See all versions of this document

Data Center Acceleration Using Vitis User Guide

UG1700 (v2025.2) November 20, 2025

AMD

Table of Contents

Chapter 1

Getting Started with Vitis

The goal of this guide is to introduce key concepts and provide a pathway to begin accelerating applications using FPGA-based AMD Alveo™ Accelerator Cards, AMD Vitis™ compiler, and unified integrated design environment (IDE).

Navigating Content by Design Process

AMD Adaptive Computing documentation is organized around a set of standard design processes to help you find relevant content for your current development task. You can access the AMD Versal™ adaptive SoC design processes on the Design Hubs page. You can also use the Design Flow Assistant to better understand the design flows and find content that is specific to your intended design needs. This document covers the following design processes:

- **System and Solution Planning:** Identifying the components, performance, I/O, and data transfer requirements at a system level. Includes application mapping for the solution to PS, PL, and AI Engine.
- **Hardware, IP, and Platform Development:** Creating the PL IP blocks for the hardware platform, creating PL kernels, functional simulation, and evaluating the AMD Vivado™ timing, resource use, and power closure. Also involves developing the hardware platform for system integration.
- **Software Development for Acceleration:** Create an algorithm accelerator kernel with HLS and/or AI Engine. Includes platform design, organization, and management.

Vitis Software Platform Installation

Installing the Vitis Software Platform

Ensure your system meets all requirements described in Installation Requirements in the *Vitis Software Platform Release Notes* (UG1742).

TIP: *To reduce installation time, disable anti-virus software and close all open programs that are not needed.*

1. Go to the AMD Adaptive Computing Downloads Website.
2. Download the installer for your operating system.
3. Run the installer, `xsetup` on Linux, or `xsetup.exe` on Windows, which opens the Welcome screen. Extract the installer package.
4. Click **Next**.
5. If installing with the web installer:
   a. At the Select Install Type screen, enter your AMD user account credentials, and select **Download and Install Now**.
   b. Click **Next** to open the Accept License Agreements screen of the installer.
   c. Accept the terms and conditions by clicking each **I Agree** check box.
   d. Click **Next** to open the next screen (only required on the web installer).
6. On the Select Product to Install screen:
   a. To install the full Vitis Software Platform for embedded software and application acceleration development, choose **Vitis** and click **Next**. This full Vitis installation includes Vivado, v++ and AI Engine toolchains.
   b. If you only need to perform software development for embedded processors, and require a small installation footprint, choose **Vitis Embedded Development** and click **Next**. If you downloaded the Vitis Embedded installer, this is the only option available on the installer.
7. Customize your installation by selecting design tools and devices (this step is only for the full Vitis installation).

   The default Design Tools selections are for standard Vitis Unified Software Platform installations, and include Vitis, Vivado, Vitis HLS and Vitis Model Composer. You do not need to separately install Vivado tools.

   Although not required, you can enable **Vitis IP Cache** to install cache files (for example designs found in the release). The Vitis tool installs the files at `<install_dir>/<release>/Vitis/data/cache/xilinx`.

   You can enable the devices required for your development.

   IMPORTANT! *For the Vitis acceleration flow, the following device choice is required for installation:* ***Devices → Install devices for Alveo and Edge acceleration platforms***
8. Click **Next** to open the Accept License Agreements screen of the installer and accept the terms as appropriate.
9. Click **Next** to open the Select Destination Directory screen of the installer.
10. Specify the installation directory and review the location summary. Ensure there is enough disk space and click **Next** to open the Installation Summary screen.
11. Click **Install** to begin the installation of the software.

After a successful installation of the full Vitis unified software, a confirmation message appears, with a prompt to run the `installLibs.sh` script.

1. Locate the script at `<install_dir>/<release>/Vitis/scripts/installLibs.sh`, where `<install_dir>` is the location of your installation, and `<release>` is the installation version.

   ***Note*:** Windows do not require this script.

2. Run the script with `sudo` privileges as follows:

   ```
   sudo installLibs.sh
   ```

The command installs a number of necessary packages for the Vitis tool based on the OS of your system.

> **IMPORTANT!** *Pay attention to any messages returned by the script. Be sure to install any missing packages manually. For example, if your installation of Linux does not include the zip command-line utility, you need to manually install it. This utility is required by some Vitis tools, and the `installLibs.sh` script will not install it for you.*

Installing Xilinx Runtime and Platforms

Xilinx Runtime (XRT)

Xilinx Runtime (XRT) is implemented as a combination of user-space and kernel driver components. XRT provides a software interface to AMD programmable logic devices.

For XRT product details, refer to AMD Vitis Runtime Library.

The XRT library is available for x86 and Arm® Linux OS. It is needed on both application development and deployment environments.

x86 Platforms

For x86 platforms (e.g. AMD Alveo™), XRT installation uses standard Linux RPM and Linux DEB distribution files or built as custom package. Root access is required for all software and firmware installations.

For a guide on how to download and install the XRT, refer to XRT Software Stack for PCIE Accelerator Cards.

Custom Created Platforms

For custom created platforms, the required components are assembled and compiled using tools like PetaLinux or Yocto. Adding XRT to the target Linux image is done via PetaLinux config or Yocto recipes (see Build XRT from Yocto Recipes).

***Note*:** Creating custom Linux platforms require installing PetaLinux. For details on how to use PetaLinux and determine output locations, refer to the *PetaLinux Tools Documentation: Reference Guide* (UG1144).

For examples on how to create a fully customizable design, refer to *Versal Custom Thin Platform Extensible System* in the *Vitis Tutorials: AI Engine Development*.

For examples on how to customize a common image using PetaLinux Tools, refer to *PetaLinux Building and System Customization* in the *Vitis Tutorials: Vitis Platform Creation*.

Setting Up the Vitis Environment

The AMD Vitis™ unified software platform includes two elements that must be installed and configured to work together properly: the Vitis core development kit and XRT. The requirements of installation and configuration are described in Vitis Software Platform Installation in the *Vitis Software Platform Release Notes* (UG1742).

If you have the elements of the Vitis software platform installed, you need to setup the environment to run in a specific command shell by running the following 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
```

TIP: *The `PLATFORM_REPO_PATHS` environment variable points to directories containing platform files (`.xpfm`).*

TIP: *`.csh` scripts are also provided.*

This sets up the tools for the Vitis application development flow and the AI Engine tools for development on Versal adaptive SoC AI Engine devices.

To use any platforms you have downloaded as described in Installing Xilinx Runtime and Platforms in the *Embedded Design Development Using Vitis* (UG1701), set the following environment variable to point to the location of the platforms:

```
export PLATFORM_REPO_PATHS=<path to platforms>
```

This identifies the location of platform files for the tools, and makes them accessible to your design projects.

Chapter 2

Introduction to Vitis

The AMD Vitis™ unified software platform is a development environment for heterogeneous applications supporting AMD devices such as AMD Alveo™ Data Center Accelerator and adaptive SoC devices. In the Vitis environment, heterogeneous systems include software applications running on x86 host processors and compute kernels running in programmable-logic (PL) regions that provide the foundation for building and running the heterogeneous systems. The Vitis unified software platform consists of the following elements:

- The software development tool stack, such as compilers and cross-compilers to build your software application.
- Debuggers to help you locate and fix any problems in your system design.
- Program analyzers to let you profile and analyze the performance of your application.
- Xilinx Runtime (XRT) provides an API and drivers for your software program to connect with the target platform, and handles transactions and data transfers between the software application and the hardware design.
- Vitis accelerated libraries provide performance-optimized hardware functions with minimal code changes, and without the need to re-implement your algorithms to harness the benefits of AMD adaptive computing. Vitis accelerated libraries are available for common functions of math, statistics, linear algebra and DSP, in addition to 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 Vitis Libraries.

The Vitis unified software platform combines all aspects of AMD hardware and software development into one unified environment using standard C/C++ for both software and hardware components. The Vitis tools provide compilation, linking, profiling and debug capabilities for heterogeneous systems in a number of different design flows including Data Center application acceleration, RTL kernel design, and traditional embedded hardware and software design.

Document Focus

This document focuses on the Data Center use cases, and the following flows are described:

- Introduction to the Vitis Unified IDE
- Introduction to Data Center Acceleration for Software Programmers
- Introduction to Data Center Acceleration for RTL Designers

- Tutorials and Examples

Introduction to Data Center Acceleration for Software Programmers

This chapter is intended for C/C++ software developers who want to accelerate their data center applications using AMD FPGA-based AMD Alveo™ Accelerator Cards. The goal of this guide is to introduce key concepts and provide a pathway for software developers to begin accelerating applications using the AMD Vitis™ compiler and integrated development environment (IDE).

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 an AMD device enable a high degree of parallelism in application execution.

The following are key concepts for creating accelerated applications on FPGAs resulting in greater acceleration performance vs CPU:

- Applications written for CPU and FPGA are quite different, and rewriting the functions to be accelerated on an FPGA is required. Functions are executed sequentially on the CPU and must infer parallelism on FPGA for greater performance.
- For application acceleration, the software program is split into a host application that runs on the CPU and compute functions, or kernels that run on the Alveo data center accelerator card. The XRT runtime library provides an API enabling the host application to interact with the kernels on the accelerator cards.
- 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 performance achieved by the hardware acceleration kernels must outweigh the added latency of the data transfers.
- The software developer must profile the original application and identify functions with the potential to be accelerated. Once the target functions are identified, a performance budget for each kernel must be determined to meet the overall application performance goal.
- The memory hierarchy plays a key role in overall application performance. The memory accessed by kernels must be grouped as memory reads and writes in separate functions using a load-compute-store architecture. The kernels must access contiguous memory if possible and the number of accesses must be optimized by removing redundant accesses or by creating a local cache.

You are encouraged to review this material and the extended material referred to in the following topics. After reviewing this document that lists key concepts and examples along with extended reference material, you must have a practical understanding for developing or modifying existing functions to be targeted for acceleration with proper architecture that meets your performance needs.

Terminology

The following introduces some of the tools and terms used in this document:

- **Vitis core development kit:** 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 a CPU with XRT API calls to manage runtime interactions with the accelerator. The hardware component, or kernel, can be developed using C/C++ or using Verilog or VHDL.
- **Alveo data center accelerator cards:** Are PCI Express® Gen3 x16 compliant cards designed to accelerate compute intensive applications such as machine learning, data analytics, and video processing.
- **Platform:** A predefined configuration of the Alveo accelerator card with features implemented for specific applications. A platform has multiple partitions. The base logic partition (BLP) is a static region that contains fixed logic for essential functions (such as PCIe and DMA). A user logic partition (ULP), which is a dynamic region where the C++ or RTL kernel logic, is programmed for execution.
- **XRT:** The Xilinx Runtime library that provides an API and drivers for your host program to connect with the target FPGA platform and handles transactions between your host program and accelerated kernels.
- **Host and Global Memory:** The distinction between memory on the host machine used by the CPU, and memory on the Alveo data center accelerator card used by the accelerated functions.
- **Vitis HLS:** A high-level synthesis tool that translates C/C++ functions into device logic using programmable logic (PL) elements and RAM/DSP blocks. Vitis HLS synthesizes the C/C++ code into an RTL design and packages it as a compiled object (`.xo`) file that can be imported into the Vitis environment. There can be multiple functions targeted on the FPGA, each being a separate kernel. Vitis HLS will synthesize these kernels one by one and generate separate `.xo` files.
- **Register transfer level (RTL):** An abstraction level used for modeling digital circuits. Often the term RTL is used interchangeably for Verilog or VHDL which are both hardware description languages.
- **PL Kernel (.xo) file:** Is the term to designate the custom logic implementation of your accelerator function. Each kernel is packaged as an `.xo` file and contains the IP for the function and associated metadata used by the Vitis tool.

- **RTL Kernel:** Is an RTL design that uses standard AXI4 interfaces to enable the Vitis compiler to link it into the target platform to quickly build the system design. The RTL kernel is packaged as an `.xo` file.
- **Vivado Design Suite:** An RTL language synthesis and implementation tool that takes the RTL design generated by Vitis HLS or an RTL designer and generates the bitstream that can be loaded and executed on the FPGA.
- **Bitstream:** Is the configuration data that is used to program the FPGA so that its functionality can be changed. The kernel design will result in a bitstream used to program the dynamic region of the FPGA on the Alveo accelerator card.
- **Device Binary (.xclbin) file:** Contains the bitstream and other metadata needed to be used to program the FPGA. This is used by XRT APIs to actually program the FPGA. In the Vitis flow, the device binary files have the extension `.xclbin`.
- **Vitis analyzer:** A utility that allows you to view and analyze the reports generated while building and running the application through Vitis, Vitis HLS, and AMD Vivado™.

Working with Alveo Accelerator Cards

What is an FPGA

An FPGA (field-programmable gate array) is an integrated circuit that uses an array of interconnected programmable logic elements to implement any type of digital function as a physical circuit. Because the elements and the routing resources that connect them are configured after power-up, the FPGA can be repeatedly programmed to implement any set of functions required. By creating multiple copies of these functions, FPGAs are particularly well suited at implementing functions in parallel, making them extraordinarily good at serving as hardware accelerators for applications that contain high levels of parallelism. FPGAs come in different sizes with different quantities of programmable logic resources. Larger devices contain more resources, allowing designers to implement more parallel circuits, leading to higher levels of acceleration. The variety of devices provides designers with multiple cost/performance trade-offs.

Unlike GPUs, which contain processing cores that must fetch and execute instructions, FPGAs have a flexible architecture that maps code to physical logic circuitry. Like GPUs, however, it is necessary for you to understand some of the basics of how this is done to architect your code for best results.

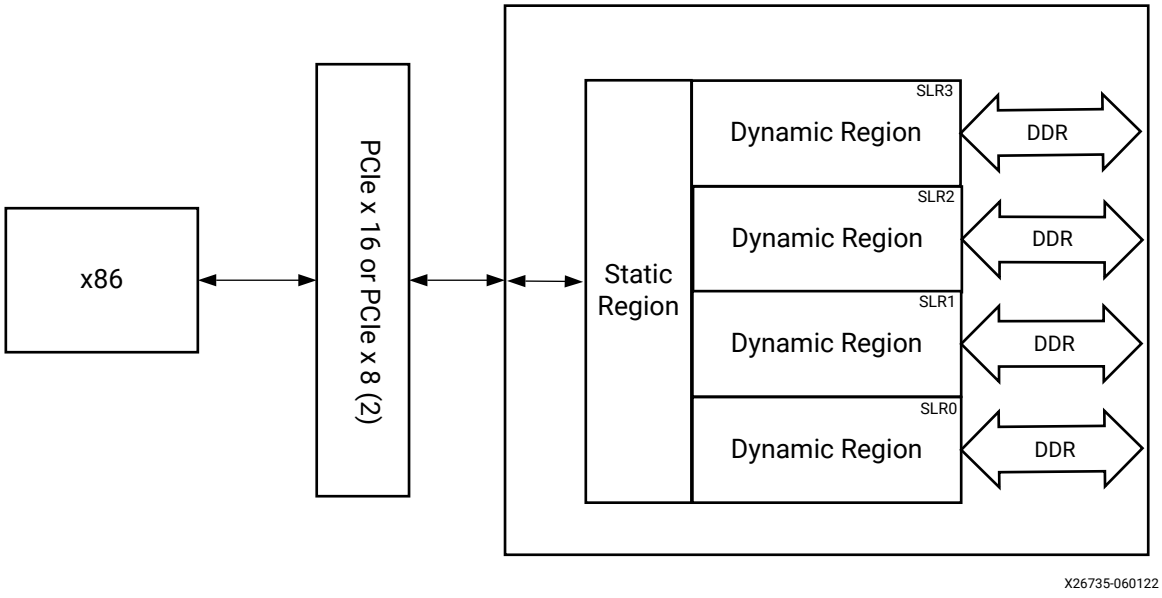
Alveo Block Diagram and Data Movement between Host and FPGA

Using FPGAs at its core, AMD has developed the Alveo family of PCIe Data Center accelerator cards. Each Alveo card combines three essential things: a powerful FPGA for acceleration, high-bandwidth device memory banks, and connectivity to a host server via a high-bandwidth PCIe Gen3x16 link. A number of different cards are available to provide designers with a choice of features and quantity of programmable resources. Below is the block diagram for the Alveo U250.

Although FPGAs are essentially blank devices that get configured at power-up, all Alveo cards are shipped with target platforms that provide the firmware to configure the accelerator card for specific uses. The platform must be installed with Xilinx Runtime (XRT); flashed into the device during installation, or when changing the configuration of the accelerator card.

On the AMD device, the platform consists of two physical FPGA partitions: *Shell* and *User*. Shell partition is a static region and provides basic infrastructure for the platform like PCIe connectivity, board management, sensors, clocking, and reset. User partition is a dynamic region that contains user compiled binary called `.xclbin` which is loaded by XRT during execution. RTL kernels are the custom logic created by the developer and programmed into the dynamic region. In this document, kernels refer to the functions that the designer is implementing into the dynamic region of the Alveo accelerator card.

Figure 1: **Alveo Block Diagram**



The PCIe interface is used for communication between the host and accelerator card, and to transfer data from the host into the Alveo card's device memory. This device memory serves as a Global memory, accessible by both host and hardware accelerators. The device memory included on the Alveo platform are PLRAM (small size but fast access with the lowest latency), HBM (moderate size and access speed with some latency), and DDR (large size but slow access with high latency). Depending upon the Alveo card, you might have DDR or HBM, or even both.

The block diagram shown above is of U250 and has 4 banks of DDR, each with 16GB of memory. The FPGA on the Alveo card is further subdivided into multiple super logic regions (SLRs), which aid in the architecture of very high-performance designs. But this is a slightly more advanced topic that will remain largely unnoticed as you take your first steps into Alveo development.

To further improve performance, and minimize access to DDR memory, FPGAs have large quantities of small, internal RAM blocks. These are completely configurable by the compiler to ensure that buffering can be created between tasks to enable pipeline-style computation. This effectively eliminates the need for caches and is one of the key strengths of FPGAs.

There are many more details you could learn about the FPGA architecture and Alveo cards, but this is sufficient for introductory purposes. From the perspective of designing an FPGA-based acceleration architecture, the important points to remember are:

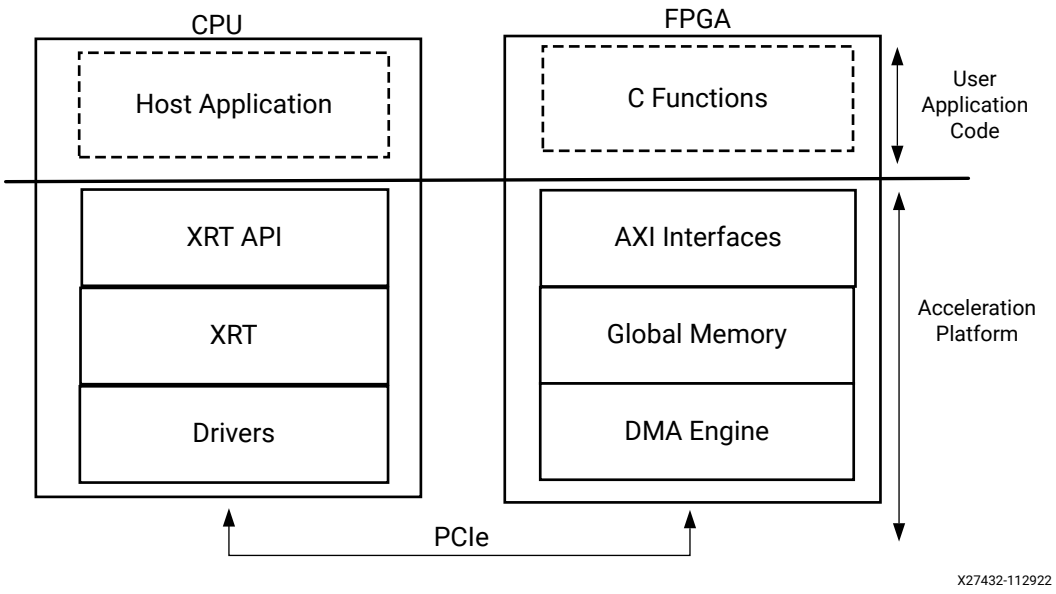
- Moving data across PCIe is expensive, even at Gen3x16, latency is high. For larger data transfers, bandwidth can easily become a system bottleneck.
- Bandwidth and latency between the DDR4 and the FPGA are significantly better than over PCIe, but touching external memory is still expensive in terms of overall system performance.

A Sample Application

This section provides a snapshot of the evolution of a program written for CPU into an application written for FPGA-based acceleration. This section is primarily intended to showcase key ideas for building your application without going into details. You might come across several new terms here, but you can refer to Terminology for some definitions.

The figure below illustrates the execution flow of the Vitis application acceleration environment. The application program is split between an application running on CPU (called the host program) and hardware-accelerated kernels running on FPGA with a communication channel between them. The host program, written in C/C++ and using the XRT API, is compiled into an executable that runs on an x86 based host processor while hardware-accelerated kernels are compiled into an executable device binary (`.xclbin`) that runs within the programmable logic (PL) region of an AMD device on the Alveo accelerator card.

Figure 2: **CPU/FPGA Interaction**



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. The execution model of a Vitis application 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 accelerator card.
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 following is a simple program written in C++ for execution on the CPU. This program includes the `compute()` function to be accelerated as a kernel on an Alveo accelerator card.

```
#include <vector>
#include <iostream>
#include <ap_int.h>
#include "hls_vector.h"

#define  totalNumWords 512
unsigned char data_t;

int main(int, char**) {
```

```
    // initialize input vector arrays on CPU
    for (int i = 0; i < totalNumWords; i++) {
      in[i] = i;
    }
    compute(data_t in[totalNumWords], data_t Out[totalNumWords]);
    check_results();
}

void compute (data_t in[totalNumWords ], data_t Out[totalNumWords ]) {
  data_t tmp1[totalNumWords], tmp2[totalNumWords];
  A: for (int i = 0; i < totalNumWords ; ++i) {
    tmp1[i] = in[i] * 3;
    tmp2[i] = in[i] * 3;
  }
  B: for (int i = 0; i < totalNumWords ; ++i) {
    tmp1[i] = tmp1[i] + 25;
  }
  C: for (int i = 0; i < totalNumWords ; ++i) {
    tmp2[i] = tmp2[i] * 2;
 }
  D: for (int i = 0; i <  totalNumWords ; ++i) {
     out[i] = tmp1[i] + tmp2[i] * 2;
   }
}
```

The program looks very similar to any other C++ program where there the main function calls a compute function, setting up the data to be sent to compute function, and checking the results with golden results after compute function completes. The execution of this program is sequential on the CPU. This program can also run sequentially on an FPGA, producing correct results without any performance gain compared to the CPU. For the application to execute with higher performance on an FPGA, the program needs to be re-architected to enable parallelism at various levels. Examples of parallelism can include:

- The compute function can start before all the data is transferred from the host to the compute function
- Multiple compute functions can run in an overlapping fashion, for example a "for" loop can start the next iteration before the previous iteration has completed
- The operations within a "for" loop can run concurrently on multiple words and doesn't need to be executed on a per-word basis

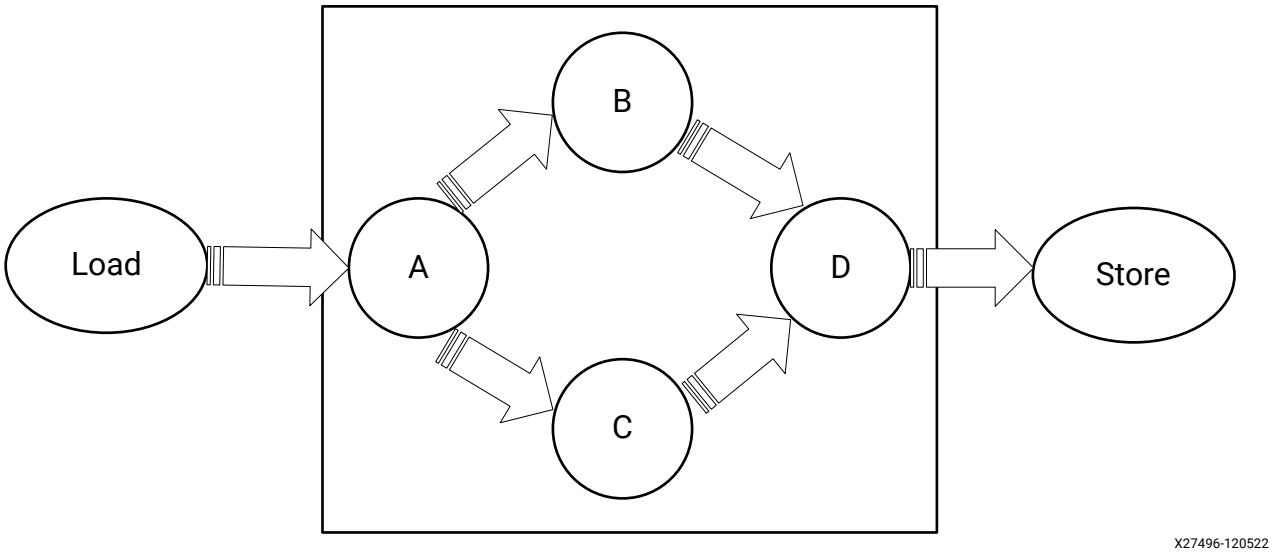
You will need to re-architect the compute function that resides on the FPGA as an accelerated kernel, and the host application that runs on the CPU and communicates with the accelerated kernels.

Re-Architecting Kernel Code

From the prior example it is the `compute()` function that needs to be re-architected for FPGA-based acceleration.

In the `compute()` function Loop A multiplies the input with 3 and creates two separate paths, B and C. Loop B and C performs operations and feed the data to D. This is a simple representation of a realistic case where you have several tasks to be performed one after another and these tasks are connected to each other as a network like the one shown below.

Figure 3: **Kernel Architecture**



The key takeaways for re-architecting the kernel code are:

- Task-level parallelism is implemented at the function level. To implement task-level parallelism loops are pushed into separate functions. The original `compute()` function is split into multiple sub-functions. As a rule of thumb, sequential functions can be made to execute concurrently, but sequential loops will execute sequentially.
- These tasks (or sub-functions) are communicating with each other using `hls::stream` which acts as a FIFO channel. The `hls::stream` class is a C++ template class for modeling streams behavior between functions.
- Instruction-level parallelism is implemented by reading 16 32-bit words from memory (or 512-bits of data). Computations can be performed on all these words in parallel. The `hls::vector` class is a C++ template class for executing vector operations on multiple samples concurrently.
- The `compute()` function needs to be re-architected into load-compute-store sub-functions, as shown in the example below. The load and store functions encapsulate the data accesses and isolate the computations performed by the various compute functions.

- Additionally, there are compiler directives starting with `#pragma` that can transform the sequential code into parallel execution.

```
#include "diamond.h"
#define NUM_WORDS 16
extern "C" {

void diamond(vecOf16Words* vecIn, vecOf16Words* vecOut, int size)
{
  hls::stream<vecOf16Words> c0, c1, c2, c3, c4, c5;
  assert(size % 16 == 0);

  #pragma HLS dataflow
  load(vecIn, c0, size);
  compute_A(c0, c1, c2, size);
  compute_B(c1, c3, size);
  compute_C(c2, c4, size);
  compute_D(c3, c4,c5, size);
  store(c5, vecOut, size);
}
}

void load(vecOf16Words *in, hls::stream<vecOf16Words >& out, int size)
{
Loop0:
  for (int i = 0; i < size; i++)
  {
    #pragma HLS performance target_ti=32
    #pragma HLS LOOP_TRIPCOUNT max=32
    out.write(in[i]);
  }
}

void compute_A(hls::stream<vecOf16Words >& in, hls::stream<vecOf16Words >&
out1, hls::stream<vecOf16Words >& out2, int size)
{
Loop0:
  for (int i = 0; i < size; i++)
  {
    #pragma HLS performance target_ti=32
    #pragma HLS LOOP_TRIPCOUNT max=32
    vecOf16Words t = in.read();
    out1.write(t * 3);
    out2.write(t * 3);
  }
}
void compute_B(hls::stream<vecOf16Words >& in, hls::stream<vecOf16Words >&
out, int size)
{
Loop0:
  for (int i = 0; i < size; i++)
  {
    #pragma HLS performance target_ti=32
    #pragma HLS LOOP_TRIPCOUNT max=32
    out.write(in.read() + 25);
  }
}


void compute_C(hls::stream<vecOf16Words >& in, hls::stream<vecOf16Words >&
out, int size)
{
```

```
Loop0:
  for (data_t i = 0; i < size; i++)
  {
    #pragma HLS performance target_ti=32
    #pragma HLS LOOP_TRIPCOUNT max=32
    out.write(in.read() * 2);
  }
}
void compute_D(hls::stream<vecOf16Words >& in1, hls::stream<vecOf16Words >&
in2, hls::stream<vecOf16Words >& out, int size)
{
Loop0:
  for (data_t i = 0; i < size; i++)
  {
    #pragma HLS performance target_ti=32
    #pragma HLS LOOP_TRIPCOUNT max=32
    out.write(in1.read() + in2.read());
  }
}

void store(hls::stream<vecOf16Words >& in, vecOf16Words *out, int size)
{
Loop0:
  for (int i = 0; i < size; i++)
  {
    #pragma HLS performance target_ti=32
    #pragma HLS LOOP_TRIPCOUNT max=32
    out[i] = in.read();
  }
}
```

Re-Architecting the Host Application

The main function in the original program is responsible for setting up the data, calling the compute function, checking the results, etc. In the case of an accelerated application, the host code is responsible for initializing the data to be sent/received over the PCIe® bus to the device memory. It also sets the kernel function arguments similar to how the main function calls compute functions. The API calls, managed by XRT, are used to process transactions between the host program and the hardware accelerators.

In general, the structure of the host application can be divided into the following steps:

1. Loading the `.xclbin` generated into the program.
2. Allocate buffers in the global memory
3. Create the input test data and map the buffers to the host memory
4. Setting up the kernel and kernel arguments.
5. Transferring buffers between the host and kernels
6. Execute the kernel.
7. Receive the output results back to the host into output buffers

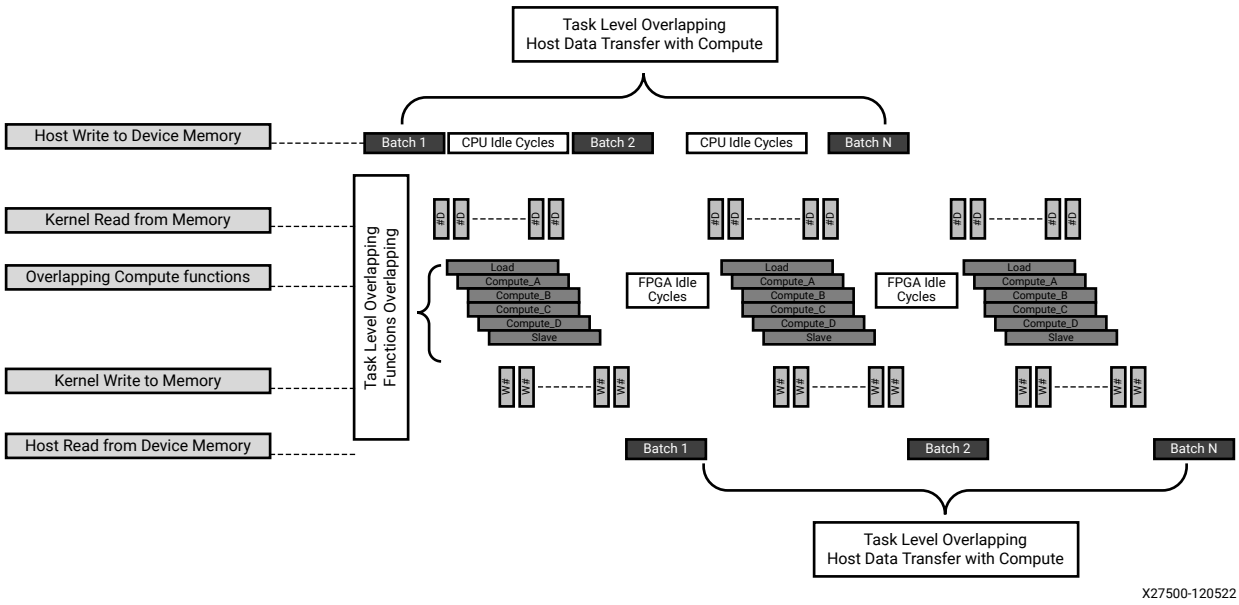
The host application re-written for the `compute()` function described above, making use of the XRT native API to run on the Alveo accelerator card is shown below:

```
// XRT includes
#include "experimental/xrt_bo.h"
#include "experimental/xrt_device.h"
#include "experimental/xrt_kernel.h"

#include "types.h"

int main(int argc, char** argv) {
    unsigned int device_index = 0;
    auto uuid = device.load_xclbin("diamond.hw.xclbin");

    size_t vector_size_bytes = sizeof(int) * totalNumWords;
    auto krnl = xrt::kernel(device, uuid, "diamond");

    std::cout << "Allocate Buffer in Global Memory\n";
    auto bufIn = xrt::bo(device, vector_size_bytes, krnl.group_id(0));
    auto bufOut = xrt::bo(device, vector_size_bytes, krnl.group_id(1));

    // Map the contents of the buffer object into host memory
    auto bufIn_map = bufIn.map<int*>();
    auto bufOut_map = bufOut.map<int*>();
    std::fill(bufIn_map, bufIn_map + totalNumWords, 0);
    std::fill(bufOut_map, bufOut_map + totalNumWords, 0);

   // Create the input data
    for (int i = 0; i < totalNumWords; i++)
          bufIn_map[i] = (uint32_t)i;

    // Create the output golden data
    int bufReference[totalNumWords];
    for (int i = 0; i < totalNumWords; ++i) {
        bufReference[i] = ((i*3)+25)+((i*3)*2);
    }

    // Synchronize buffer content with device side
    bufIn.sync(XCL_BO_SYNC_BO_TO_DEVICE);

    std::cout << "Execution of the kernel\n";
    auto run = krnl(bufIn,bufOut,totalNumWords/16);
    run.wait();

    // Get the output;
    std::cout << "Get the output data from the device" << std::endl;
    bufOut.sync(XCL_BO_SYNC_BO_FROM_DEVICE);

    for (int i = 0; i < totalNumWords; i++)
    {
       std::cout << "Referece  "  << bufReference[i] << std::endl;
       std::cout << "Out  "  << bufOut_map[i] << std::endl;
    }

