xffffc900064e0088) = 0x0000000f (first_desc_adjacent).
[ 9902.320802] xocl:engine_reg_dump: 0-H2CO-MM: ioread32(0xffffc900064e0048) = 0x00000000 (completed_desc_count).
[ 9902.321279] xocl:engine_reg_dump: 0-H2CO-MM: ioread32(0xffffc900064e0090) = 0x00f83e1e (interrupt_enable_mask)
[ 9902.321759] xocl:engine_status_dump: SG engine 0-H2CO-MM status: 0x00000001: BUSY
[ 9902.322233] xocl:transfer_abort: abort transfer 0xfffff8001c1be1018, desc 240, engine desc queued 0.
[ 9902.322752] [drm:xdma_migrate_bo [xocl]] *ERROR* DMA failed to device addr 0x0, tld 19897, channel 0
[ 9902.323232] [drm:xdma_migrate_bo [xocl]] *ERROR* Dumping SG Page Table
```

In the example shown above, the AXI Firewall 2 has tripped, which is better examined using the `xbutil` utility.

Using the Xilinx `xbutil` Utility

The Xilinx board utility (`xbutil`) is a powerful standalone command line utility that can be used to debug lower level hardware/software interaction issues. A full description of this utility can be found in [Xilinx Board Utility](#).

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

- `query`: Provides an overall status of a card including information on the kernels in card memory.
- `program`: Downloads a binary (`xclbin`) to the programmable region of the Xilinx device.
- `status`: Extracts the status of the Performance Monitors (`spm`) and the Lightweight AXI Protocol Checkers (`lapc`).

Examples

This section presents examples to illustrate particular debug techniques and flows.

A Complete Debug Command Line Example

To get familiar with the command line flow in the Vitis core development kit, take the [IDCT example](#) available from the Xilinx GitHub repository and compile it manually (no makefile) for debugging.

1. In a terminal, set up your environment as described in [Setting up the Vitis Integrated Design Environment](#).
2. Clone the complete [Vitis Examples](#) GitHub repository to acquire all of the Vitis examples:

```
git clone https://github.com/Xilinx/Vitis_Examples.git
```

This creates a `Vitis_Examples` directory which includes the IDCT example.

3. CD to the IDCT example directory:

```
cd Vitis_Examples/vision/idct/
```

The host code is fully contained in `src/idct.cpp` and the kernel code is part of `src/krnل_idct.cpp`.

4. Compile and link the kernel software, targeting the software emulation build, `-t sw_emu`.

- a. Compile the kernel object file for debugging. The kernel is compiled using the `v++` compiler:

```
v++ -g -c -k krnl_idct -t sw_emu --platform <DEVICE> \
-o krnl_idct.xo src/krnl_idct.cpp
```

Where:

- `-g` indicates that the code is compiled for debugging.
 - `-c` indicates the compilation of the kernel.
 - `-k krnl_idct` specifies the kernel to compile.
 - `-t sw_emu` specifies the software emulation build.
 - `--platform <DEVICE>` indicates the accelerator platform for the build.
 - `-o krnl_idct.xo` specifies the output file name.
 - `src/krnl_idct.cpp` specifies the source file to compile.
- b. Link the kernel object file. Linking allows multiple kernels to be combined, and provides the means to specify implementation directives. The following is the link command for the IDCT example:

```
v++ -g -l -t sw_emu --platform <DEVICE> -config config.txt \
-o krnl_idct.xclbin krnl_idct.xo
```

The options for the `-l` line are similar to the `-c` line. In fact, the `-t` and `--platform` for both lines must match, as you are compiling and linking for the same build target, and the same platform.

The `-o` option specifies the name of the FPGA binary file.

The `--config` option specifies the configuration file, `config.txt`, that contains the directives for the build process as described in the [Vitis Compiler Configuration File](#). The contents of the configuration file are as follows:

```
[connectivity]
nk=krnl_idct:1

sp=krnl_idct_1.m_axi_gmem0:DDR[0]
sp=krnl_idct_1.m_axi_gmem1:DDR[0]
sp=krnl_idct_1.m_axi_gmem2:DDR[1]

[advanced]
prop=solution.hls_pre_tcl='src/hls_config.tcl'
```

The `connectivity.nk` option is used to specify multiple instances of a kernel. In this case, only one instance of `krnl_idct` is implemented in the final bitstream.

The `connectivity.sp` option is used to map kernel port bindings to specific DDR banks, HBM, and PLRAMs. For optimization purposes, it is a good idea to provide port binding for larger designs. Each port can be bound individually to a DDR/PLRAM. You are required to adhere to this same binding in the host code when allocating buffers.

The specified property, `solution.hls_pre_tcl`, is an example of how you can control the downstream tools such as Vivado HLS, and Vivado. See [Managing FPGA Synthesis and Implementation Results in the Vivado Tool](#) for more information.

5. Compile and link the host code for debugging. The host code is compiled with the GNU compiler chain, `g++`. Thus, separate compile and linking phases can also be performed.

Note: For embedded processors you will use the GNU Arm cross-compiler as described in [Compiling and Linking for Arm](#).

- a. Compile host code C++ files:

```
g++ -c -I${XILINX_XRT}/include -g -o idct.o src/idct.cpp
```

The `-c` option specifies a compile-only run, which creates an object file. The name of the object file is specified by the `-o` option. The `-I` option is using the runtime environment variable `XILINX_XRT` to specify the location of the common header files used by the host code. The `-g` option states that a debug compile is initiated. The final argument is the source file to be compiled in this step.

- b. Link the object files:

```
g++ -g -lOpenCL -lpthread -lrt -lstdc++ -L${XILINX_XRT}/lib/ -o idct idct.o
```

Linking is performed again using the `-g` option to ensure debug information is included. Because the example uses the OpenCL™ interfaces and the runtime library, several additional libraries are included in the link process (`-l`) which are picked up in addition to the default library path from the path specified by `-L` option. Finally, the name of the executable is specified by the `-o` option and the previously generated object file is provided through the last argument.

6. Prepare the emulation environment. The following command is required for emulation runs:

```
emconfigutil --platform <device>
```

The actual emulation mode (`sw_emu` or `hw_emu`) then needs to be set through the `XCL_EMULATION_MODE` environment variable. In C-shell this would be as follows:

```
setenv XCL_EMULATION_MODE sw_emu
```

7. Run the debugger on host and kernel. As stated in the earlier section, running the debugger is best performed in the Vitis technology. The following steps guide you through the command line debug process which requires three separate terminals, all prepared by sourcing the Vitis core development kit as described in the first section of this description.

- a. In the first terminal, start the Vitis debug server:

```
${XILINX_VITIS}/bin/xrt_server --sdx-url
```

- b. In a second terminal, set the emulation mode:

```
setenv XCL_EMULATION_MODE sw_emu
```

Create an `xrt.ini` file in the same directory as the host application, with the following content:

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

Run GDB by executing the following:

```
xgdb --args idct krnl_idct.xclbin
```

Enter the following on the `gdb` prompt:

```
run
```

- c. In the third terminal, attach the software emulation or hardware emulation model to GDB to allow stepping through the design. Here, there is a difference between running software emulation and hardware emulation. In either flow, start up another `xgdb`:

```
xgdb
```

- For software emulation:
 - Type the following on the `gdb` prompt:

```
file <XILINX_VITIS>/data/emulation/unified/cpu_em/generic_pcie/
model/genericpciemodel
```

Note: Because GDB does not expand the environment variable, it is easiest to replace `<XILINX_VITIS>` with the actual value of `$XILINX_VITIS`.

- For hardware emulation:
 1. Locate the `xrt_server` temporary directory: `/tmp/sdx/$uid`.
 2. Find the `xrt_server` process ID (PID) containing the DWARF file of this debug session.
 3. At the `gdb` prompt, run:

```
file /tmp/sdx/$uid/$pid/NUM.DWARF
```

- In either case, connect to the kernel process:

```
target remote :NUM
```

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

At this point, debugging of the `sw_emu` and `hw_emu` can be done as usual with GDB. The only difference is that the host code and the kernel code are debugged in two different GDB sessions. This is common when dealing with different processes. It is most important to understand that a breakpoint in one process might be hit before the next breakpoint in the current process is hit. In these cases, the debugging session appears to hang, while the second terminal is waiting for input.

Vitis Environment Reference Materials

The reference materials contained here include the following:

- **Vitis Compiler Command:** A description of the compiler options (`-c`), the linking options (`-l`), options common to both compile and linking, and a discussion of the `--config` options.
- The `xrt.ini` file is used to initialize XRT to produce reports, debug, and profiling data as it transacts business between the host and kernels. This file is used when the application is run, for emulation or hardware builds, and must be created manually when the build process is run from the command line.
- Vitis command line utilities include various Xilinx® utilities that provide detailed information about the platform resources, including SLR and memory resource availability, to help you construct the `v++` command line, and manage the build and run process.
 - **Platforminfo:** The `platforminfo` utility queries the platforms for which Vitis™ installation to use.
 - **Kernelinfo:** The `kernelinfo` utility prints the function definitions in the given Xilinx object file (O) file.
 - **Xclbinutil Utility:** The `xclbinutil` utility operates on a `xclbin` produced by Xilinx OpenCL™ Compiler.
 - **Emconfigutil :** The emulation configuration utility (`emconfigutil`) is used to automate the creation of the emulation configuration file.
 - **Xilinx Board Utility:** The Xilinx Board Utility (`xbutil`) is a command line tool used to perform various board installation, administration, and debug tasks.
- **package_xo Command:** The Tcl command used in the Vivado Design Suite to package an RTL IP into an `.xo` file, as described in [RTL Kernels](#).
- **HLS Pragmas:** A description of pragmas used by the Vivado HLS tool in synthesizing C/C++ kernels.
- **OpenCL Attributes:** Descriptions of `__attributes` that can be added to OpenCL™ kernels to direct the results of the kernel build process.



TIP: The Xilinx® Runtime (XRT) Architecture reference material is available on the Xilinx Github site at this link: <https://xilinx.github.io/XRT/>.

Vitis Compiler Command

This section describes the Vitis compiler command, `v++`, and the various options it supports for both compiling and linking FPGA binary.

The Vitis compiler is a standalone command line utility for both compiling kernel accelerator functions into Xilinx object (`.xo`) files, and linking them with other `.xo` files and supported platforms to build an FPGA binary.

The following sections describe the `v++` command options for compile, link, and general processes:

- [Compiling Kernels with Vitis Compiler](#)
- [Linking the Kernels](#)
- [Vitis Compiler General Options](#)

Vitis Compiler General Options

The Vitis compiler supports many options for both the compilation process and the linking process. These options provide a range of features, and some apply specifically to compile or link, while others can be used, or are required for both compile and link.



TIP: All Vitis compiler options can be specified in a configuration file for use with the `--config` option, as discussed in the [Vitis Compiler Configuration File](#). For example, the `--platform` option can be specified in a configuration file without a section head using the following syntax:

```
platform=xilinx_u200_xdma_201830_2
```

--board_connection

- **Applies to:** Compile and link

```
--board_connection
```

Specifies a dual in-line memory module (DIMM) board file for each DIMM connector slot. The board is specified using the Vendor:Board:Name:Version (vbnv) attribute of the DIMM card as it appears in the board repository.

For example:

```
<DIMM_connector:vbvn_of_DIMM_board>
```

-c | --compile

- **Applies to:** Compile

```
--compile
```

Required for compilation, but mutually exclusive with `--link`. Run `v++ -c` to generate `.xo` files from kernel source files.

--config

- **Applies to:** Compile and link

```
--config <config_file> ...
```

Specifies a configuration file containing `v++` switches. The configuration file can be used to capture compilation or linking strategies, that can be easily reused by referring to the config file on the `v++` command line. In addition, the config file allows the `v++` command line to be shortened to include only the options that are not specified in the config file. Refer to the [Vitis Compiler Configuration File](#) for more information.

Multiple configuration files can be specified on the `v++` command line. A separate `--config` switch is required for each file used. For example:

```
v++ -l --config cfg_connectivity.txt --config cfg_vivado.txt ...
```

--custom_script

- **Applies to:** Compile and link

```
--custom_script <kernel_name>:<file_name>
```

This option is intended for use with the `<kernel_name>.tcl` file generated with `--export_script`. The argument lets you specify the kernel name, and path to the Tcl script to apply to that kernel.

For example:

```
v++ -l -k kernel1 -export_script ...
v++ -l --custom_script kernel1:./kernel1.tcl ...
```

-D | --define

- **Applies to:** Compile and link

```
--define <arg>
```

Valid macro name and definition pair: `<name>=<definition>`.

Pre-define name as a macro with definition. This option is passed to the `v++` pre-processor.

--dk

- **Applies to:** Compile and link

```
--dk <arg>
```

This option enables debug IP core insertion in the FPGA binary for hardware debugging, and lets you to specify which compute unit and interfaces to monitor with ChipScope™. The `--dk` option allows you to attach System ILA cores at the interfaces to the kernels for debugging and performance monitoring purposes.

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

Valid values include:

```
[protocol|chipscope|list_ports]:<cu_name>:<interface_name>
```

Where:

- `protocol` can be specified with the keyword `all`, or the `<cu_name>:<interface_name>`.
- The `chipscope` option requires the explicit name of the compute unit to be provided for the `<cu_name>` and `<interface_name>`.



TIP: The `chipscope` option can not accept the keyword `all`.

- The `list_ports` option shows a list of valid compute units and port combinations in the current design.
- `<cu_name>` specifies the compute unit to apply the `--dk` option to.
- `<interface_name>` is optional. If not specified, all ports on the specified CU are expected to be analyzed.

For example:

```
v++ --link --dk chipscope:vadd_1
```

--export_script

- **Applies to:** Compile and link

```
--export_script
```

Generates a Tcl script, `<kernel_name>.tcl`, that can be used to execute Vivado HLS, but stops the build process before actually launching HLS. This lets you stop the build process, edit the generated Tcl script to alter the build process in Vivado HLS, and then restart the build process using the `--custom_script` option, as shown in the following example:

```
v++ -l -k kernel1 -export_script ...
v++ -l --custom_script kernel1:./kernel1.tcl ...
```



TIP: This option is not supported for software emulation (`-t sw_emu`) of OpenCL kernels.

--from_step

- **Applies to:** Compile and link

```
--from_step <arg>
```

Specifies a step name, for either the compile or link process, to start the build process from that step. If intermediate results are available, the link will fast forward and begin execution at the named step if possible. This allows you to run the build through a `--to_step`, and then resume the build process at the `--from_step`, after interacting with your project in some method.

Note: You can use the `--list_step` option to determine the list of valid compile or link steps.

For example:

```
v++ --link --from_step vpl.update_bd
```

-g | --debug

- **Applies to:** Compile and link

```
-g
```

Generates code for debugging the kernel. Using this option adds features to facilitate debugging the kernel as it is compiled and the FPGA binary is built.

For example:

```
v++ -g ...
```

-h | --help

```
-h
```

Prints the help contents for the `v++` command. For example:

```
v++ -h
```

-I | --include

- **Applies to:** Compile and link

```
--include <arg>
```

Add the specified directory to the list of directories to be searched for header files. This option is passed to the Vitis compiler pre-processor.

<input_file>

- **Applies to:** Compile and link

```
<input_file1> <input_file2> ...
```

Specifies an OpenCL or C/C++ kernel source file for `v++` compilation, or Xilinx object files (`.xo`) for `v++` linking.

For example:

```
v++ -l kernel1.xo kernelRTL.xo ...
```

--interactive

- **Applies to:** Compile and link

```
--interactive [ synth | impl ]
```

`v++` configures necessary environment and launches the Vivado toolset with either synthesis or implementation project.

Because you are interactively launching the Vivado tool, the linking process is stopped at the `vp1` step, which is the equivalent of using the `--to_step vp1` option in your `v++` command. When you are done using the Vivado tool, and you save the design checkpoint (DCP), you can rerun the linking command using the `-from_step` to pick the command up at the `vp1` process.

For example:

```
v++ --interactive impl
```

-j | --jobs

- **Applies to:** Compile and link

```
--jobs <arg>
```

Valid values specify a number of parallel jobs.

This option specifies the number of parallel jobs the Vivado Design Suite uses to implement the FPGA binary. Increasing the number of jobs allows the Vivado implementation step to spawn more parallel processes and complete faster.

For example:

```
v++ --link --jobs 4
```

-k | --kernel

- **Applies to:** Compile and link

```
--kernel <arg>
```

Compile only the specified kernel from the input file. Only one `-k` option is allowed per `v++` command. Valid values include the name of the kernel to be compiled from the input `.cl` or `.c/.cpp` kernel source code.

This is required for C/C++ kernels, but is optional for OpenCL kernels. OpenCL uses the `kernel` keyword to identify a kernel. For C/C++ kernels, you must identify the kernel by `-k` or `--kernel`.

When an OpenCL source file is compiled without the `-k` option, all the kernels in the file are compiled. Use `-k` to target a specific kernel.

For example:

```
v++ -c --kernel vadd
```

--kernel_frequency

- **Applies to:** Compile and link

```
--kernel_frequency <clockID>:<freq>|<clockID>:<freq>
```

Specifies a user-defined clock frequency (in MHz) for the kernel, overriding the default clock frequency defined on the hardware platform. The `<freq>` specifies a single frequency for kernels with only a single clock, or can be used to specify the `<clockID>` and the `<freq>` for kernels that support two clocks.

The syntax for overriding the clock on a platform with only one kernel clock, is to simply specify the frequency in MHz:

```
v++ --kernel_frequency 300
```

To override a specific clock on a platform with two clocks, specify the clock ID and frequency:

```
v++ --kernel_frequency 0:300
```

To override both clocks on a multi-clock platform, specify each clock ID and the corresponding frequency. For example:

```
v++ --kernel_frequency 0:300|1:500
```

-l | --link

```
--link
```

This is a required option for the linking process, which follows compilation, but is mutually exclusive with `--compile`. Run `v++` in link mode to link `.xo` input files and generate an `.xclbin` output file.

--list_steps

- **Applies to:** Compile and link

```
--list_steps
```

List valid run steps for a given target. This option returns a list of steps that can be used in the `--from_step` or `--to_step` options. The command must be specified with the following options:

- `-t | --target [sw_emu | hw_emu | hw]`:
- `[--compile | --link]`: Specifies the list of steps from either the compile or link process for the specified build target.

For example:

```
v++ -t hw_emu --link --list_steps
```

--log_dir

- **Applies to:** Compile and link

```
--log_dir <dir_name>
```

Specifies a directory to store log files into. If `--log_dir` is not specified, the tool saves the log files to `./_x/logs`. Refer to [Output Directories from the v++ Command](#) for more information.

For example:

```
v++ --log_dir /tmp/myProj_logs ...
```

--lsf

- **Applies to:** Compile and link

```
--lsf <arg>
```

Specifies the `bsub` command line as a string to pass to an LSF cluster. This option is required to use the IBM Platform Load Sharing Facility (LSF) for Vivado implementation and synthesis.

For example:

```
v++ --link --lsf '{bsub -R \"select[type=X86_64]\" -N -q medium}'
```

--message_rules

- **Applies to:** Compile and link

```
--message-rules <file_name>
```

Specifies a message rule file with rules for controlling messages. Refer to [Using the Message Rule File](#) for more information.

For example:

```
v++ --message_rules ./minimum_out.mrf ...
```

--no_ip_cache

- **Applies to:** Compile and link

```
--no_ip_cache
```

Disables the IP cache for Vivado Synthesis.

For example:

```
v++ --no_ip_cache ...
```

-O | --optimize

- **Applies to:** Compile and link

```
--optimize <arg>
```

This option specifies the optimization level of the Vivado implementation results. Valid optimization values include the following:

- 0: Default optimization. Reduces compilation time and makes debugging produce the expected results.
- 1: Optimizes to reduce power consumption. This takes more time to build the design.
- 2: Optimizes to increase kernel speed. This option increases build time, but also improves the performance of the generated kernel.
- 3: This optimization provides the highest level performance in the generated code, but compilation time can increase considerably.

- `s`: Optimizes for size. This reduces the logic resources of the device used by the kernel.
- `quick`: Reduces Vivado implementation time, but can reduce kernel performance, and increases the resources used by the kernel.

For example:

```
v++ --link --optimize 2
```

-o | --output

- **Applies to:** Compile and link

```
-o <output_name>
```

Specifies the name of the output file generated by the `v++` command. The compilation (`-c`) process output name must end with the `.xo` suffix, for Xilinx object file. The linking (`-l`) process output file must end with the `.xclbin` suffix, for Xilinx executable binary.

For example:

```
v++ -o krnl_vadd.xo
```

If `--o` or `--output` are not specified, the output file names will default to the following:

- `a.o` for compilation.
- `a.xclbin` for linking.

-p | --platform

- **Applies to:** Compile and link

```
--platform <platform_name>
```

Specifies the name of a supported acceleration platform as specified by the `$PLATFORM_REPO_PATHS` environment variable, or the full path to the platform `.xpfm` file. For a list of supported platforms for the release, see the [Vitis 2019.2 Software Platform Release Notes](#).

This is a required option for both compilation and linking, to define the target Xilinx platform of the build process. The `--platform` option accepts either a platform name, or the path to a platform file `xpfm`, using the full or relative path.



IMPORTANT! *The specified platform and build targets for compiling and linking must match. The `--platform` and `-t` options specified when the `.xo` file is generated by compilation, must be the `--platform` and `-t` used during linking. For more information, see [Platforminfo](#).*

For example:

```
v++ --platform xilinx_u200_xdma_201830_2 ...
```



TIP: All Vitis compiler options can be specified in a configuration file for use with the `--config` option. For example, the `platform` option can be specified in a configuration file without a section head using the following syntax:

```
platform=xilinx_u200_xdma_201830_2
```

--profile_kernel

- **Applies to:** Compile and link

```
--profile_kernel <arg>
```

This option enables capturing profile data for data traffic between the kernel and host, kernel stalls, and kernel execution times. There are three distinct forms of `--profile_kernel`:

- `data`: Enables monitoring of data ports through the monitor IPs. This option needs to be specified during linking.
- `stall`: Includes stall monitoring logic in the FPGA binary. However, it requires the addition of stall ports on the kernel interface. To facilitate this, the `stall` option is required for both compilation and linking.
- `exec`: This option records the execution times of the kernel and provides minimum port data collection during the system run. The execution time of the kernel is also collected by default for `data` or `stall` data collection. This option needs to be specified during linking.



IMPORTANT! Using the `--profile_kernel` option in `v++` also requires the addition of the `profile=true` statement to the `xrt.ini` file. Refer to [xrt.ini File](#).

The syntax for `data` profiling is:

```
data:[ <kernel_name> | all ]:[ <cu_name> | all ]:[ <interface_name> | all ]
(:[ counters | all ])
```

The `kernel_name`, `cu_name`, and `interface_name` can be specified to determine the specific interface the performance monitor is applied to. However, you can also specify the keyword `all` to apply the monitoring to all existing kernels, compute units, and interfaces with a single option.

The last option, `<counters|all>` is not required, as it defaults to `all` when not specified. It allows you to restrict the information gathering to just `counters` for larger designs, while `all` will include the collection of actual trace information.

The syntax for `stall` or `exec` profiling is:

```
[ stall | exec ]:[ <kernel_name> | all ]:[ <cu_name> | all ](:[ counters | all ])
```



TIP: For `stall` or `exec`, the `<interface_name>` field is not used.

The following example enables logging profile data for all interfaces, on all CUs for all kernels:

```
v++ -g -l --profile_kernel data:all:all:all ...
```

Note: The `--profile_kernel` option is additive and can be used multiple times to specify profiling for different kernels, CUs, and interfaces.

--remote_ip_cache

- **Applies to:** Compile and link

```
--remote_ip_cache <dir_name>
```

Specifies the remote IP cache directory for Vivado Synthesis.

For example:

```
v++ --remote_ip_cache /tmp/IP_cache_dir ...
```

--report_dir

- **Applies to:** Compile and link

```
--report_dir <dir_name>
```

Specifies a directory to store report files into. If `--report_dir` is not specified, the tool saves the report files to `./_x/reports`. Refer to [Output Directories from the v++ Command](#) for more information.

For example:

```
v++ --report_dir /tmp/myProj_reports ...
```

-R | --report_level

- **Applies to:** Compile and link

```
--report_level <arg>
```

Valid report levels: 0, 1, 2, estimate.

These report levels have mappings kept in the `optMap.xml` file. You can override the installed `optMap.xml` to define custom report levels.

- `-R0` specification turns off all intermediate design checkpoint (DCP) generation during Vivado implementation. Turns on post-route timing report generation.
- The `-R1` specification includes everything from `-R0`, plus `report_failfast pre-opt_design, report_failfast post-opt_design`, and enables all intermediate DCP generation.
- The `-R2` specification includes everything from `-R1`, plus `report_failfast post-route_design`.
- The `-Restimate` specification forces Vivado HLS to generate a `design.xml` file if it does not exist and then generates a System Estimate report, as described in [System Estimate Report](#).



TIP: This option is useful for the software emulation build (`-t sw_emu`), when `design.xml` is not generated by default.

For example:

```
v++ -R2 ...
```

--reuse_impl

```
--reuse_impl <arg>
```

- **Applies to:** Compile and link

Specifies the path and file name of an implemented design checkpoint (DCP) file to use when generating the FPGA binary (`xclbin`) file. The link process uses the specified implemented DCP to extract the FPGA bitstream and generates the `xclbin`. This allows you to work interactively with Vivado Design Suite to change the design and use DCP in the build process.

For example:

```
v++ --link --reuse_impl ./manual_design.dcp
```

-s | --save-temps

- **Applies to:** Compile and link

```
--save-temps
```

Directs the `v++` command to save intermediate files/directories created during the compilation and link process. Use the `--temp_dir` option to specify a location to write the intermediate files to.



TIP: This option is useful for debugging when you encounter issues in the build process.

For example:

```
v++ --save_temps ...
```

-t | --target

- **Applies to:** Compile and link

```
-t [ sw_emu | hw_emu | hw ]
```

Specifies the build target, as described in [Build Targets](#). The build target determines the results of the compilation and linking processes. You can choose to build an emulation model for debug and test, or build the actual system to run in hardware. The build target defaults to `hw` if `-t` is not specified.



IMPORTANT! The specified platform and build targets for compiling and linking must match. The `--platform` and `-t` options specified when the `.xo` file is generated by compilation must be the `--platform` and `-t` used during linking.

The valid values are:

- `sw_emu`: Software emulation.
- `hw_emu`: Hardware emulation.
- `hw`: Hardware.

For example:

```
v++ --link -t hw_emu
```

--temp_dir

- **Applies to:** Compile and link

```
--temp_dir <dir_name>
```

This allows you to manage the location where the tool writes temporary files created during the build process. The temporary results are written by the `v++` compiler, and then removed, unless the `--save_temps` option is also specified.

If `--temp_dir` is not specified, the tool saves the temporary files to `./_x/temp`. Refer to [Output Directories from the v++ Command](#) for more information.

For example:

```
v++ --temp_dir /tmp/myProj_temp ...
```

--to_step

- **Applies to:** Compile and link

```
--to_step <arg>
```

Specifies a step name, for either the compile or link process, to run the build process through that step. The build process will terminate after completing the named step. At this time, you can interact with the build results. For example, manually accessing the HLS project or the Vivado Design Suite project to perform specific tasks before returning to the build flow, launch the `v++` command with the `--from_step` option.

Note: You can use the `--list_step` option to determine the list of valid compile or link steps.

For example:

```
v++ --link --to_step vpl.update_bd
```

--trace_memory

- **Applies to:** Compile and link

```
--trace_memory <arg>
```

`<FIFO:<size>|<MEMORY>[<n>]` specifies trace buffer memory type for profiling. FIFO size is specified in KB. Use with `--profile_kernel` option when linking with hardware target. Default is FIFO:8K. The maximum is 4G.

Note: When using `--trace_memory` during the linking step, you should also use the `[Debug] trace_buffer_size` in the `xrt.ini` file as described in [xrt.ini File](#).

-v | --version

```
-v
```

Prints the version and build information for the `v++` command. For example:

```
v++ -v
```

--user_board_repo_paths

- **Applies to:** Compile and link

```
--user_board_repo_paths
```

Specifies an existing user board repository for DIMM board files. This value will be pre-pended to the `board_part_repo_paths` property of the Vivado project.

--user_ip_repo_paths

- **Applies to:** Compile and link

```
--user_ip_repo_paths <repo_dir>
```

Specifies the directory location of one or more user IP repository paths to be searched first for IP used in the kernel design. This value is appended to the start of the `ip_repo_paths` used by the Vivado tool to locate IP cores. IP definitions from these specified paths are used ahead of IP repositories from the hardware platform (`.xsa`) or from the Xilinx IP catalog.



TIP: Multiple `--user_ip_repo_paths` can be specified on the `v++` command line.

The following lists show the priority order in which IP definitions are found during the build process, from high to low. Note that all of these entries can possibly include multiple directories in them.

- For the system hardware build (`-t hw`):
 1. IP definitions from `--user_ip_repo_paths`.
 2. Kernel IP definitions (`vpl --iprepo` switch value).
 3. IP definitions from the IP repository associated with the platform.
 4. IP cache from the installation area (for example, `<Install_Dir>/Vitis/2019.2/data/cache/`).
 5. Xilinx IP catalog from the installation area (for example, `<Install_Dir>/Vitis/2019.2/data/ip/`)
- For the hardware emulation build (`-t hw_emu`):
 1. IP definitions and User emulation IP repository from `--user_ip_repo_paths`.
 2. Kernel IP definitions (`vpl --iprepo` switch value).
 3. IP definitions from the IP repository associated with the platform.
 4. IP cache from the installation area (for example, `<Install_Dir>/Vitis/2019.2/data/cache/`).
 5. `$::env(XILINX_VITIS)/data/emulation/hw_em/ip_repo`
 6. `$::env(XILINX_VIVADO)/data/emulation/hw_em/ip_repo`
 7. Xilinx IP catalog from the installation area (for example, `<Install_Dir>/Vitis/2019.2/data/ip/`)

For example:

```
v++ --user_ip_repo_paths ./myIP_repo ...
```

--advanced Options

The `--advanced.XXX` options are a collection of miscellaneous parameters and properties for the `v++` command. When compiling or linking, fine-grain control over the hardware generated by the Vitis core development kit, the hardware emulation process can be specified by using the `--advanced.XXX` options.

The arguments for the `--advanced.XXX` options are specified as `:<keyword>=<value>`. For example:

```
v++ --link --advanced.param:compiler.enableXSAIntegrityCheck=true
--advanced.prop:kernel.foo.kernel_flags="-std=c++0x"
```



TIP: All Vitis compiler options can be specified in a configuration file for use with the `--config` option, as discussed in [Vitis Compiler Configuration File](#). For example, the `--platform` option can be specified in a configuration file without a section head using the following syntax:

```
platform=xilinx_u200_xdma_201830_2
```

--advanced.param

```
--advanced.param:<arg>
```

Specifies advanced parameters for kernel compilation where `<arg>` is one of the values described in the table below.

Table 13: Param Options

Parameter Name	Valid Values	Description
<code>param:compiler.acceleratorBinaryContent</code>	Type: String Default Value: <code><empty></code>	Content to insert in <code>xclbin</code> . Valid options are <code>bitstream</code> and <code>dcp</code> .
<code>param:compiler.errorOnHoldViolation</code>	Type: Boolean Default Value: <code>TRUE</code>	Error out if there is hold violation.
<code>param:compiler.maxComputeUnits</code>	Type: Int Default Value: <code>-1</code>	Maximum compute units allowed in the system. Any positive value will overwrite the <code>numComputeUnits</code> setting in the hardware platform (<code>.xsa</code>). The default value of <code>-1</code> preserves the setting in the platform.
<code>compiler.addOutputTypes</code>	Type: String Default Value: <code><empty></code>	Additional output types produced by the Vitis compiler. Valid values include: <code>xclbin</code> , <code>sd_card</code> , <code>hw_export</code> , and <code>qspi</code> .
<code>param:hw_em.compiledLibs</code>	Type: String Default Value: <code><empty></code>	Uses mentioned <code>libs</code> for the specified simulator.

Table 13: Param Options (cont'd)

Parameter Name	Valid Values	Description
<code>param:hw_em.platformPath</code> <absolute_path_of_custom_platform_directory>	Type: String Default Value: <empty>	Specifies the path to the custom platform directory. The <platformPath> directory should meet the following requirements to be used in platform creation: <ul style="list-style-type: none"> The directory should contain a subdirectory called <code>ip_repo</code>. The directory should contain a subdirectory called <code>scripts</code> and this <code>scripts</code> directory should contain a <code>hw_em_util.tcl</code> file. The <code>hw_em_util.tcl</code> file should have the following two procedures defined in it: <ul style="list-style-type: none"> <code>hw_em_util::add_base_platform</code> <code>hw_em_util::generate_simulation_scripts_and_compile</code>
<code>param:hw_em.enableProtocolChecker</code>	Type: Boolean Default Value: FALSE	Enables the lightweight AXI protocol checker (lapc) during HW emulation. This is used to confirm the accuracy of any AXI interfaces in the design.
<code>hw_em.simulator</code>	Type: String Values: XSIM, QUESTA Default Value: XSIM	Uses the specified simulator for the hardware emulation run.
<code>param:compiler.xclDataflowFifoDepth</code>	Type: Int Default Value: -1	Specifies the depth of FIFOs used in kernel data flow region.
<code>param:compiler.interfaceWrOutstanding</code>	Type: Int Range Default Value: 0	Specifies how many outstanding writes to buffer are on the kernel AXI interface. Values are 1 through 256.
<code>param:compiler.interfaceRdOutstanding</code>	Type: Int Range Default Value: 0	Specifies how many outstanding reads to buffer are on the kernel AXI interface. Values are 1 through 256.
<code>param:compiler.interfaceWrBurstLen</code>	Type: Int Range Default Value: 0	Specifies the expected length of AXI write bursts on the kernel AXI interface. This is used with option <code>compiler.interfaceWrOutstanding</code> to determine the hardware buffer sizes. Values are 1 through 256.
<code>param:compiler.interfaceRdBurstLen</code>	Type: Int Range Default Value: 0	Specifies the expected length of AXI read bursts on the kernel AXI interface. This is used with option <code>compiler.interfaceRdOutstanding</code> to determine the hardware buffer sizes. Values are 1 through 256.

For example:

```
--advanced.param "compiler.addOutputTypes=qspi,sd_card"
```



TIP: This option can be specified in a configuration file under the `[advanced]` section head using the following format:

```
[advanced]
param=compiler.addOutputTypes="qspi, sd_card"
```

--advanced.prop

```
--advanced.prop <arg>
```

Specifies advanced kernel or solution properties for kernel compilation where `<arg>` is one of the values described in the table below.

Table 14: Prop Options

Parameter Name	Valid Values	Description
<code>prop:kernel.<kernel_name>.kernel_flags</code>	Type: String Default Value: <code><empty></code>	Sets specific compile flags on the kernel <code><kernel_name></code> .
<code>prop:solution.device_repo_path</code>	Type: String Default Value: <code><empty></code>	Specifies the path to a repository of hardware platforms. The <code>--platform</code> option with full path to the <code>.xpfm</code> platform file should be used instead.
<code>prop:solution.hls_pre_tcl</code>	Type: String Default Value: <code><empty></code>	Specifies the path to a Vivado HLS Tcl file, which is executed before the C code is synthesized. This allows Vivado HLS configuration settings to be applied prior to synthesis.
<code>prop:solution.hls_post_tcl</code>	Type: String Default Value: <code><empty></code>	Specifies the path to a Vivado HLS Tcl file, which is executed after the C code is synthesized.
<code>prop:solution.kernel_compiler_margin</code>	Type: Float Default Value: 12.5% of the kernel clock period.	The clock margin (in ns) for the kernel. This value is subtracted from the kernel clock period prior to synthesis to provide some margin for P&R delays.

--advanced.misc

```
--advanced.misc:<arg>
```

Specifies advanced tool directives for kernel compilation.

--clock Options

The `--clock.XXX` options provide a method for assigning clocks to kernels from the `v++` command line and locating the required kernel clock frequency source during the linking process. There are a number of options that can be used with increasing specificity. The order of precedence is determined by how specific a clock option is. The rules are listed in order from general to specific, where the lower rules take precedence over the higher rules:

- When no `--clock.XX` option is specified, the platform default clock will be applied. For 2-clock kernels, clock ID 0 will be assigned to `ap_clk` and clock ID 1 will be assigned to `ap_clk_2`.
- Specifying `--clock.defaultId=<id>` defines a specific clock ID for all kernels, overriding the platform default clock.
- Specifying `--clock.defaultFreq=<Hz>` defines a specific clock frequency for all kernels that overrides a user specified default clock ID, and the platform default clock.
- Specifying `--clock.id=<id>:<cu>` assigns the specified clock ID to all clock pins on the specified CU, overriding user specified default frequency, ID, and the platform default clock.
- Specifying `--clock.id=<id>:<cu>.<clk0>` assigns the specified clock ID to the specified clock pin on the specified CU.
- Specifying `--clock.freqHz=<Hz>:<cu>` assigns the specified clock frequency to all clock pins on the specified CU.
- Specifying `--clock.freqHz=<Hz>:<cu>.<clk0>` assigns the specified clock frequency to the specified clock pin on the specified CU.

--clock.defaultFreqHz

```
--clock.defaultFreqHz <arg>
```

Specifies a default clock frequency in Hz to use for all kernels. This lets you override the default platform clock, and assign the clock with the specified clock frequency as the default. Where `<arg>` is specified as the clock frequency in Hz.

For example:

```
v++ --link --clock.defaultFreqHz 300000000
```



TIP: This option can be specified in a configuration file under the `[clock]` section head using the following format:

```
[clock]
defaultFreqHz=300000000
```

--clock.defaultId

```
--clock.defaultId <arg>
```

Specifying `--clock.defaultId=<id>` defines a specific clock ID for all kernels, overriding the platform default clock. Where `<arg>` is specified as the clock ID from one of the clocks defined on the target platform, other than the default clock ID.



TIP: You can determine the available clock IDs for a target platform using the `platforminfo` utility as described in [Platforminfo](#).

For example:

```
v++ --link --clock.defaultId 1
```



TIP: This option can be specified in a configuration file under the `[clock]` section head using the following format:

```
[clock]
defaultId=1
```

--clock.freqHz

```
--clock.freqHz <arg>
```

Specifies a clock frequency in Hz and assigns it to a list of associated compute units (CUs) and optionally specific clock pins on the CU. Where `<arg>` is specified as `<frequency_in_Hz>:<cu_0>[.<clk_pin_0>][,<cu_n>[.<clk_pin_n>]]:`

- `<frequency_in_Hz>`: Defines the clock frequency specified in Hz.
- `<cu_0>[.<clk_pin_0>][,<cu_n>[.<clk_pin_n>]]`: Applies the defined frequency to the specified CUs, and optionally to the specified clock pin on the CU.

For example:

```
v++ --link --clock.freqHz 300000000:vadd_1,vadd_3
```



TIP: This option can be specified in a configuration file under the `[clock]` section head using the following format:

```
[clock]
freqHz=300000000:vadd_1,vadd_3
```

--clock.id

```
--clock.id <arg>
```

Specifies an available clock ID from the target platform and assigns it to a list of associated compute units (CUs) and optionally specific clock pins on the CU. Where `<arg>` is specified as `<reference_ID>:<cu_0>[.<clk_pin_0>][,<cu_n>[.<clk_pin_n>]]:`

- `<reference_ID>`: Defines the clock ID to use from the target platform.



TIP: You can determine the available clock IDs for a target platform using the `platforminfo` utility as described in [Platforminfo](#).

- `<cu_0>[.<clk_pin_0>][,<cu_n>[.<clk_pin_n>]]`: Applies the defined frequency to the specified CUs and optionally to the specified clock pin on the CU.

For example:

```
v++ --link --clock.id 1:vadd_1,vadd_3
```



TIP: This option can be specified in a configuration file under the `[clock]` section head using the following format:

```
[clock]
id=1:vadd_1,vadd_3
```

--connectivity Options

As discussed in [Linking the Kernels](#), there are a number of `--connectivity.XXX` options that let you define the topology of the FPGA binary, specifying the number of CUs, assigning them to SLRs, connecting kernel ports to global memory, and establishing streaming port connections. These commands are an integral part of the build process, critical to the definition and construction of the application.

--connectivity.nk

```
--connectivity.nk <arg>
```

Where `<arg>` is specified as

```
<kernel_name>:#:<cu_name1>.<cu_name2>...<cu_name#>.
```

This instantiates the specified number of CU for the specified kernel in the generated FPGA binary (`.xclbin`) file during the linking process. The CU name is optional. If the CU name is not specified, the instances of the kernel are simply numbered: `kernel_name_1`, `kernel_name_2`, and so forth. By default, the Vitis compiler instantiates one compute unit for each kernel.

For example:

```
v++ --link --connectivity.nk vadd:3:vadd_A.vadd_B.vadd_C
```



TIP: This option can be specified in a configuration file under the `[connectivity]` section head using the following format:

```
[connectivity]
nk=vadd:3:vadd_A.vadd_B.vadd_C
```

--connectivity.slr

```
--connectivity.slr <arg>
```

Use this option to assign a CU to a specific SLR on the device. The option must be repeated for each kernel or CU being assigned to an SLR.



IMPORTANT! If you use `--connectivity.slr` to assign the kernel placement, then you must also use `--connectivity.sp` to assign memory access for the kernel.

Valid values include:

```
<cu_name>:<SLR_NUM>
```

Where:

- `<cu_name>` is the name of the compute unit as specified in the `--connectivity.nk` option. Generally this will be `<kernel_name>_1` unless a different name was specified.
- `<SLR_NUM>` is the SLR number to assign the CU to. For example, SLR0, SLR1.

For example, to assign CU `vadd_2` to SLR2, and CU `fft_1` to SLR1, use the following:

```
v++ --link --connectivity.slr vadd_2:SLR2 --connectivity.slr fft_1:SLR1
```



TIP: This option can be specified in a configuration file under the `[connectivity]` section head using the following format:

```
[connectivity]
slr=vadd_2:SLR2
slr=fft_1:SLR1
```

--connectivity.sp

```
--connectivity.sp <arg>
```

Use this option to specify the assignment of kernel interfaces to specific memory resources. A separate `--connectivity.sp` option is required to map each interface of a kernel to a particular memory resource. Any kernel interface not explicitly mapped to a memory resource through the `--connectivity.sp` option will be automatically connected to an available memory resource during the build process.

Valid values include:

```
<cu_name>.<kernel_interface_name>:<sptag[min:max]>
```

Where:

- `<cu_name>` is the name of the compute unit as specified in the `--connectivity.nk` option. Generally this will be `<kernel_name>_1` unless a different name was specified.
- `<kernel_interface_name>` is the name of the function argument for the kernel, or compute unit port.
- `<sptag>` represents a memory resource name from the target platform. Valid `<sptag>` names include DDR, PLRAM, and HBM.

- `[min:max]` enables the use of a range of memory, such as `DDR[0:2]`. A single index is also supported: `DDR[2]`.



TIP: The supported `<sptag>` and range of memory resources for a target platform can be obtained using the `platforminfo` command. Refer to the Platforminfo section for more information.

The following example maps the input argument (A) for the specified CU of the VADD kernel to `DDR[0:3]`, input argument (B) to `HBM[0:31]`, and writes the output argument (C) to `PLRAM[2]`:

