

# Vitis Unified Software Platform Documentation

## *Application Acceleration Development*

UG1393 (v2022.2) December 7, 2022

Xilinx is creating an environment where employees, customers, and partners feel welcome and included. To that end, we're removing non-inclusive language from our products and related collateral. We've launched an internal initiative to remove language that could exclude people or reinforce historical biases, including terms embedded in our software and IPs. You may still find examples of non-inclusive language in our older products as we work to make these changes and align with evolving industry standards. Follow this [link](#) for more information.





# Table of Contents

<b>Section I: Getting Started with Vitis.....</b>	<b>12</b>
<b>    Chapter 1: Navigating Content by Design Process.....</b>	<b>13</b>
<b>    Chapter 2: Vitis Software Platform Release Notes.....</b>	<b>14</b>
What's New.....	14
Supported Platforms.....	14
Embedded GNU Toolchain Details.....	15
Changed Behavior.....	16
Known Issues.....	16
<b>    Chapter 3: Installation.....</b>	<b>17</b>
Installation Requirements.....	17
Vitis Software Platform Installation.....	19
<b>Section II: Introduction to Vitis Flows.....</b>	<b>23</b>
<b>    Chapter 4: Introduction to Vitis Tools for Embedded System Designers.....</b>	<b>24</b>
Terminology for Embedded System Design.....	25
Fixed Platforms versus Extensible Platforms.....	26
PS Software Development.....	29
Vitis PL Kernel Development Flow.....	30
Versal AI Engine Graph Development.....	31
Building and Packaging the System.....	33
Next Steps.....	35
<b>    Chapter 5: Introduction to Data Center Acceleration for Software Programmers.....</b>	<b>36</b>
Terminology.....	37
Working with Alveo Accelerator Cards.....	38

A Sample Application.....	40
Vitis Application Development Flow.....	47
Best Practices for Kernel Development.....	52
Best Practices for Host Programming.....	55
Next Steps.....	57
<b>Chapter 6: Introduction to Data Center Acceleration for RTL Designers.....</b>	<b>58</b>
Terminology.....	59
Vitis Development Flow for RTL Designers.....	60
Using Alveo Accelerator Cards.....	62
Differences between Vivado and Vitis Development Flows.....	64
Creating and Packaging RTL Kernels.....	65
Building Kernels and Managing Implementation .....	67
Writing Host Applications with XRT API.....	67
Next Steps.....	68
<b>Chapter 7: Tutorials and Examples.....</b>	<b>70</b>
<b>Section III: Developing Applications.....</b>	<b>71</b>
<b>Chapter 8: Programming Model.....</b>	<b>72</b>
Design Topology.....	72
<b>Chapter 9: Writing the Software Application.....</b>	<b>73</b>
Specifying the Device ID and Loading the XCLBIN.....	74
Setting Up XRT-Managed Kernels and Kernel Arguments.....	74
Transferring Data between Software and PL Kernels.....	76
Working with XRT-Managed Kernels.....	77
Working with User-Managed Kernels.....	78
Working with AI Engine Graphs.....	82
Summary.....	83
<b>Chapter 10: PL Kernel Properties.....</b>	<b>84</b>
SW-Controllable Kernels.....	84
Non-Software Controlled Kernels.....	87
Clock and Reset Requirements.....	87
<b>Chapter 11: Developing PL Kernels using C++.....</b>	<b>89</b>

<b>Chapter 12: Packaging RTL Kernels.....</b>	<b>90</b>
Requirements of an RTL Kernel.....	90
Creating User-Managed RTL Kernels.....	94
RTL Kernel Development Flow.....	95
Design Recommendations for RTL Kernels.....	101
<b>Chapter 13: Programming Versal AI Engines.....</b>	<b>104</b>
<b>Section IV: Building and Running the Application.....</b>	<b>105</b>
<b>Chapter 14: Setting Up the Vitis Environment.....</b>	<b>106</b>
<b>Chapter 15: Build Targets.....</b>	<b>107</b>
Software Emulation.....	107
Hardware Emulation.....	108
System Hardware Target.....	109
<b>Chapter 16: Building the Software Application.....</b>	<b>110</b>
Compiling and Linking for x86.....	110
Compiling and Linking for Arm.....	111
<b>Chapter 17: Building the Device Binary.....</b>	<b>114</b>
Compiling C/C++ PL Kernels.....	115
Compiling AI Engine Graph Applications.....	116
Linking the Kernels.....	117
Managing Vivado Synthesis and Implementation Results.....	136
Controlling Report Generation.....	145
<b>Chapter 18: Packaging the System.....</b>	<b>147</b>
Packaging for Embedded Platforms.....	147
Packaging for Data Center Platforms.....	149
<b>Chapter 19: Running the Application.....</b>	<b>150</b>
<b>Section V: Simulating the Application with the Emulation Flow.....</b>	<b>152</b>
<b>Chapter 20: Running Emulation Targets.....</b>	<b>154</b>
<b>Chapter 21: Data Center vs. Embedded Platforms.....</b>	<b>155</b>

<b>Chapter 22: QEMU.....</b>	<b>156</b>
<b>Chapter 23: Running Emulation on Data Center Accelerator Cards.....</b>	<b>157</b>
<b>Chapter 24: Running Emulation on an Embedded Processor Platform.....</b>	<b>159</b>
Running Emulation on an Embedded Processor Platform Using PS on x86.....	161
<b>Chapter 25: Speed and Accuracy of Hardware Emulation.....</b>	<b>163</b>
<b>Chapter 26: Working with Simulators in Hardware Emulation... </b>	<b>165</b>
Simulator Support.....	165
Using the Simulator Waveform Viewer .....	167
AXI Transactions Display in XSIM Waveform.....	168
Generating Test Vectors for Vitis HLS during Hardware Emulation.....	168
<b>Chapter 27: Working with Functional Model of the HLS Kernel..</b>	<b>170</b>
<b>Chapter 28: Working with SystemC Models.....</b>	<b>173</b>
Coding a SystemC Model.....	173
Creating the XO.....	176
Linking with the v++ Command.....	176
Xilinx TLM – SystemC Library for ESL Modelling.....	176
<b>Chapter 29: Debug Techniques in Hardware Emulation.....</b>	<b>179</b>
<b>Chapter 30: Working with I/O Traffic Generators.....</b>	<b>182</b>
Adding Traffic Generators to Your Design.....	182
AXI4-Stream I/O Model for Streaming Traffic through Python/C++/Verilog.....	183
AXI4 Memory Map External Traffic through Python/C++.....	193
<b>Section VI: Profiling and Debugging the Application.....</b>	<b>197</b>
<b>Chapter 31: Profiling the Application.....</b>	<b>198</b>
Enabling Profiling in Your Application.....	199
Guidance.....	210
System Estimate Report.....	214

HLS Report.....	218
Profile Summary Report.....	220
Timeline Trace.....	231
Detailed Kernel Trace.....	235
Waveform View and Live Waveform Viewer.....	240
<b>Chapter 32: Debugging Applications and Kernels.....</b>	<b>246</b>
Debugging Flows.....	246
Debugging in Software Emulation.....	247
Debugging in Hardware Emulation.....	256
Debugging During Hardware Execution.....	261
Debugging on Embedded Processor Platforms.....	285
Example of Command Line Debugging.....	290
<b>Section VII: Vitis Commands and Utilities.....</b>	<b>293</b>
<b>Chapter 33: v++ Command.....</b>	<b>294</b>
v++ General Options.....	294
--advanced Options.....	310
--clock Options.....	316
--connectivity Options.....	319
--debug Options.....	324
--drc Options.....	326
--hls Options.....	327
--linkhook Options.....	330
--package Options.....	331
--profile Options.....	337
--vivado Options.....	340
Vitis Compiler Configuration File.....	343
Using the Message Rule File.....	346
<b>Chapter 34: emconfigutil Utility.....</b>	<b>349</b>
<b>Chapter 35: kernelinfo Utility.....</b>	<b>351</b>
Kernel Definition.....	352
Arguments.....	352
Ports.....	353
<b>Chapter 36: launch_emulator Utility.....</b>	<b>354</b>

Versal PS and PMC Arguments for QEMU.....	360
Zynq UltraScale+ MPSoC PS and PMU Arguments for QEMU.....	363
Zynq-7000 PS Arguments for QEMU.....	366
<b>Chapter 37: manage_ipcache Utility.....</b>	<b>368</b>
<b>Chapter 38: package_xo Command.....</b>	<b>369</b>
RTL Kernel XML File.....	371
<b>Chapter 39: platforminfo Utility.....</b>	<b>375</b>
Basic Platform Information.....	376
Hardware Platform Information.....	377
Interface Information.....	377
Clock Information.....	377
Valid SLRs.....	378
Resource Availability.....	378
Memory Information.....	379
Feature ROM Information.....	380
Software Platform Information.....	380
Platforminfo for xilinx_zcu104_base_202010_1.....	381
<b>Chapter 40: xbutil Utility.....</b>	<b>383</b>
<b>Chapter 41: xbmgmt Utility.....</b>	<b>384</b>
<b>Chapter 42: xclbinutil Utility.....</b>	<b>385</b>
xclbin Information.....	386
Hardware Platform Information.....	387
Clocks.....	387
Memory Configuration.....	387
Kernel Information.....	388
Tool Generation Information.....	389
<b>Chapter 43: xrt.ini File.....</b>	<b>391</b>
<b>Chapter 44: HLS Pragmas.....</b>	<b>401</b>
<b>Section VIII: Using the Vitis Analyzer.....</b>	<b>403</b>
<b>Chapter 45: Working with Summary Reports.....</b>	<b>407</b>

Configuring the Vitis Analyzer.....	409
<b>Chapter 46: Vitis Analyzer GUI and Window Manager.....</b>	<b>412</b>
Diff Two Text Files.....	415
Compare Two Reports.....	416
Cross-Probing between Reports.....	417
Using the Floating Ruler.....	419
<b>Chapter 47: Platform and System Diagrams.....</b>	<b>420</b>
<b>Chapter 48: AI Engine Graphs and Arrays.....</b>	<b>423</b>
<b>Chapter 49: Link Summary: Multiple Strategies and Timing Reports.....</b>	<b>425</b>
<b>Chapter 50: Setting Guidance Thresholds.....</b>	<b>427</b>
<b>Chapter 51: Creating an Archive File.....</b>	<b>430</b>
<b>Section IX: Using the Vitis IDE.....</b>	<b>432</b>
<b>Chapter 52: Launching Vitis IDE.....</b>	<b>433</b>
<b>Chapter 53: Creating a Vitis IDE Project.....</b>	<b>434</b>
Launch a Vitis IDE Workspace.....	434
Create an Application Project.....	435
Understanding the Vitis IDE.....	442
Adding Sources.....	443
Working in the Project Editor View.....	446
Working in the Assistant View.....	448
<b>Chapter 54: Building the System.....</b>	<b>451</b>
Vitis IDE Guidance View.....	453
Working with Vivado Tools from the Vitis IDE.....	454
<b>Chapter 55: Vitis IDE Debug Flow.....</b>	<b>456</b>
Using the Standalone Debug Flow .....	458
vitis -debug Command Line.....	460

<b>Chapter 56: Configuring the Vitis IDE.....</b>	<b>464</b>
Vitis Project Settings.....	464
Vitis Build Configuration Settings.....	466
Vitis Hardware Function Settings.....	467
Vitis Binary Container Settings.....	469
Vitis Toolchain Settings.....	471
Vitis Run and Debug Configuration Settings.....	476
<b>Chapter 57: Project Export and Import.....</b>	<b>480</b>
Export a Vitis Project.....	480
Import a Vitis Project.....	481
Import Projects from Git.....	482
<b>Chapter 58: Getting Started with Examples.....</b>	<b>485</b>
Installing Examples and Libraries.....	485
Using Local Copies.....	487
<b>Chapter 59: RTL Kernel Wizard.....</b>	<b>489</b>
Launch the RTL Kernel Wizard.....	489
Using the RTL Kernel Wizard.....	490
Using the RTL Kernel Project in Vivado IDE.....	498
<b>Section X: Using Vitis Embedded Platforms.....</b>	<b>505</b>
<b>Chapter 60: Vitis Embedded Platforms.....</b>	<b>506</b>
Introduction.....	506
Platform Types.....	506
Platform Naming Convention.....	512
Embedded Platform Components and Architecture.....	512
Installing Embedded Platforms.....	514
<b>Chapter 61: Using Vitis Embedded Platforms.....</b>	<b>515</b>
Packaging Images.....	515
Writing Images to the SD Card.....	517
Configuring the PL Kernel in DFX Platforms and Non-DFX Platforms.....	518
Running an Acceleration Application on the Board.....	519
Software Package Management in PetaLinux rootfs.....	519