    // Validate our results
    if (std::memcmp(bufOut_map, bufReference, totalNumWords))
        throw std::runtime_error("Value read back does not match
reference");
    std::cout << "TEST PASSED\n";
```

Application Execution Timeline

When run on the Alveo accelerator card, the application timeline looks like the following.

Figure 4: **Application Timeline**



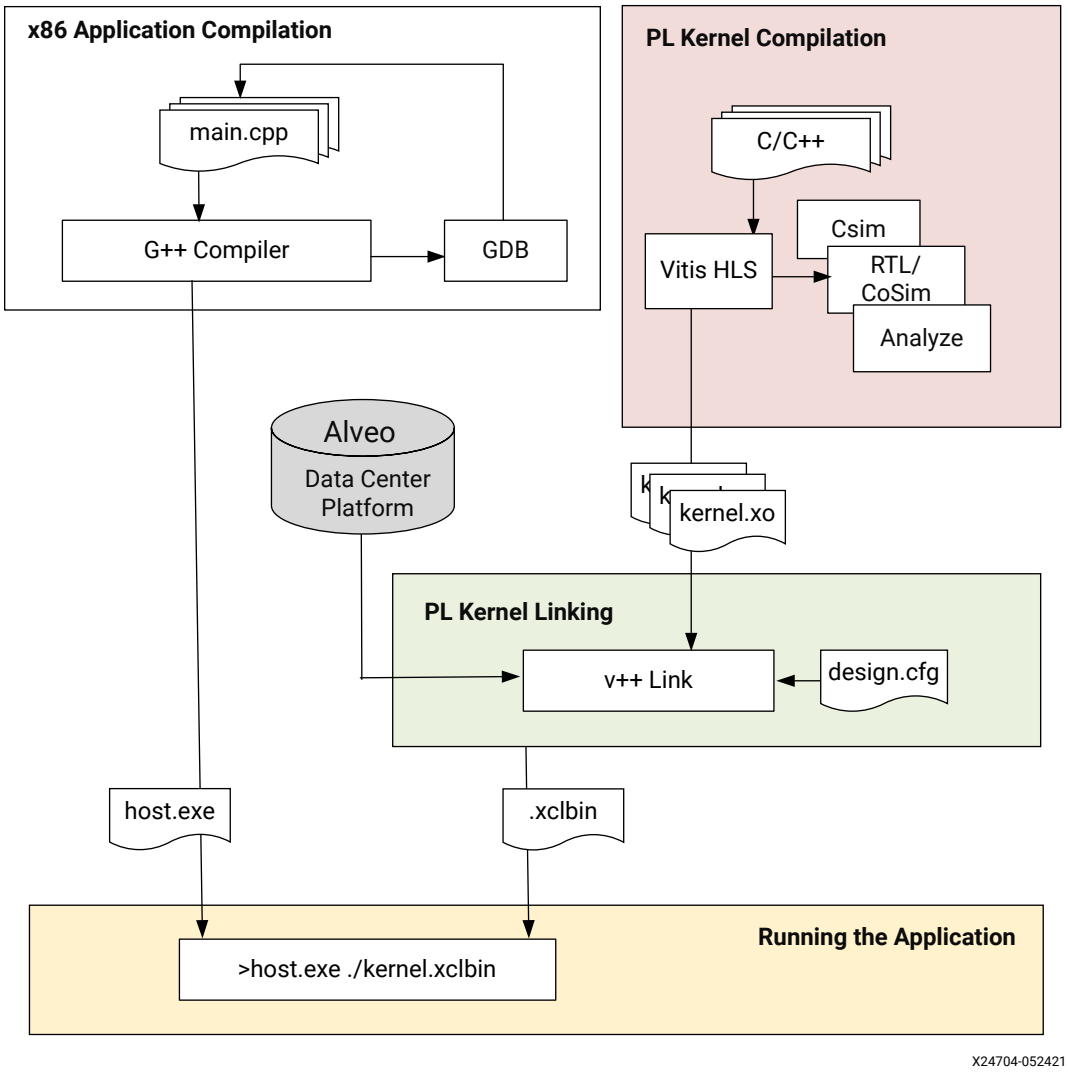
The execution of the application on an FPGA is quite different than on a CPU due to several types of parallelism that can be observed from the preceding figure. The kernel code was written to leverage task-level parallelism by creating sub-functions for each loop. The result is that `compute_A`, `compute_B`, `compute_C`, and `compute_D` are running in an overlapping fashion. In fact, `compute_A`, `compute_B`, `compute_C`, and `compute_D` are sub-function calls within the compute function. A similar execution overlap can be accomplished for multiple kernels.

While the hardware device and its kernels are designed to offer potential parallelism, the software application must be engineered to take advantage of this potential parallelism. Task-level parallelism is further enabled by overlapping host-to-device data transfers and overlapping the compute function execution.

Vitis Application Development Flow

In this section, you will learn about the Vitis application development flow first and then review a methodology for re-architecting a CPU application for FPGA-based acceleration; re-coding each kernel to meet the overall performance objective. An image of the Vitis application development flow is shown below.

Figure 5: **Vitis Development Flow**



The development flow includes the following steps:

- **Application Compilation using G++:** The host program written in C/C++, and using XRT native API, is compiled using a g++ compiler to create a host executable file to run on the x86 processor. The host program interacts with kernels in the PL region on the FPGA device.
- **Kernel Compilation using Vitis HLS:** Vitis HLS is a compiler that takes C/C++ source code as an input and synthesizes it into an RTL design that is optimized for AMD FPGA products. Each C++ kernel must be synthesized using Vitis HLS to produce a Xilinx object (`.xo`) file. One or more `.xo` files can be paired for linking using Vitis linker to produce the `.xclbin` file.

  The steps for kernel development in Vitis HLS are as follows:

  1. Write the C/C++ code for the function
  2. Verify the Code using C-simulation

3. Build the kernel using C-synthesis
4. Verify the kernel generated with C++ outputs
5. Review the HLS synthesis reports and co-simulation reports to analyze performance
6. Repeat previous steps until performance goals are met.

- **PL Kernel Linking using Vitis Tools:** Xilinx object (`.xo`) files are linked with the target hardware platform by the Vitis linker to create a device binary file (`.xclbin`) that is loaded for execution on the Alveo accelerator card.

  TIP: *This step will call Vivado place and route to generate the `.xclbin` file.*

  To help define the architecture of the device binary, you can create a configuration file that specifies how many instances of a kernel (or compute unit) must be built in the device binary, and how the kernels are connected to the global memory or to other kernels. This configuration file is passed to the Vitis linker to generate the `.xclbin`.

  There are two different build targets of the Vitis Compiler that defines the nature and contents of the generated `.xclbin` file. One emulation target used for validation and debugging purposes: hardware emulation for RTL co-simulation; and the hardware target for building the final project output to run on the Alveo card. The same host program can be used to run any of the `.xclbin` targets.

  TIP: *Compiling for an emulation target is significantly faster than compiling for actual hardware. The emulation run is performed in a simulation environment, which offers enhanced debug visibility and does not require a physical accelerator card.*

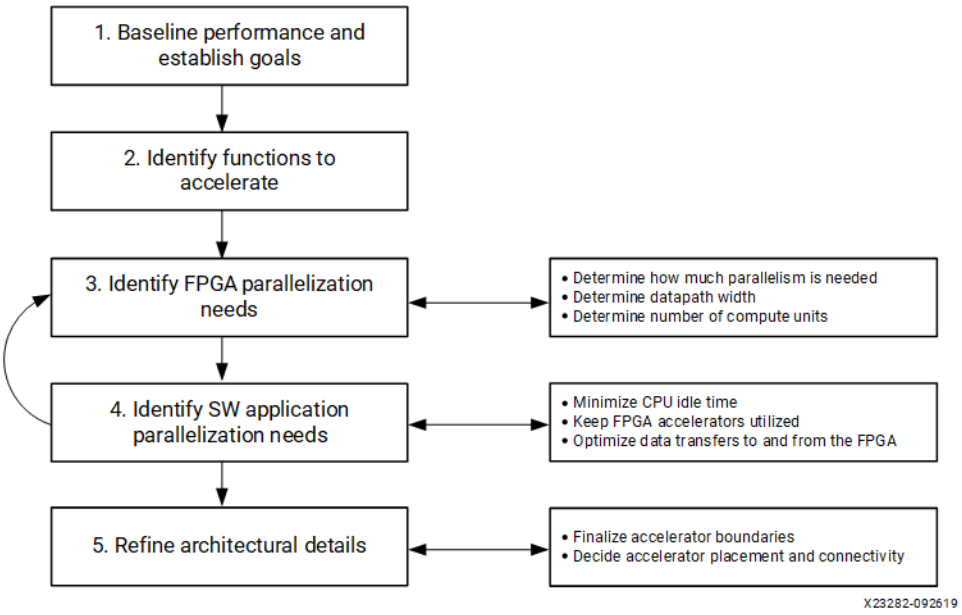
- **Running the Application:** Finally, when you run the application the host program loads the `.xclbin` file generated by the Vitis Compiler. The host application always runs on the CPU and can be run in emulation mode on x86, or run on the actual physical accelerator platform.

Developing Vitis Accelerated Applications

The methodology is comprised of two major phases:

1. Architecting the application and identifying kernels with performance goals defined. The developer makes key decisions about the application architecture by determining which software functions must be mapped to device kernels, how much parallelism is needed, and how it must be delivered.
2. Developing the C/C++ kernels to meet the goals established. The developer implements the kernels. This task primarily involves structuring source code and applying the desired compiler pragma to create the desired kernel architecture and meet the performance target. Review Design Principles in the *Vitis High-Level Synthesis User Guide* (UG1399), which is intended for software developers who want to understand the process of synthesizing accelerated hardware from a software algorithm written in C/C++

Figure 6: **Methodology for Architecting the Application**



The preceding image highlights the key tasks related to architecting the application:

- Profile the C++ application using Valgrind, callgrind, and gprof to create the baseline for analysis. The functions that consume the most execution time are good candidates to be offloaded and accelerated onto FPGAs.
- The maximum achievable throughput is limited by the PCIe bus. PCIe performance is influenced by many different aspects, such as motherboard, drivers, target platform, 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 `xrt-smi dma` test.
- Identify the performance bottlenecks by reviewing the algorithm and analyzing any parallel paths. Accelerating one path might not give the expected acceleration for the overall application. When looking for acceleration candidates, consider the performance of the entire application instead of only individual functions.
- Identify the overall acceleration potential, set the application performance goal.
- After the functions to be accelerated are identified and the overall acceleration goals are established, the next step is to determine what level of parallelization is needed to meet the goals.
- Enable parallelism between host and device data transfer and compute on FPGA so that there is minimal idle time. Keep the device kernels active performing new computations as early and often as possible. Optimize data transfers to and from the device.

For a more complete examination of this topic, refer to Accelerating Data Center Applications with Vitis Software Platform, or refer to the Design Principles for Software Programmers section of the *Vitis High-Level Synthesis User Guide* (UG1399).

Methodology for Developing C/C++ Kernels

The software program can be automatically converted (or synthesized) into hardware, but achieving acceptable quality of results (QoR) requires additional work such as rewriting the software to help the Vitis HLS tool achieve the desired performance goals. To help, you need to understand the best practices for writing good software for execution on the FPGA device. The next few sections will discuss how you can first identify some macro-level architectural optimizations to structure your program and then focus on some fine-grained micro-level architectural optimizations to boost your performance goals. The following key kernel requirements for optimal application performance must have already been identified during the architecture definition phase:

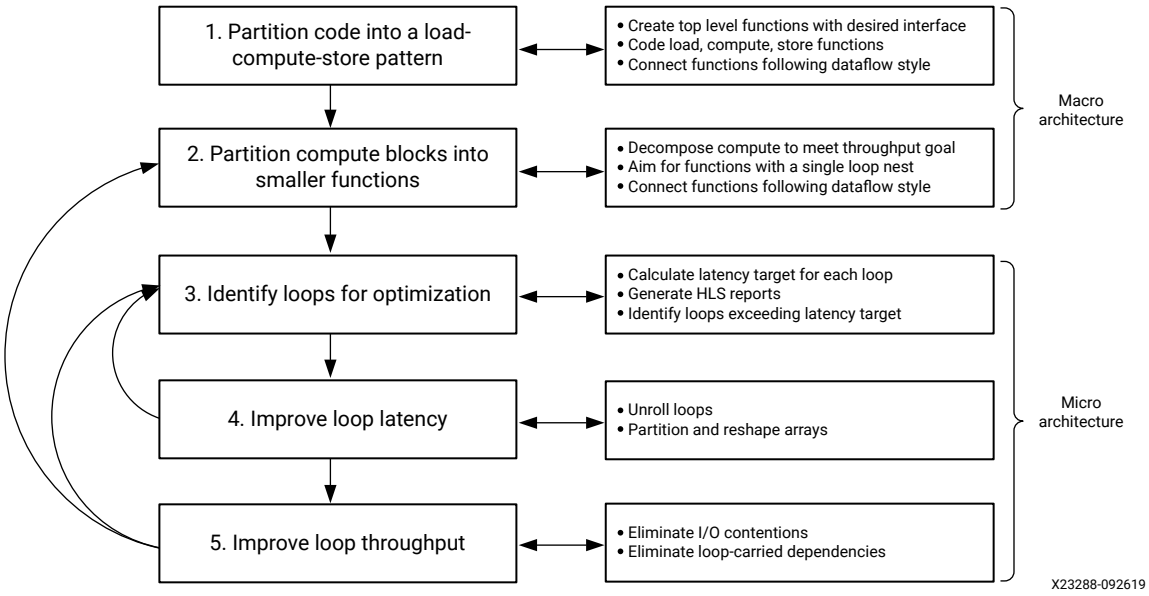
- Throughput goal
- Latency goal
- Datapath width
- The number of accelerated kernels.
- Interface bandwidth

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

Figure 7: **Kernel Development Methodology**



Refer to Methodology for Developing C/C++ Kernels for a detailed view on requirements, considerations, and how to re-architect the code for achieving higher performance.

Best Practices for Kernel Development

You have reviewed some basic understanding of the Alveo accelerator card and its key components, how the data moves between the CPU and Alveo card. You have also been exposed to the recommended guidelines for creating Vitis applications. This section will cover more in-depth topics that are key concepts of coding using Vitis HLS.

Mapping Function Arguments to HW Interfaces

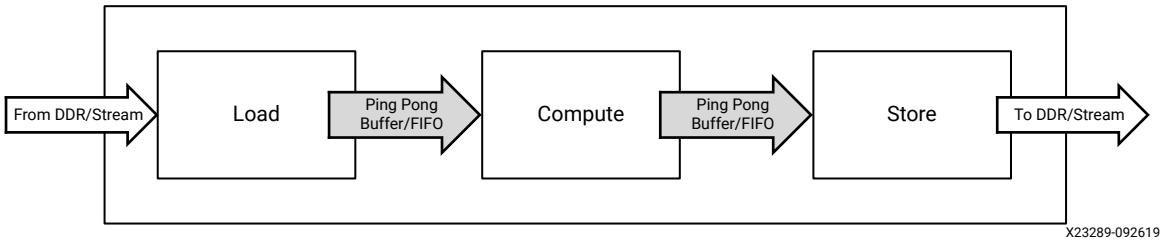
The Vitis HLS tool automatically assigns interface ports for the arguments of your C/C++ kernel function. These function arguments are of either scalar or pointer/array types. The parameters from the host are written directly to the registers of the accelerators. The buffers are kept external in the global memory and the accelerator reads and writes from this global memory.

The scalar type function arguments are used for parameters and the pointer or array type arguments are used for accessing global memory. The Vitis HLS implements these interface ports as AXI Protocol. Refer to Introduction to AXI for more information on this interface protocol.

Load - Compute - Store

The algorithm must be structured as load-compute-store with communications channels in between as shown below.

Figure 8: **Load-Compute-Store Pattern**



- The `load` function is responsible for moving data into the kernel from the device memory. This function does not perform 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 is not 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.

The developer needs to code memory accesses in a way to minimize the overhead of global memory accesses, which means maximizing the use of consecutive accesses so that bursting can be inferred. The burst access hides the memory access latency and improves the memory bandwidth.

Additionally, the maximum data width from the global memory to and from the kernel is 512 bits. To maximize the data transfer rate, it is recommended that you use this full data width. By default in the Vitis kernel flow the Vitis HLS tool automatically re-sizes the kernel interface ports up to 512-bits to improve burst access.

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?

Load-Compute and Compute-Store communicate over the streaming channel. 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. As soon as sufficient data is available for the compute function, the computation can start. Similarly, as soon as the data is available for the Store function, the data can be sent to the DDR over the AXI4 Master interface.

Example - Dataflow using FIFOs

Task Level Parallelism

The developer needs to assess the algorithm and determine how task-level parallelism can be accomplished. This type of parallelism can be enabled in two dimensions.

1. The tasks can execute in an overlapping fashion with each other. In other words, Compute functions can start based on the data availability and does not require the previous function to finish first. With the data flow enabled, the tool will infer this type of parallelism.
2. The task can restart itself within a given time, called the "Transaction Interval." In other words, the next invocation of the same compute function can be restarted before its previous invocation is completely done. The Vitis tool provides the compiler directive for the performance target for any loop. When this directive is added, the compiler will automatically do the necessary transformations or combinations of transformations like partitioning the arrays, unrolling the nested loops, or pipeline the loops to meet the "Target Interval" goal.

For more information on function and loop pipelining, loop unrolling, and array partitioning, see *Vitis High-Level Synthesis User Guide* (UG1399).

Verifying Functional Correctness of the Kernel

When using the Vitis HLS design flow, it is time-consuming to synthesize functionally incorrect C code and then analyze the implementation details to determine why the function does not perform as expected. Therefore, the first step in high-level synthesis is to validate that the C function is correct, before generating RTL code, by performing C-simulation using a well-written test bench. Writing a good test bench can greatly increase your productivity, as C functions execute in orders of magnitude faster than RTL simulations. Using C to develop and validate the algorithm before synthesis is much faster than developing and debugging RTL code. The same C-based test bench can be used to run C/RTL co-simulation to automatically verify the RTL design generated.

For further review of this subject, see *Vitis High-Level Synthesis User Guide* (UG1399), which includes the following material:

- Writing a Test Bench
- Verifying Code with C Simulation
- C/RTL Co-Simulation
- The Vitis HLS Analysis and Optimization Tutorial will work through the Vitis HLS tool GUI to build, analyze, and optimize a hardware kernel.
- For best practices to use when designing interfaces for your application, review the checklist in Best Practices for Designing with M_AXI Interfaces in the *Vitis High-Level Synthesis User Guide* (UG1399).

Best Practices for Host Programming

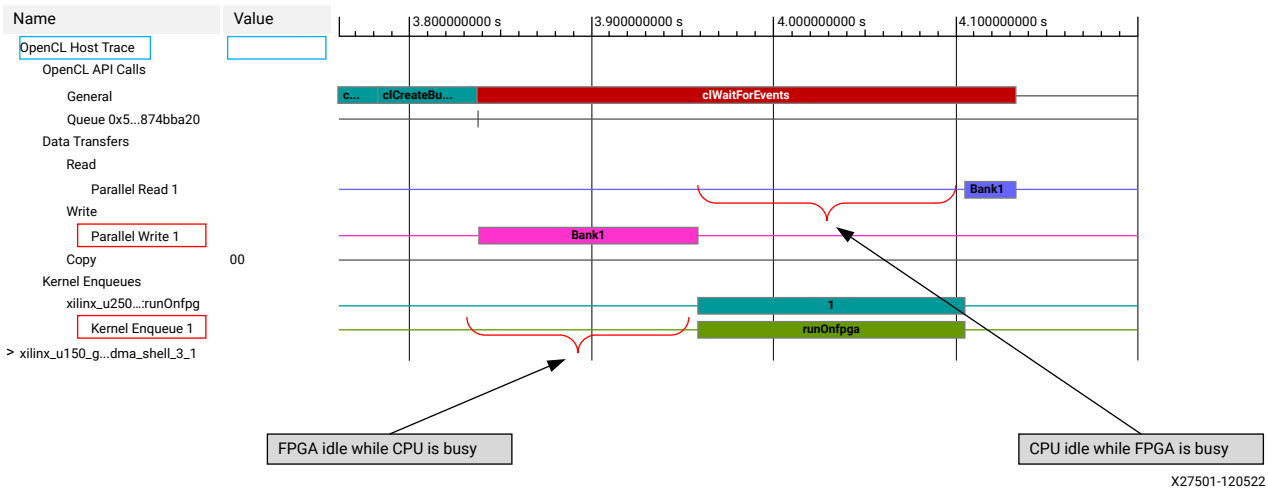
In the Vitis environment, the host application can be written in C++ using the Xilinx Runtime (XRT) native C++ API. Just as re-architecting the kernel code is required to enable parallelism on the hardware and optimize the memory accesses, host programming is equally important to ensure high performance over the CPU. You can have an optimized kernel to meet the required performance but the application performance will not be optimal if the utilization of CPU and FPGA is not high. The following are some of the considerations while creating a host program:

- Reducing the overhead of kernel enqueuing: There is an overhead of dispatching the commands and arguments by the host to the kernel before enqueuing the kernel. You can reduce the impact of this overhead by minimizing the number of times the kernel needs to be enqueued by the host.
- Maximize the data transfer bandwidth between the host and device: The data transfer between host and device memory must be large enough to maximize the PCIe bandwidth. At the same time, the buffer must not be too large to initiate the kernel execution.
- Data availability for the kernel compute: The host must send the data to the FPGA memory as soon as possible so that the compute can be initiated and the kernel is not starved due to data availability.
- Overlapping data transfers with kernel computation: Applications, such as database analytics or video, have a much larger data set than can be stored in the available global device memory on the acceleration device. They require the data to be processed in blocks. Techniques that overlap the data transfers with the computation are critical to achieving high performance for these applications.

You might need to try out different buffer sizes for data movement between host and device memory to optimize the application performance. Using the Vitis tools, you can explore applications using an emulation target for estimated performance results and finally run on the hardware for the accurate performance results. After running the application, you can use Vitis analyzer to visualize the data movement between host and FPGA memory, in addition to kernel and device memory.

The following snapshot is based on the Timeline Trace (host application timeline) as seen in the Analysis view of the Vitis analyzer. This view displays the data movement on the horizontal axis as "Time" to identify any potential performance improvement opportunities.

Figure 9: **Timeline Trace (host application timeline)**

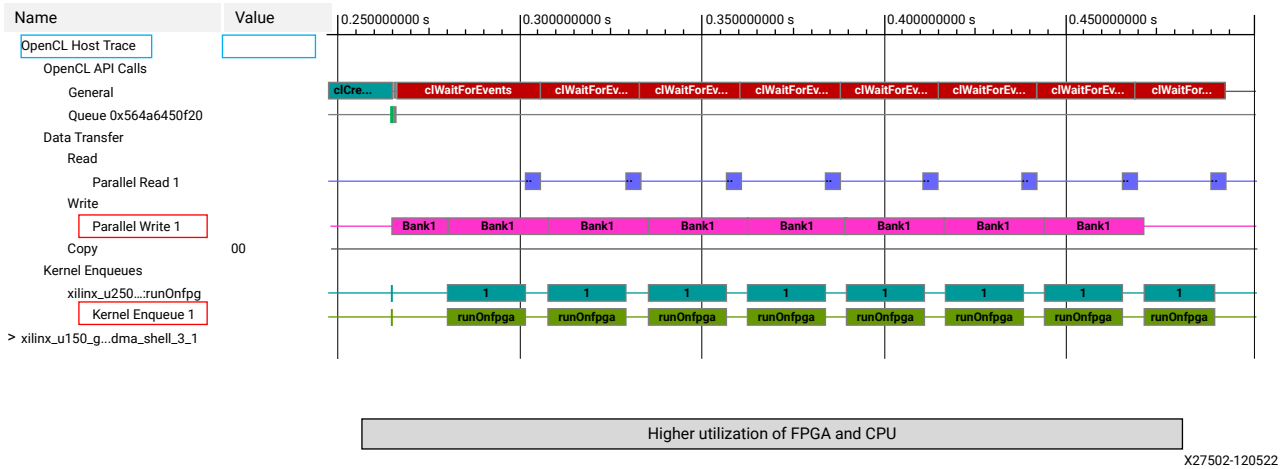


The row with "Data Transfer:Read" displays when the host is reading the data from the device memory and "Data Transfer:Write" displays when the host is writing the data to the device memory. The row with "Kernel Enqueue" displays when the kernel is executing.

Like Task level parallelism achieved within the kernel, similar parallelism can also be achieved between the host and CPU. By enabling this, both CPU and FPGA can be active at the same time and result in high performance. As you program the host code, you need to think of ways to keep the FPGA and CPU both busy at the same time. This is usually the property of the algorithm and the designer has to carefully write the host program to get the max utilization of FPGA and CPU.

Looking at the above Application Timeline, it's easy to identify the gaps when the CPU is idle. Similarly, the FPGA is idle when the host is transferring the data. So the host program needs to feed the data to the device memory as soon as possible so that kernel on FPGA is not starved for data. Here, a large buffer is sent one time from the host to the CPU for computing. The FPGA is idle until the data is completely transferred to the device memory. When the FPGA is executing the compute function, the CPU is idle at that time. For this application, it's not required to send the large buffer and the application can produce better results by overlapping host transfer and kernel compute.

Figure 10: **Improved Timeline**



With the change on the host side by splitting the data into multiple chunks, the kernel compute can be initiated earlier and data transfer between the host and FPGA can be hidden. The host data transfer and the accelerated function on FPGA can now run in parallel and improve overall application performance. This technique has been demonstrated and explained in detail in Bloom Filter Tutorial.

Vitis analyzer is not limited to showing the timeline trace but also provides application guidance on profiling, presents the synthesis reports with timing and resource information, in addition to the critical path in your design. For more information, refer to Working with the Analysis View (Vitis Analyzer) in the *Vitis Reference Guide* (UG1702).

For more information on host programming refer to XRT Native API.

Next Steps

The following tutorials walk through the methodologies on architecting the application, developing the accelerator to meet performance goals, and writing the host code. The tutorials also demonstrate the debugging and visualization capabilities of Vitis.

Bloom Filter Tutorial - Demonstrates how to make key decisions about the architecture of the application, develop the accelerator to meet your desired performance goals, and optimize the host code.

Convolution Example - Walks you through the process of analyzing and optimizing a 2D convolution used for real-time processing of a video stream.

Introduction to Data Center Acceleration for RTL Designers

This chapter is intended for RTL designers who want to accelerate data center applications using AMD FPGA-based Alveo accelerator cards. The AMD Vitis™ application acceleration development flow provides a model to combine RTL designs and a host application into a unified system running on AMD Alveo™ accelerator cards. The goal of this guide is to introduce key concepts for understanding and using the Vitis tools for RTL designers.

The following are key concepts for using RTL designs to create accelerated applications on FPGAs:

- The accelerated data center application is split into host code that runs on the CPU and the RTL design, or RTL kernels that run on programmable logic (PL) region of an Alveo accelerator card.
- Vitis enables existing RTL designs to be used, with limited changes to satisfy interface requirements, by packaging the IP as an RTL kernel using the AMD Vivado™ IP packager.
- The host application running on the x86 CPU uses Xilinx Runtime (XRT) APIs to interact with the device and the accelerators. The XRT APIs let the application read or write any address-mapped register in the accelerators and transfer data buffers to and from global memory in the Alveo card.
- Data transfers between the host and global memory of the accelerator card introduce latency which can be costly to the overall application. To achieve acceleration in a real system, the performance of the RTL kernels must outweigh the added latency of data transfers.
- RTL kernels from Vivado IP have few signal requirements for integration by the Vitis tools, but they must include AXI4-Lite interfaces for accessing address-mapped registers, AXI4 memory-mapped interfaces for connecting to global memory, and clocks and resets for operation.

The next sections of this guide provide an overview discussion of the details of working with Alveo accelerator cards, packaging RTL designs as kernels for use by the Vitis compiler, and using the XRT native API to create host programs for the integrated application. The sections provide references to additional information which you will need to review for a deeper understanding of the Vitis tools and development environment.

Terminology

The following introduces some of the tools and terms used in this document:

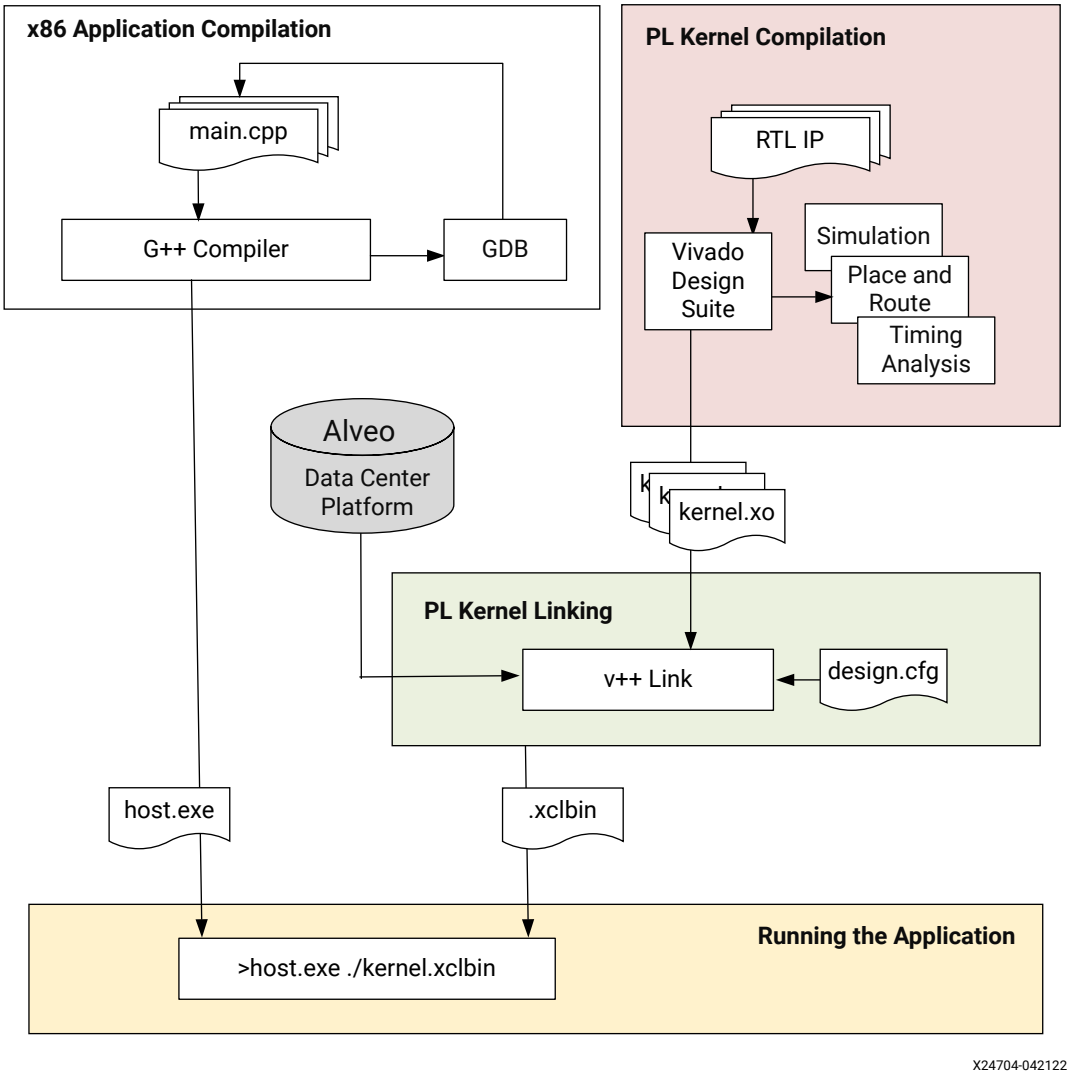
- **Vitis core development kit:** Provides a framework for developing and delivering FPGA accelerated applications using standard programming languages for both software and hardware components. The software component, or host program, is developed using C/C++ to run on a CPU with XRT API calls to manage runtime interactions with the accelerator. The hardware component, or kernel, can be developed using C/C++ or using Verilog or VHDL.
- **Alveo data center accelerator cards:** Are PCI Express® Gen3 x16 compliant cards designed to accelerate compute intensive applications such as machine learning, data analytics, and video processing.
- **Platform:** A predefined configuration of the Alveo accelerator card with features implemented for specific applications. A platform has multiple partitions. The base logic partition (BLP) is a static region that contains fixed logic for essential functions (such as PCIe and DMA). A user logic partition (ULP), which is a dynamic region where the C++ or RTL kernel logic, is programmed for execution.
- **XRT:** The Xilinx Runtime library that provides an API and drivers for your host program to connect with the target FPGA platform and handles transactions between your host program and accelerated kernels.
- **Host and Global Memory:** The distinction between memory on the host machine used by the CPU, and memory on the Alveo data center accelerator card used by the accelerated functions.
- **Vitis HLS:** A high-level synthesis tool that translates C/C++ functions into device logic using programmable logic (PL) elements and RAM/DSP blocks. Vitis HLS synthesizes the C/C++ code into an RTL design and packages it as a compiled object (`.xo`) file that can be imported into the Vitis environment. There can be multiple functions targeted on the FPGA, each being a separate kernel. Vitis HLS will synthesize these kernels one by one and generate separate `.xo` files.
- **Register transfer level (RTL):** An abstraction level used for modeling digital circuits. Often the term RTL is used interchangeably for Verilog or VHDL which are both hardware description languages.
- **PL Kernel (.xo) file:** Is the term to designate the custom logic implementation of your accelerator function. Each kernel is packaged as an `.xo` file and contains the IP for the function and associated metadata used by the Vitis tool.
- **RTL Kernel:** Is an RTL design that uses standard AXI4 interfaces to enable the Vitis compiler to link it into the target platform to quickly build the system design. The RTL kernel is packaged as an `.xo` file.
- **Vivado Design Suite:** An RTL language synthesis and implementation tool that takes the RTL design generated by Vitis HLS or an RTL designer and generates the bitstream that can be loaded and executed on the FPGA.
- **Bitstream:** Is the configuration data that is used to program the FPGA so that its functionality can be changed. The kernel design will result in a bitstream used to program the dynamic region of the FPGA on the Alveo accelerator card.

- **Device Binary (.xclbin) file:** Contains the bitstream and other metadata needed to be used to program the FPGA. This is used by XRT APIs to actually program the FPGA. In the Vitis flow, the device binary files have the extension `.xclbin`.
- **Vitis analyzer:** A utility that allows you to view and analyze the reports generated while building and running the application through Vitis, Vitis HLS, and AMD Vivado™.

Vitis Development Flow for RTL Designers

This section provides a brief overview of the Vitis application development flow for RTL designers. An image of this flow is shown below.

Figure 11: **Vitis Development Flow with RTL Kernels**



The development flow includes the following steps:

- Application Compilation using G++

  The host program is written in C/C++ using XRT native API, and compiled using a `g++` compiler to create a host executable file to run on the x86 processor. The host program interacts with RTL kernels in the PL region on the FPGA to complete the accelerated application.

- PL Kernel Creation using Vivado Design Suite

  The RTL design is developed and optimized for AMD FPGA devices using the Vivado tools. The steps for kernel development in the Vivado tools include:

  1. Edit RTL code to design the function
  2. Verify the RTL using behavioral simulation in Vivado simulator
  3. Verify the RTL synthesis, place and route, and timing
  4. Analyze performance by reviewing timing reports
  5. Repeat previous steps until performance goals are met
  6. Package the RTL IP as a kernel (`.xo`) for use in the Vitis flow

- PL Kernel Linking using Vitis Tools

  Xilinx object (`.xo`) files are linked with the target hardware platform by the Vitis linker to create a device binary file (`.xclbin`) that is loaded for execution on the Alveo accelerator card.

  TIP: *This step will call Vivado place and route to generate the bitstream as part of the `.xclbin` file.*

  To help define the architecture of the device binary, a configuration file (`design.cfg`) can be created to specify how the kernels are connected to the global memory or to each other, or to define how many instances of a kernel (or Compute Unit) to build in the device binary to allow multiple functions to run in parallel. This configuration file is passed to the Vitis linker to generate the `.xclbin`.

- Running the Application:

  Finally, when you run the application the host program loads the `.xclbin` file generated by Vitis Compiler. The host application always runs on the CPU, and the RTL kernel can be run in emulation mode on the x86, or load the `.xclbin` on the FPGA on the Alveo accelerator card.

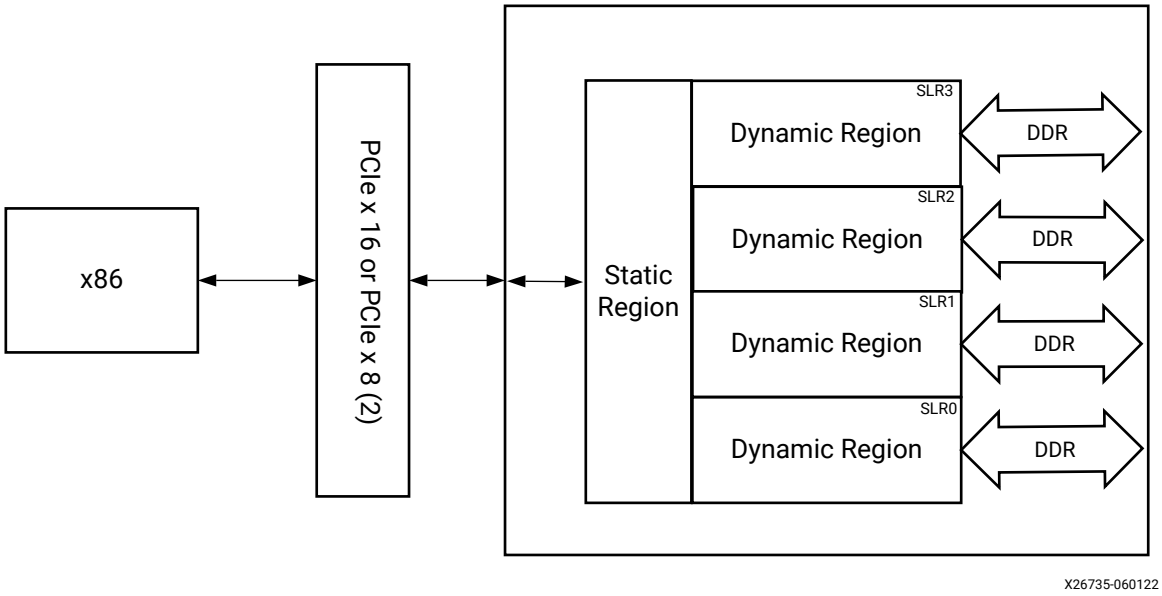
Using Alveo Accelerator Cards

AMD has developed the Alveo family of PCIe Data Center accelerator cards using FPGAs at its core. Each Alveo card combines three essential things: a powerful FPGA for acceleration, high-bandwidth device memory banks, and connectivity to a host server via a high-bandwidth PCIe Gen3x16 link. A number of different cards are available to provide designers with a choice of features and quantity of programmable resources. Below is the block diagram for the Alveo U250.

Although FPGAs are essentially blank devices that get configured at power-up, all Alveo cards are shipped with target platforms that provide the firmware to configure the accelerator card for specific uses. The platform must be installed with Xilinx Runtime (XRT); flashed into the device during installation, or when changing the configuration of the accelerator card.

On the AMD device, the platform consists of two physical FPGA partitions: *Shell* and *User*. The Shell partition is a static region and provides basic infrastructure for the platform like PCIe connectivity, board management, sensors, clocking, and reset. The User partition is a dynamic region that contains user compiled binary called `.xclbin` which is loaded by XRT during execution. RTL kernels are the custom logic created by the developer and programmed into the dynamic region. In this document, kernels refer to the functions that the designer is implementing into the dynamic region of the Alveo accelerator card.

Figure 12: **Alveo Block Diagram**



The PCIe interface is used for communication between the host and accelerator card, and to transfer data from the host into the Alveo card's device memory. This device memory serves as a global memory, accessible by both host and hardware accelerators. The device memory included on the Alveo platform are PLRAM (small size but fast access with the lowest latency), HBM (moderate size and access speed with some latency), and DDR (large size but slow access with high latency). Depending upon the Alveo card, you can have DDR or HBM, or even both.

The block diagram shown above is of U250 and has four banks of DDR, each with 16 GB of memory. The FPGA on the Alveo card is further subdivided into multiple super logic regions (SLRs), which aid in the architecture of very high-performance designs. As you develop RTL kernels for implementation into the dynamic region of the platform you will need to manage the design constraints of SLRs and global memory.

To further improve performance, and minimize access to DDR memory, FPGAs have large quantities of small, internal RAM blocks. These are completely configurable by the compiler to ensure that buffering can be created between tasks to enable pipeline-style computation. This effectively eliminates the need for caches and is one of the key strengths of FPGAs.

There are many more details you could learn about the FPGA architecture and Alveo cards, but this is sufficient for introductory purposes. From the perspective of designing an FPGA-based acceleration architecture, the important points to remember are:

- Moving data across PCIe is expensive - even at Gen3x16, latency is high. For larger data transfers, bandwidth can easily become a system bottleneck.
- Bandwidth and latency between the DDR4 and the FPGA are significantly better than over PCIe, but touching external memory is still expensive in terms of overall system performance.

Differences between Vivado and Vitis Development Flows

As an RTL designer, you might be familiar with or have worked with the Vivado Design Suite. This tool is also at the center of the Vitis design environment, and your experience working with the Vivado tool will benefit you in this design flow.

RTL Development

The RTL kernel development flow requires you to standardize any RTL IP you have for use in the Vitis design flow. This means creating RTL kernel (`.xo`) files from existing IP, making any necessary modifications to support the AXI4 interfaces required for the Vitis tools as described in Requirements of an RTL Kernel. The RTL kernel development flow lets you modify existing custom IP that you might have already packaged for use in the Vivado tool, or start your RTL design from scratch and package it for use in the Vitis flow.

Creating an RTL kernel follows the tradition RTL IP development process: you will write the RTL code, simulate it with the Vivado logic simulator, or any supported third-party simulator, and place and route the design to insure it passes timing and routes to completion. You might also define XDC constraints for use with your design as needed to complete implementation. When you are ready, or when your design is complete, you can package the RTL design as an IP and generate the RTL kernel (`.xo`) files for use by the Vitis compiler in building the system design. In terms of the RTL design and IP packaging process everything is the same, with the additional output of the `.xo` file.

System Design

In the Vivado development flow you would manually add and stitch the required IP together using the IP integrator of the tool, or define your top-down system using RTL. In the Vivado flow you will need to specify the overall system design, complete with PCIe bus, global memory, and peripheral features, outside of the FPGA design. You would need to create the custom host code incorporating drivers to access features of the system card or the programmable logic.

In the Vitis application acceleration flow the compiler links the RTL kernel, or multiple kernels, with the target platform of the Alveo accelerator card, automatically building the system design using the IP integrator feature. The Vitis compiler automatically instantiates a Memory Subsystem (MSS) IP into the system design to manage AXI traffic between kernels, the host processor, and memory resources. Configuration of the MSS is derived from the connectivity section of a configuration file used during linking as described in Linking the System. XRT provides the underlying runtime and drivers, and provides an API for developing the host application to access the accelerator card.

The Vivado development flow requires synthesis, place and route, and timing closure for your design. The Vitis flow creates the Vivado project during the linking process, and automates the synthesis and implementation of the design. While this is automated within the Vitis tool flow, you can completely control the process, using Tcl scripts or working interactively in the Vivado tool to address design problems and generate the desired results.

While the Vivado and Vitis tools both provide the system design capability, the Vitis tools standardize much of the required ecosystem. The Vitis flow automates several steps like integrating with PCIe, and adding global memories. This lets you focus on developing the RTL function and reduces the overall development time. With the Vitis flow, it is also easier to migrate to another accelerator card seamlessly, and most of the time without any change in the RTL component or the host code.

Creating and Packaging RTL Kernels

Creating an RTL kernel begins with creating an IP within the Vivado Design Suite. You might have an existing RTL IP in your repository, or might want to create a new RTL design to package as an IP. Either approach is a good place to start in creating a new RTL kernel.

Execution Protocol

The RTL kernel employs a user-managed execution scheme where the host application generally uses register reads and writes to manage the execution and completion of the RTL kernel function. This user-managed execution protocol lets you use the control scheme of existing IP in the Vitis environment with little or no redesign, as explained in Creating User-Managed RTL Kernels. Typically this includes the use of an `s_axilite` interface and using XRT native API object classes and methods for reading and writing to register addresses on the kernel.

Port Interface Protocols

RTL kernels, like all kernels in the Vitis development flow, support four types of interfaces:

- AXI4-Lite (`S_AXILITE`) for control registers, buffer pointers, scalar values, and kernel interactions with the host. The data is accessed by register reads and writes
- AXI4 Memory Mapped (`M_AXI`) for access from the kernel to global memory or host memory. Data is accessed by the kernel through memory such as DDR, HBM, PLRAM/Block RAM/URAM
- AXI4-Stream ports to stream data between kernels, or other streaming sources such as a video processor or camera
- Custom (non-AXI) interfaces are also supported using the `--connectivity.connect` command to make connection to these ports during the `v++` linking process. This process can be used to connect your kernel to GT ports on the platform for instance

In addition, the kernels must have at least one clock, but can support multiple clocks, reset signals, and interrupts, as discussed in Kernel Interface Requirements. If your original IP does not use AXI4 interfaces, or provide the needed clock signal, you will need to modify and repackage the current IP to provide these signals.

A platform can have scalable clocks and fixed clocks. The Vitis flow can generate any number of derived fixed clocks that are not provided by the platform and are commonly used for RTL kernel flow. When fixed clocks are used, Vitis flow will insert an MMCM into the system design to generate the required frequencies. Refer to Managing Clock Frequencies for more information.

The data bus width of the AXI master and stream ports are configurable. Normally this depends on data transfer bandwidth and FPGA resource considerations.

The design of the RTL kernel is highly flexible. You can decide the interaction method between the kernel and the host application, the internal clock generation scheme, clock gating strategy, handling of interrupts.

TIP: *You might need to use the AXI Clock Converter as part of the AXI Interconnect Core IP to manage cross-domain clocking from outside the kernel to inside. It might be hard to achieve timing closure if you treat the external bus clock and internal bus clock as synchronous.*

Packaging the IP

To generate the RTL kernel you must run the Package IP process in the Vivado tool, as described in Packaging the RTL Code as a Vitis XO. This is the standard IP packaging flow as described in *Vivado Design Suite User Guide: Creating and Packaging Custom IP* (UG1118), with the additional step of specifying the kernel for use in the Vitis design flow.

Building Kernels and Managing Implementation

With the RTL kernel generated from a packaged IP into a compiled `.xo` file you are ready to link the kernel with the target platform and with other kernels and build the system design. Designs with user-managed RTL kernels support hardware emulation builds and hardware builds as described at Selecting the Build Target.

During the build process the Vitis compiler launches the Vivado Design Suite to automatically place and route the design, and generate the bitstream and the `.xclbin` file. While the build process is automated by the Vitis compiler, it also offers the opportunity for specifying constraints on complex designs or interactively working in the Vivado tools to help you resolve timing and generate the `.xclbin` file, as described in Managing Vivado Synthesis, Implementation, and Timing Closure.

The Vivado tools also provide a rich Tcl programming language for scripting and automating elements of the design flow. You can provide Tcl scripts to be run at different stages of the build process for the Vitis compiler or the Vivado tool. These scripts can be enabled for specific steps in the build process using --linkhook Options, or using Vivado properties as explained in --vivado Options.

Writing Host Applications with XRT API

The general structure of a host application in the Vitis development flow can be divided into the following steps, with example code provided:

1. Specify the accelerator device ID and load the `.xclbin`:

```
auto device = xrt::device(device_index);
auto uuid = device.load_xclbin(binaryFile);
```

2. Setup the kernel IP, and define the argument buffers:

```
auto ip1 = xrt::ip(device, uuid, "Vadd_A_B:{Vadd_A_B_1}");
// Allocate Buffer in Global Memory
auto ip1_boA = xrt::bo(device, vector_size_bytes, 1);
auto ip1_boB = xrt::bo(device, vector_size_bytes, 1);
// Map the contents of the buffer object into host memory
auto bo0_map = ip1_boA.map<int*>();
auto bo1_map = ip1_boB.map<int*>();
```

3. Transfer data from the host to the device memory, if the kernel is written to access device memory:

TIP: *The kernel can also be written to access host memory directly when the platform supports that ability, as described in Host Memory Access.*

```
// Get the buffer physical address
buf_addr[0] = ip1_boA.address();
buf_addr[1] = ip1_boB.address();
// Synchronize buffer content with device side
ip1_boA.sync(XCL_BO_SYNC_BO_TO_DEVICE);
ip1_boB.sync(XCL_BO_SYNC_BO_TO_DEVICE);

// Setting Register A (Input Address)
ip1.write_register(A_OFFSET, buf_addr[0]);
ip1.write_register(A_OFFSET + 4, buf_addr[0] >> 32);

//Setting Register B (Input Address)
ip1.write_register(B_OFFSET, buf_addr[1]);
ip1.write_register(B_OFFSET + 4, buf_addr[1] >> 32);
```

4. Run the kernel and returning results:

```
// Start Kernel by writing to the Control Register
uint32_t axi_ctrl = IP_START;
ip1.write_register(USER_OFFSET, axi_ctrl);

// Wait until the IP is done by reading from the Control Register
while (axi_ctrl != IP_IDLE) {
    axi_ctrl = ip1.read_register(USER_OFFSET);
}

// Sync the output buffer back to the Host memory
ip1_boB.sync(XCL_BO_SYNC_BO_FROM_DEVICE);
```

To support user-managed RTL kernels, you will write your host application using the XRT Native API, defining the RTL kernel as an `xrt::ip` object as shown above, and accessing the kernel through register read and write commands. Refer to Writing the Software Application for more information.

With the host application written, you can compile it using the standard g++ compiler as described in Compiling and Linking for x86.

Next Steps

The Vitis development flow provides a rich environment for RTL designers. The environment provides the tools necessary to develop use existing IP or develop new IP for use in Data Center applications. The Alveo accelerator cards provide advanced systems for application acceleration, and the Vitis compiler provides the tools and drivers to let you quickly connect your RTL designs with the target platform and host applications, greatly increasing your productivity for RTL design based application acceleration.

For an example on creating and building a user-managed RTL kernel and host application refer to RTL Kernel Workflow tutorial for step by step instructions on the process. The Vitis Accel Examples repository provides additional examples of RTL kernels and host application.

Introduction to the Vitis Unified IDE

The Vitis Unified IDE is a design environment for developing applications for AMD Adaptive SoC and FPGA devices. The Vitis Unified IDE embraces a bottom-up design flow that lets you develop components of a system: C/C++ sourced HLS components, RTL design kernels, software applications, and then integrate the components into a top-level system project as shown in the following figure. The single unified development environment provides all the features you need to compile, run, debug, and analyze the different elements of an heterogeneous embedded system design, or an FPGA-accelerated Data Center application.

The Vitis Unified IDE lets you create, synthesize C/C++ code into RTL designs using HLS components, run C-simulation and C/RTL Co-simulation; review and analyze build and run summaries in the newly integrated Vitis analyzer tool.

The Vitis Unified IDE works with the common command-line features of the `v++` and `vitis-run` commands. Whether working from the command-line or from the Vitis Unified IDE, the environment provides a single tool set for end-to-end application development without the need to jump between multiple tools for design, debug, integration and analysis.

You can perform the following tasks with the Vitis Unified IDE:

- Design programmable logic with C/C++ by creating HLS components
- Develop System projects for AMD Alveo™ Data Center accelerator cards

The Vitis Unified IDE provides the following benefits:

- **Architected for ease-of-use:**
  - The Flow Navigator helps you manage the work flow for different designs
  - The design flow supports example template for new users to view all available examples, increasing productivity
  - Non-blocking commands can now run multiple build and analysis jobs at the same time
- **Easier switching between GUI and command-line (CLI) mode:**
  - Combining strengths of both GUI and CLI
  - Configuration files are rendered in real time
  - CLI can be used to build projects, and the unified IDE for debugging and analysis
  - GUI operations are saved in python log for batch rebuilding
- **Modern look and framework:**
  - Light and dark themes
  - Quick actions with fully customizable shortcut keys

- User-friendly command palette
- Up-to-date C++ syntax highlighting and IntelliSense

Features

The next generation Vitis Unified IDE supports the following features:

- Support for data center application acceleration development
- Support for AI Engine applications on embedded platforms
- Support for standalone AI Engine component workflow
- Creation of HLS components for synthesizing RTL from C/C++ code
- Integrated Vitis analyzer for helping you review, analyze, and iterate in a single tool
- Ability to create applications from templates or examples
- Support for run and debug applications in emulation or on hardware
- Support for project flow and custom Makefile flow
- Runs in Jupyter Notebook environment
- Runs in batch or interactive mode
- Support for source control with git
- Command line interface for running project creation and building actions and query project status

The editor supports the following features:

- Syntax highlighting for supported languages including C, C++, Python, `CMakeList.txt`
- Smart editor for `v++` config file and `xrt.ini`
- Quick viewing of function definitions

Components and System Projects

The Vitis Unified IDE manages designs at the component level and the System project level. The top-level System project can contain different components, such as the Application component that holds the source code for X86 applications, the HLS component (PL kernel) and the Platform component, all of which can be separately developed, built, and analyzed in the new Vitis Unified IDE.

Components are the basic building blocks in the Vitis Unified IDE. Each component contains source code and configuration files. The component can be built within their own scope and can be run or debug on their own, or together with other components in a System project. The System assembles multiple components for a system design. It contains the component link configuration. When building a System project, the new Vitis Unified IDE will run, link and package using the build output of the included components. If the required component output files are not available, the Vitis Unified IDE will prompt you to build each component.

The metadata for components are stored in `vitis-comp.json`; the metadata for applications are stored in `vitis-sys.json`, in addition to links to the configuration files for building, linking and packaging the system and its components.

System Project Configuration Files

There are two types of configuration files for a Vitis system project:

CMake Files

There are two CMake files: `CMakefileList.txt` and `UserConfig.cmake`.

The parameters defined in `CmakeList.txt` are translated to configuration settings to the compilers by CMake, a commonly used build utility. The `UserConfig.cmake` is included by the `CMakeList.txt` file. It provides the user interface to modify the configurations for compilers, and supports both GUI mode and text editor mode.

***Note*:** Users are advised against direct modification of the `CMakeList.txt`. Instead, it is recommended for users to adjust compiler settings within the `UserConfig.cmake` file.

Configuration Files

The new Vitis Unified IDE provides a Config File editor with GUI rendering for option selection with example syntax. The default view is GUI rendering, but you can view the text form of the config file by clicking the `</>` button to switch to the code view. The code view can be used for manual entry of configuration commands, which can be necessary in some circumstances. There are four configuration files used in the system:

- **HLS Configuration (`hls_config.cfg`):** This is the configuration file for the HLS component to drive synthesis of the C/C++ code into an RTL module. It can be used to define properties of the design for synthesis, simulation, co-simulation, and implementation. The `hls_config.cfg` file is used by the `v++ -c --mode hls` command in addition to the `vitis-run` command.
- **Hardware Link Configuration (`hw_link/binary_containers_1-link.cfg`):** This is a configuration file for the System project and defines build instructions for linking the system of components with the platform. The `hw_link/binary_containers_1-link.cfg` configuration file is used by the `v++ --link` command.

Accelerating Data Center Applications with Vitis Software Platform

Introduction

This content 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. The device developer builds the factories, assembly lines, and workstations, and then customizes them for the required task instead of using general purpose tools. Similar to lot size, device real-estate is not infinite, which limits the number and size of the factories which can be built in the device. Proper architecture and configuring of 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

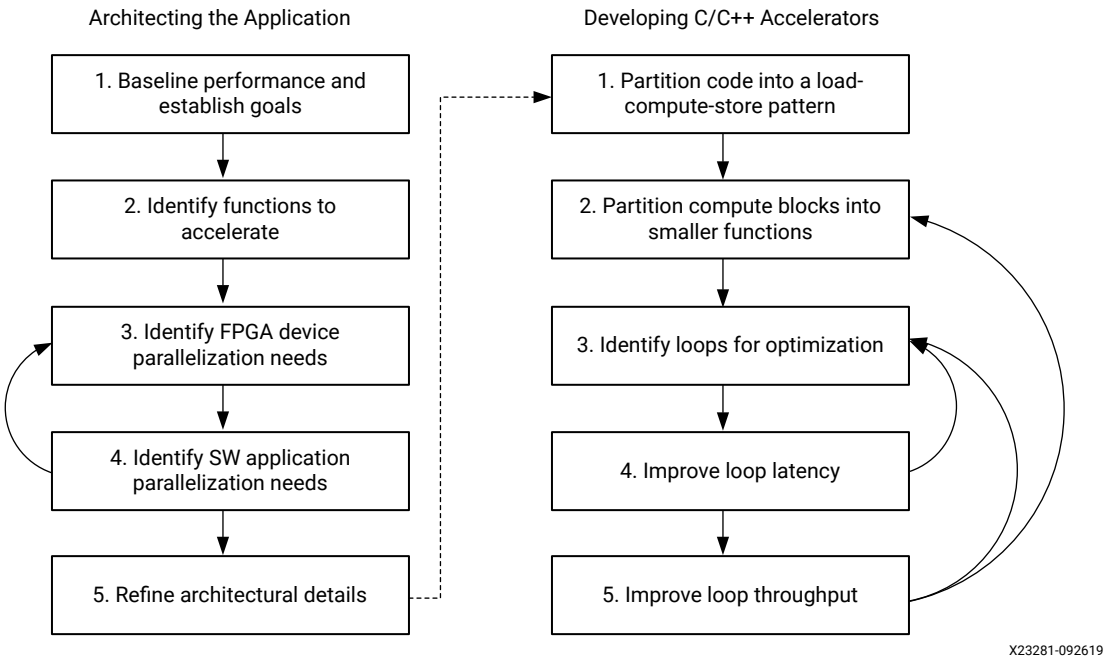
The methodology is comprised of two major phases:

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 to map to device kernels, how much parallelism is needed, and how it must 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 13: **Methodology Overview**



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 AMD Vitis™ unified 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 3: Developing Vitis Kernels and Applications
- Vitis Application Acceleration Development Flow Tutorials on GitHub.

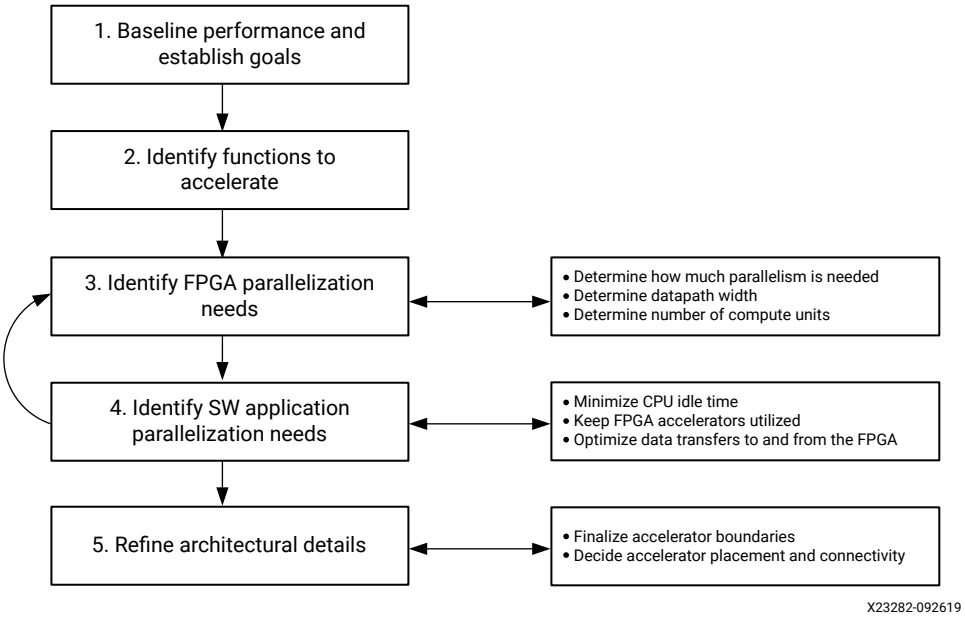
In addition to understanding the key aspects of the Vitis software platform, a good understanding of the following topics will help achieve optimal results with this methodology:

- Application domain
- Software acceleration principles
- Concepts, features and architecture of device
- Features of the targeted device accelerator card and corresponding target platform
- 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 to map to device kernels, how much parallelism is needed, and how to deliver it.

Figure 14: **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, to identify bottlenecks of the current application running on your existing platform. These performance numbers must be generated for the entire application (end-to-end), in addition for each major function in the application. 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. These numbers provide the baseline for most of the subsequent analysis process. The functions that consume the most execution time are good candidates to be offloaded and accelerated onto FPGAs.

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 15: **Gprof Output Example**

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
time   seconds   seconds    calls  ms/call  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, target platform, 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 xrt-smi dma test.

Figure 16: **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 is 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 drives 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 to accelerate in the device.

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.*

When selecting functions to accelerate in hardware, there are two aspects to consider:

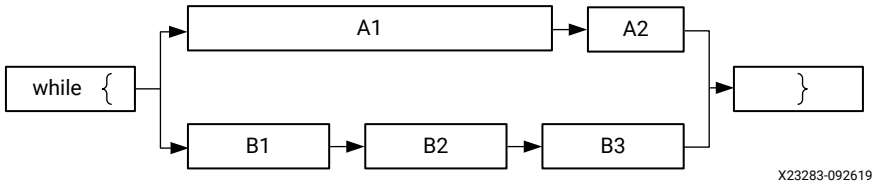
- **Performance bottlenecks:** Which functions are in application hot spots?
- **Acceleration potential:** Do these functions have the potential for acceleration?

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 the 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 17: **Application with Two Parallel Paths**



The above performance visualization in the context of parallelism shows that accelerating only one of the two paths does 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, does 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, instead of only 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 is 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 because the overhead of moving data to the accelerator is 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, because 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 does 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.

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

  On AMD Alveo™ Data Center accelerator cards, the PCIe bus imposes a throughput limit on data transfers. While it might 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 10 Gb/s and a software throughput of 50 MB/s, 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 are identified and the overall acceleration goals are 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, similar to putting 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 needs to determine which combination of parallelization techniques is 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 target platform. 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 must then determine the best combination given their needs and the characteristics of their application.

Determine How Many Samples the Datapath Must Process 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 to process in parallel to meet the performance goal.

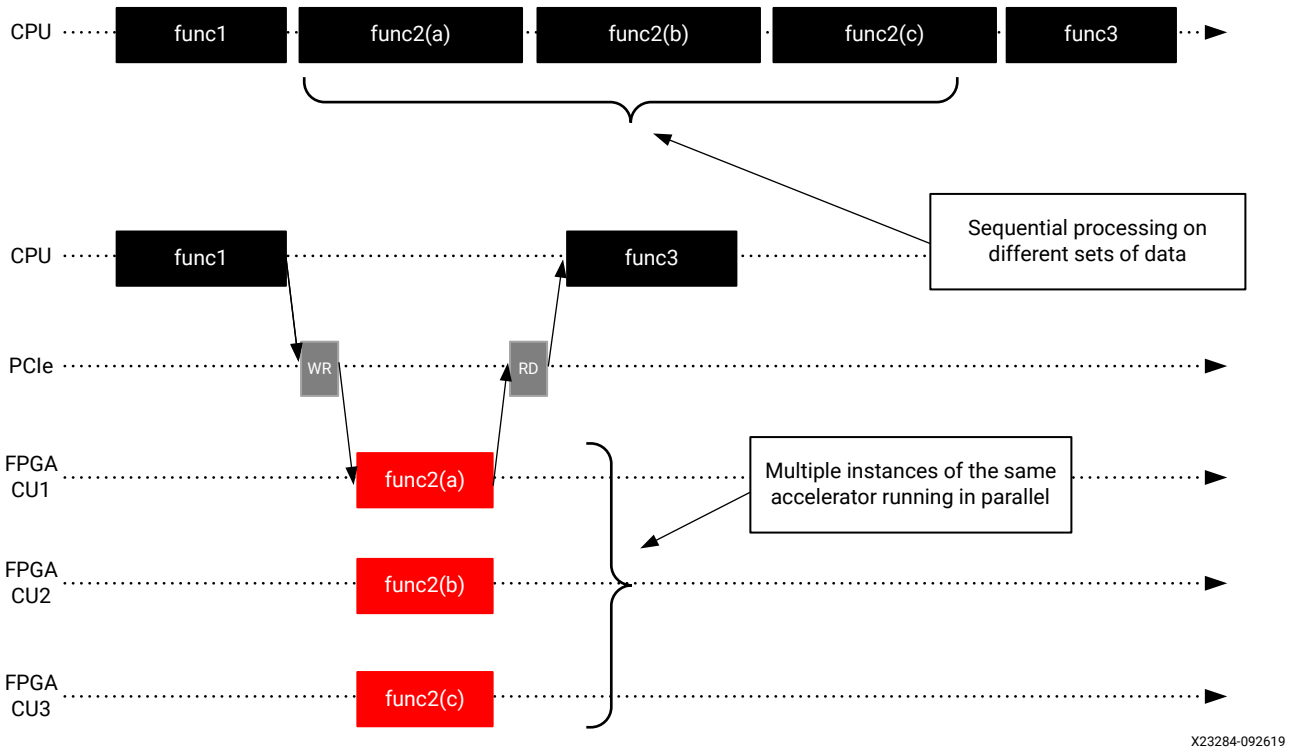
Processing more samples in parallel using a wider datapath improves performance by reducing the latency (running time) of the accelerated function.

Determine How Many Kernels To Instantiate 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 18: **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, you are likely to 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.

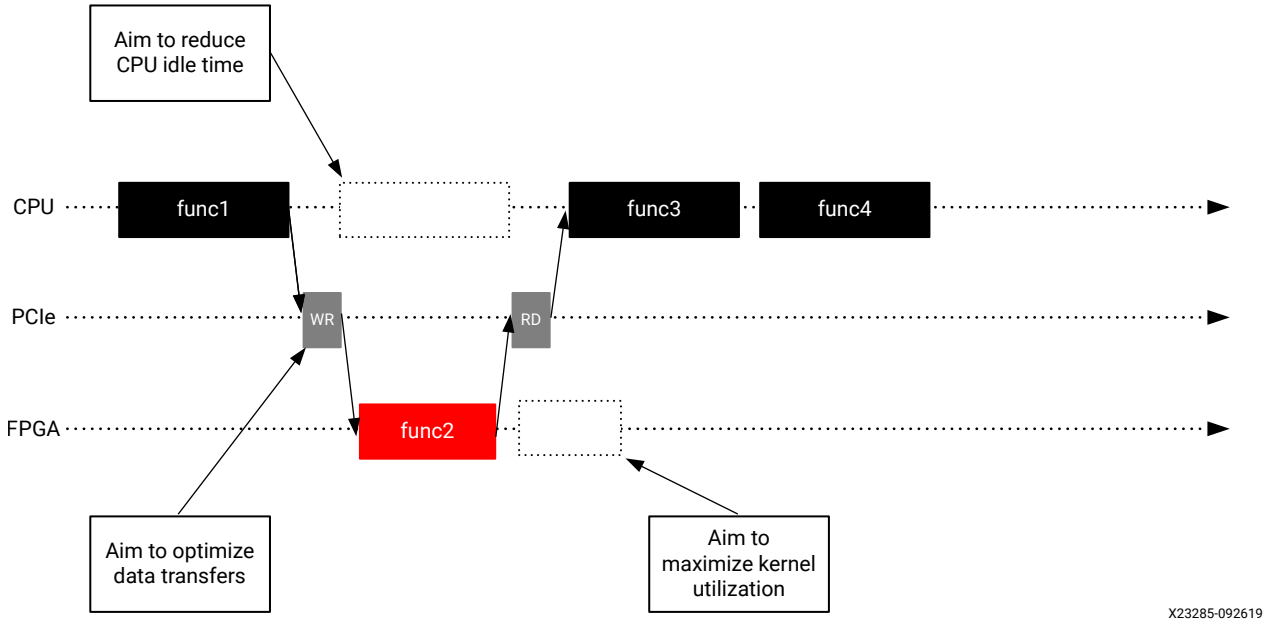
Step 4: Identify Software Application Parallelization Needs

While the hardware device and its kernels are designed 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 19: **Software Optimization Goals**



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 does not 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.

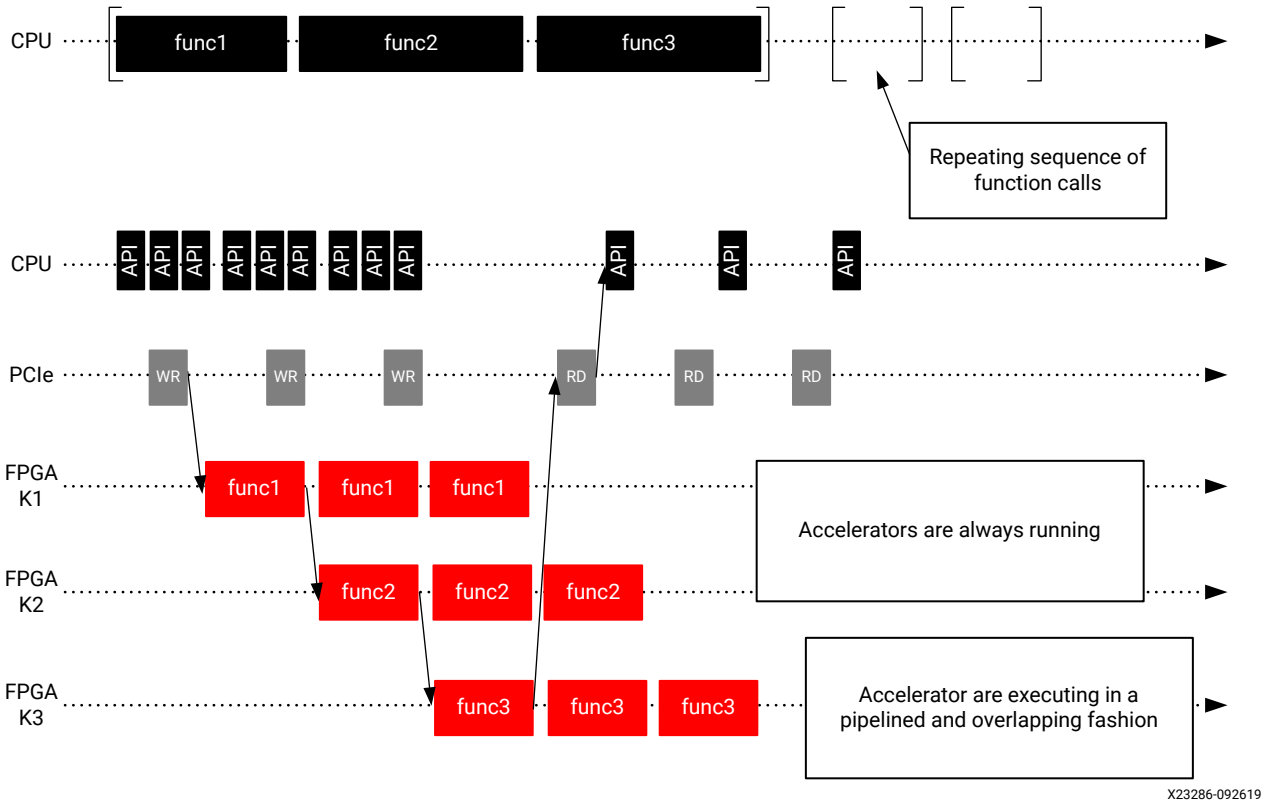
Instead, engineer the software application to avoid such idle cycles. Begin by identifying parts of the application that do not depend on the results of the kernel. Then structure the application so that these functions can be executed on the host in parallel to the kernel running in the device.

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 keeps 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 20: **Pipelined Execution of Accelerators**



In this example, the original application repeatedly calls func1, func2 and func3. Corresponding kernels (K1, K2, K3) are 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 Vitis Application Acceleration Development Flow Tutorials.

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.

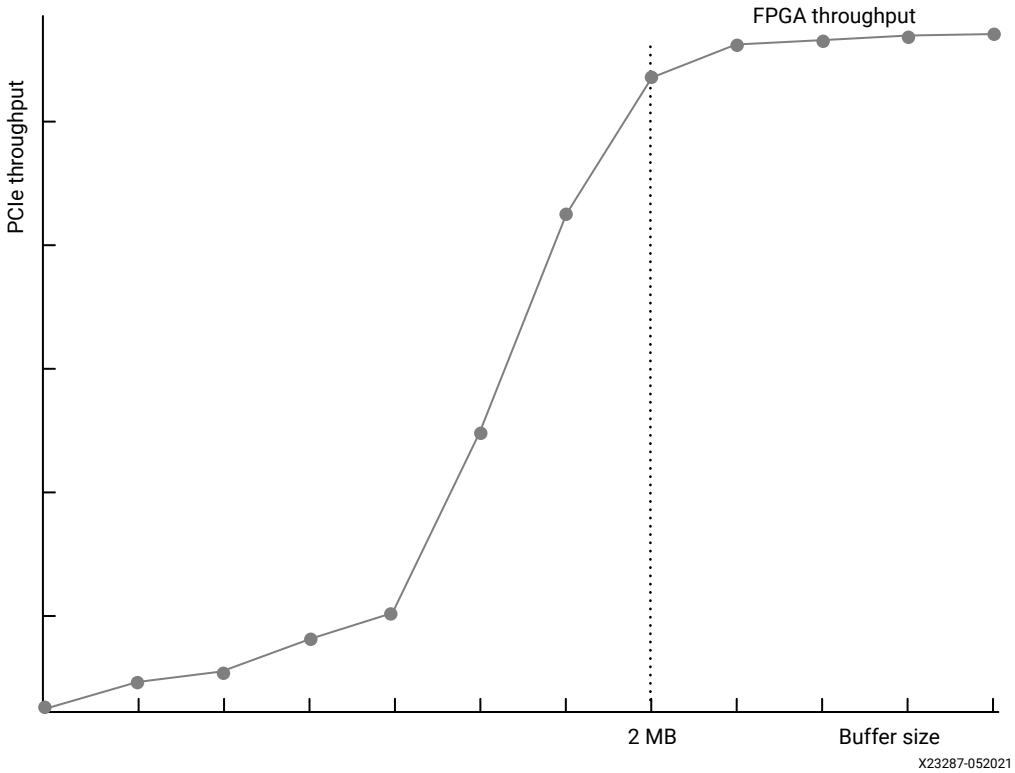
Data needs to be transferred at the right time, otherwise the application performance is negatively impacted if the kernel must wait for data to be available. It is therefore important to transfer data ahead of when the kernel needs it. This is achieved by overlapping data transfers and kernel execution, 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 1 MB 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 1 MB or more. Running DMA tests in advance using the `xrt-smi` utility can be useful for finding the optimal buffer sizes.

Figure 21: **Performance of PCIe Transfers as a Function of Buffer Size**



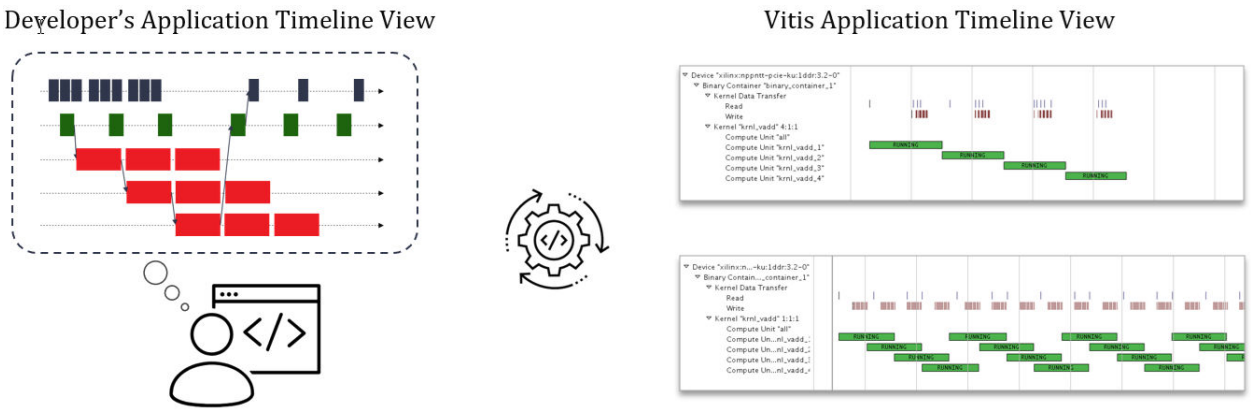
AMD recommends grouping multiple sets of data in a common buffer to achieve the highest possible throughput.

Conceptualize the Desired Application Timeline

You now have a good understanding of what functions need to be accelerated, what parallelism is needed to meet performance goals, and how the application will be delivered.

At this point, it is very useful to summarize this information in the form of an expected application timeline. The timeline sequences, such as the ones shown in Keep the Device Kernels Utilized, 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 22: **Timeline Trace**



The Vitis software platform generates timeline views from actual application runs. If the developer has a desired timeline in mind, they can 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 are 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 target platform 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 Software Platform Release Notes* (UG1742).

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 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 must 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 Packaging RTL Kernels.

The following key kernel requirements for optimal application performance need to 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.

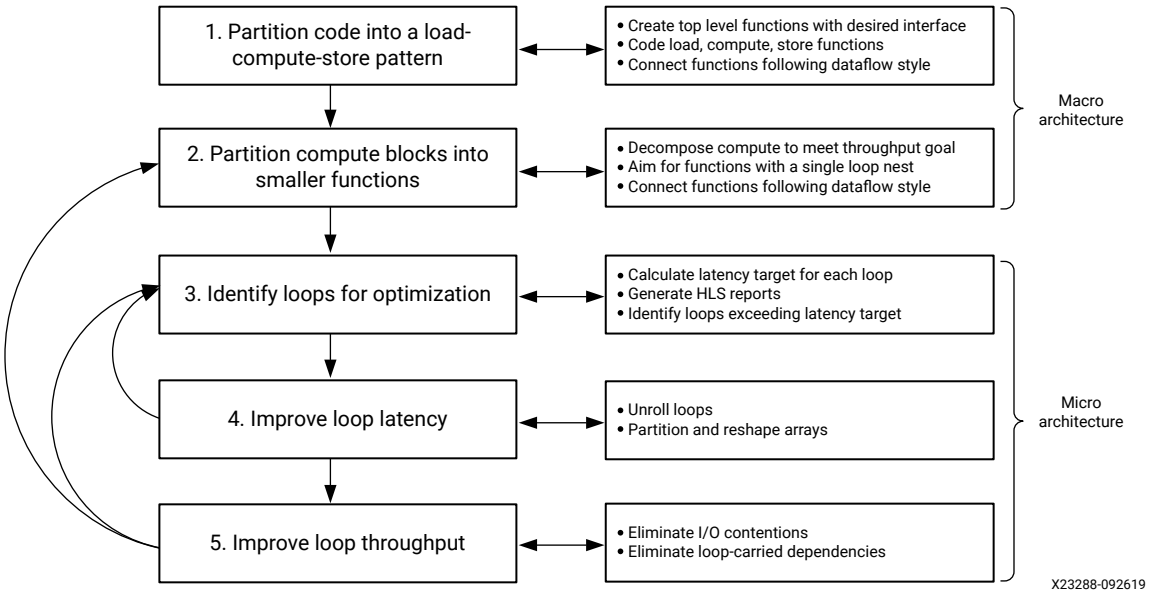
- Functionality is the mathematical relationship between input parameters and output results.
- Algorithm is a series of steps for performing a specific functionality. A given functionality can be performed using a variety of different algorithms. For instance, a sort function can be implemented using 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 generates optimized hardware architectures from algorithms written in C/C++. However, it does not transform a particular algorithm into another one. Even if the software program can be automatically converted (or synthesized) into hardware, achieving acceptable quality of results (QoR), requires additional work such as rewriting the software to help the HLS tool achieve the desired performance goals.

Therefore, because the algorithm directly influences data access locality, in addition to 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. To help, you need to understand the best practices for writing good software for execution on the FPGA device. The next few sections discuss how you can first identify some macro-level architectural optimizations to structure your program and then focus on some fine-grained micro-level architectural optimizations to boost your performance goals.

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

Figure 23: **Kernel Development Methodology**



About the High-Level Synthesis Compiler

Before starting the kernel development process, it is recommended that you 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. For more information, see the *Vitis High-Level Synthesis User Guide* (UG1399).

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 helps you structure the code to meet the application throughput goals. For specific information on programming kernels, see Developing PL Kernels using C++.

Verification Considerations

This methodology described in this guide is iterative in nature and involves successive code modifications. AMD 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.

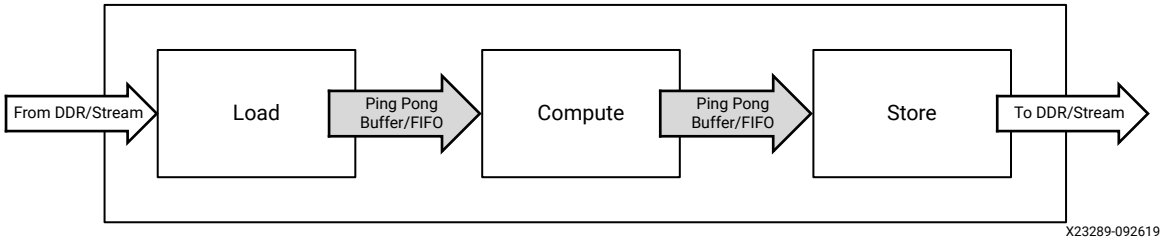
Knowing that kernel accesses to global memory are expensive and that bandwidth is limited, it is very important to carefully plan this aspect of the kernel.

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 24: **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 Task Parallelism Using Different Kernels in the *Data Center Acceleration using Vitis* (UG1700).

The load function is responsible for moving data external to the kernel (that is, global memory) to the compute function inside the kernel. This function does not perform 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 is not 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 must be passed as scalars. Blocks of input and output data must be passed as pointers. Compiler pragmas must be used to finalize the interface definition.

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.

AMD recommends matching 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, ports must be made 128-bit wide to allow reading and writing 16 bytes in parallel.

In some cases, it might be useful to access the full width bits of the interface even if the datapath does not 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 must always be avoided unless absolutely required. The load and store functions must be coded to always infer bursting transaction. This can be done using a `memcpy` operation as shown in the `vadd.cpp` file in the Vitis Accel Examples, or by creating a tight `for` loop accessing all the required values sequentially, as explained in Chapter 3: Developing Vitis Kernels and Applications.

Minimize the Number of Data Transfers from Global Memory

***Note*:** Only make necessary transfers as accesses to global memory can add significant latency to the application.

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 the topic on Using HLS Streams in the Vitis HLS User Guide (UG1399).

When connecting the functions, use the canonical form required by the HLS compiler. For more information, see the topic on Dataflow Optimization in the Vitis HLS User Guide (UG1399). This helps the compiler build a high-throughput set of tasks using the dataflow optimization. Key recommendations include:

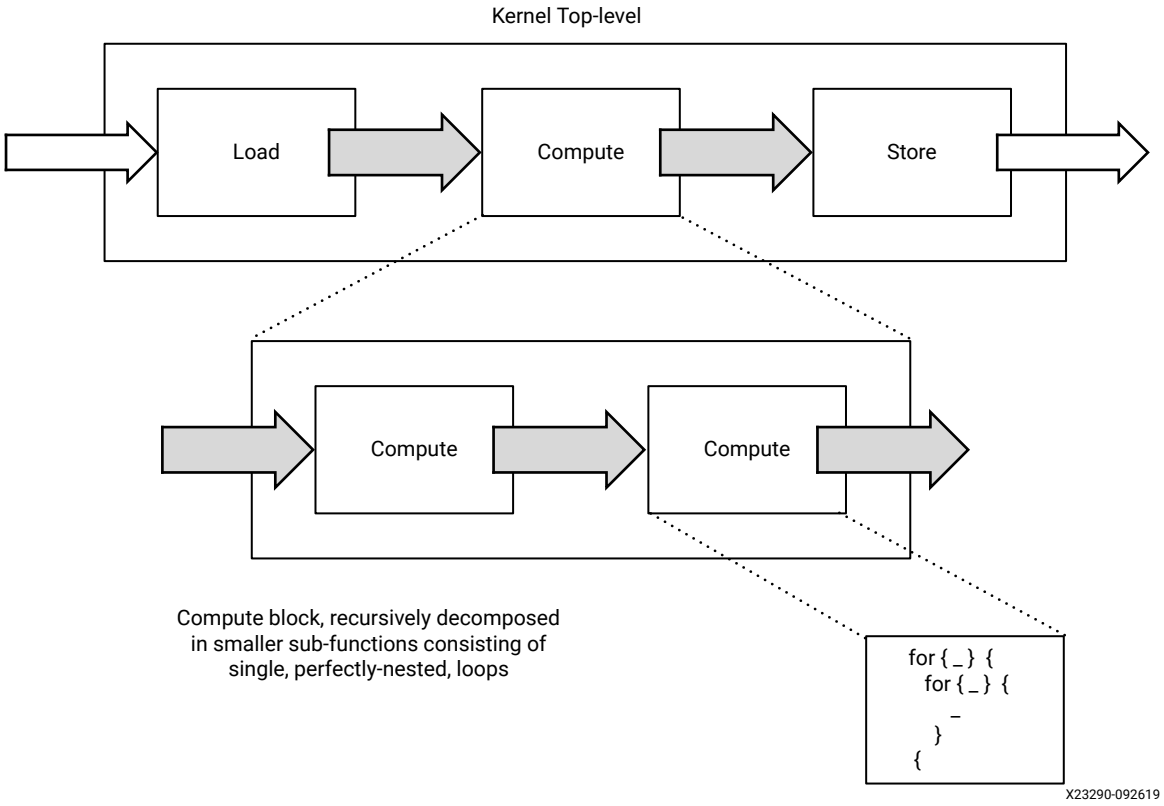
- Data must be transferred in the forward direction only, avoiding feedback whenever possible.
- Each connection must have a single producer and a single consumer.
- Only the load and store functions can access the primary interface of the kernel.

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 25: **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.

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

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 Vitis 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 Task Parallelism Using Different Kernels in the *Data Center Acceleration using Vitis* (UG1700).

During this partitioning and refining process, put sequential loops into individual functions. Ideally, the lowest-level compute block only contains a single perfectly-nested loop.

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, it must 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.

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

Both the target throughput and the volume of data consumed and produced by the function must 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 Vitis HLS compiler generates detailed reports on the throughput and latency of functions and loops. After the target latencies are determined, use the HLS reports to identify which functions and loops do not meet their latency target and require attention, as described in HLS Synthesis Report.

The latency of a loop can be calculated as follows:

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

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 Vitis 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.

Step 4: Improve Loop Latencies

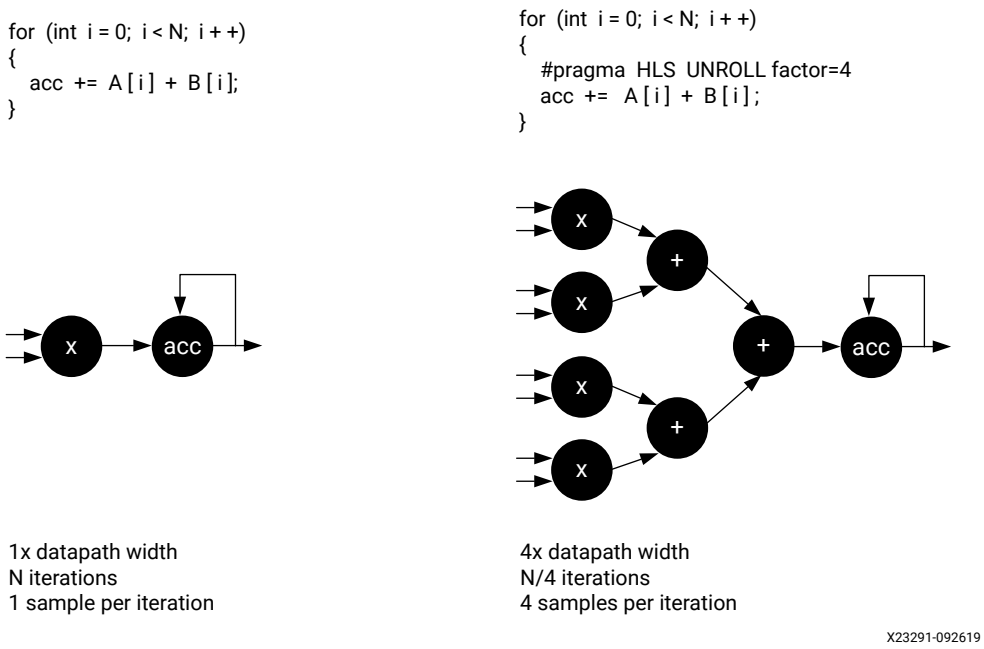
After identifying loops latencies that exceed their target, the first optimization to consider is loop unrolling.

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 conveyer belt, which results in higher performance.

Figure 26: **Loop Unrolling**



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. The interconnect can 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, ensure that the resulting datapath can access all the data it needs in parallel.

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

By default, the Vitis 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.

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 was not 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 Vitis 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 in single-port RAM.

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 Accel 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 not 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 Unrolling Loops in the *Vitis HLS User Guide* (UG1399).

- 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 might 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 must 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 can help you 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 in the *Vitis Reference Guide* (UG1702).

Best Practices for Data Center 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.

- Review the Accelerating Data Center Applications with Vitis Software Platform section for information about acceleration methodology.
- 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 AMD 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.

Tutorials and Examples

To help you quickly get started with the AMD Vitis™ core development kit, you can find tutorials, example applications, and hardware kernels in the following repositories:

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

  The Getting Started pathway tutorials are an excellent place to start as a new user.

- **Vitis Acceleration 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 C and C++, when applicable. All examples include a Makefile to enable building for hardware emulation, running on hardware, and a `README.md` file with a detailed explanation of the example.

Chapter 3

Developing Vitis Kernels and Applications

This chapter describes the fundamental elements of the Vitis development environment, creating software applications for x86 processors using the XRT API to access PL kernels, developing PL Kernels using C++, Packaging RTL Kernels.

Programming Model

The Vitis development flow enables development of FPGA-accelerated data center applications. In this environment, a software application running on an x86 processor uses the Xilinx Runtime (XRT) API to offload compute intensive tasks to execute a hardware kernel running on the programmable logic (PL) of an AMD data center accelerator card.

The elements of these FPGA-accelerated systems include the following:

- AMD Alveo™ Data Center accelerator cards
- Software applications using XRT running on x86 processors on AMD Adaptive SoC devices, as described in Writing the Software Application
- Programmable logic acceleration functions developed in RTL as described in RTL Kernel Development Flow
- Programmable logic acceleration functions as described in Developing PL Kernels using C++
- The Vitis tools to compile and link the various elements as described in Chapter 4: Building and Running the System, or Creating a System Project for Heterogeneous Computing in the *Vitis Reference Guide* (UG1702)

Design Topology

In the Vitis core development kit, Alveo Data Center accelerator cards provide a foundation for designs. These devices contain a programmable logic (PL) region that loads and executes a device binary (`.xclbin`) file that contains and connects PL kernels as compiled object (`.xo`) files.

Extensible Alveo acceleration card contain one or more interfaces to global memory (DDR or HBM), and optional streaming interfaces connected to other resources such as external I/O. PL kernels can access data through global memory interfaces (`m_axi`) or streaming interfaces (`axis`). The memory interfaces of PL kernels must be connected to memory interfaces of the extensible platform. The streaming interfaces of PL kernels can be connected to any streaming interfaces of the platform, of other PL kernels. Both memory-based and streaming connections are defined through Vitis linking options, as described in Linking the System.

Multiple kernels (`.xo`) can be implemented in the PL region of the AMD device binary (`.xclbin`), allowing for significant application acceleration. A single kernel can also be instantiated multiple times. The number of instances, or compute units of a kernel is programmable up to 31, and determined by linking options specified when building the device binary.

PL Kernel Properties

In the Vitis development flow, PL kernels as compiled object files (.xo) are the processing elements executing in the programmable logic region of the AMD device. The Vitis core development kit supports PL kernels written in C/C++ compiled by the `v++` HLS compiler, and RTL IP packaged in the Vivado Design Suite.

Regardless of source language, all PL kernels have the same properties and must adhere to same set of requirements:

- Control Scheme: Kernels can be defined as software controllable, or data-driven. This means that the PL kernel is controlled through a software application running XRT or using available drivers, or is not managed by software and is instead driven by the arrival of data.
- HW Interfaces: Kernels must use AXI4 interfaces in the design regardless of whether software controlled or data driven.
- Clocks and Resets: Kernels must have clocks to sync kernels and platforms into a system, and optional resets for additional control.

SW-Controllable Kernels

Software controllable kernels expose a programmable register interface, allowing a host software application to interact with kernels through register reads and write. These are the most common and widely applicable types of PL kernels. There are two types of SW controllable kernels: user-managed and XRT-managed.

Note: XRT-managed kernels are a specialized form of user-managed kernels.

The primary difference between user-managed and XRT-managed kernels is related to the kernel execution mode. Because XRT relies on the `ap_ctrl_chain` and `ap_ctrl_hs` execution protocols generated by the HLS compiler, XRT-managed kernels are better for C++ developers as described in Developing PL Kernels using C++ in the *Data Center Acceleration using Vitis* (UG1700). Alternatively, user-managed kernels can support many different user-defined execution protocols as found in existing Vivado RTL IP, and so are a better fit for RTL designers described in Packaging RTL Kernels.

The Vitis development flow supports software applications written for use with PL kernels using the XRT native C/C++ API, which supports both user-managed kernels and XRT-managed kernels, in addition to some advanced designs as discussed in Execution Modes. The next sections briefly describe the programming API and the different hardware interfaces required for XRT-managed or user-managed kernels.

TIP: *The general documentation for XRT can be found at Xilinx Runtime Architecture. The XRT API described below can be found at XRT Native C++ API.*

Table 1: **Software Control Using the XRT API**

XRT-Managed Kernels	User-Managed Kernels
• The object class for an XRT-managed kernel is `xrt::kernel` • The software application communicates with the XRT-managed kernel using higher-level commands such as `set_arg`, `run`, and `wait` • The user does not need to know the low-level details of the programmable registers and kernel execution protocols • Control and status registers provide XRT with a known interface to interact with the kernel, which makes these high-level commands possible • If needed, it is also possible to control an XRT-managed kernel as a user-managed kernel (using atomic register reads and write)	• The object class for a user-managed kernel is `xrt::ip` • The software application communicates with the user-managed kernel using atomic register reads and writes through the AXI4-Lite interface • The application developer is responsible for knowing the address offset and purpose of each register in the kernel, and using them properly • There are no checks, high-level controls, or profiling capabilities. The user is responsible for running the simulations for performance analysis/debugging.

Design Languages

SW controllable kernels can be developed using either RTL or C/C++:

- **RTL:** User-managed kernels are the most natural and recommended type of kernel for RTL developers. They offer greater flexibility, offer a wider range of control possibilities, and have fewer requirements than XRT-managed kernels. For more information, see Packaging RTL Kernels.
- **C++:** XRT-managed kernels are the default and recommended type of kernel for C/C++ developers as described in Developing PL Kernels using C++. The Vitis compiler (`v++`) automatically generates interfaces compatible with the high-level XRT API, leaving fewer details for the developer to worry about.

HW Interfaces

The kernel interfaces are used to exchange data with the host application, other kernels, or device I/Os. Both user-managed and XRT-managed have exactly the same interface requirements as listed here:

- **Programmable interface:** AXI4-Lite slave interface. Kernels can only have a single AXI4-Lite interface.
- **Data interfaces:** Any number and combination of AXI4 memory mapped and AXI4-Stream interfaces.
- **Clock and resets:** As described in Clock and Reset Requirements.

> **TIP:** *XRT-managed kernels have specific requirements for control registers in the AXI4-Lite interface (including start and stop bits) as described in Control Requirements for XRT-Managed Kernels. User-managed kernels can implement whatever control scheme the user specifies.*

The following table elaborates the type of interface required based on the characteristics of the data movement in your application.

Table 2: **Kernel Interface Types**

Register (AXI4-Lite)	Memory Mapped (M_AXI)	Streaming (AXI4-Stream)
• Register interfaces must be implemented using a single AXI4-Lite interface. • Designed for transferring scalars between the host application and the kernel. • Register reads and writes are initiated by the host application. • The kernel acts as a slave.	• Memory mapped interfaces must be implemented using one or more AXI4 Masters interfaces. • Designed for bi-directional data transfers with global memory (DDR, PLRAM, HBM). • Introduces additional latency for memory transfers. • The kernel acts as a master accessing data stored into global memory. • The host application allocates the buffer for the size of the dataset. • The base address of the buffer is provided by the host application to the kernel via the AXI4-Lite interface.	• Streaming interfaces must be implemented using one or more AXI4-Stream interfaces. • Designed for uni-directional data transfers between kernels. • The access pattern is sequential. • Does not use global memory. • Data set is unbounded. • A sideband signal can be used to indicate the last value in the stream.

Execution Modes

User-managed PL kernels have no predefined execution mode. It is up to the kernel designer to implement the control protocol and the execution mechanism. It is the application developer's responsibility to manage the operation of the kernel by executing appropriate sequences of register reads and writes from the software application, in accordance with the user-defined control protocol of the kernel.

XRT-managed PL kernels, as described in Supported Kernel Execution Models in the XRT documentation, provide defined kernel execution modes supporting overlapping execution of the kernel, or sequential execution.

- A kernel is started by the software application using an XRT API call. When the kernel is ready for new data it notifies the host application through bits in the control register.
- The default control protocol, `ap_ctrl_chain`, supports pipelined execution enabling multiple executions of the same PL kernel to be overlapped to improve the overall application throughput.
- If required, pipelined execution can be disabled by using the `ap_ctrl_hs` control protocol which forces kernels to run sequentially, waiting until the prior run has completed before starting the next run.
- Finally, a kernel can be auto-restarting, allowing it to run for a specified number of iterations, or until reset by the host application as described in Auto-Restarting Kernels in *Vitis High-Level Synthesis User Guide* (UG1399).

Data Driven Kernels

These kernels are present in the device but are not directly managed by the software application, instead triggered by the arrival of data at the interface. The kernels must have at least one AXI4-Stream interface. The kernel synchronizes with the rest of the system through these streaming interfaces. These kernels can have scalar inputs and outputs connected through the AXI4-Lite interface. You can read/write to the kernels by native XRT APIs (`xrt::ip::read_register`, `xrt::ip::write_register`).

Data driven kernels do not require a software API, as the host application is not required to interact directly with the kernel. The kernel can be developed as either an RTL IP or synthesized from C/C++ source code.

Kernel Interface Requirements

The kernel interfaces are used to exchange the data with the host application, other kernels, or device I/Os. Data driven kernels have the interface requirements listed here:

- **Programmable interface:** The AXI4-Lite interface is optional and used to pass scalar values to the kernel.
- **Data interfaces:** The kernel requires at least one AXI4-Stream interfaces.
- **Clock and resets:** See Clock and Reset Requirements for details.

Clock and Reset Requirements

These clock and reset requirements apply to both software controllable and non-software controllable kernels.

Table 3: **Requirements**

C/C++/OpenCL C Kernel	RTL Kernel
• C kernel does not require any input from user on clock ports and reset ports. The HLS tool always generates RTL with clock port `ap_clk` and reset port `ap_rst_n`. • HLS kernels can only have one clock/reset.	• RTL kernels require at least one clock port, but a kernel can have multiple clocks. The number of clocks that an RTL can have is primarily determined by the number of clocks that the platform supports. Most data center platforms only support two clocks. • An active-Low reset port can optionally be associated with a clock through the ASSOCIATED_RESET parameter on the clock.

Writing the Software Application

In the AMD Vitis™ environment the software application component can be written in native C++ using the Xilinx Runtime (XRT) native API. The XRT native API is described here in brief, with additional details available under XRT Native API on the XRT documentation site. The Application component is generally referred to as the host application for Data Center acceleration as it runs on an x86 server.

TIP: *For examples of software application programming using the XRT native API refer to host_xrt in the Vitis_Accel_Examples.*

In general, the structure of the host application can be divided into the following steps:

1. Specifying the platform device ID and loading the `.xclbin`
2. Setting up the PL kernel and kernel arguments
3. Transferring data between the software application and PL kernels
4. Running the system and returning results

To use the native XRT APIs, the host application must link with the `xrt_coreutil` library. For example:

```
g++ -g -std=c++17 -I$XILINX_XRT/include -L$XILINX_XRT/lib -lxrt_coreutil
-pthread
```

Compiling host code with XRT native C++ API requires C++ standard with `-std=c++17`. On GCC version older than 4.9.0, use `-std=c++1y` instead because `-std=c++17` is introduced to GCC from 4.9.0.

IMPORTANT! *For multithreading the host application, exercise caution when calling a `fork()` system call. 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.*

Specifying the Device ID and Loading the XCLBIN

To use the Xilinx Runtime (XRT) environment properly, the software application needs to identify the accelerator card, or target platform, and the device ID that the kernel will run on. Then it needs to load the device binary (`.xclbin`) into the device to program the PL kernels.

The XRT API includes a Device class (`xrt::device`) that can be used to specify the device ID on the target platform, and an XCLBIN class (`xrt::xclbin`) that defines the hardware and PL kernels for the runtime. You must use the following `#include` statements in your source code to load these classes:

```
#include <xrt/xrt_device.h>
#include <experimental/xrt_xclbin.h>
```

The following code snippet creates a device object by specifying the device ID from the target platform, and then loads the `.xclbin` into the device, returning the UUID for the program.

```
//Setup the Environment
unsigned int device_index = 0;
std::string binaryFile = parser.value("kernel.xclbin");
std::cout << "Open the device" << device_index << std::endl;
auto device = xrt::device(device_index);
std::cout << "Load the xclbin " << binaryFile << std::endl;
auto uuid = device.load_xclbin(binaryFile);
```

TIP: *The device ID can be obtained using the `xrt-smi` command for a specific accelerator card or hardware platform.*

Setting Up XRT-Managed Kernels and Kernel Arguments

After identifying devices and loading the program, the software application identifies the kernels that execute on the device, and set up the kernel arguments. All kernels the software application interacts with are defined within the loaded `.xclbin` file, and so can be identified from there.

For XRT-managed kernels, the XRT API provides a Kernel class (`xrt::kernel`), that is used to access the kernels contained within the `.xclbin` file. The kernel object identifies an XRT-managed kernel in the `.xclbin` loaded into the AMD device that can be run by the host application.

TIP: *As discussed in Working with User-Managed Kernels, use the IP class (`xrt::ip`) to identify the user-managed kernels in the `.xclbin` file.*

The use of the kernel and buffer objects require the addition of the following `include` statements in your source code:

```
#include <xrt/xrt_kernel.h>
#include <xrt/xrt_bo.h>
```

The following code example identifies a kernel (`"vadd"`) defined in the program (`uuid`) loaded onto the `device`:

```
auto krnl = xrt::kernel(device, uuid, "vadd");
```

TIP: *You can use the `xclbinutil` command to examine the contents of an existing `.xclbin` file and determine the kernels contained within.*

After identifying the kernel, or kernels to be run, you need to define buffer objects to associate with the kernel arguments and enable data transfer from the host application to the kernel instance or compute unit (CU):

```
std::cout << "Allocate Buffer in Global Memory\n";
auto bo0 = xrt::bo(device, vector_size_bytes, krnl.group_id(0));
auto bo1 = xrt::bo(device, vector_size_bytes, krnl.group_id(1));
auto bo_out = xrt::bo(device, vector_size_bytes, krnl.group_id(2));
```

The kernel object (`xrt::kernel`) includes a method to return the memory associated with each kernel argument, `kernel.group_id()`.

***Note*:** A buffer is not required for scalar arguments.

Creating Multiple Compute Units

When building the `.xclbin` file you can specify the number of kernel instances, or compute units (CU) to implement into the hardware by using the `--connectivity.nk` option as described in Creating Multiple Instances of a Kernel. After the `.xclbin` has been built, you can access specific CUs from the software application.

A single kernel object (`xrt::kernel`) can be used to execute multiple CUs as long as the CUs have identical interface connectivity, meaning the CUs have the same memory connections (`krnl.group_id`). If all CUs do not have the same kernel connectivity, then you can create a separate kernel object for each unique configuration of the kernel, as shown in the example below.

```
krnl1 = xrt::kernel(device, xclbin_uuid, "vadd:{vadd_1,vadd_2}");
krnl2 = xrt::kernel(device, xclbin_uuid, "vadd:{vadd_3}");
```

In the example above, `krnl1` can be used to launch the CUs `vadd_1` and `vadd_2` which have matching connectivity, and `krnl2` can be used to launch `vadd_3`, which has different connectivity.

IMPORTANT! *If you create a single kernel object for multiple CUs without matching connectivity, then XRT assigns one or more CUs with matching connectivity to the kernel object, and ignores the other CUs in the hardware when executing the kernel.*

Transferring Data between Software and PL Kernels

Transferring data to and from the memory in the accelerator card or device uses the buffer objects (`xrt::bo`) created when Setting Up XRT-Managed Kernels and Kernel Arguments.

The class constructor typically allocates a regular 4K aligned buffer object. The following code creates regular buffer objects that have a software application backing pointer allocated by user space in heap memory, and a device-side buffer allocated in the memory bank associated with the kernel argument (`krnl.group_id`). Optional flags in the `xrt::bo` constructor let you create non-standard types of buffers for use in special circumstances as described in Creating Special Buffers.

```
std::cout << "Allocate Buffer in Global Memory\n";
auto bo0 = xrt::bo(device, vector_size_bytes, krnl.group_id(0));
auto bo1 = xrt::bo(device, vector_size_bytes, krnl.group_id(1));
auto bo_out = xrt::bo(device, vector_size_bytes, krnl.group_id(2));
```

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

With the buffer established, and filled with data, there are a number of methods to enable transfers between the software application and the kernel, as described below:

- **Using `xrt::bo::sync()`:** Use `xrt::bo::sync` to sync data from the host to the device with `XCL_BO_SYNC_TO_DEVICE` flag, or from the device to the host with `XCL_BO_SYNC_FROM_DEVICE` flag using `xrt::bo::write`, or `xrt::bo::read` to write the buffer from the host application, or read the buffer from the device.

  ```
  bo0.write(buff_data);
  bo0.sync(XCL_BO_SYNC_BO_TO_DEVICE);
  bo1.write(buff_data);
  bo1.sync(XCL_BO_SYNC_BO_TO_DEVICE);
  ...
  bo_out.sync(XCL_BO_SYNC_BO_FROM_DEVICE);
  bo_out.read(buff_data);
  ```

  Note: If the buffer is created using a user-pointer as described in Creating Buffers from User Pointers, the `xrt::bo::sync` call is sufficient, and the `xrt::bo::write` or `xrt::bo::read` commands are not required.

- **Using `xrt::bo::map()`:** This method maps the host-side buffer backing pointer to a user pointer.

  ```
  // Map the contents of the buffer object into host memory
  auto bo0_map = bo0.map<int*>();
  auto bo1_map = bo1.map<int*>();
  auto bo_out_map = bo_out.map<int*>();
  ```

The software code can subsequently exercise the user pointer for data reads and writes. However, after writing to the mapped pointer (or before reading from the mapped pointer), use the `xrt::bo::sync()` command with the required direction flag for the DMA operation.

```
for (int i = 0; i < DATA_SIZE; ++i) {
   bo0_map[i] = i;
   bo1_map[i] = i;
}

// Synchronize buffer content with device side
bo0.sync(XCL_BO_SYNC_BO_TO_DEVICE);
bo1.sync(XCL_BO_SYNC_BO_TO_DEVICE);
```

There are additional buffer types and transfer scenarios supported by the XRT native API, as described in Miscellaneous Other Buffers.

Working with XRT-Managed Kernels

The execution of a PL kernel is associated with a class called `xrt::run` that implements methods to start and wait for kernel execution. Most interaction with kernel objects are accomplished through `xrt::run` objects, created from a kernel to represent an execution of the kernel.

The run object can be explicitly constructed from a kernel object, or implicitly constructed by starting a kernel execution as shown below.

```
std::cout << "Execution of the kernel\n";
auto run = krnl(bo0, bo1, bo_out, DATA_SIZE);
run.wait();
```

The above code example demonstrates launching the kernel execution using the `xrt::kernel()` operator with the list of arguments for the kernel that returns an `xrt::run` object. This is an asynchronous operator that returns after starting the run. The `xrt::run::wait()` member function is used to block the current thread until the run is complete.

TIP: *Upon finishing the kernel execution, the `xrt::run` object can be used to relaunch the same kernel function if desired.*

An alternative approach to run the kernel is shown in the code below:

```
auto run = xrt::run(krnl);
run.set_arg(0,bo0); // Arguments are specified starting from 0
run.set_arg(0,bo1);
run.set_arg(0,bo_out);
run.start();
run.wait();
```

In this example, the run object is explicitly constructed from the kernel object, the kernel arguments are specified with `run.set_args()`, and the run execution is launched by the `run.start()` command. Finally, the current thread is blocked as it waits for the kernel to finish.

After the kernel has completed its execution, you can sync the kernel results back to the host application using code similar to the following example:

```
// Get the output;
bo_out.sync(XCL_BO_SYNC_BO_FROM_DEVICE);

// Validate our results
if (std::memcmp(bo_out_map, bufReference, DATA_SIZE))
   throw std::runtime_error("Value read back does not match reference");
```

Working with User-Managed Kernels

TIP: *For an example of a software application working with user-managed RTL kernel, refer to the following:*

- Vitis Tutorials
- rtl_user_managed example in the Vitis-Tutorials on GitHub.

User-managed kernels require the use of the XRT native API for the software application, and are specified as an IP object of the `xrt::ip` class. The following is a high-level overview of how to structure your host application to access user-managed kernels from an `.xclbin` file.

IMPORTANT! *XRT has a 16-bit (64K) address width limitation for the `s_axilite` interface.*

1. Add the following header files to include the XRT native API:

   ```
   #include "experimental/xrt_ip.h"
   #include "xrt/xrt_bo.h"
   ```

   - `experimental/xrt_ip.h`: Defines the IP as an object of `xrt::ip`.
   - `xrt/xrt_bo.h`: Lets you create buffer objects in the XRT native API.
2. Set up the application environment as described in Specifying the Device ID and Loading the XCLBIN.
3. The IP object (`xrt::ip`) is constructed from the `xrt::device` object, the `uuid` of the loaded `.xclbin`, and the `name` of the user-managed kernel:

   ```
   //User Managed Kernel = IP
   auto ip = xrt::ip(device, uuid, "Vadd_A_B");
   ```
4. Create buffers for the IP arguments:

   ```
   auto <buf_name> = xrt::bo(<device>,<DATA_SIZE>,<flag>,<bank_id>);
   ```

   Where the buffer object constructor uses the following fields:

- `<device>`: `xrt::device` object of the accelerator card.
- `<DATA_SIZE>`: Size of the buffer as defined by the width and quantity of data.
- `<flag>`: Flag for creating the buffer objects.
- `<bank_id>`: Defines the memory bank on the device where the buffer is allocated for IP access. The memory bank specified must match with the corresponding IP port's connection inside the `.xclbin` file. Otherwise you will get `bad_alloc` when running the application. You can specify the assignment of the kernel argument using the `--connectivity.sp` command as explained in Mapping Kernel Ports to Memory.

For example:

```
auto buf_in_a = xrt::bo(device,DATA_SIZE,xrt::bo::flags::normal,0);
auto buf_SIZE,xrt::bo::flags::normal,0);
```

TIP: *Verify the IP connectivity to determine the specific memory bank, or you can get this information from the Vitis generated `.xclbin.info` file.*

For example, the following information for a user-managed kernel from the `.xclbin` could guide the construction of buffer objects in your host code:

```
Instance:        Vadd_A_B_1
   Base Address: 0x1c00000

   Argument:          scalar00
   Register Offset:   0x10
   Port:              s_axi_control
   Memory:            <not applicable>

   Argument:          A
   Register Offset:   0x18
   Port:              m00_axi
   Memory:            bank0 (MEM_DDR4)

   Argument:          B
   Register Offset:   0x24
   Port:              m01_axi
   Memory:            bank0 (MEM_DDR4)
```

5. Get the buffer addresses and transfer data between host and device:

```
    auto a_data = buf_in_a.map<int*>();
    auto b_data = buf_in_b.map<int*>();

    // Get the buffer physical address
    long long a_addr=buf_in_a.address();
    long long b_addr=buf_in_b.address();

    // Sync Buffers
    buf_in_a.sync(XCL_BO_SYNC_BO_TO_DEVICE);
    buf_in_b.sync(XCL_BO_SYNC_BO_TO_DEVICE);
```

`xrt::bo::map()` allows mapping the host-side buffer backing pointer to a user pointer. However, before reading from the mapped pointer or after writing to the mapped pointer, use `xrt::bo::sync()` with direction flag for the DMA operation.

6. After preparing the buffer (buffer create, sync operation as shown above), you are free to pass all the necessary information to the IP with the direct register write operation.

   IMPORTANT! *The `xrt::ip` differs from the standard `xrt::kernel`, and indicates that XRT does not manage the IP but does provide access to read or write the registers.*

   For example, the code below shows the information passing the buffer base address through the `xrt::ip::write_register()` command.

```
ip.write_register(REG_OFFSET_A,a_addr);
ip.write_register(REG_OFFSET_A+4,a_addr>>32);

ip.write_register(REG_OFFSET_B,b_addr);
ip.write_register(REG_OFFSET_B+4,b_addr>>32);
```

7. Start the IP execution. Because the IP is user-managed, you can employ any number of register write/read to control the start/check status/restart the IP to trigger the execution of the IP. The following example uses an `s_axilite` interface to access control signals in the control register:

```
uint32_t axi_ctrl = 0;
std::cout << "INFO:IP Start" << std::endl;
axi_ctrl = IP_START;
ip.write_register(CSR_OFFSET, axi_ctrl);

// Wait until the IP is DONE
axi_ctrl =0;
while((axi_ctrl & IP_IDLE) != IP_IDLE) {
    axi_ctrl = ip.read_register(CSR_OFFSET);
}
```

8. After IP execution is finished, you can transfer the data back to host by the `xrt::bo::sync` command with the appropriate flag to dictate the buffer transfer direction.

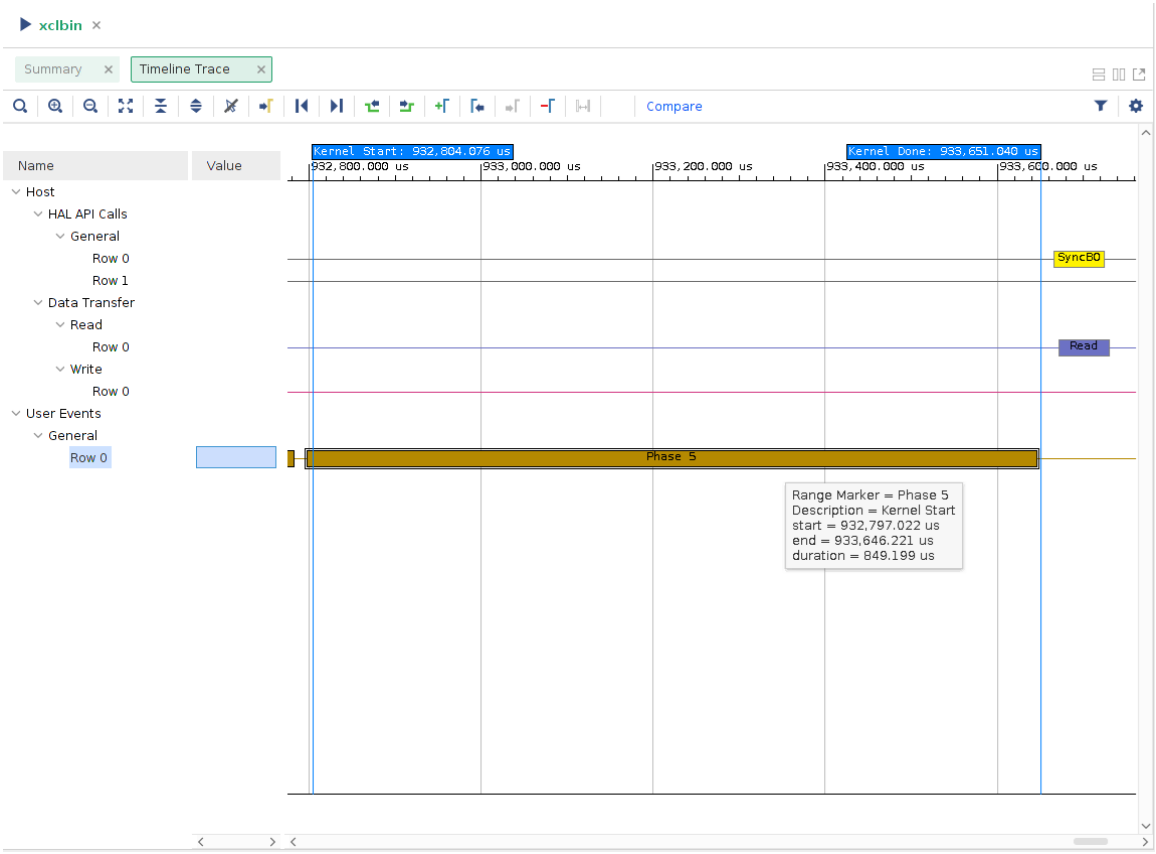
```
buf_in_b.sync(XCL_BO_SYNC_BO_FROM_DEVICE);
```

9. Optionally profile the application.

   Because XRT is not in charge of starting or stopping the kernel, you cannot directly profile the operation of `user_managed` kernels as you would XRT managed kernels. However, you can use the `user_range` and `user_event` objects as discussed in Custom Profiling of the Host Application to profile elements of the host application. For example the following code captures the time it takes to write the registers from the host application:

```
// Write Registers
range.start("Phase 4a", "Write A Register");
ip.write_register(REG_OFFSET_A,a_addr);
ip.write_register(REG_OFFSET_A+4,a_addr>>32);
range.end();
range.start("Phase 4b", "Write B Register");
ip.write_register(REG_OFFSET_B,b_addr);
ip.write_register(REG_OFFSET_B+4,b_addr>>32);
range.end()
```

You can observe some aspects of the application and kernel operation in the Vitis analyzer as shown in the following figure.



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. In the Vitis core development kit, one or more kernels are separately compiled/linked to build the `.xclbin` file. The `device.load_xclbin(binaryFile)` command is used to load the kernel binary.
2. Create `xrt::kernel` objects from the loaded device binary, and associate buffer objects (`xrt::bo`) with the memory banks assigned to kernel arguments.
3. Transfer data back and forth from the host application to the kernel using `xrt::bo::sync` commands and buffer reads and write commands.
4. Execute the kernel using an `xrt::run` object to start the kernel and wait for kernel execution.
5. Additionally, you can add error checking after XRT API calls for debugging purpose, if required.

Developing PL Kernels using C++

The PL kernel code is generally a compute-intensive part of the system intended to run on the programmable logic (PL) region of an AMD device. The Vitis development environment supports PL kernels written in C or C++, and also written in RTL.

For C/C++ based kernels the HLS compiler (`v++`) is available as a standalone command, or as part of the Vitis unified IDE. The IDE provides a language sensitive editor, simulation and code analysis tools, high-level synthesis of RTL from the C or C++ code, and C-RTL co-simulation for an in-depth examination of the resulting hardware design. The HLS compiler is the recommended tool for developing PL kernels for use in the AMD Vivado™ traditional design flow, or in the Vitis heterogeneous design flow.

Generally, off-the-shelf software cannot be efficiently converted into hardware running on programmable logic. Even if the software program can be automatically converted (or synthesized) into hardware, achieving acceptable quality of results (QoR) will require additional work such as structuring the algorithm to help the HLS compiler achieve the desired performance goals. To help, you need to understand the best practices for writing good software for execution on programmable logic as discussed in Design Principles in the *Vitis High-Level Synthesis User Guide* (UG1399).

The code for C++ kernels written and optimized in an HLS component can be compiled in the Vitis tools either from the `v++` command line as explained in Compiling C/C++ PL Kernels, or from a System project in the Vitis unified IDE as explained in Using the Vitis Unified IDE in the *Vitis Reference Guide* (UG1702).

IMPORTANT! *The kernel function declaration must be wrapped with the `extern "C"` linkage in the header file, or the whole function in the kernel source code must be wrapped.*

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

Packaging RTL Kernels

In the AMD Vitis™ development flow, RTL IP from the Vivado Design Suite can be packaged as compiled AMD object (`.xo`) files) that can be linked into a device binary (`.xclbin`), as long as they adhere to Vivado IP Packaging guidelines, and requirements of the Vitis compiler for linking the system.

As explained in PL Kernel Properties, RTL kernels can be user-managed kernels that do not meet XRT requirements for execution control, but rather implement any number of possible control schemes specified by existing RTL designs. Alternatively, RTL kernels can adhere to the requirements of the `ap_ctrl_chain` or `ap_ctrl_hs` control protocols needed for XRT-managed kernels.

RTL kernels support the hardware emulation build, and the hardware build described in Selecting the Build Target.

The following sections describe the kernel interface requirements for the Vitis compiler to link kernels into a system. These requirements are common to software controllable and non-software controlled kernels. The control requirements for XRT-managed kernels are also described, in addition to any additional requirements. Finally, the development flow is described to help you package RTL IP in theVivado Design Suite as RTL kernels for use in the Vitis environment.

Requirements of an RTL Kernel

To be integrated into the Vitis tool flow, an RTL module must minimally meet the requirements enumerated in Kernel Interface Requirements. The need to meet the kernel interface requirements applies to both XRT-managed and user-managed kernels.

In addition, XRT-managed kernels must satisfy the requirements described in Control Requirements for XRT-Managed Kernels to be executed and profiled by XRT.

User-managed kernels must have the signal interfaces needed by the Vitis compiler to allow it to link the kernels to other kernels and to the target platform, but do not need to adhere to the strict execution protocol of XRT. In this way, existing RTL IP can be more rapidly and simply integrated into the Vitis environment.

It might be necessary to revise your RTL module to meet the kernel requirements outlined in the following sections.

Kernel Interface Requirements

To enable the Vitis compiler to connect kernels into the target platform, an RTL kernel must adhere to the requirements described in PL Kernel Properties. The various interface requirements are summarized in the following table.

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

Table 4: **RTL Kernel Interface and Port Requirements**

Port or Interface	Description	Comment
Clock	One or more clock inputs.	• At least one clock is required for the kernel.[1] • Can be named anything, but must be packaged with a bus interface. **IMPORTANT!** *All ports in the RTL IP must be associated with an interface when packaging the RTL for use in the Vitis environment. If this is not the case, an error similar to the following occurs:* `ERROR: UNDEF When packaging for Vitis, pins that are not part of an interface are not supported`
Reset	Primary active-Low reset input port	• Optional port. • Can be named anything, but must be associated with a Clock signal through the ASSOCIATED_RESET property on the Clock. • This signal must be internally pipelined to improve timing. • The signal is driven by a synchronous reset in the associated Clock domain.
interrupt	Active-High interrupt.	• Optional port. • When used, the name must be exactly as shown.
s_axi_control	One (and only one) AXI4-Lite slave control interface	• Required port. The `s_axilite` interface is generally required with exception for some cases using AXI4-Stream interfaces. It is not required for non-software controlled kernels. • When used, the name must be exactly as shown, and is case-sensitive. ***Note***: The address range of the `s_axilite` interface can be edited in the `kernel.xml` file and repackaged using the `package_xo` command if needed. However, XRT imposes a 64K (16 bit) address range limitation. The tool will return an error if the `s_axilite` interface is greater than 16 bits wide.
AXI4_Memory Mapped Interface (m_axi)	AXI4 memory mapped interfaces for global memory access	• Optional port. • All AXI4 memory mapped 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 provided by the SW applications to the kernel through a register in the AXI4-Lite interface. • AXI4 memory mapped must not use Wrap or Fixed burst types and must not use narrow (sub-size) bursts. This means that AxSIZE must 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.

Table 4: **RTL Kernel Interface and Port Requirements** *(cont'd)*

Port or Interface	Description	Comment
AXI4_STREAM (axis)	AXI4-Stream interfaces for one-way data transfers between kernels or between the host application and kernels.	• Optional port. • Cannot be used with bi-directional ports. • Use the STREAM interface template in the Vivado Design Suite. • Refer to AXI4-Stream Interfaces for additional information on interface requirements in the *Vitis High-Level Synthesis User Guide* (UG1399).

Notes:

1. The clock requirements listed here are for newer platform shells which include fixed clocks as discussed in Managing Clock Frequencies. RTL kernels for use on legacy platforms support two clocks named `ap_clk` and `ap_clk_2` specifically, and two optional resets named `ap_rst_n` and `ap_rst_n_2`.

Control Requirements for XRT-Managed Kernels

IMPORTANT! *User-managed kernels do not require the control registers and signals described below, but they can implement a control structure using registers in an `s_axilite` interface as discussed in Creating User-Managed RTL Kernels. If your RTL module implements a different control structure, you can define it as a `user_managed` kernel, or it must be adapted to conform to the XRT-managed requirements described here.*

The following table outlines the required register map for an XRT-managed kernel to be used within the Vitis tools and XRT. The control register is required by kernels that specify `ap_ctrl_hs` and `ap_ctrl_chain` control protocols as described in Execution Modes. Kernels that implement `ap_ctrl_none` and `user_managed` control protocols do not require the control registers described below.

TIP: *The interrupt related registers are only required for designs that implement interrupts.*

All user-defined registers must begin at location `0x10`; locations below this are reserved. These include registers for kernel arguments such as scalar values and address offsets passed to memory mapped interfaces.

Table 5: **Register Address Map**

Offset	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 signals 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 example.

The following table shows the control signals that are accessed through the control register (`offset 0x0`). The control register and its signals are determined by the kernel execution mode, `ap_ctrl_hs` and `ap_ctrl_chain`.

The available signals are used by different control protocols as explained in Supported Kernel Execution Models. For example, for the sequential execution mode `ap_ctrl_hs`, the host typically writes `0x00000001` to the offset 0 control register which sets Bit 0, clears Bits 1 and 2, and polls on reading `ap_done` signal until it is a 1.

Table 6: **Control Register Signals**

Bit	Name	Description
0	ap_start	Asserted when the kernel can start processing data. Cleared on handshake with ap_done being asserted.
1	ap_done	Asserted when the kernel has completed operation. Cleared on read.
2	ap_idle	Asserted when the kernel is idle.
3	ap_ready	Asserted by the kernel when it is ready to accept the new data
4	ap_continue	Asserted by the XRT to allow kernel keep running
7	auto_restart	Used to enable automatic kernel restart as described in the chapter Auto-Restarting Kernels in *Vitis High-Level Synthesis User Guide* (UG1399).
31:5	Reserved	Reserved

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

Table 7: **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 8: **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 9: **IP Interrupt Status (0xC)**

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

Interrupt

XRT-managed 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 in the Control Register block.

The Vitis compiler (`v++`) will link the interrupt signal of a PL kernel into the available signals on the platform, provided the platform has interrupts available for connection as described in Adding Hardware Interfaces in the *Embedded Design Development Using Vitis* (UG1701). If no interrupt is enabled on the platform, then you must manually connect the interrupt of the kernel.

By default, the IER uses the internal `ap_done` signal to trigger an interrupt. Further, the interrupt is cleared only when writing a 1 to bit-0 of the IP Interrupt Status Register.

This logic must be reflected in the Verilog code for the RTL kernel, and also in the associated `component.xml` and `kernel.xml` files. The `kernel.xml` file is stored inside the `kernel.xo` file and is generated automatically when using the `package_xo` command or RTL Kernel Wizard.

IMPORTANT! *The XRT native API does not support triggering or catching interrupts in the host application for user-managed RTL kernels.*

Creating User-Managed RTL Kernels

If your RTL IP does not satisfy the AXI interface requirements for the Vitis compiler as outlined in Kernel Interface Requirements in the *Embedded Design Development Using Vitis* (UG1701), you must modify the IP to implement the required interfaces. However, if your RTL IP does not satisfy the XRT control protocols of `ap_ctrl_hs` or `ap_ctrl_chain`, you can define it as a user-managed kernel rather than having to rewrite your IP.

A user-managed kernel does not need to satisfy the control requirements of XRT, and can implement any of a variety of execution mechanisms. User-managed kernels are meant to let you take advantage of the system building capabilities of the Vitis compiler, while letting your kernel implement your own control scheme. There is no prescribed method of starting or stopping, or otherwise controlling your kernel. This is largely up to you, and the specific requirements of your application or system. Some of the available control schemes include:

- Accessing registers through an `s_axilite` control interface, similar to the method used by XRT though open to your own implementation
- Accessing the hardware through software drivers, such as UIO drivers, implemented in your host application
- Triggering the start or stop response of your kernel from a signal provided by a separate component, or from another kernel

- Providing a data-driven approach, as described in the topic Auto-Restarting Mode in the *Vitis High-Level Synthesis User Guide* (UG1399).

> **IMPORTANT!** *One limitation of implementing a control register in an `s_axilite` interface for a user-managed kernel is that the control register cannot be named CTRL. That name is specifically reserved for XRT-managed kernels, and returns a Critical Warning when found on a user-managed kernel, or `ap_ctrl_none` kernel.*

RTL Kernel Development Flow

This section explains the process of creating RTL kernels using the Package IP feature inside the Vivado Design Suite. The Package IP command provides a Package for Vitis option which greatly simplifies packaging an existing RTL IP as a compiled object (.xo) file for use in the Vitis environment.

The packaged XO file is a container encapsulating the Vivado IP object (including source files) and associated kernel XML file. Using the Vitis compiler, the XO file can be combined with other kernels, and linked with the target platform and built for hardware or hardware emulation flows.

The Package for Vitis feature provides DRCs to check the completeness of the packaged IP prior to generating the XO file, and also automates the `package_xo` command to simplify the production of the packaged RTL kernel.

Packaging the RTL Code as a Vitis XO

> **IMPORTANT!** *The RTL IP must first be thoroughly verified with traditional RTL verification methods before being packaged as a kernel.*

As discussed in Kernel Interface Requirements, the RTL kernel must be packaged with the following required interfaces:

- The AXI4-Lite interface name must be packaged as `S_AXI_CONTROL`, but the underlying AXI ports can be named differently.
- Any memory-mapped AXI4 interfaces must be packaged as AXI4 master endpoints with 64-bit address support.

  > **RECOMMENDED:** *AMD 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.*

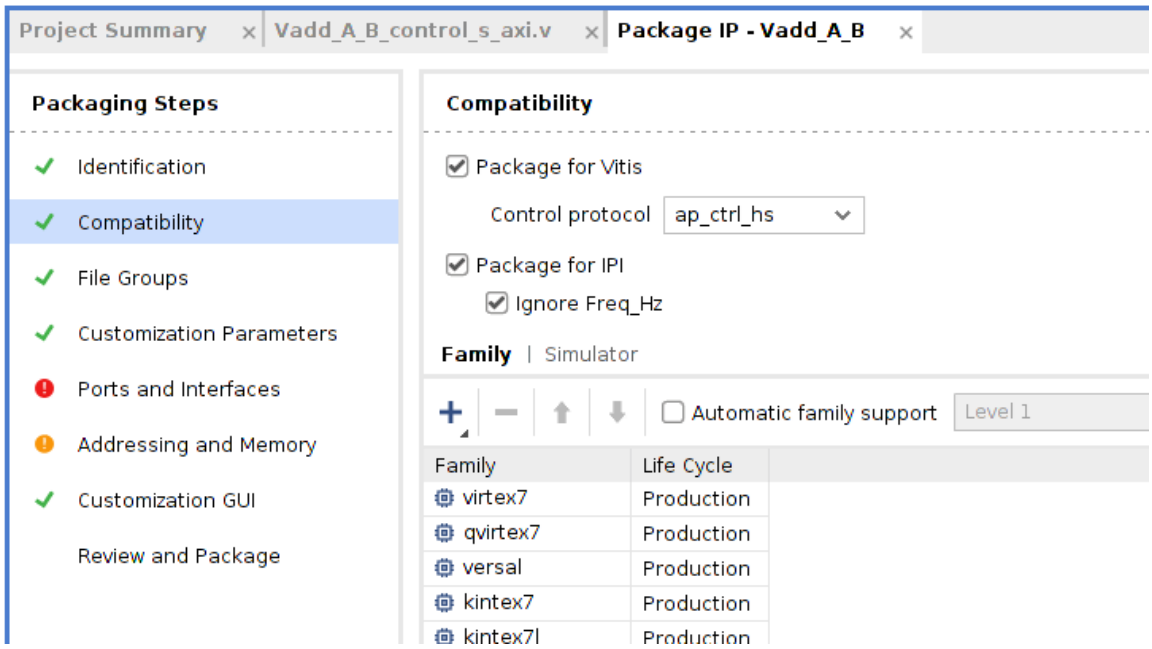
- You can also implement the AXI4-Stream interface.
- At least one clock is required for the kernel, though it can support multiple clocks.
  - Each clock must have an associated Bus Interface identifying it as a clock.

- Each clock can have an optional active-Low reset, specified by the ASSOCIATED_RESET property on the clock.
- A clock must be associated with each AXI4-Lite, AXI4, and AXI4-Stream interface on the kernel.

To package the IP, use the following steps:

1. Create and package a new IP.
   a. From a Vivado project, with your RTL source files added, select **Tools → Create and Package New IP**.
   b. Select **Package your current project**, and click **Next**.
   c. Specify the location for your packaged IP. You can select the default location, or choose a different location.
   d. Review the Summary page and click **Finish** to open the Package IP window.

   The Package IP window opens to display the Identification page. For details on working with the IP packager in the Vivado tool, refer to the *Vivado Design Suite User Guide: Creating and Packaging Custom IP* (UG1118).
2. Select **Compatibility** under the Packaging Steps. This displays the Compatibility view as shown in the following figure.



   a. Select the **Package for Vitis** check box to enable the process of packaging the RTL IP as an XO for use in the Vitis environment.
   b. Select the **Control Protocol** for the RTL Kernel. This determines the control mechanism used to operate the kernel. The choices are:

- `user_managed`: Defines a SW-controllable kernel, that is user-managed rather than XRT-managed. This is the preferred option. Refer to Creating User-Managed RTL Kernels for additional information.
- `ap_ctrl_hs`: This is the default, and specifies the simple sequential execution model for XRT -managed kernels as described in SW-Controllable Kernels.
- `ap_ctrl_chain`: Specifies a pipelined execution model for XRT-managed kernels.
- `ap_ctrl_none`: Indicates no control protocol as described in Data Driven Kernels.

c. Check to ensure that both **Package for IPI** and **Ignore Freq_Hz** are enabled as well.

Enabling these check boxes enables design rule checks (DRC) that the `ipx::check_integrity` command runs prior to packaging the IP and generating the XO. The DRCs include checks for required signals as described in Requirements of an RTL Kernel, and checks for control protocols and registers for XRT-managed kernels. As shown in the preceding figure, any issues are reported to the Package IP tool as they are encountered.

3. Associate the clock to the AXI interfaces.

Select the **Ports and Interfaces** step of the Package IP window, you can associate the primary kernel clock with the AXI4 interfaces, and reset signal if needed.

a. Right-click an AXI4 interface, and select **Associate Clocks**.

This opens the Associate Clocks dialog box which lists any identified clock signals.

b. Select the appropriate clock and click **OK** to associate it with the interface.

c. Ensure to repeat this step to a clock signal with each of the AXI interfaces.

4. Click the **Addressing and Memory** step to add control registers and offsets.

XRT-managed kernels using the `ap_ctrl_hs` or `ap_ctrl_chain` control protocol require control registers as discussed in Control Requirements for XRT-Managed Kernels. The following table shows a list of the required registers.

TIP: *While `ap_ctrl_none` and `user_managed` control protocols do not require control registers, they can still use them if an `s_axilite` interface is included as part of the RTL design. In this case, the specific registers can differ from the following table, but the process of assigning `names`, `offsets`, and `widths` is the same.*

Table 10: **Address Map**

Register Name	Description	Address Offset	Size
CTRL	Control Signals as described in Control Requirements for XRT-Managed Kernels.	0x000	32
GIER	Global Interrupt Enable Register. Used to enable interrupt to the host.	0x004	32
IP_IER	IP Interrupt Enable Register. Used to control which IP generated signal are used to generate an interrupt.	0x008	32

Table 10: **Address Map** *(cont'd)*

Register Name	Description	Address Offset	Size
IP_ISR	IP Interrupt Status Register. Provides interrupt status.	0x00C	32
<kernel_args>	This includes a separate entry for each kernel argument as needed on the software function interface. All user-defined registers must begin at location `0x10`; locations below this are reserved.	0x010	32/64 Scalar arguments are 32-bits wide.`m_axi` and `axis` interfaces are 64 bits wide.

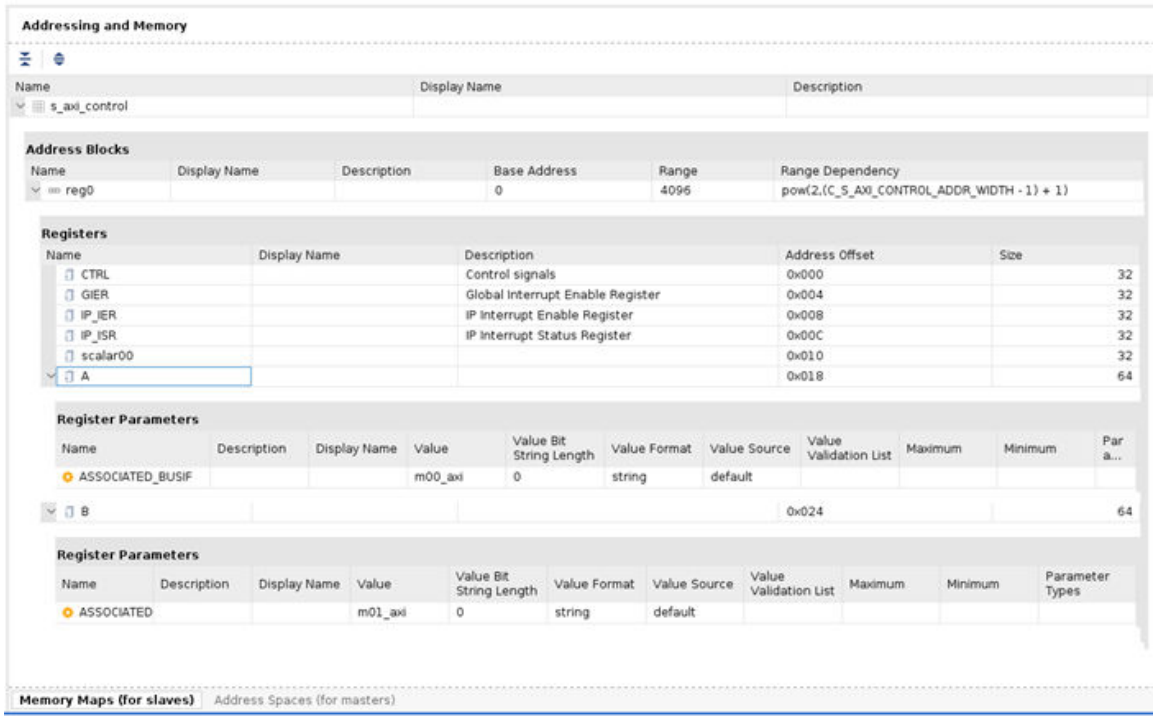
a. To create the address map described in the table, right-click in the **Address Blocks** and select the **Add Register** command.

   This opens the Add Register view in which you can enter one of the register names from the table above.

> **IMPORTANT!** *The **Range** value under Address Blocks specifies the address range for the `s_axilite` interface. You can modify this value to change the range for the kernel.*

b. Repeat as needed to add all required registers.

   This creates a Registers table in the Addressing and Memory section. You can edit the table to add the Description, Address Offset, and Size to each register. The Registers table needs to look similar to the following example.



> **TIP:** *The Tcl commands for each step of this process are written to the Tcl Console. You can use this fact to execute the process, and then use the Tcl transcript to create scripts to automate the process for future iterations.*

c. Finally, select the register for each of the pointer arguments from your table, right-click and select the **Add Register Parameter** command. Enter the name `ASSOCIATED_BUSIF` into the dialog box that opens, and click **OK**.

   This lets you define an association between the register and the AXI4 Interface. In the value field of the added parameter, enter the name of the `m_axi` interface assigned to the specific argument you are defining. In the example above, the argument `A` uses the `m00_axi` interface, and the argument `B` uses the `m01_axi` interface.

5. At this point you are ready to package your IP.

   a. Select the **Review and Package** section of the Package IP view, review the Summary and After Packaging sections, and make whatever changes are needed.

   > **IMPORTANT!** *You must enable the generation of an IP archive file. If the After Packaging section indicates An archive will not be generated, you must select the* ***Edit packaging settings*** *link and enable the* ***Create archive of IP*** *setting.*

   b. When you are ready, click **Package IP**.

      The Vivado tool packages your kernel IP, automatically runs the `package_xo` command as needed to produce the XO file, and opens a dialog box to inform you of success.

      The generated XO file for the RTL kernel can be used by the Vitis compiler during the linking process to connect to other HLS or RTL kernels, and for linking with the target platform to complete the system. Refer to Chapter 4: Building and Running the System for more information.

   c. If your RTL kernel has some custom features that are not standard for the `package_xo` command that is run automatically, you can run the command manually to regenerate the XO file and kernel with custom settings. Refer to package_xo Command in the *Vitis Reference Guide* (UG1702) for details of the command. Some specific reasons why you might need to manually run the `package_xo` command include:

      - Specify a different IP directory or XO path
      - Output a copy of the `kernel.xml` file to modify it and repackage

6. Optional: Test the Packaged IP.

   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 shows 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 view, 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.

7. Optional: Configure Design Constraints.

If the RTL kernel has design constraints (`.xdc`) which refer to elements of the static region of the platform, such as clocks, then the 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 for late processing:

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

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

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

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

Design Recommendations for RTL Kernels

Design RTL kernels with recommendations from the *UltraFast Design Methodology Guide for FPGAs and SoCs* (UG949). In addition to adhering to the interface and packaging requirements, design the kernels with the following performance goals in mind:

- Memory Performance Optimizations for AXI4 Interface
- 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.

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 crosses between them must be flip-flop to flip-flop timing paths.

Debug and Verification Considerations

- Verify RTL kernels 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.
- You can add ILA inside of RTL kernels as described in Adding Debug IP to RTL Kernels.

- Use hardware emulation to test the host code software integration or to view the interaction between multiple kernels.

Chapter 4

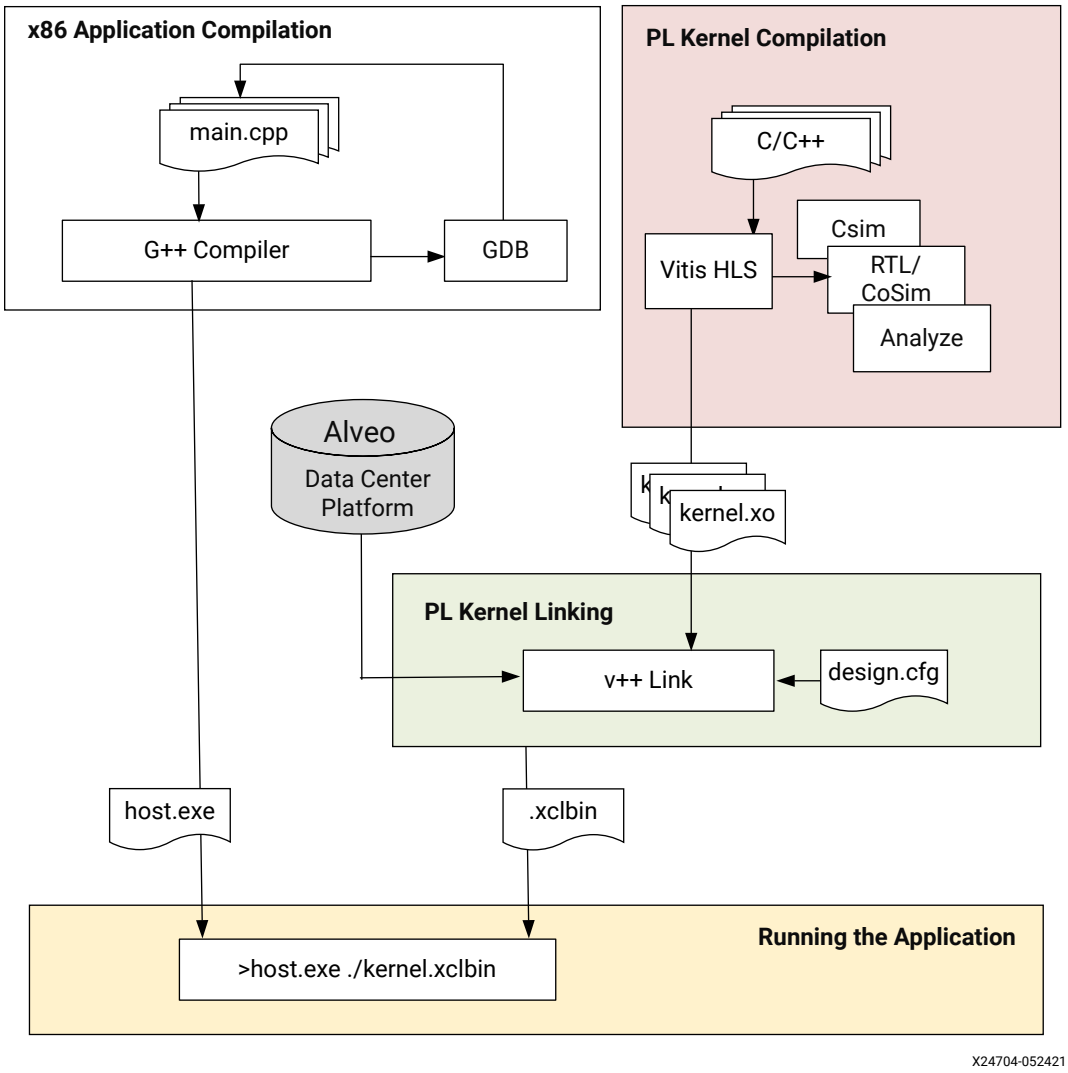
Building and Running the System

After the software program and the kernel code is written, you can build the application, which includes compiling the software program, and linking the platform with the PL kernel file to create the device binary (`.xclbin`). The build process follows a standard compilation and linking process for both the software program and the kernel code. After building, both the host program and the FPGA binary, you will be ready to run the application.

The following figure shows the high-level steps required to use the Vitis tools flow to integrate your application. The command-line process to run this flow is described here.

***Note*:** You can also use this flow from within the Vitis IDE as explained in Launching Vitis Unified IDE in the *Vitis Reference Guide* (UG1702).

Figure 27: **Vitis Development Flow**



Selecting the Build Target

The first step when building the application is to specify a build target. The build target of the AMD Vitis™ tool defines the nature and contents of the FPGA binary (`.xclbin`) created during compilation and linking. There are two different build targets: one emulation target used for validation and debugging purposes: hardware emulation, and the default system hardware target used to generate the FPGA binary (`.xclbin`) loaded into the AMD device.

Compiling the 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 11: **Comparison of Emulation Flows with Hardware Execution**

Hardware Emulation	Hardware Execution
Host application runs with a simulated RTL model of the kernels. SystemC models and external TGs are also supported.	Host application runs with actual hardware implementation of the kernels.
Test the host / kernel integration, get performance estimates.	Confirm that the system runs correctly and with desired performance.
Best debug capabilities, moderate compilation time with increased visibility of the kernels.	Final FPGA implementation, long build time with accurate (actual) performance results.

Hardware Emulation Target

Hardware emulation runs an RTL simulation of the programmable logic design, where the PL kernels are integrated with a cycle-approximate model of the hardware platform.

Hardware emulation is especially useful for the following tasks:

- Checking the functional correctness of the RTL code synthesized from the C, C++ kernel code
- Testing the interactions between different kernels or multiple CUs
- Using hardware waveforms to gain detailed visibility into internal activity of the kernels
- Getting initial performance estimates for the application
- C/++ or Python traffic generators can be used to inject traffic in the simulation

Each kernel is compiled to a hardware model (RTL). During hardware emulation, kernels are run in the logic simulator, with a waveform viewer to examine the kernel design. Some third-party simulators are also supported as described in Simulator Support in Hardware Emulation. In addition, hardware emulation provides performance and resource estimates for the hardware implementation.

Hardware emulation provides a detailed, cycle-accurate, view of kernel activity. AMD recommends using small data sets for validation during hardware emulation to keep runtimes 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 are intended to be used for guidance and comparing relative performance between different kernel implementations.*

As discussed in v++ Command in the *Vitis Reference Guide* (UG1702), the hardware emulation target is specified in the `v++` command with the `-t` option:

```
v++ -t hw_emu ...
```

Hardware Execution Target

When the build target is the hardware, `v++` builds the FPGA binary for the AMD device by running Vivado synthesis and implementation on the design. 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 and the application can be run in its actual operating environment.

As discussed in v++ Command in the *Vitis Reference Guide* (UG1702), the hardware execution is specified in the `v++` command with the `-t` option:

```
v++ -t hw ...
```

Building the Software Application

The software application, written in C/C++ using the XRT native API, 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 an x86 processor.

To use the native XRT APIs, the host application must link with the `xrt_coreutil` library. For example:

```
g++ -g -std=c++17 -I$XILINX_XRT/include -L$XILINX_XRT/lib -lxrt_coreutil
-pthread
```

Compiling host code with XRT native C++ API requires C++ standard with `-std=c++17`. On GCC version older than 4.9.0, use `-std=c++1y` instead because `-std=c++17` is introduced to GCC from 4.9.0.

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: *Set up the command shell or window as described in Setting Up the Vitis Environment in the Data Center Acceleration using Vitis (UG1700) prior to running the tools.*

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. Each source file of the host application is compiled into an object file (`.o`) using the `g++` compiler. The generated object files (`.o`) are linked with the Xilinx Runtime (XRT) shared library to create the executable host program using the `g++ -l` option.

The host application can be written in native C++ using the Xilinx Runtime (XRT) native C++ API. The required include files and libraries depend on the API your host application uses, and any specific requirements of your host code.

To use the native XRT API, the host application must link with the `xrt_coreutil` library. The command line uses a few different settings as shown in the following example, which combines compilation and linking:

```
$CXX -std=c++17 -O0 -g -Wall -c -I./src -o host.o sw/host.cpp
```

When compiling the source code using XRT native API, the following `g++` options are required:

- `-std=c++17`: Define the C++ language standard. Compiling host code with XRT native C++ API requires C++ standard with `-std=c++17` or newer. However, on GCC versions older than 4.9.0, use `-std=c++1y` instead.
- `-I$XILINX_XRT/include/`: XRT include directory

When linking the executable, the following `g++` options are required:

- `-L$XILINX_XRT/lib/`: Look in XRT library.
- `-lxrt_coreutil`: Search the named library during linking.
- `-pthread`: Search the named library during linking.

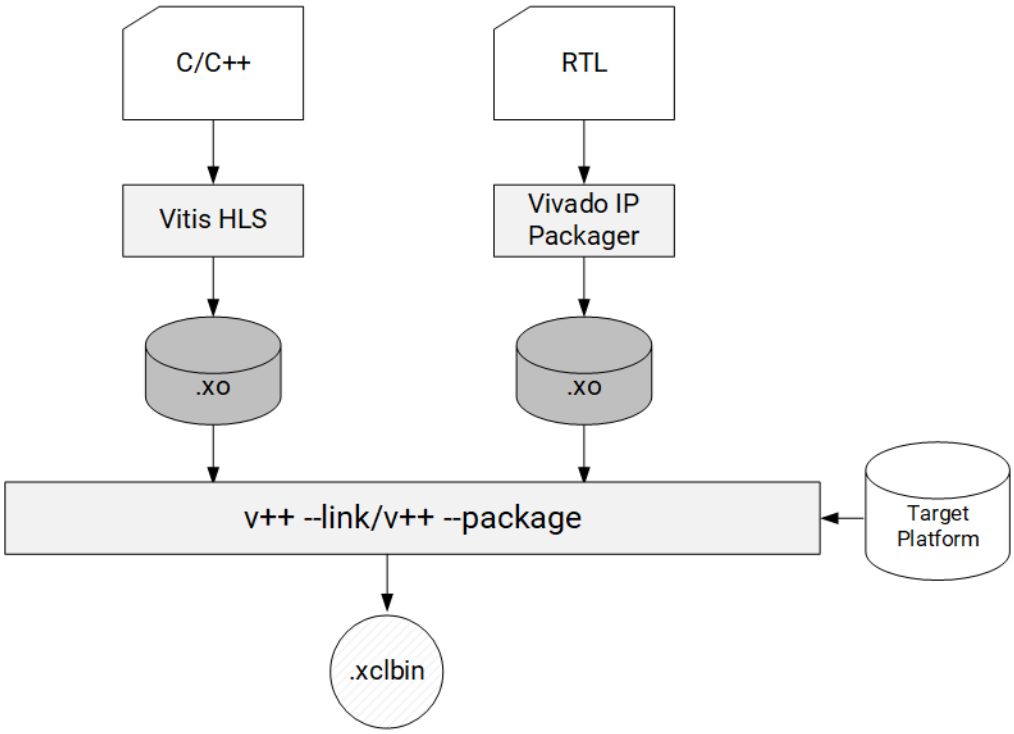
Command to Link the Host Application Against Required XRT APIs and Generate the Executable:

```
g++ *.o -lxrt_coreutil -L${XILINX_XRT}/lib -o $(EXECUTABLE)
```

Building the Device Binary

The Device Binary (`.xclbin` file) contains the bitstream and other metadata needed to program the FPGA accelerator card. It it built by compiling the acceleration kernels and linking them with the extensible platform.

Figure 28: **Device Build Process**



The process, as outlined above, has two steps:

1. Compile the acceleration kernels from source code.
   - For C, C++ kernels, the `v++ -c --mode hls` command compiles the source code into object (XO) files. Multiple kernels are compiled into separate XO files.
   - For RTL kernels, the Vivado IP packager produces the XO file to be used for linking. Refer to Packaging RTL Kernels for more information.
2. After compilation, the `v++ -l` command links one or multiple kernel objects (XO), together with the hardware platform, to produce the Xilinx binary `.xclbin` file.

The following sections describe building the different components of the system design to produce the device binary.

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 Vitis unified IDE as discussed in Using the Vitis Unified IDE in the Vitis Reference Guide (UG1702).*

Compiling C/C++ PL Kernels

IMPORTANT! *Set up the command shell or window as described in Setting Up the Vitis Environment in the Data Center Acceleration using Vitis (UG1700) prior to running the tools.*

The techniques of programming, simulating, and synthesizing the PL kernel objects are described in the *Vitis High-Level Synthesis User Guide* (UG1399). The kernel object file (`.xo`) can be compiled by the HLS compiler (`v++ -c --mode hls`) as described below.

To compile the PL kernel use the following command line as an example:

```
v++ -c --mode hls --config ./src/hls_config.cfg --work_dir vadd
```

The various arguments used are described below:

- `-c`: Specifies the compilation mode of the `v++` command. This can be specified using `-c` or `--compile`
- `--mode hls`: Launches the HLS compiler form of the Vitis compiler
- `--config <config_filename>`: Specify a Configuration file for use with the HLS compiler. The configuration file has HLS compiler options as described in v++ Mode HLS in the *Vitis Reference Guide* (UG1702). For additional details, refer to Creating an HLS Component in the *Vitis High-Level Synthesis User Guide* (UG1399). The Configuration file will specify:
  - The target platform or part for the build. The platform specified in the `v++ --link` command must match the target platform or part used for the HLS compile command
  - The source C++ file or files for the PL kernel
  - The Configuration file will also specify an output name if one is needed. The default output name is the same as the top-level function with the `.xo` extension
- `--work_dir`: Specifies the location of the HLS component created by the `v++` compiler

TIP: *The HLS compiler mode does not require a target. The compiled object file (.xo) is suitable for both the hardware emulation build and the hardware build.*

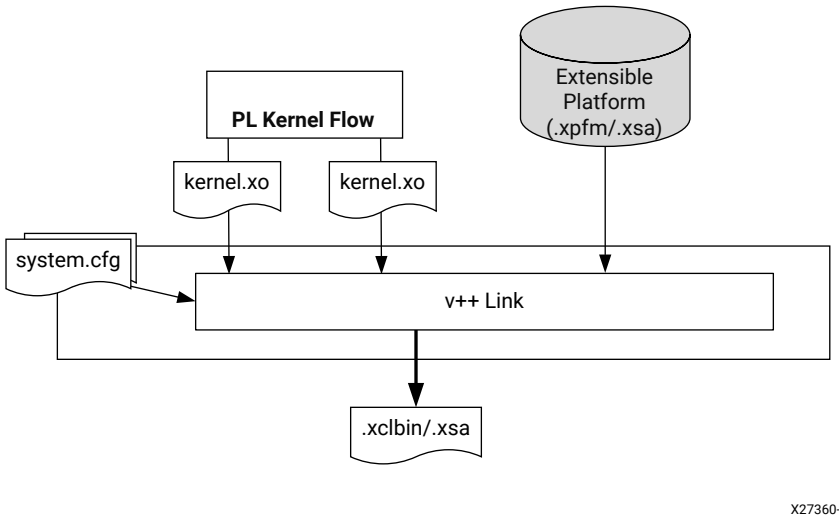
Refer to Output Directories of the v++ Command in the *Vitis Reference Guide* (UG1702) to get an understanding of the location of various output files generated by the HLS compiler.

After the compilation step is complete, any reports generated during this process are collected into the `<kernel_name>.compile_summary`. This collection of reports can be viewed by opening the compile_summary in the Analysis view of Vitis unified IDE, and includes a Summary report, Kernel Estimate for timing and resource estimates, Kernel Guidance offering any suggestions for compilation, and the HLS Synthesis log. Refer to Working with the Analysis View (Vitis Analyzer) in the *Vitis Reference Guide* (UG1702) for additional information.

Linking the System

TIP: *Set up the command shell or window as described in Setting Up the Vitis Environment in the Data Center Acceleration using Vitis (UG1700) prior to running the tools.*

Figure 29: **Linking the System Design**



During the linking stage, one or more PL kernel objects (.xo files) are linked with the extensible platform to create the FPGA binary container file (`.xclbin`).

The linking stage is when the developer specifies the main characteristics of the accelerated system, including setting up the command shell or window as described in Setting Up the Vitis Environment in the *Data Center Acceleration using Vitis* (UG1700) prior to running the tools.

- Which accelerators are to be included in the device
- How many instances of each accelerator are needed
- How are accelerators connected to system memory
- How are accelerators connected between each other
- Where are accelerators physically placed in the FPGA

The following is an example command line to link the `vadd` kernel (`.xo`) with a `libadf.a` graph archive and a Versal adaptive SoC platform, specifying the `.xsa` file as the output:

```
v++ -t hw_emu --platform xilinx_vck190_base_202420_1 --link vadd.xo
libadf.a \
--config ./system.cfg -o binary_container.xsa
```

This command contains the following arguments:

- **`-t <arg>`:** Specifies the build target. When linking, you must use the same `-t` and `--platform` arguments specified when compiling the PL kernels.

- `--platform <arg>`: Specifies the platform to link with the system design. In the example command above the `custom_vck190` platform is a custom platform designed to work with the `--export_archive` command.
- `--link`: Link the kernels, graph, and platform into a system design.
- `<input>.xo`: Specifies the input PL kernel object files (`.xo`) to link with the target platform. This is a positional parameter.
- `--config ./system.cfg`: Specify a configuration file that is used to provide `v++` command options for a variety of uses. Refer to Vitis Compiler Configuration File in the *Vitis Reference Guide* (UG1702) for more information on the `--config` option.

After the linking step is complete, any reports generated during this process are collected into the `<kernel_name>.link_summary`. This collection of reports can be viewed by opening the link_summary in the Analysis view of Vitis analyzer, and includes a Summary report, System Estimate providing timing and resources estimates, System Guidance offering any suggestions for improving linking and the performance of the system. Refer to Working with the Analysis View (Vitis Analyzer) in the *Vitis Reference Guide* (UG1702) for additional information.

TIP: *Refer to Output Directories of the v++ Command in the Vitis Reference Guide (UG1702) to get an understanding of the location of various output files.*

The linking process defines important architectural details of the system design. In particular, this is where the design is enabled for profiling or debug, where you specify the number of compute unit (CUs) to instantiate into hardware, where CUs are assigned to SLRs, and where you define connections from PL kernel ports to global memory. The following sections discuss some of these build options.

Enabling Profile and Debug when Linking

To capture and visualize profiling and trace information, or to enable your design for debugging, you will need to add specific commands during the `v++` linking phase, and sometimes during `v++` compilation. The tool must instrument the profile IP using --profile Options in the `v++` linking phase and enable the profiling during the runtime. To enable debugging the application you can specify one of the --debug Options.

During `v++` linking, the application developer needs to add profile IP to the design to capture the profiling data and preferably choose the memory resources for storing and offloading data during the runtime.

- You can add PL monitors to capture tracing information on their design by using `--profile` command. This adds the logic to capture profile data for data traffic between the kernel and host, kernel stalls, the execution times of kernels, and compute units (CUs). The instrumentation can be added using options, `--profile.data`, `--profile.stall`, and `--profile.exec`, as described in --profile Options.

- You also can specify the choice of memory resources or FIFO in the PL to store captured data. On large designs that span multiple SLRs, the tracing infrastructure can cause timing issues as there is one offload point and all trace data must cross SLRs to reach it. For these use cases, multiple memory resources can be used for offloading the trace data.

To enable the capture of profile data or trace information during the application runtime, you can choose from a variety of options to add to the xrt.ini File, which configures the runtime. See Profiling the Application in the *Data Center Acceleration using Vitis* (UG1700) for more information.

Creating Multiple Instances of a Kernel

By default, the linker builds a single hardware instance from a kernel. However, you can instantiate multiple hardware compute units (CUs) from a single kernel. This can improve performance as the host program can now make multiple calls to a given kernel, and see them executed in parallel on the different 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 v++ Command in the *Vitis Reference Guide* (UG1702).

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. The delimiter between kernel instances is a comma. In the example above, the `kernel_name` and the `number` of CUs are specified, but not the `cu_name`. In this case vadd_1 and vadd_2 will be added to the design.

Then the config file is specified on the `v++` command line:

```
v++ --config vadd_config.cfg ...
```

TIP: *You can check the results by using the `xclbinutil` command to examine the contents of the `xclbin` file. Refer to xclbinutil Utility in the Vitis Reference Guide (UG1702).*

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 Memory

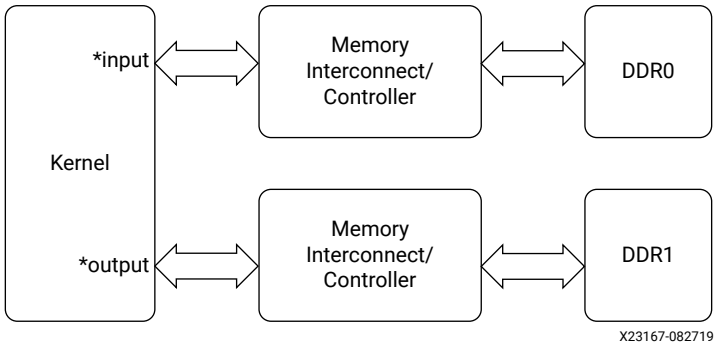
The link phase is when the memory ports of the kernels are connected to memory resources such as DDR, HBM, and PLRAM.

By default, all kernel memory interfaces are connected to the same global memory bank (or `gmem`). As a result, only one kernel interface can transfer data to or from the memory bank at one time, limiting the performance of the application due to memory access.

Because of this, it is important to explicitly specify which global memory bank each kernel argument (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 arguments to different global memory banks can improve performance by enabling simultaneous accesses to input and output data.

The following block diagram illustrates the Using Multiple DDR Banks example in Vitis-Tutorials 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 30: **Global Memory Two Banks Example**



IMPORTANT! *Up to 15 separate kernel interfaces can be connected to a given global memory bank. Therefore, if there are more than 15 memory interfaces in the design you must explicitly perform the memory mapping as described here, using the `--conectivity.sp` option to distribute connections across different memory banks.*

Start by assigning the kernel arguments to separate bundles to increase the available interface ports, then assign the arguments to separate memory banks. The following example uses the interfaces described in HW Interfaces in the *Data Center Acceleration using Vitis* (UG1700).

1. In the C/C++ kernel code assign arguments to separate bundles using the INTERFACE pragma prior to compiling them:

```
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 this example, the `cnn` kernel has 3 arguments: `pixel`, `weights` and `out`. Using the bundle attribute of the INTERFACE pragma, each argument is mapped to a specific interface. The `pixel` and `out` arguments are both mapped to the same interface called `gmem`. The weights argument is mapped to a different interface called `gmem1`. The resulting kernel has therefore 2 distinct interfaces (`gmem` and `gmem1`) which can be connected to different 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. Use the `--connectivity.sp` option, or include it in a config file, as described in --connectivity Options.

   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>.<argument>:<bank name>
sp=cnn_1.pixel:DDR[0]
sp=cnn_1.weights:DDR[1]
sp=cnn_1.out:DDR[0]
#sp=<aie_instance>.<gmio_port>:<memory_sp_tag_name>[bank_number]
sp=ai_engine_0.my_gmio:LPDDR[0]
```

   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.
   - `<argument>` is the name of the kernel argument. Alternatively, you can specify the name of the kernel interface as defined by the HLS INTERFACE pragma for C/C++ kernels, including `m_axi_` and the `bundle` name. In the `cnn` kernel above, the ports would be `m_axi_gmem` and `m_axi_gmem1`.

     TIP: *For RTL kernels, the interface is specified by the interface name defined in the `kernel.xml` file.*

   - `<bank_name>` is denoted as `DDR[0]`, `DDR[1]`, `DDR[2]`, and `DDR[3]` for a platform with four DDR banks. You can also specify the memory as a contiguous range of banks, such as DDR[0:2], in which case XRT will assign the memory bank at runtime.

Some platforms also provide support for LPDDR, PLRAM, HBM, HP or MIG memory, in which case you would use LPDDR[0], PLRAM[0], HBM[0], HP[0] or MIG[0]. You can use the `platforminfo` utility to get information on the global memory banks available in a specified platform. Refer to platforminfo Utility in the *Vitis Reference Guide* (UG1702) for more information.

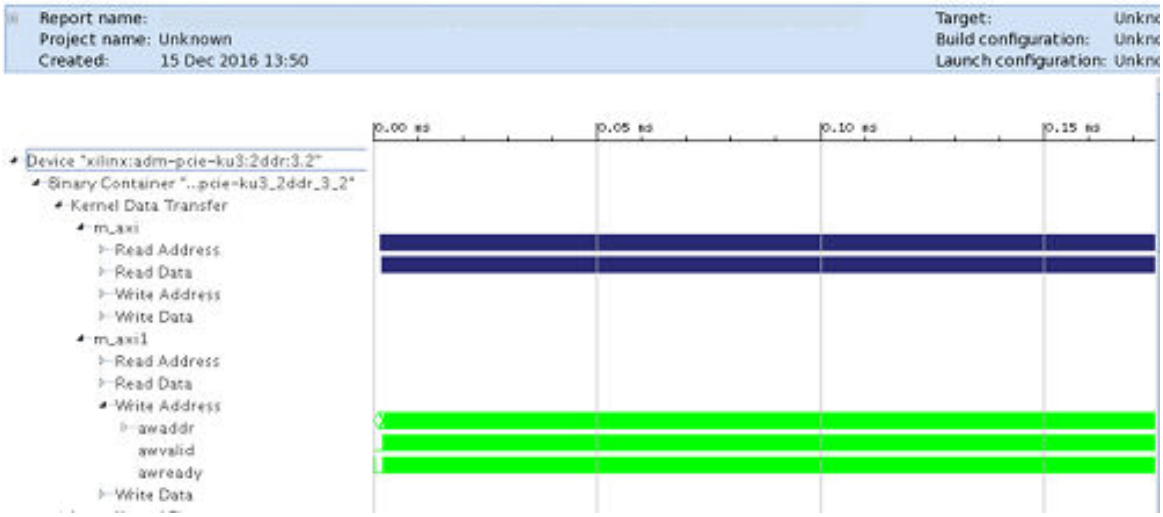
In platforms that include both DDR and HBM memory banks, kernels must use separate AXI interfaces to access the different memories. DDR and PLRAM access can be shared from a single port.

Note: The SP tag name is declared in Vivado block design as part of the platform properties. The SP tag names can be customized by user to help distinguish specific memory controllers.

TIP: *Assigning kernel interfaces to specific memory banks might also require you to specify the SLR to place the kernel into. For more information, refer to Assigning Compute Units to SLRs on Alveo Accelerator Cards in the Data Center Acceleration using Vitis (UG1700).*

You can use the Device Hardware Transaction view in Vitis Analyzer to observe the actual DDR Bank communication, and to analyze DDR usage.

Figure 31: **Device Hardware Transaction View Transactions on DDR Bank**



Additional Memory Mapping Techniques for Alveo Accelerator Cards

Alveo accelerator cards support additional features such as host-memory access, HBM, or PLRAM utilization, depending on the specific card you are working with. The following sections describe some of these additional techniques.

Assigning AXI Interfaces to PLRAM

For platforms that support PLRAM use the `--connectivity.sp` option as previously described in Mapping Kernel Ports to Memory, but use the name `PLRAM[id]` to specify the bank. Valid PLRAM banks supported by a platform can be found in the Memory Information section reported by the `platforminfo` command.

For example, in the Alveo U250 platform the following PLRAM information is reported:

```
Bus SP Tag: PLRAM
  Segment Index: 0
    Consumption: explicit
    SLR:         SLR0
    Max Masters: 60
  Segment Index: 1
    Consumption: explicit
    SLR:         SLR1
    Max Masters: 60
  Segment Index: 2
    Consumption: explicit
    SLR:         SLR2
    Max Masters: 60
  Segment Index: 3
    Consumption: explicit
    SLR:         SLR3
    Max Masters: 60
```

To access the PLRAM you would use the following command for example:

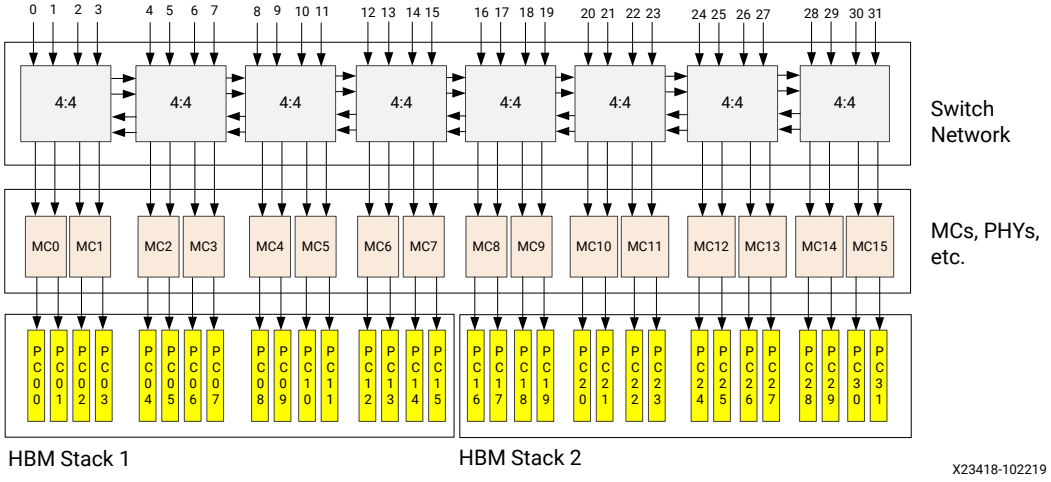
```
--connectivity.sp cnn_1.weights:PLRAM3
```

TIP: *To reduce latency and improve performance, the kernel accessing this PLRAM must be placed into SLR3 in addition to assigning compute units to SLRs. For more information, see Assigning Compute Units to SLRs on Alveo Accelerator Cards in the Data Center Acceleration using Vitis (UG1700).*

HBM Configuration and Use

Some algorithms are memory bound, limited by the 77 Gb/s bandwidth available on DDR-based Alveo cards. For those applications there are High Bandwidth Memory (HBM) based Alveo cards, providing up to 460 Gb/s memory bandwidth. For the Alveo implementation, two 16-layer HBM (HBM 2 specification) stacks are incorporated into the FPGA package and connected into the FPGA fabric with an interposer. A high-level diagram of the two HBM stacks is as follows.

Figure 32: **High-Level Diagram of Two HBM Stacks**



This implementation provides:

- 16 GB HBM using 512 MB pseudo channels (PCs) for Alveo U55C accelerator cards, as described in *Alveo U55C Data Center Accelerator Cards Data Sheet* (DS978)
- 8 GB total HBM using 256 MB PCs for Alveo U280 accelerator cards, and U50 cards as described in *Alveo U50 Data Center Accelerator Cards Data Sheet* (DS965)
- An independent AXI channel for communication between the Vitis kernels and the HBM PCs through a segmented crossbar switch network
- A two-channel memory controller for addressing two PCs
- 14.375 Gb/s max theoretical bandwidth per PC
- 460 Gb/s (32 × 14.375 Gb/s) max theoretical bandwidth for the HBM subsystem

Although each PC has a theoretical maximum performance of 14.375 Gb/s, this is less than the theoretical maximum of 19.25 Gb/s for a DDR channel. To get better than DDR performance, designs must efficiently integrate multiple AXI masters into the HBM subsystem. The programmable logic has 32 HBM AXI interfaces that can access any memory location in any of the PCs on either of the HBM stacks through a built-in switch network providing full access to the memory space.

Connection to the HBM is managed by the HBM Memory Subsystem (HMSS) IP, which enables all HBM PCs, and automatically connects the XDMA to the HBM for host access to global memory. When used with the Vitis compiler, the HMSS is automatically customized to activate only the necessary memory controllers and ports as specified by the `--connectivity.sp` option to connect both the user kernels and the XDMA to those memory controllers for optimal bandwidth and latency.

***Note*:** Because of the complexity and flexibility of the built-in switch network, there are many implementations that result in congestion at a particular memory location or in the switch network. Interleaved read and write transactions cause a drop in efficiency with respect to read-only or write-only due to memory controller timing parameters (bus turnaround). Write transactions that span both HBM stacks will also experience degraded performance, and must be avoided. It is important to plan memory accesses so that kernels access limited memory where possible, and configure kernel connectivity to isolate the memory accesses for different kernels into different HBM PCs.

The `--connectivity.sp` option to connect kernels to HBM PCs is:

```
sp=<compute_unit_name>.<argument>:<HBM_PC>
```

In the following config file example, the kernel input ports `in1` and `in2` are connected to HBM PCs 0 and 1 respectively, and the output buffer `out` is connected to HBM PCs 3–4

```
[connectivity]
sp=krnl.in1:HBM[0]
sp=krnl.in2:HBM[1]
sp=krnl.out:HBM[3:4]
```

Each HBM PC is 256 MB, giving a total of 1 GB of memory access for this kernel. Refer to the Using HBM Tutorial for additional information and examples.

TIP: *The config file specifies the mapping of a kernel argument to one or more HBM PCs. When mapping to multiple PCs each AXI interface must only access a contiguous subset of the available 32 HBM PCs. For example, HBM[3:7].*

When implementing the connection between the kernel argument and the specified PC, the HMSS automatically selects the appropriate channel in the switch network to connect the AXI Slave interface port to access memory, maximize bandwidth, and minimize latency given the pseudo channel number or range. However, the `--connectivity.sp` syntax for HBM also lets you specify which channel index for the switch network the HMSS must use for connecting the kernel interface. The `--connectivity.sp` syntax for specifying the switch network index is:

```
sp=<compute_unit_name>.<argument>:<bank_name>.<index>
```

When specifying the switch index, only one index can be specified per `sp` option. You cannot reuse an index which has already been used by another `sp` option or line in the config file.

In this case, the last line of the previous example could be rewritten as: `sp=krnl.out:HBM[3:4].3` to use switch channel 3 (`S_AXI03`) or `sp=krnl.out:HBM[3:4].4` to use switch channel 4 (`S_AXI04`), as shown in the figure above.

Either `sp` option would route the kernel's data transactions through one of the left-most network switch blocks to reduce implementation complexity. Using any index in the range 0 to 7 would use one of the two left-most switch blocks in the network, Using any other index would force the use of additional switch blocks, which adds routing complexity and could negatively impact performance depending on the application.

The HBM ports are located in the bottom SLR of the device. The HMSS automatically handles the placement and timing complexities of AXI interfaces crossing super logic regions (SLR) in SSI technology devices. By default, without specifying the `--connectivity.sp` or `--connectivity.slr` options on `v++`, all kernel AXI interfaces access HBM[0] and all kernels are assigned to SLR0.

However, you can specify the SLR assignments of kernels using the `--connectivity.slr` option. For devices or platforms that use multiple SLRs, you are strongly recommended to define CU assignments to specific SLRs. Refer to Assigning Compute Units to SLRs on Alveo Accelerator Cards in the *Data Center Acceleration using Vitis* (UG1700) for more information.

Random Access and the RAMA IP

HBM performs well in applications where sequential data access is required. However, for applications requiring random data access, performance can vary significantly depending on the application requirements (for example, the ratio of read and write operations, minimum transaction size, and size of the memory space being addressed). In these cases, the addition of the Random Access Memory Attachment (RAMA) IP to the target platform can significantly improve random memory access efficiency in cases where the required memory exceeds the 256 MB limit of a single HBM PC.

The RAMA IP will improve random access performance when two or more HBM PC are used. Refer to *RAMA LogiCORE IP Product Guide* (PG310) for more information.

TIP: *To effectively use the RAMA IP in your application the kernel must access memory from multiple HBM PCs and must use a static single ID on the AXI transaction ID ports (AxID), or slowly changing (pseudo-static) AXI transaction IDs. If these conditions are not met, the thread creation used in the RAMA IP to improve performance has little effect, and consumes programmable logic resources for no purpose.*

To use the RAMA IP add the keyword `RAMA` to the `sp` option in the config file, with the following format.

***Note*:** The `--connectivity.sp` option requires the use of the <index> as described in the prior section.

```
sp=<compute_unit_name>.<argument>:<bank_name>.<index>.RAMA
```

For example:

```
sp=krnl.out:HBM[3:4].3.RAMA
```

Directly Accessing Host Memory on Alveo Accelerator Cards

The PCIe® Slave-Bridge IP is provided on some data center platforms to let kernels access directly to host memory. Only certain Alveo cards support the host memory connection, as reported in the Memory Information section returned by the `platforminfo` command.

For example, the following information is returned for the Alveo U250 card:

```
   Name: HOST[0]
   Index: 8
   Type: MEM_DRAM
   Base Address: 0x2000000000
   Address Size: 0x400000000
   Bank Used: No
```

Configuring the device binary to connect directly to host memory uses the `--connectivity.sp` command below.

```
[connectivity]
## Syntax
##sp=<cu_name>.<interface_name>:HOST[0]
sp=cnn_1.weights:HOST[0]
```

IMPORTANT! *Using host memory also requires changes to the accelerator card setup and your host application as described at Host-Memory Access in the XRT documentation.*

Specifying Streaming Connections

Support for hardware accelerator pipelines that communicate through streams is one of the major advantages of FPGAs, FPGA-based SoCs, and Versal adaptive SoC devices; it can be used in DSP and image processing applications, in addition to communication systems. Kernel ports involved in streaming are defined within the kernel, and are not addressed by the host program. There is no need to send data back to global memory before it is forwarded to another kernel for processing. The connections between the kernels are directly defined during the `v++` linking process as described below.

A streaming data output port of one kernel can be connected to the streaming data input port of another kernel, or between a PL kernel and the PLIO of an ADF graph application, during linking using the `--connectivity.sc` option. This option can be specified at the command line, or from a `config` file that is specified using the `--config` option, as described in v++ Command in the *Vitis Reference Guide* (UG1702).

IMPORTANT! *An error occurs if the `--connectivity.sc` kernel drives itself.*

To connect the streaming output port of a producer kernel to the streaming input port of a consumer kernel, set up 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>:
[<fifo_depth>]
stream_connect=vadd_1.stream_out:vadd_2.stream_in
stream_connect=vadd_2.stream_in:ai_engine_0.DataIn0
```

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. The `cu_name` can be specified in the config file as described in Creating Multiple Instances of a Kernel, or is defined automatically by the tool when not otherwise specified.
- `<output_port>` or `<input_port>` is the streaming port defined in the producer or consumer kernel.

  IMPORTANT! *If the port-width of the output and input ports do not match, the Vitis compiler automatically inserts a data-width converter between the two ports as part of the build process. The inclusion of the data-width converter is either truncate a larger bit-width output to a smaller bit-width input, or expand a smaller bit-width to a larger bit-width.*

- `[:<fifo_depth>]` inserts a FIFO of the specified depth between the two streaming ports to prevent stalls. The value is specified as an integer.

Assigning Compute Units to SLRs on Alveo Accelerator Cards

Alveo Data Center accelerator cards use stacked silicon devices consisting of multiple Super Logic Regions (SLR) to provide device resources, including global memory. 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. For the best performance, assign kernels or CUs to specific SLRs to improve placement and timing results. SLR assignment is especially important when assigning kernel ports to specific memory banks as described in Mapping Kernel Ports to Memory.

Specific availability of SLR in an Alveo accelerator card can be determined with the `platforminfo` command For instance, the U250 card reports the following information with regard to SLRs:

```
Valid SLRs
:
SLR0, SLR1, SLR2, SLR3
```

You can use the actual kernel resource usage 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, Block RAMs, etc.) used by the kernels early in the design cycle. Use this information along with the available SLR resources to help assign CUs to SLRs such that no one SLR is over-utilized.

IMPORTANT! *If your kernel is too large to fit into a single SLR, the Vitis compiler automatically places the logic across multiple SLRs. In this case, do not assign the SLR or this could result in an error during implementation.*

A CU can be assigned to an SLR using the `connectivity.slr` option in a config file. 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,...

AMD recommends assigning a kernel to a DDR memory resource in the same SLR as the kernel is placed. This reduces competition for limited SLR-crossing connection resources, and the use of super long line (SLL) routing resources which incur a greater delay than a standard routing. It might be necessary to connect a kernel to a DDR resource in a different SLR. However, if both the `connectivity.sp` and the `connectivity.slr` directives are explicitly defined, the tool automatically adds additional crossing logic to minimize the effect of the SLL delay, and facilitates better timing closure.

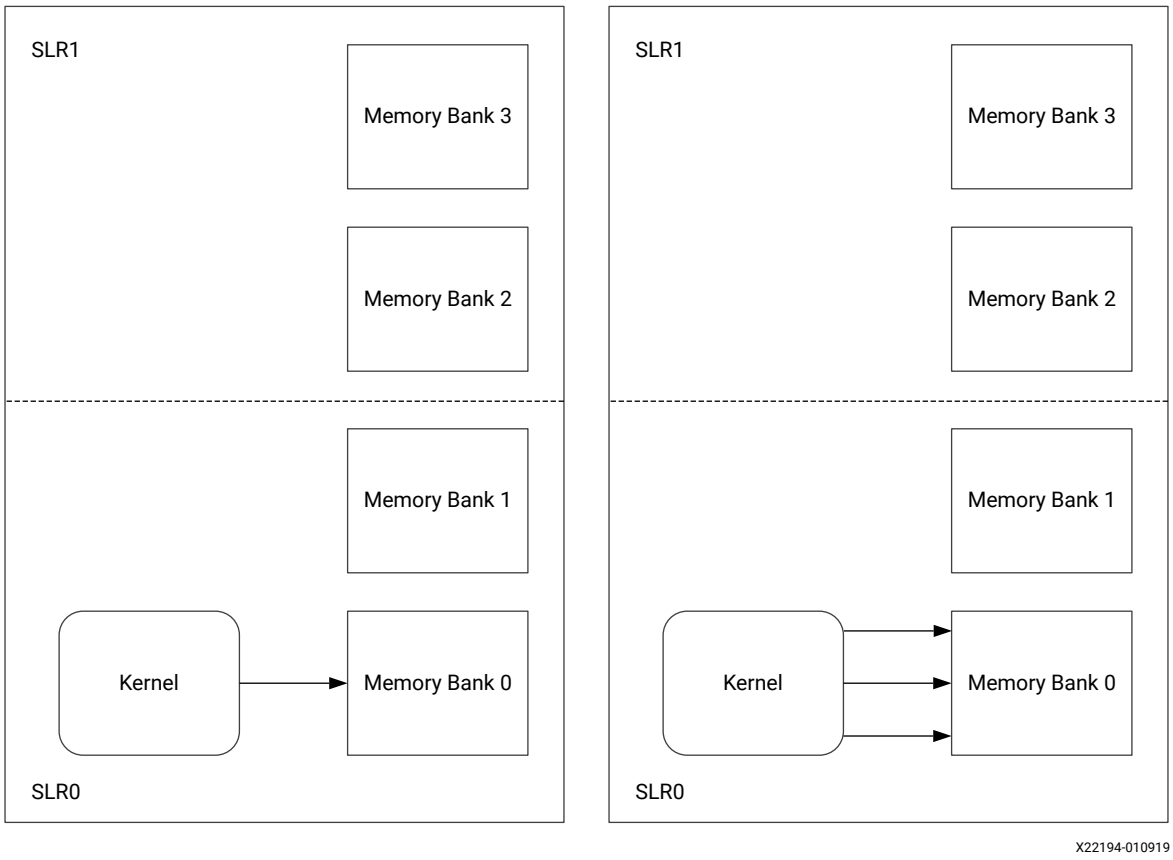
IMPORTANT! *When building the hardware and specifying trace memory for the profile command, as described in --profile Options, be aware of CU placement and assign memory resources according to SLR use. This can improve timing by limiting and managing SLR crossings. The command syntax is as follows:*

```
--profile.trace_memory <memory>:<SLR>
```

SLR 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 33: **Kernel and Memory in Same SLR**



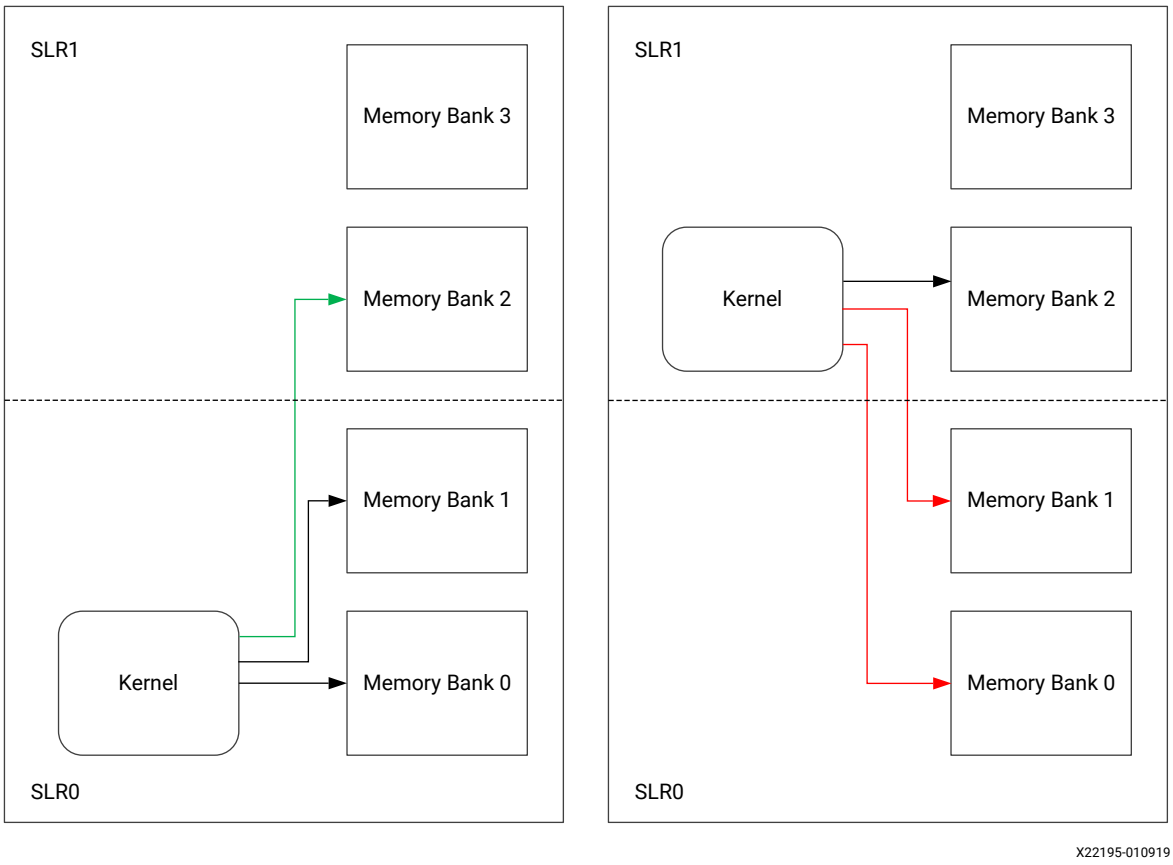
Note: The image on the left shows a single AXI interface mapped to a single memory bank. The image on the right shows multiple AXI interfaces mapped to the same memory bank.

As shown in the previous figure, when a kernel has a single AXI interface that maps only a single memory bank, the `platforminfo` utility described in platforminfo Utility in the *Vitis Reference Guide* (UG1702) 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 must reside in the same SLR as the memory bank that its AXI interfaces are mapping.