```
v++ --link --connectivity.sp vadd_1.A:DDR[0:3] --connectivity.sp
vadd_1.B:HBM[0:31] \
--connectivity.sp vadd_1.C:PLRAM[2]
```



TIP: This option can be specified in a configuration file under the `[connectivity]` section head using the following format:

```
[connectivity]
sp=vadd_1.A:DDR[0:3]
sp=vadd_1.B:HBM[0:31]
sp=vadd_1.C:PLRAM[2]
```

--connectivity.stream_connect

```
--connectivity.stream_connect <arg>
```

Create a streaming connection between two compute units through their AXI4-Stream interfaces. Use a separate `--connectivity.stream_connect` command for each streaming interface connection. Valid values include:

```
<cu_name>.<kernel_interface_name>:<cu_name>.<kernel_interface_name>
```

Where:

- `<cu_name>` is the compute unit name specified in the `--connectivity.nk` option. Generally this will be `<kernel_name>_1` unless a different name was specified.
- `<kernel_interface_name>` is the function argument for the compute unit port that is declared as AXI4-Stream.

For example, to connect the AXI4-Stream port `s_out` of the compute unit `mem_read_1` to AXI4-Stream port `s_in` of the compute unit `increment_1`, use the following:

```
--connectivity.stream_connect mem_read_1.s_out:increment_1.s_in
```



TIP: This option can be specified in a configuration file under the `[connectivity]` section head using the following format:

```
[connectivity]
stream_connect=mem_read_1.s_out:increment_1.s_in
```

--hls Options

The `--hls.XXX` options described below are used to specify options for the Vivado HLS synthesis process invoked during kernel compilation.

--hls.clock

```
--hls.clock <arg>
```

Specifies a frequency in Hz at which the listed kernel(s) should be compiled by Vivado HLS.

Where `<arg>` is specified as:

```
<frequency_in_Hz>:<cu_name1>,<cu_name2>,...,<cu_nameN>
```

- `<frequency_in_Hz>`: Defines the kernel frequency specified in Hz.
- `<kernel1>,<kernel2>,...`: Defines a list of kernels to be compiled at the specified target frequency.

For example:

```
v++ -c --hls.clock 300000000:mmult,mmadd --hls.clock 100000000:fifo_1
```



TIP: This option can be specified in a configuration file under the `[hls]` section head using the following format:

```
[hls]
clock=300000000:mmult,mmadd
clock=100000000:fifo_1
```

--hls.export_mode

```
--hls.export_mode
```

Specifies an export mode from HLS with the path to an exported file. The value is specified as `<file_type>:<file_path>`.

Where `<file_type>` can be specified as:

- `xo`: For Xilinx object file.
- `tcl`: For Tcl script.

For example:

```
v++ --hls.export_mode tcl:./hls_export
```



TIP: This option can be specified in a configuration file under the `[hls]` section head using the following format:

```
[hls]
export_mode=tcl:./hls_export
```

--hls.export_project

```
--hls.export_project
```

Specifies a directory where the HLS project setup script is exported.

For example:

```
v++ --hls.export_project ./hls_export
```



TIP: This option can be specified in a configuration file under the `[hls]` section head using the following format:

```
[hls]
export_project=./hls_export
```

--hls.max_memory_ports

```
--hls.max_memory_ports <arg>
```

Indicates that a separate AXI interface port should be created for each argument of a kernel. If not enabled, the compiler creates a single AXI interface combining all kernel ports. Valid values include `all` kernels, or specify a `<kernel_name>`.

This option is valid only for OpenCL kernels.

For example:

```
v++ --hls.max_memory_ports vadd
```



TIP: This option can be specified in a configuration file under the `[hls]` section head using the following format:

```
[hls]
max_memory_ports=vadd:vadd_1
```

--hls.memory_port_data_width

```
--hls.memory_port_data_width <arg>
```

Sets the memory port data width to the specified `<number>` for all kernels, or for a given `<kernel name>`. Valid values include `<number>` or `<kernel_name>:<number>`.

Valid for OpenCL kernels.

For example:

```
v++ --hls.memory_port_data_width 256
```



TIP: This option can be specified in a configuration file under the `[hls]` section head using the following format:

```
[hls]
memory_port_data_width=256
```

--vivado Options

The `--vivado.XXX` options are paired with parameters and properties to configure the Vivado tools. For instance, you can configure optimization, placement, and timing, or specify which reports to output.



IMPORTANT! Familiarity with the Vivado Design Suite is required to make the best use of these options. See the Vivado Design Suite User Guide: Implementation ([UG904](#)) for more information.

--vivado.param

```
--vivado.param <arg>
```

Specifies parameters for the Vivado Design Suite to be used during synthesis and implementation of the FPGA binary (`xclbin`).

--vivado.prop

```
--vivado.prop <arg>
```

Specifies properties for the Vivado Design Suite to be used during synthesis and implementation of the FPGA binary (`xclbin`).

Table 15: Prop Options

Parameter Name	Valid Values	Description
<code>vivado.prop:<object_type>.<object_name>.<prop_name></code>	Type: Various	<p>This allows you to specify any property used in the Vivado hardware compilation flow.</p> <p><code><object_type></code> is <code>run fileset file project</code>.</p> <p>The <code><object_name></code> and <code><prop_name></code> values are described in <i>Vivado Design Suite Properties Reference Guide (UG912)</i>.</p> <p>Examples:</p> <pre>vivado_prop:run.impl_1. {STEPS.PLACE_DESIGN.ARGS.MORE OPTIONS}={-fanout_opt}</pre> <pre>vivado_prop:fileset. current.top=foo</pre> <p>If <code><object_type></code> is set to <code>file</code>, <code>current</code> is not supported.</p> <p>If <code><object_type></code> is set to <code>run</code>, the special value of <code>__KERNEL__</code> can be used to specify run optimization settings for ALL kernels, instead of the need to specify them one by one.</p>

For example:

```
v++ --link --vivado.prop:run.impl_1.STEPS.PHYS_OPT_DESIGN.IS_ENABLED=true
--vivado.prop:run.impl_1.STEPS.PHYS_OPT_DESIGN.ARGS.DIRECTIVE=Explore
```



TIP: This option can be specified in a configuration file under the `[vivado]` section head using the following format:

```
[vivado]
prop=run.impl_1.STEPS.PHYS_OPT_DESIGN.IS_ENABLED=true
prop=run.impl_1.STEPS.PHYS_OPT_DESIGN.ARGS.DIRECTIVE=Explore
```

Vitis Compiler Configuration File

A configuration file can also be used to specify the Vitis compiler options. A configuration file provides an organized way of passing options to the compiler by grouping similar switches together, and minimizing the length of the `v++` command line. Some of the features that can be controlled through config file entries include:

- HLS options to configure kernel compilation
- Connectivity directives for system linking such as the number of kernels to instantiate or the assignment of kernel ports to global memory
- Directives for the Vivado Design Suite to manage hardware synthesis and implementation.

In general, any `v++` command option can be specified in a configuration file. However, the configuration file supports defining sections containing groups of related commands to help manage build options and strategies. The following table lists the defined sections.

Table 16: Section Tags of the Configuration File

Section Name	Compiler/Linker	Description
[hls]	compiler	HLS directives <code>--hls</code> Options: <ul style="list-style-type: none"> clock export_project export_mode max_memory_ports memory_port_data_width
[clock]	compiler	Clock commands to <code>--clock</code> Options: <ul style="list-style-type: none"> defaultFreqHz defaultID freqHz id
[connectivity]	linker	<code>--connectivity</code> Options: <ul style="list-style-type: none"> nk sp stream_connect slr connect
[vivado]	linker	<code>--vivado</code> Options: <ul style="list-style-type: none"> param prop
[advanced]	either	<code>--advanced</code> Options: <ul style="list-style-type: none"> param prop misc



TIP: Comments can be added to the configuration file by starting the line with a "#". The end of a section is specified by an empty line at the end of the section.

Because the `v++` command supports multiple config files on a single `v++` command line, you can partition your configuration files into related options that define compilation and linking strategies or Vivado implementation strategies, and apply multiple config files during the build process.

Configuration files are optional. There are no naming restrictions on the files and the number of configuration files can be zero or more. All `v++` options can be put in a single configuration file if desired. However, grouping related switches into separate files can help you organize your build strategy. For example, group `[connectivity]` related switches in one file, and `[Vivado]` options into a separate file.

The configuration file is specified through the use of the `v++ --config` option as discussed in the [Vitis Compiler General Options](#). An example of the `--config` option follows:

```
v++ --config ../src/connectivity.cfg
```

Switches are read in the order they are encountered. If the same switch is repeated with conflicting information, the first switch read is used. The order of precedence for switches is as follows, where item one takes highest precedence:

1. Command line switches.
2. Config files (on command line) from left-to-right.
3. Within a config file, precedence is from top-to-bottom.

Using the Message Rule File

The `v++` command executes various Xilinx tools during kernel compilation and linking. These tools generate many messages that provide build status to you. These messages might or might not be relevant to you depending on your focus and design phase. The Message Rule file (`.mrf`) can be used to better manage these messages. It provides commands to promote important messages to the terminal or suppress unimportant ones. This helps you better understand the kernel build result and explore methods to optimize the kernel.

The Message Rule file is a text file consisting of comments and supported commands. Only one command is allowed on each line.

Comment

Any line with “#” as the first non-white space character is a comment.

Supported Commands

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

- `promote`: This command indicates that matching messages should be promoted to the `v++` output.
- `suppress`: This command indicates that matching messages should be suppressed or filtered from the `v++` output. Note that errors cannot be suppressed.

Enter only one command per line.

Command Options

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

- `-id [<message_id>]`: All messages matching the specified message ID are promoted or suppressed. The message ID is in format of nnn-mmm. As an example, the following is a warning message from HLS. The message ID in this case is 204-68.

```
WARNING: [V++ 204-68] Unable to enforce a carried dependence constraint
(II = 1, distance = 1, offset = 1)
between bus request on port 'gmem'
(/matrix_multiply_cl_kernel/mmult1.cl:57) and bus request on port 'gmem'-
severity [severity_level]
```

For example, to suppress messages with message ID 204-68, specify the following:

```
suppress -id 204-68.
```

- `-severity [<severity_level>]`: The following are valid values for the severity level. All messages matching the specified severity level will be promoted or suppressed.
 - `info`
 - `warning`
 - `critical_warning`

For example, to promote messages with severity of 'critical-warning', specify the following:

```
promote -severity critical_warning.
```

Precedence of Message Rules

The `suppress` rules take precedence over `promote` rules. If the same message ID or severity level is passed to both `promote` and `suppress` commands in the Message Rule file, the matching messages are suppressed and not displayed.

Example of Message Rule File

The following is an example of a valid Message Rule file:

```
# promote all warning, critical warning
promote -severity warning
promote -severity critical_warning
# suppress the critical warning message with id 19-2342
suppress -id 19-2342
```

xrt.ini File

The Xilinx runtime (XRT) library uses various control parameters to specify debugging, profiling, and message logging when running the host application and kernel execution. These control parameters are specified in a runtime initialization file, `xrt.ini` and used to configure features of XRT at startup.

If you are a command line user, the `xrt.ini` file needs to be created manually and saved to the same directory as the host executable. The runtime library checks if `xrt.ini` exists in the same directory as the host executable and automatically reads the file to configure the runtime.



TIP: The Vitis IDE creates the `xrt.ini` file automatically based on your run configuration and saves it with the host executable.

Runtime Initialization File Format

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

The following is an example `xrt.ini` file that enables the timeline trace feature, and directs the runtime log messages to the Console view.

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

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

There are three groups of initialization keys:

- Debug
- Runtime
- Emulation

The following tables list all supported keys for each group, the supported values for each key, and a short description of the purpose of the key.

Table 17: Runtime Group

Key	Valid Values	Descriptions
<code>api_checks</code>	<code>[true false]</code>	Enable or disable OpenCL API checks. <ul style="list-style-type: none"> • true: Enable. This is the default value. • false: Disable.

Table 17: Runtime Group (cont'd)

Key	Valid Values	Descriptions
<code>cpu_affinity</code>	<code>[{N,N,...}]</code>	Pin all runtime threads to specified CPUs. Example: <pre>cpu_affinity = {4,5,6}</pre>
<code>polling_throttle</code>	<code>[N]</code>	Specify the time interval in microseconds that the runtime library polls the device status. The default value is 0.
<code>runtime_log</code>	<code>[null console syslog filename]</code>	Specify where the runtime logs are printed <ul style="list-style-type: none"> <code>null</code>: Do not print any logs. This is the default value. <code>console</code>: Print logs to <code>stdout</code> <code>syslog</code>: Print logs to Linux syslog. <code>filename</code>: Print logs to the specified file. For example, <code>runtime_log=my_run.log</code>.
<code>verbosity</code>	<code>[0 1 2 3]</code>	Verbosity of the log messages. The default value is 0.

Table 18: Debug Group

Key	Valid Values	Descriptions
<code>app_debug</code>	<code>[true false]</code>	Enable <code>xprint</code> and <code>xstatus</code> command during debugging with GDB. <ul style="list-style-type: none"> <code>true</code>: Enable. <code>false</code>: Disable. This is the default value.
<code>data_transfer_trace=<arg></code>	<code><coarse fine off></code>	Enable device-level AXI transfers trace. <ul style="list-style-type: none"> <code>coarse</code>: Show CU transfer activity from beginning of first transfer to end of last transfer (before compute unit transfer ends). <code>fine</code>: Show all AXI-level burst data transfers. <code>off</code>: Turn off reading and reporting of device-level trace during runtime. This is the default value.
<code>device_profile</code>	<code>[true false]</code>	Enable or disable device profiling. <ul style="list-style-type: none"> <code>true</code>: Enable. <code>false</code>: Disable. This is the default value.

Table 18: Debug Group (cont'd)

Key	Valid Values	Descriptions
<code>profile</code>	<code>[true false]</code>	Enable or disable OpenCL code profiling. <ul style="list-style-type: none"> <code>true</code>: Enable. <code>false</code>: Disable. This is the default value. When this option is specified as <code>true</code> , the runtime enables basic profile monitoring. Without any additional options, this enables the host runtime logging profile summary. However, when <code>false</code> , no profile monitoring is performed at all.
<code>stall_trace=<arg></code>	<code><dataflow memory pipe all off></code>	Specify what types of stalls to capture and report in the timeline trace. The default is <code>off</code> . <p>Note: Enabling stall tracing can often fill the trace buffer, which results in incomplete and potentially corrupt timeline traces. This can be avoided by setting <code>stall_trace=off</code>.</p> <ul style="list-style-type: none"> <code>off</code>: Turn off any stall trace information gathering. <code>all</code>: Record all stall trace information. <code>dataflow</code>: Intra-kernel streams (for example, writing to full FIFO between dataflow blocks). <code>memory</code>: External memory stalls (for example, AXI4 read from the DDR). <code>pipe</code>: Inter-kernel pipe for OpenCL kernels (for example, writing to full pipe between kernels).
<code>timeline_trace</code>	<code>[true false]</code>	Enable or disable profile timeline trace <ul style="list-style-type: none"> <code>true</code>: Enable. <code>false</code>: Disable. This is the default value. This option will enable data gathering for the timeline trace. However, without adding Acceleration Monitors and AXI Performance Monitor IP into the kernels, the timeline will only show host information. At a minimum, to get compute unit start and end times in the timeline, the CU needs to be built with <code>--profile_kernel_exec</code> as described in Vitis Compiler Command .
<code>trace_buffer_size</code>	<code><value{K M G}></code>	Specifies the size of the memory allocated to capture trace data. This helps to ensure you can capture enough trace data. The value is specified as the amount of memory to allocate, for example, 64K, 200M, 1G.

Table 19: Emulation Group

Key	Valid Values	Descriptions
<code>aliveness_message_interval</code>	Any integer	Specify the interval in seconds that aliveness messages need to be printed. The default is 300.
<code>launch_waveform</code>	<code>[off batch gui]</code>	Specify how the waveform is saved and displayed during emulation. <ul style="list-style-type: none"> • <code>off</code>: Do not launch simulator waveform GUI, and do not save <code>wdb</code> file. This is the default value. • <code>batch</code>: Do not launch simulator waveform GUI, but save <code>wdb</code> file • <code>gui</code>: Launch simulator waveform GUI, and save <code>wdb</code> file <p>Note: The kernel needs to be compiled with debug enabled (<code>v++ -g</code>) for the waveform to be saved and displayed in the simulator GUI.</p>
<code>print_infos_in_console</code>	<code>[true false]</code>	Controls the printing of emulation info messages to users console. Emulation info messages are always logged into a file called <code>emulation_debug.log</code> <ul style="list-style-type: none"> • <code>true</code>: Print in users console. This is the default value. • <code>false</code>: Do not print in user console.
<code>print_warnings_in_console</code>	<code>[true false]</code>	Controls the printing emulation warning messages to users console. Emulation warning messages are always logged into a file called <code>emulation_debug.log</code> . <ul style="list-style-type: none"> • <code>true</code>: Print in users console. This is the default value. • <code>false</code>: Do not print in user console.
<code>print_errors_in_console</code>	<code>[true false]</code>	Controls printing emulation error messages in users console. Emulation error messages are always logged into the <code>emulation_debug.log</code> file. <ul style="list-style-type: none"> • <code>true</code>: Print in users console. This is the default value. • <code>false</code>: Do not print in user console.

Platforminfo

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

The following command options are available to use with `platforminfo`:

Table 20: platforminfo Commands

Option	Description
-h [--help]	Print help message and exit.
-k [--keys]	Get keys for a given platform. Returns a list of JSON paths.
-l [--list]	List platforms. Searches the user repo paths \$PLATFORM_REPO_PATHS and then the install locations to find .xpfm files.
-e [--extended]	List platforms with extended information. Use with '--list'.
-d [--hw] <arg>	Hardware platform definition (*.dsa) on which to operate. The value must be a full path, including file name and .dsa extension.
-s [--sw] <arg>	Software platform definition (*.spfm) on which to operate. The value must be a full path, including file name and .spfm extension.
-p [--platform] <arg>	<p>Xilinx® platform definition (*.xpfm) on which to operate. The value for --platform can be a full path including file name and .xpfm extension, as shown in example 1 below. If supplying a file name and .xpfm extension without a path, this utility will search only the current working directory. You can also specify just the base name for the platform. When the value is a base name, this utility will search the \$PLATFORM_REPO_PATHS, and the install locations, to find a corresponding .xpfm file, as shown in example 2 below.</p> <p>Example 1: --platform /opt/xilinx/platforms/xilinx_u200_xdma_201830_1.xpfm</p> <p>Example 2: --platform xilinx_u200_xdma_201830_1</p>
-o [--output] <arg>	Specify an output file to write the results to. By default the output is returned to the terminal (stdout).
-j [--json] <arg>	<p>Specify JSON format for the generated output. When used with no value, the platforminfo utility prints the entire platform in JSON format. This option also accepts an argument that specifies a JSON path, as returned by the -k option. The JSON path, when valid, is used to fetch a JSON subtree, list, or value.</p> <p>Example 1: platforminfo --json="hardwarePlatform" --platform <platform base name></p> <p>Example 2: Specify the index when referring to an item in a list. platforminfo --json="hardwarePlatform.devices[0].name" --platform <platform base name></p> <p>Example 3: When using the short option form (-j), the value must follow immediately. platforminfo -j"hardwarePlatform.systemClocks[]" -p <platform base name></p>
-v [--verbose]	Specify more detailed information output. The default behavior is to produce a human-readable report containing the most important characteristics of the specified platform.

Note: Except when using the --help or --list options, a platform must be specified. You can specify the platform using the --platform option, or using either --hw, --sw. You can also simply insert the platform name or full path into the command line positionally.

To understand the generated report, condensed output logs, based on the following command are reviewed. Note that the report is broken down into specific sections for better understandability.

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

Basic Platform Information

Platform information and high-level description are reported.

```
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: Vitis
```

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 v++` directive when overriding the default value.

```
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 `v++` command as shown below:

```
v++ ... --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 `v++` build. For example in the report below, DDR with `Segment Index 1` is used by default.
- **automatic:** When the maximum number of memory interfaces have been used under `Consumption: default` have been fully applied, then the interfaces under `automatic` is used. The maximum number of interfaces per memory resource are given in the **Max Masters** field.
- **explicit:** For PLRAM, consumption is set to `explicit` which indicates this memory resource is only used when explicitly indicated through the `v++` command line.

Feature ROM Information

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

```
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 users that have an OS running on the device, and not applicable to users that 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

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

The following command options are available:

Table 21: kernelinfo Commands

Option	Description
<code>-h [--help]</code>	Print help message.
<code>-x [--xo_path] <arg></code>	Absolute path to O file including file name and <code>.xo</code> extension
<code>-l [--log] <arg></code>	By default, information is displayed on the screen. Otherwise, you can use the <code>--log</code> option to output the information as a file.
<code>-j [--json]</code>	Output the file in JSON format.
<code>[input_file]</code>	XO file. Specify the XO file positionally or use the <code>--xo_path</code> option.
<code>[output_file]</code>	Output from Xilinx OpenCL™ Compiler. Specify the output file positionally, or use the <code>--log</code> option.

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

```
kernelinfo <filename.o>
```

The output is divided into three sections:

- Kernel Definitions
- Arguments
- Ports

The report generated by the following command is reviewed to help better understand the report content. Note that the report is broken down into specific sections for better understandability.

```
kernelinfo krnl_vadd.o
```

Where `krnl_vadd.o` is a packaged kernel.

Kernel Definition

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

```

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

Arguments

Reports kernel function arguments.

In the following example, there are four arguments: `a`, `b`, `c`, and `n_elements`.

```

=== 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.

The available command options are shown in the following table.

Table 22: xclbinutil Commands

Option	Description
<code>-h [--help]</code>	Print help messages.
<code>-i [--input]<arg></code>	Input file name. Reads xclbin into memory.
<code>-o [--output]<arg></code>	Output file name. Writes in memory xclbin image to a file.
<code>-v [--verbose]</code>	Display verbose/debug information
<code>-q [--quiet]</code>	Minimize reporting information.
<code>--migrate-forward</code>	Migrate the xclbin archive forward to the new binary format.

Table 22: xclbinutil Commands (cont'd)

Option	Description
<code>--remove-section<arg></code>	Section name to remove.
<code>--add-section<arg></code>	Section name to add. Format: <section>:<format>:<file>
<code>--dump-section<arg></code>	Section to dump. Format: <section>:<format>:<file>
<code>--replace-section<arg></code>	Section to replace.
<code>--key-value<arg></code>	Key value pairs. Format: [USER SYS]:<key>:<value>
<code>--remove-key<arg></code>	Removes the given user key from the xclbin archive.
<code>--add-signature<arg></code>	Adds a user defined signature to the given xclbin image.
<code>--remove-signature</code>	Removes the signature from the xclbin image.
<code>--get-signature</code>	Returns the user defined signature (if set) of the xclbin image.
<code>--info</code>	Report accelerator binary content. Including: generation and packaging data, kernel signatures, connectivity, clocks, sections, etc
<code>--list-names</code>	List all possible section names (Stand Alone Option).
<code>--version</code>	Version of this executable.
<code>--force</code>	Forces a file overwrite.

The following are various use examples of the tool.

- Reporting xclbin information:** `xclbinutil --info --input binary_container_1.xclbin`
- Extracting the bitstream image:** `xclbinutil --dump-section BITSTREAM:RAW:bitstream.bit --input binary_container_1.xclbin`
- Extracting the build metadata:** `xclbinutil --dump-section BUILD_METADATA:HTML:buildMetadata.json --input binary_container_1.xclbin`
- Removing a section:** `xclbinutil --remove-section BITSTREAM --input binary_container_1.xclbin --output binary_container_modified.xclbin`

For most users, the contents and how the xclbin was created is desired. This information can be obtained through the `--info` option and reports information on the xclbin, hardware platform, clocks, memory configuration, kernel, and how the xclbin was generated.

The output of the xclbinutil command using the `--info` option is shown below divided into sections.

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

xclbin Information

```

Generated by:      v++ (2019.1) 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 [Platforminfo](#).

```

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

```

Generated By
-----
Command:      v++
Version:      2018.3 - Tue Nov 20 19:42:42 MST 2018 (SW BUILD: 2394611)
Command Line: v++ -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.o
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.o
=====
==
User Added Key Value Pairs
-----
      <empty>
=====
==
    
```

Emconfigutil

When running software or hardware emulation in the command line flow, it is necessary to create an emulation configuration file, `emconfig.json`, used by the runtime library during emulation. The emulation configuration file defines the device type and quantity of devices to emulate for the specified platform. A single `emconfig.json` file can be used for both software and hardware emulation.

Note: When running on real hardware, the runtime and drivers query the installed hardware to determine the device type and quantity installed, along with the device characteristics.

The `emconfigutil` utility automates the creation of the emulation file using the following steps:

1. Specify the target platform and additional options in the `emconfigutil` command line:

```
emconfigutil --platform <platform_name> [options]
```

At a minimum, you need to specify the target platform through the `-f` or `--platform` option to generate the required `emconfig.json` file. The specified platform must be the same as specified during the host and hardware builds.

The `emconfigutil` options are provided in the following table.

Table 23: emconfigutil Options

Option	Valid Values	Description
<code>-f</code> or <code>--platform</code>	Target device	Required. Defines the target device from the specified platform. For a list of supported devices, refer to Supported Platforms .
<code>--nd</code>	Any positive integer	Optional. Specifies number of devices. The default is 1.
<code>--od</code>	Valid directory	Optional. Specifies the output directory. When running emulation, the <code>emconfig.json</code> file must be in the same directory as the host executable. The default is to write the output in the current directory.
<code>-s</code> or <code>--save-temps</code>	N/A	Optional. Specifies that intermediate files are not deleted and remain after the command is executed. The default is to remove temporary files.
<code>--xp</code>	Valid Xilinx parameters and properties.	Optional. Specifies additional parameters and properties. For example: <pre>--xp prop:solution.platform_repo_paths=<xsa_path></pre> This example sets the search path for the target platforms.
<code>-h</code> or <code>--help</code>	N/A	Prints command help.

2. The `emconfigutil` command generates the `emconfig.json` configuration file in the output directory or the current working directory.



TIP: When running emulation, the `emconfig.json` file must be in the same directory as the host executable.

The following example creates a configuration file targeting two `xilinx_u200_qdma_201910_1` devices.

```
$emconfigutil --xilinx_u200_qdma_201910_1 --nd 2
```

Xilinx Board Utility

The Xilinx® Board Utility (`xbutil`) is a standalone command line utility that is included with the Xilinx Run Time (XRT) installation package. The `xbutil` command supports both Alveo Data Center accelerator cards, and embedded processor-based platforms. It includes multiple commands to validate and identify the installed accelerator card(s) along with additional card details including DDR, PCIe®, target platform name, and system information. This information can be used for both card administration and application debugging. Some of these include:

- Card administration tasks:
 - Flash card firmware.
 - Reset hung cards.
 - Query card status, sensors, and PCI Express AER registers.
- Debug operations:
 - Download the binary (`.xclbin`) to FPGA.
 - Test DMA for PCIe bandwidth.
 - Show status of compute units.

The `xbutil` command line format is:

```
xbutil <command> [options]
```

where the available commands and options are given below.



IMPORTANT! Although `xbutil` supports embedded processor platforms, only the following commands are available for use with those platforms: `dump`, `help`, `list`, `mem read`, `mem write`, `program`, and `query`.

- [clock](#)
- [dmatest](#)
- [dump](#)
- [flash](#)
- [flash scan](#)
- [help](#)
- [list](#)
- [m2mtest](#)
- [mem read](#)
- [mem write](#)

- [p2p](#)
- [program](#)
- [query](#)
- [reset](#)
- [scan](#)
- [status](#)
- [top](#)
- [validate](#)

To run the `xbutil` command without prepending the path `/opt/xilinx/xrt/bin/`, run the following command.

Use the following command in `cs` shell:

```
$ source /opt/xilinx/xrt/setup.csh
```

Use the following command in `bash` shell:

```
$ source /opt/xilinx/xrt/setup.sh
```

Note: The `sudo` access is required for the `flash` and `flash scan` options.

clock



IMPORTANT! *This option cannot be used with embedded processor platforms.*

The `clock` command allows you to change the clock frequencies driving the computing units. Note that your compute units must be capable or running at the specified clock. You can modify both `clock1` and `clock2` using this command.

It has the following options:

- `-d <card>` (Optional): Specifies the target card. Default = 0 if not specified.
- `-r <region>` (Optional): Specifies the target region. Default = 0 if not specified.
- `-f <clock1_freq_MHz>` (Required): Specifies clock frequency (in MHz) for the first clock. All platforms have this clock.
- `-g <clock2_freq_MHz>` (Optional): Specifies clock frequency (in MHz) for the second clock. Some platforms may not have this clock.
- `-h <clock3_freq_MHz>` (Optional): Specifies clock frequency (in MHz) for the third clock. Some platforms may not have this clock.

For example, to change clock1 in card 0 to 100 MHz, run the following command:

```
$ xbutil clock -d 0 -f 100
```

Similarly, to change two clocks in card 0, such that clock1 is set to 200 MHz and clock2 is set to 250 MHz, run this command:

```
$ xbutil clock -d 0 -f 200 -g 250
```

The following example is an output after running this command:

```
INFO: Found total 1 card(s), 1 are usable
INFO: xbutil clock succeeded.
```

dmatest



IMPORTANT! *This option cannot be used with embedded processor platforms.*

The `dmatest` command performs throughput data transfer tests between the host machine and global memory on a specified card. Note, it is necessary to download an `xclbin` on the card prior to running `dmatest`, else running this command returns an error. The `dmatest` command only performs throughput tests on those DDR banks accessed by the `xclbin` downloaded to the card.

The command has the following options:

- `-d card_id` (Optional): Specifies the target card. Default = 0 if not specified.
- `-b blocksize` (Optional): Specifies the test block size (in KB). Default = 65536 (KB) if not specified. The block size can be specified in both decimal or hexadecimal formats. For example, both `-b 1024` and `-b 0x400` set the block size to 1024 KB.

To run the `dmatest` command, enter the following:

```
$ xbutil dmatest
```

An example of the command output with an `xclbin` using DDR banks 0, 1, 2, and 3 is shown below:

```
INFO: Found total 1 card(s), 1 are usable
Total DDR size: 65536 MB
Reporting from mem_topology:
Data Validity & DMA Test on bank0
Host -> PCIe -> FPGA write bandwidth = 11341.5 MB/s
Host <- PCIe <- FPGA read bandwidth = 11097.3 MB/s
Data Validity & DMA Test on bank1
Host -> PCIe -> FPGA write bandwidth = 11414.6 MB/s
Host <- PCIe <- FPGA read bandwidth = 10981.7 MB/s
Data Validity & DMA Test on bank2
Host -> PCIe -> FPGA write bandwidth = 11345.1 MB/s
```

```
Host <- PCIe <- FPGA read bandwidth = 11189.2 MB/s
Data Validity & DMA Test on bank3
Host -> PCIe -> FPGA write bandwidth = 11121.7 MB/s
Host <- PCIe <- FPGA read bandwidth = 11375.7 MB/s
INFO: xbutil dmatetest succeeded.
```

dump

The `dump` command prints out device information in JSON format to the terminal.

The command has the following options:

- `-d <card>` (Optional): Specifies the target card. Default = 0 if not specified.

To run the `dump` command, run the following command:

```
$ xbutil dump
```

An example of the command output on a U250 card is shown below:

```
{
  "version": "1.1.0",
  "system": {
    "sysname": "Linux",
    "release": "4.15.0-43-generic",
    "version": "#46~16.04.1-Ubuntu SMP Fri Dec 7 13:31:08 UTC 2018",
    "machine": "x86_64",
    "glibc": "2.23",
    "linux": "Ubuntu 16.04.4 LTS",
    "now": "Tue May 21 14:13:00 2019"
  },
  "runtime": {
    "build": {
      "version": "2.3.0",
      "hash": "42f90bb223343431a63d520c036f03c28fff2550",
      "date": "2019-05-20 20:06:52",
      "branch": "master",
      "xocl": "2.3.0,42f90bb223343431a63d520c036f03c28fff2550",
      "xclmgmt": "2.3.0,42f90bb223343431a63d520c036f03c28fff2550"
    }
  },
  "board": {
    "info": {
      "dsa_name": "xilinx_u250_xdma_201830_2",
      "vendor": "4334",
      "device": "20485",
      "subdevice": "14",
      "subvendor": "4334",
      "xmversion": "0",
      "ddr_size": "68719476736",
      "ddr_count": "4",
      "clock0": "300",
      "clock1": "500",
      "clock2": "0",
      "pcie_speed": "3",
      "pcie_width": "16",
      "dma_threads": "2",
      "mig_calibrated": "false",
```

```

        "idcode": "0x4b57093",
        "fpga_name": "xcu250-figd2104-2L-e",
        "dna": "",
        "p2p_enabled": "0"
    },
    "physical": {
        "thermal": {
            "pcb": {
                "top_front": "40",
                "top_rear": "35",
                "btm_front": "42"
            },
            "fpga_temp": "48",
            "tcrit_temp": "41",
            "fan_speed": "1262",
            "cage": {
                "temp0": "0",
                "temp1": "0",
                "temp2": "0",
                "temp3": "0"
            }
        }
    },
    "electrical": {
        "12v_pex": {
            "voltage": "11960",
            "current": "2128"
        },
        "12v_aux": {
            "voltage": "468",
            "current": "0"
        },
        "3v3_pex": {
            "voltage": "3301"
        },
        "3v3_aux": {
            "voltage": "3307"
        },
        "ddr_vpp_bottom": {
            "voltage": "2500"
        },
        "ddr_vpp_top": {
            "voltage": "2500"
        },
        "sys_5v5": {
            "voltage": "5487"
        },
        "1v2_top": {
            "voltage": "1201"
        },
        "1v2_btm": {
            "voltage": "151"
        },
        "1v8_top": {
            "voltage": "1847"
        },
        "0v85": {
            "voltage": "855"
        },
        "mgt_0v9": {
            "voltage": "906"
        },
        "12v_sw": {
            "voltage": "11971"
        }
    }

```

```

    },
    "mgt_vtt": {
      "voltage": "1200"
    },
    "vccint": {
      "voltage": "850",
      "current": "8657"
    }
  },
  "power": "25"
},
"error": {
  "firewall": {
    "firewall_level": "0",
    "status": "(GOOD)"
  }
},
"pcie_dma": {
  "transfer_metrics": {
    "chan": {
      "0": {
        "h2c": "0 Byte",
        "c2h": "20 Byte"
      },
      "1": {
        "h2c": "0 Byte",
        "c2h": "0 Byte"
      }
    }
  }
},
"memory": {
  "mem": {
    "0": {
      "type": "MEM_DDR4",
      "temp": "4294967295",
      "tag": "bank0",
      "enabled": "true",
      "size": "16 GB",
      "mem_usage": "0 Byte",
      "bo_count": "0"
    },
    "1": {
      "type": "***UNUSED**",
      "temp": "4294967295",
      "tag": "bank1",
      "enabled": "false",
      "size": "16 GB",
      "mem_usage": "0 Byte",
      "bo_count": "0"
    },
    "2": {
      "type": "***UNUSED**",
      "temp": "4294967295",
      "tag": "bank2",
      "enabled": "false",
      "size": "16 GB",
      "mem_usage": "0 Byte",
      "bo_count": "0"
    },
    "3": {
      "type": "***UNUSED**",
      "temp": "4294967295",

```

```

        "tag": "bank3",
        "enabled": "false",
        "size": "16 GB",
        "mem_usage": "0 Byte",
        "bo_count": "0"
    },
    "4": {
        "type": "***UNUSED**",
        "temp": "4294967295",
        "tag": "PLRAM[0]",
        "enabled": "false",
        "size": "128 KB",
        "mem_usage": "0 Byte",
        "bo_count": "0"
    },
    "5": {
        "type": "***UNUSED**",
        "temp": "4294967295",
        "tag": "PLRAM[1]",
        "enabled": "false",
        "size": "128 KB",
        "mem_usage": "0 Byte",
        "bo_count": "0"
    },
    "6": {
        "type": "***UNUSED**",
        "temp": "4294967295",
        "tag": "PLRAM[2]",
        "enabled": "false",
        "size": "128 KB",
        "mem_usage": "0 Byte",
        "bo_count": "0"
    },
    "7": {
        "type": "***UNUSED**",
        "temp": "4294967295",
        "tag": "PLRAM[3]",
        "enabled": "false",
        "size": "128 KB",
        "mem_usage": "0 Byte",
        "bo_count": "0"
    }
}
},
"xclbin": {
    "uuid": "1e941bf2-3945-4951-8f67-7bd78664513d"
},
"compute_unit": {
    "0": {
        "name": "hello:hello_1",
        "base_address": "25165824",
        "status": "(IDLE)"
    }
}
},
"debug_profile": {
    "device_info": {
        "error": "0",
        "device_index": "0",
        "user_instance": "128",
        "nifd_instance": "0",
    }
}

```

```

        "device_name": "\\dev\\dri\\renderD128",
        "nifd_name": "\\dev\\nifd0"
    }
}
}

```

flash



IMPORTANT! *This option cannot be used with embedded processor platforms.*

The `flash` command programs the flash configuration memory on the card with a specified deployment shell.

It has the following options:

- `-d <card_>`

(Optional): Specifies the target card BDF, else flashes all cards if not specified.

Note: The Bus Device Function (BDF) is a set of numbers in the format `Bus:device.f` that identifies a particular device. This number can be found using the `lspci(8)` Linux command.

- `-a <all | shell_>`: Specifies the name of the deployment shell to program the card or you can set the `shell_name` to `all`. This will attempt to flash all the cards in the system with the installed deployment shell.
- `-t <timestamp_>`: Specifies the timestamp associated with the `shell_name`.

For example, to flash the card with a deployment shell called `xilinx_u200_xdma_201820_1` and timestamp `1535712995`, enter the following command:

```
sudo xbutil flash -a xilinx_u200_xdma_201820_1 -t 1535712995
```

Below is an example of the output after the card has been flashed:

```

INFO: ***Found 880 ELA Records
Idcode byte[0] ff
Idcode byte[1] 20
Idcode byte[2] bb
Idcode byte[3] 21
Idcode byte[4] 10
Enabled bitstream guard. Bitstream will not be loaded until flashing is
finished.
Erasing flash.....
Programming flash.....
Cleared bitstream guard. Bitstream now active.
DSA image flashed succesfully
Cold reboot machine to load the new image on FPGA

```

flash scan



IMPORTANT! *This option cannot be used with embedded processor platforms.*

The `flash scan` command returns the current firmware installed on both the card and the system.

It has the following option:

- `-v` (Optional): Verbose output displays additional information including the MAC address.

To run the `flash scan` command in verbose mode, enter the following:

```
sudo xbutil flash scan -v
```

You should see an output similar to the example below. In this example, the deployment shell name is `xilinx_u200_xdma_201830_1`, the timestamp is `0x000000005bece8e1`, and the BMC version is 3.1. In this output, platform is referring to the deployment shell, TS is the timestamp, and BMC is referring to the Satellite Controller.

```
XBFLASH -- Xilinx Card Flash Utility
Card_ID[0]
  Card BDF:                0000:d8:00.0
  Card type:               u200
  Flash type:              SPI
  Shell running on FPGA:
    xilinx_u200_xdma_201830_1, [TS=0x000000005bece8e1], [BMC=3.1]
  Shell package installed in system:
    xilinx_u200_xdma_201830_1, [TS=0x000000005bece8e1], [BMC=3.1]
  Card name                A S00A64G
  Card S/N:                2129048BF083
  Config mode:             7
  Fan presence:            A
  Max power level:         225W
  MAC address0:            00:0A:35:05:EC:5A
  MAC address1:            00:0A:35:05:EC:5B
  MAC address2:            FF:FF:FF:FF:FF:FF
  MAC address3:            FF:FF:FF:FF:FF:FF
```

help

The `help` command displays the available `xbutil` commands.

list

The `list` command lists all supported working cards installed on the system along with the card ID. The card ID is used in other `xbutil` commands or in your host code when specifying a particular card.

The output format displays the following three items in this order:

```
[card_id] BDF shell_name
```

There are no options for this command.

To run the `list` command, enter the following:

```
$ xbutil list
```

In this example, the card ID is 0, the BDF is 65:00.0, and the shell name is `xilinx_u250_xdma_201820_1`.

```
INFO: Found total 1 card(s), 1 are usable
[0] 65:00.0 xilinx_u250_xdma_201820_1
```

m2m



IMPORTANT! *This option cannot be used with embedded processor platforms.*

The `m2mtest` command performs throughput data transfer tests between two device memory banks on a specified card. Note, it is necessary to download an `xclbin` on the card which uses at least two memory banks prior to running `m2mtest`, else running this command returns an error.

The `m2mtest` command only performs throughput tests on those memory banks accessed by the `xclbin` downloaded to the card.

The command has the following option:

- `-d <card>` (Optional): Specifies the target card. Default = 0 if not specified.

To run the `dmatest` command, run the following command:

```
$ xbutil dmatest
```

An example of the command output with an `xclbin` using DDR banks 0, 1, 2, and 3 is shown below:

```
INFO: Found total 2 card(s), 2 are usable
bank0 -> bank1 M2M bandwidth: 12050.5 MB/s
bank0 -> bank2 M2M bandwidth: 12074.3 MB/s
bank0 -> bank3 M2M bandwidth: 12082.9 MB/s
bank1 -> bank2 M2M bandwidth: 12061.8 MB/s
bank1 -> bank3 M2M bandwidth: 12105.2 MB/s
bank2 -> bank3 M2M bandwidth: 12065.8 MB/s
INFO: xbutil m2mtest succeeded.
```

mem read

The `mem --read` command reads the specified number of bytes starting at a specified memory address and writes the contents into an output file.

- `-a <address>` (Optional): Specifies the starting address (in hexadecimal). Default address is `0x0`.

- `-i <size>`: Specifies the size of memory read (in bytes).
- `-o <file_name>` (Optional): Specifies the output file name. Default output file is `memread.out`.

To run the `mem --read` command to read 256 bytes of data starting at memory address `0x0`, enter the following:

```
$ xbutil mem --read -a 0x0 -i 256 -o read.out
```

This is an example output:

```
INFO: Found total 1 card(s), 1 are usable
INFO: Reading from single bank, 256 bytes from DDR address 0x4000000000
INFO: Read size 0x100 B. Total Read so far 0x100
INFO: Read data saved in file: read.out; Num of bytes: 256 bytes
INFO: xbutil mem succeeded.
```

mem write

The `mem --write` command writes a defined value to a specified memory location and size.

- `-a <address>` (Optional): Specifies the starting address (in hexadecimal). Default address is `0x0`.
- `-i <size>`: Specifies the size of memory read (in bytes).
- `-e <pattern>`: Specifies the pattern (in bytes) to write to memory.

To write the value `0xaa` to 256 locations starting at memory address `0x0`, enter the following:

```
$ xbutil mem --write -a 0x0 -i 256 -e 0xaa
```

This is an example output:

```
INFO: Found total 1 card(s), 1 are usable
INFO: Writing to single bank, 256 bytes from DDR address 0x4000000000
INFO: Writing DDR with 256 bytes of pattern: 0xaa from address 0x4000000000
INFO: xbutil mem succeeded.
```

p2p



IMPORTANT! *This option cannot be used with embedded processor platforms.*

The `p2p` command is used to enable/disable P2P feature and check current configuration. P2P configuration is persistent across warm reboot. Enabling or disabling P2P requires root privilege.