<b>Chapter 62: Creating Embedded Platforms in Vitis.....</b>	<b>522</b>
Platform Creation Basics.....	522
Platform Creation Requirements.....	523
Creating an Embedded Platform.....	524
Validating an Embedded Platform.....	550
<b>Section XI: Additional Information.....</b>	<b>555</b>
<b>Chapter 63: Methodology for Accelerating Data Center Applications with the Vitis Software Platform.....</b>	<b>556</b>
Introduction.....	556
Methodology for Architecting a Device Accelerated Application.....	559
Methodology for Developing C/C++ Kernels.....	572
Best Practices for Data Center Acceleration with Vitis.....	585
<b>Chapter 64: OpenCL Programming.....</b>	<b>587</b>
OpenCL Host Application.....	587
<b>Chapter 65: Streaming Data Transfers.....</b>	<b>617</b>
Streaming Kernel Coding Guidelines.....	617
Free-Running Kernels.....	618
<b>Chapter 66: Migrating to a New Target Platform.....</b>	<b>620</b>
Design Migration.....	620
Migrating Releases.....	625
Modifying Kernel Placement.....	626
Address Timing.....	632
<b>Chapter 67: Output Structure of the Vitis Tools.....</b>	<b>635</b>
Output Directories of the v++ Command.....	635
Output Directories from the Vitis IDE.....	641
<b>Chapter 68: Using Vitis System Compilation Mode.....</b>	<b>652</b>
Introduction to Vitis System Compilation Mode.....	653
Quick Start Example.....	657
Creating a VSC Makefile.....	662
Building Hardware.....	665
Application Software Interface.....	684

Debugging and Validation.....	705
Supported Platforms and Startup Examples.....	713
<b>Section XII: Additional Resources and Legal Notices.....</b>	<b>716</b>
Xilinx Resources.....	716
Documentation Navigator and Design Hubs.....	716
Revision History.....	717
Please Read: Important Legal Notices.....	721

# Getting Started with Vitis

This section contains the following chapters:

- [Navigating Content by Design Process](#)
- [Vitis Software Platform Release Notes](#)
- [Installation](#)

# Navigating Content by Design Process

Xilinx® documentation is organized around a set of standard design processes to help you find relevant content for your current development task. All Versal® ACAP design process [Design Hubs](#) and the [Design Flow Assistant](#) materials can be found on the [Xilinx.com](#) website. 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.
- **Embedded Software Development:** Creating the software platform from the hardware platform and developing the application code using the embedded CPU. Also covers XRT and Graph APIs.
- **Host Software Development:** Developing the application code, accelerator development, including library, XRT, and Graph API use.
- **AI Engine Development:** Creating the AI Engine graph and kernels, library use, simulation debugging and profiling, and algorithm development. Also includes the integration of the PL and AI Engine kernels.
- **Hardware, IP, and Platform Development:** Creating the PL IP blocks for the hardware platform, creating PL kernels, functional simulation, and evaluating the 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 Release Notes

This section contains information regarding the features and updates of the Vitis™ software platform in this release. It also contains information regarding the features and updates of the Vitis software platform for Versal® AI Engine development.

---

## What's New

For information about what's new in this version of the Vitis™ unified software development platform, see the [Vitis What's New Page](#).

---

## Supported Platforms

### Data Center Accelerator Cards

Access the latest Vitis target platforms for Alveo™ Data Center accelerator cards at [www.xilinx.com/products/boards-and-kits/alveo.html](http://www.xilinx.com/products/boards-and-kits/alveo.html).

Refer to *Alveo Data Center Accelerator Card Platforms User Guide* ([UG1120](#)) for specifications of each accelerator card and available target platforms. The *Getting Started* section for each accelerator card has information for deploying your applications on that card.

Refer to [Installing Xilinx Runtime and Platforms](#) for more information on setting up XRT and platforms.

### Embedded Platforms

Pre-built embedded base platforms are installed with the Vitis installer. The following platforms are included:

- `xilinx_vck190_base_202220_1`
- `xilinx_vck190_base_dfx_202220_1`

- xilinx\_vmk180\_base\_202220\_1
- xilinx\_zcu102\_base\_202220\_1
- xilinx\_zcu102\_base\_dfx\_202220\_1
- xilinx\_zcu104\_base\_202220\_1

Common software images can be downloaded from the [Embedded Platforms download page](#). They provide pre-built Linux kernel, `rootfs`, `sysroot`, and boot components and can also work with the pre-built embedded base platforms mentioned above. They can work with the pre-built embedded base platforms mentioned above.

**Note:** The ZC706 base platform is discontinued from 2022.2. The Vitis embedded acceleration flow will only support device families later than Zynq® UltraScale+™ MPSoC.

### Versal Platform for AI Engine Development

The VCK190 platform is available for use with the Vitis application acceleration development flow, as described in *AI Engine Tools and Flows User Guide* ([UG1076](#)). The platform enables development of designs that include:

- AI Engine graphs and kernels
- Programmable logic kernels
- Host application targeting the Linux or a bare metal OS running on the Arm® processor in the Versal device.

---

## Embedded GNU Toolchain Details

The following GNU toolchain components are installed with the Vitis software platform:

- **binutils:** 2.37
- **gcc:** 11.2.0
- **gdb:** 10.2
- **glibc:** 2.34
- **newlib:** 4.1.0

# Changed Behavior

The following table specifies differences between this release and prior releases that impact behavior or flow when migrating.

**Table 1: Changed Behavior Summary**

Area	Behavior
Vitis embedded platform	Vitis base embedded platform source code removed the flow for compiling PetaLinux project from scratch and now only uses pre-built common images. This source code change doesn't bring behavioral changes in the pre-built base platforms.
Vitis HLS	Vitis HLS automatically tries to select the correct pipeline style to use for a given pipelined loop of function. If the pipeline control requires high fanout (and meets other FRP requirements), Vitis HLS selects the free-running pipeline (FRP) style to limit the high fanout. If the pipeline is used with hls::tasks, the flushing pipeline (FLP) style is automatically selected to avoid deadlocks. Finally, if neither of the above cases apply, then the standard pipeline (STP) style is selected.
Vitis IDE	Help message UG643 now directs to the online version by default.
	Formalized the QEMU support platform to Linux only.
	The BSP and Libraries Help documentation which was default available as part of the Vitis install is now available at the <a href="#">Xilinx Documentation Portal</a> . You need one time internet connectivity to access Help documentation from within Vitis IDE. For more information, navigate to Vitis IDE > Help > OS and Libraries Help > BSP and Libraries Document Collection (UG643).
Vitis Emulation	Vitis software emulation for embedded acceleration can use x86 processes to simulate PS processors. This features requires XRT on the host machine. The <code>Vitis_Accel_Examples</code> makefile flow uses x86 for software emulation. The Vitis IDE will keep using QEMU in this release. It is recommended to install XRT on the host machine for all acceleration development workflows.

# Known Issues

Known issues for the Vitis software platform are available on the [Vitis 2022 Known Issues](#) web page.

# Installation

---

## Installation Requirements

The Vitis™ software platform consists of an integrated design environment (IDE) for interactive project development, and command-line tools for scripted or manual application development. The Vitis software platform also includes the Vivado® Design Suite for implementing the kernel on the target device, and for developing custom hardware platforms.

Some requirements listed here are only *required* for software acceleration features, but not for embedded software development features. Xilinx recommends installing all the required packages to have the best experience with the Vitis software platform.

To install and run on a computer, your system must meet the following minimum requirements.

**Note:** The application acceleration development flow is not supported by the Windows OS.

**Note:** This release, 2022.2, is the last release that supports the following operating systems:

- RHEL 8.2
- Ubuntu 18.04.4
- Ubuntu 20.04
- Ubuntu 20.04.1

The latest versions of the related major release (for example, RHEL 8, Ubuntu 18.04 LTS, and Ubuntu 20.04 LTS) will still be supported.

**Table 2: Application Acceleration Development Flow Minimum System Requirements**

Component	Requirement	
	Development (Build Machine OS)	Deployment (Host OS) Enabled via XRT
Operating system	Linux, 64-bit: <ul style="list-style-type: none"> <li>Red Hat Enterprise Workstation/Server 7/CentOS: 7.9</li> <li>Red Hat Enterprise Workstation/Server: 8.2, 8.3, 8.4, 8.5, 8.6</li> <li>Ubuntu 18.04.4 LTS, 18.04.5 LTS, 18.04.6 LTS, 20.04 LTS, 20.04.1 LTS, 20.04.2 LTS, 20.04.3 LTS, 20.04.4 LTS and 22.04 LTS</li> <li>Amazon Linux 2 AL2 LTS</li> </ul>	For on-premise acceleration (Alveo Data Center accelerator cards): <ul style="list-style-type: none"> <li>Red Hat Enterprise Workstation/Server 7/CentOS: 7.9</li> <li>Red Hat Enterprise Workstation/Server: 8.2, 8.3, 8.4, 8.5, 8.6</li> <li>Ubuntu 18.04.4 LTS, 18.04.5 LTS, 18.04.6 LTS, 20.04 LTS, 20.04.1 LTS, 20.04.2 LTS, 20.04.3 LTS, 20.04.4 LTS and 22.04 LTS</li> <li>Amazon Linux 2 AL2 LTS</li> <li>Linux in VM on Azure Windows Server 2019</li> </ul> For edge acceleration (embedded platforms): <ul style="list-style-type: none"> <li>PetaLinux 2022.2</li> </ul>
System memory	For Alveo cards: 64 GB (80 GB is recommended) For embedded: 32 GB	
Internet connection	Required for downloading drivers and utilities.	
Hard disk space	150 GB	

**Table 3: AI Engine Development Flow Minimum System Requirements**

Component	Requirement
Operating system	Linux, 64-bit: <ul style="list-style-type: none"> <li>Red Hat Enterprise Workstation/Server 7/CentOS: 7.9</li> <li>Red Hat Enterprise Workstation/Server: 8.2, 8.3, 8.4, 8.5, 8.6</li> <li>Ubuntu 18.04.4 LTS, 18.04.5 LTS, 18.04.6 LTS</li> <li>Ubuntu 20.04 LTS, 20.04.1 LTS, 20.04.2 LTS, 20.04.3 LTS, 20.04.4 LTS</li> <li>Ubuntu 22.04 LTS</li> </ul>
System memory	64 GB (80 GB is recommended)
Internet connection	Required for downloading drivers and utilities.
Hard disk space	100 GB

## OpenCL Installable Client Driver Loader

The 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.

Refer to [OpenCL Host Application](#) for information about the OpenCL host application and how to install the ICD Loader.

---

# Vitis Software Platform Installation

## Installing the Vitis Software Platform

Ensure your system meets all requirements described in [Installation Requirements](#).



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

1. Go to the [Xilinx Downloads Website](#).
2. Download the installer for your operating system.
3. Run the installer, xsetup, or xsetup.exe, which opens the Welcome page. Extract the installer package.
4. Click **Next** to open the Select Install Type page of the Installer.
5. If installing with the web installer, enter your Xilinx user account credentials, and then select **Download and Install Now** (only needed by web installer).
6. Click **Next** to open the Accept License Agreements page of the Installer.
7. Accept the terms and conditions by clicking each **I Agree** check box.
8. Click **Next** to open the Select Product to Install page of the Installer.
9. Select **Vitis** and click **Next** to open the Vitis Unified Software Platform page of the Installer.
10. Customize your installation by selecting design tools and devices (optional).

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

You can enable **Vitis IP Cache** to install cache files for example designs found in the release. This is not required, but when selected the files will be installed at <installdir>/Vitis/<release>/data/cache/xilinx.

The default **Devices** selections are for devices used on standard acceleration platforms supported by the Vitis tools. You can disable some devices that may not be of interest in your installation.



**IMPORTANT!** Do not deselect the following option. It is required for installation.

- Devices → Install devices for Alveo and Xilinx Edge acceleration platforms

11. Click **Next** to open the Accept License Agreements page of the Installer and accept as appropriate.
12. Click **Next** to open the Select Destination Directory page of the Installer
13. Specify the installation directory, review the location summary, review the disk space required to insure there is enough space, and click **Next** to open the Installation Summary page of the Installer.
14. Click **Install** to begin the installation of the software.

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

1. Locate the script at: `<install_dir>/Vitis/<release>/scripts/installLibs.sh`, where `<install_dir>` is the location of your installation, and `<release>` is the installation version.
2. Run the script using `sudo` privileges as follows:

```
sudo installLibs.sh
```

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



**IMPORTANT!** Pay attention to any messages returned by the script. You might need to install any missing packages manually.