Figure 34: **Memory Bank in Adjoining SLR**

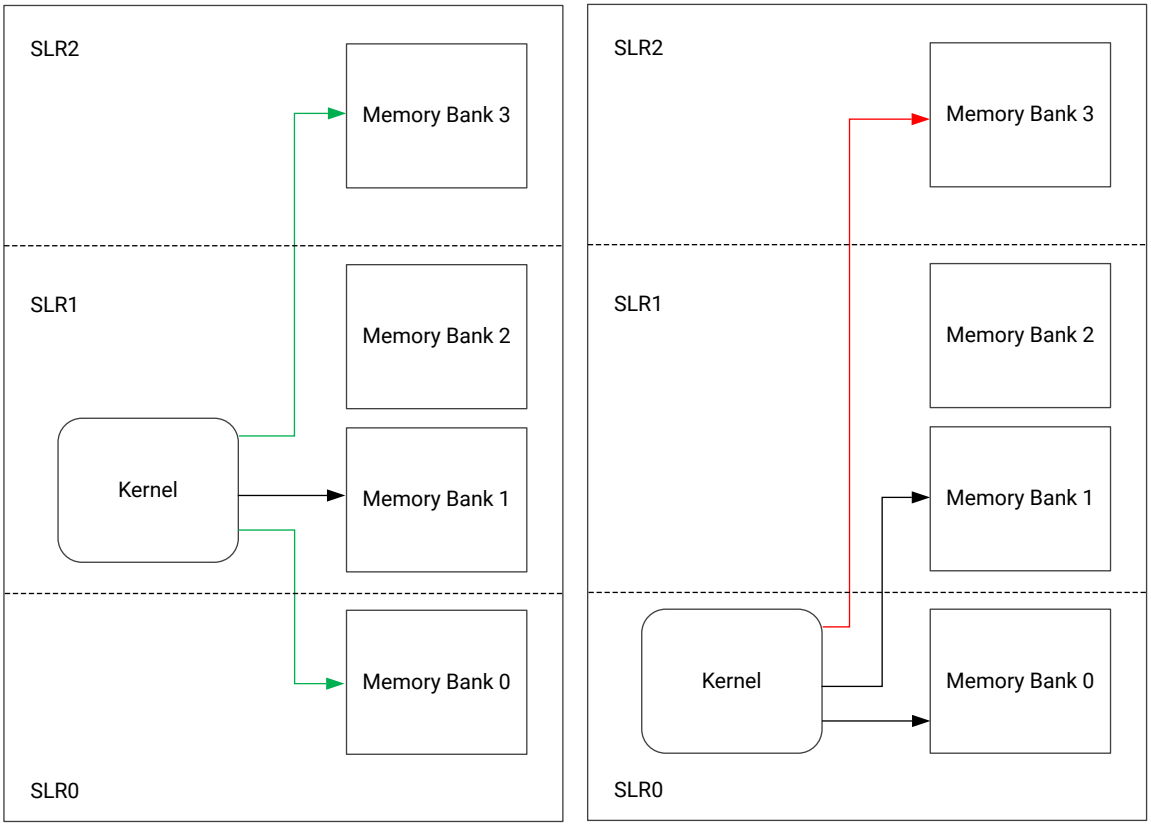


Note: The image on the left shows one SLR crossing is required when the kernel is placed in SLR0. The image on the right shows two SLR crossings are required for kernel to access memory banks.

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

When the kernel is mapping memory banks from different SLRs, explicitly specify the SLR assignment as described in Assigning Compute Units to SLRs on Alveo Accelerator Cards.

Figure 35: **Memory Banks Two SLRs Away**



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

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

Managing Clock Frequencies

AMD data-center acceleration platforms can have multiple kernels running at different clock frequencies under user control. The selection of kernel clocks and associated frequencies is an important part of defining the performance of algorithms and signal processing blocks of a system. This section describes the clocking of PL kernels added with Vitis to an extensible platform.

For details on how to use link options, refer to --clock Options.

Using --freqhz for Clock Management

Resource Identifier

The value of clock directive `--freqhz` supports either no units or MHz. For example, to connect clock frequency 100 MHz to any kernel in v++ command, `--freqhz` can be given as: `--freqhz=100000000` or `--freqhz=100MHz`. The unit MHz is case insensitive and supports values using up to 6 decimals. Example, `--freqhz=312.005MHz` is supported whereas `--freqhz=312.0000005MHz` is not supported.

The process for managing clock frequencies is specified below:

Default Values

The default clock is derived from the platform, which is passed to the `v++` option and can be overridden using the --`--freqhz` option.

v++ -c --mode hls

The default clock is derived from the platform, and can be overridden using the --`freqhz` option. Specifying a different `--hls.clock` frequency will result in an error during the `v++ --mode hls` option.

`--freqhz`can be specified in `v++ -c --mode hls` as:

```
v++ -c --mode hls -platform <pfm_name> --freqhz=150000000 --config hls.cfg
```

`--freqhz` can be specified in the configuration file as:

```
v++ -c --mode aie --platform  <pfm_name> --config hls.cfg
```

In `hls.cfg`, `--freqhz` can be given as:

```
[clock]
```

```
freqhz=200000000 OR freqhz=200MHz
```

v++ --link

- **--platform:** The default clock is derived from the platform, and can be overridden using the `--freqhz` option.

```
v++ -l -t hw -platform <pfm_name> --freqhz=200000000:mm2s \
--freqhz=200MHz:s2mm -config system.cfg
```

In `system.cfg`, `--freqhz` can be given as:

```
[clock]
```

```
freqhz=200000000:vadd.clk OR freqhz=200MHz:vadd.clk
```

Identifying Platform Clocks

Scalable Clocks (AMD Alveo™ Platforms only)

***Note*:** See *Embedded Design Development Using Vitis* (UG1701) for Platform Clocks information.

An AMD Alveo platform provides a frequency scalable kernel clock with ID (or Index) 0 that drives all XRT managed kernels. XRT can set the clock frequency of this clock according to the metadata contained in the `xclbin` file, when loading the file. An Alveo platform also provides a second scalable clock with ID 1 that is also controllable based on `xclbin` metadata. You do not need to provide an option to connect to scalable clocks, but you can specify the clock frequency during `v++` linking using the `--freqhz` option or `--kernel_frequency`, as described in v++ General Options in the *Vitis Reference Guide* (UG1702). The `v++` linker automatically connects PL kernel clocks `ap_clk` to clock ID 0, and `ap_clk2` to clock ID 1.

TIP: *In practice, `ap_clk2` is primarily found on RTL kernels, because the HLS compiler does not generate kernels with multiple clocks.*

You can determine the clocks available in the target platform by using the `platforminfo` command.

```
=================
Clock Information
=================
  Default Clock Index: 2
  Default Clock Frequency: 312.499712
  Default Clock Pretty Name: PL 2
  Clock Index:          0
    Frequency:          156.249856
    Status:             fixed
    Name:               clk_wizard_0_clk_out2
    Pretty Name:        PL 0
    Inst Ref:           clk_wizard_0
    Comp Ref:           clk_wizard
    Period:             6.400006
    Normalized Period:  .006400
  Clock Index:          1
    Frequency:          104.166570
    Status:             fixed
    Name:               clk_wizard_0_clk_out1
    Pretty Name:        PL 1
    Inst Ref:           clk_wizard_0
    Comp Ref:           clk_wizard
    Period:             9.600009
    Normalized Period:  .009600
  Clock Index:          2
    Frequency:          312.499712
    Status:             fixed
```

```
    Name:               clk_wizard_0_clk_out3
    Pretty Name:        PL 2
    Inst Ref:           clk_wizard_0
    Comp Ref:           clk_wizard
    Period:             3.200003
    Normalized Period:  .003200
  Clock Index:          3
    Frequency:          78.124928
    Status:             fixed
    Name:               clk_wizard_0_clk_out4
    Pretty Name:        PL 3
    Inst Ref:           clk_wizard_0
    Comp Ref:           clk_wizard
    Period:             12.800012
    Normalized Period:  .012800
  Clock Index:          4
    Frequency:          208.333141
    Status:             fixed
    Name:               clk_wizard_0_clk_out5
    Pretty Name:        PL 4
    Inst Ref:           clk_wizard_0
    Comp Ref:           clk_wizard
    Period:             4.800004
    Normalized Period:  .004800
  Clock Index:          5
    Frequency:          416.666283
    Status:             fixed
    Name:               clk_wizard_0_clk_out6
    Pretty Name:        PL 5
    Inst Ref:           clk_wizard_0
    Comp Ref:           clk_wizard
    Period:             2.400002
    Normalized Period:  .002400
  Clock Index:          6
    Frequency:          624.999425
    Status:             fixed
    Name:               clk_wizard_0_clk_out7
    Pretty Name:        PL 6
    Inst Ref:           clk_wizard_0
    Comp Ref:           clk_wizard
    Period:             1.600001
    Normalized Period:  .001600
```

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.

  TIP: *`report_failfast` is a utility that highlights potential device usage challenges, clock constraint problems, and potential unreachable target frequency (MHz).*
- `-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.
- `-Restimate`: Forces Vitis HLS to generate a System Estimate report, as described in System Estimate Report in the *Data Center Acceleration using Vitis* (UG1700).

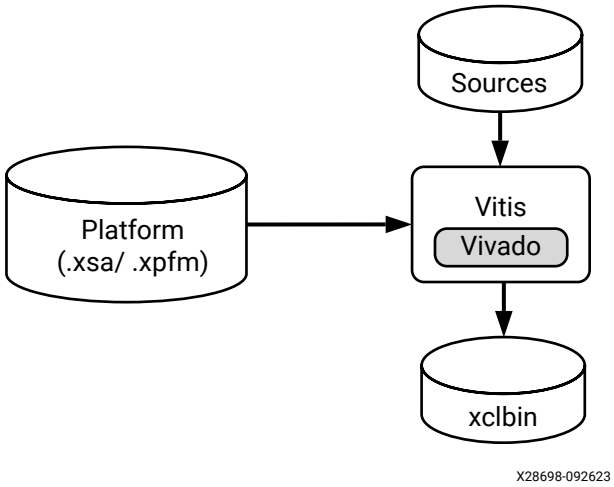
Managing Vivado Synthesis, Implementation, and Timing Closure

TIP: *This topic requires an understanding of the Vivado Design Suite tools and design methodology as described in UltraFast Design Methodology Guide for FPGAs and SoCs (UG949).*

The Vitis linking process use the Vivado Design Suite for synthesis and implementation of the linked system design.

When building a device binary for an AMD data-center accelerator card, the system design is automatically synthesized and implemented in the Vivado Design Suite during the `v++` linking phase.

Figure 36: **Vitis Integrated Flow**



Working with Vivado in the Vitis Integrated Flow

The Vitis Integrated Flow automatically launches the Vivado Design Suite to synthesize the linked system design, place and route the elements of the design, resolve timing, and generate the bitstream for the design. In most cases, the Vitis Integrated Flow completely abstract away the underlying process of synthesis and implementation of the hardware design. This removes the application developer from the typical hardware development process and the need to manage constraints such as logic placement and routing delays. The Vitis Integrated Flow automates much of the FPGA implementation process.

While automated, this flow does offer some opportunity for manual intervention. The process is broken down into a series of major steps that can be interrupted to enable customization when necessary. In some cases, you might want to exercise some control over the synthesis and implementation processes deployed by the Vitis linker, especially when large designs are being implemented. The Vitis Integrated Flow offers some control through specific options that can be specified in a `v++` configuration file, or from the command line. The following sections describe some of the methods you can use to control the Vivado synthesis and implementation results.

- Using the `--vivado` options to manage the Vivado tool.
- Using multiple implementation strategies to achieve timing closure on challenging designs.
- Using the `-to_step` and `-from_step` options to run the compilation or linking process to a specific step, perform some manual intervention on the design, and resume from that step.
- Interactively editing the Vivado project, and using the results for generating the FPGA binary.

Using the --vivado and --advanced Options

Using the `--vivado` option, as described in --vivado Options, and the `--advanced` option as described in --advanced Options, you can perform a number of interventions on the standard Vivado synthesis or implementation.

1. You can specify the strategy to be used by the Vivado tool during synthesis, implementation, or when generating reports during the build process. The strategy specified can be one of the standard tool strategies, or can be a custom-defined strategy that you have previously created in the Vivado tool. Use the `--vivado.prop` command as shown below.

   The original Tcl command to set the property on a run object looks like the following:

   ```
   set_property strategy Flow_AreaOptimized_medium [get_runs synth_1]
   ```

   The `v++` command is rewritten as shown below:

   - Synthesis Strategy:

     ```
     --vivado.prop run.synth_1.strategy=Flow_AreaOptimized_medium
     ```

   - Implementation Strategy:

     ```
     --vivado.prop run.impl_1.strategy=Performance_ExtraTimingOpt
     ```

   - Report Strategy: Can be specified for synthesis or implementation runs.

     ```
     --vivado.prop run.synth_1.report_strategy=MyCustom_Reports
     ```

     ```
     --vivado.prop run.impl_1.report_strategy={Timing Closure Reports}
     ```

   The command line is broken down as follows:

   - `--vivado.prop` is the `v++` command-line option to assign properties to objects as described in --vivado Options.
   - `run.<run_name>.strategy=<strategy_name>` to assign a `strategy` property (or `report_strategy`) to the specified synthesis or implementation run. Default run names are `synth_1` for synthesis or `impl_1` for implementation.
   - Strategy names with spaces in them, such as `{Timing Closure Reports}` require braces or double-quotes to group the words as shown above.

   > **TIP:** *You can also specify multiple implementation strategies to run as described in Running Multiple Implementation Strategies for Timing Closure.*

2. Pass Tcl scripts with custom design constraints or scripted operations.

You can create Tcl scripts to assign XDC design constraints to objects in the design, and pass these Tcl scripts to the Vivado tools using the PRE and POST Tcl script properties of the synthesis and implementation steps. For more information on Tcl scripting, refer to the *Vivado Design Suite User Guide: Using Tcl Scripting* (UG894). While there is only one synthesis step, there are a number of implementation steps as described in the *Vivado Design Suite User Guide: Implementation* (UG904). You can assign Tcl scripts for the Vivado tool to run before the step (PRE), or after the step (POST). The specific steps you can assign Tcl scripts to include the following: `SYNTH_DESIGN`, `INIT_DESIGN`, `OPT_DESIGN`, `PLACE_DESIGN`, `ROUTE_DESIGN`, `WRITE_BITSTREAM`.

TIP: *There are also some optional steps that can be enabled using the `--vivado.prop run.impl_1.steps.phys_opt_design.is_enabled=1` option. When enabled, these steps can also have Tcl PRE and POST scripts.*

An example of the Tcl PRE and POST script assignments follow:

```
--vivado.prop run.impl_1.STEPS.PLACE_DESIGN.TCL.PRE=/…/xxx.tcl
```

In the preceding example a script has been assigned to run before the PLACE_DESIGN step. The command line is broken down as follows:

- `--vivado` is the `v++` command-line option to specify directives for the Vivado tools.
- `prop` keyword to indicate you are passing a property setting.
- `run.` keyword to indicate that you are passing a run property.
- `impl_1.` indicates the name of the run.
- `STEPS.PLACE_DESIGN.TCL.PRE` indicates the run property you are specifying.
- `/.../xx.tcl` indicates the property value.

TIP: *Both the `--advanced` and `--vivado` options can be specified on the `v++` command line, or in a configuration file specified by the `--config` option. The example above shows the command line use, and the following example shows the config file usage. Refer to Vitis Compiler Configuration File in the Vitis Reference Guide (UG1702) for more information.*

3. Setting properties on run, file, and fileset design objects.

   This is very similar to passing Tcl scripts as described above, but in this case you are passing values to different properties on multiple design objects. For example, to use a specific implementation strategy such as `Performance_Explore` and disable global buffer insertion during placement, you can define the properties as shown below:

```
[vivado]
prop=run.impl_1.STEPS.OPT_DESIGN.ARGS.DIRECTIVE=Explore
prop=run.impl_1.STEPS.PLACE_DESIGN.ARGS.DIRECTIVE=Explore
prop=run.impl_1.{STEPS.PLACE_DESIGN.ARGS.MORE OPTIONS}={-no_bufg_opt}
prop=run.impl_1.STEPS.PHYS_OPT_DESIGN.IS_ENABLED=true
prop=run.impl_1.STEPS.PHYS_OPT_DESIGN.ARGS.DIRECTIVE=Explore
prop=run.impl_1.STEPS.ROUTE_DESIGN.ARGS.DIRECTIVE=Explore
```

In the example above, the `Explore` value is assigned to the `STEPS.XXX.DIRECTIVE` property of various steps of the implementation run. Note the syntax for defining these properties is:

```
<object>.<instance>.property=<value>
```

Where:

- `<object>` can be a design run, a file, or a fileset object.
- `<instance>` indicates a specific instance of the object.
- `<property>` specifies the property to assign.
- `<value>` defines the value of the property.

In this example the object is a run, the instance is the default implementation run, `impl_1`, and the property is an argument of the different step names, In this case the DIRECTIVE, IS_ENABLED, and {MORE OPTIONS}. Refer to --vivado Options for more information on the command syntax.

4. Enabling optional steps in the Vivado implementation process.

   The build process runs Vivado synthesis and implementation to generate the device binary. Some of the implementation steps are enable and run as part of the default build process, and some of the implementation steps can be optionally enabled at your discretion.

   Optional steps can be listed using the `--list_steps` command, and include: `vpl.impl.power_opt_design`, `vpl.impl.post_place_power_opt_design`, `vpl.impl.phys_opt_design`, and `vpl.impl.post_route_phys_opt_design`.

   An optional step can be enabled using the `--vivado.prop` option. For example, to enable PHYS_OPT_DESIGN step, use the following config file content:

   ```
   [vivado]
   prop=run.impl_1.steps.phys_opt_design.is_enabled=1
   ```

   When an optional step is enabled as shown above, the step can be specified as part of the `-from_step`/`-to_step` command as described below in *Running --to_step or --from_step*, or enable a Tcl script to run before or after the step as described in --linkhook Options.

5. Passing parameters to the tool to control processing.

   The `--vivado` option also allows you to pass parameters to the Vivado tools. The parameters are used to configure the tool features or behavior prior to launching the tool. The syntax for specifying a parameter uses the following form:

   ```
   --vivado.param <object><parameter>=<value>
   ```

   The keyword `param` indicates that you are passing a parameter for the Vivado tools, rather than a property for a design object. You must also define the `<object>` it applies to, the `<parameter>` that you are specifying, and the `<value>` to assign it.

The following example project indicates the current Vivado project, `writeIntermedateCheckpoints`, is the parameter being passed and the value is 1, which enables this boolean parameter.

```
--vivado.param project.writeIntermediateCheckpoints=1
```

6. Managing the reports generated during synthesis and implementation.

> **IMPORTANT!** *You must also specify `--save-temps` on the `v++` command line when customizing the reports generated by the Vivado tool to preserve the temporary files created during synthesis and implementation, including any generated reports.*

You might also want to generate or save more than the standard reports provided by the Vivado tools when run as part of the Vitis tools build process. You can customize the reports generated using the `--advanced.misc` option as follows:

```
[advanced]
misc=report=type report_utilization name
synth_report_utilization_summary steps {synth_design} runs {__KERNEL__}
options {}
misc=report=type report_timing_summary name
impl_report_timing_summary_init_design_summary steps {init_design} runs
{impl_1} options {-max_paths 10}
misc=report=type report_utilization name
impl_report_utilization_init_design_summary steps {init_design} runs
{impl_1} options {}
misc=report=type report_control_sets name
impl_report_control_sets_place_design_summary steps {place_design} runs
{impl_1} options {-verbose}
misc=report=type report_utilization name
impl_report_utilization_place_design_summary steps {place_design} runs
{impl_1} options {}
misc=report=type report_io name impl_report_io_place_design_summary
steps {place_design} runs {impl_1} options {}
misc=report=type report_bus_skew name
impl_report_bus_skew_route_design_summary steps {route_design} runs
{impl_1} options {-warn_on_violation}
misc=report=type report_clock_utilization name
impl_report_clock_utilization_route_design_summary steps {route_design}
runs {impl_1} options {}
```

The syntax of the command line is explained using the following example:

```
misc=report=type report_bus_skew name
impl_report_bus_skew_route_design_summary steps {route_design} runs
{impl_1} options {-warn_on_violation}
```

- **`misc=report=`**: Specifies the `--advanced.misc` option as described in --advanced Options, and defines the report configuration for the Vivado tool. The rest of the command line is specified in name/value pairs, reflecting the options of the `create_report_config` Tcl command as described in *Vivado Design Suite Tcl Command Reference Guide* (UG835).
- **`type report_bus_skew`**: Relates to the `-report_type` argument, and specifies the type of the report as the `report_bus_skew`. Most of the `report_*` Tcl commands can be specified as the report type.

- **`name impl_report_bus_skew_route_design_summary`:** Relates to the `-report_name` argument, and specifies the name of the report. Note this is not the file name of the report, and generally this option can be skipped as the report names will be auto-generated by the tool.
- **`steps {route_design}`:** Relates to the `-steps` option, and specifies the synthesis and implementation steps that the report applies to. The report can be specified for use with multiple steps to have the report regenerated at each step, in which case the name of the report will be automatically defined.
- **`runs {impl_1}`:** Relates to the `-runs` option, and specifies the name of the design runs to apply the report to.
- **`options {-warn_on_violation}`:** Specifies various options of the `report_*` Tcl command to be used when generating the report. In this example, the `-warn_on_violation` option is a feature of the `report_bus_skew` command.

> **IMPORTANT!** *There is no error checking to ensure the specified options are correct and applicable to the report type specified. If you indicate options that are incorrect the report will return an error when it is run.*

Running Multiple Implementation Strategies for Timing Closure

For challenging designs, it can take multiple iterations of Vivado implementation using multiple different strategies to achieve timing closure. This topic shows you how to launch multiple implementation strategies at the same time in the hardware build (`-t hw`), and how to identify and use successful runs to generate the device binary and complete the build.

As explained in --vivado Options, the `--vivado.impl.strategies` command enables you to specify multiple strategies to run in a single build pass. The command line would look as follows:

```
v++ --link -s -g -t hw --platform xilinx_zcu102_base_202010_1 -I . \
--vivado.impl.strategies "Performance_Explore,Area_Explore" -o
kernel.xclbin hello.xo
```

In the example above, the `Performance_Explore` and `Area_Explore` strategies are run simultaneously in the Vivado build to see which returns the best results. You can specify the `ALL` to have all available strategies run within the tool.

> **IMPORTANT!** *Running ALL implementation strategies might launch 30 or more runs in the Vivado tool, including any user-defined strategies stored in your home directory (`~/.Xilinx/Vivado/202X.X/strategies`). This can be a tremendous drain on resources, and is not advised. You can prevent this by defining specific strategies to run, and using a command queue to distribute the process load in some managed way, such as through the `--vivado.impl.jobs` or the `--vivado.impl.lsf` commands.*

You can also determine this option in a configuration file in the following form:

```
#Vivado Implementation Strategies
[vivado]
impl.strategies=Performance_Explore,Area_Explore
```

The Vitis compiler automatically picks the first completed run results that meets timing to proceed with the build process and generate the device binary. However, you can also direct the tool to wait for all runs to see the result of all strategies. This would require the use of the `{}ALL_IMPL{}` macro to apply custom settings to all runs and the `multiStrategiesWaitOnAllRuns` directive to see the result of all strategies:

```
[advanced]
#param=compiler.multiStrategiesWaitOnAllRuns=1

[vivado]

impl.strategies=ALL
prop=run.{}ALL_IMPL{}.STEPS.PLACE_DESIGN.TCL.PRE=../../vpp_cfg/
place_design_pre.tcl
prop=run.{}ALL_IMPL{}.STEPS.ROUTE_DESIGN.TCL.PRE=../../vpp_cfg/
route_design_pre.tcl
```

`compiler.multiStrategiesWaitOnAllRuns=0` represents the default behavior. If you want `v++` to wait for all runs to complete, which will get their report files, change that parameter value to 1. This includes an overview of the implementation results, in addition to a Timing Summary report. Seeing all results will give you an indication on how hard it is to close timing for the tool for the all strategies that are allowed to run to completion. You can use this feature to review the different strategies and results.

You can also manually review the results of all implementation strategies after they have completed. Then, use the results of any of the implementation runs by using the `--reuse_impl` option as described in Using --to_step and Launching Vivado Interactively.

Using --to_step and Launching Vivado Interactively

The Vitis compiler lets you stop the build process after completing a specified step (`--to_step`), manually intervene in the design or files in some way, and then continue the build by specifying a step the build resumes from (`--from_step`). The `--from_step` directs the Vitis compiler to resume compilation from the step where `--to_step` left off, or some earlier step in the process. The `--to_step` and `--from_step` are described in v++ Command in the *Vitis Reference Guide* (UG1702).

> **IMPORTANT!** *The `--to_step` and `--from_step` options are sequential build options that require you to use the same project directory when launching `v++ --link --from_step` as you specified when using `v++ --link --to_step`.*

The Vitis compiler also provides a `--list_steps` option to list the available steps for the compilation or linking processes of a specific build target. For example, the list of steps for the link process of the hardware build can be found by:

```
v++ --list_steps --target hw --link
```

This command returns a number of steps, both default steps and optional steps that the Vitis compiler goes through during the linking process of the hardware build. Some of the default steps include: `system_link`, `vpl`, `vpl.create_project`, `vpl.create_bd`, `vpl.generate_target`, `vpl.synth`, `vpl.impl.opt_design`, `vpl.impl.place_design`, `vpl.impl.route_design`, and `vpl.impl.write_bitstream`.

Optional steps include: `vpl.impl.power_opt_design`, `vpl.impl.post_place_power_opt_design`, `vpl.impl.phys_opt_design`, and `vpl.impl.post_route_phys_opt_design`.

TIP: *An optional step must be enabled before specifying it with `--from_step` or `--to_step` as previously described in Using the --vivado and --advanced Options.*

Launching the Vivado IDE for Interactive Design

***Note*:** The Packaging for Vitis Export to Vivado Flow in the *Embedded Design Development Using Vitis* (UG1701) is the preferred method.

With the `--to_step` command, you can launch the build process to Vivado synthesis and then start the Vivado IDE on the project to manually place and route the design. To perform this you would use the following command syntax:

```
v++ --target hw --link --to_step vpl.synth --save-temps --platform
<PLATFORM_NAME> <XO_FILES>
```

TIP: *As shown in the example above, you must also specify `--save-temps` when using `--to_step` to preserve any temporary files created by the build process.*

This command specifies the link process of the hardware build, runs the build through the synthesis step, and saves the temporary files produced by the build process.

You can launch the Vivado tool directly on the project built by the Vitis compiler using the `--interactive` command. This opens the Vivado project found at `<temp_dir>/link/vivado/vpl/prj` in your build directory, letting you interactively edit the design:

```
v++ --target hw --link --interactive impl --save-temps --platform
<PLATFORM_NAME> <XO_FILES>
```

When invoking the Vivado IDE in this mode, you can open the synthesis or implementation runs to manage and modify the project. You can change the run details as needed to close timing and try different approaches to implementation. You can save the results to a design checkpoint (DCP), or generate the project bitstream (`.bit`) to use in the Vitis environment to generate the device binary.

After saving the DCP from within the Vivado IDE, close the tool and return to the Vitis environment. Use the `--reuse_impl` option to apply a previously implemented DCP file in the `v++` command line to generate the `xclbin`.

> **IMPORTANT!** *The `--reuse_impl` option is an incremental build option that requires you to apply the same project directory when resuming the Vitis compiler with `--reuse_impl` that you specified when using `--to_step` to start the build.*

The following command completes the linking process by using the specified DCP file from the Vivado tool to create the `project.xclbin` from the input files.

```
v++ --link --platform <PLATFORM_NAME> -o'project.xclbin' project.xo --
reuse_impl ./_x/link/vivado/routed.dcp
```

You can also use a bitstream file generated by the Vivado tool to create the `project.xclbin`:

```
v++ --link --platform <PLATFORM_NAME> -o'project.xclbin' project.xo --
reuse_bit ./_x/link/vivado/project.bit
```

***Note*:** The `project.bit` used for `--reuse_bit` is a partial bit and not a full bit.

Additional Vivado Options

Some additional switches that can be used in the `v++` command line or config file include the following:

- `--export_script`/`--custom_script` edit and use Tcl scripts to modify the compilation or linking process.
- `--remote_ip_cache` specify a remote IP cache directory for Vivado synthesis.
- `--no_ip_cache` turn off the IP cache for Vivado synthesis. This causes all IP to be re-synthesized as part of the build process, scrubbing out cached data.

Running the System on Hardware

Running the system depends on the build target. The process of running on the physical hardware is different from running hardware emulation.

Running the hardware build lets you see your application running an accelerator card such as the AMD Alveo™ Data Center accelerator card. The performance data and results captured here are the actual performance of your accelerated application. Yet the profiling data from this run might still reveal opportunities to optimize your design.

> **TIP:** *To use the accelerator card, you must have it installed as described in Getting Started with Alveo Data Center Accelerator Cards (UG1301).*

1. Edit the `xrt.ini` file as described in xrt.ini File.

   This is optional, but recommended when running on hardware for evaluation purposes. You can configure XRT with the `xrt.ini` file to capture debugging and profile data as the application is running. To capture event trace data when running the hardware, refer to Enabling Profiling in Your Application. To debug the running hardware, refer to Debugging During Hardware Execution.

   TIP: *Ensure to use the `v++ -g` option when compiling your kernel code for debugging.*

2. Unset the `XCL_EMULATION_MODE` environment variable.

   IMPORTANT! *The hardware build will not run if the `XCL_EMULATION_MODE` environment variable is set to an emulation target.*

3. Run your application.

   The specific command line to run the application will depend on your host code. A common implementation used in AMD tutorials and examples is as follows:

   ```
   ./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 AMD Vitis™ examples and tutorials do. However, your application can have the name of the `xclbin` file hard-coded into the host program, or can require a different approach to running the application.*

When running the design you can specify a number of trace options as described in Enabling Profiling in Your Application to capture design data during runtime. Any reports generated during the run are collected into the `xrt.run_summary` file. This collection of reports can be viewed by opening the run_summary in Vitis analyzer, and includes a Summary report, System and Platform Diagrams to illustrate the hardware design, Run Guidance offering any suggestions for improving the performance of the system, and a Profile Summary and Timeline Trace when enabled in the `xrt.ini` file during runtime. Refer to Working with the Analysis View (Vitis Analyzer) in the *Vitis Reference Guide* (UG1702) for additional information.

Application Verification Using Vitis Emulation Flow

Development of an application and hardware kernels targeting an FPGA requires a phased development approach. Because FPGA are programmable devices, building the device binary for hardware takes some time. To enable quicker iterations without having to go through the full hardware compilation flow, the AMD Vitis™ tool provides hardware emulation target to perform co-simulation of the software application and PL kernels. Compiling for hardware emulation is significantly faster than compiling for the actual hardware. Additionally, hardware emulation target provides full visibility into the application or accelerator, thus making it easier to perform debugging. Once your design passes in hardware emulation, then in the late stages of development you can compile and run the application on the hardware platform.

The Vitis tool provides following emulation target:

- **Hardware emulation (hw_emu):** The host program runs natively on x86, but the kernel code is compiled into an RTL behavioral model which is run in the AMD Vivado™ simulator or other supported third-party simulators. This build and run loop takes longer but provides a cycle-accurate view of kernel logic.

Compiling and linking for emulation is seamlessly integrated into the Vitis command line and IDE flows. You can compile your host and kernel source code for hardware emulation target, without making any change to the source code. For your host code, you do not need to compile differently for emulation as the same host executable can be used in emulation. Hardware emulation target support most of the features including XRT APIs, buffer transfer, platform memory SP tags, kernel-to-kernel connections, etc. The following sections detail the features and requirements of the hardware emulation flow.

While running emulation you can specify a number of trace options as described in Enabling Profiling in Your Application to capture design data during runtime. Any reports generated during the run are collected into the `xrt.run_summary` file. This collection of reports can be viewed by opening the run_summary in Vitis analyzer, and includes a Summary report, System and Platform Diagrams to illustrate the hardware design, Run Guidance offering any suggestions for improving the performance of the system, and a Profile Summary and Timeline Trace when enabled in the `xrt.ini` file during runtime. Refer to Working with the Analysis View (Vitis Analyzer) in the *Vitis Reference Guide* (UG1702) for additional information.

Installing the x86 XRT automatically sets the LD_LIBARY_PATH variable to point to XRT libraries.

Running Hardware Emulation

TIP: *Set up the command shell or window as described in Setting Up the Vitis Environment in the Data Center Acceleration using Vitis (UG1700) prior to running the builds.*

1. Set the desired runtime settings in the `xrt.ini` file. This step is optional.

   As described in xrt.ini File, the 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 data 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 `v++ -g` option when compiling your kernel code for emulation mode.*

2. Generate an `emconfig.json` file from the target platform as described in emconfigutil Utility in the *Vitis Reference Guide* (UG1702). This is required for running hardware emulation.

   The emulation configuration file, `emconfig.json`, is generated from the specified platform using the `emconfigutil` command, and provides information used by the XRT library during emulation. The following example creates the `emconfig.json` file for the specified target platform:

   ```
   emconfigutil --platform xilinx_u200_xdma_201830_2
   ```

   In emulation mode, the runtime looks for the `emconfig.json` file at a location specified by the `$EMCONFIG_PATH` variable, or in the same directory as the host executable.

   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 `hw_emu` (hardware emulation). This changes the application execution to emulation mode.

   Use the following syntax to set the environment variable for C shell (csh):

   ```
   setenv XCL_EMULATION_MODE hw_emu
   ```

   Bash shell:

   ```
   export  XCL_EMULATION_MODE=hw_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 file (`xrt.ini`), emulation configuration file (`emconfig.json`), and the `XCL_EMULATION_MODE` environment set, run the host executable with the desired command line argument.

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 AMD Vitis™ examples and tutorials do. However, your application might have the name of the `xclbin` file hard-coded into the host program, or might require a different approach to running the application.*

Speed and Accuracy of Hardware Emulation

Hardware emulation uses a mix of SystemC and RTL co-simulation to provide a balance between accuracy and speed of simulation. The SystemC models are composed of purely functional models and performance approximate models. Hardware emulation does not mimic hardware accuracy 100%, so you can expect some differences in behavior between running emulation and executing your application on hardware. This can lead to significant differences in application performance, and sometimes differences in functionality can also be observed.

Functional differences with hardware typically point to a race condition or some unpredictable behavior in your design. So, an issue seen in hardware might not always be reproducible in hardware emulation, though most behavior related to interactions between the host and the accelerator, or the accelerator and the memory are reproducible in hardware emulation. This makes hardware emulation an excellent tool to debug issues with your accelerator prior to running on hardware.

The following table lists models that are used to mimic the hardware platform and their accuracy levels.

Table 12: **Hardware Platform**

Hardware Functionality	Description
Host to Card PCIe® Connection and DMA (XDMA, SlaveBridge)	For data center platforms, the connection to the x86 host server over PCIe is done as a purely functional model and does not have any performance modeling. Thus, any issues related to PCIe bandwidth cannot be reflected in hardware emulation runs.
AMD UltraScale™ DDR Memory, SmartConnect	The SystemC models for the DDR memory controller, AXI SmartConnect, and other data path IPs are usually throughput approximate. They typically do not model the exact latency of the hardware IP. The model can be used to gauge a relative performance trend as you modify your application or the accelerator kernel.
User Kernel (accelerator)	Hardware emulation uses RTL for the user accelerator. As follows, the accelerator behavior by itself is 100% accurate. However, the accelerator is surrounded by other approximate models.

Table 12: **Hardware Platform** *(cont'd)*

Hardware Functionality	Description
Other Functionality	For hardware emulation, there is generic Python or C-based traffic generator which can be interfaced with the emulation process. You can generate abstract traffic at AXI protocol level. Because these models are abstract, any issues observed on the specific hardware board will not be shown in hardware emulation.

Because hardware emulation uses RTL co-simulation as its execution model, the speed of execution is orders of magnitude slower as compared to real hardware. AMD recommends using small data buffers. For example, if you have a configurable vector addition and in hardware you are performing a 1024 element `vadd`, in emulation you might restrict it to 16 elements. This will enable you to test your application with the accelerator, while still completing execution in reasonable time.

Simulator Support in Hardware Emulation

The AMD Vitis™ tool uses the AMD Vivado™ logic simulator (`xsim`) as the default simulator for all platforms.

For data center platforms, hardware emulation supports the U250_XDMA platform with Questa Advanced Simulator. This support does not include features like peer-to-peer (P2P), SlaveBridge, or other features unless explicitly mentioned.

Enabling a third-party simulator requires some additional configuration options during generation of the device binary (`.xclbin`) and supporting Tcl scripts. The specific requirements for each simulator are discussed below. In addition, run the Vivado setup for third-party simulators before using those simulators in Vitis. Specifically, you must pre-compile the simulation models using the `compile_sim_lib` Tcl command. For more details, see the *Vivado Design Suite User Guide: Logic Simulation* (UG900) for third-party simulator setup.

- **Questa:** Add the following advanced parameters and Vivado properties to a configuration file for use during linking:

```
[advanced]
param=hw_emu.simulator=QUESTA
[vivado]
prop=project.__CURRENT__.simulator.questa_install_dir=<Questa_install_dir>
"
prop=project.__CURRENT__.compxlib.questa_compiled_library_dir=<Questa_comp
iled_lib_path>
prop=fileset.sim_1.questa.compile.sccom.cores={16}
prop=fileset.sim_1.questa.elaborate.vopt.more_options={-stats=all}
prop=fileset.sim_1.questa.simulate.vsim.more_options={-stats=all}
```

After generating the configuration file you can use it in the `v++` command line as follows:

```
v++ -link --config questa_sim.cfg
```

- **Xcelium:** Add the following advanced parameters and Vivado properties to a configuration file for use during linking:

```
## Final set of additional options required for running simulation using
Xcelium Simulator
[advanced]
param=hw_emu.simulator=XCELIUM
[vivado]
prop=project.__CURRENT__.simulator.xcelium_install_dir=<Xcelium_install_di
r>
prop=project.__CURRENT__.compxlib.xcelium_compiled_library_dir=<Xcelium_pr
e-compiled_lib_path>
prop=fileset.sim_1.xcelium.elaborate.xmelab.more_options={-timescale
1ns/1ps -STATUS}
```

After generating the configuration file you can use it in the `v++` command line as follows:

```
v++ -link --config xcelium.cfg
```

- **VCS:** Add the following advanced parameters and Vivado properties to a configuration file for use during linking:

```
## Final set of additional options required for running simulation using
VCS Simulator
[advanced]
param=hw_emu.simulator=VCS
[vivado]
prop=project.__CURRENT__.simulator.vcs_install_dir=<VCS_install_dir>
prop=project.__CURRENT__.compxlib.vcs_compiled_library_dir=<Vcs_pre-
compiled_lib_path>
prop=project.__CURRENT__.simulator.vcs_gcc_install_dir=<VCS_gnu_pkg_instal
l_dir>
param=project.alignLibraryPathEnvForVCS=true
prop=fileset.sim_1.vcs.compile.vlogan.more_options={-v2005}
```

After generating the configuration file you can use it in the `v++` command line as follows:

```
v++ -link --config vcs_sim.cfg
```

- **Riviera:** Add the following advanced parameters and Vivado properties to a configuration file for use during linking:

```
## Final set of additional options required for running simulation using
VCS Simulator
[advanced]
param=hw_emu.simulator=RIVIERA
[vivado]
prop=project.__CURRENT__.simulator.riviera_install_dir=<Riviera_install_di
r>
```

```
prop=project.__CURRENT__.compxlib.riviera_compiled_library_dir=<Riviera_pr
e-compiled_lib_path>
prop=project.__CURRENT__.simulator.riviera_gcc_install_dir=<Riviera_gcc_pa
th>
prop=fileset.sim_1.riviera.simulate.asim.more_options={+access +r}
```

After generating the configuration file you can use it in the `v++` command line as follows:

```
v++ -link --config riviera.cfg
```

Using the Simulator Waveform Viewer

Hardware emulation uses RTL and SystemC models for execution. A regular application and HLS-based kernel developer does not need to be aware of the hardware level details. The Vitis analyzer provides sufficient details of the hardware execution model. However, for advanced users who are familiar with HW signal and protocols, they can launch hardware emulation with the simulator waveform running, as described in Waveform View and Live Waveform Viewer.

By default, when running `v++ --link -t hw_emu`, the tool compiles the simulation models in optimized mode. However, when you also specify the `-g` switch, you enable hardware emulation models to be compiled in debug mode. During the application runtime, use the `-g` switch with the `launch_hw_emu.sh` command to run the simulator interactively in GUI mode with waveforms displayed. By default, the hardware emulation flow adds common signals of interest to the waveform window. However, you can pause the simulator to add signals of interest and resume simulation.

AXI Transactions Display in XSIM Waveform

Many models in hardware emulation use SystemC transaction-level modeling (TLM). In these cases, interactions between the models cannot be viewed as RTL waveforms. However, Vivado simulator (xsim) provides a transaction level viewer. For standard platforms, these interface objects can be added to the waveform view, similar to how RTL signals are added. As an example, to add an AXI interface to the waveform, use the following Tcl command in `xsim`:

```
add_wave <HDL_objects>
```

Using the `add_wave` command, you can specify full or relative paths to HDL objects. For additional details on how to interpret the TLM waveform see Interpreting TLM Waveform Data for Third-Party Simulators in the *Data Center Acceleration using Vitis* (UG1700), or refer to *Vivado Design Suite User Guide: Logic Simulation* (UG900).

Generating Test Vectors for Vitis HLS during Hardware Emulation

Vitis tool to generate test vectors for simulation during hardware emulation, without re-running `v++` compilation and linking. The test vectors will enable Vitis HLS to run C/RTL Co-simulation without a dedicated C++ test bench for:

- deadlock analysis
- FIFO depth optimization
- other performance optimizations

Use the following steps:

1. Create an `hlsPre.tcl` file and insert this command:

   ```
   config_export -cosim_trace_generation
   ```

2. Run `v++ --compile` and keep the HLS project directories, under the `<compile_dir>`
3. Run `v++ --link --target hw_emu`
4. Run the application for hardware emulation
5. Locate the `hls_cosim` inside the HW_EMU run directory
   a. This directory contains one directory for each kernel, with one directory for each kernel CU (kernel instance) below it:

      ```
      <build_dir>/.run/<run_number>/hw_em/device0/binary_0/behav_waveform/
      xsim/hls_cosim/<kernel_name>
      ```

6. Copy the appropriate kernel directory to the HLS project directory, i.e.:

   ```
   cp -r <build_dir>/.run/<run_number>/.../xsim/hls_cosim/<kernel_name>
   <compile_dir>/<kernel_name>/<kernel_name>
   ```

7. Open the Vitis HLS tool and run C/RTL Co-simulation in batch mode or GUI mode:

   ```
   cosim_design -hwemu_trace_dir <kernel_name>/<instance_name> ...
   ```

The traces generated from HW_EMU are valid only as long as:

- The functionality of the kernel does not change
- The top interface of the kernel does not change
- The number of top interface reads and writes (`s_axilite` registers, `m_axi` interfaces, `axis` interfaces) does not change.

Chapter 6

Profiling 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 AMD Vitis™ core development kit generates various system and kernel resource performance reports during compilation. These reports help you establish a baseline of performance for your application, identify bottlenecks, and help to identify target functions that can be accelerated in hardware kernels as discussed in Methodology for Architecting a Device Accelerated Application. The Xilinx Runtime (XRT) collects profiling data during application execution in both emulation and hardware builds. Examples of profiling and event data that can be reported includes:

- Host and device timeline events
- OpenCL™ or XRT native API call sequences
- Kernel execution sequence
- Kernel start and stop signals
- FPGA trace data including AXI transactions
- Power profile data for the accelerator card
- User event and range profiling

Profiling reports and data can be used to isolate performance bottlenecks in the application, identify problems in the system, 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 must be optimized for performance and resource usage.

There are four distinct areas to be considered when performing algorithm optimization in the Vitis environment: System resource usage and performance, kernel optimization, host optimization, and data transfer optimization. The following Vitis reports and graphical tools support your efforts to profile and optimize these areas:

- Guidance
- System Estimate Report
- HLS Synthesis Report
- Profile Summary Report
- Timeline Trace
- Waveform View and Live Waveform Viewer

When enabled as described in Enabling Profiling in Your Application, these reports are automatically generated while running the active build, either from the command line as described in Chapter 4: Building and Running the System, or when Using the Vitis Unified IDE in the *Vitis Reference Guide* (UG1702). Separate reports are generated for the different build targets and can be found in the respective report directories. Reports can be viewed in the Analysis view in the IDE as described in Working with the Analysis View (Vitis Analyzer) in the *Vitis Reference Guide* (UG1702).

Enabling Profiling in Your Application

To enable event trace data during the execution of your application, you must instrument your application for this task. You must enable additional logic, and consume additional device resources to track the host and kernel execution steps, and capture event data. This process requires optionally modifying your host application to capture custom data, modifying your kernel XO during compilation and the `xclbin` during linking to capture different types of profile data from the device side activity, and configuring the Xilinx Runtime (XRT) as described in the xrt.ini File to capture data during the application runtime.

TIP: *While capturing profile data is a critical part of the profiling and optimization process for building your accelerated application, it does consume additional resources and impacts performance. Remember to clean these elements out of your final production build.*

There are many different types of profiling for your applications, depending on which elements your system includes, and what type of data you want to capture. The following table shows some of the levels of profiling that can be enabled, and discusses which are complimentary and which are not.

Table 13: **Profiling Host and Kernels**

Profile/Trace	Description	Comments
Host Application OpenCL API and some limited device side (kernel) profiling.	Specified by the use of the `opencl_trace` option in the `xrt.ini` file.	Generates the `opencl_trace.csv` file and the `xrt.run_summary` for viewing in Vitis analyzer.
Host Application XRT Native API	Specified by the use of the `native_xrt_trace` option in the `xrt.ini` file.	Generates profile summary and trace events for the XRT API as described in Writing the Software Application.
Host Application User-Event Profiling	Requires additional code in the host application as described in Custom Profiling of the Host Application.	Generates user range data and user events for the host application. **TIP:** *Can be used to capture event data for user-managed kernels as described in Working with User-Managed Kernels in the Data Center Acceleration using Vitis (UG1700).*
Low Overhead Profiling	Specified by the use of the `lop_trace` option in the `xrt.ini` file.	Generates the `lop_trace.csv` file as described in Enabling Low Overhead Profiling.
Device Side Profiling	Enabled by the use of `--profile` options during `v++` compilation and linking, as described in --profile Options, and the use of `device_trace` in the `xrt.ini` file.	Enables capturing data traffic between the host and kernel, kernel stalls, the execution times of kernels and compute units (CUs).
Power Profile	Specified by the use of the `power_profile` option in the `xrt.ini` file.	Generates the `power_profile_<device>.csv` report. ***Note***: This feature is not supported on embedded platforms or AWS.

The device binary (`xclbin`) file is configured for capturing limited device-side profiling data by default. However, using the `--profile` option during the Vitis compiler linking process instruments the device binary by adding Acceleration Monitors, AXI Performance Monitors, and Memory Monitors to the system. This option has multiple instrumentation options: `--profile.data`, `--profile.stall`, and `--profile.exec`, as described in the --profile Options.

As an example, add `--profile.data` to the `v++` linking command line:

```
v++ -g -l --profile.data all:all:all ...
```

TIP: *Be sure to also use the `v++ -g` option when compiling your kernel code for debugging with software or hardware emulation.*

After your application is enabled for profiling during the `v++` compile and link process, data gathering during application runtime must also be enabled in XRT by editing the `xrt.ini` file as discussed above. For example, the following `xrt.ini` file enables OpenCL profiling, power profiling, and event and stall trace capture when the application is run:

```
[Debug]
opencl_trace=true
power_profile=true
device_trace=fine
stall_trace=all
```

To enable the profiling of Kernel Internals data, you must also add the `debug_mode` tag in the `[Emulation]` section of the `xrt.ini`:

```
[Emulation]
debug_mode=batch
```

If you are collecting a large amount of trace data, you can increase the amount of available memory for capturing data by specifying the `--profile.trace_memory` option during `v++` linking, and add the `trace_buffer_size` keyword in the `xrt.ini`.

- **`--profile.trace_memory`:** Indicates what type of memory to use for capturing trace data.
- **`trace_buffer_size`:** Specifies the amount of memory to use for capturing the trace data during the application runtime.

TIP: *When `--profile.trace_memory` is not specified but `device_trace` is enabled in the xrt.ini File, the profile data is captured to the default platform memory with 1 MB allocated for the trace buffer size.*

Finally, as discussed in Continuous Trace Capture you can enable continuous trace capture to continuously offload device trace data while the application is running, so in the event of an application or system crash, some trace data is available to help debug the application.

Continuous Trace Capture

The Vitis tool supports recording continuous trace data while the application is running. The application can run for a very long time thus leading to the capture of significant trace data, which can result in issues like incomplete trace data especially when the memory resource used for trace data is not large enough. Using continuous trace, analysis of the trace can be carried out while the application is still running or if the application has crashed before completion.

With the ability to continuously capture trace data, the Timeline Trace reports can be dynamically updated in the Vitis analyzer tool while your application is running. Once these reports are loaded in Vitis Analyzer, there is a hyperlink available indicating that the current report is being modified on the disk. If new data needs to be loaded, **Reload** or **Auto-Reload** options are available on the banner to let you view the updated report as your application runs and trace data is generated.

Continuous trace is not enabled by default. Additionally, the memory resources of an FPGA are not unlimited. So if the application generates large trace data, a circular buffer for storing the data can be used. The circular buffer can be written, offloaded to the host, and reused again. By enabling a circular buffer with continuous trace, the memory resources needed are even smaller thus saving available resources on the device. However, an application run with continuous trace/circular buffer can result in multiple device trace files.

TIP: *For Hardware emulation, only host side continuous trace is available, for hardware runs both host side and device side continuous trace are available.*

Here are some scenarios where it is recommended to use the memory resource as a circular buffer.

The circular buffer implementation is automatically turned on when continuous trace is enabled in the `xrt.ini`. The flow requires the following settings for enabling continuous trace.

- In the `xrt.ini` file, `continuous_trace` is set to TRUE
- v++ linking option `--profile.trace_memory` is set to DDR or HBM

You can optionally set:

- The size of the trace buffer using `trace_buffer_size` in the `xrt.ini` file. This defaults to 1 MB.
- The interval at which the trace buffer is offloaded from the device using `trace_buffer_offload_interval_ms` in the `xrt.ini` file. The default is 10 ms.
- The interval at which files are dumped by setting `trace_file_dump_interval_s`. The default is 3 seconds.

IMPORTANT! *Circular Buffer can be force enabled by setting `trace_buffer_offload_interval_ms` to 0 ms.*

As an example, if you enable `continuous_trace` with `trace_buffer_size` as 8k and default `trace_buffer_offload_interval_ms` of 10 ms, the trace data rate is 819200 bytes/s which is less than the default of 100 MB/s. In this scenario, the circular buffer is *NOT enabled* by default and an XRT warning is reported:

```
[XRT] WARNING: Unable to use circular buffer for continuous trace offload.
Please increase trace buffer size and/or reduce continuous
trace interval. Minimum required offload rate (bytes per second) :
104857600 Requested offload rate : 819200
```

Here is an example of `xrt.ini` settings:

```
[Debug]
opencl_trace=true
device_trace=fine
stall_trace=all
continuous_trace=true
```

```
// The following are optional and needed only in rare circumstances

trace_buffer_size=20M
trace_buffer_offload_interval_ms=10
trace_file_dump_interval_s=2
```

The following are the results of these settings:

- `opencl_trace`: Enables the generation of host-related OpenCL API trace, `opencl_trace.csv` files is created.
- `device_trace`: Enables the collection of kernel activity to be added to profile summary and trace, `device_trace_0.csv` files are created with 0 being the device number.
- `stall_trace`: Enables the hardware generation of stalls into compute units.
- `continuous_trace`: Enables the continuous dumping of files for trace and the continuous reading of device data into the host.
- `trace_buffer_size`: Specifies the amount of memory to consume for trace data capture.
- `trace_buffer_offload_interval_ms`: Controls the reading of device data from the device to the host in milliseconds.
- `trace_file_dump_interval_s`: Controls the time between dumping of trace files in seconds.

As a result, there are several CSV files generated in addition to the `xrt.run_summary` as part of the application run using the above `xrt.ini` file. Vitis Analyzer only needs the generated `run_summary` file and will use the relevant CSV files to display the profile summary and timeline trace.

Here are the recommendations on setting up an application for trace data dumping:

1. By default the memory used for trace capture is the first memory resource on the platform, which can be determined using the platforminfo Utility in the *Vitis Reference Guide* (UG1702). In most platforms this is either DDR or HBM. The amount of memory reserved for trace data is determined by the `trace_buffer_size` switch in the `xrt.ini` file, which defaults to 1 MB.

   ***Note*:** You can also specify the use of FIFO and the size to allocate using the `--profile.trace_memory` option.

2. If still unable to dump maximum trace, disable stall trace by setting `stall_trace=off` or `stall_trace=on` with `device_trace=coarse`.
3. If the application requires larger size of trace buffer, enable circular buffer by setting `continuous_trace=true` with default settings of `trace_buffer_offload_interval_ms=10` and `trace_file_dump_interval_s=5`. Ideally, a continuous trace feature must be used for the following cases:
   - Long-running design with minimal trace generated

- Debugging application crashes where some `.csv` files might still be available for debugging

4. If the application run is still unable to dump the maximum trace, the `trace_buffer_size` can further be increased.
5. If the application still creates huge trace data that the host cannot keep up, use the smaller size of `trace_file_dump_interval`, which creates multiple files equivalent to the interval provided.
6. Lastly, continuous trace can generate several trace files as part of the application run in addition to `xrt.run_summary` file. The Vitis Analyzer only needs the generated `run_summary` file and can pick the relevant CSV files generated to display profile summary and timeline trace to provide a better experience.

Custom Profiling of the Host Application

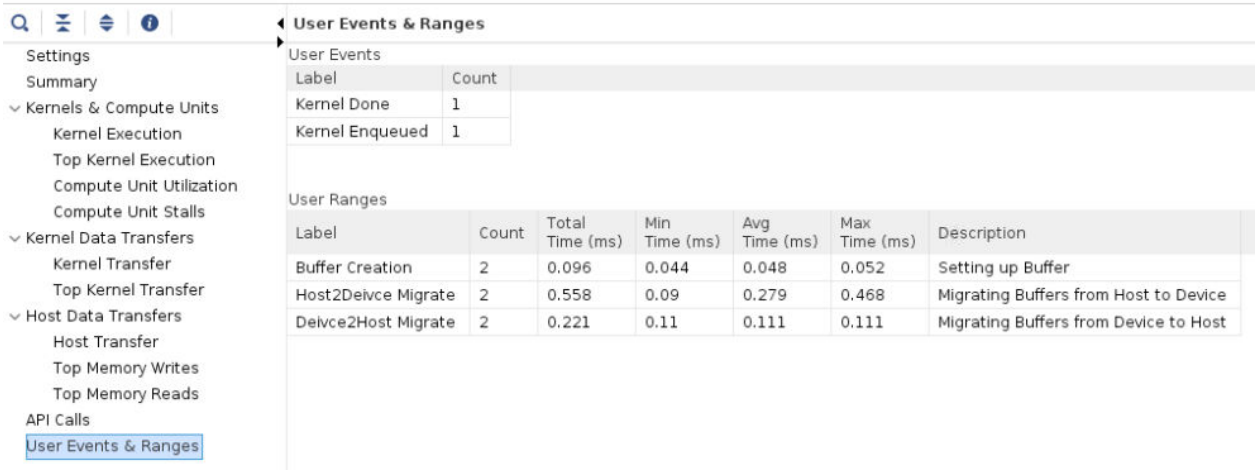
All XRT related actions from the host application are automatically tracked for profiling, through either theOpenCL API calls, or the XRT API calls. However, you can also profile the host application beyond the XRT related events, capturing event data based on user-specified actions or events.