See [PCIe Peer-to-Peer Support](#) for more information.

program

The `program` command downloads an `xclbin` binary to the programmable region on the card.

It has the following options:

- `-d <card_id>` (Optional): Specifies the target card ID. Default = 0 if not specified.
- `-p <xclbin>` (Required): Specifies the `xclbin` binary file to download to the card.

For example, to program the `filter.xclbin` file to card ID one, you would use the following command:

```
$ xbutil program -d 1 -p filter.xclbin
```

This output is displayed after the `xclbin` has been successfully downloaded to the card:

```
INFO: Found total 1 card(s), 1 are usable
INFO: xbutil program succeeded.
```

query

The `query` command returns detailed status information for the specified card.

It has the following option:

- `-d <card_id>` (Optional): Specifies the target card. Default = 0 if not specified.

For example, to query card ID zero, run the following command:

```
xbutil query -d 0
```

An example of the output is given below. The output has been divided into separate sections to better describe the content.

The first section gives details of the installed card including the shell name (platform name), vendor information, installed DDR, clock, and PCIe information.

```
INFO: Found total 1 card(s), 1 are usable
platform name
xilinx_u250_xdma_201820_1

Vendor          Device          SubDevice        SubVendor        XMC fw
version
10ee           5004            000e             10ee
2018203

DDR size        DDR count        OCL Frequency    Clock0
Clock1
64 GB          4                300 MHz          500
```

```
MHz
PCIe          DMA bi-directional threads  MIG Calibrated
GEN 3x16      2                          true
```

Card power and thermal information are given next.

```
#####
###
Power
34.9W

PCB TOP FRONT  PCB TOP REAR  PCB BTM FRONT
33 C           28 C           32 C

FPGA Temp      TCRIT Temp     Fan Speed
35 C           33 C           1100 rpm

12V PEX        12V AUX        12V PEX Current 12V AUX Current
11.9V          0.45V          2928mA          32mA

3V3 PEX        3V3 AUX        DDR VPP BOTTOM  DDR VPP TOP
3.36V          3.31V          2.50V           2.50V

SYS 5V5        1V2 TOP        1V8 TOP        0V85
5.49V          1.20V          1.82V           0.85V

MGT 0V9        12V SW         MGT VTT
0.90V          11.9V          1.20V

VCCINT VOL     VCCINT CURR
0.85V          10094mA
```

The firewall provides information when an error has been detected in hardware. This includes a timestamp and the level of the firewall. The firewall has three levels, as discussed in [AXI Firewall Trips](#). In the following output, there are no detected firewall errors.

```
Firewall Last Error Status:
Level 0: 0x0 (GOOD)
```

The `xclbin` ID along with the contained Compute Units (CU) are displayed. For each CU, it displays the name, PCIe BAR address, and the status, which can be IDLE, START, and DONE. The output below shows the `xclbin` ID and two CUs both with IDLE status.

```
Xclbin ID:
0x5b996b13

Compute Unit Status:
CU[0]: bandwidth1:kernel_1@0x1800000 (IDLE)
CU[1]: bandwidth2:kernel_2@0x1810000 (IDLE)
```

The memory topology along with the DMA transfer metrics are provided next. The DMA metrics include the transfer of data between the host and card. Host to card transfers are indicated by `h2c`, while card to host transfer are defined by `c2h`.

```
#####
###
Mem Topology                               Device Memory
Usage
Tag      Type      Temp      Size      Mem Usage      BO nums
[0] bank0 MEM_DDR4    31 C      16 GB     0 Byte         0
[1] bank1 MEM_DDR4    31 C      16 GB     0 Byte         0
[2] bank2 MEM_DDR4    33 C      16 GB     0 Byte         0
[3] bank3 MEM_DDR4    31 C      16 GB     0 Byte         0
[4] PLRAM[0] **UNUSED** Not Supp   128 KB     0 Byte         0
[5] PLRAM[1] **UNUSED** Not Supp   128 KB     0 Byte         0
[6] PLRAM[2] **UNUSED** Not Supp   128 KB     0 Byte         0
[7] PLRAM[3] **UNUSED** Not Supp   128 KB     0 Byte         0

Total DMA Transfer Metrics:
Chan[0].h2c: 49888 MB
Chan[0].c2h: 22656 MB
Chan[1].h2c: 8096 MB
Chan[1].c2h: 22592 MB
```

Finally, here is the successful output:

```
INFO: xbutil query succeeded.
```

reset



IMPORTANT! *This option cannot be used with embedded processor platforms.*

The `reset` command resets the programmable region on the card. All running compute units in the region are stopped and reset.

It has the following options:

- `-d <card_id>` (Optional): Specifies the target card ID number. Default = 0 if not specified.
- `-h` (Optional): Performs a hot-reset which resets the card and not just the programmable region. The card is still recognized by the operating system. It is recommended to always use this option.

Enter the following command:

```
$ xbutil reset
```

This output is displayed after the reset has been successfully completed:

```
INFO: Found total 1 card(s), 1 are usable
INFO: xbutil reset succeeded.
```

scan



IMPORTANT! *This option cannot be used with embedded processor platforms.*

The `scan` option scans the system, displays drivers, and system information.

It has no options.

To run the `scan` command, enter the following:

```
$ xbutil scan
```

An example of the output is shown below:

```
Linux:4.15.0-33-generic:#36~16.04.1-Ubuntu SMP Wed Aug 15 17:21:05 UTC
2018:x86_64
Distribution: Ubuntu 16.04.5 LTS
GLIBC: 2.23
---
XILINX_OPENCL= " "
LD_LIBRARY_PATH="/opt/xilinx/xrt/lib:"
---
[0]mgmt:[65:00.1]:0x5004:0x000e:[xclmgmt:2018.3.2:25857]
[0]user:[65:00.0]:0x5005:0x000e:[xocl_xdma:2018.3.8:128]
```

status

The `status` command displays the status of the debug IPs on the card. Currently, this command can read and report the status of Vitis™ performance monitor (SPM) and lightweight AXI protocol checker (LAPC) debug IPs. For more information on adding SPM counters and LAPC in your design, see [Application Hangs](#).

Below are the available options. If you are running without arguments, it shows the list of available debug IPs:

- `--spm` (Optional): Returns the value of the SPM counters. This option is only applicable if the `xclbin` was compiled with the necessary profiling options.
- `--lapc` (Optional): Returns the values of the violation codes detected by the LAPC. This option is only applicable if `xclbin` was compiled with necessary option to insert AXI protocol checkers at the AXI ports of the compute units.

An example output of the following command is shown below:

```
$ xbutil status

INFO: Found total 1 card(s), 1 are usable
Number of IPs found: 6
IPs found [<ipname><count>]: spm(2) tracefunnel(1) monitorfifolite(1)
monitorfifofull(1) accelmonitor(1)
Run 'xbutil status' with option --<ipname> to get more information about
the IP
INFO: xbutil status succeeded.
```

An example output using the `--spm` option is shown below:

```
$ xbutil status --spm

INFO: Found total 1 card(s), 1 are usable
Vitis Performance Monitor Counters
CU Name          AXI Portname  Write Bytes  Write Trans.
interconnect_aximm_host M00_AXI      8192         16
simple_1          M_AXI_GMEM   4096         1024

CU Name          Read Bytes  Read Trans.  Outstanding Cnt
interconnect_aximm_host 4096        1             0
simple_1          4096        1024         0

CU Name          Last Wr Addr  Last Wr Data  Last Rd Addr
interconnect_aximm_host 0x0           0             0xe00
simple_1          0x0           0             0xffc

CU Name          Last Rd Data
interconnect_aximm_host 1483476076
simple_1          1062897
INFO: xbutil status succeeded.
```

When there are no debug IPs in the `xclbin`, you will see a similar output as shown below:

```
INFO: Found total 1 card(s), 1 are usable
INFO: Failed to find any debug IPs on the platform. Ensure that a valid
bitstream with debug IPs (SPM, LAPC) is successfully downloaded.
INFO: xbutil status succeeded.
```

top



IMPORTANT! *This option cannot be used with embedded processor platforms.*

The `top` command outputs card statistics including memory topology and DMA transfer metrics. This command is similar to the Linux `top` command. When running, it continues to operate until `q` is entered in the terminal window.

It has the following option:

- `-i <seconds>` (Optional): Refreshes rate (in seconds). Default is 1 second.

To run `top` with a refresh rate of two seconds, enter the following command:

```
$ xbutil top -i 2
```

An output similar to the one below is displayed:

```
Device Memory Usage
[0] bank0 [ 0.00% ]
[1] bank1 [ 0.00% ]
[2] bank2 [ 0.00% ]
[3] bank3 [ 0.00% ]
[4] PLRAM0 [ 0.00% ]
[5] PLRAM1 [ 0.00% ]
[6] PLRAM2 [ 0.00% ]

Power
25.0W

Mem Topology
Tag Device Memory Usage
Type Temp Size Mem Usage Bo nums
[0] bank0 **UNUSED** 32 C 16 GB 0 Byte 0
[1] bank1 MEM_DDR4 37 C 17 GB 0 Byte 0
[2] bank2 **UNUSED** 34 C 18 GB 0 Byte 0
[3] bank3 **UNUSED** 32 C 19 GB 0 Byte 0
[4] PLRAM0 **UNUSED** Not Supp 128 KB 0 Byte 0
[5] PLRAM1 **UNUSED** Not Supp 128 KB 0 Byte 0
[6] PLRAM2 **UNUSED** Not Supp 128 KB 0 Byte 0

Total DMA Transfer Metrics:
Chan[0].h2c: 0 Byte
Chan[0].c2h: 0 Byte
Chan[1].h2c: 0 Byte
Chan[1].c2h: 0 Byte
```

validate



IMPORTANT! *This option cannot be used with embedded processor platforms.*

The `validate` command generates a high-level, easy to read summary of the installed card. It validates correct installation by performing the following set of tests:

1. Validates the card found.
2. Checks PCI Express link status.
3. Runs a verify kernel on the card.
4. Performs the following data bandwidth tests:
 - a. DMA test: Data transfers between host and FPGA DDR through PCI Express.
 - b. DDR test: Data transfers between kernels and FPGA DDR.

It has the following option:

- `-d <card_id>` (Optional): Specifies the target card ID. Default validates all the cards installed in the system.

For example, to run the `validate` command on card ID = 0, enter the following:

```
$ xbutil validate -d 0
```

An example of the returned information is shown below:

```
INFO: Found 1 cards

INFO: Validating card[0]: xilinx_u250_xdma_201820_1
INFO: Checking PCIE link status: PASSED
INFO: Starting verify kernel test:
INFO: verify kernel test PASSED
INFO: Starting DMA test
Host -> PCIe -> FPGA write bandwidth = 11736.3 MB/s
Host <- PCIe <- FPGA read bandwidth = 12190.3 MB/s
INFO: DMA test PASSED
INFO: Starting DDR bandwidth test: .....
Maximum throughput: 45475.441406 MB/s
INFO: DDR bandwidth test PASSED
INFO: Card[0] validated successfully.

INFO: All cards validated successfully.
```

package_xo Command

Syntax

```
package_xo -kernel_name <arg> [-force] [-kernel_xml <arg>] [-design_xml
<arg>]
          [-ip_directory <arg>] [-parent_ip_directory <arg>]
          [-kernel_files <args>] [-kernel_xml_args <args>]
          [-kernel_xml_pipes <args>] [-kernel_xml_connections <args>]
          -xo_path <arg> [-quiet] [-verbose]
```

Description

The `package_xo` command is a Tcl command within the Vivado Design Suite. Kernels written in RTL are compiled in the Vivado tool using the `package_xo` command line utility which generates a Xilinx object file (`.xo`) which can subsequently be used by the `v++` command, during the linking stage.

Table 24: Arguments

Argument	Description
-kernel_name <arg>	Required. Specifies the name of the RTL kernel.
-force	(Optional) Overwrite an existing .xo file if one exists.
-kernel_xml <arg>	(Optional) Specify the path to an existing kernel XML file.
-design_xml <arg>	(Optional) Specify the path to an existing design XML file
-ip_directory <arg>	(Optional) Specify the path to the kernel IP directory.
-parent_ip_directory	(Optional) If the kernel IP directory specified contains multiple IPs, specify a directory path to the parent IP where its component.xml is located directly below.
-kernel_files	(Optional) Kernel file name(s).
-kernel_xml_args <args>	(Optional) Generate the kernel.xml with the specified function arguments. Each argument value should use the following format: <pre>{name:addressQualifier:id:port:size:offset:type:memSize}</pre> Note: memSize is optional.
-kernel_xml_pipes <args>	(Optional) Generate the kernel.xml with the specified pipe(s). Each pipe value use the following format: <pre>{name:width:depth}</pre>
-kernel_xml_connections <args>	(Optional) Generate the kernel.xml file with the specified connections. Each connection value should use the following format: <pre>{srcInst:srcPort:dstInst:dstPort}</pre>
-xo_path <arg>	(Required) Specifies the path and file name of the compiled object (.xo) file.
-quiet	(Optional) Execute the command quietly, returning no messages from the command. The command also returns TCL_OK regardless of any errors encountered during execution. Note: Any errors encountered on the command-line, while launching the command, will be returned. Only errors occurring inside the command will be trapped.
-verbose	(Optional) Temporarily override any message limits and return all messages from this command. Note: Message limits can be defined with the set_msg_config command.

Examples

The following example creates the specified `.xo` file containing an RTL kernel of the specified name:

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

HLS Pragmas

Optimizations in Vivado HLS

In the Vitis software platform, a kernel defined in the C/C++ language, or OpenCL™ C, must be compiled into the register transfer level (RTL) that can be implemented into the programmable logic of a Xilinx device. The `v++` compiler calls the Vivado® High-Level Synthesis (HLS) tool to synthesize the RTL code from the kernel source code.

The HLS tool is intended to work with the Vitis IDE project without interaction. However, the HLS tool also provides pragmas that can be used to optimize the design, reduce latency, improve throughput performance, and reduce area and device resource usage of the resulting RTL code. These pragmas can be added directly to the source code for the kernel.

The HLS pragmas include the optimization types specified in the following table. For more information on the Vivado HLS tool, and the operation of these pragmas, refer to *Vivado Design Suite User Guide: High-Level Synthesis* (UG902).

Table 25: Vivado HLS Pragmas by Type

Type	Attributes
Kernel Optimization	<ul style="list-style-type: none"> pragma HLS allocation pragma HLS expression_balance pragma HLS latency pragma HLS reset pragma HLS resource pragma HLS top
Function Inlining	<ul style="list-style-type: none"> pragma HLS inline pragma HLS function_instantiate
Interface Synthesis	<ul style="list-style-type: none"> pragma HLS interface
Task-level Pipeline	<ul style="list-style-type: none"> pragma HLS dataflow pragma HLS stream
Pipeline	<ul style="list-style-type: none"> pragma HLS pipeline pragma HLS occurrence

Table 25: Vivado HLS Pragas by Type (cont'd)

Type	Attributes
Loop Unrolling	<ul style="list-style-type: none"> pragma HLS unroll pragma HLS dependence
Loop Optimization	<ul style="list-style-type: none"> pragma HLS loop_flatten pragma HLS loop_merge pragma HLS loop_tripcount
Array Optimization	<ul style="list-style-type: none"> pragma HLS array_map pragma HLS array_partition pragma HLS array_reshape
Structure Packing	<ul style="list-style-type: none"> pragma HLS data_pack

pragma HLS allocation

Description

Specifies instance restrictions to limit resource allocation in the implemented kernel. This defines and can limit the number of RTL instances and hardware resources used to implement specific functions, loops, operations, or cores. The `ALLOCATION` pragma is specified inside the body of a function, a loop, or a region of code.

For example, if the C source has four instances of a function `foo_sub`, the `ALLOCATION` pragma can ensure that there is only one instance of `foo_sub` in the final RTL. All four instances of the C function are implemented using the same RTL block. This reduces resources used by the function, but negatively impacts performance.

The operations in the C code, such as additions, multiplications, array reads, and writes, can be limited by the `ALLOCATION` pragma. Cores, which operators are mapped to during synthesis, can be limited in the same manner as the operators. Instead of limiting the total number of multiplication operations, you can choose to limit the number of combinational multiplier cores, forcing any remaining multiplications to be performed using pipelined multipliers (or conversely).

The `ALLOCATION` pragma applies to the scope it is specified within: a function, a loop, or a region of code. However, you can use the `-min_op` argument of the `config_bind` command to globally minimize operators throughout the design.



TIP: For more information, refer to "Controlling Hardware Resources" and `config_bind` in Vivado Design Suite User Guide: High-Level Synthesis (UG902).

Syntax

Place the pragma inside the body of the function, loop, or region where it will apply.

```
#pragma HLS allocation instances=<list> \
limit=<value> <type>
```

Where:

- `instances=<list>`: Specifies the names of functions, operators, or cores.
- `limit=<value>`: Optionally specifies the limit of instances to be used in the kernel.
- `<type>`: Specifies that the allocation applies to a function, an operation, or a core (hardware component) used to create the design, such as adders, multipliers, pipelined multipliers, and block RAM. The type is specified as one of the following:
 - `function`: Specifies that the allocation applies to the functions listed in the `instances=` list. The function can be any function in the original C or C++ code that has *not* been:
 - Inlined by the `pragma HLS inline`, or the `set_directive_inline` command, or
 - Inlined automatically by the Vivado HLS tool.
 - `operation`: Specifies that the allocation applies to the operations listed in the `instances=` list. For a complete list of operations that can be limited using the `ALLOCATION` pragma, refer to *Vivado Design Suite User Guide: High-Level Synthesis (UG902)*.
 - `core`: Specifies that the `ALLOCATION` applies to the cores, which are the specific hardware components used to create the design (such as adders, multipliers, pipelined multipliers, and block RAM). The actual core to use is specified in the `instances=` option. In the case of cores, you can specify which the tool should use, or you can define a limit for the specified core.

Example 1

Given a design with multiple instances of function `foo`, this example limits the number of instances of `foo` in the RTL for the hardware kernel to two.

```
#pragma HLS allocation instances=foo limit=2 function
```

Example 2

Limits the number of multiplier operations used in the implementation of the function `my_func` to one. This limit does not apply to any multipliers outside of `my_func`, or multipliers that might reside in sub-functions of `my_func`.



TIP: To limit the multipliers used in the implementation of any sub-functions, specify an allocation directive on the sub-functions or inline the sub-function into function `my_func`.

```
void my_func(data_t angle) {
    #pragma HLS allocation instances=mul limit=1 operation
    ...
}
```

See Also

- [pragma HLS function_instantiate](#)
- [pragma HLS inline](#)

pragma HLS array_map

Description

Combines multiple smaller arrays into a single large array to help reduce block RAM resources.

Designers typically use the `pragma HLS array_map` command (with the same `instance=target`) to combine multiple smaller arrays into a single larger array. This larger array can then be targeted to a single larger memory (RAM or FIFO) resource.

Each array is mapped into a block RAM or UltraRAM, when supported by the device. The basic block RAM unit provided in an FPGA is 18K. If many small arrays do not use the full 18K, a better use of the block RAM resources is to map many small arrays into a single larger array.



TIP: If a block RAM is larger than 18K, they are automatically mapped into multiple 18K units.

The `ARRAY_MAP` pragma supports two ways of mapping small arrays into a larger one:

- **Horizontal:** Corresponds to creating a new array by concatenating the original arrays. Physically, this gets implemented as a single array with more elements.
- **Vertical:** Corresponds to creating a new array by concatenating the original arrays. Physically, this gets implemented as a single array with more elements.

The arrays are concatenated in the order that the pragmas are specified, starting at:

- Target element zero for horizontal mapping, or
- Bit zero for vertical mapping.

Syntax

Place the pragma in the C source within the boundaries of the function where the array variable is defined.

```
#pragma HLS array_map variable=<name> instance=<instance> \
<mode> offset=<int>
```

Where:

- `variable=<name>`: A required argument that specifies the array variable to be mapped into the new target array `<instance>`.
- `instance=<instance>`: Specifies the name of the new array to merge arrays into.
- `<mode>`: Optionally specifies the array map as being either `horizontal` or `vertical`.
 - Horizontal mapping is the default `<mode>`, and concatenates the arrays to form a new array with more elements. Remapping the original N arrays will require N cycles with 1 port block RAM, or ceiling (N/2) cycles with a 2 port block RAM.
 - Vertical mapping concatenates the array to form a new array with longer words. Remapping the original N arrays is similar to the horizontal mapping above except when the same index is used: this will require only 1 cycle.
- `offset=<int>`: Applies to horizontal type array mapping only. The offset specifies an integer value offset to apply before mapping the array into the new array `<instance>`. For example:
 - Element 0 of the array variable maps to element `<int>` of the new target.
 - Other elements map to `<int+1>`, `<int+2>`... of the new target.



IMPORTANT! *If an offset is not specified, the Vivado HLS tool calculates the required offset automatically to avoid overlapping array elements.*

Example 1

Arrays `array1` and `array2` in function `foo` are mapped into a single array, specified as `array3` in the following example:

```
void foo (...) {
    int8 array1[M];
    int12 array2[N];
    #pragma HLS ARRAY_MAP variable=array1 instance=array3 horizontal
    #pragma HLS ARRAY_MAP variable=array2 instance=array3 horizontal
    ...
    loop_1: for(i=0;i<M;i++) {
        array1[i] = ...;
        array2[i] = ...;
        ...
    }
    ...
}
```

Example 2

This example provides a horizontal mapping of array A[10] and array B[15] in function `foo` into a single new array AB[25].

- Element AB[0] will be the same as A[0].
- Element AB[10] will be the same as B[0] because no `offset=` option is specified.
- The bit-width of array AB[25] will be the maximum bit-width of either A[10] or B[15].

```
#pragma HLS array_map variable=A instance=AB horizontal
#pragma HLS array_map variable=B instance=AB horizontal
```

Example 3

The following example performs a vertical concatenation of arrays C and D into a new array CD, with the bit-width of C and D combined. The number of elements in CD is the maximum of the original arrays, C or D:

```
#pragma HLS array_map variable=C instance=CD vertical
#pragma HLS array_map variable=D instance=CD vertical
```

See Also

- [pragma HLS array_partition](#)
- [pragma HLS array_reshape](#)

pragma HLS array_partition

Description

Partitions an array into smaller arrays or individual elements and provides the following:

- Results in RTL with multiple small memories or multiple registers instead of one large memory.
- Effectively increases the amount of read and write ports for the storage.
- Potentially improves the throughput of the design.
- Requires more memory instances or registers.

Syntax

Place the pragma in the C source within the boundaries of the function where the array variable is defined.

```
#pragma HLS array_partition variable=<name> \
<type> factor=<int> dim=<int>
```

Where:

- `variable=<name>`: A required argument that specifies the array variable to be partitioned.
- `<type>`: Optionally specifies the partition type. The default type is `complete`. The following types are supported:
 - `cyclic`: Cyclic partitioning creates smaller arrays by interleaving elements from the original array. The array is partitioned cyclically by putting one element into each new array before coming back to the first array to repeat the cycle until the array is fully partitioned. For example, if `factor=3` is used:
 - Element 0 is assigned to the first new array
 - Element 1 is assigned to the second new array.
 - Element 2 is assigned to the third new array.
 - Element 3 is assigned to the first new array again.
 - `block`: Block partitioning creates smaller arrays from consecutive blocks of the original array. This effectively splits the array into N equal blocks, where N is the integer defined by the `factor=` argument.
 - `complete`: Complete partitioning decomposes the array into individual elements. For a one-dimensional array, this corresponds to resolving a memory into individual registers. This is the default `<type>`.
- `factor=<int>`: Specifies the number of smaller arrays that are to be created.



IMPORTANT! For complete type partitioning, the factor is not specified. For block and cyclic partitioning, the `factor=` is required.

- `dim=<int>`: Specifies which dimension of a multi-dimensional array to partition. Specified as an integer from 0 to `<N>`, for an array with `<N>` dimensions:
 - If a value of 0 is used, all dimensions of a multi-dimensional array are partitioned with the specified type and factor options.
 - Any non-zero value partitions only the specified dimension. For example, if a value 1 is used, only the first dimension is partitioned.

Example 1

This example partitions the 13 element array, `AB[13]`, into four arrays using block partitioning:

```
#pragma HLS array_partition variable=AB block factor=4
```



TIP: Because four is not an integer factor of 13:

- Three of the new arrays have three elements each

- One array has four elements (AB[9:12])

Example 2

This example partitions dimension two of the two-dimensional array, AB[6][4] into two new arrays of dimension [6][2]:

```
#pragma HLS array_partition variable=AB block factor=2 dim=2
```

Example 3

This example partitions the second dimension of the two-dimensional `in_local` array into individual elements.

```
int in_local[MAX_SIZE][MAX_DIM];
#pragma HLS ARRAY_PARTITION variable=in_local complete dim=2
```

See Also

- [pragma HLS array_map](#)
- [pragma HLS array_reshape](#)

pragma HLS array_reshape

Description

Combines array partitioning with vertical array mapping.

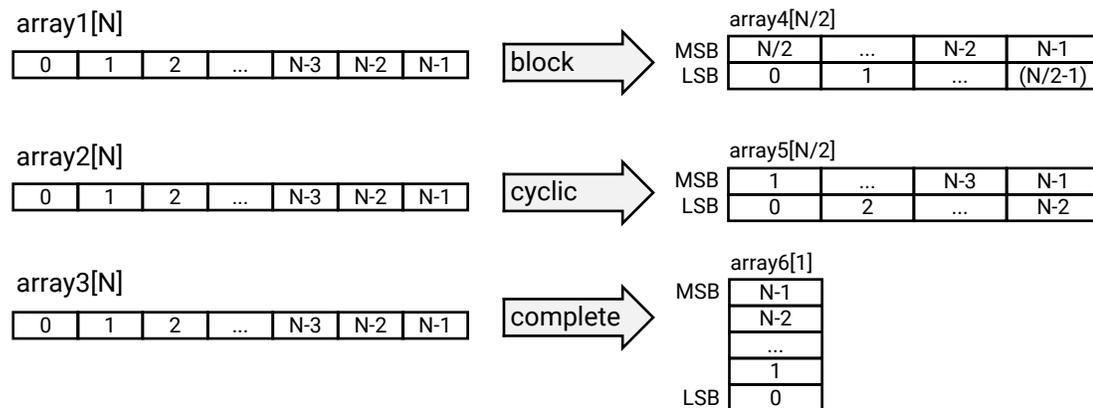
The `ARRAY_RESHAPE` pragma combines the effect of `ARRAY_PARTITION`, breaking an array into smaller arrays, with the effect of the vertical type of `ARRAY_MAP`, concatenating elements of arrays by increasing bit-widths. This reduces the number of block RAM consumed while providing the primary benefit of partitioning: parallel access to the data. This pragma creates a new array with fewer elements but with greater bit-width, allowing more data to be accessed in a single clock cycle.

Given the following code:

```
void foo (...) {
    int array1[N];
    int array2[N];
    int array3[N];
    #pragma HLS ARRAY_RESHAPE variable=array1 block factor=2 dim=1
    #pragma HLS ARRAY_RESHAPE variable=array2 cycle factor=2 dim=1
    #pragma HLS ARRAY_RESHAPE variable=array3 complete dim=1
    ...
}
```

The `ARRAY_RESHAPE` pragma transforms the arrays into the form shown in the following figure:

Figure 93: ARRAY_RESHAPE Pragma



X14307-110217

Syntax

Place the pragma in the C source within the region of a function where the array variable is defines.

```
#pragma HLS array_reshape variable=<name> \
<type> factor=<int> dim=<int>
```

Where:

- **<name>:** Required argument that specifies the array variable to be reshaped.
- **<type>:** Optionally specifies the partition type. The default type is `complete`. The following types are supported:
 - **cyclic:** Cyclic reshaping creates smaller arrays by interleaving elements from the original array. For example, if `factor=3` is used, element 0 is assigned to the first new array, element 1 to the second new array, element 2 is assigned to the third new array, and then element 3 is assigned to the first new array again. The final array is a vertical concatenation (word concatenation, to create longer words) of the new arrays into a single array.
 - **block:** Block reshaping creates smaller arrays from consecutive blocks of the original array. This effectively splits the array into `<N>` equal blocks where `<N>` is the integer defined by `factor=`, and then combines the `<N>` blocks into a single array with `word-width*N`.
 - **complete:** Complete reshaping decomposes the array into temporary individual elements and then recombines them into an array with a wider word. For a one-dimension array this is equivalent to creating a very-wide register (if the original array was `N` elements of `M` bits, the result is a register with `N*M` bits). This is the default type of array reshaping.

- `factor=<int>`: Specifies the amount to divide the current array by (or the number of temporary arrays to create). A factor of 2 splits the array in half, while doubling the bit-width. A factor of 3 divides the array into three, with triple the bit-width.



IMPORTANT! For complete type partitioning, the `factor` is not specified. For block and cyclic reshaping, the `factor=` is required.

- `dim=<int>`: Specifies which dimension of a multi-dimensional array to partition. Specified as an integer from 0 to $<N>$, for an array with $<N>$ dimensions:
 - If a value of 0 is used, all dimensions of a multi-dimensional array are partitioned with the specified type and factor options.
 - Any non-zero value partitions only the specified dimension. For example, if a value 1 is used, only the first dimension is partitioned.
- `object`: A keyword relevant for container arrays only. When the keyword is specified the `ARRAY_RESHAPE` pragma applies to the objects in the container, reshaping all dimensions of the objects within the container, but all dimensions of the container itself are preserved. When the keyword is not specified the pragma applies to the container array and not the objects.

Example 1

Reshapes (partition and maps) an 8-bit array with 17 elements, `AB[17]`, into a new 32-bit array with five elements using block mapping.

```
#pragma HLS array_reshape variable=AB block factor=4
```



TIP: `factor=4` indicates that the array should be divided into four; this means that 17 elements are reshaped into an array of five elements, with four times the bit-width. In this case, the last element, `AB[17]`, is mapped to the lower eight bits of the fifth element, and the rest of the fifth element is empty.

Example 2

Reshapes the two-dimensional array `AB[6][4]` into a new array of dimension `[6][2]`, in which dimension 2 has twice the bit-width.

```
#pragma HLS array_reshape variable=AB block factor=2 dim=2
```

Example 3

Reshapes the three-dimensional 8-bit array, `AB[4][2][2]` in function `foo`, into a new single element array (a register), 128-bits wide ($4 \times 2 \times 2 \times 8$).

```
#pragma HLS array_reshape variable=AB complete dim=0
```



TIP: `dim=0` means to reshape all dimensions of the array.

See Also

- [pragma HLS array_map](#)
- [pragma HLS array_partition](#)

pragma HLS data_pack

Description

Packs the data fields of a struct into a single scalar with a wider word width.

The `DATA_PACK` pragma is used for packing all the elements of a struct into a single wide vector to reduce the memory required for the variable, while allowing all members of the struct to be read and written to simultaneously. The bit alignment of the resulting new wide-word can be inferred from the declaration order of the struct fields. The first field takes the LSB of the vector, and the final element of the struct is aligned with the MSB of the vector.

If the struct contains arrays, the `DATA_PACK` pragma performs a similar operation as the `ARRAY_RESHAPE` pragma and combines the reshaped array with the other elements in the struct. Any arrays declared inside the struct are completely partitioned and reshaped into a wide scalar and packed with other scalar fields. However, a struct cannot be optimized with `DATA_PACK` and `ARRAY_PARTITION` or `ARRAY_RESHAPE`, as those pragmas are mutually exclusive.



IMPORTANT! You should exercise some caution when using the `DATA_PACK` optimization on struct objects with large arrays. If an array has 4096 elements of type `int`, this will result in a vector (and port) of width $4096 \times 32 = 131072$ bits. The Vivado HLS tool can create this RTL design, however it is very unlikely logic synthesis will be able to route this during the FPGA implementation.

In general, Xilinx recommends that you use arbitrary precision (or bit-accurate) data types. Standard C types are based on 8-bit boundaries (8-bit, 16-bit, 32-bit, and 64-bit); however, using arbitrary precision data types in a design lets you specify the exact bit-sizes in the C code prior to synthesis. The bit-accurate widths result in hardware operators that are smaller and faster. This allows more logic to be placed in the FPGA and for the logic to execute at higher clock frequencies. However, the `DATA_PACK` pragma also lets you align data in the packed struct along 8-bit boundaries, if needed.

If a `struct` port is to be implemented with an AXI4 interface you should consider using the `DATA_PACK <byte_pad>` option to automatically align member elements of the `struct` to 8-bit boundaries. The AXI4-Stream protocol requires that `TDATA` ports of the IP have a width in multiples of 8. It is a specification violation to define an AXI4-Stream IP with a `TDATA` port width that is not a multiple of 8, therefore, it is a requirement to round up `TDATA` widths to byte multiples. Refer to "Interface Synthesis and Structs" in *Vivado Design Suite User Guide: High-Level Synthesis (UG902)* for more information.

Syntax

Place the pragma near the definition of the struct variable to pack:

```
#pragma HLS data_pack variable=<variable> \
instance=<name> <byte_pad>
```

Where:

- **variable=<variable>**: Specifies the variable to be packed.
- **instance=<name>**: Specifies the name of resultant variable after packing. If no <name> is specified, the input <variable> is used.
- **<byte_pad>**: Optionally specifies whether to pack data on an 8-bit boundary (8-bit, 16-bit, 24-bit, etc.). The two supported values for this option are:
 - **struct_level**: Pack the whole `struct` first, then pad it upward to the next 8-bit boundary.
 - **field_level**: First pad each individual element (field) of the `struct` on an 8-bit boundary, then pack the `struct`.



TIP: Deciding whether multiple fields of data should be concatenated together before (*field_level*) or after (*struct_level*) alignment to byte boundaries is generally determined by considering how atomic the data is. Atomic information is data that can be interpreted on its own, whereas non-atomic information is incomplete for the purpose of interpreting the data. For example, atomic data can consist of all the bits of information in a floating point number. However, the exponent bits in the floating point number alone would not be atomic. When packing information into *TDATA*, generally non-atomic bits of data are concatenated together (regardless of bit width) until they form atomic units. The atomic units are then aligned to byte boundaries using pad bits where necessary.

Example 1

Packs struct array `AB[17]` with three 8-bit field fields (R, G, B) into a new 17 element array of 24-bits.

```
typedef struct{
    unsigned char R, G, B;
} pixel;

pixel AB[17];
#pragma HLS data_pack variable=AB
```

Example 2

Packs struct pointer `AB` with three 8-bit fields (typedef struct {unsigned char R, G, B;} pixel) in function `foo`, into a new 24-bit pointer.

```
typedef struct{
unsigned char R, G, B;
} pixel;

pixel AB;
#pragma HLS data_pack variable=AB
```

Example 3

In this example the `DATA_PACK` pragma is specified for `in` and `out` arguments to `rgb_to_hsv` function to instruct the compiler to do pack the structure on an 8-bit boundary to improve the memory access:

```
void rgb_to_hsv(RGBcolor* in, // Access global memory as RGBcolor struct-
wise
               HSVcolor* out, // Access Global Memory as HSVcolor struct-
wise
               int size) {
#pragma HLS data_pack variable=in struct_level
#pragma HLS data_pack variable=out struct_level
...
}
```

See Also

- [pragma HLS array_partition](#)
- [pragma HLS array_reshape](#)

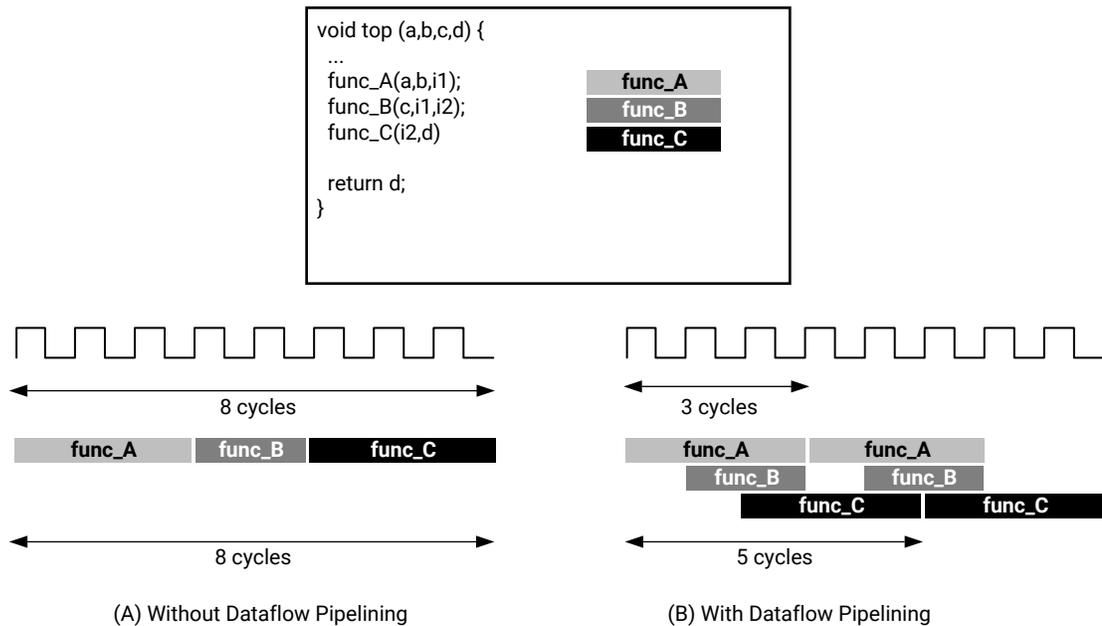
pragma HLS dataflow

Description

The `DATAFLOW` pragma enables task-level pipelining, allowing functions and loops to overlap in their operation, increasing the concurrency of the RTL implementation, and increasing the overall throughput of the design.

All operations are performed sequentially in a C description. In the absence of any directives that limit resources (such as `pragma HLS allocation`), the Vivado HLS tool seeks to minimize latency and improve concurrency. However, data dependencies can limit this. For example, functions or loops that access arrays must finish all read/write accesses to the arrays before they complete. This prevents the next function or loop that consumes the data from starting operation. The `DATAFLOW` optimization enables the operations in a function or loop to start operation before the previous function or loop completes all its operations.

Figure 94: DATAFLOW Pragma



X14266-110217

When the `DATAFLOW` pragma is specified, the HLS tool analyzes the dataflow between sequential functions or loops and creates channels (based on ping pong RAMs or FIFOs) that allow consumer functions or loops to start operation before the producer functions or loops have completed. This allows functions or loops to operate in parallel, which decreases latency and improves the throughput of the RTL.

If no initiation interval (number of cycles between the start of one function or loop and the next) is specified, the HLS tool attempts to minimize the initiation interval and start operation as soon as data is available.



TIP: The `config_dataflow` command specifies the default memory channel and FIFO depth used in dataflow optimization. For more information, refer to the `config_dataflow` command in the Vivado Design Suite User Guide: High-Level Synthesis ([UG902](#)).

For the `DATAFLOW` optimization to work, the data must flow through the design from one task to the next. The following coding styles prevent the HLS tool from performing the `DATAFLOW` optimization:

- Single-producer-consumer violations
- Bypassing tasks
- Feedback between tasks
- Conditional execution of tasks
- Loops with multiple exit conditions



IMPORTANT! If any of these coding styles are present, the HLS tool issues a message and does not perform `DATAFLOW` optimization.

Finally, the `DATAFLOW` optimization has no hierarchical implementation. If a sub-function or loop contains additional tasks that might benefit from the `DATAFLOW` optimization, you must apply the optimization to the loop, the sub-function, or inline the sub-function.

Syntax

Place the pragma in the C source within the boundaries of the region, function, or loop.

```
#pragma HLS dataflow
```

Example 1

Specifies `DATAFLOW` optimization within the loop `wr_loop_j`.

```

    wr_loop_j: for (int j = 0; j < TILE_PER_ROW; ++j) {
#pragma HLS DATAFLOW
        wr_buf_loop_m: for (int m = 0; m < TILE_HEIGHT; ++m) {
            wr_buf_loop_n: for (int n = 0; n < TILE_WIDTH; ++n) {
#pragma HLS PIPELINE
                // should burst TILE_WIDTH in WORD beat
                outFifo >> tile[m][n];
            }
        }
        wr_loop_m: for (int m = 0; m < TILE_HEIGHT; ++m) {
            wr_loop_n: for (int n = 0; n < TILE_WIDTH; ++n) {
#pragma HLS PIPELINE
                outx[TILE_HEIGHT*TILE_PER_ROW*TILE_WIDTH*i
+TILE_PER_ROW*TILE_WIDTH*m+TILE_WIDTH*j+n] = tile[m][n];
            }
        }
    }

```

See Also

- [pragma HLS allocation](#)

pragma HLS dependence

Description

The `DEPENDENCE` pragma is used to provide additional information that can overcome loop-carry dependencies and allow loops to be pipelined (or pipelined with lower intervals).

The Vivado HLS tool automatically detects the following dependencies:

- Within loops (loop-independent dependence), or
- Between different iterations of a loop (loop-carry dependence).

These dependencies impact when operations can be scheduled, especially during function and loop pipelining.

- **Loop-independent dependence:** The same element is accessed in the same loop iteration.

```
for (i=0;i<N;i++) {
    A[i]=x;
    y=A[i];
}
```

- **Loop-carry dependence:** The same element is accessed in a different loop iteration.

```
for (i=0;i<N;i++) {
    A[i]=A[i-1]*2;
}
```

Under certain complex scenarios, automatic dependence analysis can be too conservative and fail to filter out false dependencies. Under some circumstances, such as variable dependent array indexing or when an external requirement needs to be enforced (for example, two inputs are never the same index), the dependence analysis might be too conservative. The `DEPENDENCE` pragma allows you to explicitly specify the dependence and resolve a false dependence.



IMPORTANT! *Specifying a false dependency, when in fact the dependency is not false, can result in incorrect hardware. Ensure dependencies are correct (true or false) before specifying them.*

Syntax

Place the pragma within the boundaries of the function where the dependence is defined.

```
#pragma HLS dependence variable=<variable> <class> \
<type> <direction> distance=<int> <dependent>
```

Where:

- `variable=<variable>`: Optionally specifies the variable to consider for the dependence.
- `<class>`: Optionally specifies a class of variables in which the dependence needs clarification. Valid values include `array` or `pointer`.



TIP: *<class> and variable= do not need to be specified together as you can either specify a variable or a class of variables within a function.*

- `<type>`: Valid values include `intra` or `inter`. Specifies whether the dependence is:
 - `intra`: Dependence within the same loop iteration. When dependence `<type>` is specified as `intra`, and `<dependent>` is false, the HLS tool might move operations freely within a loop, increasing their mobility and potentially improving performance or area. When `<dependent>` is specified as true, the operations must be performed in the order specified.