## Installing Xilinx Runtime and Platforms

Xilinx Runtime (XRT) is implemented as a combination of user-space and kernel driver components. XRT supports Alveo™ PCIe®-based cards, as well as Versal® and Zynq® UltraScale+™ MPSoC-based embedded system platforms, and provides a software interface to Xilinx programmable logic devices.

You must install XRT for use in the Vitis application acceleration development flow. You do not need to reinstall it for every additional platform you choose to download.

**Note:** Installing XRT is *not* required when targeting Arm®-based embedded platforms: Vitis compiler has its own copy of `xclbinutil` for hardware generation, and for software compilation, you can use the XRT from the `sysroot`. Look for **Common images for Embedded Vitis platforms** on the downloads page.



**IMPORTANT!** XRT installation uses standard Linux RPM and Linux DEB distribution files, and root access is required for all software and firmware installations.

`<rpm-dir>` or `<deb-dir>` is the directory where you downloaded the packages to install.

To download and install the XRT package for your operating system, do the following.

1. Go to <https://www.xilinx.com/xrt>.

2. From the [Getting Started](#) page, you can choose to download the XRT package for a specific Alveo Data Center accelerator card, or for Embedded Platforms. After choosing the platform, you will be redirected to a website with instructions for downloading XRT and the required files for the selected platform.
3. Follow the directions to install XRT and your selected platform.



**TIP:** *The instructions for installing the Alveo Data Center accelerator cards are provided on the platform download page. Instructions for installing the Embedded Platforms can be found in the following section.*

## Installing Embedded Platforms

To support the Vitis application acceleration development flow, embedded platforms require a Linux kernel, a `rootfs` with integrated Xilinx Runtime, and a `sysroot` to cross-compile the host application.

To install a platform, download the zip file and extract it into `/opt/xilinx/platforms`, or extract it into a separate location and add that location to the `PLATFORM_REPO_PATHS` environment variable.

You can build your own platform software, or use a pre-built software common image. To download a pre-built common image, look for the **Common images for Embedded Vitis platforms** block on the downloads page, and download and extract the common image for your platform architecture.

Running `sdk.sh` extracts and installs the `sysroot`. The option `-d` gives you the option to choose where to install the `sysroot`. This package also provides a pre-compiled kernel image and `rootfs`.

You can add the `sysroot` to a Makefile for your command line project, or the Vitis IDE will prompt you to add it to your application project. For example, in your Makefile point `<SYSROOT>` to `/<install_path>/cortexa72-cortexa53-xilinx-linux`, which is generated when running `sdk.sh`.

## Setting Up the Environment to Run the Vitis Software Platform

To configure the environment to run the Vitis software platform, run the following scripts in a command shell to set up the tools to run in that shell:

```
#set up XILINX_VITIS and XILINX_VIVADO variables
source <Vitis_install_path>/Vitis/2022.2/settings64.sh
#set up XILINX_XRT for data center platforms (not required for embedded
platforms)
source /opt/xilinx/xrt/setup.sh
```



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

This sets up the tools for the Vitis application acceleration development flow, the Vitis embedded software development flow, and the AI Engine tools for development on Versal AI Engine devices.

To use any platforms you have downloaded as described in [Installing Xilinx Runtime and Platforms](#), 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.

# Introduction to Vitis Flows

The Vitis™ unified software platform is a development environment for heterogeneous applications supporting Xilinx® devices such as Alveo™ Data Center Accelerator cards, Versal ACAP devices, Kria™ SOM, and Zynq® MPSoC. In the Vitis environment, heterogeneous systems include software applications running on x86 host processors or Arm® embedded processors, compute kernels running in programmable-logic (PL) regions or Versal AI Engine arrays, and extensible platform designs 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 Xilinx adaptive computing. Vitis accelerated libraries are available for common functions of math, statistics, linear algebra and DSP, as well as for domain specific applications, like vision and image processing, quantitative finance, database, data analytics, and data compression. For more information on Vitis accelerated libraries, refer to [https://xilinx.github.io/Vitis\\_Libraries/](https://xilinx.github.io/Vitis_Libraries/).

The Vitis™ unified software platform combines all aspects of Xilinx® 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](#), [Embedded System design](#), and traditional embedded hardware and software design. Each of these flows is described in more detail in the following chapters:

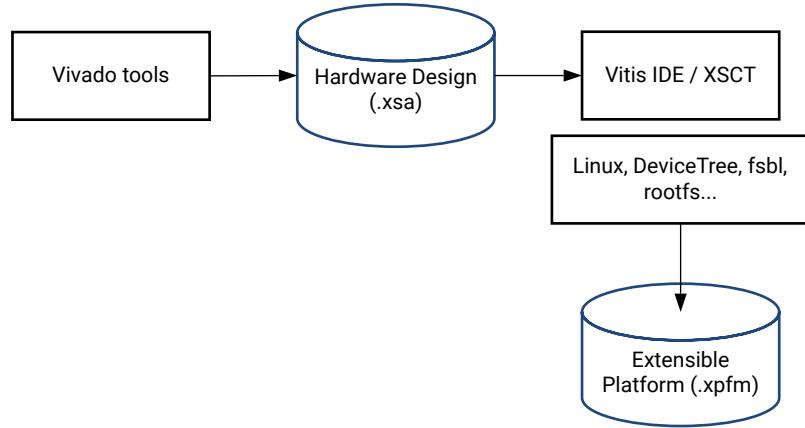
- [Introduction to Vitis Tools for Embedded System Designers](#)
- [Introduction to Data Center Acceleration for Software Programmers](#)
- [Introduction to Data Center Acceleration for RTL Designers](#)
- [Tutorials and Examples](#)

# Introduction to Vitis Tools for Embedded System Designers

This chapter is intended for Embedded System Designers who are developing systems using heterogeneous compute elements supported by Vitis tools. The chapter introduces key concepts for understanding and using Vitis tools for embedded system design. The elements of the embedded system can include Vivado exported hardware designs, Vitis extensible platforms, Arm processor applications, PL kernels, and AI Engine graph applications. Refer to [Terminology for Embedded System Design](#) for definitions of these terms.

As shown in the figure below, Vitis tools support two different embedded design flows.

*Figure 1: Vitis Embedded System Design Flows*



X27309-101922

The traditional embedded software development flow relies on a fixed hardware design, processor domains and OS, boot files, software drivers, and Arm processor based software applications. This traditional design flow is described briefly in [Fixed Platforms versus Extensible Platforms](#), and is documented with more detail in the *Vitis Embedded Software Development User Guide (UG1400)*. Refer to that document for more information on the traditional embedded software flow.

The Vitis heterogeneous system design flow enables designing and building embedded system designs using Versal ACAP devices, Kria™ SOM, and Zynq® MPSoC devices. Notice that the Embedded Software flow is part of the Vitis Heterogeneous System Design flow. The heterogeneous system design flow is the primary focus of this *Introduction*, highlighting the elements from four different disciplines included in the flow:

1. Hardware design and custom platform development
2. Embedded processor software design
3. Programmable-logic (PL) kernel design
4. Versal AI Engine graph design

The tools and techniques for creating and integrating these different components are the focus of the following sections, beginning with some terminology for a common understanding.

---

## Terminology for Embedded System Design

The following introduces some of the tools and terms you will encounter in this document.

- **Vitis core development kit:** Provides a framework for designing, building, and debugging heterogeneous applications using standard programming languages for both software and hardware components.
- **Vivado Design Suite:** An RTL language design, synthesis, and implementation tool that enables hardware designers to create and export hardware designs (.xsa).
- **Xilinx Support Archive (.xsa):** Is a hardware container exported from the Vivado Design Suite for multiple uses, including in a fixed or extensible platform.
- **Fixed Platform (.xfm):** Includes a completed hardware design (.xsa) and supporting software files defining the operating system, libraries, and boot files. In this context, "fixed" simply means that the hardware design is complete.
- **Extensible Platform (.xfm):** The target platform of the Vitis heterogeneous system design flow. In this context, the "extensible" design can be further customized by adding programmable content such as PL kernels and AI Engine graph applications to the platform to build the embedded system. Extensible Platform can also be used to develop software like the fixed platform.
- **PL kernel (.xo):** A hardware function that can be added to the PL region of an extensible platform to define custom hardware. PL kernels can be defined using C++ code in Vitis HLS, or using RTL code and the IP packager feature of the Vivado Design Suite.
- **Vitis HLS:** A high-level synthesis tool that translates C/C++ functions into RTL for implementation in the programmable logic (PL) region of a device. Vitis HLS generates a compiled object (.xo) file that can be imported into the Vitis environment.

- **Vitis Compiler:** The `v++` command used to compile PL kernels (`.xo`) from C++ code, and to link multiple PL kernels with hardware platforms and AI Engine graph applications to build the device binary.
- **PS Application:** A user-defined software application to be run on an Arm processor in a Xilinx MPSoc or ACAP device, that can control and interact with PL kernels and AI Engine graph.
- **Xilinx runtime library (XRT):** Provides an API and drivers to let your software application control, transfer data to, and read the status of the PL kernels and AI Engine graph application in the hardware design.
- **AI Engine kernel and graph applications:** Compiled by the Vitis `aiecompiler` and linked into the embedded system with `v++`. Kernels are functions that run on Versal AI Engines and form the fundamental building blocks of a data flow graph application. The AI Engine graph application is an adaptive dataflow graph with deterministic behavior.
- **aiecompiler/aiesimulator:** Vitis tools for the compilation and simulation of AI Engine graph applications.
- **Device Binary (`.xc1bin`) file:** Contains the programmable device image (PDI) for Versal ACAP or the bitstream for Zynq MPSoC, and metadata needed to control the hardware design.

---

## Fixed Platforms versus Extensible Platforms

### Traditional Fixed-Platform Design

A fixed platform design is a hardware design completed in the Vivado Design Suite. You can use the Vivado IP integrator feature to stitch together IP in a block design, or you can describe the hardware system in RTL. You can also use Vitis HLS to create hardware functions in C++ and generate Vivado IP to add to your hardware design.

You will use the `write_hw_platform -fixed` command to create a Xilinx Shell Archive (`.xsa`) of the completed hardware design; and use either the Vitis IDE, or the Xilinx Software Command-line Tool (XSCT) to define a platform project, import the fixed `.xsa`, and configure and define processor domains, or BSPs for the fixed-platform. The fixed platform (`.xpfm`) can be used in a traditional embedded software design flow to create embedded applications for Versal ACAP devices, or Zynq MPSoC devices.

The development of software applications for embedded processors requires the use of hardware drivers delivered as part of the exported hardware container (`.xsa`). You must manage and use the drivers to access your hardware design from your software application. This traditional embedded software design flow is fully documented in the *Vitis Embedded Software Development User Guide (UG1400)*.

## Extensible Platform-Based Design

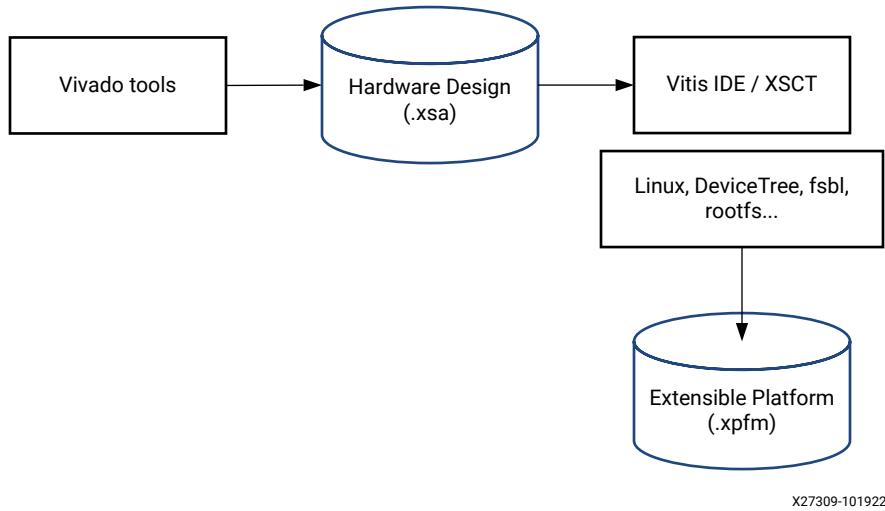
The extensible platform also starts with the Vivado Design Suite to define an extensible hardware design (.xsa) that allows the addition of programmable components such as PS and AI Engine features to quickly vary the resulting embedded system design. The extensible .xsa is a container that provides the foundation of the hardware design for the v++ linker to extend by updating a dynamic region of the design. The extensible platform allows the embedded system designer to easily extend the functionality of the extensible hardware foundation.