This feature provides two types of custom profiling:

- **User range:** Profiles the specified start/end times across a range of code. This captures the span of time within which an action occurs in the host application.
- **User events:** Marks an event in the timeline. The user event is added to the timeline waveform at whatever point in time it occurs.

The `user_range` and `user_event` data can be captured to the Profile Summary and Timeline Trace reports for display in Vitis analyzer. As seen in the figure below, the Profile Summary shows the number of occurrences of a given event and the range. The User Ranges table also reports the Min/Max/Avg/Total duration of the user-defined ranges in the host code. In the Timeline Trace report `user_range` elements in the host code are displayed in a separate row, and `user_event` markers are added at specific points on the timeline.

Figure 37: **Profile Summary – User Range**



Using custom profiling requires a few changes in your host application source code and build process. You must make use of C or C++ API in your code, as described below, and you must include the `xrt_coreutil` library when linking your host application.

- The C/C++ API are described below, but can also be found at the following URL: https://github.com/Xilinx/XRT/blob/master/src/runtime_src/core/include/experimental/xrt_profile.h.
- For both C and C++ you must add the following:

```
#include experimental/xrt_profile.h
```

- When linking host code, add `-lxrt_coreutil` to the compiler command line.

Profiling of C++ Code

For C++ code the provided objects are:

- **`user_range`:** This object captures the start time and end time of a measured range of activity with the specified ID. The object constructor is:

```
user_range(const char* label, const char* tooltip);
```

- **`user_event`:** This object marks an event occurring at single point in time, adding the specified label onto the timeline trace. The object constructor is:

```
user_event()
```

Use the `user_range` to construct an object and start keeping track of time immediately upon construction. Usage details of the `user_range` objects:

- If a `user_range` is instantiated using the default constructor, no time is marked until the user calls `user_range.start()` with the label and tooltip.

- You can instantiate a `user_range` object passing the label and tooltip strings. This starts monitoring the range immediately.
- You must call `user_range.start()` and `user_range.end()` to capture ranges of time you are interested in.
- If `user_range.end()` is not called, then any range being tracked lasts until the `user_range` object is destructed.
- The `user_range` object can be reused any number of times, by calling `user_range.start()`/`user_range.end()` pairs in the host code.
- Sequential calls to `user_range.start()` ignore all but the first call until `user_range.end()` terminates the range.
- Sequential calls to `user_range.end()` ignore all but the first call until `user_range.start()` starts a new range.

Usage of the `user_event` objects:

- A `user_event` object must be instantiated using the default constructor.
- Calls to `user_event.mark()` creates a user marker on the timeline trace at that particular time.
- `user_event.mark()` takes an optional `const char*` argument which appears as a label on the timeline trace.

With your host application properly instrumented, XRT can capture profile data from these user-defined ranges and events, in addition to the standard XRT API-based events. You must enable profiling in the`xrt.ini` file as explained previously.

Profiling of C Code

For C code the provided functions are:

- **`xrtURStart()`:** This function establishes the start time of a measured range of activity with the specified ID. The function signature is:

```
void xrtURStart(unsigned int id, const char* label, const char* tooltip)
```

- **`xrtUREnd()`:** This function marks the end time of a measured range with the specified ID. The function signature is:

```
void xrtUREnd(unsigned int id)
```

- **`xrtUEMark()`:** This function marks an event occurring at single point in time, adding the specified label onto the timeline trace. The function signature is:

```
void xrtUEMark(const char* label)
```

Use the `xrtURStart()` and `xrtUREnd()` functions to start keeping track of time immediately, and specify an ID to pair the start/end calls and define the user range. Usage details of the `user_range` functions:

- Start/End ranges of one ID can be nested inside other Start/End ranges of a different ID.
- It is your responsibility to make sure the IDs match for the Start/End range you are profiling.

> **IMPORTANT!** *Multiple calls to `xrtURStart` and `xrtUREnd` with the same ID can cause unexpected behavior.*

- The user range can have a label that is added to the timeline, and a tooltip that is displayed when you place the cursor over the user range.

A call to `xrtUEMark()` will create a user marker on the timeline trace at the point of the event.

- `xrtUEMark()` lets you specify a label for the event. The label will appear on the timeline with the mark.
- You can use `NULL` for the label to add an unlabeled mark.

The following is example code:

```
int main(int argc, char* argv[]) {
 58
 59   xrtURStart(0, "Software execution", "Whole program execution") ;
 60 ...
 61   //TARGET_DEVICE macro needs to be passed from gcc command line
 62   if(argc != 2) {
 63       std::cout << "Usage: " << argv[0] <<" <xclbin>" << std::endl;
 64       return EXIT_FAILURE;
 65    }
....
153     q.enqueueTask(krnl_vector_add);
154
155     // The result of the previous kernel execution will need to be
retrieved in
156     // order to view the results. This call will transfer the data from
FPGA to
157     // source_results vector
158
q.enqueueMigrateMemObjects({buffer_result},CL_MIGRATE_MEM_OBJECT_HOST);
159 ····
160     q.finish();
161
162     xrtUEMark("Starting verification") ;
163
```

Enabling Low Overhead Profiling

The Vitis software platform supports low overhead profiling that provides minimal information with little effect on execution time. Using this option during runtime, the timeline trace is still available but with a reduced amount of information. Low overhead profiling captures minimal information on OpenCL events and dumps a CSV file called `lop_trace.csv` at the end of execution. Low overhead profiling can be run in both hardware and hardware emulation.

To enable low overhead profiling, there is a new flag in the "Debug" section of the xrt.ini File called `lop_trace`. By default, `lop_trace` is FALSE and must be enabled by setting the `ini` parameter to TRUE.

```
xrt.ini file
[Debug]
lop_trace=true
```

When `lop_trace=true` is enabled, the runtime will generate `lop_trace.csv` which can be viewed in the Run Summary within Vitis analyzer.

```
vitis_analyzer xrt.run_summary
```

To obtain the lowest possible overhead, information collected in normal OpenCL profiling is omitted. Specifically, the following information is expected to not be available in the low overhead profiling trace:

- Device events, such as compute unit executions or kernel memory transfers
- Information about memory reads or writes, such as destination address or size
- Information about kernel enqueues, such as kernel name or NDRange sizes
- Dependencies between buffer transfers and kernel enqueue

Enabling No Overhead Profiling

When profiling is enabled in `xrt.ini` with `opencl_trace` there is operational overhead added and events in the timeline can show longer delay in between events. However, XRT provides a no-overhead option to dump the OpenCL events on the timeline. It is a simple method of displaying events on the timeline without any overhead.

TIP: *You cannot specify any host side profiling or this overrides the no-overhead approach. However, you can use this approach with `device_trace`, which profiles the device but does not add overhead to the host application.*

Because the goal is to provide visibility into events with absolutely zero overhead, there are limitations to the number of events that can be logged. Additionally, there is no command queue information in this view, so this view is not intended as a replacement of the more detailed Timeline Trace.

You can use the no overhead view to confirm OpenCL command dependencies and to observe actual event overhead for the command execution from the host application.

Add the following switch in `xrt.ini` to enable OpenCL events. The beginning and end of event capturing can be controlled as shown below:

```
[Debug]
xocl_debug=true
#xocl_event_begin= 0 (default)
#xocl_event_end=1000 (default)
```

By default only 1000 events can be visualized.

After the run, if no `xrt.run_summary` is generated, you can use the following steps to generate a `.wdb` file to view in Vitis Analyzer:

```
vp_analyze xocl -i xocl.log // generates debug_log.csv
vp_analyze trace -i debug_log.csv // generates debug_log.wdb
vitis_analyzer debug_log.wdb // loads the wdb file in Vitis analyzer
```

Figure 38: **No Overhead Timeline**

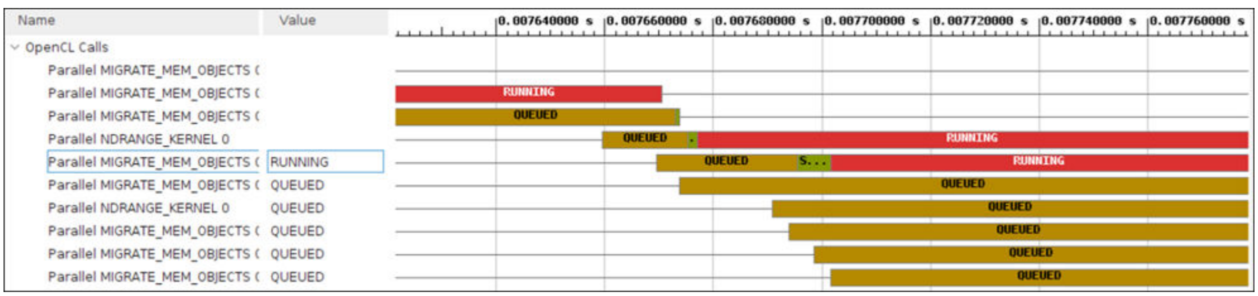


Table 14: **Options for OpenCL Profiling**

Trace Information Captured	Profile Overhead	Use Case	xrt.ini Switches
Complete	High	For debug purposes like the early stage of application development.	`opencl_trace = true`
Partial	Low	When profile overhead is High and unexpected delay is experienced.	`lop_trace = true`
Minimum	No	Only to confirm the cause of delay between events.	`xocl_debug = true`

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. Therefore, these rules might not be applicable for your specific design. 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 Vitis HLS, Vitis profiler, and AMD Vivado™ Design Suite when invoked by the `v++` compiler. The generated design guidance can have several severity levels: errors, warning messages, and informational messages are provided during hardware emulation, and system builds. The profile design guidance helps you interpret the profiling results which allows you to focus on improving performance.

Guidance includes message text for reported violations, a brief suggested resolution, and a detailed resolution provided as a web link. You can determine your next course of action based on the suggested resolution. This helps improves productivity by quickly highlighting issues and directing you to additional information in using the Vitis technology.

Design guidance is automatically generated after building or running an application from the command line or Vitis unified IDE.

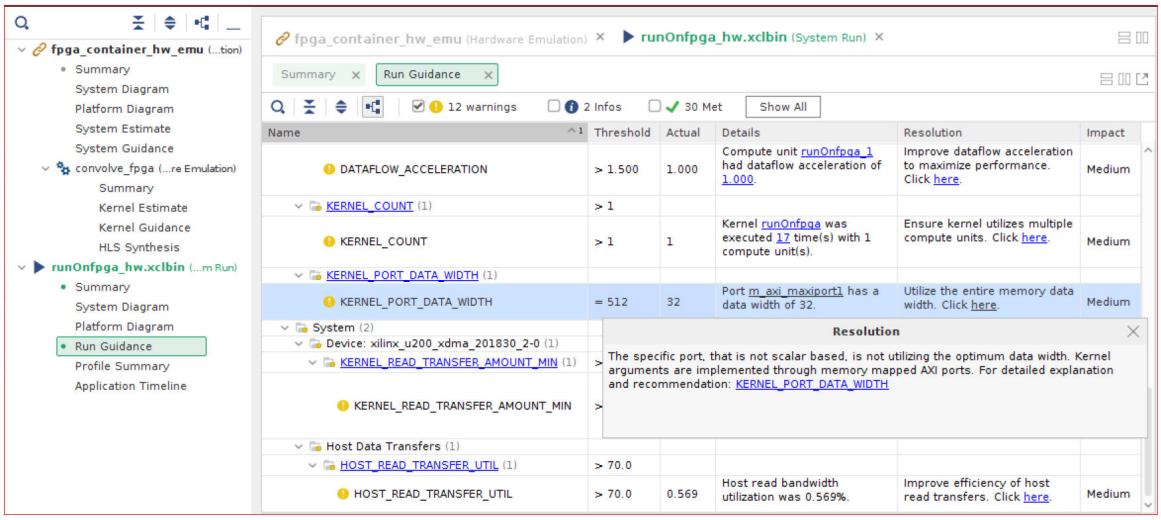
You can open the Guidance report as described in Working with the Analysis View (Vitis Analyzer) in the *Vitis Reference Guide* (UG1702). To access the Guidance report, open the Compile Summary, the Link Summary, or the Run Summary, and open the Guidance report.

- Kernel Guidance is generated by the Vitis HLS tool after kernel is built using `v++` compile command. This can be viewed in the Vitis analyzer by opening the Compile Summary report. Kernel guidance and Compile Summary files are generated for each kernel compiled. Kernel guidance includes recommendations on using Dataflow; and possible reasons why the expected throughout could not be achieved.
- System Guidance is generated after the .xclbin or .xsa is built using the `v++` link command. This can be viewed in the Vitis analyzer by opening the Link Summary report. System guidance includes all Kernel Guidance checks, and provides comprehensive review before running your application.
- Run Guidance is generated when your generated `.xclbin` is run, and is a feature of the XRT. This can be viewed by opening the Run Summary in the Vitis analyzer. Run Guidance includes checks like if Kernel Stall is above 50%, recommendations if PLRAM can be used instead of DDR memory, etc.

With the Guidance report open, the Guidance view displays the messages along with resolution columns. The resolutions also have extended weblink help available.

The following image shows an example of the Guidance report displayed in the Vitis analyzer. For example, clicking a link in the Name column opens a description of the rule check. Links in the Details column can open source code, select a design object such as a kernel, or navigate to another report.

Figure 39: **Design Guidance Example**



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

You can click the web link in the Resolution column to get additional details about the resolution.

Kernel and Compute Unit objects, in addition to profile reported data values, can also be cross-probed to other views like the Profile Report.

Opening the Guidance Report

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 summaries, 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 Working with the Analysis View (Vitis Analyzer) in the *Vitis Reference Guide* (UG1702) for more information.

Interpreting Guidance Data

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.
- **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 describes 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 popup 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 AMD 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.

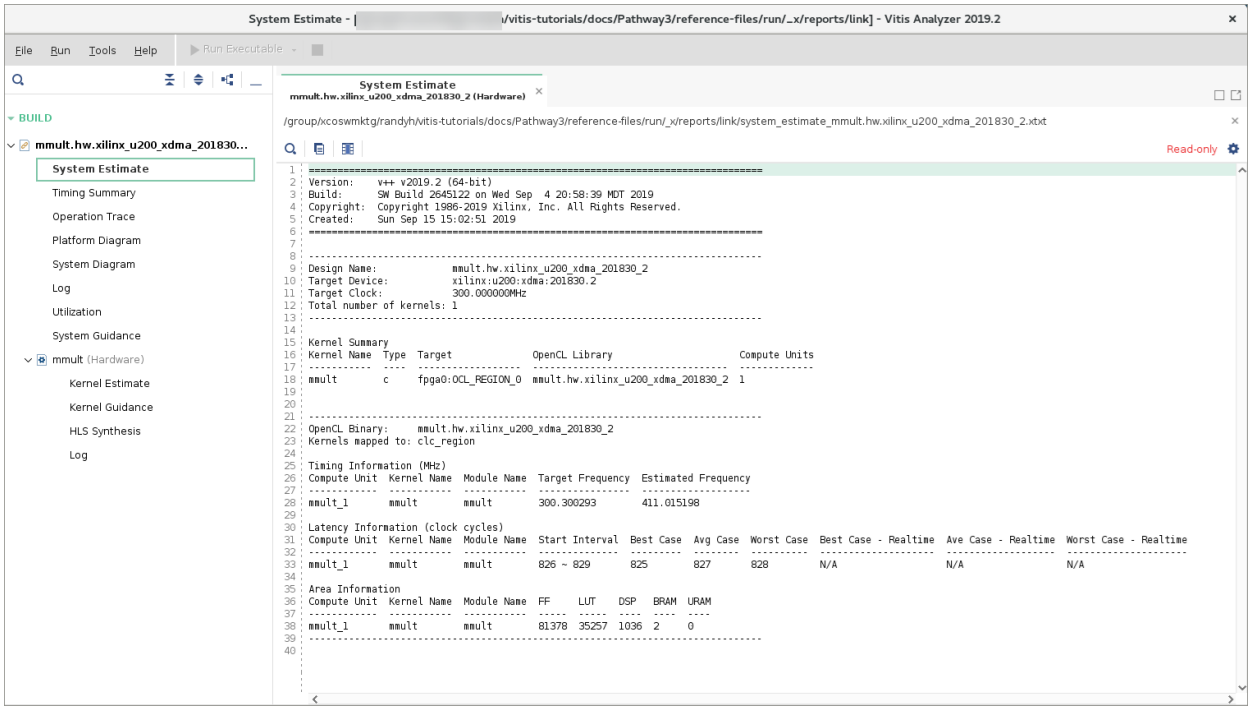
The System Estimate report provides estimates of FPGA resource usage and the estimated frequency at which the hardware accelerated kernels can operate. The report is automatically generated for hardware emulation and system hardware builds. The report contains high-level details of the user kernels, 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 40: **System Estimate**



Opening the System Estimate Report

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 library 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 (see Working with the Analysis View (Vitis Analyzer) in the *Vitis Reference Guide* (UG1702)).

Interpreting the System Estimate Report

The System Estimate report generated by the `v++` command provides information on every binary container in the application, in addition to 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 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 AMD 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, the CU must 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:

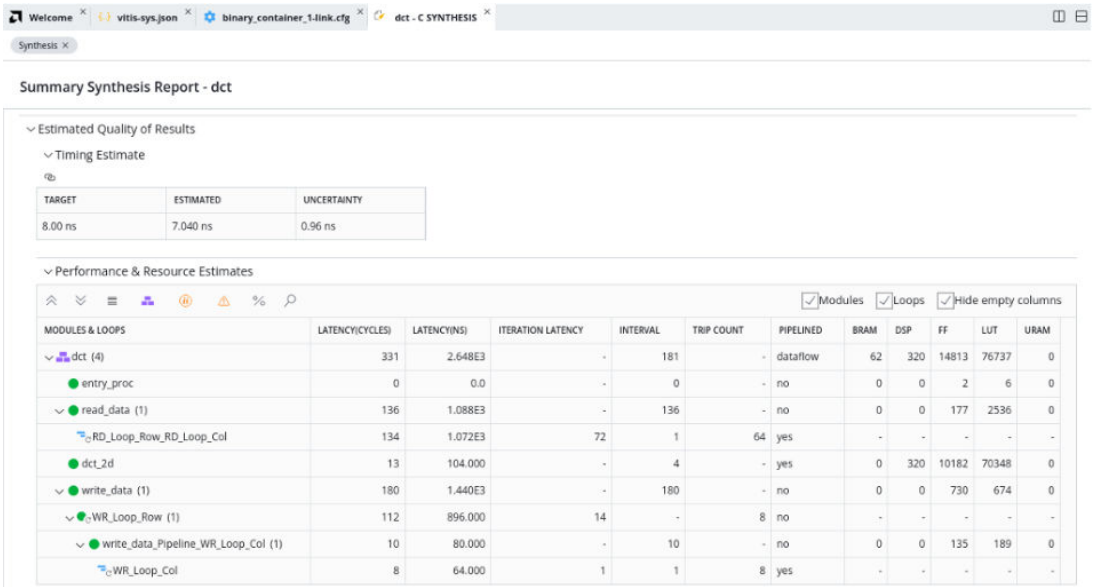
```
Area Information
Compute Unit  Kernel Name  Module Name  FF     LUT    DSP   BRAM  URAM
------------  -----------  -----------  -----  -----  ----  ----  ----
mmult_1       mmult        mmult        81378  35257  1036  2     0
-----------------------------------------------------------------------
```

HLS Synthesis Report

The HLS compiler generates a number of reports for simulation, synthesis, co-simulation. These reports provide details about the high-level synthesis (HLS) compilation of a PL kernel. The main report is the Synthesis Summary report that 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 Vitis unified IDE, this report can be found in the HLS component directory named `<hls_component>.hlscompile_summary`. The Summary report can be opened from the Flow Navigator in the HLS component under the C Synthesis/Reports heading, or by opening the Compile Summary, or the Link Summary as described in Working with the Analysis View (Vitis Analyzer) in the *Vitis Reference Guide* (UG1702).

Figure 41: **Synthesis Summary Report**



Generating and Opening the HLS Report

IMPORTANT! *You must specify the `--save-temps` option during the build process to preserve the intermediate files produced by Vitis HLS, including the reports. The HLS report and HLS guidance are only generated for hardware emulation and system builds for C and OpenCL kernels.*

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. Refer to Working with the Analysis View (Vitis Analyzer) in the *Vitis Reference Guide* (UG1702) for more information.

Interpreting the HLS Report

The HLS Synthesis report is a spreadsheet listing the module hierarchy in the left column. This section is describing one section of the HLS report: Performance and Resource Estimates. Each module and loop generated by the HLS run is represented in this hierarchy. The HLS Synthesis report contains the following columns:

- Issue Type
- Slack
- Latency in clock cycles
- Latency in absolute time (ns)
- Iteration Latency
- Interval
- Trip Count
- Pipelined
- Utilization Estimates of Block RAM, DSP, FF, and LUT

If this information is part of a hierarchical block, it can 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! *The absolute counts of cycles and latency numbers are based on estimates identified during HLS synthesis, especially with advanced transformations, such as pipelining and dataflow. Therefore, 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

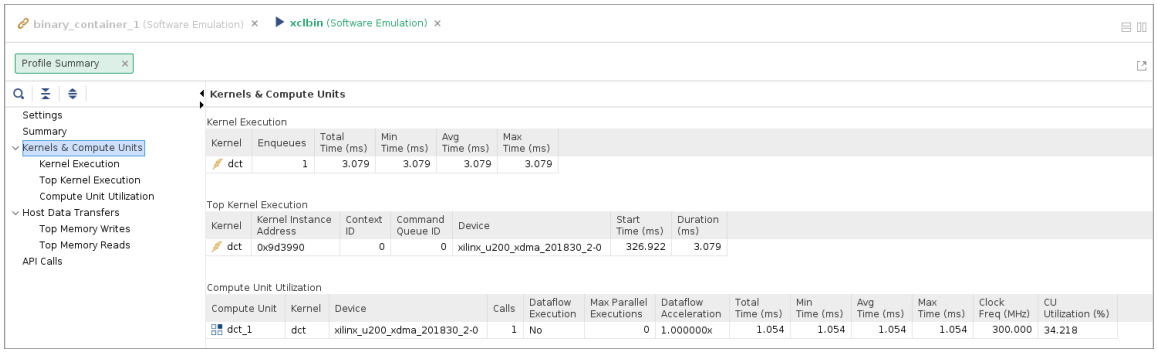
As described in Enabling Profiling in Your Application, the Xilinx Runtime (XRT) collects profiling data on host applications and kernels when specific options are enabled in the `xrt.ini` file, such as `opencl_trace`, `xrt_native_api`, and `device_trace`. XRT captures profiling data for the host application as it makes calls to the runtime either through OpenCL or XRT API calls. You can also add user calls to your host application to capture additional profiling information, as explained in Custom Profiling of the Host Application. To capture details of the kernel operations, you must implement kernels in the `.xclbin` using the --profile Options as explained in the next section.

After the application finishes running, the Profile Summary report is saved as `.csv` files in the directory where the compiled host code is executed. 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 the kernel execution and data transfer statistics.

TIP: *The Profile Summary report can be generated for all build configurations. The data transfer details under kernel execution efficiency and data transfer efficiency is only generated in hardware emulation or system builds.*

An example of the Profile Summary report is shown below.

Figure 42: **Profile Summary**



Generating and Opening the Profile Summary Report

Capturing the data required for the Profile Summary requires a few steps prior to actually running the application.

1. The FPGA binary (`xclbin`) file is configured for capturing profiling data by default. However, using the `v++ --profile` option during the linking process enables a greater level of detail in the profiling data captured. For more information, see the --profile Options.

2. The runtime requires the presence of an `xrt.ini` file, as described in xrt.ini File, that includes options for capturing trace data:

```
[Debug]
opencl_trace = true
xrt_native_api = true
device_trace = fine
```

3. To enable the profiling of Kernel Internals data, you must also add the `debug_mode` tag in the `[Emulation]` section of the `xrt.ini`:

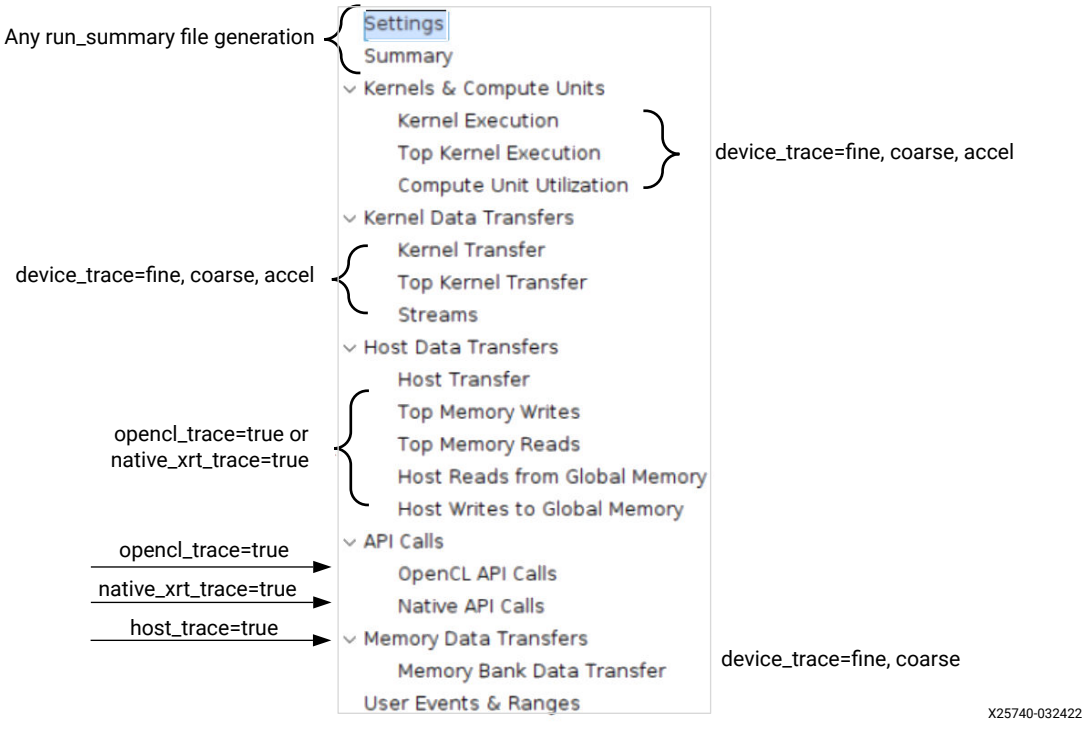
```
[Emulation]
debug_mode = batch
```

With profiling enabled in the device binary and in the `xrt.ini` file, the runtime creates different report files when running the application depending on which options are enabled. The Profile Summary report can be viewed in the Working with the Analysis View (Vitis Analyzer) in the *Vitis Reference Guide* (UG1702) by opening the Run Summary using the following command:

```
vitis_analyzer xrt.run_summary
```

As previously stated, the data contained in the Profile Summary report depends on the various options enabled in the `xrt.ini` file. The following figure illustrates some of the available data tables and the options to enable them.

Figure 43: **Profile Summary Tables**



Related Information

Running the System on Hardware
Working with the Analysis View (Vitis Analyzer) in the *Vitis Reference Guide* (UG1702)

Interpreting the Profile Summary

The profile summary includes a number of useful statistics for your host application and kernels. The report provides a general idea of the functional bottlenecks in your application.

TIP: *When viewing a table in the Analysis view, you can hover your mouse over nay field to get a definition of the field contents.*

Settings

This displays the report and XRT configuration settings.

Summary

This displays summary statistics including device execution time and device power.

Kernels & Compute Units

Displays the profile summary data for all kernel functions scheduled and executed.

Kernel Data Transfers

Displays the data transfer for kernels to the global memory, and the top data transfer for kernels to the global memory, and the data transfer streams.

Host Data Transfers

Displays profile data for all write transfers between the host and device memory through PCI Express® link, and profile data for all read transfers between the host and device memory through PCI Express® link, and the data transfer for host to the global memory.

API Calls

Displays the profile data for all OpenCL host API function calls executed in the host application. The top displays a bar graph of the API call time as a percent of total time.

Device Power

This displays the profile data for device power.

Kernel Internals

Displays the running time for compute units in microseconds (µs) and reports stall time as a percent of the running time. This section of the Profile Summary displays the data transfer for specific ports on the compute unit, and displays the functional port data transfers on the compute unit, and displays the running time and stalls on the compute unit.

TIP: *The Kernel Internals tab reports time in µs, while the rest of the Profile Summary reports time in milliseconds (ms).*

Shell Data Transfers

This following table displays the DMA data transfers.

TIP: *For DMA bypass and Global Memory to Global Memory data transfers, see the DMA Data Transfer table in Kernel Internals.*

Timeline Trace

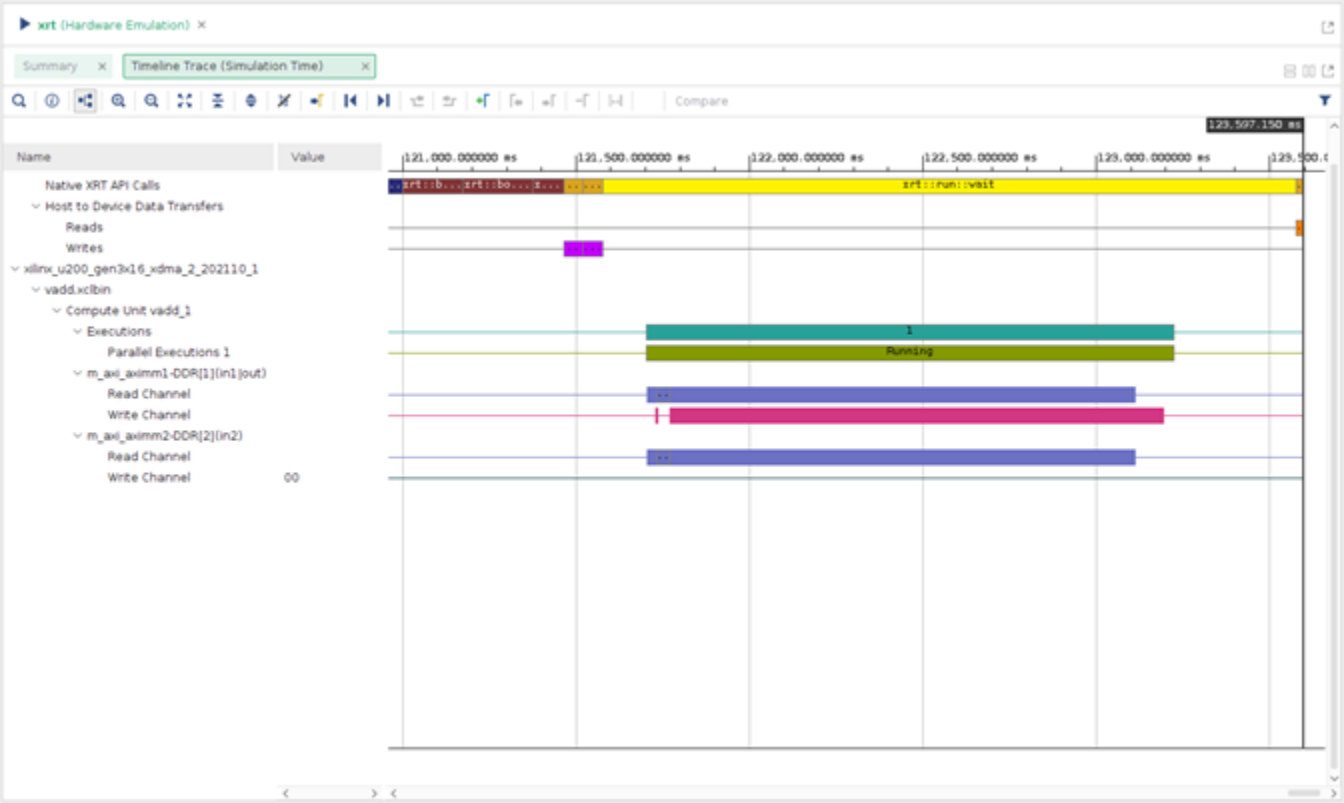
The Timeline Trace collects and displays host and kernel events on a common timeline to help you understand and visualize the overall health and performance of your systems. The graphical representation lets you see issues regarding kernel synchronization and efficient concurrent execution. The displayed events include:

- XRT API calls from the host code
- Device trace data including compute units, AXI transaction start/stop
- Host events and kernel start/stops

While the timeline and device trace data are useful for debugging and profiling the application, collecting the data can affect application performance by adding time to the execution. However, the trace data is collected with dedicated resources in the kernel, and so does not affect kernel functionality. By default, the collected trace data is offloaded at the end of the run, though the tool can be configured to offload data continuously which can help when debugging application crashes.

The following is a snapshot of the Timeline Trace 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: **Timeline Trace**



Timeline data can be enabled and collected through the command line flow. However, viewing must be done in the Vitis analyzer as described in Working with the Analysis View (Vitis Analyzer) in the *Vitis Reference Guide* (UG1702).

Generating and Opening the Timeline Trace

To generate the Timeline Trace 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` option as described in --profile Options. As an example, add `--profile.data` to the `v++` linking command line:

   ```
   v++ -g -l --profile.data all:all:all ...
   ```

2. After the kernels are instrumented during the build process, data gathering must also be enabled during the runtime execution of the application by editing the `xrt.ini` file. Refer to xrt.ini File for more information.

The following `xrt.ini` file enables maximum information gathering when the application is run:

```
[Debug]
opencl_trace=true
device_trace=fine
stall_trace=all
```

> **TIP:** *If you are collecting a large amount of trace data, you might need to use the `--profile.trace_memory` with the `v++` command, and the `trace_buffer_size` keyword in the `xrt.ini`.*

After running the application, the Timeline Trace data is captured in CSV files called `opencl_trace.csv` and `device_trace_0.csv`.

3. The CSV report can be viewed in the Vitis analyzer tool by opening the Run Summary produced during the application execution. You can launch the Vitis analyzer and open the Run Summary using the following command:

```
vitis_analyzer xrt.run_summary
```

> **TIP:** *By default, the Timeline Trace is displayed in a hierarchical view, which presents the information according to design hierarchy but consumes significant real estate in the display. As an alternative, you can "flatten" the timeline display to eliminate unnecessary spacing between lines. Perform this by selecting the **Flatten Signal** command on the toolbar. This feature is useful when there is less display area to work with, or when comparing multiple trace files.*

Interpreting the Timeline Trace

The Timeline Trace 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 Timeline Trace 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 must 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 must 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.
    - **Compute Unit "name":** Name of the compute unit on the FPGA.
      - **User Functions:** In the case of the Vitis 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"**

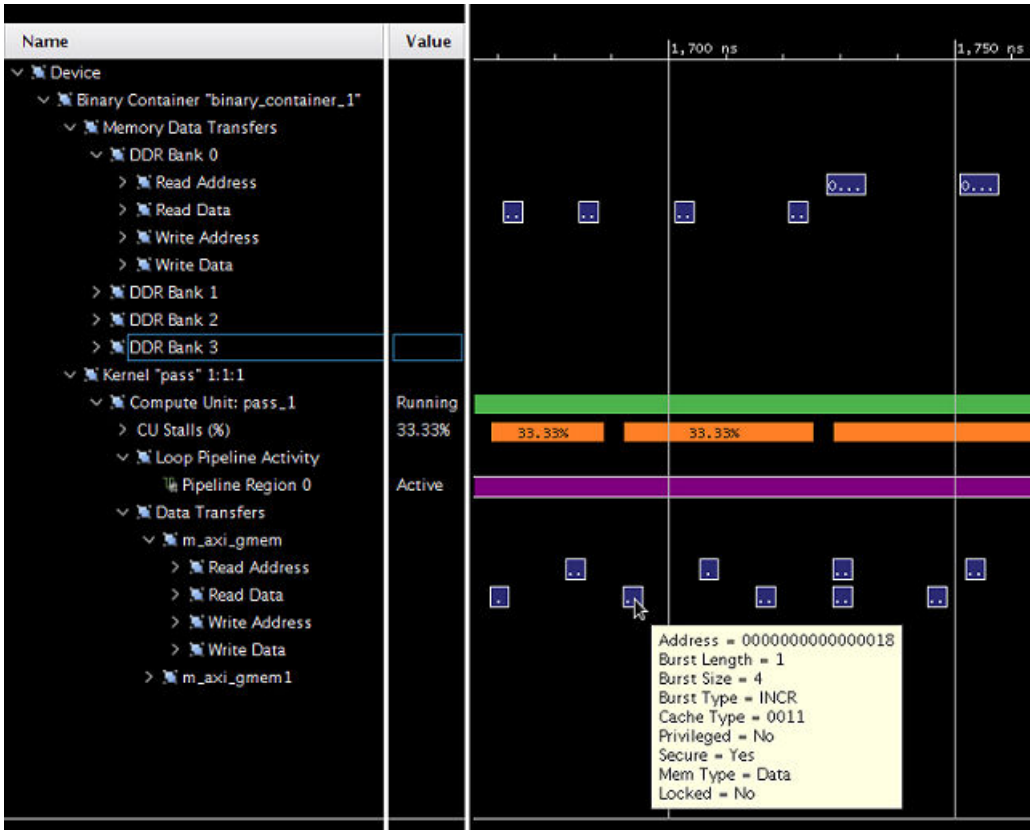
      - **Read:** A CU reads from the DDR over AXI-MM ports. The trace of a data read by a CU is shown here. The activity is shown as transaction and the tool-tip for each transaction shows more details of the AXI transaction. These names are generated when `--profile.data` is for the CU.

- **Write:** A CU 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.data` is specified for the CU.

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, in addition to 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 45: **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 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 and burst sizes considerably less than 512 bits are therefore good opportunities to optimize interface performance.

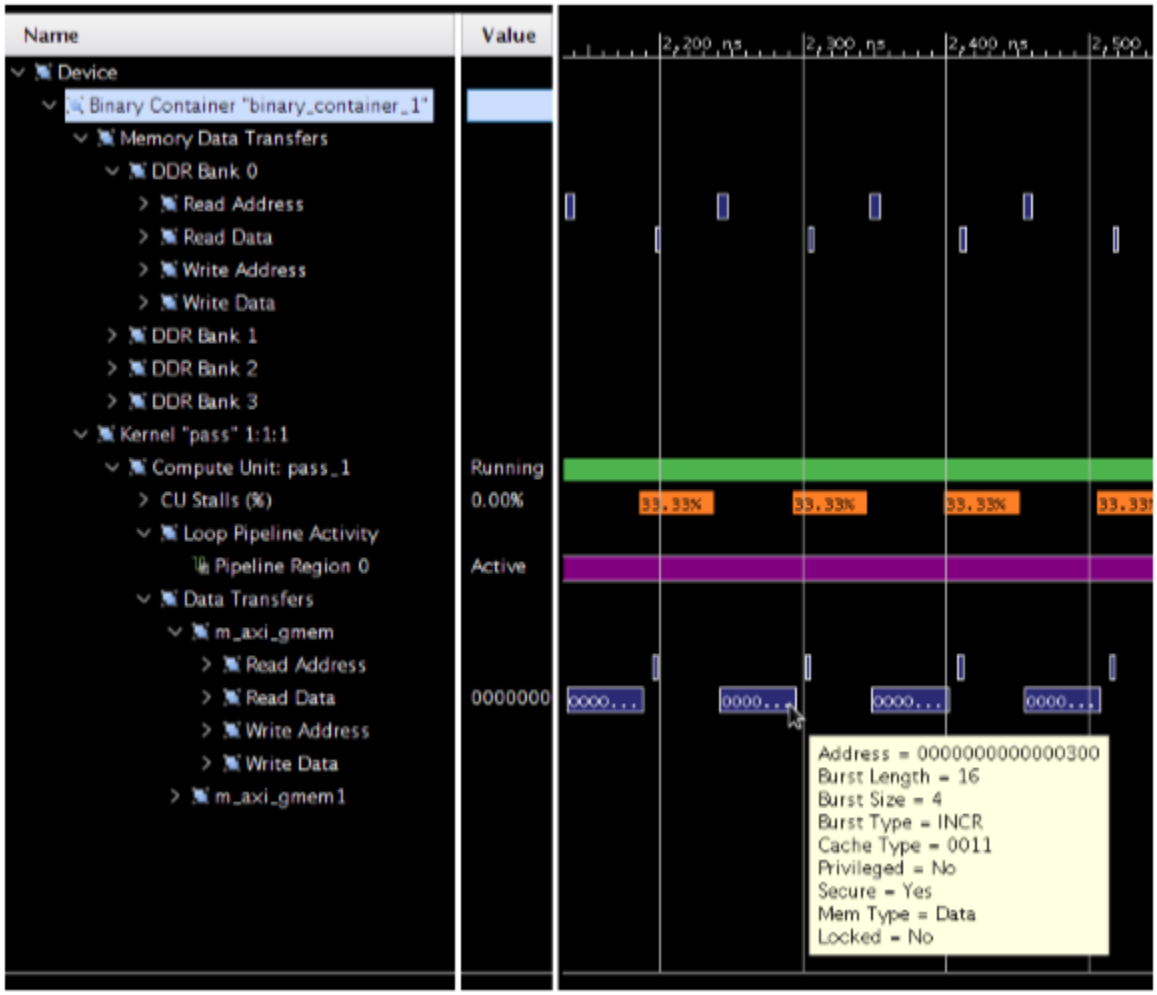
Using Burst Data Transfers

Transferring data in bursts hides the memory access latency and improves bandwidth usage and efficiency of the memory controller. Also, check the HLS report for bursting information.

RECOMMENDED: *Infer burst transfers from successive requests of data from consecutive address locations. Refer to the topic AXI Burst Transfers in the Vitis HLS User Guide (UG1399) 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 46: **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 16 x 4 byte transaction into a single package transaction of 1 x 64 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. For more information on burst, see AXI Burst Transfers in the Vitis HLS User Guide (UG1399).

The isolation of the read and write function from the computation results in:

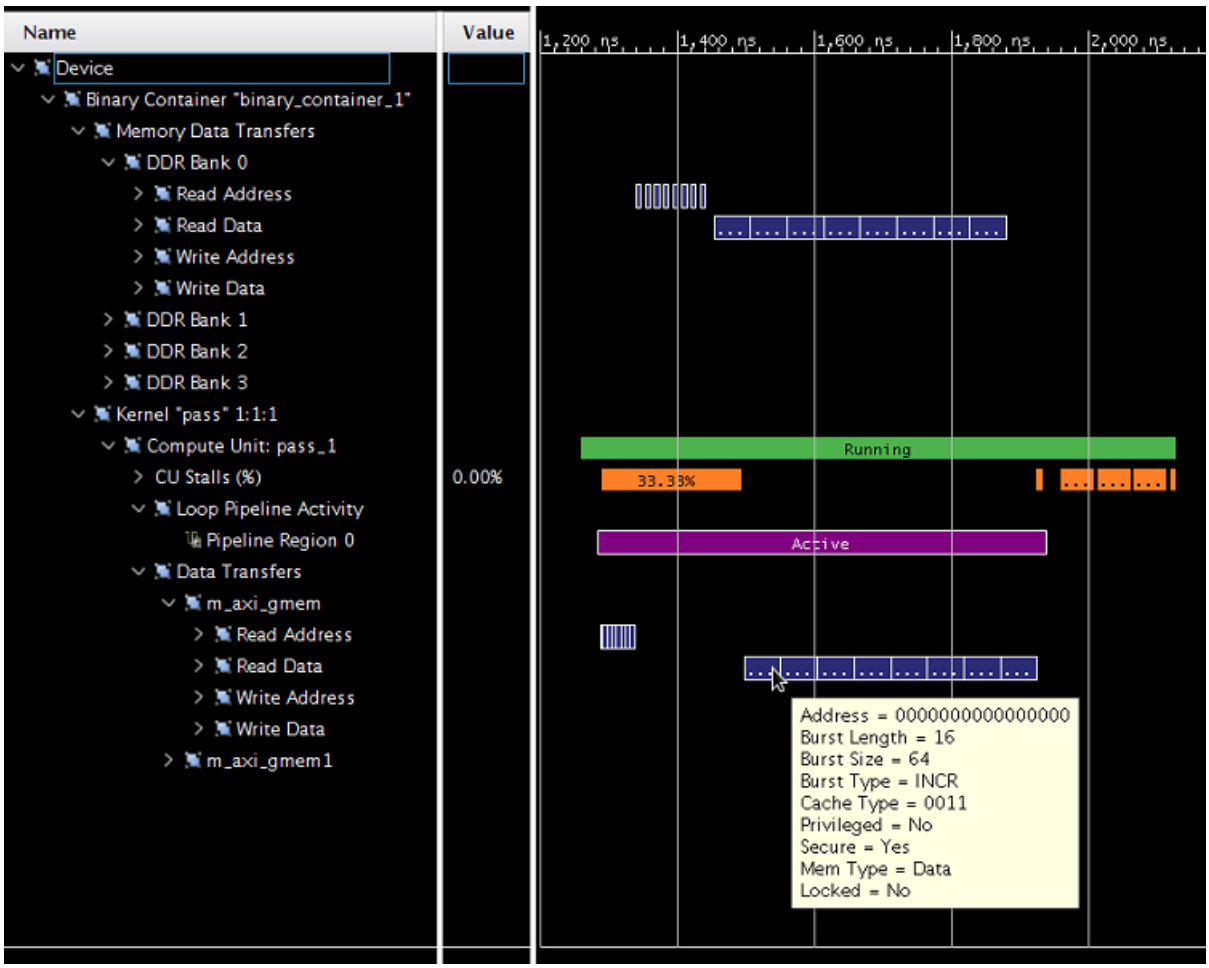
- Simple control structures (loops) in the read/write function which makes burst detection simpler.
- The isolation of the computational function away from the AXI interfaces, simplifies potential kernel optimization.
- The internal variables are mapped to on-chip memory, which allow faster access compared 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, AMD recommends that you choose data types that map to the full data width on the memory controller. The memory controller in all supported accelerator cards supports 512-bit user interface, which can be mapped to C/C++ arbitrary precision data type `ap_int<512>` or OpenCL vector data types such as `int16`.

The default is for Vitis HLS to automatically re-size the kernel interface ports up to 512-bits to improve burst access. 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 47: **Burst AXI Transactions**



This example shows good interface configuration as it maximizes AXI data width, in addition to 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.*

Waveform View and Live Waveform Viewer

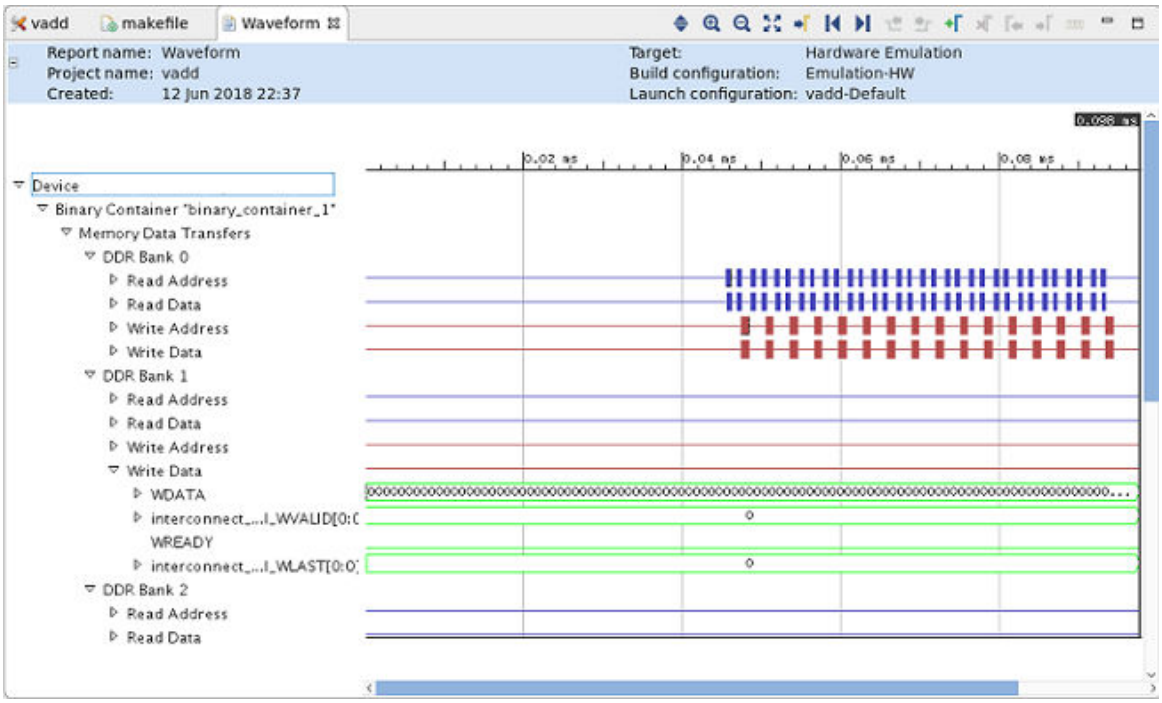
The Vitis core development kit can generate a Waveform view when running hardware emulation. It displays in-depth details at the system-level, CU level, and at the function level. The details include data transfers between the kernel and global memory and data flow through inter-kernel pipes. These details 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 (see Working with the Analysis View (Vitis Analyzer) in the *Vitis Reference Guide* (UG1702)). In contrast, the Live Waveform Viewer opens the Vivado simulation waveform viewer to examine the hardware transactions in addition to any user selected signals.

Waveform data is 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 Generating and Opening the Waveform Reports for instructions on enabling these features.

Figure 48: **Waveform View**



You can also open the waveform database (`.wdb`) file with the Vivado logic simulator through the Linux command line:

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

TIP: *The `.wdb` file is written to the directory where the compiled host code is executed.*

Generating and Opening the Waveform Reports

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).

   ```
   [Emulation]
   debug_mode=batch
   ```

   The `debug_mode=batch` enables the capture of waveform data (`.wdb`) by running simulation in batch mode. You can also enable the Live Waveform Viewer to launch simulation in interactive mode using the following setting in the `xrt.ini`.

   ```
   [Emulation]
   debug_mode=gui
   ```

   TIP: *If Live Waveform Viewer is enabled, the simulation waveform opens during the hardware emulation run.*

3. Run the hardware emulation build of the application as described in Chapter 5: Application Verification Using Vitis Emulation Flow. The hardware transaction data is collected in the waveform database file, `<hardware_platform>-<device_id>-<xclbin_name>.wdb`. Refer to Output Directories of the v++ Command in the *Vitis Reference Guide* (UG1702) for more information on locating these reports.

4. Open the Waveform view in the Vitis analyzer by opening the Run Summary, and opening the Waveform report.

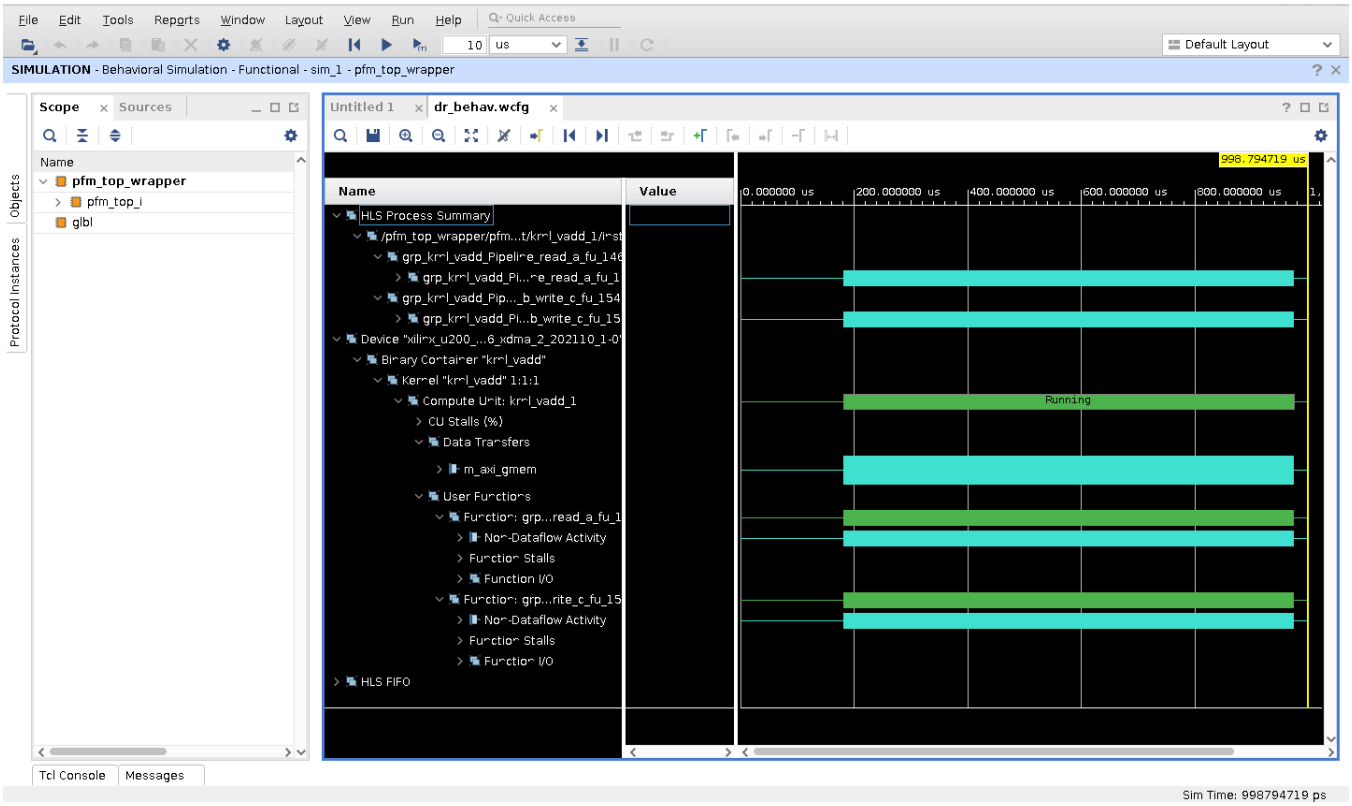
   ```
   vitis_analyzer xrt.run_summary
   ```

5. Waveforms for TLM transactions can also be dumped for third-party simulators (support limited to Mentor Graphics Questa Advanced Simulator and Cadence Xcelium). Wave data dump is enabled when `v++` link is done with `-g` option (as mentioned in step 1). The format of the wave database dumped is simulator specific (for example, `.wlf` for Questa Advanced Simulator and `.shm` for Xcelium).

Interpreting Data in the Waveform Views

The following image shows the Waveform view:

Figure 49: **Waveform View**



The Waveform and Live Waveform views are organized hierarchically for easy navigation.

- The Waveform view 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, because the Waveform view is generated from the post-processed data, no additional signals can be added to the report, and some of the runtime analysis cannot be visualized, such as DATAFLOW transactions.
- The Live Waveform viewer is displaying the Vivado logic simulator (`xsim`) run, so you can add extra signals and internals of the register transfer (RTL) design to the live view. Refer to the *Vivado Design Suite User Guide: Logic Simulation* (UG900) for information on working with the Waveform viewer.