- **inter**: dependence between different loop iterations. This is the default `<type>`. If dependence `<type>` is specified as `inter`, and `<dependent>` is false, it allows the HLS tool to perform operations in parallel if the function or loop is pipelined, or the loop is unrolled, or partially unrolled, and prevents such concurrent operation when `<dependent>` is specified as true.
- **<direction>**: Valid values include `RAW`, `WAR`, or `WAW`. This is relevant for loop-carry dependencies only, and specifies the direction for a dependence:
 - **RAW**: (Read-After-Write - true dependence) The write instruction uses a value used by the read instruction.
 - **WAR**: (Write-After-Read - anti dependence) The read instruction gets a value that is overwritten by the write instruction.
 - **WAW**: (Write-After-Write - output dependence) Two write instructions write to the same location, in a certain order.
- **distance=<int>**: Specifies the inter-iteration distance for array access. Relevant only for loop-carry dependencies where dependence is set to `true`.
- **<dependent>**: Specifies whether a dependence needs to be enforced (`true`) or removed (`false`). The default is `true`.

Example 1

In the following example, the HLS tool does not have any knowledge about the value of `cols` and conservatively assumes that there is always a dependence between the write to `buff_A[1][col]` and the read from `buff_A[1][col]`. In an algorithm such as this, it is unlikely `cols` will ever be zero, but the HLS tool cannot make assumptions about data dependencies. To overcome this deficiency, you can use the `DEPENDENCE` pragma to state that there is no dependence between loop iterations (in this case, for both `buff_A` and `buff_B`).

```
void foo(int rows, int cols, ...)
  for (row = 0; row < rows + 1; row++) {
    for (col = 0; col < cols + 1; col++) {
      #pragma HLS PIPELINE II=1
      #pragma HLS dependence variable=buff_A inter false
      #pragma HLS dependence variable=buff_B inter false
      if (col < cols) {
        buff_A[2][col] = buff_A[1][col]; // read from buff_A[1][col]
        buff_A[1][col] = buff_A[0][col]; // write to buff_A[1][col]
        buff_B[1][col] = buff_B[0][col];
        temp = buff_A[0][col];
      }
    }
  }
```

Example 2

Removes the dependence between `Var1` in the same iterations of `loop_1` in function `foo`.

```
#pragma HLS dependence variable=Var1 intra false
```

Example 3

Defines the dependence on all arrays in `loop_2` of function `foo` to inform the HLS tool that all reads must happen after writes (RAW) in the same loop iteration.

```
#pragma HLS dependence array intra RAW true
```

See Also

- [pragma HLS pipeline](#)
- [xcl_pipeline_loop](#)

pragma HLS expression_balance

Description

Sometimes a C-based specification is written with a sequence of operations, resulting in a long chain of operations in RTL. With a small clock period, this can increase the latency in the design. By default, the Vivado HLS tool rearranges the operations using associative and commutative properties. This rearrangement creates a balanced tree that can shorten the chain, potentially reducing latency in the design at the cost of extra hardware.

The `EXPRESSION_BALANCE` pragma allows this expression balancing to be disabled, or to be expressly enabled, within a specified scope.

Syntax

Place the pragma in the C source within the boundaries of the required location.

```
#pragma HLS expression_balance off
```

Where:

- `off`: Turns off expression balancing at this location.



TIP: Leaving this option out of the pragma enables expression balancing, which is the default mode.

Example 1

This example explicitly enables expression balancing in function `my_Func`.

```
void my_func(char inval, char incr) {
    #pragma HLS expression_balance
```

Example 2

Disables expression balancing within function `my_Func`.

```
void my_func(char inval, char incr) {
    #pragma HLS expression_balance off
}
```

pragma HLS function_instantiate

Description

The `FUNCTION_INSTANTIATE` pragma is an optimization technique that has the area benefits of maintaining the function hierarchy but provides an additional powerful option: performing targeted local optimizations on specific instances of a function. This can simplify the control logic around the function call and potentially improve latency and throughput.

By default:

- Functions remain as separate hierarchy blocks in the RTL.
- All instances of a function, at the same level of hierarchy, make use of a single RTL implementation (block).

The `FUNCTION_INSTANTIATE` pragma is used to create a unique RTL implementation for each instance of a function, allowing each instance to be locally optimized according to the function call. This pragma exploits the fact that some inputs to a function may be a constant value when the function is called, and uses this to both simplify the surrounding control structures and produce smaller more optimized function blocks.

Without the `FUNCTION_INSTANTIATE` pragma, the following code results in a single RTL implementation of function `foo_sub` for all three instances of the function in `foo`. Each instance of function `foo_sub` is implemented in an identical manner. This is fine for function reuse and reducing the area required for each instance call of a function, but means that the control logic inside the function must be more complex to account for the variation in each call of `foo_sub`.

```
char foo_sub(char inval, char incr) {
    #pragma HLS function_instantiate variable=incr
    return inval + incr;
}
void foo(char inval1, char inval2, char inval3,
         char *outval1, char *outval2, char * outval3)
{
    *outval1 = foo_sub(inval1, 1);
    *outval2 = foo_sub(inval2, 2);
    *outval3 = foo_sub(inval3, 3);
}
```

In the code sample above, the `FUNCTION_INSTANTIATE` pragma results in three different implementations of function `foo_sub`, each independently optimized for the `incr` argument, reducing the area and improving the performance of the function. After `FUNCTION_INSTANTIATE` optimization, `foo_sub` is effectively be transformed into three separate functions, each optimized for the specified values of `incr`.

Syntax

Place the pragma in the C source within the boundaries of the required location.

```
#pragma HLS function_instantiate variable=<variable>
```

Where:

- `variable=<variable>`: A required argument that defines the function argument to use as a constant.

Example 1

In the following example, the `FUNCTION_INSTANTIATE` pragma, placed in function `swInt`, allows each instance of function `swInt` to be independently optimized with respect to the `maxv` function argument.

```
void swInt(unsigned int *readRefPacked, short *maxr, short *maxc, short
*maxv){
#pragma HLS function_instantiate variable=maxv
    uint2_t d2bit[MAXCOL];
    uint2_t q2bit[MAXROW];
#pragma HLS array_partition variable=d2bit,q2bit cyclic factor=FACTOR

    intTo2bit<MAXCOL/16>((readRefPacked + MAXROW/16), d2bit);
    intTo2bit<MAXROW/16>(readRefPacked, q2bit);
    sw(d2bit, q2bit, maxr, maxc, maxv);
}
```

See Also

- [pragma HLS allocation](#)
- [pragma HLS inline](#)

pragma HLS inline

Description

Removes a function as a separate entity in the hierarchy. After inlining, the function is dissolved into the calling function and no longer appears as a separate level of hierarchy in the RTL. In some cases, inlining a function allows operations within the function to be shared and optimized more effectively with surrounding operations. An inlined function cannot be shared. This can increase area required for implementing the RTL.

The `INLINE` pragma applies differently to the scope it is defined in depending on how it is specified:

- **INLINE:** Without arguments, the pragma means that the function it is specified in should be inlined upward into any calling functions or regions.
- **INLINE OFF:** Specifies that the function it is specified in should *not* be inlined upward into any calling functions or regions. This disables the inline of a specific function that may be automatically inlined, or inlined as part of a region or recursion.
- **INLINE REGION:** Applies the pragma to the region or the body of the function it is assigned in. It applies downward, inlining the contents of the region or function, but not inlining recursively through the hierarchy.
- **INLINE RECURSIVE:** Applies the pragma to the region or the body of the function it is assigned in. It applies downward, recursively inlining the contents of the region or function.

By default, inlining is only performed on the next level of function hierarchy, not sub-functions. However, the `recursive` option lets you specify inlining through levels of the hierarchy.

Syntax

Place the pragma in the C source within the body of the function or region of code.

```
#pragma HLS inline <region | recursive | off>
```

Where:

- **region:** Optionally specifies that all functions in the specified region (or contained within the body of the function) are to be inlined, applies to the scope of the region.
- **recursive:** By default, only one level of function inlining is performed, and functions within the specified function are not inlined. The `recursive` option inlines all functions recursively within the specified function or region.
- **off:** Disables function inlining to prevent specified functions from being inlined. For example, if `recursive` is specified in a function, this option can prevent a particular called function from being inlined when all others are.



TIP: *The Vivado HLS tool automatically inlines small functions, and using the `INLINE` pragma with the `off` option may be used to prevent this automatic inlining.*

Example 1

This example inlines all functions within the region it is specified in, in this case the body of `foo_top`, but does not inline any lower level functions within those functions.

```
void foo_top { a, b, c, d } {
    #pragma HLS inline region
    ...
}
```

Example 2

The following example inlines all functions within the body of `foo_top` inlining recursively down through the function hierarchy, except function `foo_sub` is not inlined. The recursive pragma is placed in function `foo_top`. The pragma to disable inlining is placed in the function `foo_sub`:

```
foo_sub (p, q) {
#pragma HLS inline off
int q1 = q + 10;
foo(p1,q);// foo_3
...
}
void foo_top { a, b, c, d} {
#pragma HLS inline region recursive
...
foo(a,b);//foo_1
foo(a,c);//foo_2
foo_sub(a,d);
...
}
```

Note: Notice in this example, that `INLINE` applies downward to the contents of function `foo_top`, but applies upward to the code calling `foo_sub`.

Example 3

This example inlines the `copy_output` function into any functions or regions calling `copy_output`.

```
void copy_output(int *out, int out_lcl[OSize * OSize], int output) {
#pragma HLS INLINE
// Calculate each work_item's result update location
int stride = output * OSize * OSize;

// Work_item updates output filter/image in DDR
writeOut: for(int itr = 0; itr < OSize * OSize; itr++) {
#pragma HLS PIPELINE
    out[stride + itr] = out_lcl[itr];
}
}
```

See Also

- [pragma HLS allocation](#)
- [pragma HLS function_instantiate](#)

pragma HLS interface

Description

In C-based design, all input and output operations are performed, in zero time, through formal function arguments. In a RTL design, these same input and output operations must be performed through a port in the design interface and typically operate using a specific input/output (I/O) protocol. For more information, refer to "Managing Interfaces" in the *Vivado Design Suite User Guide: High-Level Synthesis* ([UG902](#)).

The `INTERFACE` pragma specifies how RTL ports are created from the function definition during interface synthesis.

The ports in the RTL implementation are derived from the following:

- Any function-level protocol that is specified: Function-level protocols, also called block-level I/O protocols, provide signals to control when the function starts operation, and indicate when function operation ends, is idle, and is ready for new inputs. The implementation of a function-level protocol is:
 - Specified by the `<mode>` values `ap_ctrl_none`, `ap_ctrl_hs` or `ap_ctrl_chain`. The `ap_ctrl_hs` block-level I/O protocol is the default.
 - Associated with the function name.
- Function arguments: Each function argument can be specified to have its own port-level (I/O) interface protocol, such as valid handshake (`ap_vld`), or acknowledge handshake (`ap_ack`). Port-level interface protocols are created for each argument in the top-level function and the function return, if the function returns a value. The default I/O protocol created depends on the type of C argument. After the block-level protocol has been used to start the operation of the block, the port-level I/O protocols are used to sequence data into and out of the block.
- Global variables accessed by the top-level function, and defined outside its scope:
 - If a global variable is accessed, but all read and write operations are local to the function, the resource is created in the RTL design. There is no need for an I/O port in the RTL. If the global variable is expected to be an external source or destination, specify its interface in a similar manner as standard function arguments. See the [Examples](#) below.

When the `INTERFACE` pragma is used on sub-functions, only the `register` option can be used. The `<mode>` option is not supported on sub-functions.



TIP: The Vivado HLS tool automatically determines the I/O protocol used by any sub-functions. You cannot control these ports except to specify whether the port is registered.

Specifying Burst Mode

When specifying burst-mode for interfaces, using the `max_read_burst_length` or `max_write_burst_length` options (as described in the [Syntax](#) section) there are limitations and related considerations that are derived from the AXI standard:

1. The burst length should be less than, or equal to 256 words per transaction, because `ARLEN` & `AWLEN` are 8 bits; the actual burst length is `AxLEN+1`.
2. In total, less than 4 KB is transferred per burst transaction.
3. Do not cross the 4 KB address boundary.
4. The bus width is specified as a power of 2, between 32-bits and 512-bits (that is, 32, 64, 128, 256, 512-bits) or in bytes: 4, 8, 16, 32, 64.

With the 4 KB limit, the max burst length for a bus width of:

- 32-bits is 256 words transferred in a single burst transaction. In this case, the total bytes transferred per transaction would be 1024.
- 64-bits is 256 words transferred in a single burst transaction. The total bytes transferred per transaction would be 2048.
- 128-bits is 256 words transferred in a single burst transaction. The total bytes transferred per transaction would be 4096.
- 256-bits is 128 words transferred in a single burst transaction. The total bytes transferred per transaction would be 4096.
- 512-bits is 64 words transferred in a single burst transaction. The total bytes transferred per transaction would be 4096.

Note: The IP generated by the HLS tool might not actually perform the maximum burst length as this is design dependent. For example, pipelined accesses from a for-loop of 100 iterations when `max_read_burst_length` or `max_write_burst_length` is set to 128, will not fill the max burst length.

However, if the design is doing longer accesses in the source code than the specified maximum burst length, the access will be split into smaller bursts. For example, a pipelined for-loop with 100 accesses and `max_read_burst_length` or `max_write_burst_length` set to 64, will be split into 2 transactions, one of the max burst length (or 64) and one transaction of the remaining data (burst of length 36 words).

Syntax

Place the pragma within the boundaries of the function.

```
#pragma HLS interface <mode> port=<name> bundle=<string> \
register register_mode=<mode> depth=<int> offset=<string> \
clock=<string> name=<string> \
num_read_outstanding=<int> num_write_outstanding=<int> \
max_read_burst_length=<int> max_write_burst_length=<int>
```

Where:

- `<mode>`: Specifies the interface protocol mode for function arguments, global variables used by the function, or the block-level control protocols. For detailed descriptions of these different modes see "Interface Synthesis Reference" in the *Vivado Design Suite User Guide: High-Level Synthesis (UG902)*. The mode can be specified as one of the following:
 - `ap_none`: No protocol. The interface is a data port.
 - `ap_stable`: No protocol. The interface is a data port. The HLS tool assumes the data port is always stable after reset, which allows internal optimizations to remove unnecessary registers.
 - `ap_vald`: Implements the data port with an associated `valid` port to indicate when the data is valid for reading or writing.
 - `ap_ack`: Implements the data port with an associated `acknowledge` port to acknowledge that the data was read or written.
 - `ap_hs`: Implements the data port with associated `valid` and `acknowledge` ports to provide a two-way handshake to indicate when the data is valid for reading and writing and to acknowledge that the data was read or written.
 - `ap_ovld`: Implements the output data port with an associated `valid` port to indicate when the data is valid for reading or writing.



IMPORTANT! *The HLS tool implements the input argument or the input half of any read/write arguments with mode `ap_none`.*

- `ap_fifo`: Implements the port with a standard FIFO interface using data input and output ports with associated active-Low FIFO `empty` and `full` ports.

Note: You can only use this interface on read arguments or write arguments. The `ap_fifo` mode does not support bidirectional read/write arguments.
- `ap_bus`: Implements pointer and pass-by-reference ports as a bus interface.
- `ap_memory`: Implements array arguments as a standard RAM interface. If you use the RTL design in the Vivado IP integrator, the memory interface appears as discrete ports.
- `bram`: Implements array arguments as a standard RAM interface. If you use the RTL design in the IP integrator, the memory interface appears as a single port.
- `axis`: Implements all ports as an AXI4-Stream interface.
- `s_axilite`: Implements all ports as an AXI4-Lite interface. The HLS tool produces an associated set of C driver files during the Export RTL process.
- `m_axi`: Implements all ports as an AXI4 interface. You can use the `config_interface` command to specify either 32-bit (default) or 64-bit address ports and to control any address offset.

- `ap_ctrl_none`: No block-level I/O protocol.

Note: Using the `ap_ctrl_none` mode might prevent the design from being verified using the C/RTL co-simulation feature.

- `ap_ctrl_hs`: Implements a set of block-level control ports to `start` the design operation and to indicate when the design is `idle`, `done`, and `ready` for new input data.

Note: The `ap_ctrl_hs` mode is the default block-level I/O protocol.

- `ap_ctrl_chain`: Implements a set of block-level control ports to `start` the design operation, `continue` operation, and indicate when the design is `idle`, `done`, and `ready` for new input data.

Note: The `ap_ctrl_chain` interface mode is similar to `ap_ctrl_hs` but provides an additional input signal `ap_continue` to apply back pressure. Xilinx recommends using the `ap_ctrl_chain` block-level I/O protocol when chaining the HLS tool blocks together.

- `port=<name>`: Specifies the name of the function argument, function return, or global variable which the `INTERFACE` pragma applies to.



TIP: Block-level I/O protocols (`ap_ctrl_none`, `ap_ctrl_hs`, or `ap_ctrl_chain`) can be assigned to a port for the function `return` value.

- `bundle=<string>`:

- Groups function arguments into AXI interface ports. By default, the HLS tool groups all function arguments specified as an AXI4-Lite (`s_axilite`) interface into a single AXI4-Lite port. Similarly, all function arguments specified as an AXI4 (`m_axi`) interface are grouped into a single AXI4 port. This option explicitly groups all interface ports with the same `bundle=<string>` into the same AXI interface port and names the RTL port the value specified by `<string>`.



IMPORTANT! When specifying the `bundle=` name you should use all lower-case characters.

- `register`: An optional keyword to register the signal and any relevant protocol signals, and causes the signals to persist until at least the last cycle of the function execution. This option applies to the following interface modes:
 - `ap_none`
 - `ap_ack`
 - `ap_vld`
 - `ap_ovld`
 - `ap_hs`
 - `ap_stable`
 - `axis`

- `s_axilite`



TIP: The `-register_io` option of the `config_interface` command globally controls registering all inputs/outputs on the top function. Refer to the *Vivado Design Suite User Guide: High-Level Synthesis (UG902)* for more information.

- `register_mode= <forward|reverse|both|off>`: Used with the `register` keyword, this option specifies if registers are placed on the `forward` path (TDATA and TVALID), the `reverse` path (TREADY), on `both` paths (TDATA, TVALID, and TREADY), or if none of the port signals are to be registered (`off`). The default `register_mode` is `both`. AXI4-Stream (`axis`) side-channel signals are considered to be data signals and are registered whenever the TDATA is registered.
- `depth=<int>`: Specifies the maximum number of samples for the test bench to process. This setting indicates the maximum size of the FIFO needed in the verification adapter that the HLS tool creates for RTL co-simulation.



TIP: While `depth` is usually an option, it is required for `m_axi` interfaces.

- `offset=<string>`: Controls the address offset in AXI4-Lite (`s_axilite`) and AXI4 (`m_axi`) interfaces.
 - For the `s_axilite` interface, `<string>` specifies the address in the register map.
 - For the `m_axi` interface, `<string>` specifies one of the following values:
 - `direct`: Generate a scalar input offset port.
 - `slave`: Generate an offset port and automatically map it to an AXI4-Lite slave interface.
 - `off`: Do not generate an offset port.



TIP: The `-m_axi_offset` option of the `config_interface` command globally controls the offset ports of all M_AXI interfaces in the design.

- `clock=<name>`: Optionally specified only for interface mode `s_axilite`. This defines the clock signal to use for the interface. By default, the AXI4-Lite interface clock is the same clock as the system clock. This option is used to specify a separate clock for the AXI4-Lite (`s_axilite`) interface.



TIP: If the `bundle` option is used to group multiple top-level function arguments into a single AXI4-Lite interface, the `clock` option need only be specified on one of the bundle members.

- `latency=<value>`: When `mode` is `m_axi`, this specifies the expected latency of the AXI4 interface, allowing the design to initiate a bus request a number of cycles (`latency`) before the read or write is expected. If this figure is too low, the design will be ready too soon and may stall waiting for the bus. If this figure is too high, bus access may be granted but the bus may stall waiting on the design to start the access.

- **num_read_outstanding=<int>**: For AXI4 (`m_axi`) interfaces, this option specifies how many read requests can be made to the AXI4 bus, without a response, before the design stalls. This implies internal storage in the design, a FIFO of size:
`num_read_outstanding*max_read_burst_length*word_size`.
- **num_write_outstanding=<int>**: For AXI4 (`m_axi`) interfaces, this option specifies how many write requests can be made to the AXI4 bus, without a response, before the design stalls. This implies internal storage in the design, a FIFO of size:
`num_write_outstanding*max_write_burst_length*word_size`.
- **max_read_burst_length=<int>**:
 - For AXI4 (`m_axi`) interfaces, this option specifies the maximum number of data values read during a burst transfer.
 - **max_write_burst_length=<int>**: For AXI4 (`m_axi`) interfaces, this option specifies the maximum number of data values written during a burst transfer.



TIP: If the port is a read-only port, then set the `num_write_outstanding=1` and `max_write_burst_length=2` to conserve memory resources. For write-only ports, set the `num_read_outstanding=1` and `max_read_burst_length=2`.

- **name=<string>**: This option is used to rename the port based on your own specification. The generated RTL port will use this name.

Example 1

In this example, both function arguments are implemented using an AXI4-Stream interface:

```
void example(int A[50], int B[50]) {
    //Set the HLS native interface types
    #pragma HLS INTERFACE axis port=A
    #pragma HLS INTERFACE axis port=B
    int i;
    for(i = 0; i < 50; i++){
        B[i] = A[i] + 5;
    }
}
```

Example 2

The following turns off block-level I/O protocols, and is assigned to the function return value:

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

The function argument `InData` is specified to use the `ap_vld` interface, and also indicates the input should be registered:

```
#pragma HLS interface ap_vld register port=InData
```

This exposes the global variable `lookup_table` as a port on the RTL design, with an `ap_memory` interface:

```
pragma HLS interface ap_memory port=lookup_table
```

Example 3

This example defines the INTERFACE standards for the ports of the top-level `transpose` function. Notice the use of the `bundle=` option to group signals.

```
// TOP LEVEL - TRANSPOSE
void transpose(int* input, int* output) {
    #pragma HLS INTERFACE m_axi port=input offset=slave bundle=gmem0
    #pragma HLS INTERFACE m_axi port=output offset=slave bundle=gmem1

    #pragma HLS INTERFACE s_axilite port=input bundle=control
    #pragma HLS INTERFACE s_axilite port=output bundle=control
    #pragma HLS INTERFACE s_axilite port=return bundle=control

    #pragma HLS dataflow
```

pragma HLS latency

Description

Specifies a minimum or maximum latency value, or both, for the completion of functions, loops, and regions.

- **Latency:** Number of clock cycles required to produce an output.
- **Function latency:** Number of clock cycles required for the function to compute all output values, and return.
- **Loop latency:** Number of cycles to execute all iterations of the loop.

See the "Performance Metrics Example" in the *Vivado Design Suite User Guide: High-Level Synthesis (UG902)*.

The Vivado HLS tool always tries to minimize latency in the design. When the `LATENCY` pragma is specified, the tool behavior is as follows:

- Latency is greater than the minimum, or less than the maximum: The constraint is satisfied. No further optimizations are performed.
- Latency is less than the minimum: If the HLS tool can achieve less than the minimum specified latency, it extends the latency to the specified value, potentially increasing sharing.
- Latency is greater than the maximum: If HLS tool cannot schedule within the maximum limit, it increases effort to achieve the specified constraint. If it still fails to meet the maximum latency, it issues a warning, and produces a design with the smallest achievable latency in excess of the maximum.



TIP: You can also use the `LATENCY` pragma to limit the efforts of the tool to find an optimum solution. Specifying latency constraints for scopes within the code: loops, functions, or regions, reduces the possible solutions within that scope and improves tool runtime. Refer to "Improving Run Time and Capacity" of Vivado Design Suite User Guide: High-Level Synthesis (UG902) for more information.

Syntax

Place the pragma within the boundary of a function, loop, or region of code where the latency must be managed.

```
#pragma HLS latency min=<int> max=<int>
```

Where:

- `min=<int>`: Optionally specifies the minimum latency for the function, loop, or region of code.
- `max=<int>`: Optionally specifies the maximum latency for the function, loop, or region of code.

Note: Although both min and max are described as optional, one must be specified.

Example 1

Function `foo` is specified to have a minimum latency of 4 and a maximum latency of 8.

```
int foo(char x, char a, char b, char c) {
    #pragma HLS latency min=4 max=8
    char y;
    y = x*a+b+c;
    return y
}
```

Example 2

In the following example, `loop_1` is specified to have a maximum latency of 12. Place the pragma in the loop body as shown.

```
void foo (num_samples, ...) {
    int i;
    ...
    loop_1: for(i=0;i< num_samples;i++) {
        #pragma HLS latency max=12
        ...
        result = a + b;
    }
}
```

Example 3

The following example creates a code region and groups signals that need to change in the same clock cycle by specifying zero latency.

```
// create a region { } with a latency = 0
{
    #pragma HLS LATENCY max=0 min=0
    *data = 0xFF;
    *data_vld = 1;
}
```

pragma HLS loop_flatten

Description

Allows nested loops to be flattened into a single loop hierarchy with improved latency.

In the RTL implementation, it requires one clock cycle to move from an outer loop to an inner loop, and from an inner loop to an outer loop. Flattening nested loops allows them to be optimized as a single loop. This saves clock cycles, potentially allowing for greater optimization of the loop body logic.

Apply the `LOOP_FLATTEN` pragma to the loop body of the inner-most loop in the loop hierarchy. Only perfect and semi-perfect loops can be flattened in this manner:

- **Perfect loop nests:**
 - Only the innermost loop has loop body content.
 - There is no logic specified between the loop statements.
 - All loop bounds are constant.
- **Semi-perfect loop nests:**
 - Only the innermost loop has loop body content.
 - There is no logic specified between the loop statements.
 - The outermost loop bound can be a variable.
- **Imperfect loop nests:** When the inner loop has variable bounds (or the loop body is not exclusively inside the inner loop), try to restructure the code, or unroll the loops in the loop body to create a perfect loop nest.

Syntax

Place the pragma in the C source within the boundaries of the nested loop.

```
#pragma HLS loop_flatten off
```

Where:

- `off`: Optional keyword. Prevents flattening from taking place, and can prevent some loops from being flattened while all others in the specified location are flattened.

Note: The presence of the `LOOP_FLATTEN` pragma enables the optimization.

Example 1

Flattens `loop_1` in function `foo` and all (perfect or semi-perfect) loops above it in the loop hierarchy, into a single loop. Place the pragma in the body of `loop_1`.

```
void foo (num_samples, ...) {
    int i;
    ...
    loop_1: for(i=0;i< num_samples;i++) {
        #pragma HLS loop_flatten
        ...
        result = a + b;
    }
}
```

Example 2

Prevents loop flattening in `loop_1`.

```
loop_1: for(i=0;i< num_samples;i++) {
    #pragma HLS loop_flatten off
    ...
}
```

See Also

- [pragma HLS loop_merge](#)
- [pragma HLS loop_tripcount](#)
- [pragma HLS unroll](#)

pragma HLS loop_merge

Description

Merges consecutive loops into a single loop to reduce overall latency, increase sharing, and improve logic optimization. Merging loops:

- Reduces the number of clock cycles required in the RTL to transition between the loop-body implementations.
- Allows the loops be implemented in parallel (if possible).

The `LOOP_MERGE` pragma will seek to merge all loops within the scope it is placed. For example, if you apply a `LOOP_MERGE` pragma in the body of a loop, the Vivado HLS tool applies the pragma to any sub-loops within the loop but not to the loop itself.

The rules for merging loops are:

- If the loop bounds are variables, they must have the same value (number of iterations).
- If the loop bounds are constants, the maximum constant value is used as the bound of the merged loop.
- Loops with both variable bounds and constant bounds cannot be merged.
- The code between loops to be merged cannot have side effects. Multiple execution of this code should generate the same results (`a=b` is allowed, `a=a+1` is not).
- Loops cannot be merged when they contain FIFO reads. Merging changes the order of the reads. Reads from a FIFO or FIFO interface must always be in sequence.

Syntax

Place the pragma in the C source within the required scope or region of code.

```
#pragma HLS loop_merge force
```

Where:

- `force`: Optional keyword to force loops to be merged even when the HLS tool issues a warning.



IMPORTANT! *In this case, you must manually ensure that the merged loop will function correctly.*

Examples

Merges all consecutive loops in function `foo` into a single loop.

```
void foo (num_samples, ...) {
    #pragma HLS loop_merge
    int i;
    ...
    loop_1: for(i=0;i< num_samples;i++) {
        ...
    }
```

All loops inside `loop_2` (but not `loop_2` itself) are merged by using the `force` option. Place the pragma in the body of `loop_2`.

```
loop_2: for(i=0;i< num_samples;i++) {
    #pragma HLS loop_merge force
    ...
}
```

See Also

- [pragma HLS loop_flatten](#)
- [pragma HLS loop_tripcount](#)
- [pragma HLS unroll](#)

pragma HLS loop_tripcount

Description

When manually applied to a loop, specifies the total number of iterations performed by a loop.



IMPORTANT! *The `LOOP_TRIPCOUNT` pragma is for analysis only, and does not impact the results of synthesis.*

The Vivado HLS tool reports the total latency of each loop, which is the number of clock cycles to execute all iterations of the loop. Therefore, the loop latency is a function of the number of loop iterations, or tripcount.

The tripcount can be a constant value. It can depend on the value of variables used in the loop expression (for example, $x < y$), or depend on control statements used inside the loop. In some cases, the HLS tool cannot determine the tripcount, and the latency is unknown. This includes cases in which the variables used to determine the tripcount are:

- Input arguments or
- Variables calculated by dynamic operation.

In cases where the loop latency is unknown or cannot be calculated, the `LOOP_TRIPCOUNT` pragma lets you specify minimum and maximum iterations for a loop. This allows the tool analyze how the loop latency contributes to the total design latency in the reports, and helps you determine appropriate optimizations for the design.

Syntax

Place the pragma in the C source within the body of the loop.

```
#pragma HLS loop_tripcount min=<int> max=<int> avg=<int>
```

Where:

- `max= <int>`: Specifies the maximum number of loop iterations.
- `min=<int>`: Specifies the minimum number of loop iterations.
- `avg=<int>`: Specifies the average number of loop iterations.

Examples

In the following example, `loop_1` in function `foo` is specified to have a minimum tripcount of 12 and a maximum tripcount of 16.

```
void foo (num_samples, ...) {
    int i;
    ...
    loop_1: for(i=0;i< num_samples;i++) {
        #pragma HLS loop_tripcount min=12 max=16
        ...
        result = a + b;
    }
}
```

pragma HLS occurrence

Description

When pipelining functions or loops, specifies that the code in a region is executed less frequently than the code in the enclosing function or loop. This allows the code that is executed less often to be pipelined at a slower rate, and potentially shared within the top-level pipeline. To determine the `OCCURRENCE` pragma:

- A loop iterates $\langle N \rangle$ times.
- However, part of the loop body is enabled by a conditional statement, and as a result only executes $\langle M \rangle$ times, where $\langle N \rangle$ is an integer multiple of $\langle M \rangle$.
- The conditional code has an occurrence that is N/M times slower than the rest of the loop body.

For example, in a loop that executes 10 times, a conditional statement within the loop only executes two times has an occurrence of 5 (or $10/2$).

Identifying a region with the `OCCURRENCE` pragma allows the functions and loops in that region to be pipelined with a higher initiation interval that is slower than the enclosing function or loop.

Syntax

Place the pragma in the C source within a region of code.

```
#pragma HLS occurrence cycle=<int>
```

Where:

- `cycle=<int>`: Specifies the occurrence N/M .
 - $\langle N \rangle$: Number of times the enclosing function or loop is executed.
 - $\langle M \rangle$: Number of times the conditional region is executed.



IMPORTANT! *<N> must be an integer multiple of <M>.*

Examples

In this example, the region `Cond_Region` has an occurrence of 4 (it executes at a rate four times less often than the surrounding code that contains it).

```
Cond_Region: {  
#pragma HLS occurrence cycle=4  
...  
}
```

See Also

- [pragma HLS pipeline](#)

pragma HLS pipeline

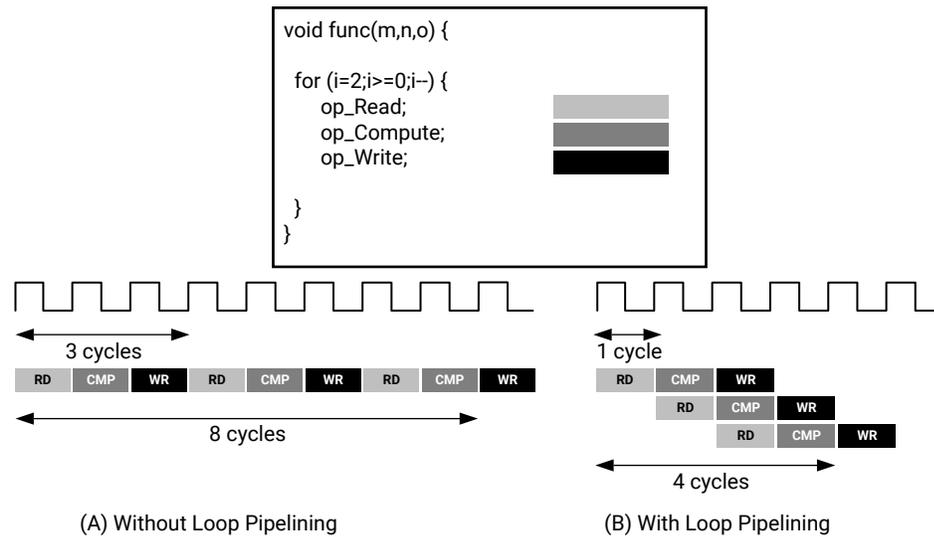
Description

Reduces the initiation interval (II) for a function or loop by allowing the concurrent execution of operations.

A pipelined function or loop can process new inputs every `<N>` clock cycles, where `<N>` is the II of the loop or function. The default II for the `PIPELINE` pragma is 1, which processes a new input every clock cycle. You can also specify the initiation interval through the use of the `II` option for the pragma.

Pipelining a loop allows the operations of the loop to be implemented in a concurrent manner as shown in the following figure. In the following figure, (A) shows the default sequential operation where there are three clock cycles between each input read (`II=3`), and it requires eight clock cycles before the last output write is performed.

Figure 95: Loop Pipeline



X14277-110217



IMPORTANT! Loop pipelining can be prevented by loop carry dependencies. Use the `DEPENDENCE` pragma to provide additional information that can overcome loop-carry dependencies and allow loops to be pipelined (or pipelined with lower intervals).

If the Vivado HLS tool cannot create a design with the specified II, it issues a warning and creates a design with the lowest possible II.

You can then analyze this design with the warning message to determine what steps must be taken to create a design that satisfies the required initiation interval.

Syntax

Place the pragma in the C source within the body of the function or loop.

```
#pragma HLS pipeline II=<int> enable_flush rewind
```

Where:

- `II=<int>`: Specifies the desired initiation interval for the pipeline. The HLS tool tries to meet this request. Based on data dependencies, the actual result might have a larger initiation interval. The default II is 1.
- `enable_flush`: Optional keyword. Implements a pipeline that will flush and empty if the data valid at the input of the pipeline goes inactive.



TIP: This feature is only supported for pipelined functions; it is not supported for pipelined loops.

- `rewind`: Optional keyword. Enables rewinding, or continuous loop pipelining with no pause between one loop iteration ending and the next iteration starting. Rewinding is effective only if there is one single loop (or a perfect loop nest) inside the top-level function. The code segment before the loop:
 - Is considered as initialization.
 - Is executed only once in the pipeline.
 - Cannot contain any conditional operations (if-else).



TIP: This feature is only supported for pipelined loops; it is not supported for pipelined functions.

Example 1

In this example, function `foo` is pipelined with an initiation interval of 1.

```
void foo { a, b, c, d} {
    #pragma HLS pipeline II=1
    ...
}
```

Note: The default value for `II` is 1, so `II=1` is not required in this example.

See Also

- [pragma HLS dependence](#)

pragma HLS reset

Description

Adds or removes resets for specific state variables (global or static).

The reset port is used in an FPGA to restore the registers and block RAM connected to the reset port to an initial value any time the reset signal is applied. The presence and behavior of the RTL reset port is controlled using the `config_rtl` configuration file. The reset settings include the ability to set the polarity of the reset, and specify whether the reset is synchronous or asynchronous, but more importantly it controls, through the `reset` option, which registers are reset when the reset signal is applied. For more information, see "Clock, Reset, and RTL Output" in the *Vivado Design Suite User Guide: High-Level Synthesis* ([UG902](#)).

Greater control over reset is provided through the `RESET` pragma. If a variable is a static or global, the `RESET` pragma is used to explicitly add a reset, or the variable can be removed from the reset by turning `off` the pragma. This can be particularly useful when static or global arrays are present in the design.

Syntax

Place the pragma in the C source within the boundaries of the variable life cycle.

```
#pragma HLS reset variable=<a> off
```

Where:

- `variable=<a>`: Specifies the variable to which the pragma is applied.
- `off`: Indicates that reset is not generated for the specified variable.

Example 1

This example adds reset to the variable `a` in function `foo` even when the global reset setting is `none` or `control`.

```
void foo(int in[3], char a, char b, char c, int out[3]) {
    #pragma HLS reset variable=a
```

Example 2

Removes reset from variable `a` in function `foo` even when the global reset setting is `state` or `all`.

```
void foo(int in[3], char a, char b, char c, int out[3]) {
    #pragma HLS reset variable=a off
```

pragma HLS resource

Description

The `RESET` pragma specifies that a specific library resource (core) is used to implement a variable (array, arithmetic operation, or function argument) in the RTL. If the `RESOURCE` pragma is not specified, the Vivado HLS tool determines the resource to use. The HLS tool implements the operations in the code using hardware cores. When multiple cores in the library can implement the operation, you can specify which core to use with the `RESOURCE` pragma. To generate a list of available cores, use the `list_core` command.



TIP: The `list_core` command obtains details on the cores available in the library. The `list_core` can only be used in the HLS tool Tcl command interface, and a Xilinx device must be specified using the `set_part` command. If a device has not been selected, the `list_core` command does not have any effect.

For example, to specify which memory element in the library to use to implement an array, use the `RESOURCE` pragma. This allows you control whether the array is implemented as a single or a dual-port RAM. This usage is important for arrays on the top-level function interface, because the memory type associated with the array determines the ports needed in the RTL.

You can use the `latency=` option to specify the latency of the core. For block RAMs on the interface, the `latency=` option allows you to model off-chip, non-standard SRAMs at the interface, for example supporting an SRAM with a latency of 2 or 3. For internal operations, the `latency=` option allows the operation to be implemented using more pipelined stages. These additional pipeline stages can help resolve timing issues during RTL synthesis.

For more information, see "Arrays on the Interface" in the *Vivado Design Suite User Guide: High-Level Synthesis* (UG902).



IMPORTANT! To use the `latency=` option, the operation must have an available multi-stage core. The HLS tool provides a multi-stage core for all basic arithmetic operations (add, subtract, multiply and divide), all floating-point operations, and all block RAMs.

For best results, Xilinx recommends that you use `-std=c99` for C and `-fno-builtin` for C and C++. To specify the C compile options, such as `-std=c99`, use the Tcl command `add_files` with the `-cflags` option. Alternatively, select the **Edit CFLAGS** button in the Project Settings dialog box. See "Creating a New Synthesis Project" in the *Vivado Design Suite User Guide: High-Level Synthesis* (UG902).

Syntax

Place the pragma in the C source within the body of the function where the variable is defined.

```
#pragma HLS resource variable=<variable> core=<core>\
latency=<int>
```

Where:

- `variable=<variable>`: Required argument. Specifies the array, arithmetic operation, or function argument to assign the RESOURCE pragma to.
- `core=<core>`: Required argument. Specifies the core, as defined in the technology library.
- `latency=<int>`: Specifies the latency of the core.

Example 1

In the following example, a two-stage pipelined multiplier is specified to implement the multiplication for variable `<c>` of the function `foo`. The HLS tool selects the core to use for variable `<d>`.

```
int foo (int a, int b) {
int c, d;
#pragma HLS RESOURCE variable=c latency=2
c = a*b;
d = a*c;
return d;
}
```

Example 2

In the following example, the `<coeffs[128]>` variable is an argument to the top-level function `foo_top`. This example specifies that `coeffs` is implemented with core `RAM_1P` from the library.

```
#pragma HLS resource variable=coeffs core=RAM_1P
```



TIP: The ports created in the RTL to access the values of `coeffs` are defined in the `RAM_1P` core.

pragma HLS stream

Description

By default, array variables are implemented as RAM:

- Top-level function array parameters are implemented as a RAM interface port.
- General arrays are implemented as RAMs for read-write access.
- In sub-functions involved in [DATAFLOW](#) optimizations, the array arguments are implemented using a RAM ping pong buffer channel.
- Arrays involved in loop-based [DATAFLOW](#) optimizations are implemented as a RAM ping pong buffer channel.

If the data stored in the array is consumed or produced in a sequential manner, a more efficient communication mechanism is to use streaming data as specified by the `STREAM` pragma, where FIFOs are used instead of RAMs.



IMPORTANT! When an argument of the top-level function is specified as [INTERFACE](#) type `ap_fifo`, the array is automatically implemented as streaming.

Syntax

Place the pragma in the C source within the boundaries of the required location.

```
#pragma HLS stream variable=<variable> depth=<int> dim=<int> off
```

Where:

- `variable=<variable>`: Specifies the name of the array to implement as a streaming interface.
- `depth=<int>`: Relevant only for array streaming in [DATAFLOW](#) channels. By default, the depth of the FIFO implemented in the RTL is the same size as the array specified in the C code. This option lets you modify the size of the FIFO and specify a different depth.

When the array is implemented in a DATAFLOW region, it is common to use the `depth=` option to reduce the size of the FIFO. For example, in a DATAFLOW region when all loops and functions are processing data at a rate of `ll=1`, there is no need for a large FIFO because data is produced and consumed in each clock cycle. In this case, the `depth=` option may be used to reduce the FIFO size to 1 to substantially reduce the area of the RTL design.



TIP: The `config_dataflow -depth` command provides the ability to stream all arrays in a **DATAFLOW** region. The `depth=` option specified here overrides the `config_dataflow` command for the assigned `<variable>`.

- `dim=<int>`: Specifies the dimension of the array to be streamed. The default is dimension 1. Specified as an integer from 0 to `<N>`, for an array with `<N>` dimensions.
- `off`: Disables streaming data. Relevant only for array streaming in dataflow channels.



TIP: The `config_dataflow -default_channel fifo` command globally implies a `STREAM` pragma on all arrays in the design. The `off` option specified here overrides the `config_dataflow` command for the assigned variable, and restores the default of using a RAM ping pong buffer-based channel.

Example 1

The following example specifies array `A[10]` to be streaming, and implemented as a FIFO.

```
#pragma HLS STREAM variable=A
```

Example 2

In this example, array `B` is set to streaming with a FIFO depth of 12.

```
#pragma HLS STREAM variable=B depth=12
```

Example 3

Array `C` has streaming disabled. In the following example, it is assumed to be enabled by `config_dataflow`.

```
#pragma HLS STREAM variable=C off
```

pragma HLS top

Description

Attaches a name to a function, which can then be used with the `set_top` command to synthesize the function and any functions called from the specified top-level. This is typically used to synthesize member functions of a class in C/C++.

Specify the pragma in an active solution, and then use the `set_top` command with the new name.

Syntax

Place the pragma in the C source within the boundaries of the required location.

```
#pragma HLS top name=<string>
```

Where:

- `name=<string>`: Specifies the name to be used by the `set_top` command.