For embedded system designs, a parallel development process can be used where different elements of the heterogeneous system can be developed concurrently. The application team starts work with a Xilinx base platform, such as the VCK190 or ZCU104, to develop the programmable components of the system. Using an available platform means that the application can be developed and tested independently using a known-good system. At the same time the platform team works to build and validate the custom platform. You are strongly encouraged to use an existing platform design as a reference for your own custom platform designs. Current Xilinx embedded platforms can be found in the Vitis installation, and their source code can be found in the [Vitis Embedded Platform Source](#) repository on GitHub.

The hardware part of the extensible platform is a Vivado project containing any required IP such as an interrupt controller and clock wizard, as well as any additional features you want to implement into the hardware design. Extensible platform designs have specific requirements as described in [Creating Embedded Platforms in Vitis](#), such as AXI-4 interfaces for linking PL kernels into the system, memory controllers, one or more clock and reset signals, and a single interrupt. More information on required interfaces can be found at [Adding Hardware Interfaces](#).



**TIP:** As described in Versal ACAP Design Guide ([UG1273](#)), you must use the IP integrator to configure and connect required IP such as the CIPS IP, the NoC/DDRMC IP, and hardware debug IP to take advantage of block design automation when building and updating your hardware design. However, you can use the IP integrator feature to configure and connect these critical Versal ACAP IP and then instantiate the resulting hierarchical module into a higher-level RTL design.

**Figure 2: Fixed or Extensible Platform**

The hardware design file (`.xsa`) is exported from the Vivado project using the `write_hw_platform` command and imported into a Platform project in the Vitis IDE, or XSCT. In the Platform project the software components (processor domains, DeviceTree, OS) are packaged with the hardware (`.xsa`) to create a custom extensible embedded processor platform (`.xpfm`):

- Operating System includes standalone (or baremetal), FreeRTOS, or Linux. For systems running XRT, Linux is required.
- Processor specifies the Arm core to use for the specified OS domain. Note that the choice of OS determines which processors are available to configure.



**TIP:** *The initial Platform project setup lets you configure one processor domain, but the IDE lets you configure additional domains once the project has been established. Advanced settings of the platform, such as for DFX platforms, must be accessed through XSCT.*

When the platform has been built, an `export` folder is added to the hierarchy of the project, and the platform `.xpfm` file and `./hw` and `./sw` folders are added to the platform. The new platform is added to the `$PLATFORM_REPO_PATHS` environment variable, and the platform can be used for application development with the Vitis tools.

The platform-based design flow uses the Vitis tool for the definition of OS, domains, and software applications in the embedded system, but also uses the Vitis compiler (`v++`) to link PL kernels and AI Engine graph applications with the extensible hardware platform. This flow promotes concurrent development of the different elements of the system and facilitates the integration process of heterogeneous systems.



**IMPORTANT!** *The platform-based embedded system design flow is a more complex but more adaptable flow. It is the required flow for using AI Engine graph applications on Versal AI Core devices. This flow also requires XRT running on a Linux OS to control the components of the hardware design, and simplify writing software applications to work with the embedded hardware design.*

The extensible platform is adapted by linking PL kernels (.xo) and AI Engine graph applications (libadff.a) to the hardware design using the `v++ --link` command. However, in designing the extensible platform, you can decide how much or how little to include in the hardware design (.xsa) of the platform. As described in [Generate XSA file](#), an extensible platform has minimum content requirements such as interrupt controller and clock wizard. You can choose to design a platform with the minimum content required and add functionality to your system through PL kernels, or you can add features directly to the platform hardware design while relying less on PL kernels.

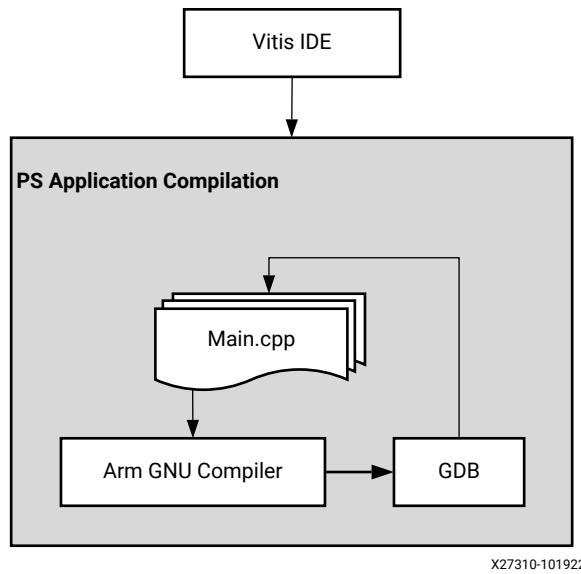
A minimal platform is more flexible, more easily defined, and quicker to iterate and implement new functionality. However, implementing functionality directly in the platform has the advantage of integrating existing RTL code rather than packaging the IP for use as PL kernel.

---

## PS Software Development

The PS software development flow is classic software development, with compilation on an Arm-based `g++` cross-compiler, and debugging using GDB. This step can be done in an Application project in the Vitis IDE; or it can be done from the command line or from a `Makefile` using `g++` and `gdb` commands.

*Figure 3: Embedded Software Development*



X27310-101922

As described in [Compiling and Linking for Arm](#), you will compile the software program to run on a Cortex®-A72 or Cortex-A53 core processor using the GNU Arm cross-compiler to create an ELF file.

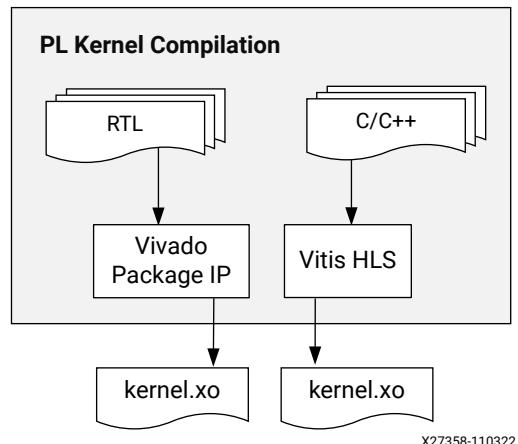
In platform-based design, the software program interacts with kernels in the PL and AI Engine regions of the device through the XRT API. For more information on writing the software program using the API, refer to [Section III: Developing Applications](#). For more information on compiling the software program, refer to [Building the Software Application](#).

## Vitis PL Kernel Development Flow

In the platform-based design flow, the platform provides an expandable foundation for hardware development. The hardware design can be expanded through the use of PL kernels linked into the platform using the Vitis tools. PL kernels are hardware functions that can be generated from C++ in the Vitis HLS tool, or described directly in RTL in the Vivado Design Suite.

This section provides a brief overview of the Vitis development flow for PL kernels. A representation of this flow is shown below.

Figure 4: Vitis PL Kernel Development Flow



PL kernels can be written in C++ and synthesized into RTL IP using the Vitis HLS tool. As described in *Vitis High-Level Synthesis User Guide (UG1399)*, there are fundamental concepts that need to be understood in order to design and write good synthesizable software in such a way that it can be successfully converted to hardware using high-level synthesis (HLS) tools.

PL kernels written in C++ can also be compiled directly into Xilinx object files (.xo) using the `v++ --compile` command as described in [Compiling C/C++ PL Kernels](#). However, the recommended method for C/C++ kernels is to use a bottom-up methodology in Vitis HLS to let you perform analysis and optimization directly on the kernel design to obtain the best results. In fact, the `v++ --compile` command uses Vitis HLS as a key part of the compilation process.

PL kernels written in C++ can also be written in RTL, or may already exist in catalogs of custom IP for use with the Vivado tools. As described in [Packaging RTL Kernels](#), these kernels have specific interface requirements to allow them to be linked into the system design. Embedded system designers can also decide to integrate the RTL as part of their custom platform, or link the RTL kernel into the system using the `v++ --compile` command as explained below.

Whether coming from Vitis HLS, RTL kernel, or `v++ --compile` command, the results of kernel development or compilation is a Xilinx object (`.xo`) file. The PL kernels have specific requirements regarding the different types of supported interfaces, and clocks and reset signals as explained in [PL Kernel Properties](#). The clocks and resets let XRT manage the execution of the kernels at run time, and the interfaces let the PL kernels be linked with an extensible hardware platform (`.xpfm`) as well as with other `.xo` files, and AI Engine graph applications (`.libadbf.a`) as described in [Building and Packaging the System](#).

---

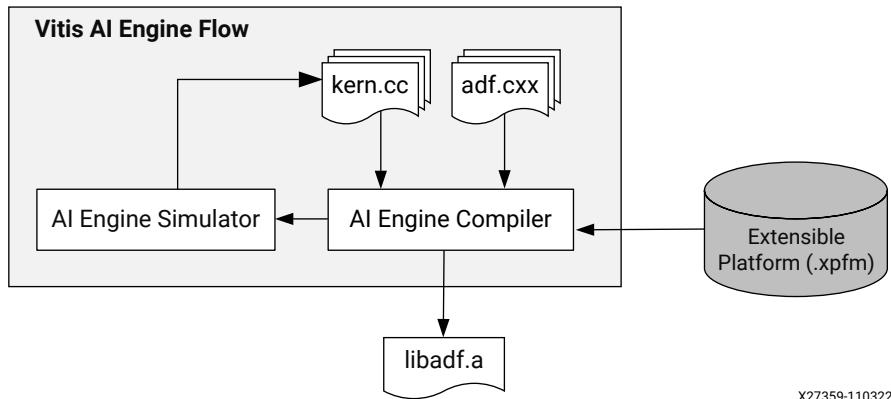
## Versal AI Engine Graph Development

Versal adaptive compute acceleration platforms (ACAPs) combine Scalar Engines, Adaptable Engines, and Intelligent Engines with leading-edge memory and interfacing technologies to deliver powerful heterogeneous computing for any application. Most importantly, Versal ACAP hardware and software are targeted for programming and optimization by data scientists as well as software and hardware developers.

Some Versal ACAP devices incorporate an array of very-long instruction word (VLIW) processors with single instruction multiple data (SIMD) vector units called, AI Engines, that are highly optimized for compute-intensive applications such as 5G wireless and artificial intelligence (AI) applications. When implementing an algorithm for a Versal AI Core or Versal AI Edge, it is important to understand what the AI Engine does well and what would be better implemented in the other engines, for example, the Scalar and Adaptable engines.



**IMPORTANT!** The inclusion of AI Engine graph applications require the use of extensible platforms in your embedded system design to control the execution of the application.

**Figure 5: Vitis AI Engine Development Flow**

Embedded system designs using Versal AI Core devices can include AI Engine graph applications developed and tested using Vitis tools. The figure above shows the development flow for AI Engine graph applications in which individual kernel code is compiled and combined into the graph application.

As described in *AI Engine Kernel and Graph Programming Guide* ([UG1079](#)), an AI Engine kernel is a C/C++ program written using specialized intrinsic calls that target the VLIW SIMD vector processor. The AI Engine kernel code is compiled using the AI Engine compiler (`aiecompiler`) to compile the kernels to produce an ELF file that is run on the AI Engine processors.

An AI Engine program requires a data flow graph specification which is written in C++. This specification can be compiled and executed using the AI Engine compiler. An adaptive data flow (ADF) graph application consists of nodes and edges where nodes represent compute kernel functions, and edges represent data connections. The ADF graph is a static dataflow graph with the AI Engine kernels operating in parallel. Kernels operate on data streams, and are the fundamental building blocks of an ADF graph specification. These kernels consume input blocks of data and produce output blocks of data. Refer to *AI Engine Kernel and Graph Programming Guide* ([UG1079](#)) for more information.