The hierarchy of the Waveform and Live Waveform views include the following:

- **HLS Process Summary:**
  - Hierarchical view of processes and their sub-processes corresponding to the user functions in CU
  - Entry for each kernel instance which has processes modeling user function in it (entries can be unfolded to show the processes in it)
  - Handshake transactions on all the processes corresponding to user-functions

- Both dataflow and non-dataflow/non-piplelined processes
- The transactions on the processes including stalls, are shown using the corresponding protocol analyzer instance
- Allows you to get an overview about the usage of the individual processes over the execution time (similar to C/C++ profile capabilities)

- **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 Vitis HLS to inform you when a portion of the circuit is stalling because of external memory accesses, or internal streams (that is, dataflow). 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 shows the related stalls.
  - **Intra-Kernel Dataflow:** FIFO activity internal to the kernel.
  - **Function I/O:** Actual interface signals.
- **HLS FIFO:**
  - This shows waveform for size of HLS FIFOs created inside non-RTL kernels
  - The waveform is in Analog style
  - It shows one entry for each kernel instance which has FIFO in it
  - The analog waveform is produced by tracing on an internal HDL signal of the kernel which gives the current number of elements in the FIFO during simulation.
  - **CU name:** Name of the CU containing FIFO
  - **FIFO instance name:** Name of the FIFO instanc
  - **mOutPtr["size:"0]:** HDL signal which gives the number of elements currently in the FIFO during simulation

Interpreting TLM Waveform Data for Third-Party Simulators

1. Under the respective design hierarchy in the waveform windows, for each TLM—TLM socket connection, the following information is visible as waveforms.
   a. For memory mapped AXI4 interfaces, the bus transaction is visible as six channels:
      - `<socket_name>`: This channel contains statistics such as whether transaction is read/write and a unique mark to differentiate information in sub channels. This also indicates the number of transactions completed.
      - `<socket_name>_AW`: This channel contains the transaction information of Write Address.
      - `<socket_name>_W`: This channel contains the transaction information of Write Data.
      - `<socket_name>_B`: This channel contains the transaction information of the corresponding Write Response.
      - `<socket_name>_AR`: This channel contains the transaction information of Read Address.
      - `<socket_name>_R`: This channel contains the transaction information of Read Data.

      Detailed attributes for each channel like burst size, burst type, response, etc. are visible as attributes in each channel.

b. For AXI4-Stream, the bus transaction is visible in one channel only named after the `socket_name`. This contains the information like TID, TDEST, TDATA, etc. as attributes.

***Note*:** TLM waveform viewing is only supported for Questa Advanced Simulator and Xcelium. The information on the usage of waveform, adding socket to waveform view, and detailed view of attributes can be referred to its respective third-party simulator user guide.

Debugging System Projects

The AMD Vitis™ unified software platform provides application-level debug features and techniques that allow the Application component, PL kernels, and the interactions between them to be debugged. However, debugging projects built from the command line is a challenge because the various elements of the system, the device binary (`.xclbin`), and the host application (`host.exe`), must be gathered together and presented as a system.

The Vitis unified IDE provides an excellent framework for debugging these heterogeneous systems. There are many advantages to working in the Debug view in the IDE. In fact, you are strongly recommended to debug your command-line driven projects in the IDE. The process for doing this is broken down into two steps:

1. Import your command-line project into the IDE as described in Getting Started with Vitis in the *Vitis Unified Software Platform Documentation: Embedded Software Development* (UG1400)
2. Debug the system in the IDE as described in Debugging the System Project and AI Engine Components in the *Vitis Reference Guide* (UG1702)

The Vitis tools provide application-level debug features which let the host code, the system project, and the interactions between them be efficiently debugged in the Vitis unified IDE. These features and techniques are split between software debugging and hardware debugging flows. The recommended debugging flow consists of two levels of debugging:

- Debugging in Hardware Emulation in the *Data Center Acceleration using Vitis* (UG1700) to compile the PL kernels into RTL, confirm the behavior of the generated logic, and evaluate the simulated performance of the hardware.
- Debugging During Hardware Execution to implement the device binary and debug the application running on hardware using Xilinx virtual cable (XVC) running over the PCIe® bus, or debugged using USB-JTAG cables for both Alveo accelerator cards and embedded processor platforms.

This two-tiered approach enables debugging the Application component and System project at different levels of abstraction. Each provides specific insights into the design providing a comprehensive view of the system from software to hardware. All flows are supported through the Vitis unified IDE using basic compile time and runtime setup options.

The AMD Vitis™ unified software platform provides application-level debug features and techniques that allow the Application component, PL kernels, and the interactions between them to be debugged. However, debugging projects built from the command line is a challenge because the various elements of the system, the compiled AI Engine graph application (`libadf.a`), the device binary (`.xclbin`), and the top-level application (`host.elf`), must be gathered together and presented as a system.

The Vitis unified IDE provides an excellent framework for debugging these heterogeneous systems. There are many advantages to working in the Debug view in the IDE. In fact, you are strongly recommended to debug your command-line driven projects in the IDE. The process for doing this is broken down into two steps:

1. Import your command-line project into the IDE as described in Getting Started with Vitis in the *Vitis Unified Software Platform Documentation: Embedded Software Development* (UG1400)
2. Debug the system in the IDE as described in Debugging the System Project and AI Engine Components in the *Vitis Reference Guide* (UG1702)

The Vitis tools provide application-level debug features which let the host code, the system project, and the interactions between them be efficiently debugged in the Vitis unified IDE. These features and techniques are split between software debugging and hardware debugging flows. The recommended debugging flow consists of two levels of debugging:

- Debugging in Hardware Emulation to compile the PL kernels into RTL, confirm the behavior of the generated logic, and evaluate the simulated performance of the hardware.
- Debugging During Hardware Execution to implement the device binary and debug the application running on hardware using Xilinx virtual cable (XVC) running over the PCIe® bus, or debugged using USB-JTAG cables for embedded processor platforms.

This three-tiered approach enables debugging the Application component and System project at different levels of abstraction. Each provides specific insights into the design providing a comprehensive view of the system from software to hardware. All flows are supported through the Vitis unified IDE using basic compile time and runtime setup options.

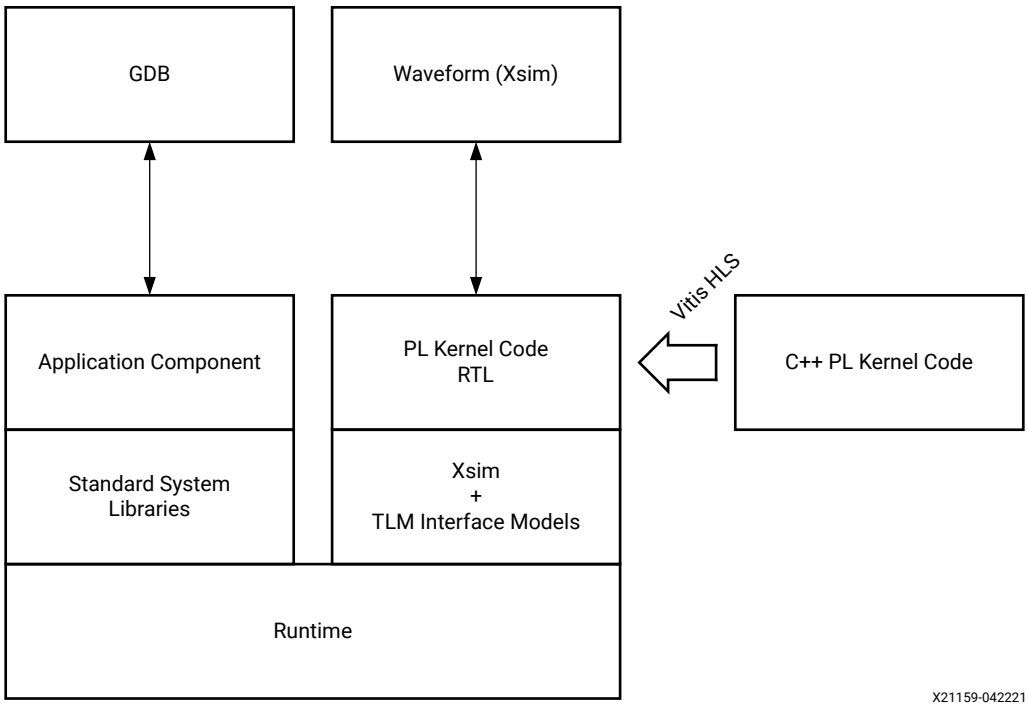
Debugging in Hardware Emulation

IMPORTANT! *The following steps describe debugging from the command line. However, you are strongly encouraged to debug command-line projects in the Vitis Unified IDE by opening them in a workspace and debugging them. For information about these steps, see Getting Started with Vitis in the Vitis Unified Software Platform Documentation: Embedded Software Development (UG1400).*

During hardware emulation, kernel code is compiled into RTL code so that you can evaluate the RTL logic of kernels prior to implementation into the AMD device. The host code can be executed concurrently with a behavioral simulation of the RTL model of the kernel, directly imported, or created through Vitis HLS from the C/C++/OpenCL kernel code. For more information, see Hardware Emulation Target.

The following figure shows the hardware emulation flow diagram which can be used in the Vitis debugger 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 RTL kernel code is analyzed in an AMD Vivado™ simulator or third-party RTL simulator. GDB is used for more traditional software-style debugging of the Application component.

Figure 50: **Hardware Emulation**



Verify the host code and the kernel hardware implementation is correct by running hardware emulation on a data set. 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 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.

TIP: *AMD recommends that you use small data sets for debug and validation in hardware emulation as it takes considerably longer than running in the hardware device.*

During hardware emulation, you can optionally modify the kernel code to improve performance. Iterate your host and kernel code design in hardware emulation until the functionality is correct, and the estimated kernel performance is satisfactory.

Enable Waveform Debugging with the Vitis Compiler Command

The waveform debugging process can be enabled through the `v++` command using the following steps:

1. Compile and Link the host code and System project for debugging by adding the `-g` option to the `g++` command line as described in Chapter 4: Building and Running the System.
2. Create an `xrt.ini` file in the same directory as the host executable, as described in xrt.ini File, with the following contents:

   ```
   [Emulation]
   debug_mode=batch
   ```

   > **TIP:** *Use `debug_mode=gui` to run the Vivado simulator in GUI mode and enable displaying the waveform for interactive use, as described in Running Live Waveform Mode. This is especially useful when debugging a `hw_emu` hang issue, because you can interrupt the simulation process in the simulator and observe the waveform up to that time.*

3. Launch the hardware emulation using the `launch_hw_emu.sh` script that is generated during the `v++ --package` process.

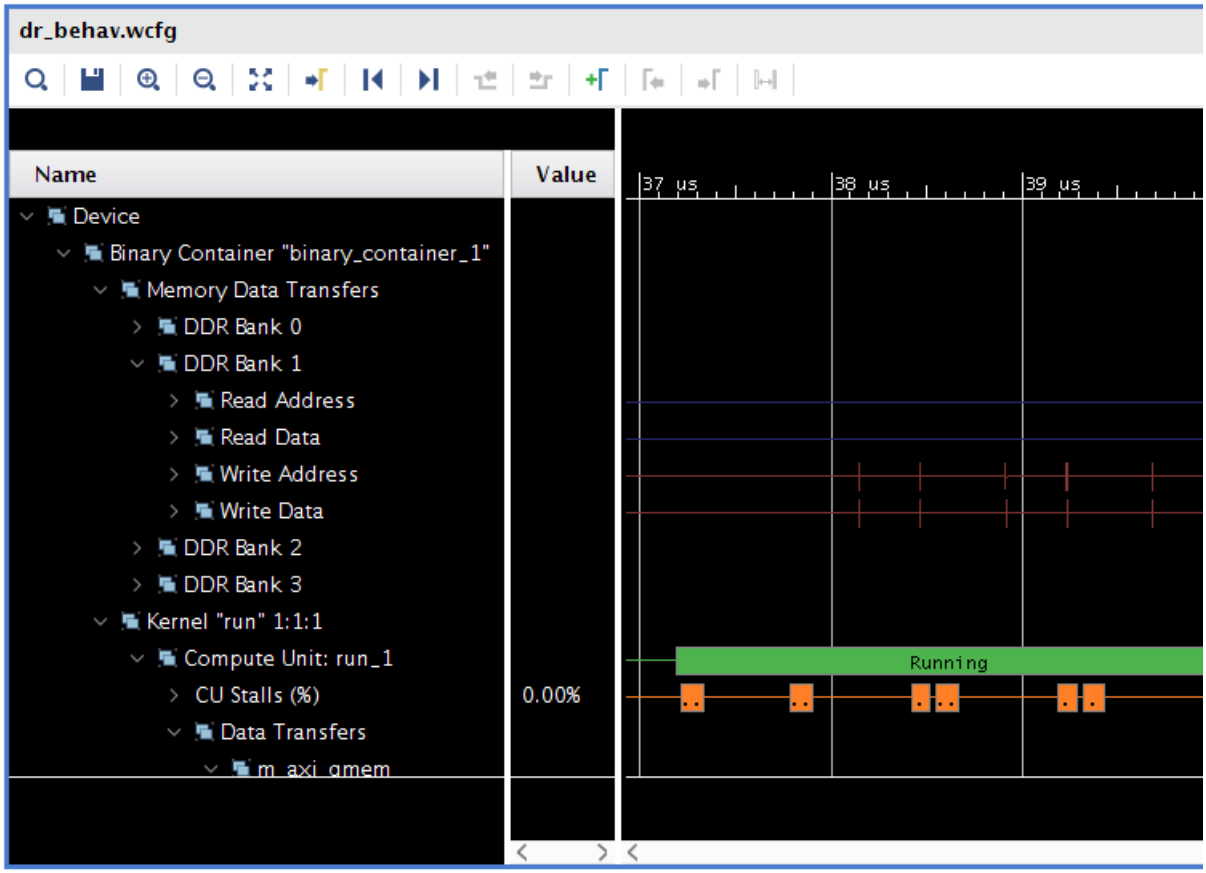
   This script will launch the Vivado simulator and enable batch simulation to capture the simulation waveform database. The waveform database is written to a `.wdb` that can be opened in the Analysis view of the Vitis unified IDE as described in Working with the Analysis View (Vitis Analyzer) in the *Vitis Reference Guide* (UG1702).

Running Live Waveform Mode

The Vitis Unified IDE provides live waveform-based HDL debugging in the hardware emulation mode. The waveform is opened in the Vivado waveform viewer, which might be familiar to Vivado logic simulation users. The Vitis Unified 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 51: **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>:** This is the 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.*

Debug Techniques for Hardware Emulation

Due to the approximate models used in hardware emulation, the behavior of an emulated system might not match the hardware. The following list provides some common issues to examine if your application does not give expected results during hardware emulation:

1. Review the host application to ensure that the event dependency between different kernel runs is correctly captured. Such issues can lead to unpredictable behavior. It is also possible that the application can pass in hardware, but there could be a logical bug in your application which can be triggered on hardware under slightly different conditions.
2. If you have an RTL kernel, run the application in debug mode and ensure that are no "X" (undriven values) in simulation in the kernel. This indicates incorrect code which can work in hardware but will fail in simulation with unpredictable behavior. If it is an HLS-generated kernel, confirm that all the variables are initialized to appropriate values.
3. Ensure that the amount of data being processed by kernels in hardware emulation is small so that emulation can finish in a reasonable time. Otherwise, it can appear that the application is running forever or has "hung". In this case, when running the application in hardware emulation look for `INFO: [Vitis-EM 22]` messages in the host application console. Check that the amount of data being read/written to or from global memory is increasing:
   a. If the RD/WR data is increasing, this indicates that application and hardware execution is progressing. The application is not hung, but is taking a really long time to complete. This could be due to large data size or due to kernels performing memory read/write in an inefficient manner. The application and kernel needs to be optimized.
   b. If the RD/WR data is not increasing in successive messages, this indicates that simulation is running but there is a deadlock in the hardware somewhere — either in the kernel or rest of the platform. Review the AXI transactions at the boundary of kernel, interconnect (for example, `sdx_memss`), and other places to check if there is an incomplete transaction or whether any transaction is being generated by the kernel.
4. Run hardware emulation in waveform mode and also review at the timeline trace. Check whether the kernel is getting "started" and "done" by observing the traffic on its AXI4-Lite interface, or by observing the output interrupt from the kernel.
5. Review the `[Emulation]` section of the xrt.ini File to enable applicable settings that can help to narrow down the issue in your application or kernel.

Debugging During Hardware Execution

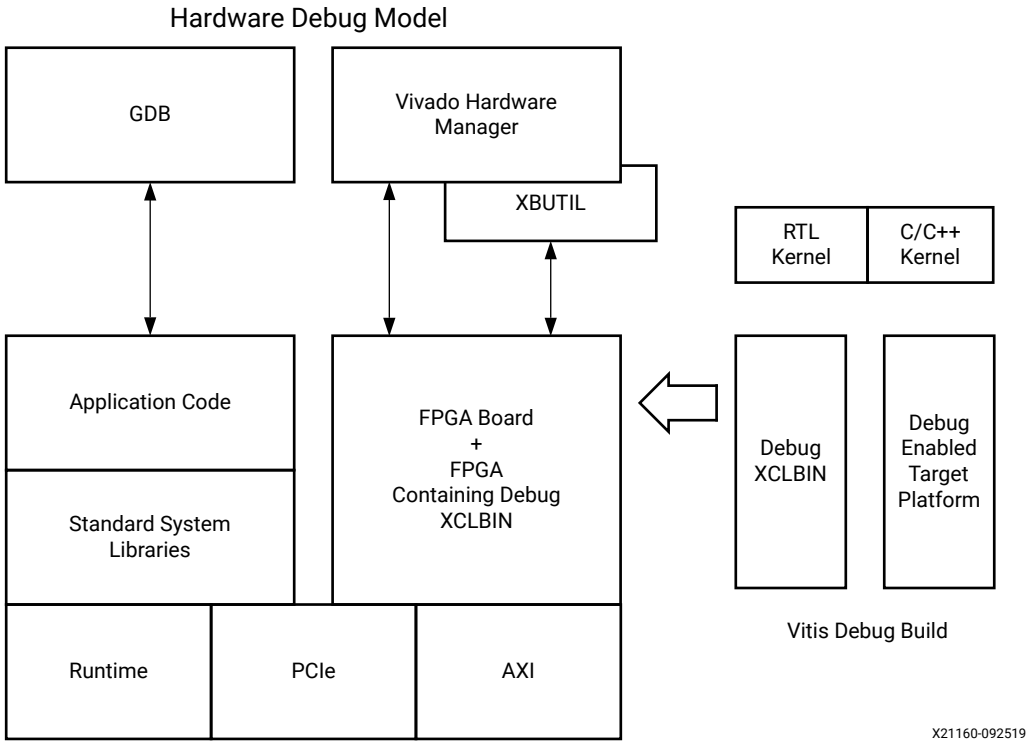
IMPORTANT! *The following steps describe debugging from the command line. However, you are strongly encouraged to debug command-line projects in the Vitis Unified IDE by opening them in a workspace as described in Getting Started with Vitis in the Vitis Unified Software Platform Documentation: Embedded Software Development (UG1400), and debugging them as described in Debugging the System Project and AI Engine Components in the Vitis Reference Guide (UG1702).*

During hardware execution, the actual hardware platform is used to execute the kernels, and you can evaluate the performance of the host program and accelerated kernels by running the application. However, debugging the hardware build requires additional logic to be incorporated into the application. This will impact both the FPGA resources consumed by the kernel and the performance of the kernel running in hardware. The debug configuration of the hardware build includes special ChipScope debug cores, such as Integrated Logic Analyzer (ILA) and Virtual Input/Output (VIO) cores, and AXI performance monitors for debug purposes.

TIP: *The additional logic required for debugging the hardware must be removed from the final production build.*

The following figure shows the debug process for the hardware build, including debugging the host code using GDB, and using the Vivado hardware manager, with waveform analysis, kernel activity reports, and memory access analysis to identify and localize hardware issues.

Figure 52: **Hardware Execution**



With the system hardware build configured for debugging, the host program running on the CPU and the Vitis accelerated kernels running on the AMD device can be confirmed to be executing correctly on the actual hardware of the target platform. Some of the conditions that can be identified and analyzed include the following:

- System hangs caused by 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, use an ILA triggered off of the AXI protocol checker, which needs to be configured on the Vitis target platform.
- Problems with the hardware kernel:
  - Problems sometimes caused by the implementation: timing issues, race conditions, and bad design constraints.
  - Functional bugs that hardware emulation does not reveal.
- 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.

Enabling Kernels for Debugging with Chipscope

System ILA

The key to hardware debugging lies in instrumenting the kernels with the required debug logic. The following topic discusses the `v++` linker options that can be used to list the available kernel ports, enable the System Integrated Logic Analyzer (ILA) core on selected ports, and enable the AXI Protocol Checker debug core for checking for protocol violations.

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 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. It can also 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.

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 `--debug` option as described in --debug Options to attach System ILA cores at the interfaces to the kernels for debugging and performance monitoring purposes.

IMPORTANT! *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.*

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

```
--debug.chipscope <cu_name>[:<interface_name>]>
```

TIP: *The `<interface_name>` is optional, and if not specified all ports on the CU will be analyzed. You can use the `--debug.list_ports` option to return the interface names on the kernel to use with `--debug` options.*

In case of a flattened design or any design where there would be multiple debug bridges in master mode, the flow will not pick one to stitch the debug cores, a constraint is needed to define the connectivity. For example in a Samsung Smart SSD U.2 flat shell, there is no partitioning between the static and dynamic regions while generating the kernels with the debug (ILA) options enabled. It is required to specify the connectivity of the kernel AXI ports that needs to be under debug to the user debug bridge in the dynamic region.

To specify the connectivity, you must provide the option below in the `v++` command line:

```
--advanced.paramcompiler.userPostDebugProfileOverlayTcl=<path to
post_dbg_profile_overlay.tcl >
```

Inside the `post_dbg_profile_overlay.tcl`, the file must call the XDC file with the connect debug core command and mention its processing order.

For example, the contents in the `post_dbg_profile_overlay.tcl` file are given below.

```
read_xdc < path to the connect_debug_core.xdc file>
set_property used_in_implementation TRUE [get_files <path to the
connect_debug_core.xdc file>]
set_property PROCESSING_ORDER EARLY [get_files <path to the
connect_debug_core.xdc file>]]
```

In the `connect_debug_core.xdc` file, you have to specify the `connect_debug_cores` constraint.

For example:

```
connect_debug_cores -master [get_cells -hierarchical -filter {NAME =~
*debug_bridge_xsdbm/inst/xsdbm}]
-slaves [get_cells -hierarchical -filter {NAME =~ level0_i/ulp/
system_ila_0}]
```

AXI Protocol Checker

The AXI Protocol Checker core monitors AXI interfaces. When attached to an interface, it actively checks for protocol violations and provides an indication of which violation occurred. You can assign it for all CUs in the design, or for specific CUs and ports.

The `--debug` option to enable AXI Protocol Checker insertion has the following syntax:

```
--debug.protocol all
```

The protocol checker can be specified with the keyword `all`, or the `<cu_name>:<interface_name>`.

Note: The `--debug.list_ports` option can be specified to return the actual names of ports on the kernel to use with `protocol` or `chipscope`.

An example flow you could use for adding ILA or protocol checkers to your design is outlined below:

1. Compile the kernel source files into an XO file, using the `-g` option to instrument the kernel for debug features:

   ```
   v++ -c -g -k <kernel_name> --platform <platform> -o <kernel_xo_file>.xo
   <kernel_source_files>
   ```

2. After the kernel has been compiled into an XO file, use `--debug.list_ports` to cause the `v++` compiler to print the list of valid compute units and port combinations for the kernel:

   ```
   v++ -l -g --platform <platform> --connectivity.nk
   <kernel_name>:<compute_units>:<kernel_nameN>
   --debug.list_ports <kernel_xo_file>.xo
   ```

3. Add the ILA or AXI debug cores on the desired ports by replacing `list_ports` with the appropriate `--debug.chipscope` or `--debug.protocol` command syntax:

   ```
   v++ -l -g --platform <platform> --connectivity.nk
   <kernel_name>:<compute_units>:<kernel_nameN>
   --debug.chipscope <compute_unit_name>:<interface_name>
   <kernel_xo_file>.xo
   ```

TIP: *The `--debug` option can be specified multiple times in a single `v++` command line, or configuration file to specify multiple CUs and interfaces.*

When the design is built, you can debug the design using the Vivado hardware manager as described in Debugging with ChipScope.

Adding Debug IP to RTL Kernels

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

You can also enable debugging in RTL kernels by manually adding ChipScope debug cores like the ILA and VIO in your RTL kernel code before packaging it for use in the Vitis development flow. From within the Vivado Design Suite, edit the RTL kernel code to manually instantiate an ILA debug core, or VIO IP from the AMD IP catalog, similar to using any other IP in Vivado IDE. Refer to the HDL Instantiation flow in the *Vivado Design Suite User Guide: Programming and Debugging* (UG908) to learn more about adding debug cores to your design.

The best time to add debug cores to your RTL kernel is when you create it. However, debug cores consume device resources and can affect performance, so it is good practice to make one kernel for debug and a second kernel for production use. The `rtl_vadd_hw_debug` of the RTL Kernels examples on GitHub shows an ILA debug core instantiated into the RTL kernel source file. The ILA monitors the output of the combinatorial adder as specified in the `src/hdl/krnl_vadd_rtl_int.sv` file.

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

You can also add the ILA debug core using a Tcl script from within an open Vivado project, using the Netlist Insertion flow described in *Vivado Design Suite User Guide: Programming and Debugging* (UG908), as shown in the following Tcl script 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]
```

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

Enabling ILA Triggers for Hardware Debug

To perform hardware debug of both the host program and the kernel code running on the target platform, the application host code must be modified to let you set up the ILA trigger conditions *after* the kernel has been programmed into the device, but *before* starting the kernel.

Adding ILA Triggers Before Starting Kernels

Pausing the host program can be accomplished through the use of a pause, or wait step in the code, such as the `wait_for_enter` function used in the RTL Kernel example on GitHub. The function is defined in the `src/host.cpp` code as follows:

```
void wait_for_enter(const std::string &msg) {
    std::cout << msg << std::endl;
    std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');
}
```

The `wait_for_enter` function is used in the `main` function as follows:

```
....
    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 use of the `wait_for_enter` function pauses the host program to give you time to set up the required ILA triggers and prepare to capture data from the kernel. After the Vivado hardware manager is set up and configured, press `Enter` to continue running the application.

- For C++ host code, add a pause after the creation of the `cl::Kernel` object, as shown in the example above.
- For C-language host code, add a pause after the `clCreateKernel()` function call:

Figure 53: **C-language Host Code**

```
// 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;
}
```

Pausing the Host Application Using GDB

If you are running GDB to debug the host program at the same time as performing hardware debug on the kernels, you can also pause the host program as needed by inserting a breakpoint at the appropriate line of code. Instead of making changes to the host program to pause the application as needed, you can set a breakpoint prior to the kernel execution in the host code. When the breakpoint is reached, you can set up the debug ILA triggers in Vivado hardware manager, arm the trigger, and then resume the host program in GDB.

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. These tools enable a wide range of capabilities from logic to system-level debug while your kernel is running in hardware. To achieve this, at least one of the following must be true:

- Your Vitis application project has been designed with debug cores, using the `--debug.xxx` compiler switch, as described in Enabling Kernels for Debugging with Chipscope.
- The RTL kernels used in your project need to be instantiated with debug cores (as described in Adding Debug IP to RTL Kernels).

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 an AMD 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:

```
$ platforminfo --json="hardwarePlatform.extensions.chipscope_debug"
xilinx_u250_gen3x16_xdma_4_1_202210_1
{
    "debug_networks": {
        "user": {
            "bar_number": "0",
            "supports_jtag_fallback": "false",
            "name": "User Debug Network",
            "supports_microblaze_debug": "true",
            "pcie_pf": "1",
            "axi_baseaddr": "0x000001C00000",
            "is_user_visible": "true"
        },
        "mgmt": {
            "bar_number": "0",
            "supports_jtag_fallback": "true",
            "name": "Management Debug Network",
            "supports_microblaze_debug": "true",
            "pcie_pf": "0",
            "axi_baseaddr": "0x01F60000",
            "is_user_visible": "false"
        }
    }
}
```

The response shows that the target platform contains `user` and `mgmt` debug networks, supports debugging a MicroBlaze™ processor, and also supports JTAG fallback for the Management Debug Network.

Running XVC and HW Servers

The following steps are required to run the Xilinx virtual cable (XVC) and HW servers, host applications, and also trigger and arm the debug cores in the Vivado hardware manager.

1. Add debug IP to the kernel as discussed in Enabling Kernels for Debugging with Chipscope.
2. Modify the host program to pause at the appropriate point as described in Enabling ILA Triggers for Hardware Debug.
3. Set up the environment for hardware debug, using an automated script described in Automated Setup for Hardware Debug, or manually as described in Manual Setup for Hardware Debug.
4. Run the hardware debug flow using the following process:
   a. Launch the required XVC and the `hw_server` of the Vivado hardware manager.
   b. Run the host program and pause at the appropriate point to enable setup of the ILA triggers.
   c. Open the Vivado hardware manager and connect to the XVC server.
   d. Set up ILA trigger conditions for the design.
   e. Continue execution of the host program.
   f. Inspect kernel activity in the Vivado hardware manager.
   g. Rerun iteratively from step b (above) as required.

Automated Setup for Hardware Debug

1. Set up your Vitis core development kit as described in Setting Up the Vitis Environment in the *Data Center Acceleration using Vitis* (UG1700).
2. Use the `debug_hw` script to launch the `xvc_pcie` and `hw_server` apps as follows:

   ```
   debug_hw --xvc_pcie /dev/xfpga/xvc_pub.<driver_id> --hw_server
   ```

   The `debug_hw` script returns the following:

   ```
   launching xvc_pcie...
   xvc_pcie -d /dev/xfpga/xvc_pub.<driver_id> -s TCP::10200
   launching hw_server...
   hw_server -sTCP::3121
   ```

   TIP: *The `/dev/xfpga/xvc_pub.<driver_id>` driver character path is defined on your machine, and can be found by examining the `/dev` folder.*

3. Modify the host code to include a pause statement *after* the kernel has been created/downloaded and *before* the kernel execution is started, as described in Enabling ILA Triggers for Hardware Debug.
4. Run your modified host program.

5. Launch Vivado Design Suite using the `debug_hw` script:

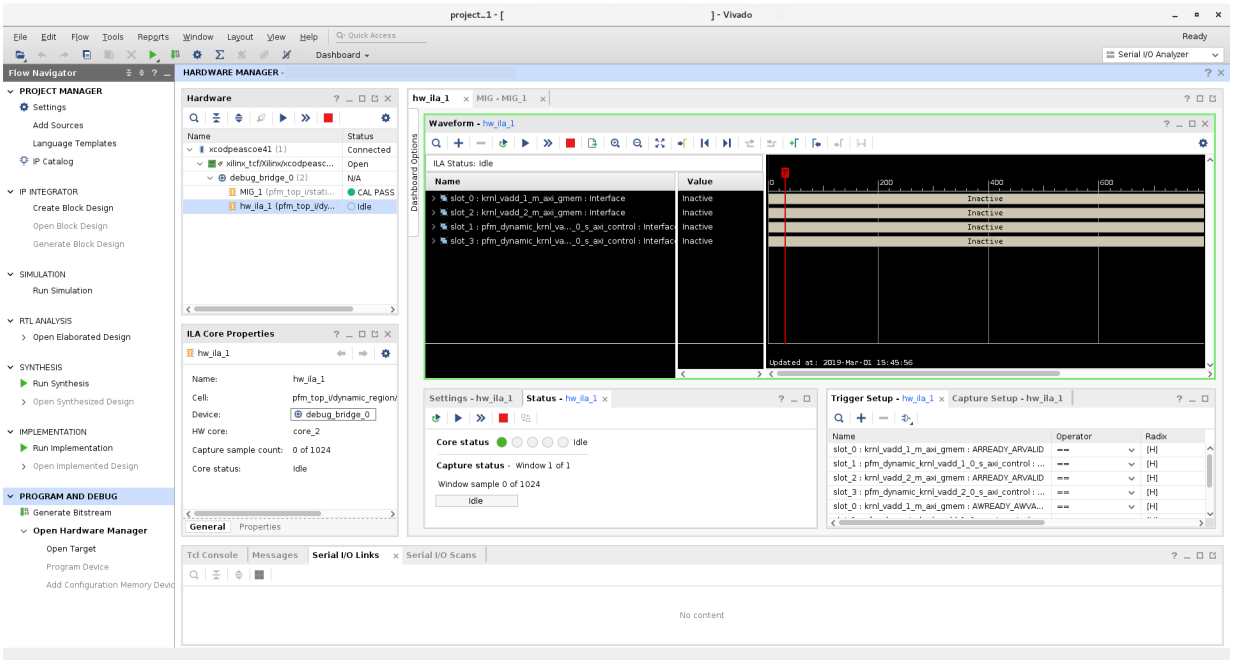
```
debug_hw --vivado --host <host_name> --ltx_file ./_x/link/vivado/vpl/prj/
prj.runs/impl_1/debug_nets.ltx
```

> **TIP:** *The `<host_name>` is the name of your system.*

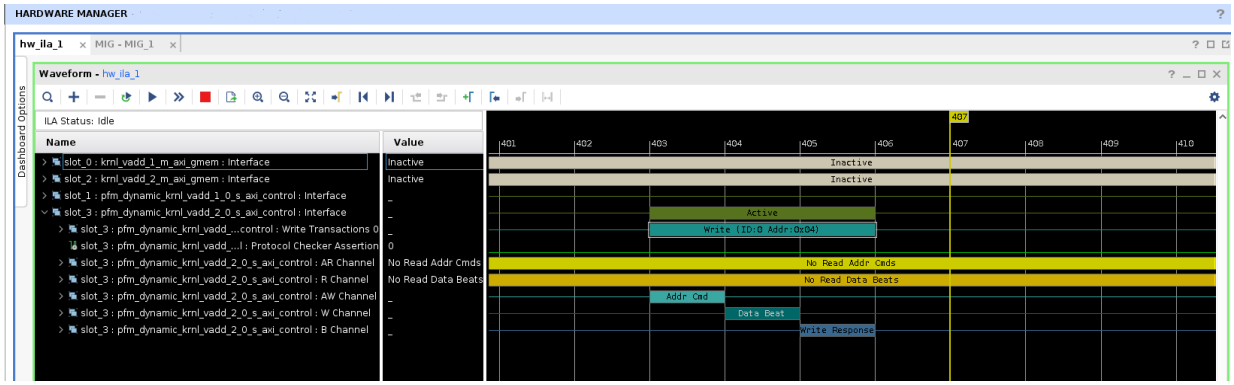
As an example, the command window displays the following results:

```
launching vivado... ['vivado', '-source', 'vitis_hw_debug.tcl', '-
tclargs',
'/tmp/project_1/project_1.xpr', 'workspace/vadd_test/System/
pfm_top_wrapper.ltx',
'host_name', '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 continue running the host program.
8. In the Vivado hardware manager, see the interface transactions on the kernel compute unit slave control interface in the Waveform view.



Manual Setup for Hardware Debug

TIP: *The following steps can be used when setting up Nimbix and other cloud platforms.*

There are a few steps required to start the debug servers prior to debugging the design in the Vivado hardware manager.

1. Set up your Vitis core development kit as described in Setting Up the Vitis Environment in the *Data Center Acceleration using Vitis* (UG1700).
2. Launch the `xvc_pcie` server. The file name passed to `xvc_pcie` must match the character driver file installed with the kernel device driver, where `<driver_id>` can be found by examining the `/dev` folder.

```
>xvc_pcie -d /dev/xfpga/xvc_pub.<device_id>
```

TIP: *The `xvc_pcie` server has many useful command line options. You can issue `xvc_pcie -help` to obtain the full list of available options.*

3. Start the `hw_server` on port `3121`, and connect to the XVC server on port `10201` using the following command:

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

4. Launch Vivado Design Suite and open the hardware manager:

```
vivado
```

Starting Debug Servers on an Amazon F1 Instance

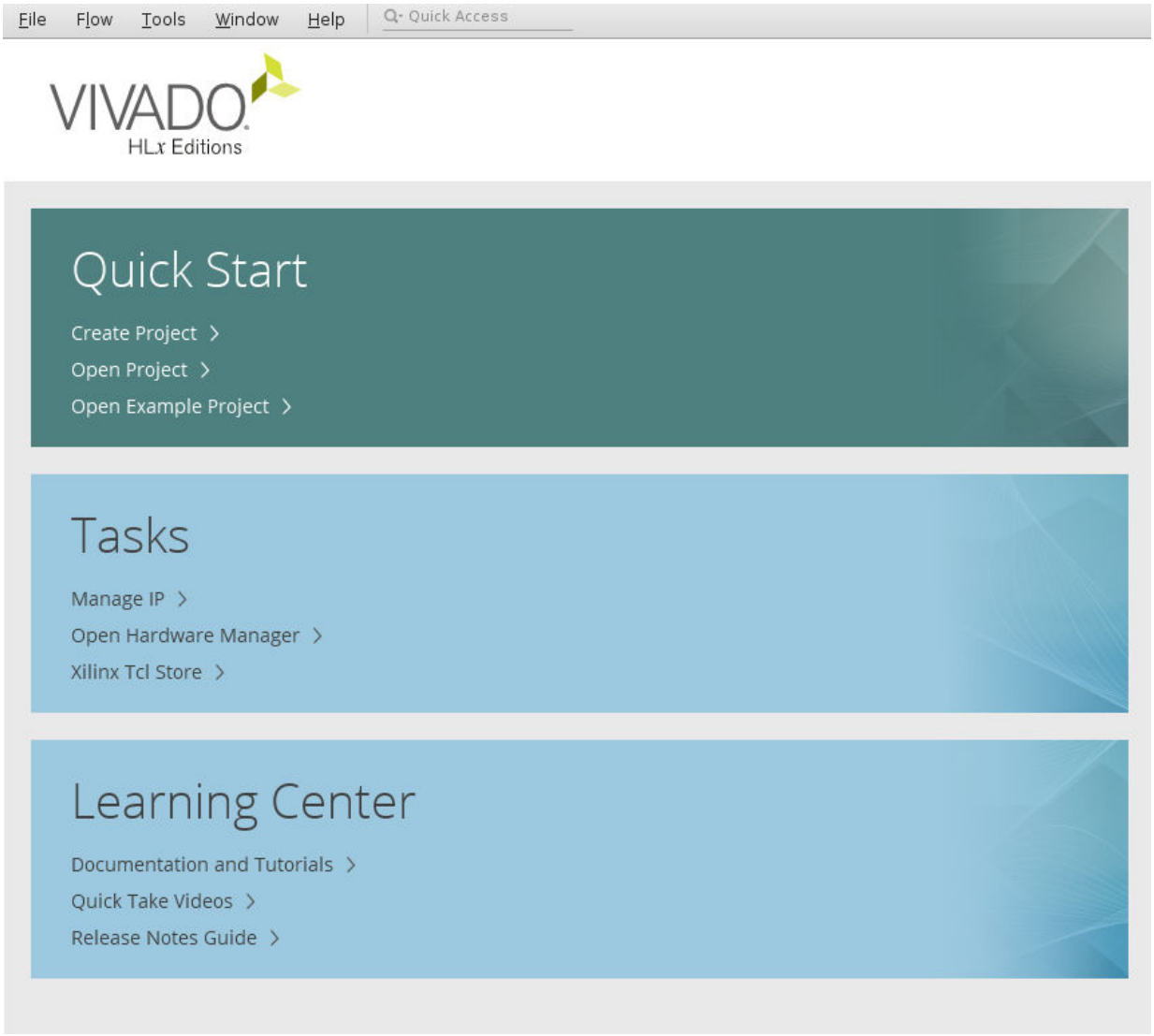
Instructions to start the debug servers on an Amazon F1 instance can be found here: https://github.com/aws/aws-fpga/blob/master/hdk/docs/Virtual_JTAG_XVC.md.

Debugging Designs Using Vivado Hardware Manager

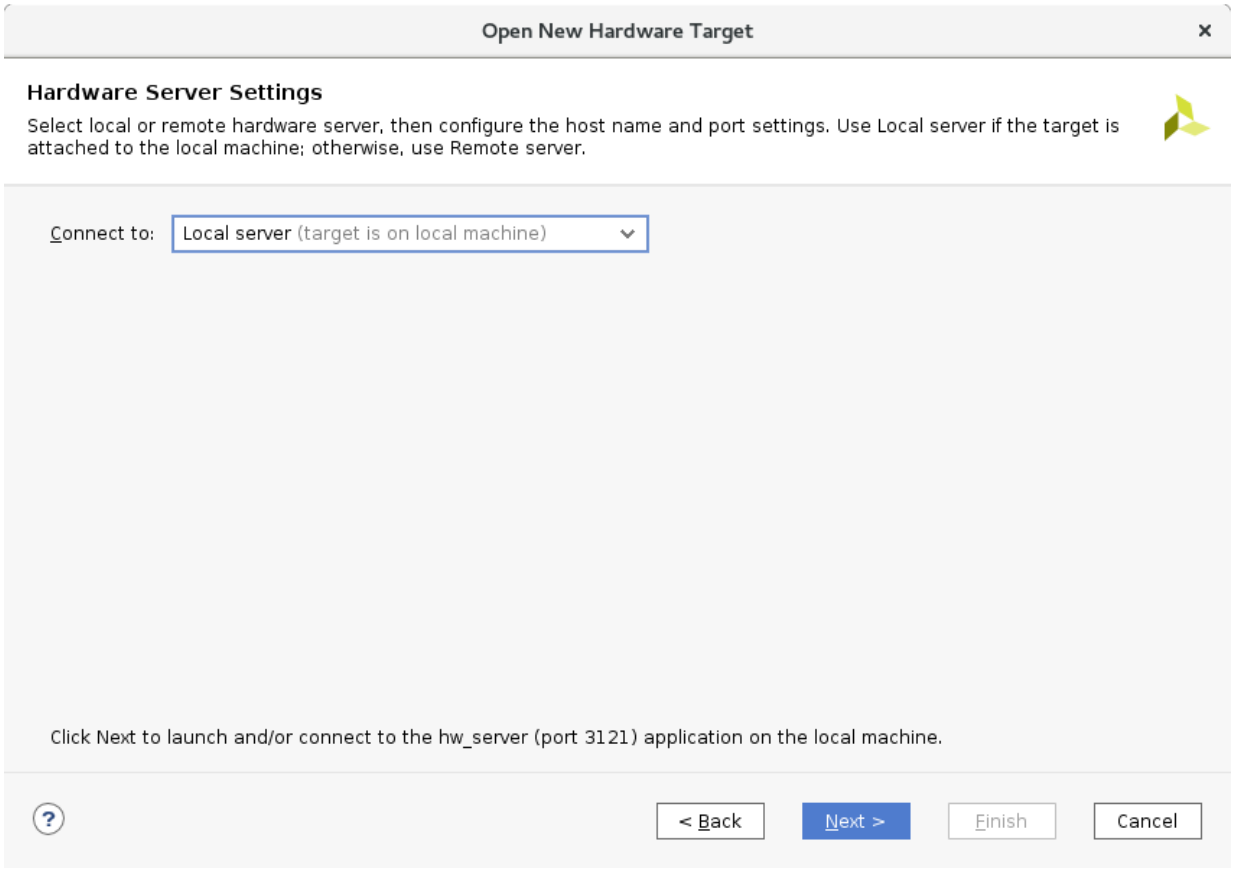
Traditionally, a physical JTAG connection is used to perform hardware debug for AMD devices with the Vivado hardware manager. The Vitis unified software platforms also makes use of the Xilinx virtual cable (XVC) for hardware debugging on remote accelerator cards. To take advantage of this capability, the Vitis debugger uses the XVC server, an implementation of the XVC protocol that allows the Vivado hardware manager to connect to a local or remote target device for debug, using the standard AMD debug cores like the ILA or the VIO IP.

The Vivado hardware manager, from the Vivado Design Suite or Vivado debug feature, 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 must be followed on the machine hosting the Vivado tools:

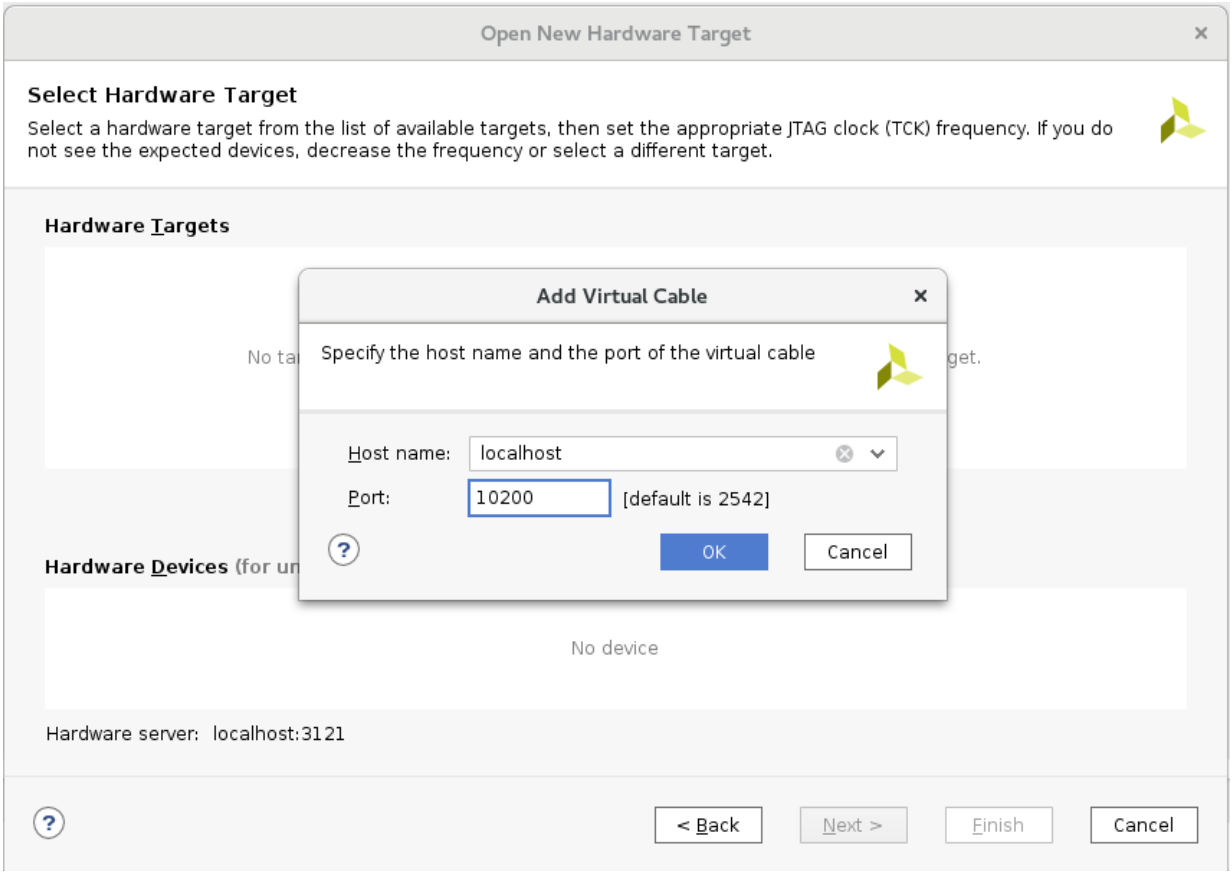
1. Launch the Vivado debug feature, or the full Vivado Design Suite.
2. Select **Open Hardware Manager** from the Tasks menu, as shown in the following figure.



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.

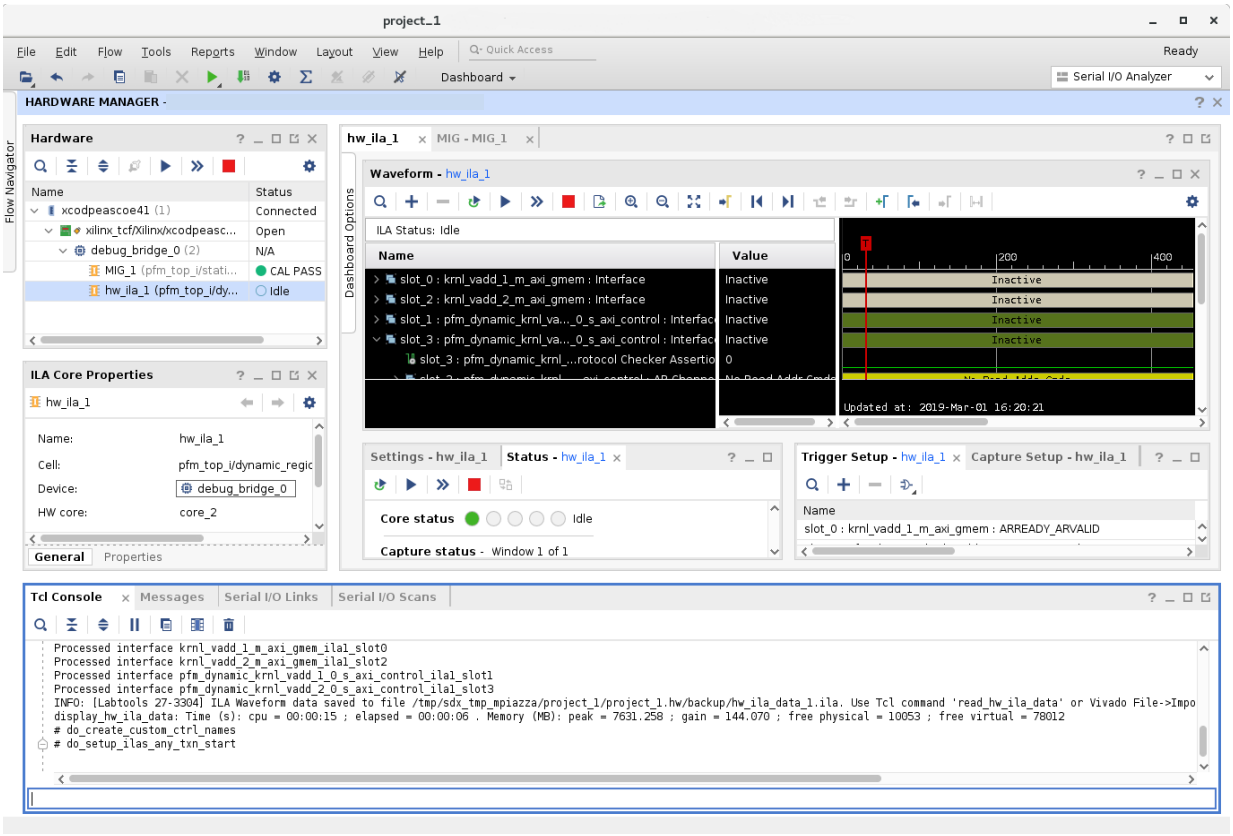


5. Select the `debug_bridge` instance from the Hardware window in the Vivado hardware manager.

   Specify the probes file (`.ltx`) for your design adding it to the **Probes → File** entry in the Hardware Device Properties window. Adding the probes file refreshes the hardware device, and the Hardware window now shows the debug cores in your design.

   > **TIP:** *If the kernel has debug cores as specified in Enabling Kernels for Debugging with Chipscope, the probes file (`.ltx`) is written out during the implementation of the kernel by the Vivado tool.*

6. The Vivado hardware manager can now be used to debug the kernels running on the Vitis software platform. Arm the ILA cores in your kernels and run your host application.



TIP: *Refer to the Vivado Design Suite User Guide: Programming and Debugging (UG908) for more information on working with the Vivado hardware manager to debug the design.*

JTAG Fallback for Private Debug Network

Hardware debug for the Alveo Data Center accelerator cards typically uses the XVC-over-PCIe connection due to the inaccessibility of the physical card, and the JTAG connector on the card. While XVC-over-PCIe allows you to remotely debug your application running on the target platform, certain conditions such as AXI interconnect system hangs can prevent you from accessing the hardware debug functionality that depends on these PCIe/AXI features. Being able to debug these kinds of conditions 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 hardware manager user connects through the `hw_server` to a JTAG cable that is connected to the physical JTAG pins of the accelerator card, or device under test (DUT), the `hw_server` disables the XVC-over-PCIe pathway to the hardware. This lets you use the XVC-over-PCIe cable as your primary debug path, but enable debug over the JTAG cable directly when it is required in certain situations. When you disconnect from the JTAG cable, the `hw_server` re-enables the XVC-over-PCIe pathway to the hardware.

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.

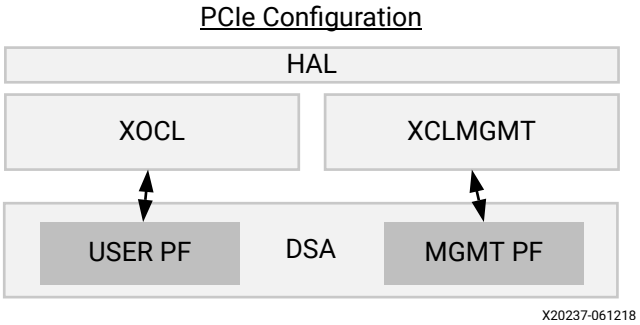
The `dmesg` utility is a Linux tool that lets you read the kernel ring buffer. The kernel ring buffer holds kernel information messages in a circular buffer. A circular buffer of fixed size is used to limit the resource requirements by overwriting the oldest entry with the next incoming message.

TIP: *In most cases, it is sufficient to work with the less verbose `xrt-smi` feature to localize an issue. Refer to Using the xrt-smi Utility for more information on using this tool for debug.*

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.), use the `dmesg` tool to check the ring buffer.

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

Figure 54: **Software Platform Layers**



To review messages from the Linux tool, 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:

Figure 55: **dmesg Utility Example**

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

In the example shown above, the AXI Firewall 2 has tripped, which is better examined using the `xrt-smi` utility.

Using the xrt-smi Utility

The Xilinx board utility (`xrt-smi`) 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 xrt-smi Utility in the *Vitis Reference Guide* (UG1702).

With respect to debugging, the following `xrt-smi` options are of special interest:

- **`examine`:** 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 AMD device.
- **`xrt-smi exmaine -r debug-ip-status -d <BDF>`:** Extracts the status of the Performance Monitors (`aim` and `asm`) and the Lightweight AXI Protocol Checkers (`lapc`).

Techniques for Debugging 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 `xrt-smi` 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 prevents host hangs. This is why the AXI Protocol Firewall IP is included in all production Vitis 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 `xrt-smi` to program the FPGA:

   ```
   xrt-smi program -p <xclbin>
   ```

   TIP: *Refer to xrt-smi Utility in the Vitis Reference Guide (UG1702) for more information on `xrt-smi`.*

2. Run the `xrt-smi examine` option to check memory topology:

   ```
   xrt-smi examine -r memory -d <bdf>
   ```

   In the following example, there are no kernels associated with memory banks:

```
##############################################################################
Mem Topology                                          Device Memory Usage
Tag             Type        Temp        Size    Mem Usage       BO nums
 [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 must indicate an issue. In this case, it is necessary to check kernel and host code expectations. If the host code is using the AMD OpenCL extensions, it is necessary to check which DDR banks must be used by the kernel. These must match the `connectivity.sp` options specified as discussed in Mapping Kernel Ports to 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 (`--link`). Both of these options add hardware to your implementation, and based on resource usage it can be necessary to limit instrumentation.

   a. Add Lightweight AXI Protocol Checkers (`lapc`). These protocol checkers are added using the `--debug.protocol` option, as explained in --debug Options. The following syntax is used:

   ```
   --debug.protocol <compute_unit_name>:<interface_name>
   ```

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

   Note: Multiple `--debug.xxx` options can be specified in a single command line, or configuration file.

b. Adding Performance Monitors (`am, aim, asm`) enables the listing of detailed communication statistics (counters). Although this is most useful for performance analysis, it provides insight during debugging on pending port activities. The Performance Monitors are added using the `--profile` option as described in --profile Options. The basic syntax for the `--profile` option is:

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

Three fields are required to determine the specific interface to attach the performance monitor to. However, if resource consumption is not an issue, the keyword `all` lets you 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 `counters` for large designs, while `all` (default) includes the collection of actual trace information.

Note: Multiple `--profile` options can be specified in a single command line, or configuration file.

```
[profile]
dataernel1:cu1:m_axi_gmem0
dataernel1:cu1:m_axi_gmem1
dataernel2:cu2:m_axi_gmem
```

2. When the application is rebuilt, rerun the host application using the `xclbin` with the added AIM IP and LAPC IP.
3. When the application hangs, you can use `xrt-smi examine` to check for any errors or anomalies.
4. Check the AIM output:
   - Run the following command a couple of times to check if any counters are moving. If they are moving then the kernels are active.

     ```
     xrt-smi examine -d <bdf> -r debug-ip-status -e aim
     ```

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

   - If the counters are stagnant, the outstanding counts greater than zero might mean some AXI transactions are hung.
5. Check the LAPC output:
   - Run the following command to check if there are any AXI violations.