Examples

Function `foo_long_name` is designated the top-level function, and renamed to `DESIGN_TOP`. After the pragma is placed in the code, the `set_top` command must still be issued from the Tcl command line, or from the top-level specified in the IDE project settings.

```
void foo_long_name () {
    #pragma HLS top name=DESIGN_TOP
    ...
}

set_top DESIGN_TOP
```

pragma HLS unroll

Description

Unrolls loops to create multiple independent operations rather than a single collection of operations. The `UNROLL` pragma transforms loops by creating multiples copies of the loop body in the RTL design, which allows some or all loop iterations to occur in parallel.

Loops in the C/C++ functions are kept rolled by default. When loops are rolled, synthesis creates the logic for one iteration of the loop, and the RTL design executes this logic for each iteration of the loop in sequence. A loop is executed for the number of iterations specified by the loop induction variable. The number of iterations might also be impacted by logic inside the loop body (for example, `break` conditions or modifications to a loop exit variable). Using the `UNROLL` pragma you can unroll loops to increase data access and throughput.

The `UNROLL` pragma allows the loop to be fully or partially unrolled. Fully unrolling the loop creates a copy of the loop body in the RTL for each loop iteration, so the entire loop can be run concurrently. Partially unrolling a loop lets you specify a factor `<N>`, to create `<N>` copies of the loop body and reduce the loop iterations accordingly. To unroll a loop completely, the loop bounds must be known at compile time. This is not required for partial unrolling.

Partial loop unrolling does not require $\langle N \rangle$ to be an integer factor of the maximum loop iteration count. The Vivado HLS tool adds an exit check to ensure that partially unrolled loops are functionally identical to the original loop. For example, given the following code:

```
for(int i = 0; i < X; i++) {
    pragma HLS unroll factor=2
    a[i] = b[i] + c[i];
}
```

Loop unrolling by a factor of 2 effectively transforms the code to look like the following code where the `break` construct is used to ensure the functionality remains the same, and the loop exits at the appropriate point.

```
for(int i = 0; i < X; i += 2) {
    a[i] = b[i] + c[i];
    if (i+1 >= X) break;
    a[i+1] = b[i+1] + c[i+1];
}
```

Because the maximum iteration count $\langle X \rangle$ is a variable, the HLS tool might not be able to determine its value, so it adds an exit check and control logic to partially unrolled loops. However, if you know that the specified unrolling factor, 2 in this example, is an integer factor of the maximum iteration count $\langle X \rangle$, the `skip_exit_check` option lets you remove the exit check and associated logic. This helps minimize the area and simplify the control logic.



TIP: When the use of pragmas like `DATA_PACK`, `ARRAY_PARTITION`, or `ARRAY_RESHAPE`, let more data be accessed in a single clock cycle, the HLS tool automatically unrolls any loops consuming this data, if doing so improves the throughput. The loop can be fully or partially unrolled to create enough hardware to consume the additional data in a single clock cycle. This feature is controlled using the `config_unroll` command. For more information, see `config_unroll` in the Vivado Design Suite User Guide: High-Level Synthesis (UG902).

Syntax

Place the pragma in the C/C++ source within the body of the loop to unroll.

```
#pragma HLS unroll factor=<N> region skip_exit_check
```

Where:

- **factor=<N>:** Specifies a non-zero integer indicating that partial unrolling is requested. The loop body is repeated the specified number of times, and the iteration information is adjusted accordingly. If `factor=` is not specified, the loop is fully unrolled.
- **region:** Optional keyword that unrolls all loops within the body (region) of the specified loop, without unrolling the enclosing loop itself.
- **skip_exit_check:** Optional keyword that applies only if partial unrolling is specified with `factor=`. The elimination of the exit check is dependent on whether the loop iteration count is known or unknown:

- **Fixed (known) bounds:** No exit condition check is performed if the iteration count is a multiple of the factor. If the iteration count is not an integer multiple of the factor, the tool:
 1. Prevents unrolling.
 2. Issues a warning that the exit check must be performed to proceed.
- **Variable (unknown) bounds:** The exit condition check is removed as requested. You must ensure that:
 1. The variable bounds is an integer multiple of the specified unroll factor.
 2. No exit check is required.

Example 1

The following example fully unrolls `loop_1` in function `foo`. Place the pragma in the body of `loop_1` as shown.

```
loop_1: for(int i = 0; i < N; i++) {
    #pragma HLS unroll
    a[i] = b[i] + c[i];
}
```

Example 2

This example specifies an unroll factor of 4 to partially unroll `loop_2` of function `foo`, and removes the exit check.

```
void foo (...) {
    int8 array1[M];
    int12 array2[N];
    ...
    loop_2: for(i=0;i<M;i++) {
        #pragma HLS unroll skip_exit_check factor=4
        array1[i] = ...;
        array2[i] = ...;
        ...
    }
    ...
}
```

Example 3

The following example fully unrolls all loops inside `loop_1` in function `foo`, but not `loop_1` itself because the presence of the `region` keyword.

```
void foo(int data_in[N], int scale, int data_out1[N], int data_out2[N]) {
    int temp1[N];
    loop_1: for(int i = 0; i < N; i++) {
        #pragma HLS unroll region
        temp1[i] = data_in[i] * scale;
        loop_2: for(int j = 0; j < N; j++) {
```

```

        data_out1[j] = temp1[j] * 123;
    }
    loop_3: for(int k = 0; k < N; k++) {
        data_out2[k] = temp1[k] * 456;
    }
}
}

```

See Also

- [pragma HLS loop_flatten](#)
- [pragma HLS loop_merge](#)
- [pragma HLS loop_tripcount](#)

OpenCL Attributes

This section describes OpenCL™ attributes that can be added to source code to assist system optimization by the Vitis core development kit, and Vivado® HLS tool synthesis.

The Vitis core development kit provides OpenCL attributes to optimize your code for data movement and kernel performance. The goal of data movement optimization is to maximize the system level data throughput by maximizing interface bandwidth usage and DDR bandwidth usage. The goal of kernel computation optimization is to create processing logic that can consume all the data as soon as they arrive at kernel interfaces. This is generally achieved by expanding the processing code to match the data path with techniques such as function inlining and pipelining, loop unrolling, array partitioning, dataflowing, etc.

The OpenCL attributes include the types specified in the following table:

Table 26: OpenCL Attributes by Type

Type	Attributes
Kernel Optimization	<ul style="list-style-type: none"> • reqd_work_group_size • vec_type_hint • work_group_size_hint • xcl_latency • xcl_max_work_group_size • xcl_zero_global_work_offset
Function Inlining	<ul style="list-style-type: none"> • always_inline
Task-level Pipeline	<ul style="list-style-type: none"> • xcl_dataflow • xcl_reqd_pipe_depth
Pipeline	<ul style="list-style-type: none"> • xcl_pipeline_loop • xcl_pipeline_workitems

Table 26: OpenCL Attributes by Type (cont'd)

Type	Attributes
Loop Optimization	<ul style="list-style-type: none"> openccl_unroll_hint xcl_loop_tripcount xcl_pipeline_loop
Array Optimization	<ul style="list-style-type: none"> xcl_array_partition xcl_array_reshape <p>Note: Array variables only accept a single array optimization attribute.</p>



TIP: The Vitis compiler also supports many of the standard attributes supported by *gcc*, such as:

- ALWAYS_INLINE
- NOINLINE
- UNROLL
- NOUNROLL

always_inline

Description

The ALWAYS_INLINE attribute indicates that a function must be inlined. This attribute is a standard feature of GCC, and a standard feature of the Vitis compiler.



TIP: The NOINLINE attribute is also a standard feature of GCC, and is also supported by the Vitis compiler.

This attribute enables a compiler optimization to have a function inlined into the calling function. The inlined function is dissolved and no longer appears as a separate level of hierarchy in the RTL.

In some cases, inlining a function allows operations within the function to be shared and optimized more effectively with surrounding operations in the calling function. However, an inlined function can no longer be shared with other functions, so the logic might be duplicated between the inlined function and a separate instance of the function which can be more broadly shared. While this can improve performance, this will also increase the area required for implementing the RTL.

For OpenCL kernels, the Vitis compiler uses its own rules to inline or not inline a function. To directly control inlining functions, use the ALWAYS_INLINE or NOINLINE attributes.

By default, inlining is only performed on the next level of function hierarchy, not sub-functions.



IMPORTANT! When used with the `XCL_DATAFLOW` attribute, the compiler will ignore the `ALWAYS_INLINE` attribute and not inline the function.

Syntax

Place the attribute in the OpenCL API source before the function definition to always have it inlined whenever the function is called.

```
__attribute__((always_inline))
```

Examples

This example adds the `ALWAYS_INLINE` attribute to function `foo`:

```
__attribute__((always_inline))
void foo ( a, b, c, d ) {
    ...
}
```

This example prevents the inlining of the function `foo`:

```
__attribute__((noinline))
void foo ( a, b, c, d ) {
    ...
}
```

See Also

- <https://gcc.gnu.org>

openccl_unroll_hint

Description



IMPORTANT! This is a compiler hint which the compiler can ignore.

Loop unrolling is an optimization technique available in the Vitis compiler. The purpose of the loop unroll optimization is to expose concurrency to the compiler. This newly exposed concurrency reduces latency and improves performance, but also consumes more FPGA fabric resources.

The `OPENCL_UNROLL_HINT` attribute is part of the OpenCL Specification, and specifies that loops (`for`, `while`, `do`) can be unrolled by the Vitis compiler. See [Loop Unrolling](#) for more information.

The `OPENCL_UNROLL_HINT` attribute qualifier must appear immediately before the loop to be affected. You can use this attribute to specify full unrolling of the loop, partial unrolling by a specified amount, or to disable unrolling of the loop.

Syntax

Place the attribute in the OpenCL source before the loop definition:

```
__attribute__((opencl_unroll_hint(<n>)))
```

Where:

- <n> is an optional loop unrolling factor and must be a positive integer, or compile time constant expression. An unroll factor of 1 disables unrolling.



TIP: If <n> is not specified, the compiler automatically determines the unrolling factor for the loop.

Example 1

The following example unrolls the `for` loop by a factor of 2. This results in two parallel loop iterations instead of four sequential iterations for the compute unit to complete the operation.

```
__attribute__((opencl_unroll_hint(2)))
for(int i = 0; i < LENGTH; i++) {
    bufc[i] = bufa[i] * bufb[i];
}
```

Conceptually the compiler transforms the loop above to the following code.

```
for(int i = 0; i < LENGTH; i+=2) {
    bufc[i] = bufa[i] * bufb[i];
    bufc[i+1] = bufa[i+1] * bufb[i+1];
}
```

See Also

- <https://www.khronos.org/>
- [The OpenCL C Specification](#)

reqd_work_group_size

Description

When OpenCL API kernels are submitted for execution on an OpenCL device, they execute within an index space, called an ND range, which can have 1, 2, or 3 dimensions. This is called the global size in the OpenCL API. The work-group size defines the amount of the ND range that can be processed by a single invocation of a kernel compute unit (CU). The work-group size is also called the local size in the OpenCL API. The OpenCL compiler can determine the work-group size based on the properties of the kernel and selected device. After the work-group size (local size) is determined, the ND range (global size) is divided automatically into work-groups, and the work-groups are scheduled for execution on the device.

Although the OpenCL compiler can define the work-group size, the specification of the `REQD_WORK_GROUP_SIZE` attribute on the kernel to define the work-group size is highly recommended for FPGA implementations of the kernel. The attribute is recommended for performance optimization during the generation of the custom logic for a kernel.



TIP: *In the case of an FPGA implementation, the specification of the `REQD_WORK_GROUP_SIZE` attribute is highly recommended as it can be used for performance optimization during the generation of the custom logic for a kernel.*

OpenCL kernel functions are executed exactly one time for each point in the ND range index space. This unit of work for each point in the ND range is called a work-item. Work-items are organized into work-groups, which are the unit of work scheduled onto compute units. The optional `REQD_WORK_GROUP_SIZE` attribute defines the work-group size of a compute unit that must be used as the `local_work_size` argument to `clEnqueueNDRangeKernel`. This allows the compiler to optimize the generated code appropriately for this kernel.

Syntax

Place this attribute before the kernel definition, or before the primary function specified for the kernel.

```
__attribute__((reqd_work_group_size(<X>, <Y>, <Z>)))
```

Where:

- `<X>`, `<Y>`, `<Z>`: Specifies the ND range of the kernel. This represents each dimension of a three dimensional matrix specifying the size of the work-group for the kernel.

Examples

The following OpenCL C kernel code shows a vector addition design where two arrays of data are summed into a third array. The required size of the work-group is 16x1x1. This kernel will execute 16 times to produce a valid result.

```
#include <clc.h>
// For VHLS OpenCL C kernels, the full work group is synthesized
__attribute__((reqd_work_group_size(16, 1, 1)))
__kernel void
vadd(__global int* a,
     __global int* b,
     __global int* c)
{
    int idx = get_global_id(0);
    c[idx] = a[idx] + b[idx];
}
```

See Also

- <https://www.khronos.org/>

- [The OpenCL C Specification](#)

vec_type_hint

Description



IMPORTANT! This is a compiler hint which the compiler can ignore.

The optional `__attribute__((vec_type_hint(<type>)))` is part of the OpenCL Language Specification, and hints to the OpenCL compiler representing the computational width of the kernel, providing a basis for calculating processor bandwidth usage when the compiler is looking to auto-vectorize the code.

By default, the kernel is assumed to have the `__attribute__((vec_type_hint(int)))` qualifier. This lets you specify a different vectorization type.

Implicit in autovectorization is the assumption that any libraries called from the kernel must be re-compilable at runtime to handle cases where the compiler decides to merge or separate work items. This means that these libraries can never be hard-coded binaries or that hard-coded binaries must be accompanied either by source or some re-targetable intermediate representation. This might be a code security question for some.

Syntax

Place this attribute before the kernel definition, or before the primary function specified for the kernel.

```
__attribute__((vec_type_hint(<type>)))
```

Where:

- `<type>`: is one of the built-in vector types listed in the following table, or the constituent scalar element types.

Note: When not specified, the kernel is assumed to have an INT type.

Table 27: Vector Types

Type	Description
char<n>	A vector of <n> 8-bit signed two's complement integer values.
uchar<n>	A vector of <n> 8-bit unsigned integer values.
short<n>	A vector of <n> 16-bit signed two's complement integer values.
ushort<n>	A vector of <n> 16-bit unsigned integer values.
int<n>	A vector of <n> 32-bit signed two's complement integer values.
uint<n>	A vector of <n> 32-bit unsigned integer values.

Table 27: Vector Types (cont'd)

Type	Description
long<n>	A vector of <n> 64-bit signed two's complement integer values.
ulong<n>	A vector of <n> 64-bit unsigned integer values.
float<n>	A vector of <n> 32-bit floating-point values.
double<n>	A vector of <n> 64-bit floating-point values.

Note: <n> is assumed to be 1 when not specified. The vector data type names defined above where <n> is any value other than 2, 3, 4, 8 and 16, are also reserved. Therefore, <n> can only be specified as 2,3,4,8, and 16.

Examples

The following example autovectorizes assuming double-wide integer as the basic computation width.

```
#include <clc.h>
// For VHLS OpenCL C kernels, the full work group is synthesized
__attribute__((vec_type_hint(double)))
__attribute__((reqd_work_group_size(16, 1, 1)))
__kernel void
...
```

See Also

- <https://www.khronos.org/>
- [The OpenCL C Specification](#)

work_group_size_hint

Description



IMPORTANT! This is a compiler hint, which the compiler might ignore.

The work-group size in the OpenCL API standard defines the size of the ND range space that can be handled by a single invocation of a kernel compute unit. When OpenCL kernels are submitted for execution on an OpenCL device, they execute within an index space, called an ND range, which can have 1, 2, or 3 dimensions.

OpenCL kernel functions are executed exactly one time for each point in the ND range index space. This unit of work for each point in the ND range is called a work-item. Unlike `for` loops in C, where loop iterations are executed sequentially and in-order, an OpenCL runtime and device is free to execute work-items in parallel and in any order.

Work-items are organized into work-groups, which are the unit of work scheduled onto compute units. The optional `WORK_GROUP_SIZE_HINT` attribute is part of the OpenCL Language Specification, and is a hint to the compiler that indicates the work-group size value most likely to be specified by the `local_work_size` argument to `clEnqueueNDRangeKernel`. This allows the compiler to optimize the generated code according to the expected value.



TIP: In the case of an FPGA implementation, the specification of the `REQD_WORK_GROUP_SIZE` attribute, instead of the `WORK_GROUP_SIZE_HINT` is highly recommended because it can be used for performance optimization during the generation of the custom logic for a kernel.

Syntax

Place this attribute before the kernel definition, or before the primary function specified for the kernel:

```
__attribute__((work_group_size_hint(<X>, <Y>, <Z>)))
```

Where:

- `<X>`, `<Y>`, `<Z>`: Specifies the ND range of the kernel. This represents each dimension of a three dimensional matrix specifying the size of the work-group for the kernel.

Examples

The following example is a hint to the compiler that the kernel will most likely be executed with a work-group size of 1.

```
__attribute__((work_group_size_hint(1, 1, 1)))  
__kernel void  
...
```

See Also

- <https://www.khronos.org/>
- [The OpenCL C Specification](#)

xcl_array_partition

Description



IMPORTANT! Array variables only accept one attribute. While `XCL_ARRAY_PARTITION` does support multi-dimensional arrays, you can only reshape one dimension of the array with a single attribute.

An advantage of using the FPGA over other compute devices for OpenCL programs is the ability for the application programmer to customize the memory architecture all throughout the system and into the compute unit. By default, the Vitis compiler generates a memory architecture within the compute unit that maximizes local and private memory bandwidth based on static code analysis of the kernel code. Further optimization of these memories is possible based on attributes in the kernel source code, which can be used to specify physical layouts and implementations of local and private memories. The attribute in the Vitis compiler to control the physical layout of memories in a compute unit is `array_partition`.

For one-dimensional arrays, the `XCL_ARRAY_PARTITION` attribute implements an array declared within kernel code as multiple physical memories instead of a single physical memory. The selection of which partitioning scheme to use depends on the specific application and its performance goals. The array partitioning schemes available in the Vitis compiler are `cyclic`, `block`, and `complete`.

Syntax

Place the attribute with the definition of the array variable.

```
__attribute__((xcl_array_partition(<type>, <factor>,
<dimension>)))
```

Where:

- `<type>`: Specifies one of the following partition types:
 - `cyclic`: Cyclic partitioning is the implementation of an array as a set of smaller physical memories that can be accessed simultaneously by the logic in the compute unit. The array is partitioned cyclically by putting one element into each memory before coming back to the first memory to repeat the cycle until the array is fully partitioned.
 - `block`: Block partitioning is the physical implementation of an array as a set of smaller memories that can be accessed simultaneously by the logic inside of the compute unit. In this case, each memory block is filled with elements from the array before moving on to the next memory.
 - `complete`: Complete partitioning decomposes the array into individual elements. For a one-dimensional array, this corresponds to resolving a memory into individual registers. The default `<type>` is `complete`.
- `<factor>`: For cyclic type partitioning, the `<factor>` specifies how many physical memories to partition the original array into in the kernel code. For block type partitioning, the `<factor>` specifies the number of elements from the original array to store in each physical memory.



IMPORTANT! For `complete` type partitioning, the `<factor>` is not specified.

- `<dimension>`: Specifies which array dimension to partition. Specified as an integer from 1 to `<N>`. Vitis core development kit supports arrays of N dimensions and can partition the array on any single dimension.

Example 1

For example, consider the following array declaration.

```
int buffer[16];
```

The integer array, named `buffer`, stores 16 values that are 32-bits wide each. Cyclic partitioning can be applied to this array with the following declaration.

```
int buffer[16] __attribute__((xcl_array_partition(cyclic,4,1)));
```

In this example, the cyclic `<partition_type>` attribute tells the Vitis compiler to distribute the contents of the array among four physical memories. This attribute increases the immediate memory bandwidth for operations accessing the array `buffer` by a factor of four.

All arrays inside of a compute unit in the context of the Vitis core development kit are capable of sustaining a maximum of two concurrent accesses. By dividing the original array in the code into four physical memories, the resulting compute unit can sustain a maximum of eight concurrent accesses to the array `buffer`.

Example 2

Using the same integer array as found in Example 1, block partitioning can be applied to the array with the following declaration.

```
int buffer[16] __attribute__((xcl_array_partition(block,4,1)));
```

Because the size of the block is four, the Vitis compiler will generate four physical memories, sequentially filling each memory with data from the array.

Example 3

Using the same integer array as found in Example 1, complete partitioning can be applied to the array with the following declaration.

```
int buffer[16] __attribute__((xcl_array_partition(complete, 1)));
```

In this example, the array is completely partitioned into distributed RAM, or 16 independent registers in the programmable logic of the kernel. Because `complete` is the default, the same effect can also be accomplished with the following declaration.

```
int buffer[16] __attribute__((xcl_array_partition));
```

While this creates an implementation with the highest possible memory bandwidth, it is not suited to all applications. The way in which data is accessed by the kernel code through either constant or data dependent indexes affects the amount of supporting logic that the Vitis compiler has to build around each register to ensure functional equivalence with the usage in the original code. As a general best practice guideline for the Vitis core development kit, the complete partitioning attribute is best suited for arrays in which at least one dimension of the array is accessed through the use of constant indexes.

See Also

- [xcl_array_reshape](#)
- [pragma HLS array_partition](#)
- *Vivado Design Suite User Guide: High-Level Synthesis (UG902)*

xcl_array_reshape

Description



IMPORTANT! Array variables only accept one attribute. While the XCL_ARRAY_RESHAPE attribute does support multi-dimensional arrays, you can only reshape one dimension of the array with a single attribute.

This attribute combines array partitioning with vertical array mapping.

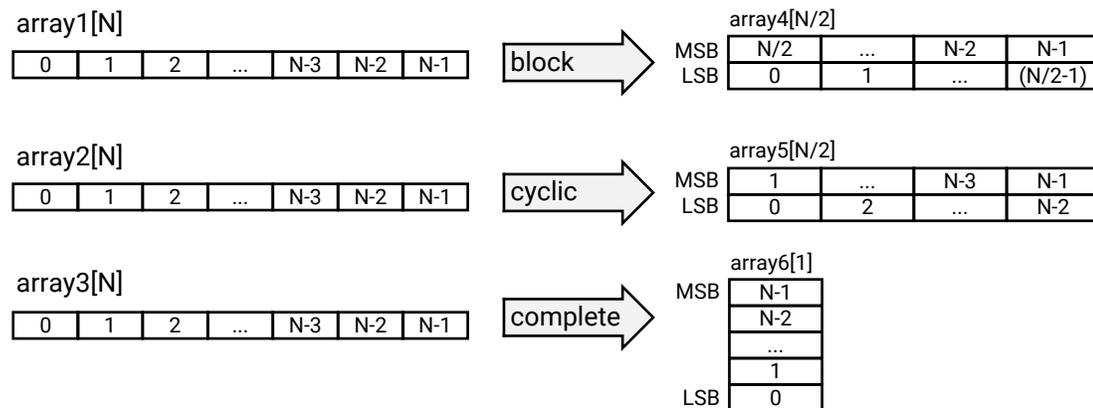
The XCL_ARRAY_RESHAPE attribute combines the effect of XCL_ARRAY_PARTITION, breaking an array into smaller arrays, and concatenating elements of arrays by increasing bit-widths. This reduces the number of block RAM consumed while providing parallel access to the data. This attribute creates a new array with fewer elements but with greater bit-width, allowing more data to be accessed in a single clock cycle.

Given the following code:

```
void foo (...) {
int array1[N] __attribute__((xcl_array_reshape(block, 2, 1)));
int array2[N] __attribute__((xcl_array_reshape(cycle, 2, 1)));
int array3[N] __attribute__((xcl_array_reshape(complete, 1)));
...
}
```

The ARRAY_RESHAPE attribute transforms the arrays into the form shown in the following figure.

Figure 96: ARRAY_RESHAPE



X14307-110217

Syntax

Place the attribute with the definition of the array variable.

```
__attribute__((xcl_array_reshape(<type>,<factor>,<dimension>)))
```

Where:

- `<type>`: Specifies one of the following partition types:
 - `cyclic`: Cyclic partitioning is the implementation of an array as a set of smaller physical memories that can be accessed simultaneously by the logic in the compute unit. The array is partitioned cyclically by putting one element into each memory before coming back to the first memory to repeat the cycle until the array is fully partitioned.
 - `block`: Block partitioning is the physical implementation of an array as a set of smaller memories that can be accessed simultaneously by the logic inside of the compute unit. In this case, each memory block is filled with elements from the array before moving on to the next memory.
 - `complete`: Complete partitioning decomposes the array into individual elements. For a one-dimensional array, this corresponds to resolving a memory into individual registers. The default `<type>` is `complete`.
- `<factor>`: For cyclic type partitioning, the `<factor>` specifies how many physical memories to partition the original array into in the kernel code. For Block type partitioning, the `<factor>` specifies the number of elements from the original array to store in each physical memory.



IMPORTANT! For `complete` type partitioning, the `<factor>` should not be specified.

- `<dimension>`: Specifies which array dimension to partition. Specified as an integer from 1 to `<N>`. The Vitis core development kit supports arrays of `<N>` dimensions and can partition the array on any single dimension.

Example 1

Reshapes (partition and maps) an 8-bit array with 17 elements, `AB[17]`, into a new 32-bit array with five elements using block mapping.

```
int AB[17] __attribute__((xcl_array_reshape(block,4,1)));
```



TIP: A `<factor>` of 4 indicates that the array should be divided into four. As a result, the 17 elements are reshaped into an array of five elements, with four times the bit-width. In this case, the last element, `AB[17]`, is mapped to the lower eight bits of the fifth element, and the rest of the fifth element is empty.

Example 2

Reshapes the two-dimensional array `AB[6][4]` into a new array of dimension `[6][2]`, in which dimension 2 has twice the bit-width:

```
int AB[6][4] __attribute__((xcl_array_reshape(block,2,2)));
```

Example 3

Reshapes the three-dimensional 8-bit array, `AB[4][2][2]` in function `foo`, into a new single element array (a register), 128-bits wide ($4 \times 2 \times 2 \times 8$):

```
int AB[4][2][2] __attribute__((xcl_array_reshape(complete,0)));
```



TIP: A `<dimension>` of 0 means to reshape all dimensions of the array.

See Also

- [xcl_array_partition](#)
- [pragma HLS array_reshape](#)
- *Vivado Design Suite User Guide: High-Level Synthesis (UG902)*

xcl_dataflow

Description

Enables task-level pipelining, allowing functions and loops to overlap in their operation, increasing the concurrency of the RTL implementation, and increasing the overall throughput of the design.

All operations are performed sequentially in a C description. In the absence of any directives that limit resources, such as `pragma HLS allocation`, the Vivado HLS tool seeks to minimize latency and improve concurrency. However, data dependencies can limit this. For example, functions or loops that access arrays must finish all read/write accesses to the arrays before they complete. This prevents the next function or loop that consumes the data from starting operation. The dataflow optimization enables the operations in a function or loop to start operation before the previous function or loop completes all its operations.

When dataflow optimization is specified, the HLS tool analyzes the dataflow between sequential functions or loops and creates channels (based on ping-pong RAMs or FIFOs) that allow consumer functions or loops to start operation before the producer functions or loops have completed. This allows functions or loops to operate in parallel, which decreases latency and improves the throughput of the RTL.

If no initiation interval (number of cycles between the start of one function or loop and the next) is specified, the HLS tool attempts to minimize the initiation interval and start operation as soon as data is available.



TIP: The HLS tool provides dataflow configuration settings. The `config_dataflow` command specifies the default memory channel and FIFO depth used in dataflow optimization. For more information, refer to the *Vivado Design Suite User Guide: High-Level Synthesis* ([UG902](#)).

For the DATAFLOW optimization to work, the data must flow through the design from one task to the next. The following coding styles prevent the HLS tool from performing the DATAFLOW optimization:

- Single-producer-consumer violations
- Bypassing tasks
- Feedback between tasks
- Conditional execution of tasks
- Loops with multiple exit conditions



IMPORTANT! If any of these coding styles are present, the HLS tool issues a message and does not perform DATAFLOW optimization.

For more information, refer to *Vivado Design Suite User Guide: High-Level Synthesis* ([UG902](#)).

Finally, the DATAFLOW optimization has no hierarchical implementation. If a sub-function or loop contains additional tasks that might benefit from the DATAFLOW optimization, you must apply the optimization to the loop, the sub-function, or inline the sub-function.

Syntax

Assign the `XCL_DATAFLOW` attribute before the function definition or the loop definition:

```
__attribute__((xcl_dataflow))
```

Examples

Specifies dataflow optimization within function `foo`.

```
__attribute__((xcl_dataflow))
void foo ( a, b, c, d ) {
    ...
}
```

See Also

- [pragma HLS dataflow](#)
- *Vivado Design Suite User Guide: High-Level Synthesis (UG902)*

xcl_latency

Description

The `XCL_LATENCY` attribute specifies a minimum, or maximum latency value, or both, for the completion of functions, loops, and regions. Latency is defined as the number of clock cycles required to produce an output. Function or region latency is the number of clock cycles required for the code to compute all output values, and return. Loop latency is the number of cycles to execute all iterations of the loop. See "Performance Metrics Example" of *Vivado Design Suite User Guide: High-Level Synthesis (UG902)*.

The Vivado HLS tool always tries to minimize latency in the design. When the `XCL_LATENCY` attribute is specified, the tool behavior is as follows:

- When latency is greater than the minimum, or less than the maximum: The constraint is satisfied. No further optimizations are performed.
- When latency is less than the minimum: If the HLS tool can achieve less than the minimum specified latency, it extends the latency to the specified value, potentially increasing sharing.
- When latency is greater than the maximum: If the HLS tool cannot schedule within the maximum limit, it increases effort to achieve the specified constraint. If it still fails to meet the maximum latency, it issues a warning, and produces a design with the smallest achievable latency in excess of the maximum.



TIP: You can also use the `XCL_LATENCY` attribute to limit the efforts of the tool to find a optimum solution. Specifying latency constraints for scopes within the code: loops, functions, or regions, reduces the possible solutions within that scope and improves tool runtime. For more information, refer to "Improving Run Time and Capacity" of Vivado Design Suite User Guide: High-Level Synthesis ([UG902](#)).

Syntax

Assign the `XCL_LATENCY` attribute before the body of the function, loop, or region:

```
__attribute__((xcl_latency(min, max)))
```

Where:

- `<min>`: Specifies the minimum latency for the function, loop, or region of code.
- `<max>`: Specifies the maximum latency for the function, loop, or region of code.

Example 1

The `for` loop in the `test` function is specified to have a minimum latency of 4 and a maximum latency of 8.

```
__kernel void test(__global float *A, __global float *B, __global float *C,
int id)
{
    for (unsigned int i = 0; i < id; i++)
    __attribute__((xcl_latency(4, 12))) {
        C[id] = A[id] * B[id];
    }
}
```

See Also

- [pragma HLS latency](#)
- *Vivado Design Suite User Guide: High-Level Synthesis* ([UG902](#))

xcl_loop_tripcount

Description

The `XCL_LOOP_TRIPCOUNT` attribute can be applied to a loop to manually specify the total number of iterations performed by the loop.



IMPORTANT! The `XCL_LOOP_TRIPCOUNT` attribute is for analysis only, and does not impact the results of synthesis.

The Vivado High-Level Synthesis (HLS) reports the total latency of each loop, which is the number of clock cycles to execute all iterations of the loop. The loop latency is therefore a function of the number of loop iterations, or tripcount.

The tripcount can be a constant value. It may depend on the value of variables used in the loop expression (for example, $x < y$), or depend on control statements used inside the loop. In some cases, the HLS tool cannot determine the tripcount, and the latency is unknown. This includes cases in which the variables used to determine the tripcount are:

- Input arguments, or
- Variables calculated by dynamic operation.

In cases where the loop latency is unknown or cannot be calculated, the `XCL_LOOP_TRIPCOUNT` attribute lets you specify minimum, maximum, and average iterations for a loop. This lets the tool analyze how the loop latency contributes to the total design latency in the reports, and helps you determine appropriate optimizations for the design.

Syntax

Place the attribute in the OpenCL source before the loop declaration.

```
__attribute__((xcl_loop_tripcount(<min>, <max>, <average>)))
```

Where:

- `<min>`: Specifies the minimum number of loop iterations.
- `<max>`: Specifies the maximum number of loop iterations.
- `<avg>`: Specifies the average number of loop iterations.

Examples

In this example, the `WHILE` loop in function `f` is specified to have a minimum tripcount of 2, a maximum tripcount of 64, and an average tripcount of 33.

```
__kernel void f(__global int *a) {
    unsigned i = 0;
    __attribute__((xcl_loop_tripcount(2, 64, 33)))
    while(i < 64) {
        a[i] = i;
        i++;
    }
}
```

See Also

- [pragma HLS loop_tripcount](#)
- *Vivado Design Suite User Guide: High-Level Synthesis (UG902)*

xcl_max_work_group_size

Description

Use this attribute instead of `REQD_WORK_GROUP_SIZE` when you need to specify a larger kernel than the 4K size.

Extends the default maximum work group size supported in the Vitis core development kit by the `reqd_work_group_size` attribute. Vitis core development kit supports work size larger than 4096 with the `XCL_MAX_WORK_GROUP_SIZE` attribute.

Note: The actual workgroup size limit is dependent on the Xilinx device selected for the platform.

Syntax

Place this attribute before the kernel definition, or before the primary function specified for the kernel:

```
__attribute__((xcl_max_work_group_size(<X>, <Y>, <Z>)))
```

Where:

- `<X>`, `<Y>`, `<Z>`: Specifies the ND range of the kernel. This represents each dimension of a three dimensional matrix specifying the size of the work-group for the kernel.

Example 1

Below is the kernel source code for an un-optimized adder. No attributes were specified for this design, other than the work size equal to the size of the matrices (for example, 64x64). That is, iterating over an entire workgroup will fully add the input matrices, `a` and `b`, and output the result. All three are global integer pointers, which means each value in the matrices is four bytes, and is stored in off-chip DDR global memory.

```
#define RANK 64
__kernel __attribute__((reqd_work_group_size(RANK, RANK, 1)))
void madd(__global int* a, __global int* b, __global int* output) {
    int index = get_local_id(1)*get_local_size(0) + get_local_id(0);
    output[index] = a[index] + b[index];
}
```

This local work size of (64, 64, 1) is the same as the global work size. This setting creates a total work size of 4096.

Note: This is the largest work size that Vitis core development kit supports with the standard OpenCL attribute `REQD_WORK_GROUP_SIZE`. Vitis core development kit supports work size larger than 4096 with the Xilinx attribute `xcl_max_work_group_size`.

Any matrix larger than 64x64 would need to only use one dimension to define the work size. That is, a 128x128 matrix could be operated on by a kernel with a work size of (128, 1, 1), where each invocation operates on an entire row or column of data.

See Also

- <https://www.khronos.org/>
- [The OpenCL C Specification](#)

xcl_pipeline_loop

Description

You can pipeline a loop to improve latency and maximize kernel throughput and performance.

Although unrolling loops increases concurrency, it does not address the issue of keeping all elements in a kernel data path busy at all times. Even in an unrolled case, loop control dependencies can lead to sequential behavior. The sequential behavior of operations results in idle hardware and a loss of performance.

Xilinx addresses this issue by introducing a vendor extension on top of the OpenCL 2.0 API specification for loop pipelining using the XCL_PIPELINE_LOOP attribute.

By default, the `v++` compiler automatically pipelines loops with a trip count more than 64, or unrolls loops with a trip count less than 64. This should provide good results. However, you can choose to pipeline loops (instead of the automatic unrolling) by explicitly specifying the `NOUNROLL` attribute and `XCL_PIPELINE_LOOP` attribute before the loop.

Syntax

Place the attribute in the OpenCL source before the loop definition:

```
__attribute__((xcl_pipeline_loop(<II_number>)))
```

Where:

- `<II_number>`: Specifies the desired initiation interval (II) for the pipeline. The Vivado HLS tool tries to meet this request; however, based on data dependencies, the loop might have a larger initiation interval. When the II is not specified, the default is 1.

Example 1

The following example specifies an `II` target of 3 for the `for` loop in the specified function:

```
__kernel void f(__global int *a) {
    __attribute__((xcl_pipeline_loop(3)))
    for (unsigned i = 0; i < 64; ++i)
        a[i] = i;
}
```

See Also

- [pragma HLS pipeline](#)
- *Vivado Design Suite User Guide: High-Level Synthesis (UG902)*

xcl_pipeline_workitems

Description

Pipeline a work item to improve latency and throughput. Work item pipelining is the extension of loop pipelining to the kernel work group. This is necessary for maximizing kernel throughput and performance.

Syntax

Place the attribute in the OpenCL API source before the elements to pipeline:

```
__attribute__((xcl_pipeline_workitems))
```

Example 1

To handle the `reqd_work_group_size` attribute in the following example, Vitis technology automatically inserts a loop nest to handle the three-dimensional characteristics of the ND range (3,1,1). As a result of the added loop nest, the execution profile of this kernel is like an unpipelined loop. Adding the `XCL_PIPELINE_WORKITEMS` attribute adds concurrency and improves the throughput of the code.

```
kernel
__attribute__((reqd_work_group_size(3,1,1)))
void foo(...)
{
    ...
    __attribute__((xcl_pipeline_workitems)) {
        int tid = get_global_id(0);
        op_Read(tid);
        op_Compute(tid);
        op_Write(tid);
    }
    ...
}
```

Example 2

The following example adds the work-item pipeline to the appropriate elements of the kernel:

```
__kernel __attribute__((reqd_work_group_size(8, 8, 1)))
void madd(__global int* a, __global int* b, __global int* output)
{
    int rank = get_local_size(0);
    __local unsigned int bufa[64];
    __local unsigned int bufb[64];
    __attribute__((xcl_pipeline_workitems)) {
        int x = get_local_id(0);
        int y = get_local_id(1);
        bufa[x*rank + y] = a[x*rank + y];
        bufb[x*rank + y] = b[x*rank + y];
    }
    barrier(CLK_LOCAL_MEM_FENCE);
    __attribute__((xcl_pipeline_workitems)) {
        int index = get_local_id(1)*rank + get_local_id(0);
        output[index] = bufa[index] + bufb[index];
    }
}
```

See Also

- [pragma HLS pipeline](#)
- [Vivado Design Suite User Guide: High-Level Synthesis \(UG902\)](#)

xcl_reqd_pipe_depth

Description



IMPORTANT! Pipes must be declared in lower case alphanumeric. *printf()* is also not supported with variables used in pipes.

The OpenCL framework 2.0 specification introduces a new memory object called pipe. A pipe stores data organized as a FIFO. Pipes can be used to stream data from one kernel to another inside the FPGA without using the external memory, which greatly improves the overall system latency.

In the Vitis core development kit, pipes must be statically defined outside of all kernel functions. The depth of a pipe must be specified by using the XCL_REQD_PIPE_DEPTH attribute in the pipe declaration:

```
pipe int p0 __attribute__((xcl_reqd_pipe_depth(512)));
```

Pipes can only be accessed using standard OpenCL `read_pipe()` and `write_pipe()` built-in functions in non-blocking mode, or using Xilinx-extended `read_pipe_block()` and `write_pipe_block()` functions in blocking mode.



IMPORTANT! A given pipe can have one and only one producer and consumer in different kernels.

Pipe objects are not accessible from the host CPU. The status of pipes can be queried using OpenCL `get_pipe_num_packets()` and `get_pipe_max_packets()` built-in functions. For more details on these built-in functions, see [The OpenCL C Specification](#) from Khronos OpenCL Working Group.

Syntax

This attribute must be assigned at the declaration of the pipe object:

```
pipe int <id> __attribute__((xcl_reqd_pipe_depth(<n>)));
```

Where:

- `<id>`: Specifies an identifier for the pipe, which must consist of lower-case alphanumeric. For example, `<infifo1>` not `<inFifo1>`.
- `<n>`: Specifies the depth of the pipe. Valid depth values are 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768.

Examples

The following is the `dataflow_pipes_ocl` example from [Xilinx GitHub](#) that use pipes to pass data from one processing stage to the next using blocking `read_pipe_block()` and `write_pipe_block()` functions:

```
pipe int p0 __attribute__((xcl_reqd_pipe_depth(32)));
pipe int p1 __attribute__((xcl_reqd_pipe_depth(32)));
// Input Stage Kernel : Read Data from Global Memory and write into Pipe P0
kernel __attribute__((reqd_work_group_size(1, 1, 1)))
void input_stage(__global int *input, int size)
{
    __attribute__((xcl_pipeline_loop))
    mem_rd: for (int i = 0 ; i < size ; i++)
    {
        //blocking Write command to pipe P0
        write_pipe_block(p0, &input[i]);
    }
    // Adder Stage Kernel: Read Input data from Pipe P0 and write the result
    // into Pipe P1
    kernel __attribute__((reqd_work_group_size(1, 1, 1)))
    void adder_stage(int inc, int size)
    {
        __attribute__((xcl_pipeline_loop))
        execute: for(int i = 0 ; i < size ; i++)
        {
            int input_data, output_data;
            //blocking read command to Pipe P0
            read_pipe_block(p0, &input_data);
            output_data = input_data + inc;
            //blocking write command to Pipe P1
            write_pipe_block(p1, &output_data);
        }
    }
}
```

```

}
// Output Stage Kernel: Read result from Pipe P1 and write the result to
// Global Memory
kernel __attribute__((reqd_work_group_size(1, 1, 1)))
void output_stage(__global int *output, int size)
{
__attribute__((xcl_pipeline_loop))
mem_wr: for (int i = 0 ; i < size ; i++)
{
//blocking read command to Pipe P1
read_pipe_block(p1, &output[i]);
}
}
    
```

See Also

- <https://www.khronos.org/>
- [The OpenCL C Specification](#)

xcl_zero_global_work_offset

Description

If you use `clEnqueueNDRangeKernel` with the `global_work_offset` set to `NULL` or all zeros, use this attribute to tell the compiler that the `global_work_offset` is always zero.

This attribute can improve memory performance when you have memory accesses like:

```
A[get_global_id(x)] = ...;
```

Note: You can specify `REQD_WORK_GROUP_SIZE`, `VEC_TYPE_HINT`, and `XCL_ZERO_GLOBAL_WORK_OFFSET` together to maximize performance.

Syntax

Place this attribute before the kernel definition or the primary function specified for the kernel.

```
__kernel __attribute__((xcl_zero_global_work_offset))
void test (__global short *input, __global short *output, __constant short
*constants) { }
```

See Also

- [reqd_work_group_size](#)
- [vec_type_hint](#)
- [clEnqueueNDRangeKernel](#)

Using the Vitis IDE

In the Vitis integrated design environment (IDE), you can create a new application project, or platform development project.

Vitis Command Options

The `vitis` command launches the Vitis integrated development environment (IDE) with your defined options. It provides options for specifying the workspace, and options of the project. The following sections describe the options of the `vitis` command.

Display Options

The following options display the specified information intended for review.

- `-help`: Displays help information for the Vitis core development kit command options.
- `-version`: Displays the Vitis core development kit release version.

Command Options

The following command options specify how the `vitis` command is configured for the current workspace and project.