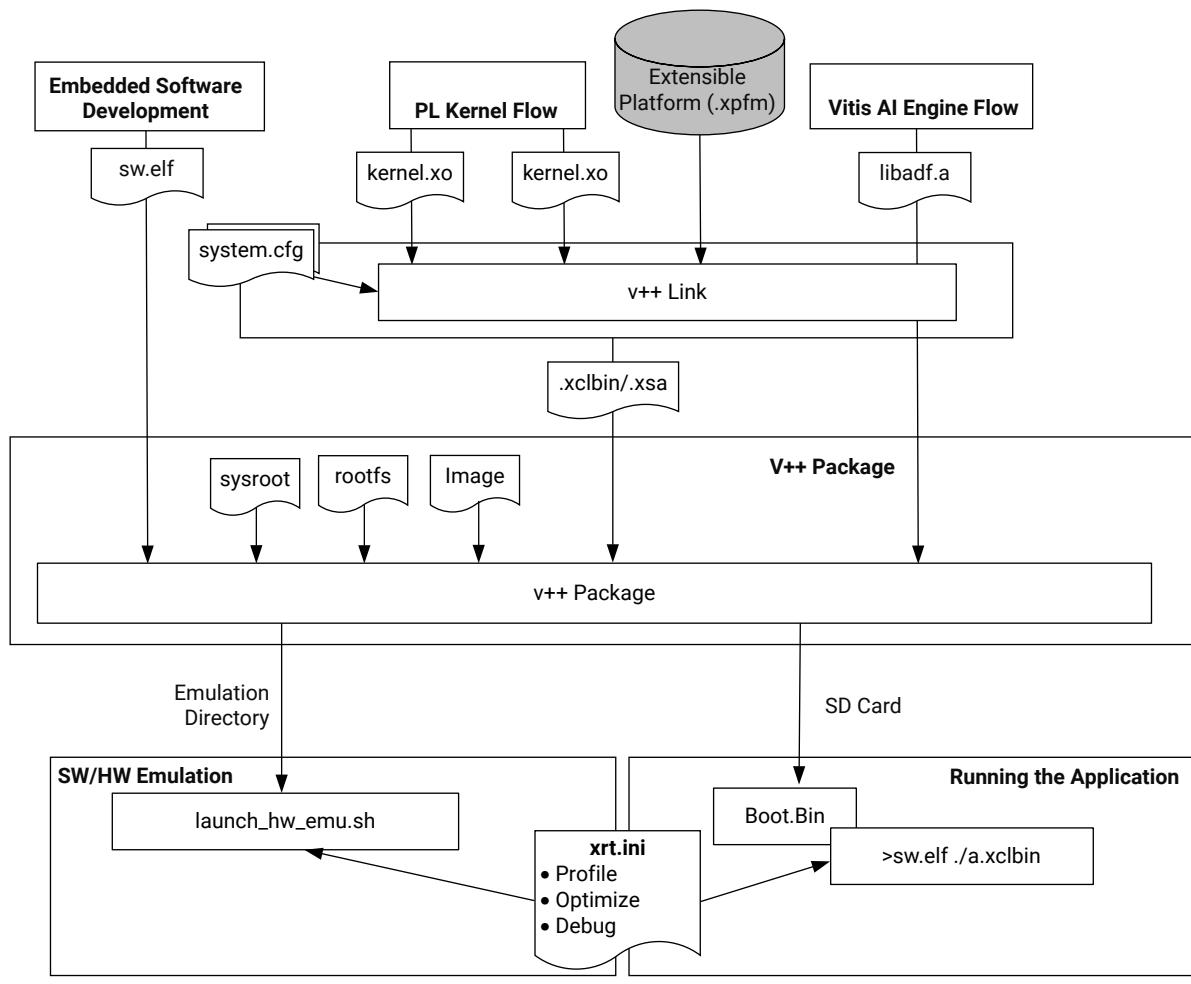
As discussed in [Using the Vitis IDE](#) in UG1076, an AI Engine project can be built in the Vitis IDE as an Application project targeting the `aiengine` domain of a Versal AI Core device. The project can also be managed from the command line. The kernels and ADF graph are written as C++ code and compiled using the `aiecompiler` command as described in [Compiling an AI Engine Graph Application](#), and simulated using the `x86simulator` or `aiesimulator` as described in [Simulating an AI Engine Graph Application](#).

By default, the AI Engine compiler writes all outputs to a directory called `./Work` and creates a file called `libadaf.a`, where `Work` is a sub-directory of the current directory where the tool was launched, and `libadaf.a` is written to the same directory as the AI Engine compiler was launched from and is used for linking with PL kernels and the extensible platform using the `v++` command as explained in the next section.

# Building and Packaging the System

With the elements of the heterogeneous embedded system design available, you are ready to build the system using the Vitis tools. The `v++ --link` command is a key part of the system build process, as is the `v++ --package` command. Both of these commands are shown in the figure below, which highlights the different elements of the system, and how they fit together. At the top of the figure are the extensible platform (`.xpfm`), the embedded software program (`.elf`), the PL kernels (`.xo`), and the AI Engine graph (`libadf.a`) that are elements of the heterogeneous system. Below those are the linking and packaging commands that build the system.

Figure 6: Building and Packaging the Embedded System Design



X27360-110422

## System Linking

After the various elements of the embedded system design become available, at least in some initial state, the system can be built using the `v++ --link` command. The Vitis compiler links the kernel `.xo` files with the hardware platform and `libadff.a` to create fixed platform (`.xsa`) for Versal ACAP devices, or a device executable (`.xclbin`) for Kria™ SOM, and Zynq MPSoC devices.

As described in [Linking the Kernels](#), the process of linking multiple kernels (`.xo`), `libadff.a`, and the extensible platform (`.xpfm`) starts with a description of the architecture of the system using a configuration file to define the number of kernel instances, or CUs, the memory layout of kernels, and connection of streaming interfaces.

Generation of the device binary or fixed hardware design launches the Vivado synthesis, place, and route tools. The process is broken down into a series of major steps, that can be interrupted to enable customization as explained in [Using -to\\_step and Launching Vivado Interactively](#), or allows insertion of Tcl scripts as described in [Using the -vivado and -advanced Options](#) to help drive the tools to pass timing and deliver the desired results.

## Packaging the System

The `v++ --package` command packages the final product at the end of system linking, and configures the boot system for the device. As stated in [Packaging the System](#), the `v++ --package` command will also generate the device binary (`.xclbin`) for Versal platform designs.

The package command controls several aspects of the completed system. For example, in the case of AI Engine designs, the `--package.defer_aie_run` command indicates that the graph application in the `libadff.a` should not be started at boot time, and should instead wait until called expressly from a software application. The `--package.boot_mode` indicates that the system is booted from an SD card, or QSPI/OSPI, and the output produced by the package process is generated accordingly. Refer to [--package Options](#) for a complete list of options.

Finally, the `--package` command lets you define the required files for all platforms to boot and run the embedded system design for software or hardware emulation, or to create an SD card to run your system on hardware. For an embedded system design this process is described in [Packaging for Embedded Platforms](#).

## Booting and Running the System

When running the application, you can run software emulation, hardware emulation, or run on the actual physical platform. Running the application on embedded processor platforms is different from running on data center accelerator cards. For more information, refer to [Running the Application](#).

- When the build target is software or hardware emulation, the QEMU environment models the hardware device. The Vitis compiler generates simulation models of the kernels in the device binary and running the application runs in the QEMU model of the system. As described in [Build Targets](#), emulation targets let you build, run, and iterate the design over relatively quick cycles; debugging the application and evaluating performance.
  - When the build target is the hardware system, the target platform is the physical device. The Vitis compiler generates the `.xclbin` using the Vivado Design Suite to run synthesis and implementation, and resolve timing. Running the application runs your system on the hardware.
- 

## Next Steps

This introduction provided numerous references to more detailed information related to topics discussed here. You are strongly encouraged to review that material to develop a deeper understanding of the tools and flows described here.

For an example, on the different elements of the flow, and putting them together to build a complete system, you can refer to the [Versal Custom Thin Platform Extensible System](#) tutorial, based on a thin custom platform (minimal clocks and AXI exposed to PL) including both HLS and RTL kernels and AI Engine graph using a full Makefile build-flow.

You can also look through the [Vitis-Tutorials/AI Engine Development](#) repository for additional examples of the extensible platform development flow.

Finally, there is a chapter on [Vitis-Tutorials/Vitis Platform Creation](#) for some examples of custom embedded processor platform development.

# 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 Xilinx FPGA-based 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 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 Xilinx 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 should 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 should 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 should be grouped as memory reads and writes in separate functions using a load-compute-store architecture. The kernels should access contiguous memory if possible and the number of accesses should be optimized by removing redundant accesses or by creating a local cache.

You are encouraged to review this material as well as 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 should 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 you will encounter 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 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 will be necessary for you to understand some of the basics of how this is done to architect your code for best results.

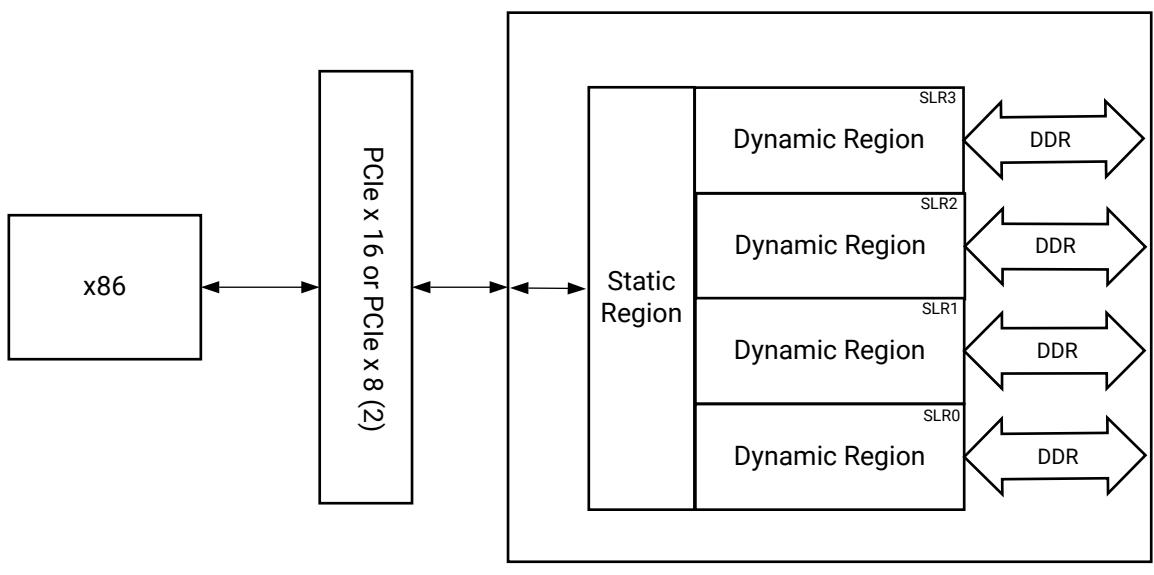
## Alveo Block Diagram and Data Movement between Host and FPGA

Using FPGAs at its core, Xilinx 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 Xilinx 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 7: Alveo Block Diagram



X26735-060122

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 may 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.

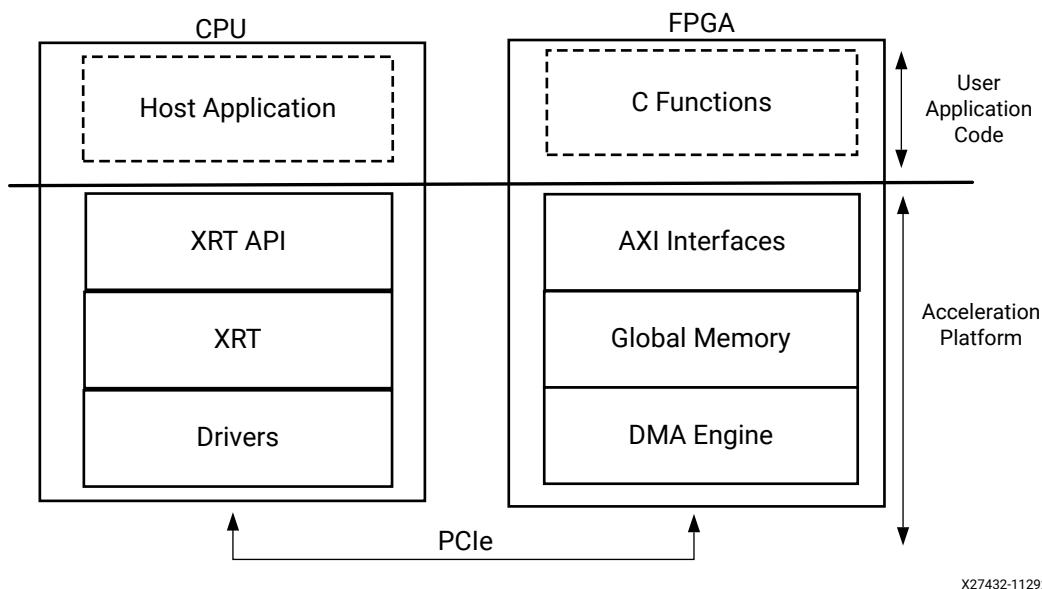
---

## 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 may 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 Xilinx device on the Alveo accelerator card.