     ```
     xrt-smi examine -d <bdf> -r debug-ip-status -e lapc
     ```

     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 the following command:

   ```
   xrt-smi examine -d <bdf> -r firewall
   ```

   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 `xrt-smi` 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 `xrt-smi` 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 compiler 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 these 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.

Chapter 7

Additional Information

Migrating from Existing Tools

Because the classic AMD Vitis™ IDE is obsolete, you must migrate to the new Vitis unified IDE by understanding a few concepts and operation changes. For each of the various components supported by the Vitis unified IDE the links to appropriate migration content are as follows:

- HLS Components: Refer to "Migrating from Vitis HLS" in *Vitis High-Level Synthesis User Guide* (UG1399)
- System Projects: see the System Project Structure chapter in the *Vitis Reference Guide* (UG1702).

Terminology Changes

The top-level System project has several components such as the Application component that stores the host application and the HLS component that defines the PL kernels. The HLS components and AI Engine components can be grouped into binary containers and linked with a hardware link, similar to the existing Vitis IDE flow.

The following table provides a brief comparison of the concepts and terminology.

Table 15: **Concepts and Terminology Comparison**

New Vitis IDE	Classic Vitis IDE
Workspace	Workspace
System project	System project
Application component	Host application
HLS component	PL kernel application
AI Engine Component	AI Engine design

Control File Differences

The new Vitis Unified IDE uses the unified command-line in the `v++` compiler. The `v++ --mode hls` is driven by new configuration commands that are based on existing tools.

For the HLS component, the configuration file command syntax is based on the existing AMD Vitis™ HLS Tcl command language. However, there are sufficient differences in the configuration file command language and syntax that it can seem unfamiliar to start. The new Vitis Unified IDE provides a simple user interface to add to and manage configuration files, so that you can use the IDE to edit configuration files and switch to text editor mode to help you become familiar with the new command syntax.

The Vitis Unified IDE uses CMake to generate `Makfile` for the project flow. You can update the `CMakeList.txt` to add your compile options. If you prefer using custom Makfile, you can use the custom flow.

Migrating Projects from Classic IDE

Classic Vitis IDE workspaces cannot be opened directly in the Vitis Unified IDE. The classic IDE provides an utility to enable moving a project or workspace into the Vitis Unified IDE. To use this utility, first launch the Vitis classic IDE in the workspace with the following command:

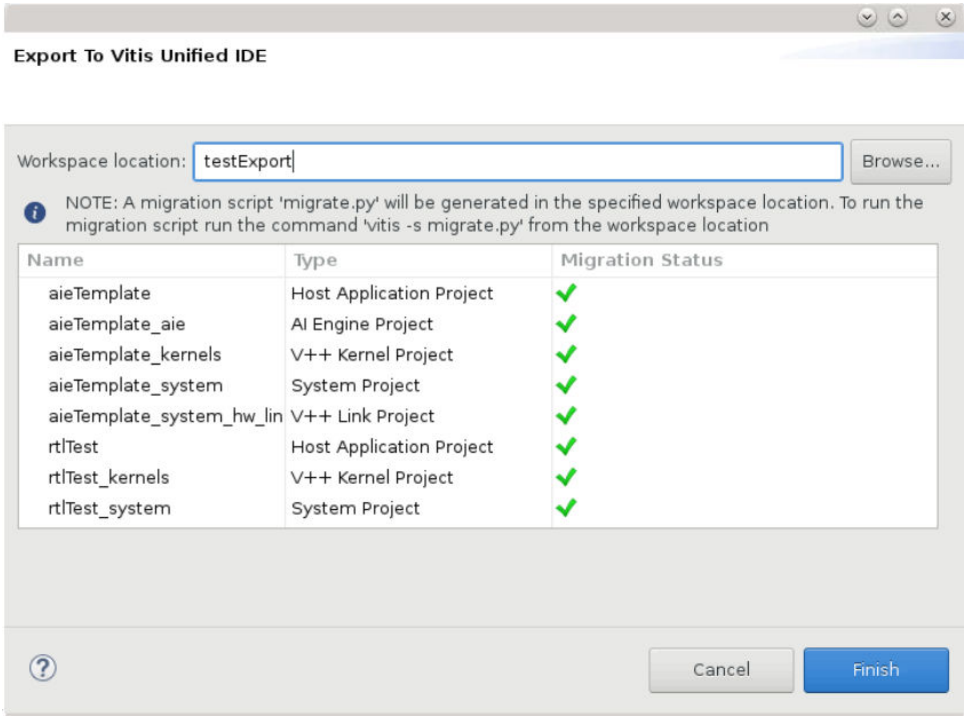
```
vitis --classic -workspace <workspace>
```

After Vitis IDE opens the workspace, use the **Vitis → Export Workspace to Unified IDE** command from the main menu. The tool opens a dialog box listing the projects contained in the workspace, and prompts you for a new workspace to export the projects to. Specify the new workspace location and click **Finish**. The tool will generate a Python migration script (`migrate.py`) and write it to the specified workspace folder.

TIP: *The Workspace location specified for the export script must be a new empty workspace. If you want to make changes to the classic IDE project and re-export it, you must start with a clean export workspace, without any hidden files.*

***Note*:** The migration utility is created to help you speed up the procedure, but you can also recreate the workspace in the Vitis Unified IDE from scratch for migration. The migration utility does not consider all the corner cases and can encounter issues for complex designs.

Figure 56: **Export to Vitis Unified IDE**



After generating the migration script, a pop-up window will display the location of the script. You will need to run the script in the Vitis Unified IDE by using the `vitis -s <script>` form of the command as described in Launch Options in the *Vitis Reference Guide* (UG1702).

```
vitis -s migrate.py
```

Table 16: **Supported Project Types**

Project Type	Limitation	Workaround
Embedded Platforms	Local changes to BSP sources will not be migrated to the new workspace. A new BSP will be created, and the settings are applied on the new BSP.	Copy the sources to the new BSP manually.
	Any embedded software repositories added to the Vitis IDE will not be migrated. A warning is displayed in the wizard if the project has any local SW repos.	All software repositories need to be migrated to lopper first. Refer to *Vitis Unified Software Platform Documentation: Embedded Software Development* (UG1400) for more information. The path to the migrated repository can be manually added to the migration script prior to running the script.
	IP drivers included in XSAs created with 2023.1 or older releases will not work.	Need to regenerate the XSA.
Embedded Software Applications	Applications referring to platforms that are outside the current workspace cannot be migrated.	Migrate the platform first and update the application to use the new platform before migrating the application.

Table 16: **Supported Project Types** *(cont'd)*

Project Type	Limitation	Workaround
HW Link	Hardware linker options defined using the **Extra V++ command line options** will not be migrated through the script.	You will need to manually define these options in the `hw_link.cfg` for the System project
Accelerated Host applications	Only compile definitions (-D), include path (-I), library paths (-L) and libraries (-l) are migrated for accelerated host applications.	Any other compiler or linker settings from the C/C++ build settings window will need to be set manually in the Application component.

Migrating to a New Target Platform

This migration content is intended for users who need to migrate their accelerated AMD Vitis™ technology application from one target platform to another. For example, moving an application from an AMD Alveo™ U200 Data Center accelerator card, to an Alveo U280 card.

Design Migration

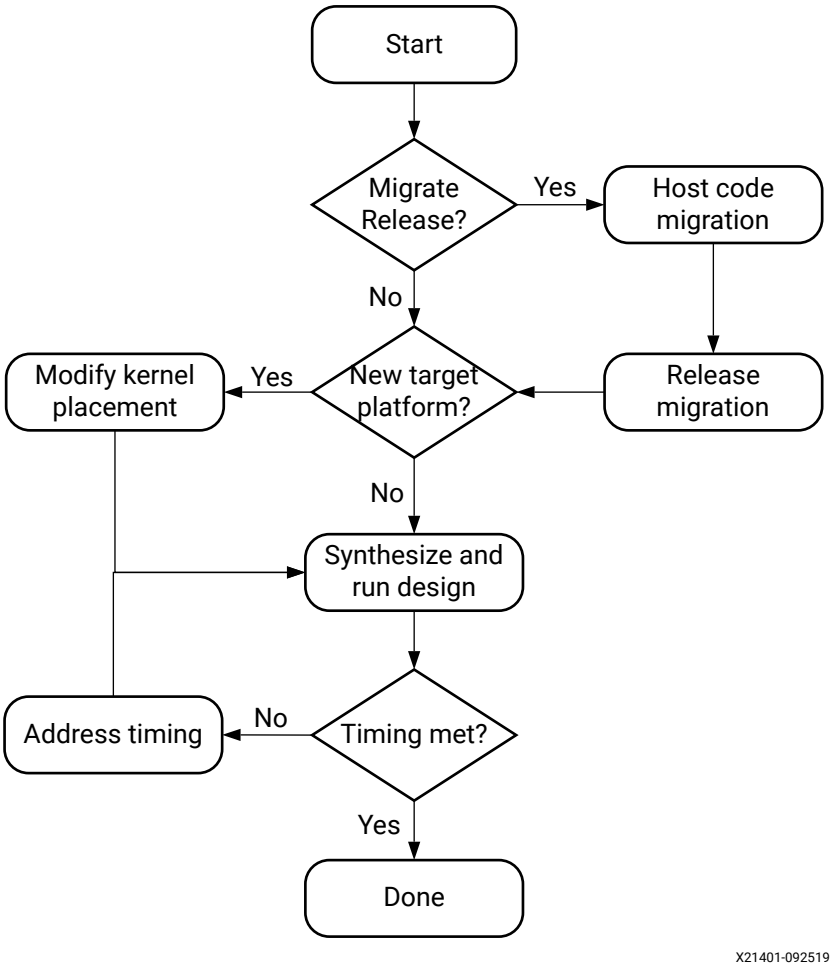
When migrating an application implemented in one target platform to another, it is important to understand the differences between the target platforms and the impact those differences have on the design.

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 57: **Target Platform Migration Flowchart**



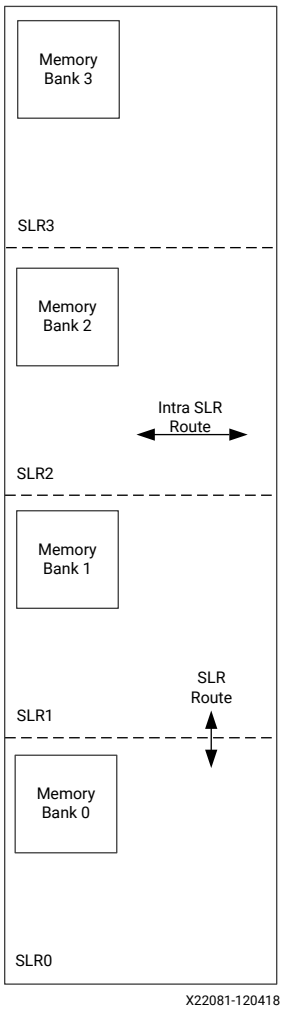
IMPORTANT! *Before starting to migrate a design, it is important to understand the architecture of an FPGA and the target platform.*

Understanding an FPGA Architecture

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

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

Figure 58: **Physical View of AMD 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* (WP380).

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 AMD 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 can be placed across multiple SLRs if needed.

SLR Routing Resources

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

- **Intra-SLR Resources:** Intra-SLR routing resource are the fast resources used to connect the hardware logic. The 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 reads data from the DDR memory and writes the results back to the same DDR memory. The memory interface connects to the pins on the FPGA and includes the memory controller logic.

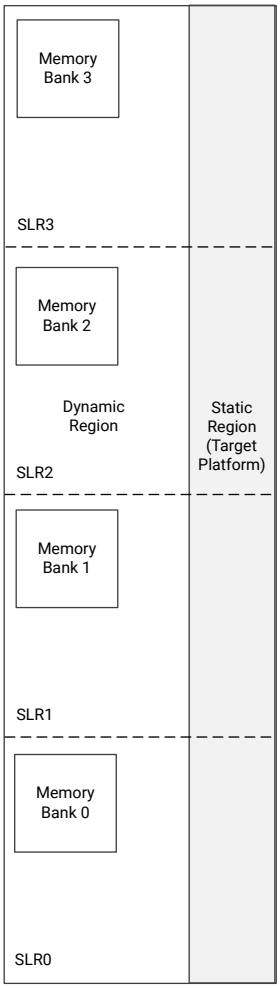
Understanding Target Platforms

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 59: **Target Platform on an FPGA with Four SLR Regions**



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 Software Platform Release Notes* (UG1742). This topic is also addressed in Modifying Kernel Placement.

Migrating Releases

Before migrating to a new target platform, you must also determine if you need to target the new platform to a different release of the Vitis technology. If you intend to target a new release, AMD 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, AMD recommends that you review the Vitis Software Platform Release Notes (UG1742).*

Host Code Migration

The `XILINX_XRT` environment variable is used to specify the location of the XRT library environment and must be set before you compile the host code. When the XRT library 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 library 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 library 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 library.

- 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.

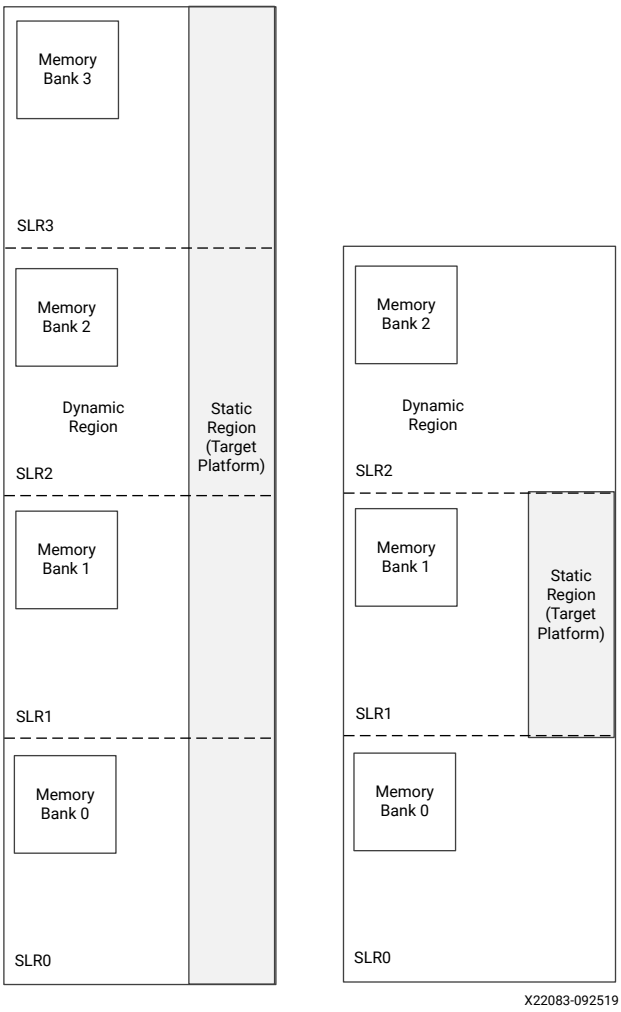
Address these issues using the same techniques you would use during the development of any kernel. At this stage, ensure the throughput performance of the target platform using the new release meets your requirements. If there are changes to the final timing (the maximum clock frequency), you can address 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 60: **Comparison of Target Platforms of the Hardware Platform**



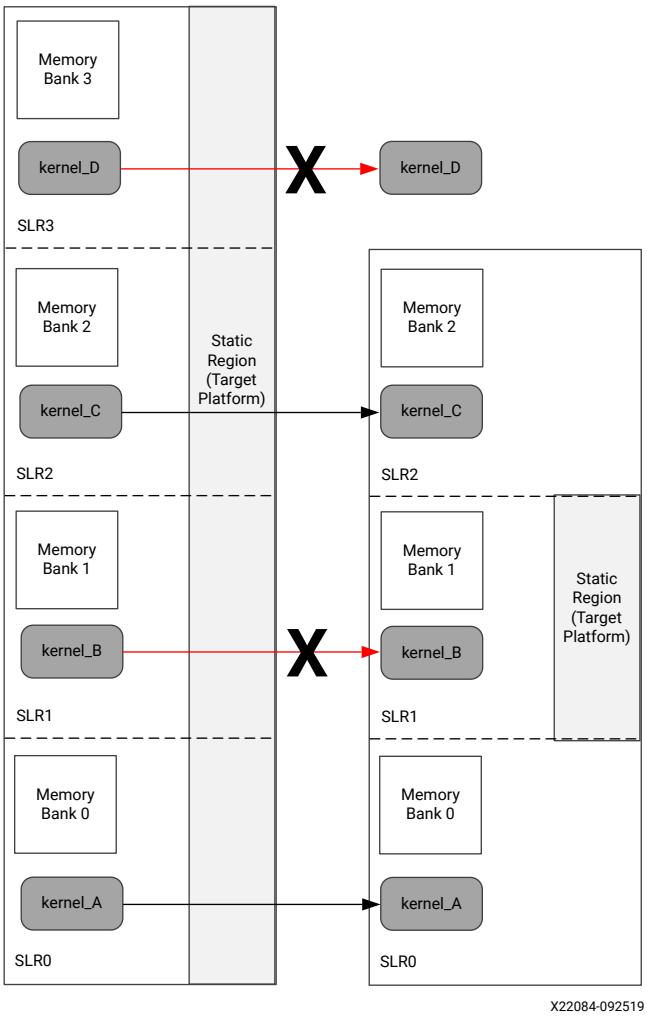
This section explains how to modify the placement of the kernels.

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 61: **Migrating Platforms – Kernel Placement**



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

- Understand the resources available in each SLR of the new target platform, as documented in the *Vitis Software Platform Release Notes* (UG1742).
- 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 on Alveo Accelerator Cards and Mapping Kernel Ports to 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 Software Platform Release Notes* (UG1742). 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 17: **SLR Resources of a Hardware Platform**

Area	SLR 0	SLR 1	SLR 2
SLR description	Bottom of device; dedicated to dynamic region.	Middle of device; shared by dynamic and static region resources.	Top of device; dedicated to dynamic region.
Dynamic region Pblock name	pfa_top_i_dynamic_region_pblock_dynamic_SLR0	pfa_top_i_dynamic_region_pblock_dynamic_SLR1	pfa_top_i_dynamic_region_pblock_dynamic_SLR2
Compute unit placement syntax	set_property CONFIG.SLR_ASSIGNMENTS SLR0[get_bd_cells<cu_name>]	set_property CONFIG.SLR_ASSIGNMENTS SLR1[get_bd_cells<cu_name>]	set_property CONFIG.SLR_ASSIGNMENTS SLR2[get_bd_cells<cu_name>]
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 Hardware run are complete. An example of this report is shown below.

Figure 62: **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.
- Block RAM 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 on Alveo Accelerator Cards for more information.

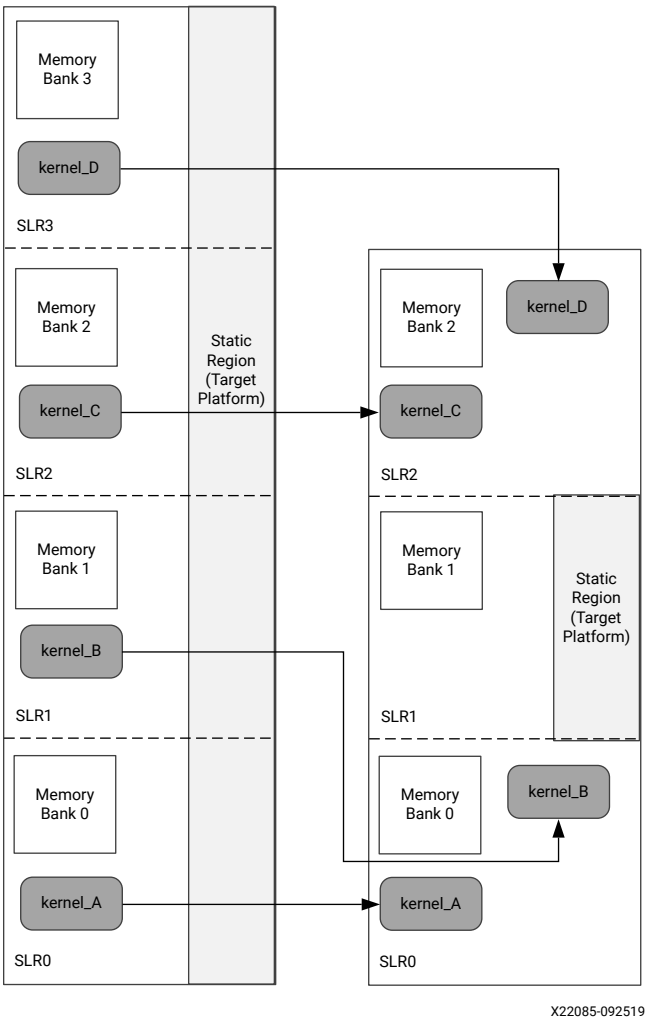
When placing kernels, AMD recommends assigning the specific DDR memory bank that the kernel will connect to using the `connectivity.sp` config option as described in Mapping Kernel Ports to Memory.

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 63: **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.cfg ...
```

Specifying Kernel DDR Interfaces

You must 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 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 Timing Closure Quick Reference Guide* (UG1292) 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 AMD 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 AMD Vivado™ tools using the `[advanced]` directives of the `v++` configuration file specified by the `--config` option. Refer to Managing Vivado Synthesis, Implementation, and Timing Closure 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 usage was higher than the typical guideline:
  - LUT usage was higher than 70%
  - DSP, RAMB, and UltraRAM usage was higher than 80%
  - FD usage was higher than 50%
- High effort compilation strategies were needed to close timing.

The usage 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 Memory, while verifying that any floorplan constraint ensures the following:

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

For designs with overall high usage, 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 Guide for FPGAs and SoCs* (UG949).

OpenCL Programming

OpenCL Host Application

In the AMD Vitis™ core development kit, host code is written in C or C++ language using the Xilinx Runtime (XRT) API or industry standard OpenCL™ API. The XRT native API is described on the XRT site at https://xilinx.github.io/XRT/master/html/xrt_native_apis.html. The Vitis core development kit supports the OpenCL 1.2 API as described at https://www.khronos.org/registry/OpenCL/specs/opencl-1.2.pdf. XRT extensions to OpenCL are described at https://xilinx.github.io/XRT/master/html/opencl_extension.html.

TIP: *The code examples shown in this text use the OpenCL C language 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.

TIP: *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. For details and installation instructions, refer to OpenCL Installable Client Driver Loader in the Data Center Acceleration using Vitis (UG1700).*

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.

OpenCL Installable Client Driver Loader

The AMD Vitis™ environment 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. The ICD Loader acts as a supervisor for all installed platforms, and provides a standard handler for all API calls.

Applications can choose an OpenCL platform from the list of installed platforms. Based on the platform ID specified by the application, the ICD dispatches the OpenCL host calls to the right runtime.

TIP: *This is an optional package to install if your system has or uses multiple versions of the OpenCL library.*

AMD does not provide the OpenCL ICD library, so the following must be used to install the library on your system.

Ubuntu

On Ubuntu the ICD library is packaged with the distribution. Install the following packages:

```
sudo apt-get install ocl-icd-libopencl1
sudo apt-get install opencl-headers
sudo apt-get install ocl-icd-opencl-dev
```

RHEL/CentOS

For RHEL/CentOS use EPEL to install the following packages:

```
sudo yum install ocl-icd
sudo yum install ocl-icd-devel
sudo yum install opencl-headers
```

***Note*:** Refer to https://fedoraproject.org/wiki/EPEL for information on installing EPEL if needed.

Setting Up the OpenCL Environment

The host code in the Vitis core development kit follows the OpenCL programming paradigm. To setup the runtime 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 Vitis Examples are written using the C++ API. For more information on this C++ wrapper API, refer to https://www.khronos.org/registry/OpenCL/specs/opencl-cplusplus-1.2.pdf.*

Platform

Upon initialization, the host application needs to identify a platform composed of one or more AMD devices. The following code fragment shows a common method of identifying an AMD 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. Then, `clGetPlatformInfo` is used to identify AMD 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 an AMD platform is found, the application needs to identify the corresponding AMD devices.

The following code demonstrates finding all the AMD 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 `platform_id` and `CL_DEVICE_TYPE_ACCELERATOR` to receive all the available AMD 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 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 an AMD device that communicates 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 AMD device. AMD recommends creating only one context per device or sub-device. However, the host program must 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 is 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.

RECOMMENDED: *For improved performance, AMD recommends using a single out-of-order command queue and manage event dependencies and synchronizations explicitly, instead of using multiple command queues.*

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

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 on 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 runtime environment, such as identifying devices, creating the context, command queue, and program, the host application must identify the kernels that execute on the device, and set up the kernel arguments.

The OpenCL API `clCreateKernel` is 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 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 in the following example for setting kernel arguments for two scalar and two buffer arguments.

```
// Create memory buffers
cl_mem dev_buf1 = clCreateBuffer(context, CL_MEM_WRITE_ONLY |
CL_MEM_USE_HOST_PTR, size, &host_mem_ptr1, NULL);
cl_mem dev_buf2 = clCreateBuffer(context, CL_MEM_READ_ONLY |
CL_MEM_USE_HOST_PTR, 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 enqueuing the kernel, it is recommended to 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.*

For all kernel buffer arguments you must allocate the buffer on the device global memories. However, sometimes the content of the buffer is not required before the start of the kernel execution. For example, the output buffer content will only be populated during the kernel execution, and hence it is not important prior to kernel execution. In this case, specify `clEnqueueMigrateMemObject` with the `CL_MIGRATE_MEM_OBJECT_CONTENT_UNDEFINED` flag so that migration of the buffer will not involve the DMA operation between the host and the device, thus improving performance.

Buffer Allocation on the Device

By default, when kernels are linked to the platform the memory interfaces from all the kernels are connected to a single default global memory bank. As a result, only a single compute unit (CU) can transfer data to and from the global memory bank at one time, limiting the overall performance of the application.

If the device 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 memory interface connection for a kernel during linking. The method for performing this is discussed in detail in Mapping Kernel Ports to 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, such as `clEnqueueMigrateMemObject`.*

Buffer Creation and Data Transfer

Interactions between the host program and hardware kernels rely on creating buffers and transferring data to and from the memory in the device. This process makes use of functions like `clCreateBuffer` and `clEnqueueMigrateMemObjects`.

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

There are two methods for allocating memory buffers, and transferring data:

1. Letting XRT Allocate Buffers
2. Using Host Pointer Buffers

In the case where XRT allocates the buffer, use `enqueueMapBuffer` to capture the buffer handle. In the second case, allocate the buffer directly with `CL_MEM_USE_HOST_PTR`, so you do not need to capture the handle.

> **TIP:** *Do not use `CL_MEM_USE_HOST_PTR` for embedded platforms. Embedded platforms require contiguous memory allocation and must use the `CL_MEM_ALLOC_HOST_PTR` method, as described in Letting XRT Allocate Buffers.*

There are a number of coding practices you can adopt to maximize performance and fine-grain control. The OpenCL API supports additional commands for reading and writing buffers. For example, you can use `clEnqueueWriteBuffer` and `clEnqueueReadBuffer` commands in place of `clEnqueueMigrateMemObjects`. However, some of these commands have different effects that must be understood when using them. For example, `clEnqueueReadBufferRect` can read a rectangular region of a buffer object to the host application, but it does not transfer the data from the device global memory to the host. You must first use `clEnqueueReadBuffer` to transfer the data from the device global memory, and then use `clEnqueueReadBufferRect` to read the desired rectangular portion into the host application.

Letting XRT Allocate Buffers

On data center platforms, it is more efficient to allocate memory aligned on 4k page boundaries. On embedded platforms it is more efficient to perform contiguous memory allocation. In either case, you can let the XRT allocate host memory when creating the buffers. This is done by using the `CL_MEM_ALLOC_HOST_PTR` flag when creating the buffers, and then mapping the allocated memory to user-space pointers using `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:

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

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


cl::Buffer in1_buf(context, CL_MEM_ALLOC_HOST_PTR | CL_MEM_READ_ONLY,
sizeof(int) * DATA_SIZE, NULL, &err);

// 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,nullpt
```

```
r,nullptr,&err);
auto host_read_ptr =
clEnqueueMapBuffer(queue,dev_mem_write_ptr,true,CL_MAP_READ,0,bytes,0,nullpt
r,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
```

Using Host Pointer Buffers

IMPORTANT! *Using `CL_MEM_USE_HOST_PTR` is not recommended for embedded platforms. Embedded platforms require contiguous memory allocation and must use the `CL_MEM_ALLOC_HOST_PTR` method, as described in Letting XRT Allocate Buffers.*

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. 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.

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.

AMD 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 are successfully migrated.

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

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. 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 | CL_MEM_USE_HOST_PTR,
                        sizeof(int) * number_of_words, host_mem_ptr, NULL);

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

Sub-Buffers

Though not very common, using sub-buffers 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 only reads 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,,,);

// Retrive 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
```

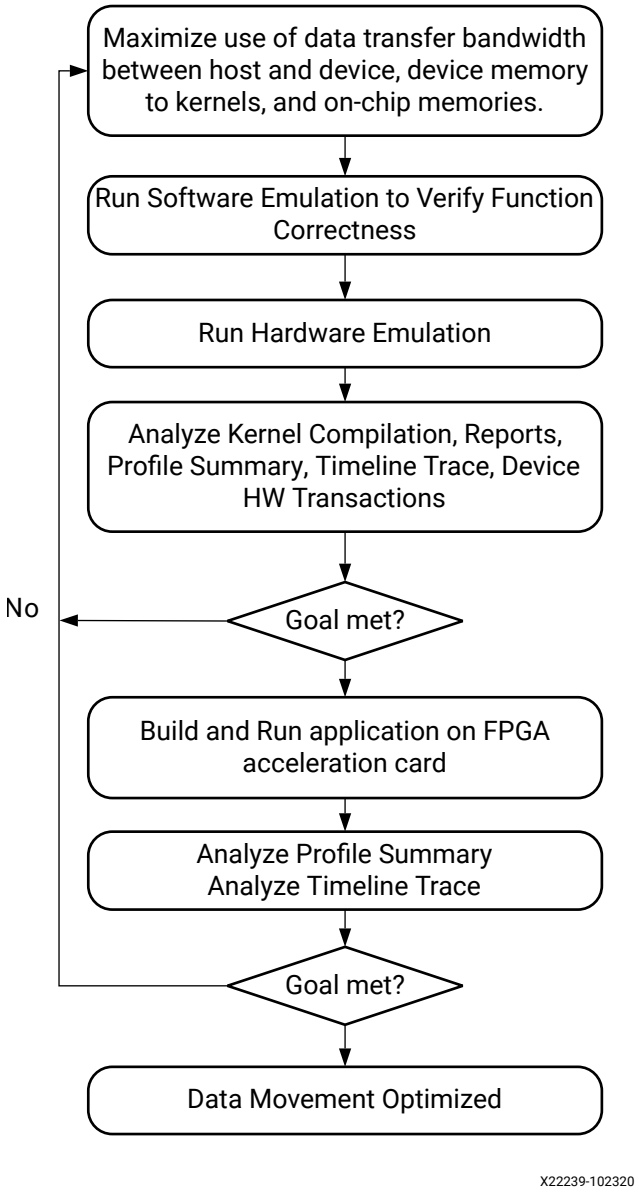
Device Buffer Shared by Multiple Memory Ports or Multiple Kernels

Sometimes memory ports of kernels only require small amounts of data. However, when managing small sized buffers, transferring small amounts of data can 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.

Optimizing Data Movement

Figure 64: **Optimizing Data Movement Flow**



In the OpenCL execution model, all data is transferred from the host main memory to the global device memory first, and then from the global device memory to the kernel for computation. The computation results are written back from the kernel to the global device memory, and lastly from the global memory to the host main memory. A key factor in determining strategies for kernel data movement optimization is understanding how data can be efficiently moved around between different level of memories maximizing the efficient use of bandwidth on all the memory interfaces.

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. Focus on modifying the data transfer logic in the host and kernel code during this optimization step. The goal is to maximize the system level data throughput by maximizing data transfer bandwidth and device global memory bandwidth usage. It usually takes multiple iterations of running software emulation, hardware emulation, in addition to execution in hardware to achieve optimum performance.

Overlapping Data Transfers with Kernel Computation

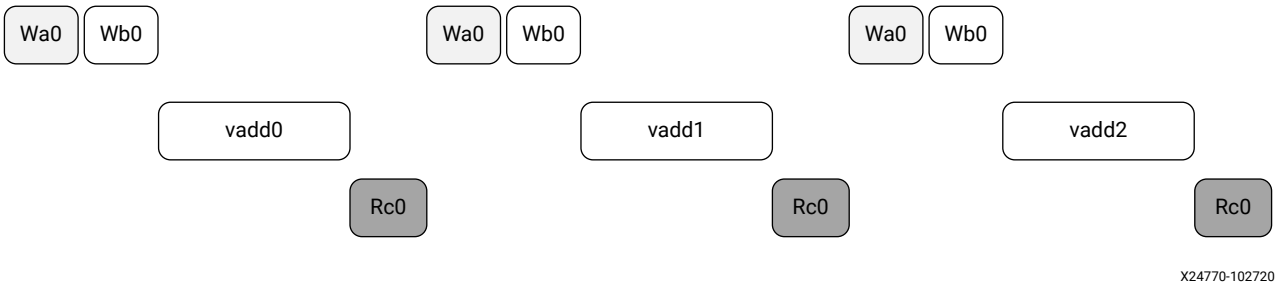
Applications, such as database analytics, have a much larger data set than can be stored in the available global device memory on the acceleration device. They require the complete data to be transferred and processed in blocks. Techniques that overlap the data transfers with the computation are critical to achieve high performance for these applications.

In the following example, the kernel processes two arrays by adding them together and writing to output. 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 a simple in-order command queue without data transfer optimization, the overall execution timeline trace looks similar to the one shown below:

Figure 65: **Host View of Tasks**

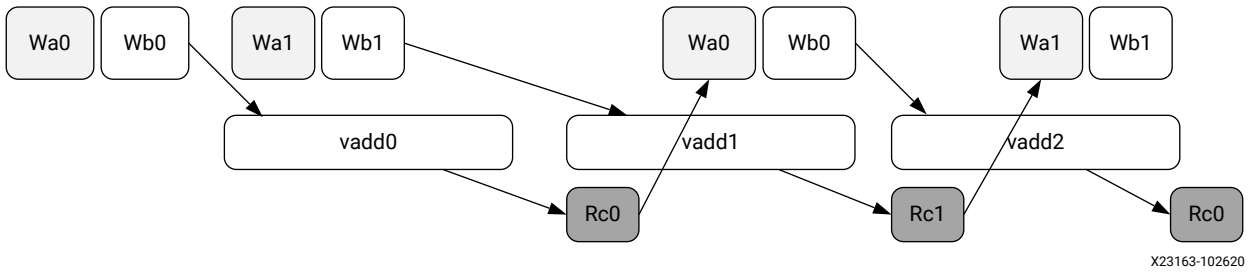


Using an out-of-order command queue, data transfer and kernel execution can overlap as illustrated in the figure below. In the host code for this example, double buffering is used for all buffers so that the kernel can process one set of buffers while the host can operate on the other set of buffers.

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`, and `copy` commands on memory objects, or user events created using `clCreateUserEvent`.

You can ensure an operation has completed by querying the events returned by these commands. The arrows in the figure below show how event triggering can be set up to achieve optimal performance.

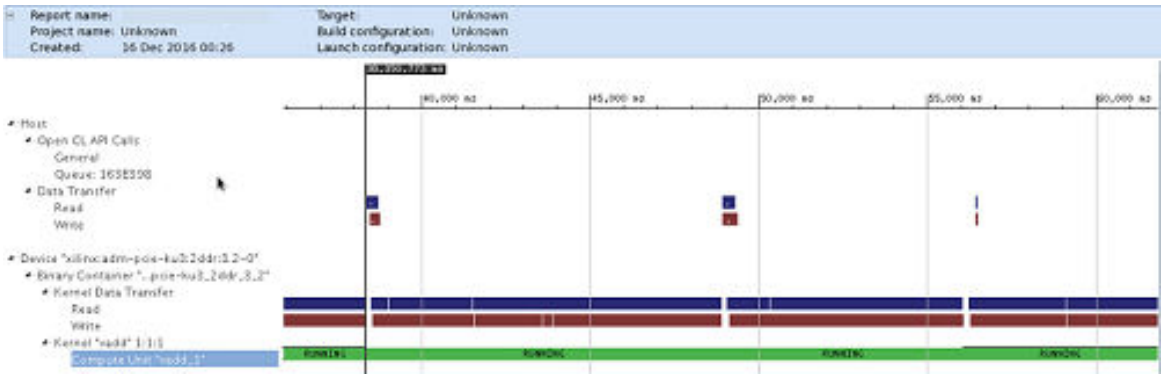
Figure 66: **Event Triggering Setup**



In the example, 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.

The Timeline Trace view below clearly shows that the data transfer time is completely hidden, while the compute unit `vadd_1` is running constantly.

Figure 67: **Data Transfer Time Hidden in Timeline Trace 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 must 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.

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 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 Developing PL Kernels using C++.

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 AMD FPGA.

IMPORTANT! *Though using `clEnqueueNDRangeKernel` is supported (only for OpenCL kernel), AMD 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. Sometimes multiple kernels can be designed performing different tasks on the FPGA in parallel, requiring multiple enqueue commands. Or the host application can be receiving data over time, and not all the data can be processed at one time. Therefore, depending on the situation and application, you might need to break the data and the task of the kernel into multiple `clEnqueueTask` commands. In this case, an out-of-order command queue, or an in-order command queue can determine how the kernel tasks are processed as explained in Command Queues. In addition, multiple kernel tasks can be implemented as blocking events, or non-blocking events as described in Event Synchronization. These can all affect the performance of the design.

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.

Reducing Overhead of Kernel Enqueing

The OpenCL-based execution model supports data parallel and task parallel programming models. An OpenCL host generally needs to call different kernels multiple times. These calls are enqueued in a command queue, either in a certain sequence, or in an out-of-order command queue. Then depending on the availability of compute resources and task data they get scheduled for execution on the device.

Kernel calls can be enqueued for execution on a command queue using `clEnqueueTask`. The dispatching process is executed on the host processor. The dispatcher invokes kernel execution after transferring the kernel arguments to the accelerator running on the device. The dispatcher uses a low-level Xilinx Runtime (XRT) library for transferring kernel arguments and issuing trigger commands for starting the compute. The overhead of dispatching the commands and arguments to the accelerator can be between 30 µs and 60 µs, depending on the number of arguments set for the kernel. You can reduce the impact of this overhead by minimizing the number of times the kernel needs to be executed, and minimizing calls to `clEnqueueTask`. Ideally, finish all the compute in a single call to `clEnqueueTask`.

You can minimize the calls to `clEnqueueTask` by batching your data and invoking the kernel one time, with a loop wrapped around the original implementation to avoid the overhead of multiple enqueue calls. It can also improve data transfer performance between the host and accelerator, by transferring fewer large data packets rather than many small data packets. For more information on reducing overhead on kernel execution, see Kernel Execution.

The following example shows a simple kernel with given work or data size to process.

```
#define SIZE 256
extern "C" {
    void add(int *a , int *b, int inc){
        int buff_a[SIZE];
        for(int i=0;i<size;i++)
        {
            buff_a[i] = a[i];
        }
        for(int i=0;i<size;i++)
        {
            b[i] = a[i]+inc;
        }
    }
}
```

The following example shows the same simple kernel optimized to process batched data. Depending on the `num_batches` argument the kernel can process multiple inputs of size 256 in a single call and avoid the overhead of multiple `clEnqueueTask` calls. The host application changes to allocate data and buffers in chunks of `SIZE * num_batches`, essentially batching the memory allocation and transfer of data between the host global and device memory.

```
#define SIZE 256
extern "C" {
    void add(int *a , int *b, int inc, int num_batches){
        int buff_a[SIZE];
        for(int j=0;j<num_batches;j++)
        {
            for(int i=0;i<size;i++)
            {
                buff_a[i] = a[i];
            }
            for(int i=0;i<size;i++)
            {
                b[i] = a[i]+inc;
            }
       }
    }
}
```

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 is 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.

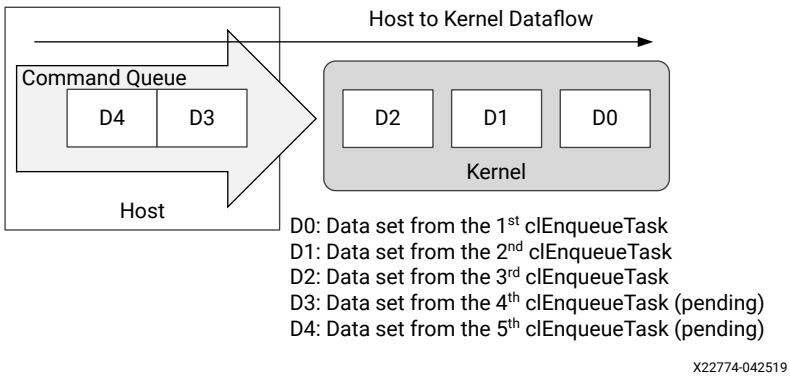
To understand this approach, assume a kernel has only one CU on the FPGA, and the host application enqueues the kernel multiple times with different sets of data. As shown in Using Host Pointer Buffers, the host application can migrate data to the device global memory ahead of the kernel execution, thus 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 68: **Host to Kernel Dataflow**



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.*

Enabling 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. To support host-to-kernel dataflow, the kernel has to implement the `ap_ctrl_chain` protocol using the pragma HLS interface 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 the Pipelining Loops chapter of the Vitis HLS User Guide (UG1399), or pipelined at the task-level as discussed in the Dataflow Style Modeling section in the Vitis HLS User Guide (UG1399).*

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 `connectivity.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 instance in a group of symmetrical CUs.
- **Asymmetrical:** CUs are considered asymmetrical when they do not have exactly the same `connectivity.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 CUs are a symmetrical group, having symmetrical connection on the device. Similarly, CUs K4 and K5 form a second symmetrical CU group. The following code segment 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);
}
```

Compute Unit Scheduling

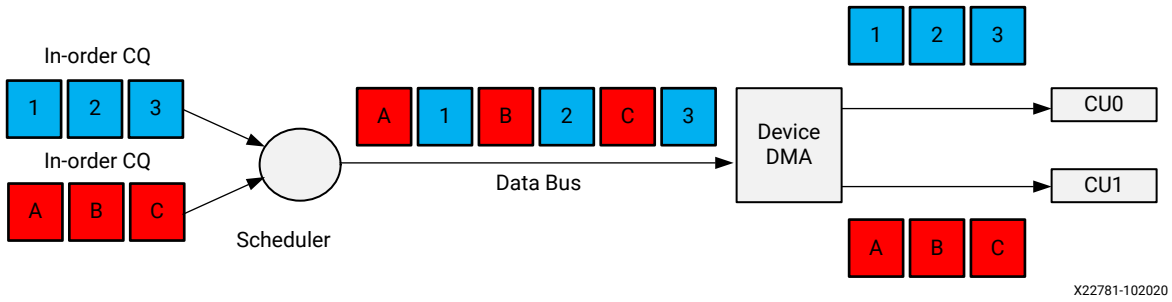
Scheduling kernel operations is key to overall system performance. This becomes even more important when implementing multiple compute units (of the same kernel or of different kernels). This section examines the different command queues responsible for scheduling the kernels.

Multiple In-Order Command Queues

RECOMMENDED: *For improved performance, AMD recommends using a single out-of-order command queue and manage event dependencies and synchronizations explicitly, instead of using multiple 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 69: **Example with Two In-Order Command Queues**



The following is code extracted from `host.cpp` of the concurrent_kernel_execution example that sets up multiple in-order command queues and enqueues commands into each queue:

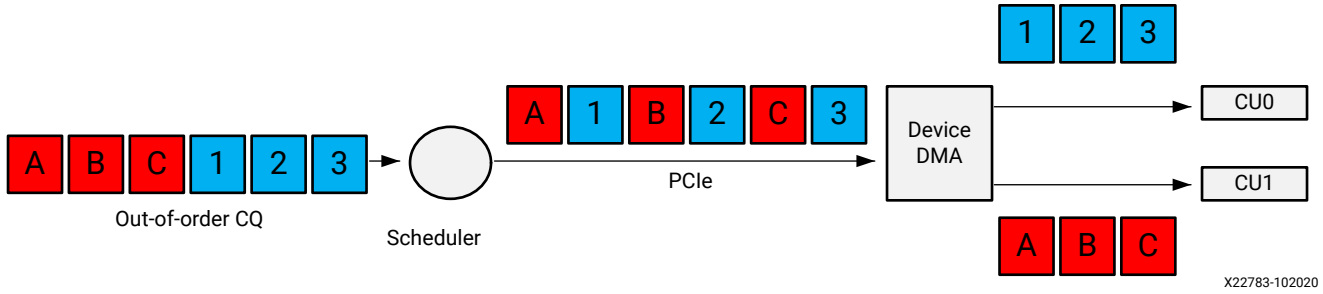
```
   OCL_CHECK(
        err,
        cl::CommandQueue ooo_queue(context,
                                   device,
                                   CL_QUEUE_PROFILING_ENABLE |

CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE,
                                   &err));
...
    printf("[OOO Queue]: Enqueueing scale kernel\n");
    OCL_CHECK(
        err,
```

```
        err = ooo_queue.enqueueTask(
            kernel_mscale,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.enqueueTask(
                  kernel_madd,
                  &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.enqueueTask(
                  kernel_mmult,
                  nullptr,
                  &ooo_events[2]));
    set_callback(ooo_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 70: **Example with Single Out-of-Order Command Queue**



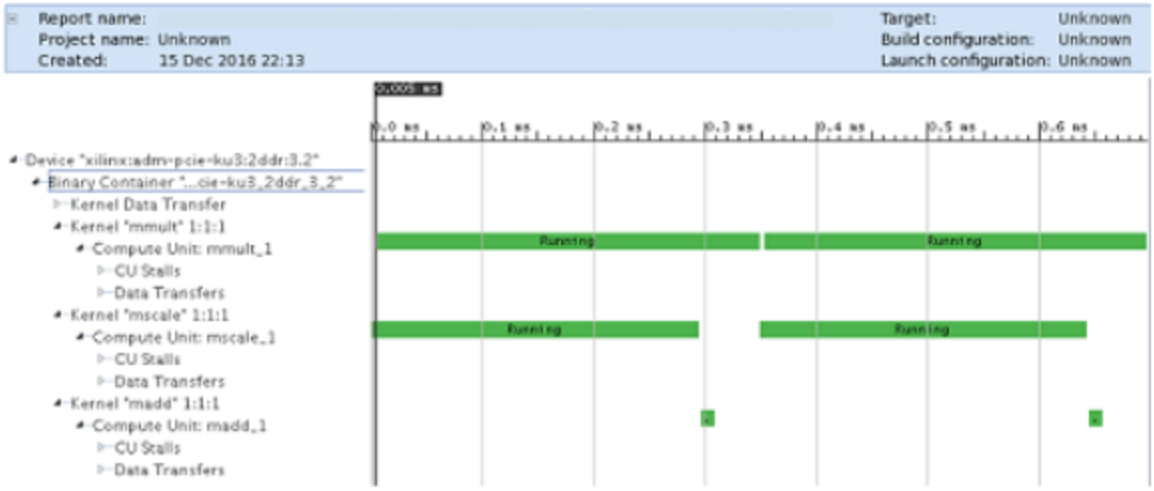
The following is code extracted from `host.cpp` of the concurrent_kernel_execution example that sets up a single out-of-order command queue and enqueues commands as needed:

```
    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.enqueueTask(
            kernel_mscale,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.enqueueTask(
                  kernel_madd,
                  &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.enqueueTask(
                  kernel_mmult,
                  nullptr,
                  &ooo_events[2]));
    set_callback(ooo_events[2], "matrix multiplication");
```

The Timeline Trace view shows 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 71: **Timeline Trace View Showing mult_1 Running with mscale_1 and madd_1**



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*:**

1. The `clFinish` API has been explicitly used to block the host execution until the kernel execution is finished. This is necessary because otherwise the host can attempt to read back from the FPGA buffer too early and might 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.

Debugging OpenCL Kernels

For OpenCL kernels, additional runtime checks can be performed during software emulation. These additional checks include:

- Checking whether an OpenCL 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 Vitis compiler options that are enabled through the `--advanced` compiler option as described in --advanced Options, using the following command syntax:

```
--advanced.param compiler.fsanitize=address,memory
```

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`.

The `fsanitize` directive can also be specified in a config file, as follows:

```
[advanced]
#param=<param_type>:<param_name>.<value>
param=compiler.fsanitize=address,memory
```

The config file is specified on the `v++` command line:

```
v++ -l -t sw_emu --config ./advanced.cfg -o bin_kernel.xclbin
```

Refer to the Vitis Compiler Configuration File in the *Vitis Reference Guide* (UG1702) for more information on the `--config` option.

Assigning DDR Bank in Host Code

IMPORTANT! *This is optional and only needed in specific cases as described below.*

During the Vitis tool flow, the kernel port to memory bank connectivity can be established using the `--connectivity.sp` switch as described in Mapping Kernel Ports to Memory. The `xclbin` generated by `v++` contains the information about the kernel port to memory connectivity so that XRT can allocate buffers appropriately. When a buffer is created in the host code, XRT automatically assigns the buffer to memory from the kernel `xclbin`, and manages the buffers internally. If a single kernel port is connected to multiple memory banks, XRT always starts from the lower numbered bank.

In most cases, this approach is sufficient. However, in some specific cases you might need to manually assign the buffer location (or special property) in the host code. For this purpose, the AMD OpenCL vendor extension provides a buffer extension called `CL_MEM_XRT_PTR_XILINX` to specifically manage bank assignment in the host code. The following code example shows the required header file and code for assigning input and output buffers to DDR bank 0 and bank 1:

```
#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;
    }
…
}
```

The extension pointer `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, HBM, 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.

Here are some specific cases where you might want to use the extension pointer:

- **P2P Buffer:** For an explanation and example, refer to https://xilinx.github.io/XRT/master/html/p2p.html.
- **Host-Memory Buffer:** For an explanation and example, refer to https://xilinx.github.io/XRT/master/html/hm.html.
- **Allocating the host buffer to a specific bank when the kernel port is connected to multiple banks:** For example, DDR[0:1]. This use case is described in detail in the Using Multiple DDR Banks lab of the *Vitis Optimizing Accelerated FPGA Applications: Bloom Filter Example* tutorial.

Example of Allocating the Host Buffer to a Specific Bank

An example of the third case listed above, where you might need to use `cl_mem_ext_ptr_t`, is when the host and kernel are both accessing the DDR bank simultaneously, and you would like to split the data so that kernel and host access memory banks in a ping-pong fashion. When the host is writing/reading to a specific memory bank, the kernel is writing/reading from another bank so that these host/kernel accesses don't compete and impact performance. For this scenario, you must manage the buffer allocation yourself.

The kernel ports in the `xclbin` are connected to DDR bank1 and bank2, and reading the data from these banks alternatively. The connectivity is established during linking by the Vitis compiler using the `--connectivity.sp` switch:

```
[connectivity]
sp=runOnfpga_1.input_words:DDR[1:2]
```

From the host code, you can send the `input_words` data to DDR banks 1 and 2 alternatively. Two AMD extension pointer (`cl_mem_ext_ptr_t`) objects are created as shown in the example code below. The object flags will determine which DDR bank each buffer will be assigned to for the kernel to access. The kernel argument can be set to `input_words[0]` and `input_words[1]` for consecutive kernel enqueues.

```
#include <CL/cl_ext.h>
…
int main(int argc, char** argv)
{
cl_mem_ext_ptr_t buffer_words_ext[2];

buffer_words_ext[0].flags = 1 | XCL_MEM_TOPOLOGY; // DDR[1]
buffer_words_ext[0].param = 0;
buffer_words_ext[0].obj   = input_doc_words;
buffer_words_ext[1].flags = 2 | XCL_MEM_TOPOLOGY; // DDR[2]
buffer_words_ext[1].param = 0;
buffer_words_ext[1].obj   = input_doc_words;
…
```

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 `ocl_kernels` category from the Vitis Accel 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 HW Interfaces for more information.

  The following is a code snippet from the `gmem_2banks` 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 Memory for more information.

Post-Processing and FPGA Cleanup

At the end of the host code, all the allocated resources must be released by using proper release functions. If the resources are not properly released, the Vitis core development kit might not 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);
```

Compiling and Linking the Host Application

Building OpenCL API Host Code for x86

The AMD Vitis™ application acceleration development flow also supports the use of OpenCL API to program your host application. Building OpenCL applications using `g++` uses the following command line:

```
g++ -g -std=c++1y -I$XILINX_XRT/include -L$XILINX_XRT/lib -o host.exe
host.cpp \
-lOpenCL -pthread
```

The only difference is the use of the `OpenCL` library for the OpenCL API in place of the `xrt_coreutil` library for the XRT native API.

***Note*:** In Vitis Accel_Examples you can see the addition of `xcl2.cpp` source file, and the `-I../xcl2` 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.

AMD

Appendix A

Additional Resources and Legal Notices

Finding Additional Documentation

Technical Information Portal

The AMD Technical Information Portal is an online tool that provides robust search and navigation for documentation using your web browser. To access the Technical Information Portal, go to https://docs.amd.com.

Documentation Navigator

Documentation Navigator (DocNav) is an installed tool that provides access to AMD Adaptive Computing documents, videos, and support resources, which you can filter and search to find information. To open DocNav:

- From the AMD Vivado™ IDE, select **Help → Documentation and Tutorials**.
- On Windows, click the **Start** button and select **Xilinx Design Tools → DocNav**.
- At the Linux command prompt, enter `docnav`.

***Note*:** For more information on DocNav, refer to the *Documentation Navigator User Guide* (UG968).

Design Hubs

AMD 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.
- Go to the Design Hubs web page.

Support Resources

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

References

These documents provide supplemental material useful with this guide.

1. *Vitis Software Platform Release Notes* (UG1742)
2. *Embedded Design Development Using Vitis* (UG1701)
3. *Vitis Reference Guide* (UG1702)
4. *Introduction to FPGA Design with Vivado High-Level Synthesis* (UG998)
5. *Vitis Unified Software Platform Documentation: Embedded Software Development* (UG1400)
6. *Vitis High-Level Synthesis User Guide* (UG1399)
7. *UltraFast Design Methodology Timing Closure Quick Reference Guide* (UG1292)
8. *UltraFast Design Methodology Guide for FPGAs and SoCs* (UG949)
9. *RAMA LogiCORE IP Product Guide* (PG310)
10. *Vitis Unified Software Platform Tutorials Landing Page* (UG1605)
11. *Vivado Design Suite User Guide: Using Tcl Scripting* (UG894)
12. *Vivado Design Suite Tcl Command Reference Guide* (UG835)
13. *Vivado Design Suite User Guide: Creating and Packaging Custom IP* (UG1118)
14. *Alveo U55C Data Center Accelerator Cards Data Sheet* (DS978)
15. *PetaLinux Tools Documentation: Reference Guide* (UG1144)
16. *Vivado Design Suite User Guide: Logic Simulation* (UG900)
17. *Getting Started with Alveo Data Center Accelerator Cards* (UG1301)
18. *Vivado Design Suite User Guide: Designing IP Subsystems using IP Integrator* (UG994)
19. *Alveo U50 Data Center Accelerator Cards Data Sheet* (DS965)
20. *Xilinx Stacked Silicon Interconnect Technology Delivers Breakthrough FPGA Capacity, Bandwidth, and Power Efficiency* (WP380)
21. *Vivado Design Suite User Guide: Implementation* (UG904)
22. *Vivado Design Suite User Guide: Programming and Debugging* (UG908)

Revision History

The following table shows the revision history for this document.

Section	Revision Summary
11/20/2025 Version 2025.2	
General updates	Renamed `xbutil` with `xrt-smi` throughout the document.
Timeline Trace	System Timeline Trace (EA, Vitis 2025.2) unifies clock-aligned Host, PL, and AIE events for precise correlation and debugging.
05/29/2025 Version 2025.1	
General updates	Removed references to OpenCL Kernel and Vitis Tutorials (Hardware Acceleration)

Please Read: Important Legal Notices

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