- `-workspace <workspace location>`: Specify the workspace directory for Vitis IDE projects.
- `{-lp <repository_path>}`: Add `<repository_path>` to the list of Driver/OS/Library search directories.
- `-report <report file>`: Specify the report file to load in Vitis IDE because some reports are more readable or easier to analyze in the IDE.
- `-builddir <build directory>`: Specify the directory containing build results to import as a Vitis Build Results project.

This is typically the directory where you run the Makefile.

- `-eclipseargs <eclipse arguments>`: Eclipse-specific arguments are passed to Eclipse.

- `-vmargs <java vm arguments>`: Additional arguments to be passed to Java VM.

Creating a Vitis IDE Project

In the Vitis IDE, you can create a new application project, or platform development project. The following section shows you how to set up a workspace, create a new Vitis IDE project, and use key features of the IDE.

Launch a Vitis IDE Workspace

1. Launch the Vitis IDE directly from the following command line.

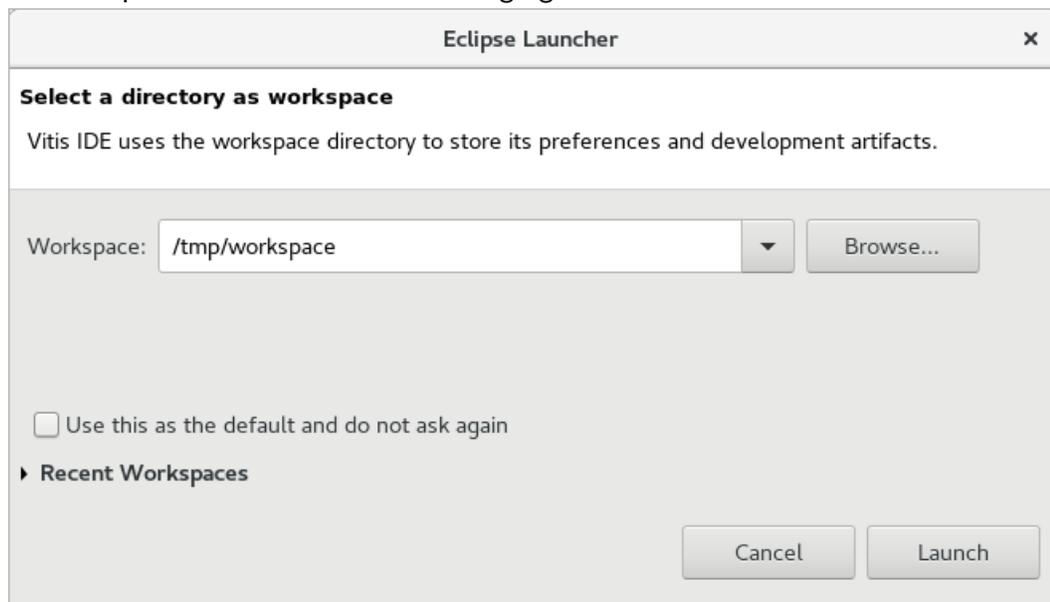
```
$vitis
```



IMPORTANT! When opening a new shell to enter a Vitis core development kit command, ensure that you set it up as described in [Setting up the Vitis Integrated Design Environment](#).

The Vitis IDE opens.

2. Select a workspace as shown in the following figure.



The workspace is the folder that stores your projects, source files, and results while working in the IDE. You can define separate workspaces for each project, or have a single workspace with multiple projects and types. The following instructions show you how to define a workspace for a Vitis IDE project.

3. Click **Browse** to navigate to and specify the workspace, or type the appropriate path in the Workspace field.

4. Select **Use this as the default and do not ask again** to set the specified workspace as your default choice and eliminate this dialog box in subsequent uses of the IDE.
5. Click **Launch**.



TIP: To change the current workspace from within the Vitis IDE, select **File** → **Switch Workspace**.

You have now created a workspace and can populate it with projects.

Create an Application Project



TIP: Example designs are provided with the Vitis core development kit installation, and also on the Xilinx [GitHub](#) repository. For more information, see [Getting Started with Examples](#).

After launching the Vitis IDE, you can create a new Application Project.

1. Select **File** → **New** → **Vitis Application Project**, or if this is the first time the Vitis IDE has been launched, you can select **Create Application Project** on the Welcome screen.

The New Vitis Project wizard opens.

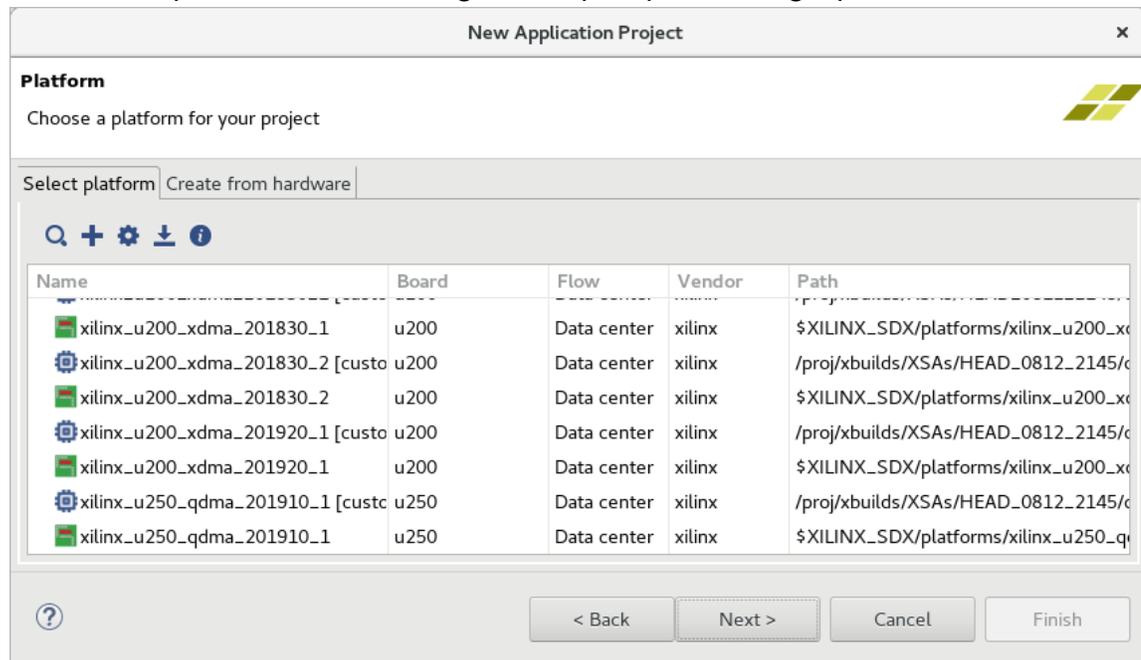
2. In the Create a New Vitis Application Project page, specify the name of the project in the Project name field.

3. The **Use default location** is selected by default to locate your project in a folder in the workspace. You can deselect this check box to specify that the project is created in a location of your choice.
4. If you specify the location, use **Choose file system** to select the default file system, JSch, or enable the Eclipse Remote File System Explorer (RSE).



IMPORTANT! *The project location cannot be a parent folder of a Vitis IDE workspace.*

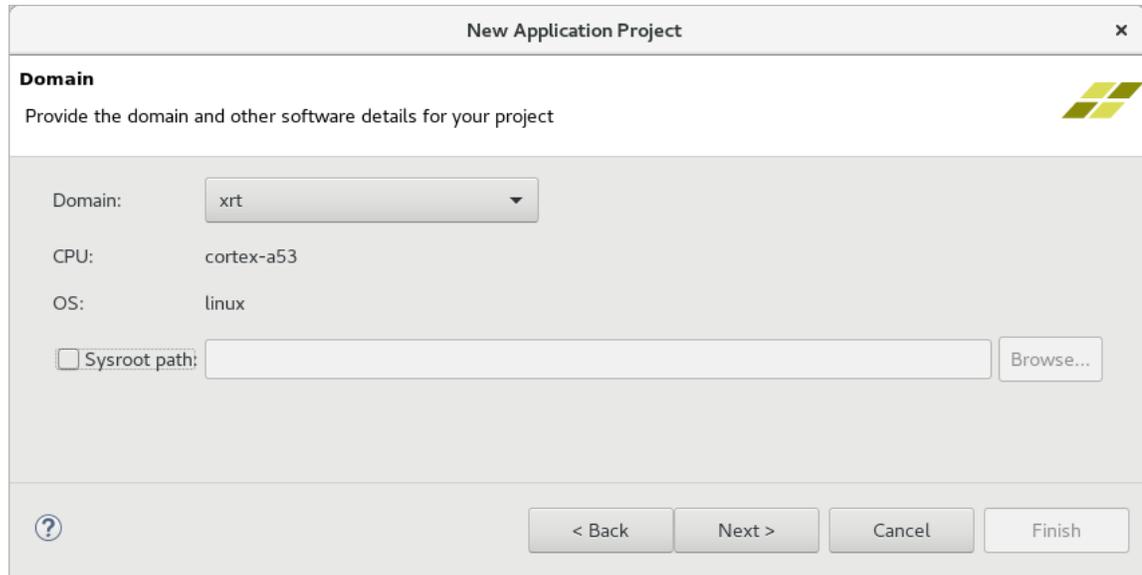
5. Click **Next** to open the Platform dialog box to specify a Vitis target platform.



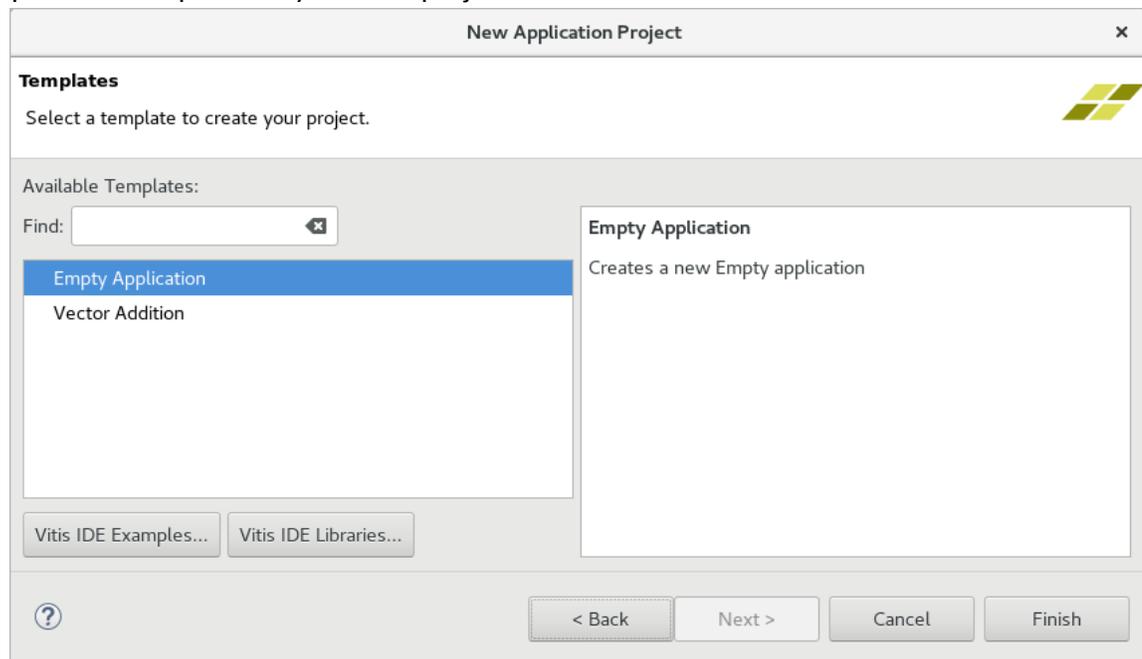
A target platform is composed of a base hardware design and the meta-data used in attaching accelerators to declared interfaces. For target platforms supported by a specific release of the Vitis core tools, refer to [Vitis 2019.2 Software Platform Release Notes](#).

You can also add custom defined or third-party platforms into a repository. For more information, see [Managing Platforms and Repositories](#).

6. Select the target platform and click **Next**.
7. If you select an **Embedded** target platform, as displayed in the Flow column of the Platform dialog box, the Domain page opens as shown in the following figure. Select **Domain** and optionally specify the Sysroot path for the selected platform.
 - The Domain defines the processor domain used for running the host program on the target platform.
 - The `sysroot` is part of the platform where the basic system root file structure is defined. The Sysroot path lets you define a new `sysroot` for your application.



- After clicking **Next**, the Templates page displays, as shown in the following figure. Select an application template for your new project.



- You can select **Empty Application** to create a blank project into which you can import files and build your project from scratch. You can use the Vector Addition template projects as an example to learn about the Vitis IDE, or as a foundation for your new application project.



TIP: Initially, the Template dialog box lists Empty Application and Vector Addition applications. To access additional Vitis IDE examples, click the **Vitis IDE Examples** button and install additional examples as discussed in [Getting Started with Examples](#).

- Click **Finish** to close the New Vitis Project wizard, and open the project.

Managing Platforms and Repositories

You can manage the platforms that are available for use in Vitis IDE projects, from **Xilinx** → **Add Custom Platform** in the main menu of an open project, or from the Platform dialog box as shown in [Create an Application Project](#). This lets you add a new platform, or a new platform repository.

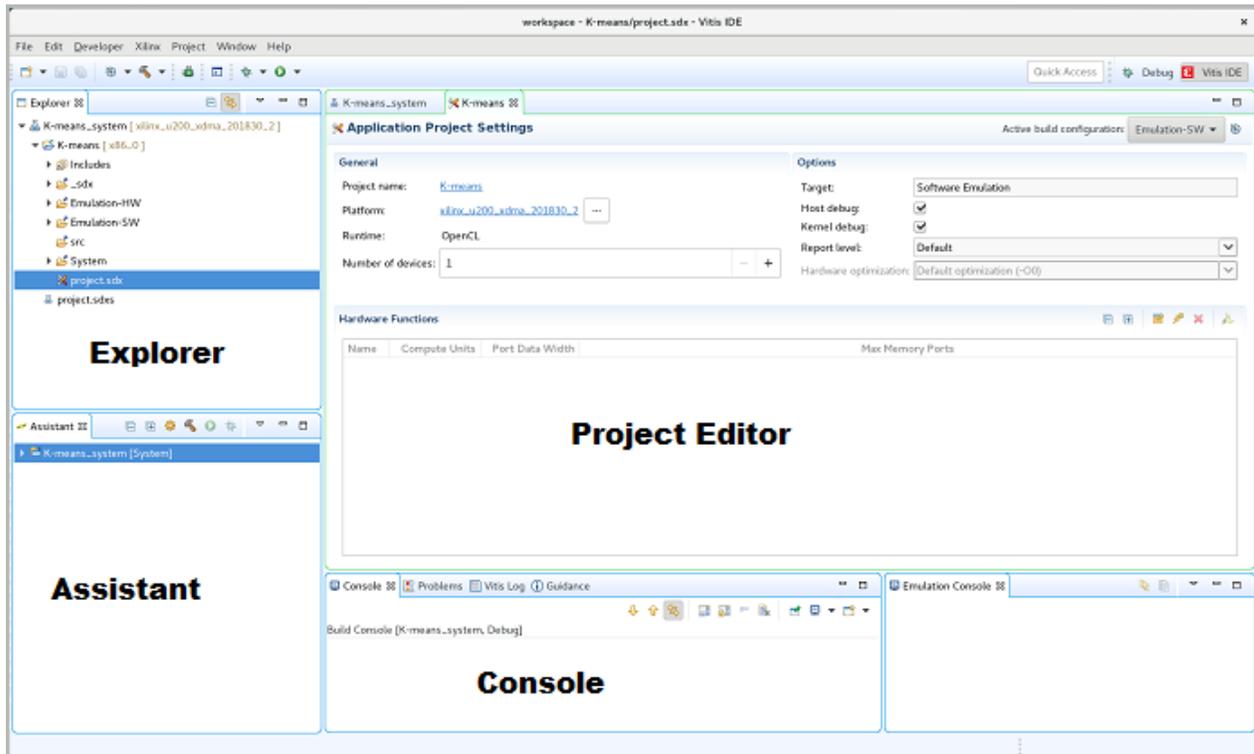
From the Platform dialog box, manage the available platforms and platform repositories using one of the following options:

- **Add Custom Platform** (+): Add your own platform to the list of available platforms. To add a new platform, navigate to the top-level directory of the custom platform, select it, and click **OK**. The custom platform is immediately available for selection from the list of available platforms.
- **Manage Platform 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. When a platform is removed from the list of repositories, it no longer displays in the list of available platforms.
- **Add Devices/Platforms** (↓): Manage the Xilinx devices and platforms installed as part of the standard software installation. If a device or platform was not selected during the installation process, you can add it later using this option. This launches the Vitis Installer so you can select extra content to install. To directly add custom platforms to the tool, select **Help** → **Add Devices/Platforms**.

Understanding the Vitis IDE

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

Figure 97: Vitis IDE – Default Perspective



Some key views and editors in the default perspective include:

- **Explorer view:** Displays a file-oriented tree view of the project folders and their associated source files, plus the build files, and reports generated by the tool. You can use this to explore your project file hierarchy.
- **Assistant view:** Provides a central location to view and manage the projects of the workspace, and the build and run configurations of the project. You can interact with the various project settings and reports of the different configurations. From this view, you can build and run your Vitis IDE application projects, and launch the Vitis analyzer to view reports and performance data as explained in [Chapter 7: Using the Vitis Analyzer](#).
- **Project Editor view:** Displays the current project, the target platform, the active build configuration, and specified hardware functions; allows you to directly edit project settings.
- **Console view:** Presents multiple views including the command console, design guidance, project properties, logs, and terminal views.

The Vitis IDE includes several predefined perspectives, such as the Vitis IDE perspective, the Debug perspective, and the Performance Analysis perspective. To quickly switch between perspectives, click the perspective name in the upper right of the Vitis IDE.

You can arrange views to suit your needs by dragging and dropping them into new locations in the IDE, and the arrangement of views is saved in the current perspective. You can close windows by selecting the **Close (X)** button on the View tab. You can open new windows by using the **Window → Show View** command and selecting a specific view.

To restore a perspective to the default arrangement of views, make the perspective active and select **Window → Reset Perspective**.

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

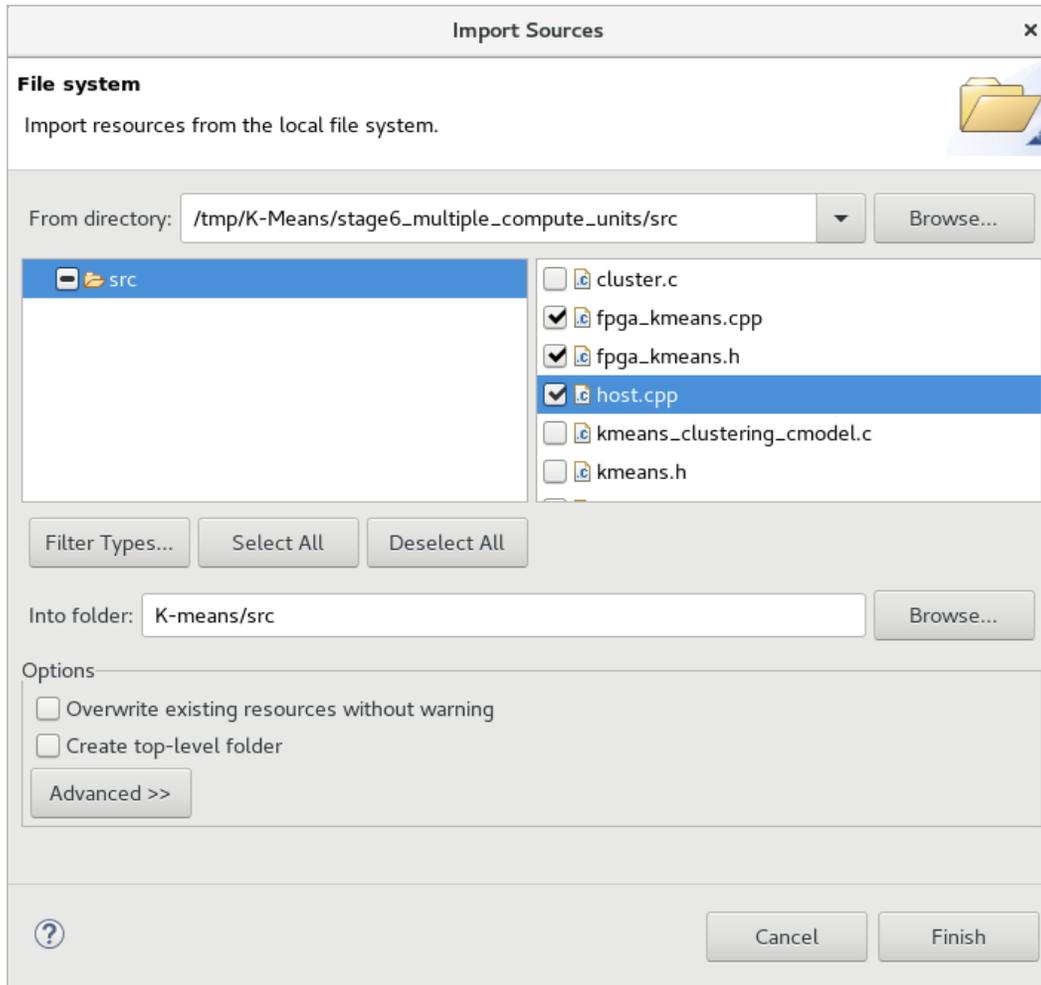
Adding Sources

The project consists of many different source files, including C/C++ files and headers, OpenCL files and headers, compiled Xilinx object files (.xo) containing RTL kernels as discussed in [RTL Kernels](#), or HLS kernels as described in [Compiling Kernels Directly in Vivado HLS](#).

Add Source Files

1. With the project open in the Vitis IDE, to add source files, right-click the `src` folder in the Project Explorer, and click **Import Sources**.

This displays the Import Sources dialog box shown in the following figure.



2. In the dialog box, for the From directory field, click the **Browse** command to select the directory from which you will import sources.
3. In the To directory field, make sure the folder specified is the `src` folder of your application project.
4. Select the desired source files, and click **Finish**.
5. Select the desired source files by enabling the check box next to the file name, and click **Finish**.



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

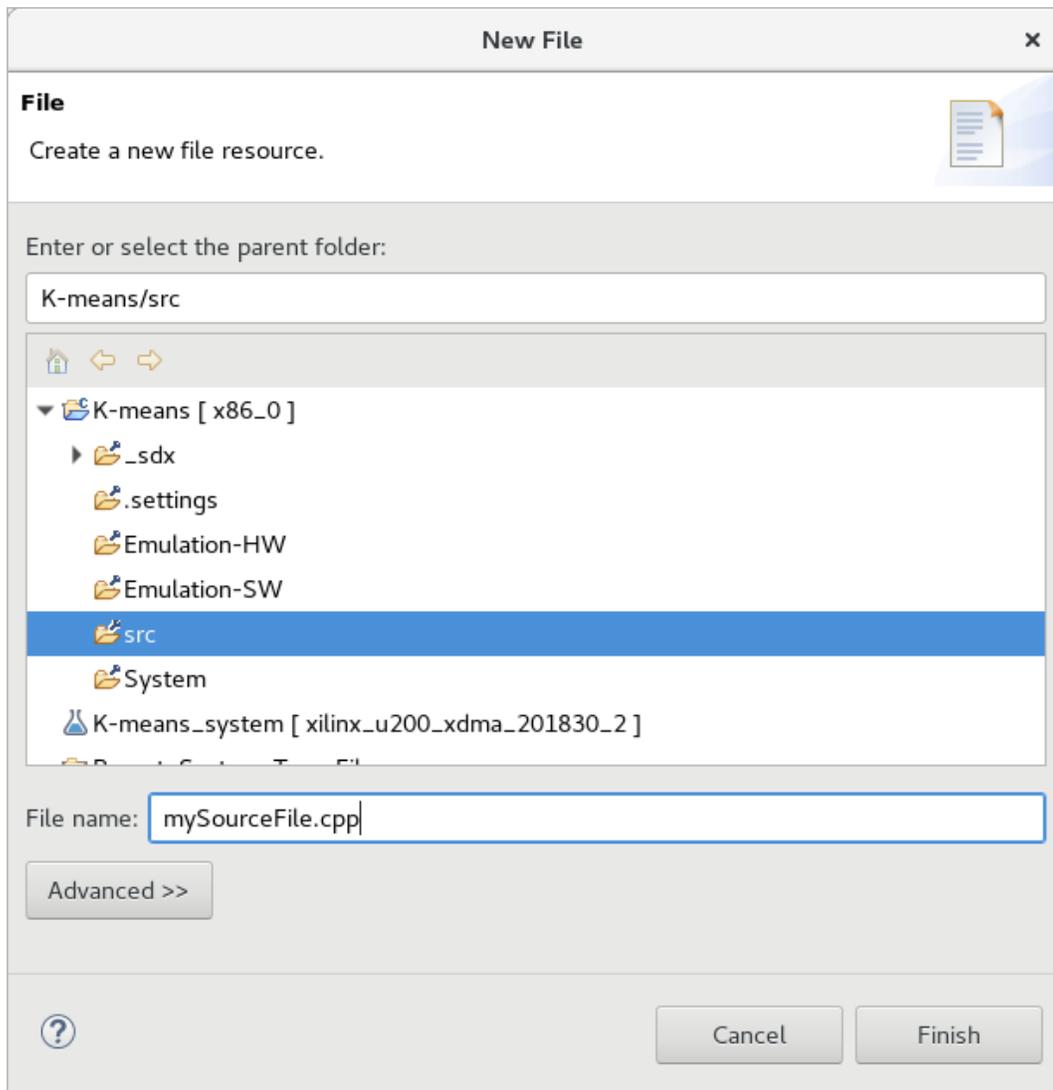
After adding source files to your project, you are ready to begin configuring, building, and running the application. To open a source file in the built-in text editor, expand the `src` folder in the Project Explorer and double-click on a specific file.

Create and Edit New Source Files

In addition to importing source files, you can create and edit new source files directly in the Vitis IDE.

1. From the open project, right-click the `src` folder and select **New → File** from the menu.

The New File dialog box is displayed as shown in the following figure.



2. Select the folder in which to create the new file and enter a file name.
3. Click **Finish** to add the file to the project.

After adding source files to your project, you are ready to begin configuring, building, and running the application. To open a source file in the built-in text editor, expand the `src` folder in the Project Explorer and double-click on a specific file.

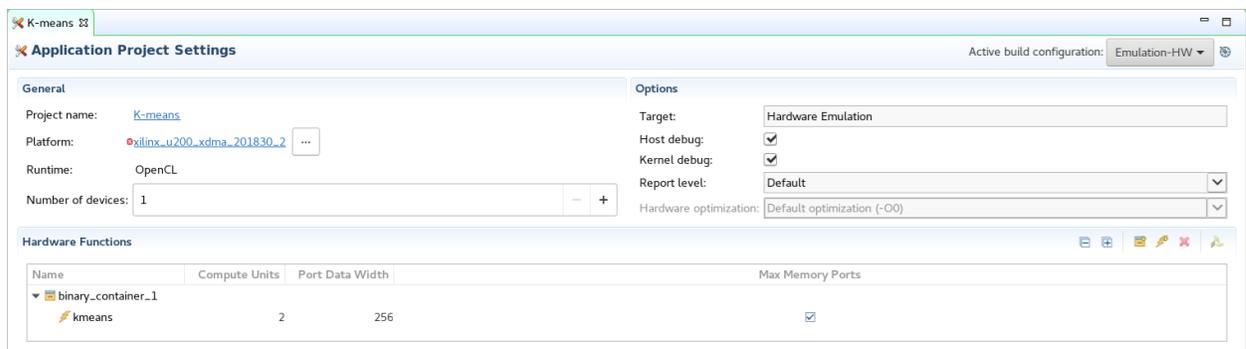
Working in the Project Editor View

Building the system requires compiling and linking both the host program and the FPGA binary (xclbin). Your defined application project includes both the host and kernel code in the `src` folder, as imported or created in the project. The Project Editor view, shown in the following figure, gives a top-level view of the project, and its various build configurations. It provides:

- General information about the project name
- Target platform
- Active build configuration
- Several configuration options related to that build configuration

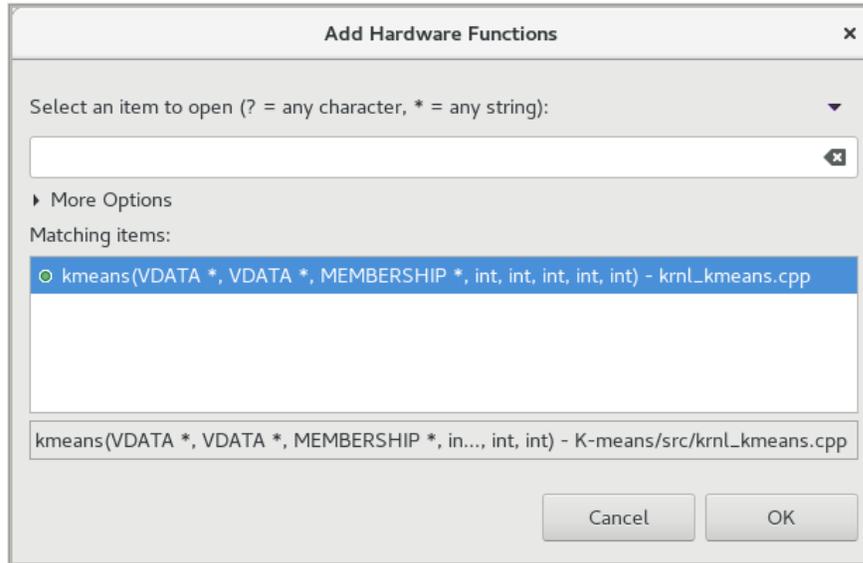
These include debug options to enable debug features of the host program or kernel code, and a menu to select the report level for the build as discussed in [Controlling Report Generation](#).

Figure 98: Project Editor View



The bottom portion of the Editor view displays the Hardware Functions window, which shows the kernels that are assigned to the binary container to be built into the xclbin. To add a kernel to the binary container, click the **Add Hardware Function** (🔧) button in the upper right of the 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 99: Adding Hardware Functions to a Binary Container

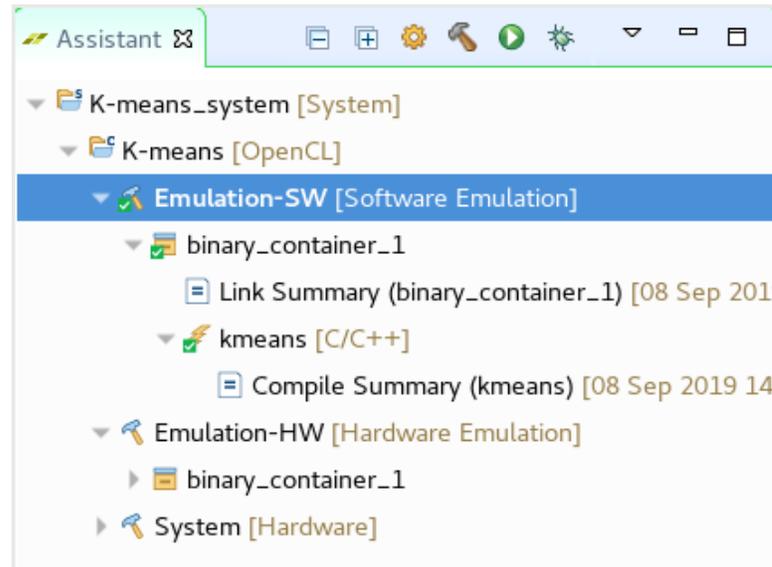


After adding the kernel, in the Hardware Functions window, enter a value under Compute Units to instantiate multiple instances of the kernel as described in [Creating Multiple Instances of a Kernel](#).

Working in the Assistant View

The Assistant view provides a project tree to manage build configurations, run configurations, and set the attributes of these configurations. It is a companion view to the Explorer view and displays directly below the view in the default Vitis IDE perspective. The following figure shows an example Assistant view and its tree structure.

Figure 100: Assistant View Tree Structure Example



The objects displayed in the Assistant view hierarchy include the top-level system project, the application project, the software and hardware emulation build configurations, and the system hardware build configuration.

The build configurations define the build target as described in [Build Targets](#), and specify options for the build process. When you select a build configuration, such as Emulation-HW build and click the **Settings** icon (), the [Vitis Build Configuration Settings](#) dialog box opens. You will use this Settings dialog box to configure the build process for the specific emulation or hardware target.

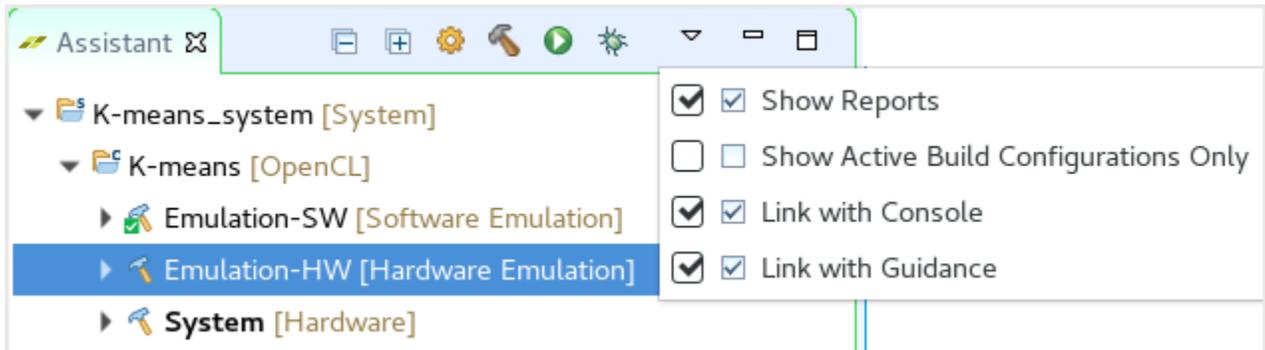


TIP: You can also open the Settings dialog box by double-clicking the configuration object.

Within the hierarchy of each build configuration is the binary container (or .xclbin), the hardware function or functions in the binary container, the run configuration, and any reports or summaries generated by the build or run process. When you select the hardware function for a specific build configuration and click the **Settings** icon, the [Vitis Hardware Function Settings](#) dialog box is displayed. You will use the Hardware Function Settings dialog box to specify the number of compute units for each kernel, assign compute units to SLRs, and assign kernel ports to global memory.

The run configuration is the profile used for running the compiled and linked application; it defines the environment and options for running the application. Selecting a build configuration and using the right-click menu to navigate to **Run** → **Run Configurations**, opens the [Vitis Run Configuration Settings](#) dialog box, where you can configure the run.

Figure 101: Assistant View Menu

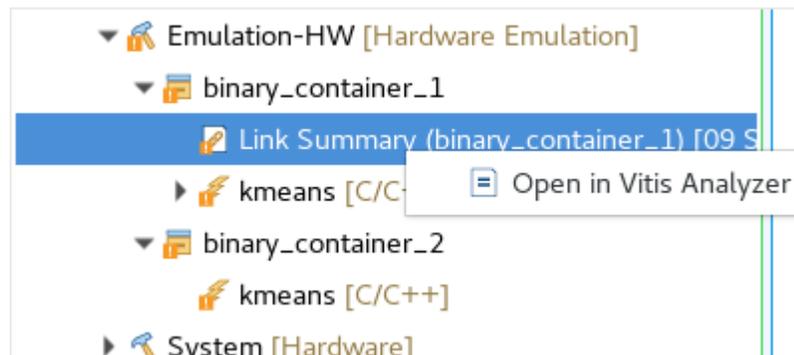


Within the Assistant view, the View menu includes options that affect what the Assistant view displays, which does not affect project data. Open the View menu by left-clicking the downward pointing arrow to display the following options:

- **Show Reports:** When this option is selected, reports are displayed in the Assistant view. Reports open in the tree only when they exist in the project, usually after a project has been built or run, with specific settings.

When reports are displayed in the Assistant view, you can right-click a report, and select **Open in Vitis Analyzer** to examine and review the results as explained in [Chapter 7: Using the Vitis Analyzer](#).

Figure 102: Open in Vitis Analyzer



- **Show Active Build Configurations Only:** When enabled, the Assistant view will only show the active build configuration for each project. This option can be useful to reduce the clutter in the Assistant view. Select **Active build configuration** in the Application Project Settings tab of the Project Editor view.
- **Link with Console:** When enabled, the build console in the Console view switches automatically to match the currently selected build configuration in the Assistant view. If not enabled, the build console does not automatically change to match the Assistant view.
- **Link with Guidance:** When enabled, the Guidance tab of the Console view automatically switches to match the current selection in the Assistant view.

Building the System

When building the system, it best practice to use the three available build targets as described in [Build Targets](#). Each build target is represented in a separate build configuration in the Assistant view. Work through these build configurations in the following order:

- **Emulation-SW:** Build for software emulation (`sw_emu`) to let you confirm the algorithm functionality of both the host program and kernel code working together.
- **Emulation-HW:** Build for hardware emulation (`hw_emu`) to compile the kernel into a hardware description language (HDL), confirm the correctness of the generated logic, and evaluate its simulated performance.
- **System:** Perform a system hardware build (`hw`) to implement the application running on the target platform.

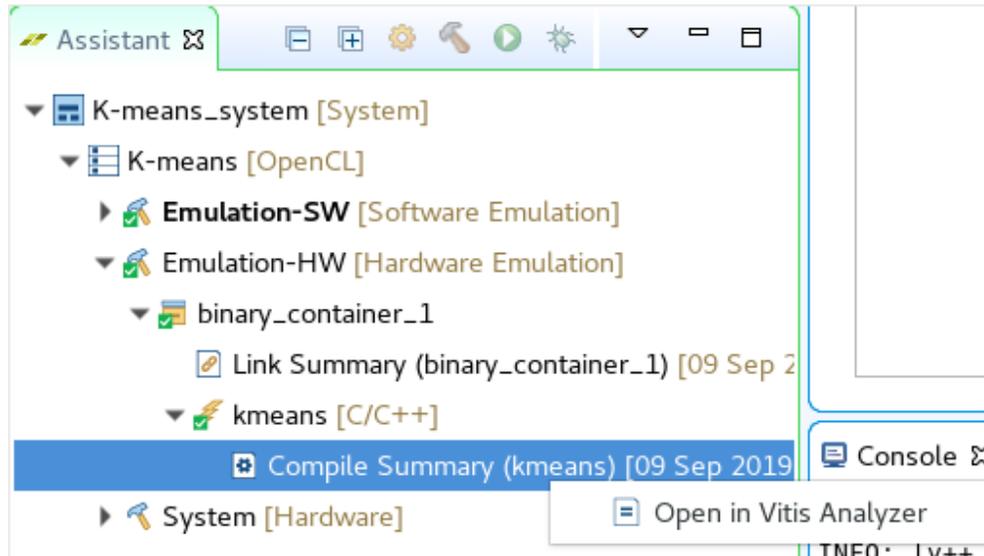
Before launching the build command, configure each of these build configurations to ensure it meets your needs. Select the specific build configuration, and click the **Settings** icon to open the Build Configuration Settings dialog box. For more information on using this dialog box, refer to [Vitis Build Configuration Settings](#).

Beyond the build configuration settings, many of the settings that will affect your application are contained in the Hardware Function, accessed through the [Vitis Hardware Function Settings](#) dialog box. It is a good idea to review each of the Settings dialog boxes as discussed in [Configuring the Vitis IDE](#).

From the Assistant view, with the various options of the build configuration specified, you can start the build process by selecting a build configuration and clicking on the **Build** () button. The Vitis core development kit uses a two part build process that generates the FPGA binary (.xclbin) for the hardware kernels using the Vitis compiler `v++` command, and compiles and links the host program code using the `g++` compiler.

After the build process is complete, the Assistant view shows the specific build configuration with a green check mark to indicate it has been successfully built, as shown in the following figure. You can open any of the build reports, such as the Compile Summary in the hardware function, or the Link Summary in the binary container. Right-click the report in the Assistant view and select **Open in Vitis Analyzer**.

Figure 103: Assistant View - Successful Builds



With the build complete, you can now run the application in the context provided by the specific build configuration. For instance, exercise a C-model of the host program and FPGA binary working together in the Emulation-SW build, or review the host program and the RTL kernel code in simulation in the Emulation-HW build, or run the application on the target platform in the System build.

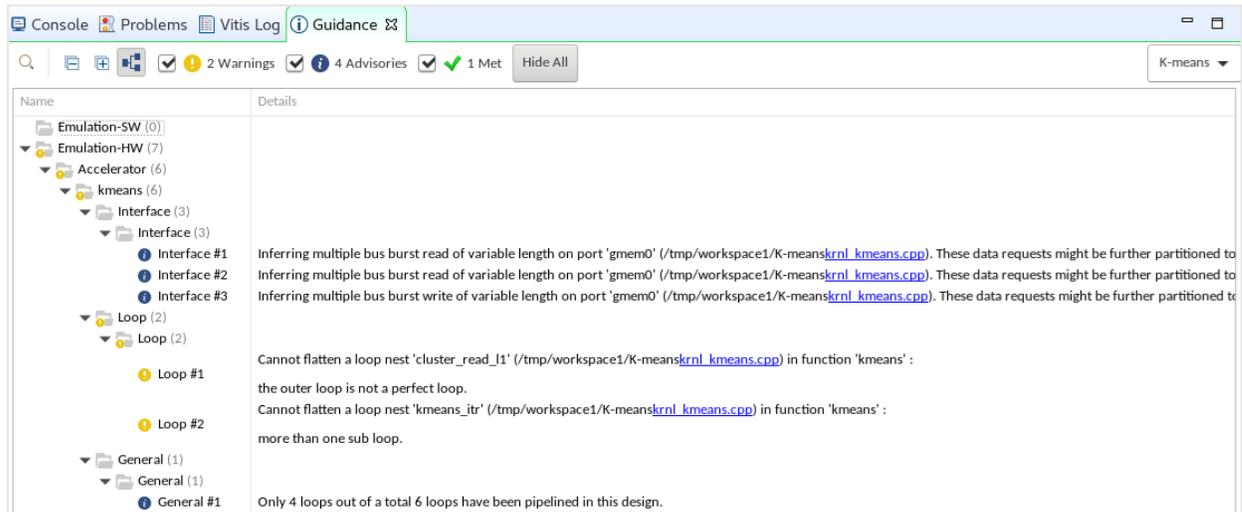
To run the application from within the Vitis IDE, select the build configuration, and click the **Run** button (🟢) to launch the default run configuration. You can also right-click the build configuration and use the **Run** menu to select a specific run configuration, or edit a run configuration as described in [Vitis Run Configuration Settings](#).

Vitis IDE Guidance View

After building or running a specific build configuration, the Guidance tab of the Console view displays a list of errors, warnings, and suggestions related to the run or the build process. The Guidance view is automatically populated and displayed in the tabs located in the Console view. You can review the guidance messages to make any changes that might be needed in your code or build process.

After running hardware emulation, the Guidance view might look like the following figure.

Figure 104: Guidance for the Build



TIP: The Guidance report can also be viewed in Vitis analyzer as discussed in the [Chapter 7: Using the Vitis Analyzer](#).