Figure 8: 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

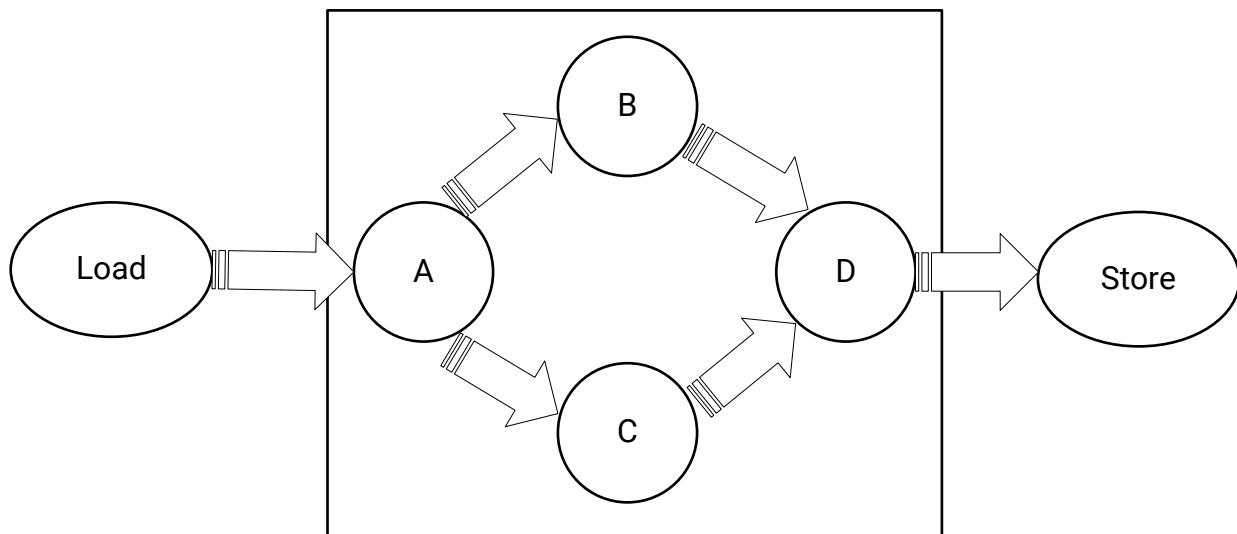
You will need to re-architect the compute function that resides on the FPGA as an accelerated kernel, and the host application that runs on the CPU and communicates with the accelerated kernels.

## 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 9: Kernel Architecture



X27496-120522

Here are 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_tti=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_tti=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_tti=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_tti=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_tti=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_tti=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);

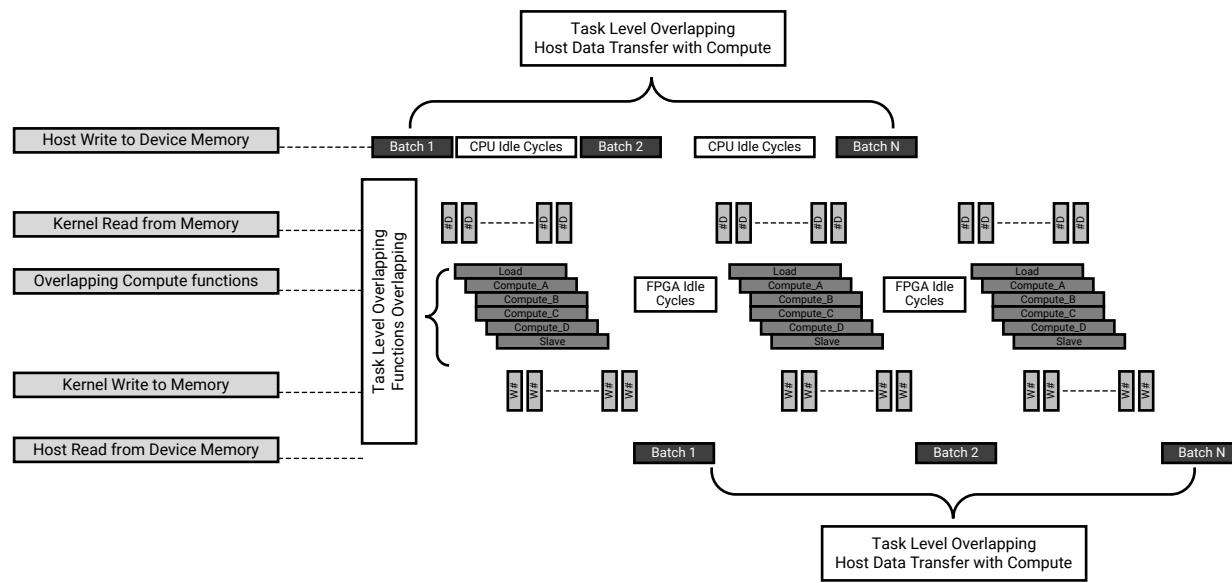
    for (int i = 0; i < totalNumWords; i++) {
        std::cout << "Referece   " << bufReference[i] << std::endl;
        std::cout << "Out      " << bufOut_map[i] << std::endl;
    }

    // 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 look like the following.

*Figure 10: Application Timeline*



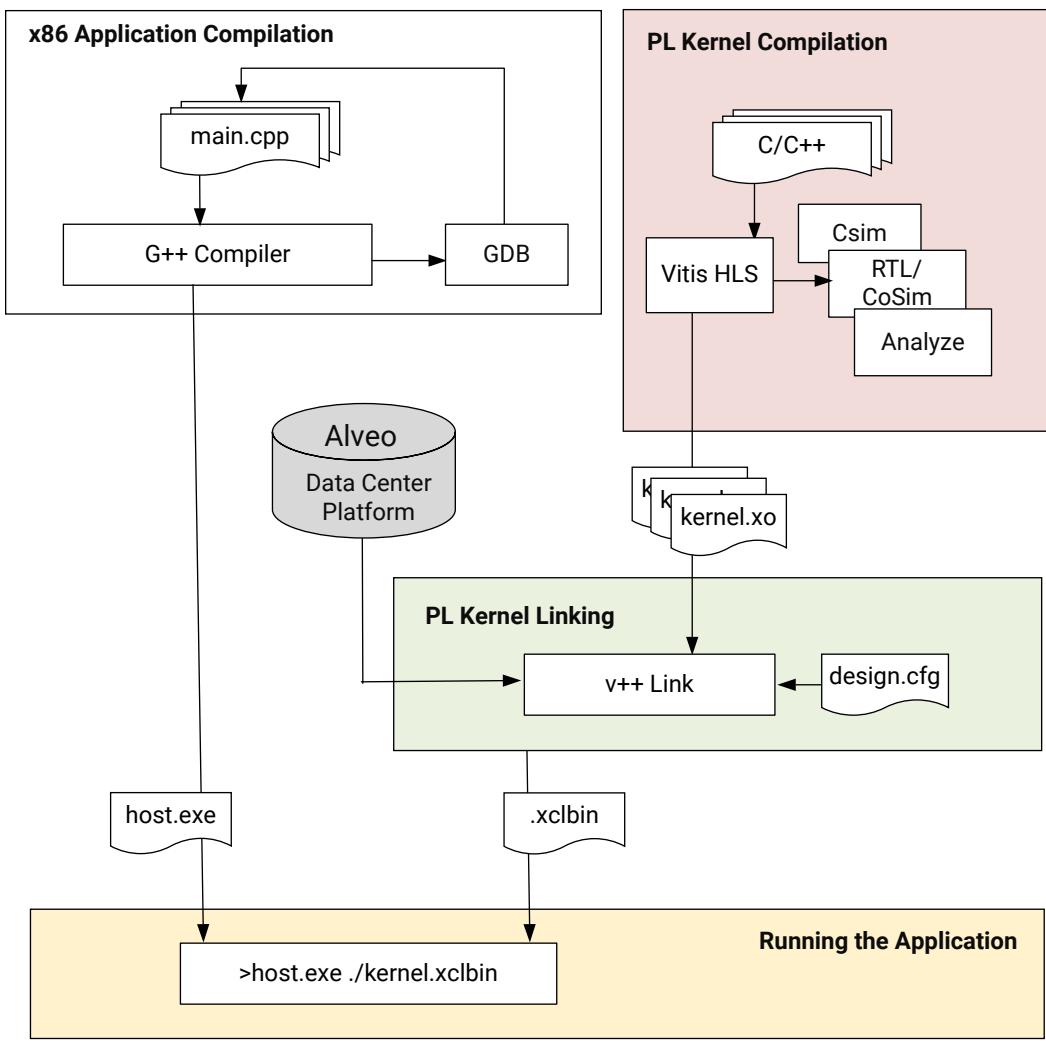
The execution of the application on an FPGA is quite different than on a CPU due to several types of parallelism that can be observed from figure above. The kernel code was written to leverage task-level parallelism by creating sub-functions for each loop. The result is that `compute_A`, `compute_B`, `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 as well as 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 11: Vitis Development Flow



X24704-052421

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 Xilinx 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, a configuration file can be created specifying option like how many instances of a kernel (or Compute Unit) should be built in the device binary, how are the kernels connected to the global memory, or to other kernels, etc. This configuration file is passed to the Vitis linker to generate the .xclbin.

There are three different build targets of the Vitis Compiler that defines the nature and contents of the generated .xclbin file. Two emulation targets used for validation and debugging purposes: software emulation for C-based simulation, and hardware emulation for RTL co-simulation; and one 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 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.

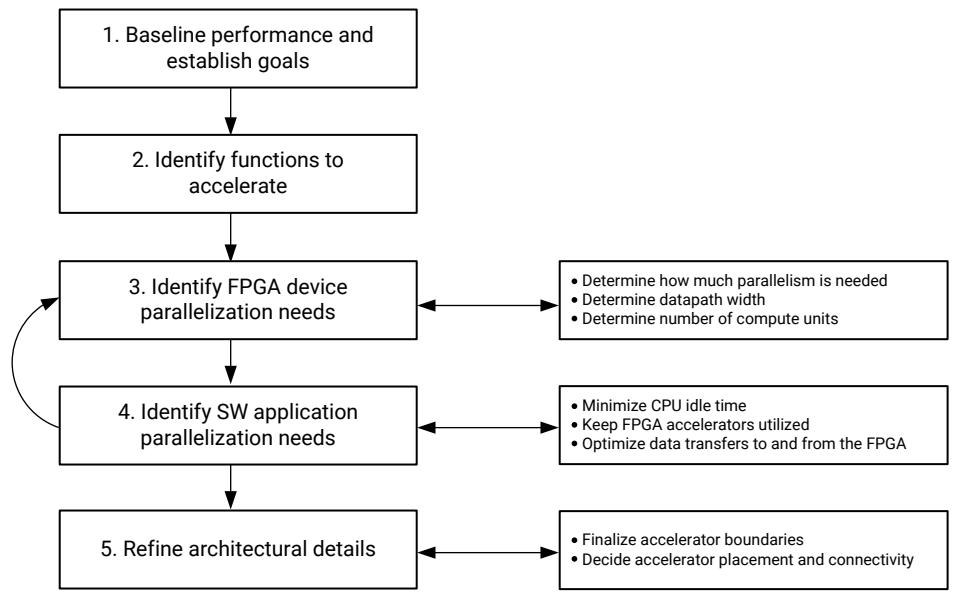
## Developing Vitis Accelerated Applications

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 should be mapped to device kernels, how much parallelism is needed, and how it should 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 the [Design Principles for Software Programmers](#) intended for software developers who want to understand the process of synthesizing accelerated hardware from a software algorithm written in C/C++

**Figure 12: Methodology for Architecting the Application**



X23282-092619

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 `xbutil dma` test.
- Identify the performance bottlenecks by reviewing the algorithm and analyzing any parallel paths. Accelerating one path may not give the expected acceleration for the overall application. When looking for acceleration candidates, consider the performance of the entire application, not just of individual functions.
- Identify the overall acceleration potential, set the application performance goal.
- After the functions to be accelerated have been identified and the overall acceleration goals have been established, the next step is to determine what level of parallelization is needed to meet the goals.

- 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 [Methodology for Accelerating Data Center Applications with the Vitis Software Platform](#), or refer to [Design Principles for Software Programmers](#) in the Vitis HLS User Guide (UG1399).

### **Methodology for Developing C/C++ Kernels**

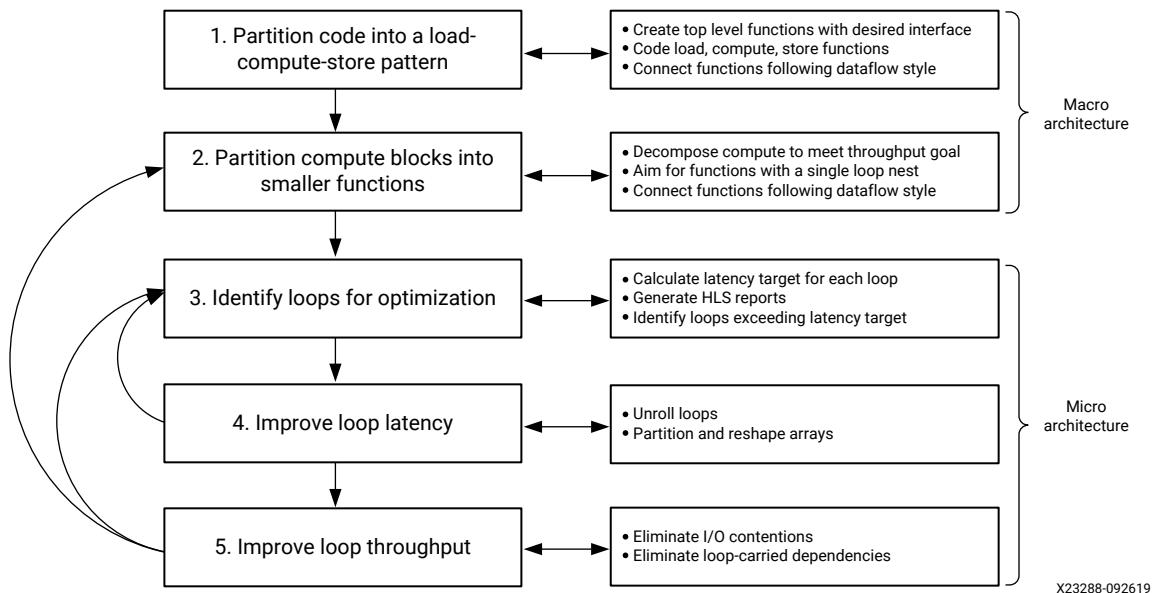
The software program can be automatically converted (or synthesized) into hardware, but achieving acceptable quality of results (QoR) will require additional work such as rewriting the software to help the Vitis HLS tool achieve the desired performance goals. To help, you need to understand the best 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 should 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 13: 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 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

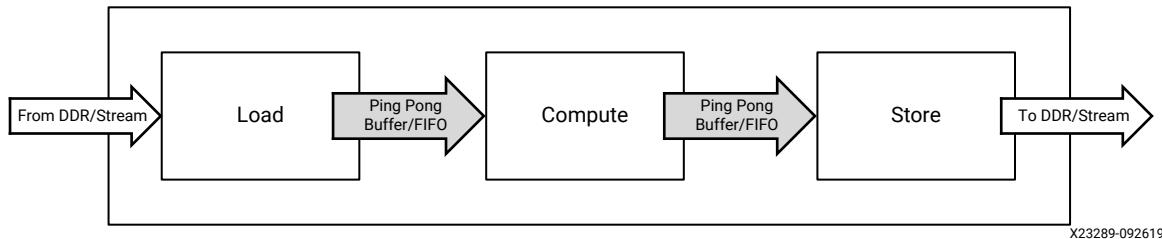
The Vitis HLS tool automatically assigns interface ports for the arguments of your C/C++ kernel function. These function arguments are of either scalar or pointer/array types. The parameters from the host are written directly to the registers of the accelerators. The buffers are kept external in the 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 should be structured as load-compute-store with communications channels in between as shown below.

Figure 14: Load-Compute-Store Pattern



X23289-092619

- 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 don't 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 should be 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.
  - Review the checklist in [Designing Efficient Kernels](#) for best practices to use when designing interfaces for your application.
- 

## Best Practices for Host Programming

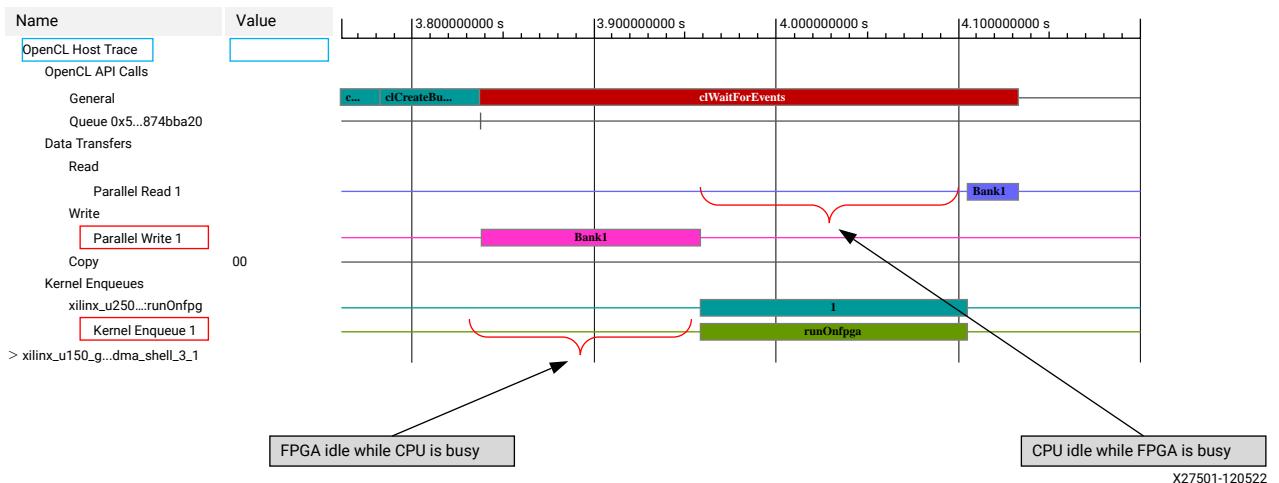
In the Vitis environment, the host application can be written in C++ using the Xilinx® runtime (XRT) native C++ API. Just as re-architecting the kernel code is required to enable parallelism on the hardware and optimize the memory accesses, host programming is equally important to ensure high performance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

- Reducing the overhead of kernel enqueueing: There is an overhead of dispatching the commands and arguments by the host to the kernel before enqueueing 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 should be large enough to maximize the PCIe bandwidth. At the same time, the buffer shouldn't be too large to initiate the kernel execution.
- Data availability for the kernel compute: The host should send the data to the FPGA device 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 may 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 as well as kernel and device memory.

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

Figure 15: Timeline Trace (host application timeline)

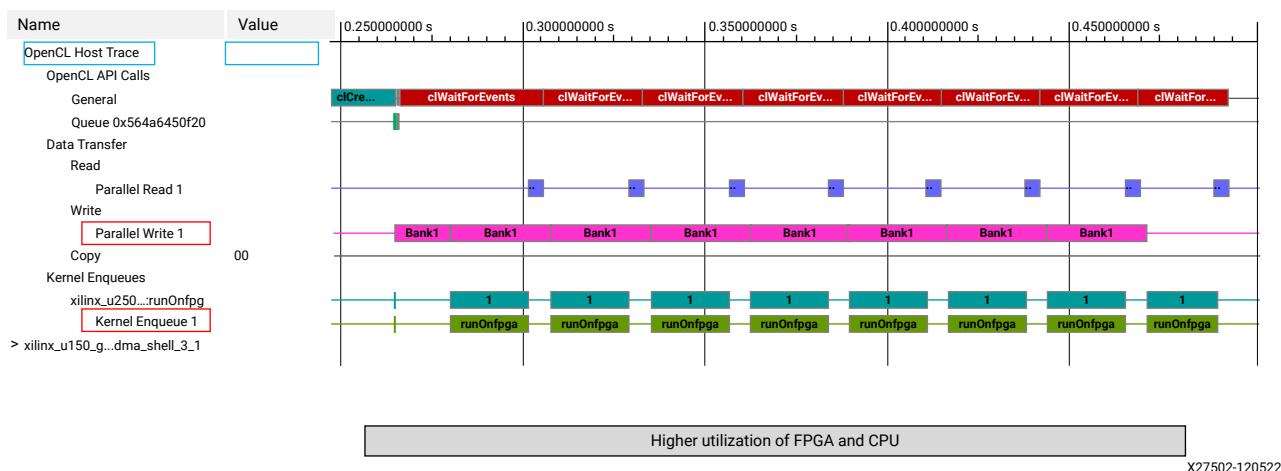


The row with "Data Transfer:Read" displays when the host is reading the data from the device memory and "Data Transfer:Write" displays when the host is writing the data to the device memory. The row with "Kernel Enqueue" displays when the kernel is executing.

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 16: 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 just limited to showing the timeline trace but also provides application guidance on profiling, presents the synthesis reports with timing and resource information as well as the critical path in your design. You can refer to [Using the Vitis Analyzer](#) for more information on the tool.

For more information on host programming refer to [XRT Native API](#), and the [Example - Overlap of Host program and Kernel execution](#).

## 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 Xilinx FPGA-based Alveo accelerator cards. The Vitis application acceleration development flow provides a model to combine RTL designs and a host application into a unified system running on 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 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 should 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 you will encounter 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.

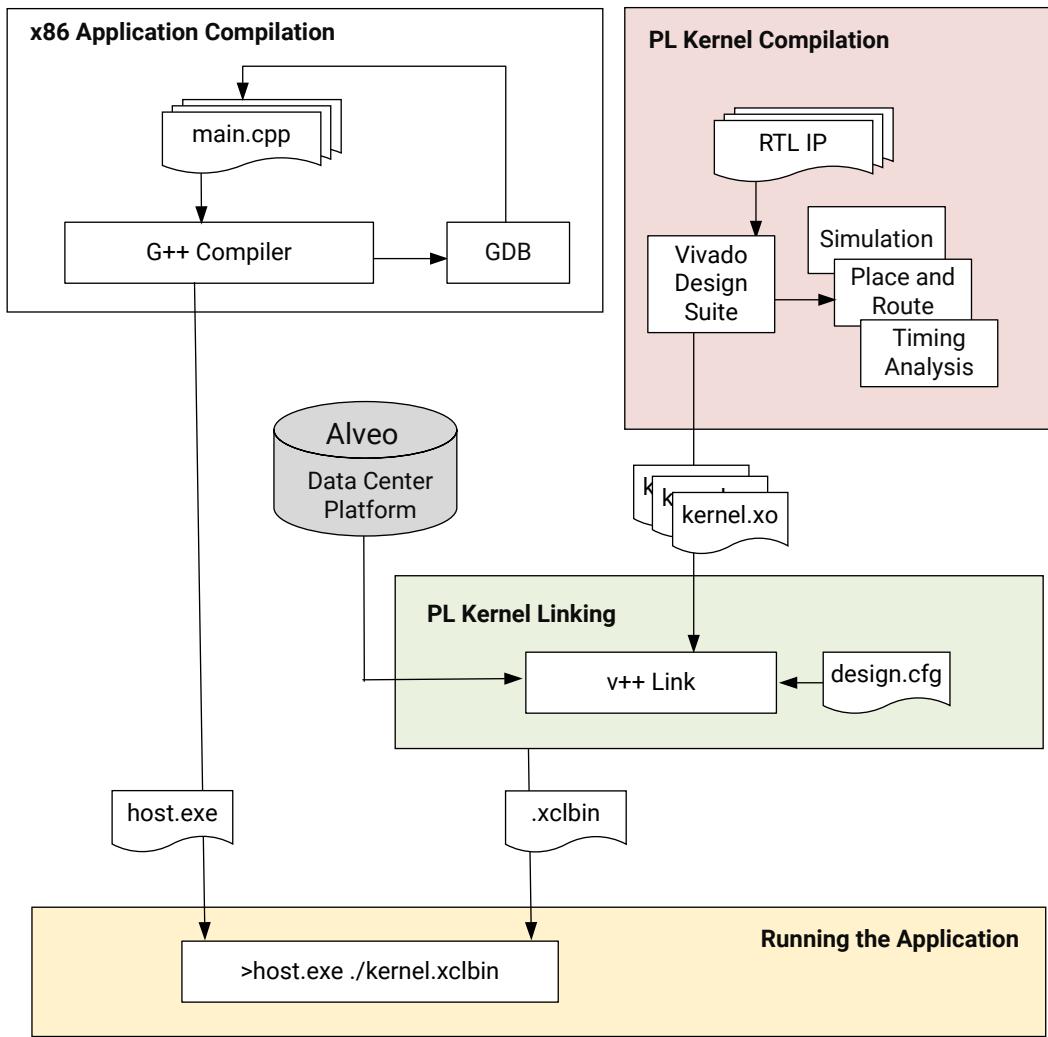
- **Vivado Design Suite:** An RTL language synthesis and implementation tool that takes the RTL design generated by Vitis HLS or an RTL designer and generates the bitstream that can be loaded and executed on the FPGA.
- **Bitstream:** Is the configuration data that is used to program the FPGA so that 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 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 17: Vitis Development Flow with RTL Kernels**



X24704-042122

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 Xilinx 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 define how many instances of a kernel (or Compute Unit) should be built 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.