To simplify sorting through the Guidance view information, the Vitis IDE lets you search, and filter the Guidance view to locate specific guidance rule entries. You can collapse or expand the tree view, or even suppress the hierarchical tree representation and visualize a condensed representation of the guidance rules. Finally, you can select what is shown in the Guidance view by enabling or disabling the display of warnings, as well as rules that have been met, and also restrict the specific content based on the source of the messages such as build and emulation.

By default, the Guidance view shows all guidance information for the project selected in the drop down. To restrict the content to an individual build or run step, do the following:

1. Select **Window** → **Preferences**
2. Select the category **Guidance**.
3. Deselect **Group guidance rule checks by project**.

Working with Vivado Tools from the Vitis IDE

The Vitis core development kit calls the Vivado Design Suite during the linking process to automatically run RTL synthesis and implementation when generating the FPGA binary (xclbin). You also have the option of launching the Vivado tool directly from within the Vitis IDE to interact with the project for synthesizing and implementing the FPGA binary. There are three commands to support interacting with the Vivado tool from the Vitis IDE, accessed through the **Xilinx** → **Vivado Integration** menu:

- **Open Vivado Project:** This automatically opens the Vivado project (.xpr) associated with the System build configuration. In order for this feature to work, you must have previously completed the System build so that a Vivado project exists for the build.

Opening the Vivado project launches the Vivado IDE and opens the implementation design checkpoint (DCP) file to edit the project, to let you manage the results of synthesis and implementation more directly. You can then use the results of this effort for generating the FPGA binary by using the **Import Design Checkpoint** command.

- **Import Design Checkpoint:** Lets you specify a Vivado design checkpoint (DCP) file to use as the basis for the System build, and for generating the FPGA binary.
- **Import Vivado Settings:** Lets you import a configuration file, as described in [Vitis Compiler Configuration File](#), for use during the linking process.

Using the Vivado IDE in standalone mode enables the exploration of various synthesis and implementation options for further optimizing the kernel for performance and area. There are additional options available to let you interact with the FPGA build process.



IMPORTANT! *The optimization switches applied in the standalone project are not automatically incorporated back into the Vitis IDE build configurations. You need to ensure that the various synthesis and implementation properties are specified for the build using the `v++ --config` file options. For more information, refer to [Vitis Compiler Command](#).*

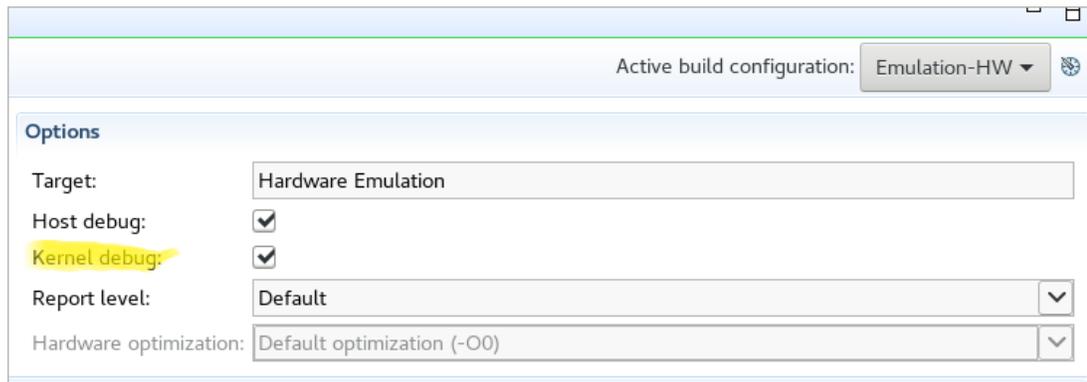
Vitis IDE Debug Flow

The Vitis IDE provides easy access to the debug capabilities. When performed manually, setting up an executable for debugging requires many steps. When you use the debug flow, these steps are handled automatically by the Vitis IDE.

Note: The debug flow in the Vitis IDE relies on shell scripts during debugging. This requires that the setup files such as `.bashrc` or `.cshrc` do not interfere with the environment setup, such as the `LD_LIBRARY_PATH`.

To prepare the executable for debugging, you must change the build configurations to enable the application of debug flags. Set these options in the Project Editor view in the Vitis IDE. There are two check boxes provided in the Options section for the Active build configuration. Host debug enables debugging constructs in the host compilation. Kernel debug enables debugging of the kernels.

Figure 105: Project Editor View - Debug Options



You can also enable the debug features from the Build Configuration Settings dialog box, as shown in [Vitis Build Configuration Settings](#), by selecting the build configuration in the Assistant view and clicking the **Settings** button. Alternatively, you can double-click on the build configuration. The same two check-boxes are presented. While you can enable host debug on all targets, kernel debug is only supported for software emulation and hardware emulation build targets.

Running a GDB session from the Vitis IDE takes care of all the setup required. It automatically manages the environment setup for hardware or software emulation. It configures Xilinx Runtime (XRT) to ensure debug support when the application is running, as described in [xrt.ini File](#), and manages the different consoles required for the execution of the host code, the kernel code, and the debug server.

After setting up the build configuration for debug, clean the build directory and rebuild the application to ensure that the project is ready to run in the GDB debug environment.

To launch a debug session, select the build configuration in the Assistant view and click the **Debug** () button. When launching the debug session in the Vitis IDE, the perspective switches to the Debug perspective, which is configured to present additional windows to manage the different debug consoles and source code windows.

After starting the application, by default the application is stopped right at the beginning of the `main` function body in the host code. As with any GDB graphical front end, you can now set breakpoints and inspect variables in the host code. The Vitis IDE enables the same capabilities for the accelerated kernel implementation in a transparent way. For more information, refer to [Debugging Applications and Kernels](#).

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

Configuring the Vitis IDE

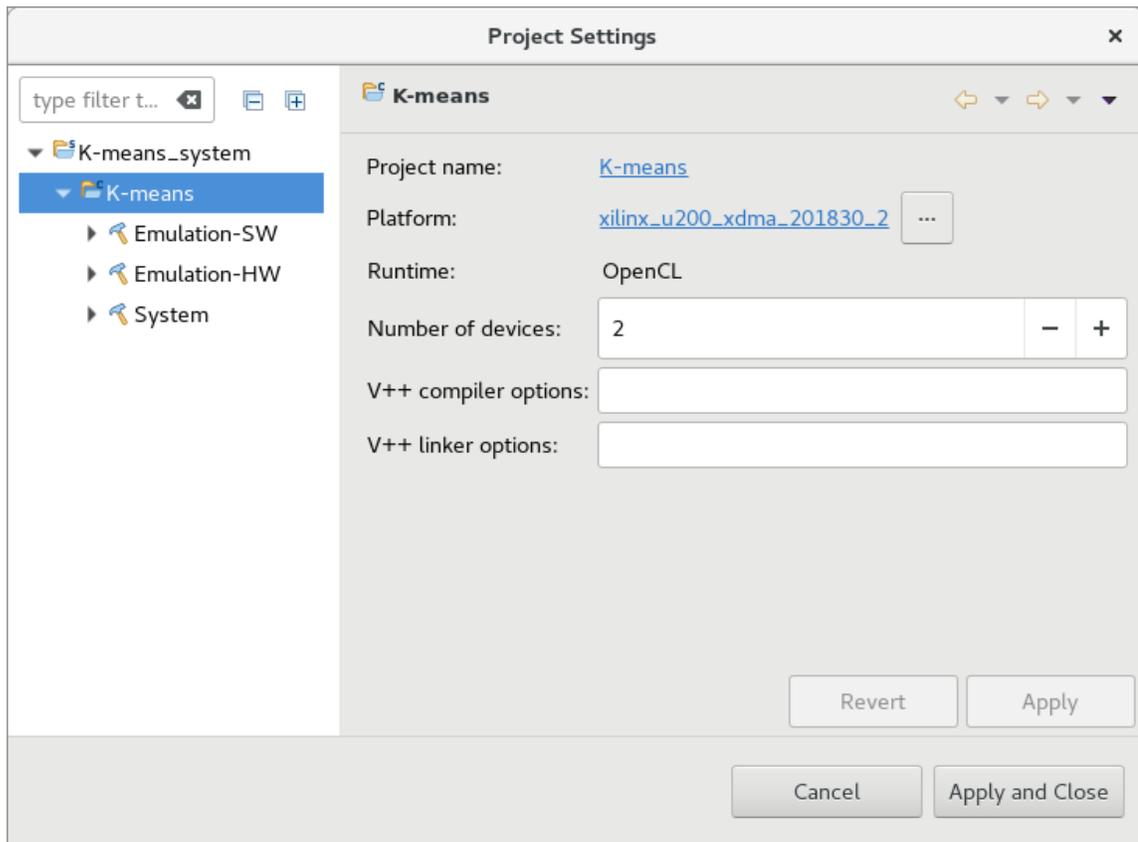
From the Assistant view, use the **Settings** button () to configure a selected project or configuration object. For more information, refer to the following topics:

- [Vitis Project Settings](#)
- [Vitis Build Configuration Settings](#)
- [Vitis Run Configuration Settings](#)
- [Vitis Binary Container Settings](#)
- [Vitis Hardware Function Settings](#)
- [Vitis Toolchain Settings](#)

Vitis Project Settings

To edit the Vitis project settings, select the project in the Assistant view and click the Settings button () to bring up the Project Settings dialog box. This dialog box lets you specify both linking and compile options for the Vitis compiler `v++` command to let you customize the project build process.

Figure 106: Project Settings

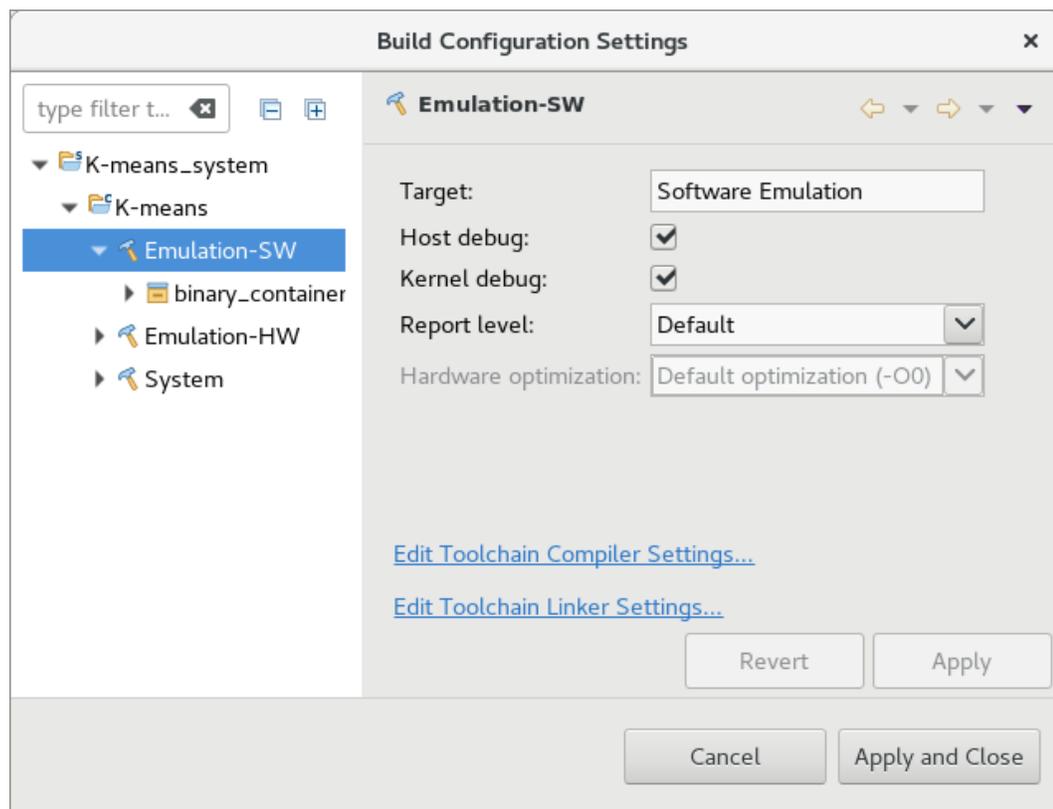


- **Project name:** Name of the project. Click the link to open the Properties dialog box for the project.
- **Platform:** Target platform for this project. Click the link to open the Platform Description dialog box. Click **Browse** to change the platform
- **Runtime:** Displays the runtime used in this project.
- **Number of devices:** Specify the number of OpenCL accelerator devices that can be used by the host program during emulation.
- **V++ compiler options:** Specify compiler options for the `v++` command, as described in [Vitis Compiler Command](#).
- **V++ linker options:** Specify linking options for the `v++` command.

Vitis Build Configuration Settings

To edit the settings for any of the build configurations under the project, select the build configuration in the Assistant view and click the **Settings** button () to bring up the Build Configuration Settings dialog box. In this dialog box, you can enable host and kernel debug, specify the level of information to report during the build process, and specify the level of optimization for the hardware build.

Figure 107: Build Configuration Settings



- **Target:** The build configuration target as described in [Build Targets](#).
- **Host debug:** Select to enable debug of the host code.
- **Kernel debug:** Select to enable debug of the kernel code.
- **Report level:** Specify what report level to generate as described in [Controlling Report Generation](#).
- **Hardware optimization:** Specify how much effort to use on optimizing the hardware. Hardware optimization is a compute intensive task. Higher levels of optimization might result in more optimal hardware but with increased build time. This option is only available in the Build Configuration System.

The Build Configuration dialog box also contains links to the Compiler and Linker Toolchain settings. These provide complete access to all the settings in the standard Eclipse environment, and can be used to configure the Vitis core development kit as described in [Vitis Toolchain Settings](#). Vitis specific settings, such as the Vitis compiler and linker flags, which are not part of the standard C/C++ tool chain, are provided in the Miscellaneous tab.

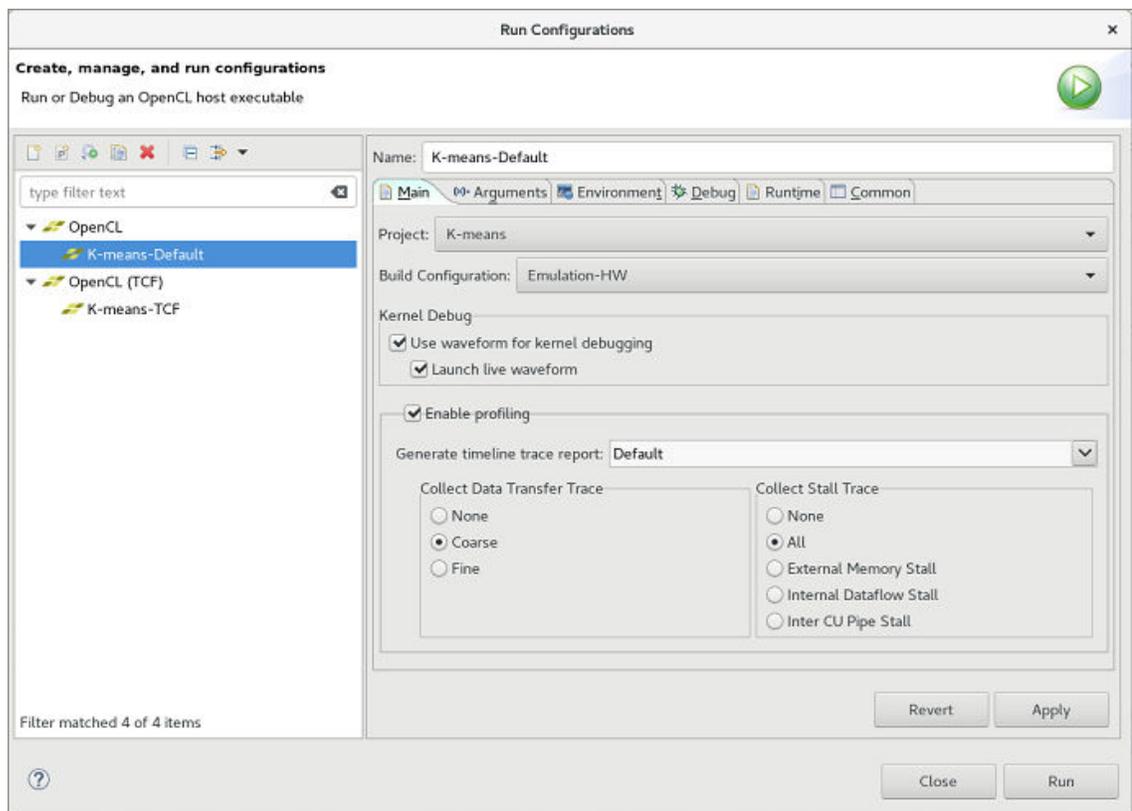
Vitis Run Configuration Settings

To edit the settings for a run configuration, select a build configuration and use the right-click menu to navigate to **Run → Run Configurations** to open the Run Configurations dialog box for the selected build configuration. The Run Configuration dialog box, as shown below, lets you specify debug options, enable profiling of the running application, and specify the types of profiling data to collect.



TIP: Some of the following options are available in the hardware build, or the hardware emulation build, but are not supported in all builds.

Figure 108: Run Configuration Settings

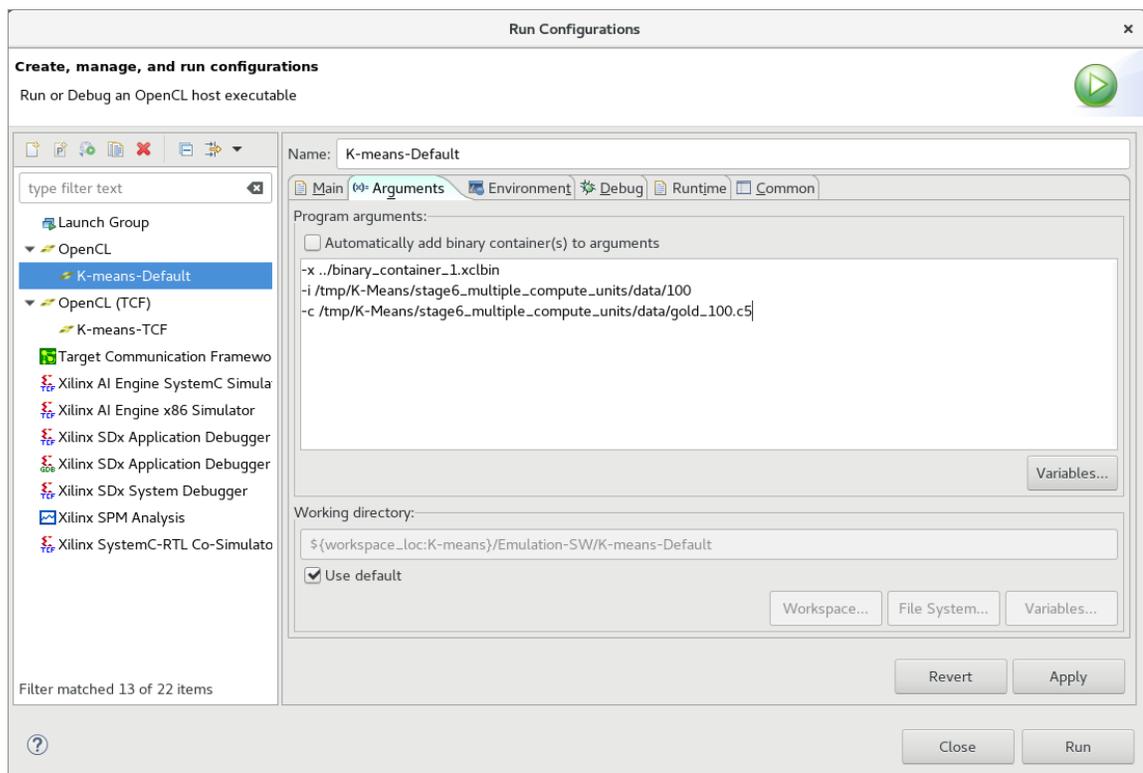


- **Name:** Specifies the name of the run configuration.
- **Project:** Displays the current project, but can be changed to other open projects.

- **Build Configuration:** Specifies the currently selected build configuration, or applies your settings to the active build configuration.
- **Kernel Debug:** Select to enable the waveform view for kernel debugging, or enable the live waveform view as discussed in [Waveform-Based Kernel Debugging](#).
- **Enable profiling:** Enables the collection of profile data when the application is running as discussed in [Profile Summary Report](#).
- **Generate timeline trace:** Enables the production of a timeline trace report for the application as described in [Application Timeline](#).
- **Collect Data Transfer Trace:** Specifies the collection of data transfer data as described in [Vitis Compiler General Options](#), and [xrt.ini File](#).
- **Collect Stall Trace:** Lets you indicate the capture of stall data for a variety of conditions, as describe in [Vitis Compiler General Options](#), and [xrt.ini File](#).

The Run Configuration dialog box has additional tabs to help configure the runtime environment when running your application. In the example below, the Argument tab is shown, with various arguments required to make the application run successfully.

Figure 109: Run Configuration Settings - Arguments

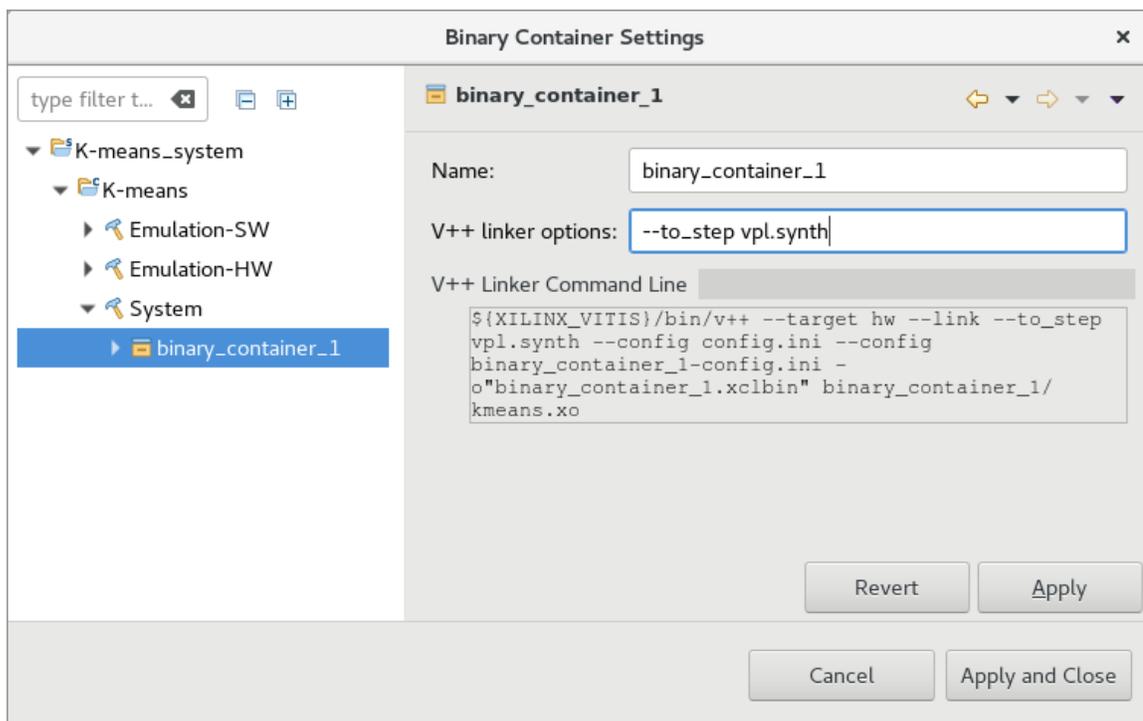


- **Program arguments:** Specify arguments and values required by the application. Options involving paths are specified relative to the Working directory, as shown in the `xclbin` file in the previous example.
- **Automatically add binary containers to arguments:** Enable this check box to have the binary container included automatically.
- **Working directory:** Specify the location for the running application. This is defined automatically by the Vitis IDE, or can be overridden by disabling the Use default check box and specifying a path.

Vitis Binary Container Settings

To edit the settings for any of the binary containers under the project, select the binary container in the Assistant view and click the **Settings** button () to bring up the Binary Container Settings dialog box. This lets you specify a new name for the binary container, and to specify link options for the `v++` command.

Figure 110: Binary Container Settings



- **Name:** Specifies the binary container name.
- **V++ Linker Options:** Enter link options for the selected binary container, in the specified build configuration. For more information on the available options, refer to [Vitis Compiler Command](#).

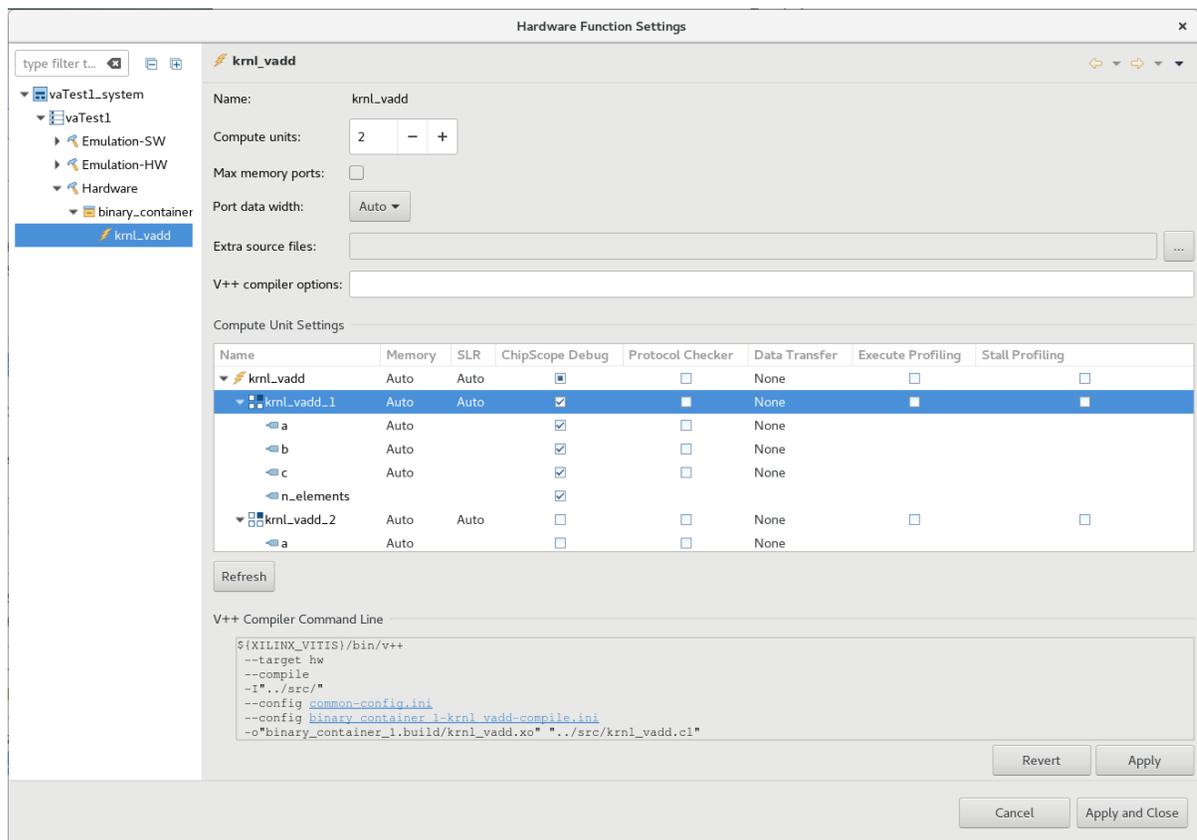
- **V++ Linker Command Line:** Displays the current `v++` command line with any link options you have specified.

Vitis Hardware Function Settings

You can edit the settings for the hardware functions of any build configurations in the project. In the Assistant view, select a hardware function for a specific build configuration, such as

Emulation-HW, and click the **Settings** button () to open the Hardware Function Settings dialog box as shown in the following figure.

Figure 111: Hardware Function Settings



This dialog box lets you set options related to the hardware function in the selected build configuration, such as specifying the number of compute units for a kernel instance, or mapping the kernel ports to specific global memory banks. Specific options include:

- **Compute Units:** Number of compute units to create for the kernel, as described in [Creating Multiple Instances of a Kernel](#).

- **Max Memory Ports:** For OpenCL kernels, when enabled, generates a separate physical memory interface (`m_axi`) for every global memory buffer declared in the kernel function signature. If not enabled, a single physical memory interface is created for the memory mapped kernel ports.
- **Port Data Width:** For OpenCL kernels, specify the width of the data port.
- **Extra Source Files:** Define any additional source files required by this hardware function, such as input data files.
- **Compiler Options:** Specify Vitis compiler options for the selected hardware function.
- **Compute Unit Settings:** Define the SLR placement for each compute unit of the kernel as discussed in [Assigning Compute Units to SLRs](#), and global memory assignments for each port of a compute unit as discussed in [Mapping Kernel Ports to Global Memory](#).
- **Hardware Debug and Profiling Settings:** When the selected build configuration is the System build, the Compute Unit Settings also let you enable debug and profile monitors to be inserted at the interface to each compute unit to capture and review signal traces. These monitors enable the following:
 - **ChipScope Debug:** Add monitors to capture hardware trace debug information.
 - **Protocol Checker:** Add AXI Protocol Checker.
 - **Data Transfer:** Add performance monitors to capture the data transferred between compute unit and global memory. Captured data includes counters, trace, or both.
 - **Execute Profiling:** Add an accelerator monitor to capture the start and end of compute unit executions.
 - **Stall Profiling:** Add an accelerator monitor with functionality to capture stalls in the flow of data inside a kernel, between two kernels, or between the kernel and external memory.



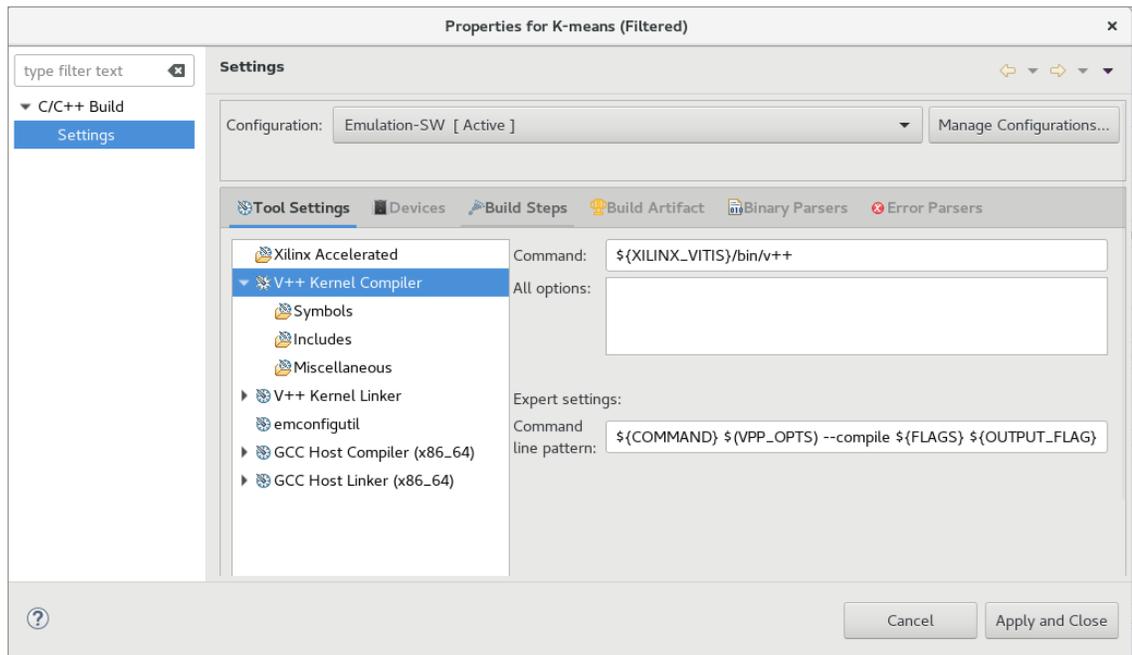
TIP: The settings specified by the Hardware Function Settings dialog box are written to a configuration file used by the Vitis compiler with the `--config` option as described in [Vitis Compiler Configuration File](#).

Vitis Toolchain Settings

The toolchain settings provide a standard Eclipse-based view of the project, providing all options for the C/C++ build in the Vitis IDE.

From the Build Configuration Settings dialog box, click **Edit Toolchain Compiler Settings** or **Edit Toolchain Linker Settings** from the bottom of the Build Configuration window to bring up the compiler and Linker Settings dialog box containing all of the C/C++ build settings. This dialog box lets you set standard C++ paths, include paths, libraries, project wide defines, and host defines.

Figure 112: Toolchain Settings



When working in a Vitis IDE project, the five main settings under the Tool Settings tab are:

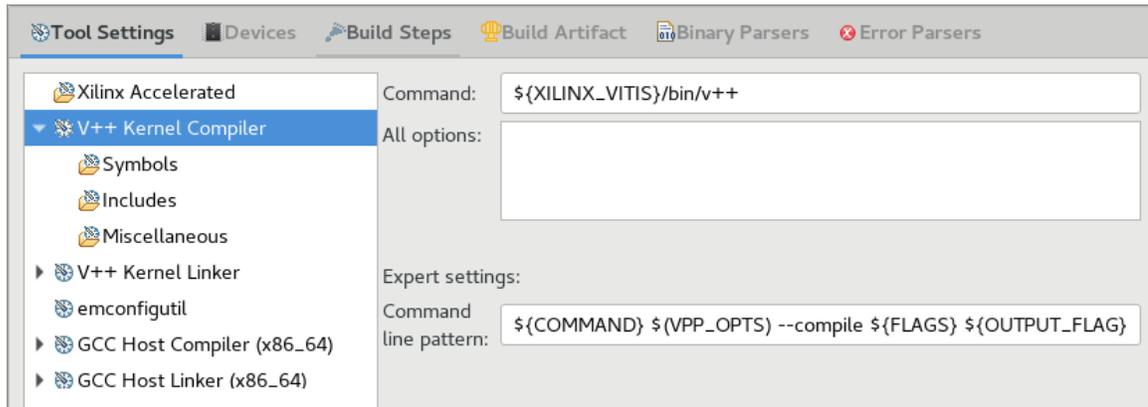
- **V++ Kernel Compiler:** Specify the `v++` command and any additional options that must be passed when calling the `v++` command for the kernel compilation process. See [Vitis Kernel Compiler and Linker Options](#).
- **V++ Kernel Linker:** Specify the `v++` command and any additional options to be passed when calling the `v++` command for the kernel linking process. See [Vitis Kernel Compiler and Linker Options](#),
- **emconfigutil:** Specify the command line options for [Emconfigutil](#). See [emconfigutil Settings](#).
- **GCC Host Compiler (x86_64):** Specify `g++` linker arguments that must be passed during the host compilation process. See [G++ Host Compiler and Linker Settings](#).
- **GCC Host Linker (x86_64):** Specify `g++` linker arguments that must be passed during the host linking process. See [G++ Host Compiler and Linker Settings](#).

Vitis Kernel Compiler and Linker Options

Vitis Kernel Compiler Options

The V++ Kernel compiler section shows the `v++` command and any additional options that must be passed when calling the `v++` command for the kernel compilation process. The `v++` command options can be symbols, include paths, or miscellaneous valid options, which include any of the `v++` command line options you want to add.

Figure 113: Vitis Compiler Options



- **Symbols:** Click **Symbols** under Vitis compiler to define any symbols that are passed with the `-D` option when calling the `v++` command.
- **Includes:** To add include paths to the Vitis compiler, select **Includes** and click the **Add** () button.
- **Miscellaneous:** Any additional compile options that must be passed to the Vitis compiler can be added as flags in the Miscellaneous section. For more information on the available compiler options, refer to [Vitis Compiler Command](#).

Vitis Kernel Linker Options

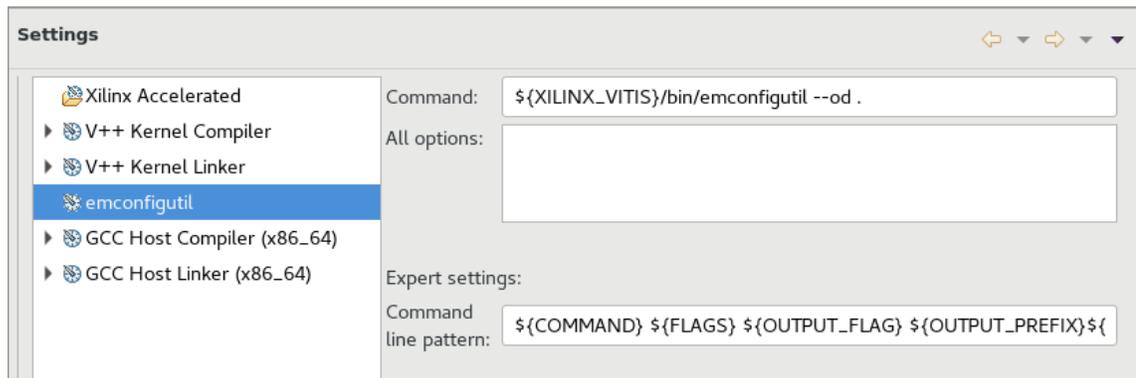
The Vitis kernel linker settings shows the `v++` command and any additional options to be passed when calling the `v++` command for the kernel linking process.

Any additional options that need to be passed to the Vitis compiler can be added as flags in the Miscellaneous section. For more information, refer to [Vitis Compiler Command](#) for the available options in the linking process.

emconfigutil Settings

The `emconfigutil` command options can be entered in the Command field to create an emulation configuration file as described in [Emconfigutil](#). The Vitis IDE creates the `emconfig.json` file by running the specified `emconfigutil` command before launching the run configuration.

Figure 114: emconfigutil Settings

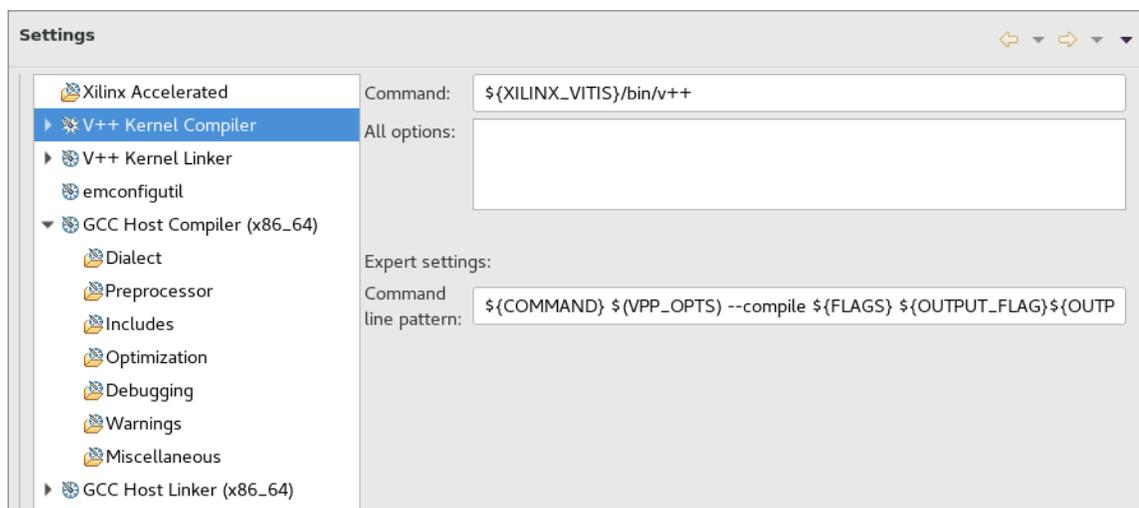


G++ Host Compiler and Linker Settings

G++ Compiler Options

The arguments for the `g++` compiler used by the Vitis core development kit can be accessed under the G++ Host Compiler section of the Toolchain Settings.

Figure 115: G++ Host Compiler Settings



- **Dialect:** Specify the command options that select the C++ language standard to use. Standard dialect options include C++ 98, C++ 2011, and C++ 2014 (1Y).
- **Preprocessor:** Specify preprocessor arguments to the host compiler such as symbol definitions. The default symbols already defined include the platform so that the host code can check for the specific platform.
- **Includes:** Specify the include paths and include files.
- **Optimization:** Specify the compiler optimization flags and other Optimization settings.

- **Debugging:** Specify the debug level and other debugging flags.
- **Warnings:** Specify options related to compiler warnings.
- **Miscellaneous:** Specify any other flags that are passed to the `g++` compiler.

G++ Linker Options

The linker arguments for the Vitis technology G++ Host Linker are provided through the options available here. Specific sections include general options, libraries and library paths, miscellaneous linker options, and shared libraries.

Project Export and Import

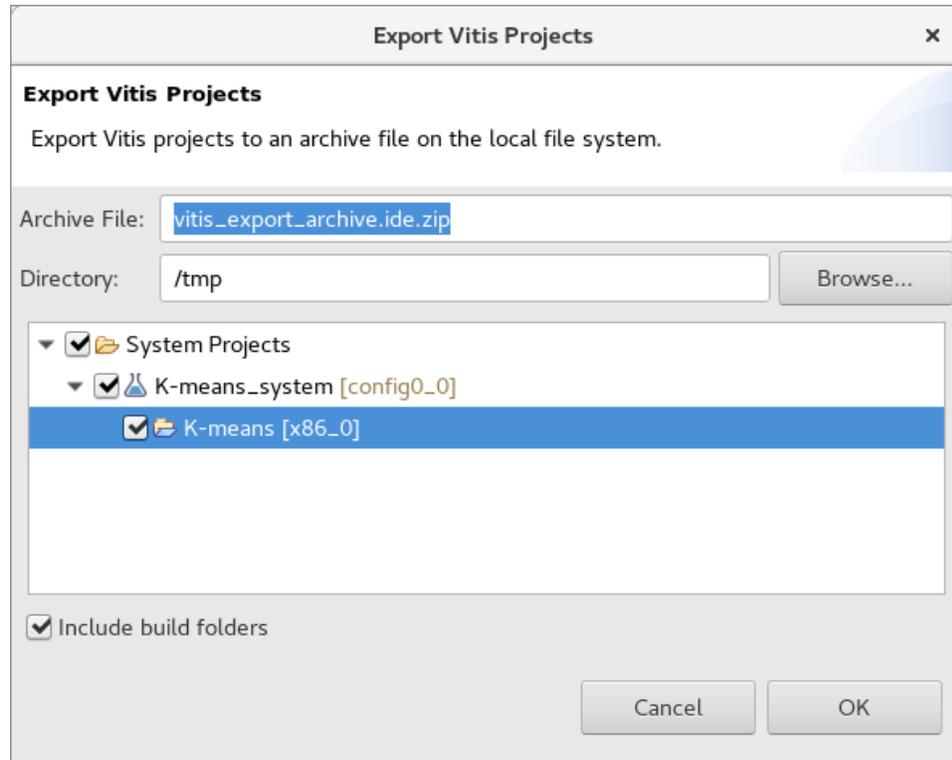
The Vitis IDE provides a simplified method for exporting or importing one or more Vitis IDE projects within your workspace. You can optionally include associated project build folders.

Export a Vitis Project

When exporting a project, the project is archived in a zip file with all the relevant files necessary to import to another workspace.

1. To export a project, select **File** → **Export** from the main menu.

The Export Vitis Projects dialog box opens, where you select the project or projects in the current workspace to export as shown in the following figure.



2. To change the name for the archive, edit the Archive File field.
3. To include the current build configurations, enable **Include build folders** at the bottom of the window.



TIP: This can significantly increase the size of the archive, but may be necessary in some cases.

4. To create the archive with your selected files, click **OK** to create the archive.

The selected Vitis IDE projects will be archived in the specified file and location, and can be imported into the Vitis IDE under a different workspace, on a different computer, by a different user.

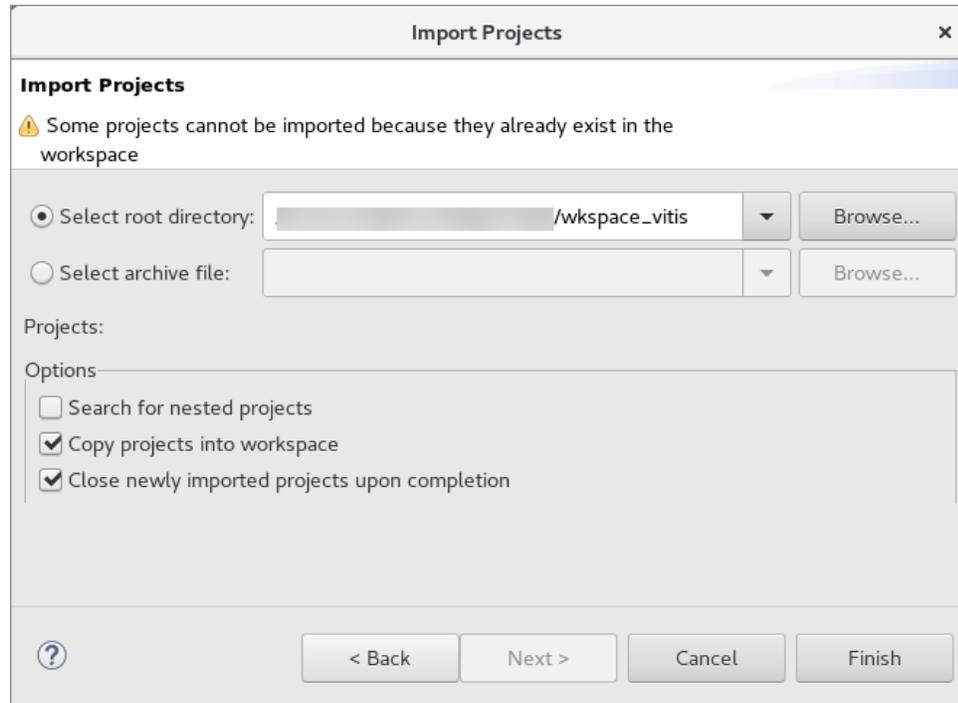
Import a Vitis Project

1. To import a project, select **File** → **Import** from the top menu.

This opens the Import Projects dialog box to select the import file type. There are two types of files you can select to import:

- **Vitis project exported zip files:** Lets you import projects previously exported from the Vitis IDE as discussed in [Export a Vitis Project](#).
- **Eclipse workspace or zip file:** Lets you import projects from another Vitis IDE workspace.

2. The following figure shows the dialog box that is opened when you select **Eclipse workspace or zip file** and click **Next**.



3. For **Select root directory**, point to a workspace for the Vitis IDE, and specify the following options as needed:
 - **Search for nested projects:** Looks for projects inside of other projects in the workspace.
 - **Copy projects into workspace:** Creates a physical copy of the project in the current open workspace.
 - **Close newly created imported projects upon completion:** Closes the projects in the open workspace after they are created.
4. Click **Finish** to import the projects into the open workspace in the Vitis IDE.

Getting Started with Examples

The Vitis core development kit is provided with example designs. These examples can:

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

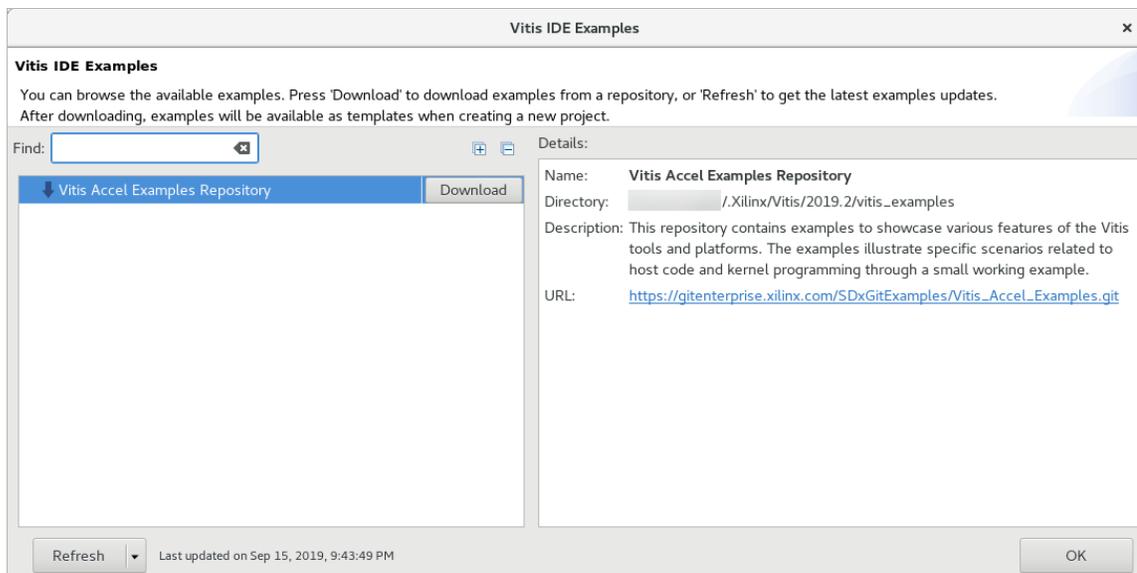
Every target platform provided within the Vitis IDE contains sample designs to get you started, and are accessible through the project creation flow as described in [Create an Application Project](#).

A limited number of sample designs are available in the `<vitis_root>/samples` folder, and many examples are also available for download from the Xilinx GitHub repository. Each of these designs is provided with a Makefile, so you can build, emulate, and run the code entirely on the command line if preferred.

Installing Examples

Select a template for new projects when working through the **New Vitis Project** wizard. You can also load template projects from within an existing project, by selecting **Xilinx → Vitis IDE Examples**.

Figure 116: Vitis IDE Examples – Empty



The left side of the dialog box shows Vitis IDE 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 Vitis IDE Examples to download the examples and populate the dialog box.

The command menu at the bottom left of the Vitis IDE Examples dialog box provides two commands to manage the repository of examples:

- **Refresh:** Refreshes the list of downloaded examples to download any updates from the [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 Vitis IDE always downloads the examples into your local `.Xilinx/vitis/<version>` folder:

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

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

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

When you clone the examples using the `git` command as shown above, you can use the example files as a resource for application and kernel code to use in your own projects. However, many of the files use include statements to include other example files that are managed in the makefiles of the various examples. These include files are automatically populated into the `src` folder of a project when the template is added through the New Vitis Project wizard. To make the files local, locate the files and manually make them local to your project.

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

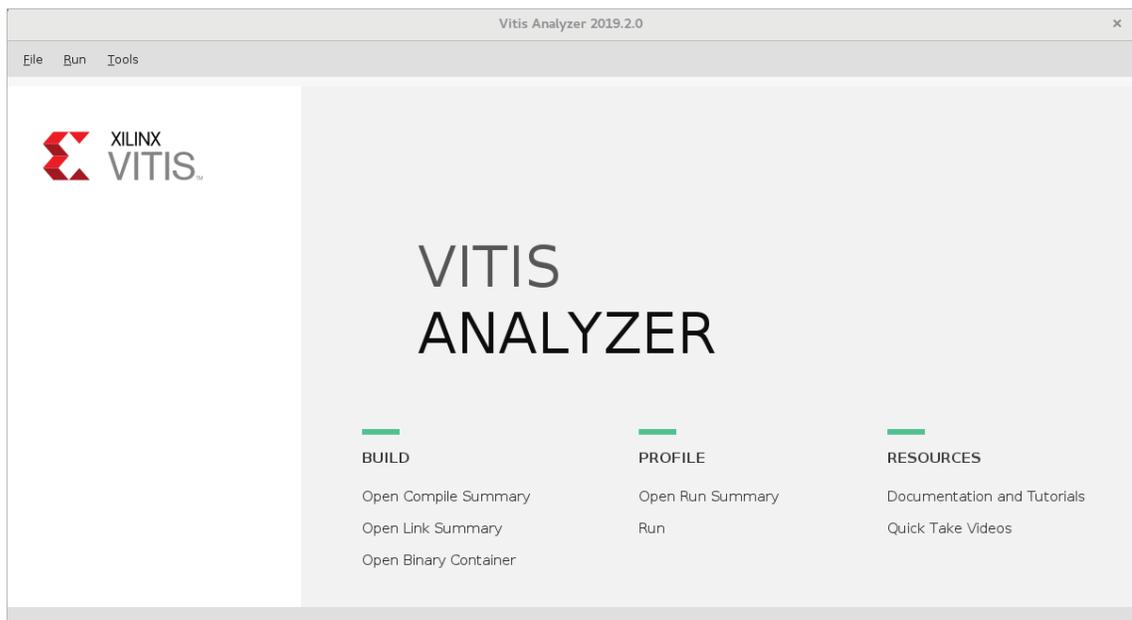
```
find -name xcl2.hpp
```

Using the Vitis Analyzer

The Vitis analyzer is a utility that allows you to view and analyze the reports generated while building and running the application. It is intended to let you review reports generated by both the Vitis compiler when the application is built, and the XRT runtime when the application is run. The Vitis analyzer can be used to view reports from both the `v++` command line flow, and the Vitis integrated design environment (IDE). You will launch the tool using the `vitis_analyzer` command (see [Setting up the Vitis Integrated Design Environment](#)).

When first launched, the Vitis analyzer opens with a home screen that lets you choose from Build and Run reports. Clicking any of these links opens a file browser that allows you to select a specific file of the type described.

Figure 117: Vitis Analyzer – Home Screen



For the Build section:

- **Open Compile Summary:** The Compile Summary report is generated by the `v++` command during compilation, provides the status of the kernel compilation process. When viewing the Compile Summary report the tool also references the following reports generated during compilation: Kernel Estimate, Kernel Guidance, HLS Synthesis, and compilation log.

- **Open Link Summary:** The Link Summary report is created by the `v++` command during linking, provides the status of the FPGA binary build process. When viewing the Link Summary report the tool also references the following reports generated during linking: System Estimate, System Guidance, Timing Summary, Platform and System Diagrams, and linking logs.
- **Open Binary Container:** Opens the selected `xclbin` file to display the Platform Diagram and the System Diagram for the build.

For the Profile section:

- **Open Run Summary:** The Run Summary report is created by XRT during the application execution, and provides a summary of the run process. When viewing the Run Summary report the tool also references the following reports generated during application run: Run Guidance, Profile Summary, Application Timeline, Platform and System Diagrams.
- **Run:** This option lets you select and launch a Run Configuration and displays the command console reflecting a transcript of the run.



TIP: The *Open Recent* section of the home screen provides a list of recently opened summaries and reports to let you quickly reopen them.

The `vitis_analyzer` command allows you to open the tool to the home screen, as discussed above, or specify a file to load when opening the tool. You can specify a file to open using the `-open` option, or by specifying the name of the file to open. You can open the Vitis analyzer with any of the files supported by the tool, as described in [Working with Reports](#). For example:

```
vitis_analyzer profile_summary.csv
```

You can access the command help for `vitis_analyzer` command by typing the following:

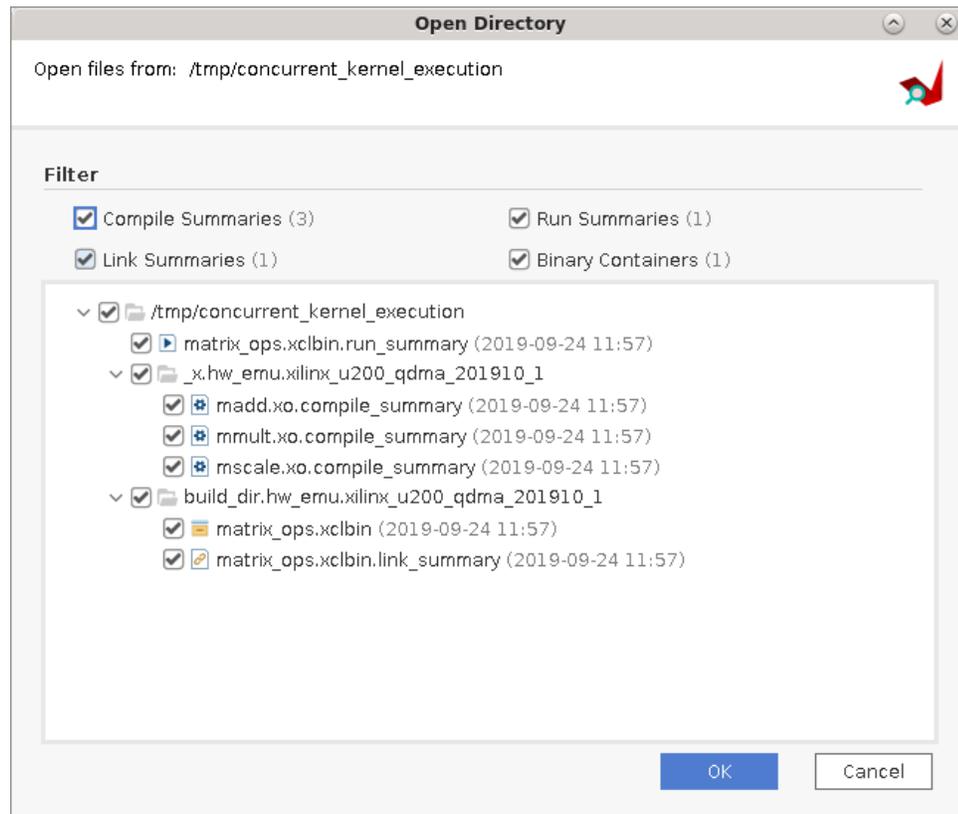
```
vitis_analyzer -help
```

Working with Reports

Generally, the Compile Summary, Link Summary, and Run Summary reports provide you a great overview of the specific steps in building and running the application, as well as related information to get a good view of where the application is with regard to performance and optimization (see the figure below). For individual kernels, start with the Compile Summary. For the FPGA binary (`xclbin`), start with the Link Summary, which also loads the Compile Summaries for included kernels. For the application execution start with the Run Summary.

In addition, the **File** menu offers commands to let you open individual reports, and directories of reports. Refer to [Profiling the Application](#) for more information on the individual reports generated by the build and run processes.

Figure 118: Open Directory



- **Open Directory:** Specifies a directory to open. The tool examines the contents of the directory and displays a dialog box allowing you to select which type of files to open and which individual files to open.
- **Open Binary Container:** FPGA binary, `<name>.xclbin`, created by the compilation and linking process as described in [Chapter 3: Building and Running the Application](#).
- **Open Report File:** Opens one of the report files generated by the Vitis core development kit during compilation, linking, or running the application. The following reports you can open include:
 - **Application Timeline:** Refer to [Application Timeline](#).
 - **Profile Summary:** Refer to [Profile Summary Report](#).
 - **Waveform:** Waveform database or waveform config file as described in [Waveform View and Live Waveform Viewer](#).
 - **Utilization:** A resource utilization report generated by the Vivado® tool when you build the system hardware (HW) target.

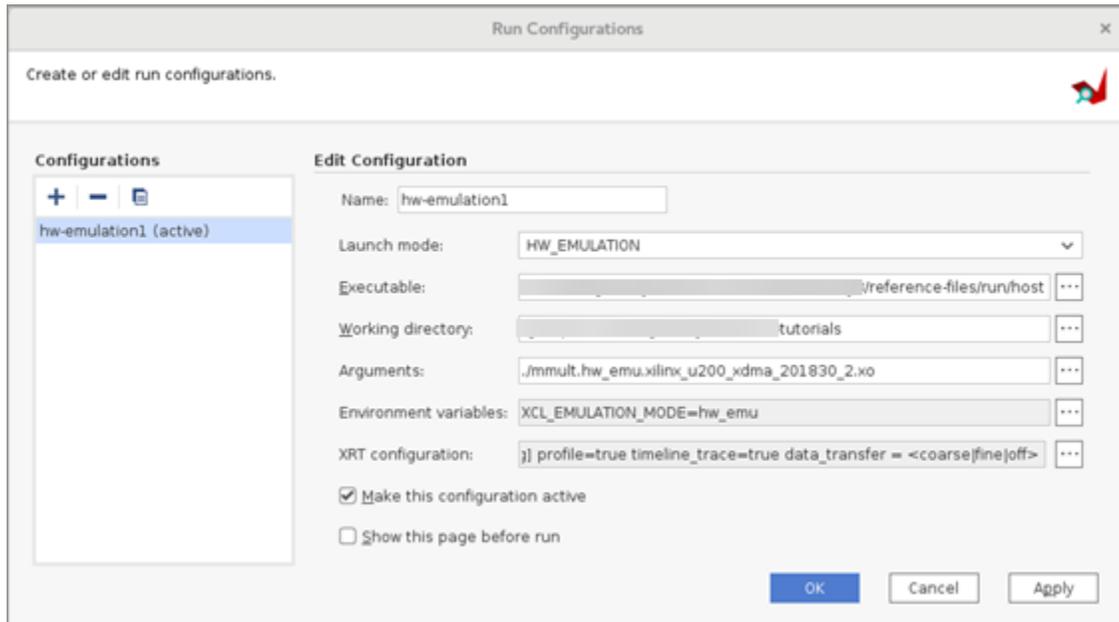
The features of the Vitis analyzer depend on the specific report you are viewing. When the report is structured like a spreadsheet, you interact with the report like a spreadsheet. When the report is graphical in nature, you can interact with the report by zooming into the report to view details, and zooming out to view more information. The Vitis analyzer supports the following mouse strokes to let you quickly zoom into and out of a graphical report:

- **Zoom In:** Press and hold the left mouse button while dragging the mouse from top left to bottom right to define an area to zoom into.
- **Zoom Out:** Press and hold the left mouse button while drawing a diagonal line from lower left to upper right. This zooms out the window by a variable amount. The length of the line drawn determines the zoom factor applied. Alternatively, press **Ctrl** and scroll the wheel mouse button down to zoom out.
- **Zoom Fit:** Press and hold the left mouse button while drawing a diagonal line from lower right to upper left. The window zooms out to display the entire device.
- **Horizontal scrolling:** In a report such as the Application Timeline, you can hold the shift button down while scrolling the middle mouse roller to scroll across the timeline.
- **Panning:** Press and hold the wheel mouse button while dragging to pan.

Creating a Run Configuration

The `vitis_analyzer` tool allows you to create run configurations that specifies how a built configuration is run. You can create and save run configuration builds for software emulation, hardware emulation, and system hardware.

1. From the Run menu, select the **Edit Run Configurations** command to open the Run Configurations dialog box as shown below.



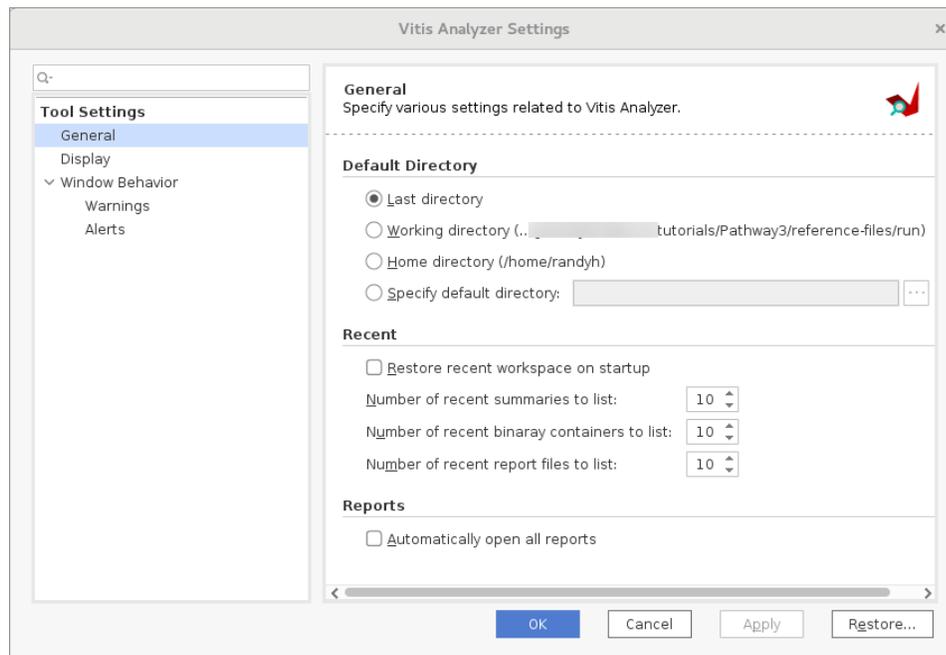
2. The Run Configurations dialog box lets you create new run configurations or edit and manage existing run configurations. Also, you can select a run configuration from the list of **Configurations** in the left pane, and the fields on the right pane under **Edit Configurations** are populated by the settings of the selected configuration. The configuration settings include:
 - **Name:** Specifies the name for the run configuration.
 - **Launch mode:** Specifies the run configuration as an emulation run, or hardware run.
 - **Executable:** This defines the host program used to run the application. Include the full path to the executable, or click ... button to locate the required file.
 - **Working directory:** This is the directory where the run outputs are collected and reports are generated. The specified directory must exist prior to the run, or `vitis_analyzer` will return an error.
 - **Arguments:** These are arguments to the host program specified by the **Executable** field. These arguments can include the `xclbin` file, as is common in Vitis examples, and any additional inputs required by the host program.
 - **Environment variables:** This specifies any environment variables required by the host program. It does not require the definition of `XCL_EMULATION_MODE` as described in [Running an Application](#), as this environment variable is set automatically when running an emulation type build.
 - **XRT configuration:** This is informational only and displays the settings of the `xrt.ini` file that will be used when the run configuration is launched. Refer to [xrt.ini File](#) for more information.

- **Make this configuration active:** This check box specifies that the current run configuration should be the active run in the `vitis_analyzer`. The active run is displayed in the quick start menu command.
 - **Show this page before run:** This check box opens the Run Configuration dialog box with the specified run, which allows you to edit any of the current settings prior to the actual run.
3. After editing the run configuration, select **OK** to accept your changes and close the dialog box.
 4. Select **Apply** to accept your changes and keep the dialog box open, or select **Cancel** to reject your changes and close the dialog box.
 5. If the Run Configuration dialog box has opened prior to launching a selected run, select **Run** to launch the run, or press **Cancel** to close the dialog box without launching the run.
 6. When a run is launched within the `vitis_analyzer`, a console window opens with a transcript of the run, and any reports are written to the specified working directory. Various reports are generated during the run as explained in [Running an Application](#).

Configuring the Vitis Analyzer

The **Tools** → **Settings** command opens the Vitis Analyzer Settings dialog box as shown below.

Figure 119: Settings Dialog Box



In the General settings, the following can be configured:

- **Default Directory:** Specifies the default directory used by the Vitis analyzer when it is opened.
- **Recent:** Configures the tool for the Open Recent listings, and to restore the workspace when you re-open the tool.
- **Reports:** Configures the Vitis analyzer to open all related reports when opening a compile, link, or run summary.

In the Display settings, the following can be configured:

- **Scaling:** Sets the font scaling to make the display easier to read on high resolution monitors. Use OS font scaling uses the value set by the OS for your primary monitor. User-defined scaling allows you to specify a value specific to the Vitis analyzer.
- **Spacing:** Sets the amount of space used by the Vitis (IDE. Comfortable is the default setting. Compact reduces the amount of space between elements to fit more elements into a smaller space.

For Window Behavior settings, the following can be configured:

- **Warnings:** Shows warning when exiting or just exits the Vitis analyzer.
- **Alerts:** Issues an alert when you are running the tool on an unsupported operating system.

After configuring the tool, click **OK**, **Apply**, or **Cancel**. You can also use the **Restore** command to restore the defaults settings of the tool.

Migrating to a New Target Platform

This migration guide is intended for users who need to migrate their accelerated Vitis™ technology application from one target platform to another. For example, moving an application from an Alveo™ U200 Data Center accelerator card, to an Alveo U280 card.

The following 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 target platform which might require additional options to achieve performance.

Design Migration

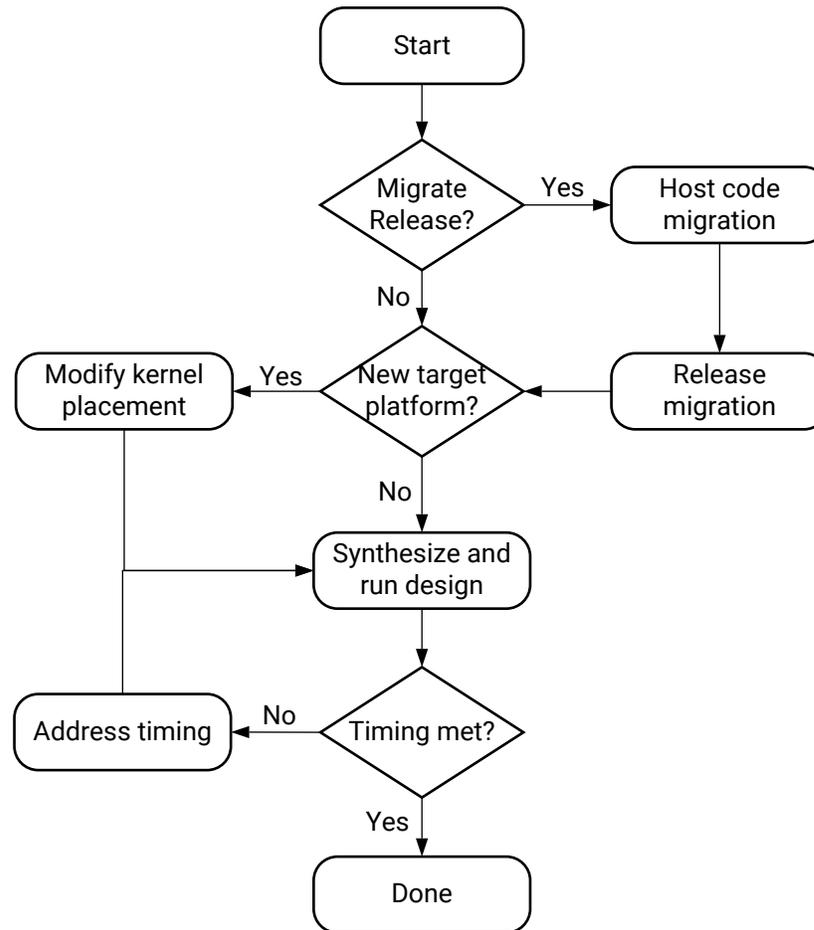
When migrating an application implemented in one target platform to another, it is important to understand the differences between the target platforms and the impact those differences have on the design.

Key considerations:

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

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

Figure 120: Target Platform Migration Flowchart



X21401-092519



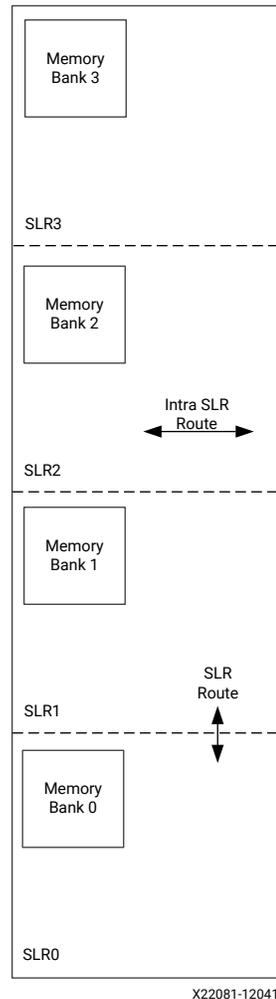
IMPORTANT! Before starting to migrate a design, it is important to understand the architecture of an FPGA and the target platform.

Understanding an FPGA Architecture

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

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

Figure 121: Physical View of Xilinx FPGA with Four SLR Regions



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

Stacked Silicon Interconnect Devices

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

Super Logic Region

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

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

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

SLR Routing Resources

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

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

Memory Interfaces

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

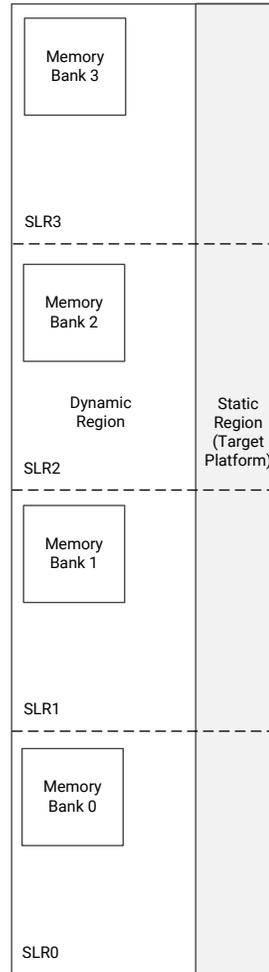
Understanding Target Platforms

In the Vitis technology, a target platform is the hardware design that is implemented onto the FPGA before any custom logic, or accelerators are added. The target platform defines the attributes of the FPGA and is composed of two regions:

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

The figure below shows an FPGA with the target platform applied.

Figure 122: Target Platform on an FPGA with Four SLR Regions



X22082-092519

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

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

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

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

Details on how much logic can be implemented in the dynamic region of each target platform is provided in the [Vitis 2019.2 Software Platform Release Notes](#). This topic is also addressed in [Modifying Kernel Placement](#).

Migrating Releases

Before migrating to a new target platform, you should also determine if you will need to target the new platform to a different release of the Vitis technology. If you intend to target a new release, Xilinx highly recommends to first target the existing platform using the new software release to confirm there are no changes required, and then migrate to a new target platform.

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

- Host Code Migration
- Release Migration



IMPORTANT! Before migrating to a new release, Xilinx recommends that you review the [Vitis 2019.2 Software Platform Release Notes](#).

Host Code Migration

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, source the `<INSTALL_DIR>/settings64.csh` or `<INSTALL_DIR>/settings64.sh` file from the Vitis installation.

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

However, if you are using your own custom `makefile`, you must use the `XILINX_XRT` environment variable to set up the XRT.

- Include directories are now specified as: `-I${XILINX_XRT}/include` and `-I${XILINX_XRT}/include/CL`
- Library path is now: `-L${XILINX_XRT}/lib`
- OpenCL library will be: `libxilinxopencl.so`, use `-lxilinxopencl` in your `makefile`

Release Migration

After migrating the host code, build the code on the existing target platform using the new release of the Vitis technology. Verify that you can run the project in the Vitis unified software platform using the new release, ensure it completes successfully, and meets the timing requirements.

Issues which can occur when using a new release are:

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

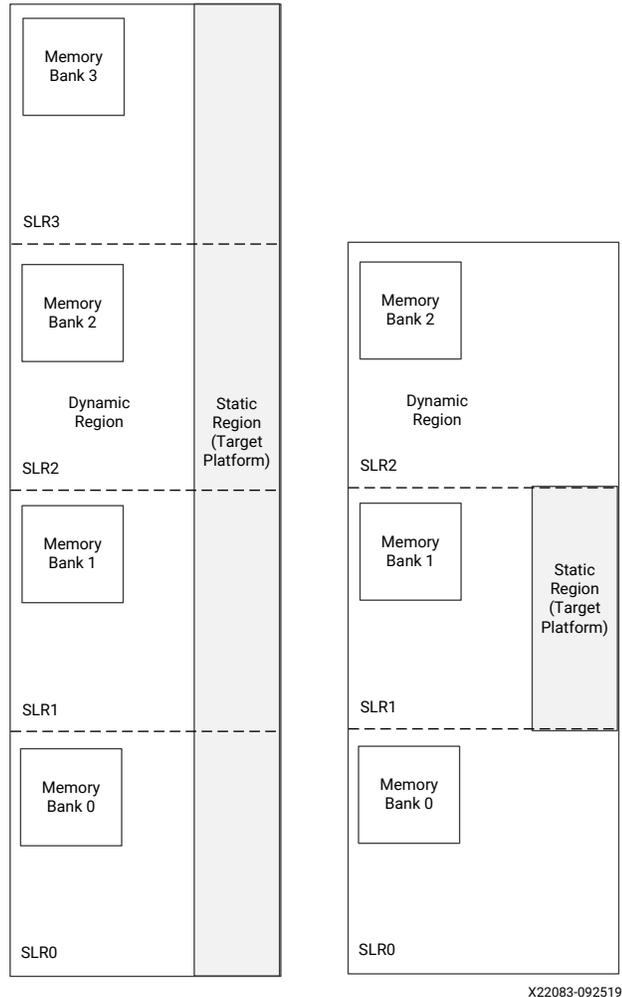
Address these issues using the same techniques you would use during the development of any kernel. At this stage, ensure the throughput performance of the target platform using the new release meets your requirements. If there are changes to the final timing (the maximum clock frequency), you can address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

Modifying Kernel Placement

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

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

Figure 123: Comparison of Target Platforms of the Hardware Platform



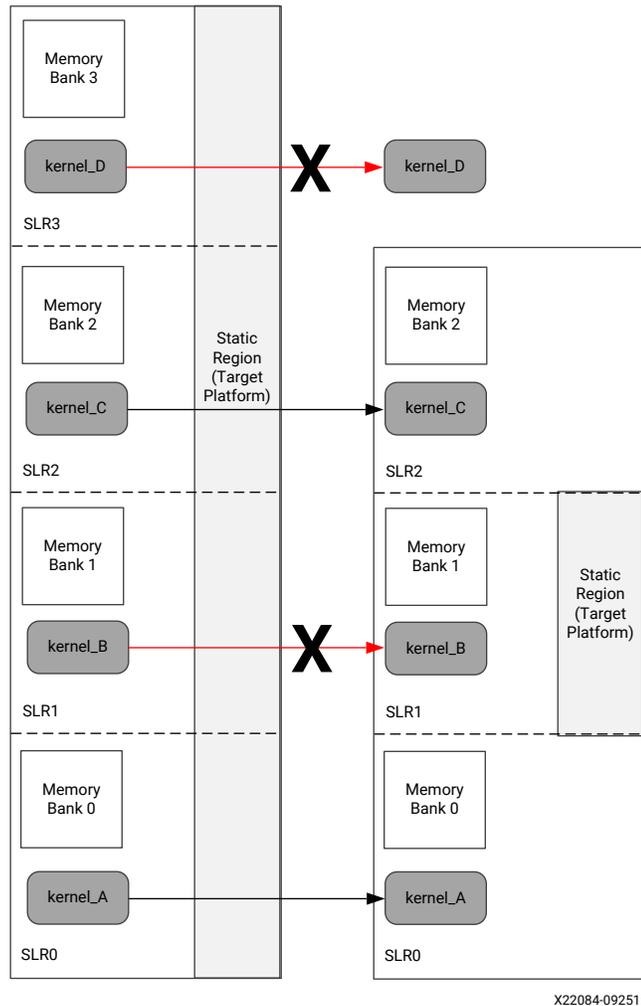
This section explains how to modify the placement of the kernels.

Implications of a New Hardware Platform

The figure below highlights the issue of kernel placement when migrating to a new target platform. 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 124: Migrating Platforms – Kernel Placement



X22084-092519

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

- Understand the resources available in each SLR of the new target platform, as documented in the [Vitis 2019.2 Software Platform Release Notes](#).
- Understand the resources required by each kernel in the design.
- Use the `v++ --config` option to specify which SLR each kernel is placed in, and which DDR bank each kernel connects to. For more details, refer to [Assigning Compute Units to SLRs](#) and [Mapping Kernel Ports to Global Memory](#).

These items are addressed in the remainder of this section.

Determining Where to Place the Kernels

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

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

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

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

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

Available SLR Resources

The resources available in each SLR of the various platforms supported by a release can be found in the [Vitis 2019.2 Software Platform Release Notes](#). The table shows an example target platform. In this example:

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

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

Table 28: 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>]
Global memory resources available in dynamic region			
Memory channels; system port name	bank0 (16 GB DDR4)	bank1 (16 GB DDR4, in static region) 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 125: 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

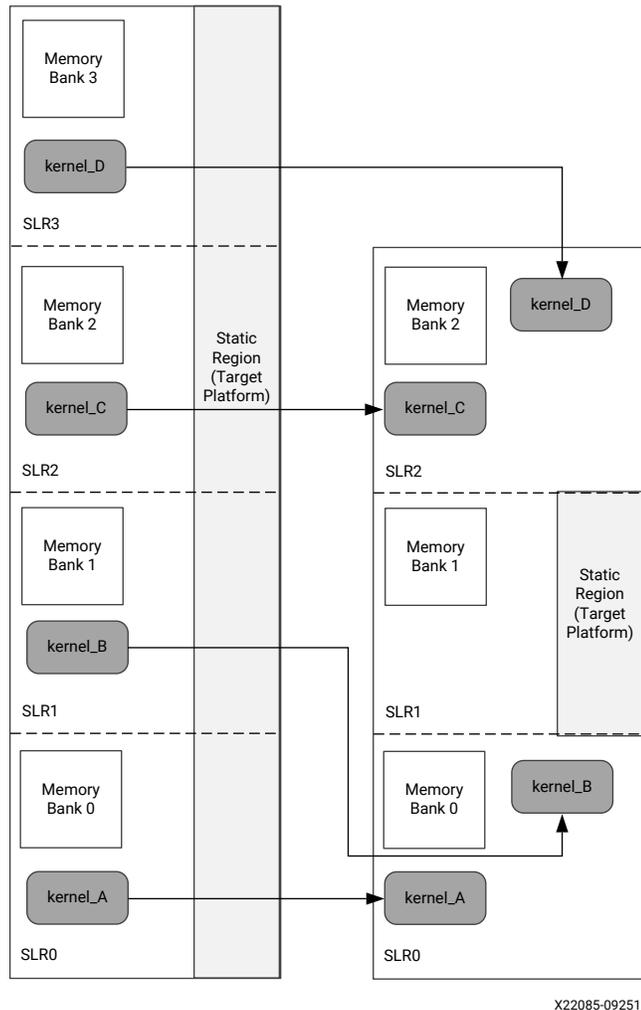
Each kernel in a design can be assigned to an SLR region using the `connectivity.slr` option in a configuration file specified for the `v++ --config` command line option. Refer to [Assigning Compute Units to SLRs](#) for more information.

When placing kernels, Xilinx recommends assigning the specific DDR memory bank that the kernel will connect to using the `connectivity.sp` config option as described in [Mapping Kernel Ports to Global Memory](#).

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

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

The kernel mappings are illustrated in the figure below.

Figure 126: Mapping of Kernels Across SLRs


Specifying Kernel Placement

For the example above, the configuration file to assign the kernels would be similar to the following:

```
[connectivity]
nk=kernel:4:kernel_A.kernel_B.kernel_C.kernel_D

slr=kernel_A:SLR0
slr=kernel_B:SLR0
slr=kernel_C:SLR2
slr=kernel_D:SLR2
```

The `v++` command line to place each of the kernels as shown in the figure above would be:

```
v++ -l --config config.txt ...
```

Specifying Kernel DDR Interfaces

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

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

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

The configuration file to perform these connections would be as follows, and passed through the `v++ --config` command:

```
[connectivity]
nk=kernel:4:kernel_A.kernel_B.kernel_C.kernel_D

slr=kernel_A:SLR0
slr=kernel_B:SLR0
slr=kernel_C:SLR2
slr=kernel_D:SLR2

sp=kernel_A.arg1:DDR[0]
sp=kernel_B.arg1:DDR[1]
sp=kernel_C.arg1:DDR[2]
sp=kernel_D.arg1:DDR[1]
```



IMPORTANT! When using the `connectivity.sp` option to assign kernel ports to memory banks, you must map all interfaces/ports of the kernel. Refer to [Mapping Kernel Ports to Global Memory](#) for more information.

Address Timing

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

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

Custom Constraints

Custom Tcl constraints for floorplanning, placement, and timing of the kernels will need to be reviewed in the context of the new target platform (.xsa). For example, if a kernel needs to be moved to a different SLR in the new target platform, the placement constraints for that kernel will also need to be modified.

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

Custom constraints can be passed to the Vivado® tools using the [advanced] directives of the v++ configuration file specified by the --config option. Refer to [Managing FPGA Synthesis and Implementation Results in the Vivado Tool](#) more information.

Timing Closure Considerations

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

- 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 with the v++ --config option, as described in [Mapping Kernel Ports to Global Memory](#), while verifying that any floorplan constraint ensures the following:

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

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

For quickly reviewing all aspects listed above, use the fail-fast reports generated throughout the Vitis application acceleration development flow using the -R option as described below (refer to [Controlling Report Generation](#) for more information):

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

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

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

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

Additional Resources and Legal Notices

Xilinx Resources

For support resources such as Answers, Documentation, Downloads, and Forums, see [Xilinx Support](#).

Documentation Navigator and Design Hubs

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

- From the Vivado[®] IDE, select **Help** → **Documentation and Tutorials**.
- On Windows, select **Start** → **All Programs** → **Xilinx Design Tools** → **DocNav**.
- At the Linux command prompt, enter `docnav`.

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

- In DocNav, click the **Design Hubs View** tab.
- On the Xilinx website, see the [Design Hubs](#) page.

Note: For more information on DocNav, see the [Documentation Navigator](#) page on the Xilinx website.

Please Read: Important Legal Notices

The information disclosed to you hereunder (the "Materials") is provided solely for the selection and use of Xilinx products. To the maximum extent permitted by applicable law: (1) Materials are made available "AS IS" and with all faults, Xilinx hereby DISCLAIMS ALL WARRANTIES AND CONDITIONS, EXPRESS, IMPLIED, OR STATUTORY, INCLUDING BUT NOT LIMITED TO WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, OR FITNESS FOR ANY PARTICULAR PURPOSE; and (2) Xilinx shall not be liable (whether in contract or tort, including negligence, or under any other theory of liability) for any loss or damage of any kind or nature related to, arising under, or in connection with, the Materials (including your use of the Materials), including for any direct, indirect, special, incidental, or consequential loss or damage (including loss of data, profits, goodwill, or any type of loss or damage suffered as a result of any action brought by a third party) even if such damage or loss was reasonably foreseeable or Xilinx had been advised of the possibility of the same. Xilinx assumes no obligation to correct any errors contained in the Materials or to notify you of updates to the Materials or to product specifications. You may not reproduce, modify, distribute, or publicly display the Materials without prior written consent. Certain products are subject to the terms and conditions of Xilinx's limited warranty, please refer to Xilinx's Terms of Sale which can be viewed at <https://www.xilinx.com/legal.htm#tos>; IP cores may be subject to warranty and support terms contained in a license issued to you by Xilinx. Xilinx products are not designed or intended to be fail-safe or for use in any application requiring fail-safe performance; you assume sole risk and liability for use of Xilinx products in such critical applications, please refer to Xilinx's Terms of Sale which can be viewed at <https://www.xilinx.com/legal.htm#tos>.

AUTOMOTIVE APPLICATIONS DISCLAIMER

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

Copyright

© Copyright 2019 Xilinx, Inc. Xilinx, the Xilinx logo, Alveo, Artix, Kintex, Spartan, Versal, Virtex, Vivado, Zynq, and other designated brands included herein are trademarks of Xilinx in the United States and other countries. All other trademarks are the property of their respective owners.