---

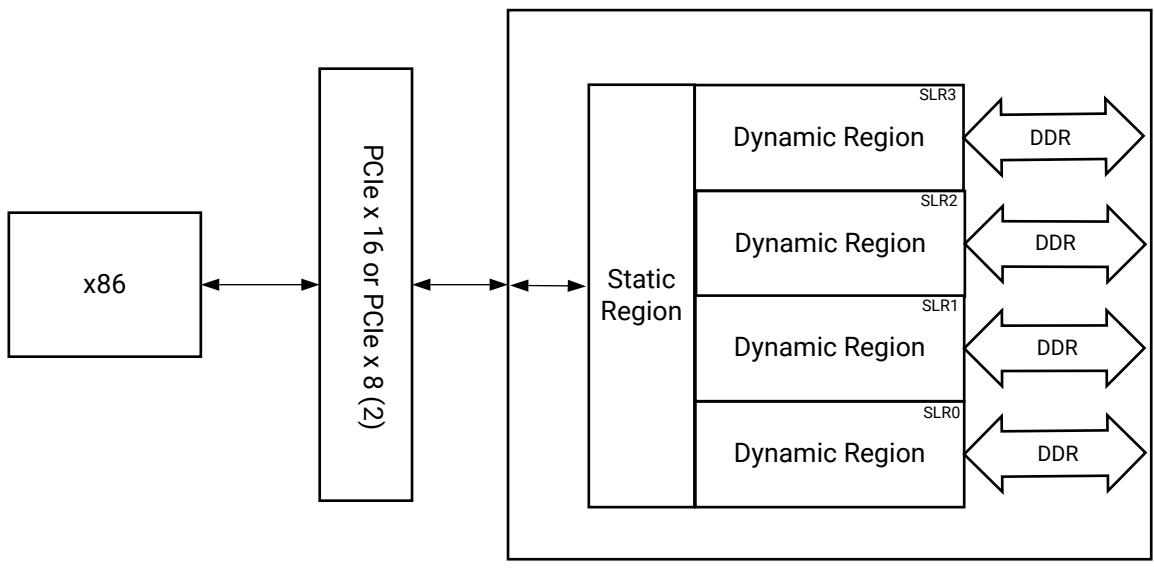
## Using Alveo Accelerator Cards

Xilinx 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 Xilinx 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 18: Alveo Block Diagram



X26735-060122

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 may have DDR or HBM, or even both.

The block diagram shown above is of U250 and has 4 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 heart 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 traditional 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 Kernels](#). 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/BRAM/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 [Build Targets](#), but do not support software emulation builds, as the RTL kernel does not support inclusion of a C-model.

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 and Implementation Results](#).

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.

# Tutorials and Examples

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

- **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 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 software emulation, hardware emulation, running on hardware, and a `README.md` file with a detailed explanation of the example.

# Developing Applications

This section contains the following chapters:

- [Programming Model](#)
- [PL Kernel Properties](#)
- [Writing the Software Application](#)
- [Developing PL Kernels using C++](#)
- [Packaging RTL Kernels](#)
- [Programming Versal AI Engines](#)
- [Best Practices for Data Center Acceleration with Vitis](#)

**Note:** For information about developing applications using a unified host and kernel composition refer to [Using Vitis System Compilation Mode](#).

# Programming Model

The Vitis™ core development kit supports heterogeneous computing using the Xilinx Run Time (XRT) application programming interface (API). The software application executes on the processor (x86 or Arm®) and offloads compute intensive tasks through XRT to execute on a hardware kernel running on programmable logic (PL) of a Xilinx device or on Versal® AI Engines. Review the [Section II: Introduction to Vitis Flows](#) for the different design flows and application supported by the Vitis tools.

---

## Design Topology

In the Vitis core development kit, targeted devices can include Xilinx® MPSoCs, Kria™ SOMs, Versal ACAPs, or UltraScale+™ FPGAs. The FPGA contains a programmable region that implements and executes a device binary (`.xclbin`) file that contains and connects hardware kernels as compiled Xilinx object (`.xo`) files and AI Engine graphs when appropriate.

The extensible FPGA platform contains one or more interfaces to global memory (DDR or HBM), and optional streaming interfaces (to other user-defined PL resources such as external I/Os).

PL kernels can access data through 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, or of the AI Engine array. Both memory-based and streaming connections are defined through Vitis linking options, as described in [Linking the Kernels](#).

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

For Versal AI Core devices the `.xclbin` file can also contain the AI Engine graph application (`libadff.a`). The `libadff.a` and PL kernels (`.xo`) are linked with the target platform (`.xpfm`) to define the hardware design. The AI Engine can be driven by PL kernels through `axis` interfaces. The AI Engine can also be controlled through the Arm processor (PS) via run-time parameters (RTP) in the graph and GMIO on Versal ACAP devices. Refer to *Versal ACAP AI Engine Programming Environment User Guide* ([UG1076](#)) for more information.

# Writing the Software Application

In the Vitis™ environment, the software application 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 software application is generally referred to as the host application for Data Center acceleration, and simply as the application for embedded processor-based systems.



**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 accelerator 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 kernel and returning results.

For Versal® ACAP devices, the PS application manages the whole heterogeneous system, including the PL hardware and the AI Engine graph application. Refer to [Programming the PS Host Application](#) in the *Versal ACAP AI Engine Programming Environment User Guide (UG1076)* for more information.

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` statement in your source code to load these classes:

```
#include <xrt/xrt_kernel.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 `xbutil` 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 should identify 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 should 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 Xilinx device that can be run by the host application.



**TIP:** As discussed in [Working with User-Managed Kernels](#), you should 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 also 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()`. You will assign a buffer object to each kernel buffer argument because buffer is not created 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 the 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.



**TIP:** 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, Xilinx 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) the `xrt::bo::sync()` command should be used 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 [Vitis-Tutorials/Hardware\\_Acceleration/Feature\\_Tutorials/01-rtl-kernel-workflow](#).

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.

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. Optionally, for Versal AI Core devices with AI Engine graph applications, you can also specify the graph application to load at run time. This process requires a few sub-tasks as shown:
  - a. Add required header to `#include` statement:

```
#include <experimental/xrt_aie.h>
```

- b. Identify the AI Engine graph from the `xrt::device` object, the `uuid` of the loaded `.xclbin`, and the name of the graph application:

```
auto my_graph = xrt::graph(device, uuid, "mygraph_top");
```

- c. Reset and run the graph application from the software program as needed:

```
my_graph.reset();
std::cout << STR_PASSED << "my_graph.reset()" << std::endl;
my_graph.run();
std::cout << STR_PASSED << "my_graph.run()" << std::endl;
```



**TIP:** For more information on building and running AI Engine applications, refer to Versal ACAP AI Engine Programming Environment User Guide ([UG1076](#)).

5. 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 should be 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_in_b = xrt::bo(device, DATA_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)
```

## 6. 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, you should use `xrt::bo::sync()` with direction flag for the DMA operation.

## 7. 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);
```

8. 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);
}
```

9. 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.

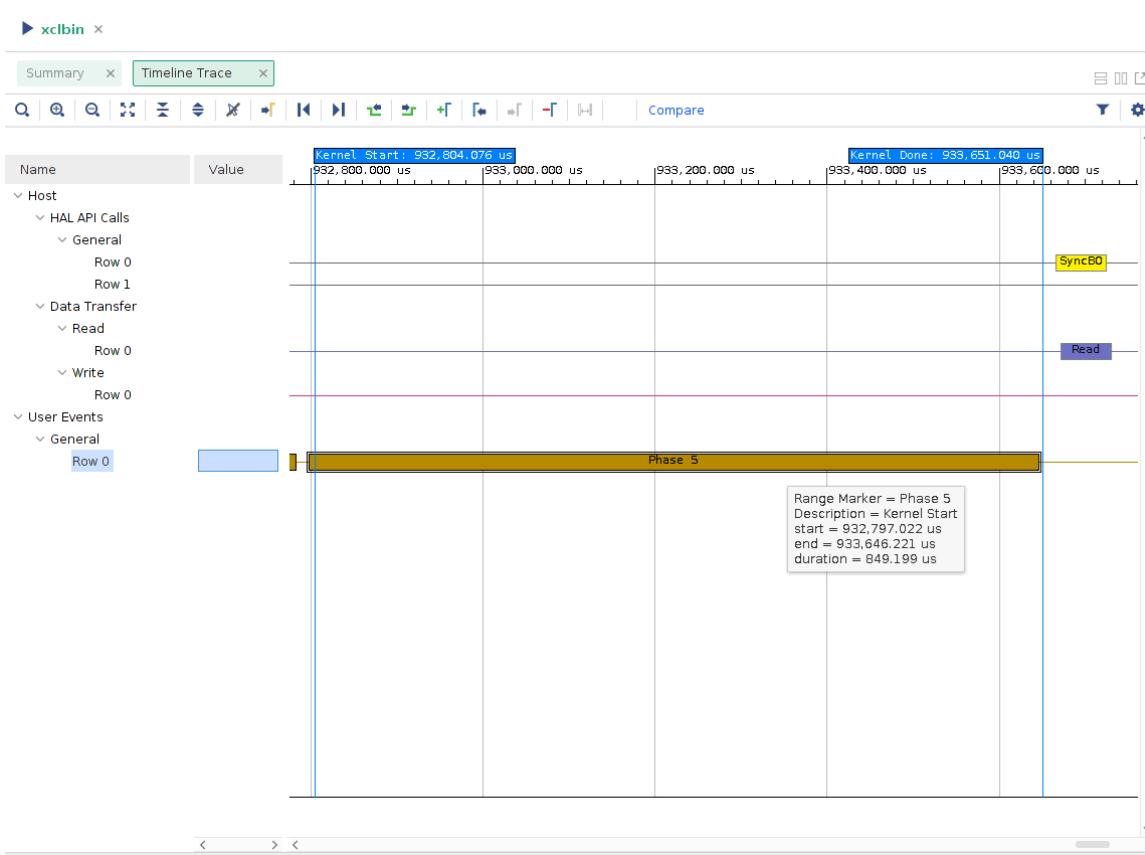
```
buf_in_b.sync(XCL_BO_SYNC_BO_FROM_DEVICE);
```

10. 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.



## Working with AI Engine Graphs



**TIP:** For an example of an embedded system design that includes an AI Engine graph application refer to [Vitis-Tutorials/Developer\\_Contributed/01-Versal\\_Custom\\_Thin\\_Platform\\_Extensible\\_System/](#).

For Versal AI Core devices, the AI Engine graph application is an xxx. The graph application (`libadaf.a`) is packaged as part of the device binary (`.xclbin`) by the `v++ --package` command, and also copied to the SD card. The following is an example of how you would load and run the graph application from the software program.

1. Add required header to `#include` statement:

```
#include <experimental/xrt_aie.h>
```

2. Set up the application environment as described in [Specifying the Device ID and Loading the XCLBIN](#).
3. Identify the AI Engine graph from the `xrt::device` object, the `uuid` of the loaded `.xclbin`, and the `name` of the graph application:

```
auto my_graph = xrt::graph(device, uuid, "mygraph_top");
```

4. Reset and run the graph application from the software program as needed:

```
my_graph.reset();
std::cout << STR_PASSED << "my_graph.reset()" << std::endl;
my_graph.run(0);
std::cout << STR_PASSED << "my_graph.run()" << std::endl;
```



**TIP:** For more information on building and running AI Engine applications refer to Versal ACAP AI Engine Programming Environment User Guide ([UG1076](#)).

---

## 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.

# PL Kernel Properties

In the Vitis application acceleration development flow, PL kernels (or compiled Xilinx object (.xo) files) are the processing elements executing in the programmable logic region of the Xilinx device. The Vitis core development kit supports kernels written in C/C++ and compiled in Vitis HLS, or designed in RTL IP and 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.

Kernels can be defined as software controllable, or non-software controlled. This means that the kernel is controlled through software such as the host application, or is un-managed by software and is instead data driven.

---

## 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 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 Vitis HLS, XRT-managed kernels are better for C++ developers as described in [Developing PL Kernels using C++](#). 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 using the XRT native C/C++ API, which supports both user-managed kernels and XRT-managed kernels, as well as 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.

**Table 4: Software Control Using the XRT API**

XRT-Managed Kernels	User-Managed Kernels
<ul style="list-style-type: none"><li>The object class for an XRT-managed kernel is <code>xrt::kernel</code></li><li>The software application communicates with the XRT-managed kernel using higher-level commands such as <code>set_arg</code>, <code>run</code>, and <code>wait</code></li><li>The user does not need to know the low-level details of the programmable registers and kernel execution protocols</li><li>Control and status registers provide XRT with a known interface to interact with the kernel, which makes these high-level commands possible</li><li>If needed, it is also possible to control an XRT-managed kernel as a user-managed kernel (using atomic register reads and write)</li><li>OpenCL API can also be used with XRT-managed kernels (<code>cl::kernel</code>)</li></ul>	<ul style="list-style-type: none"><li>The object class for a user-managed kernel is <code>xrt::ip</code></li><li>The software application communicates with the user-managed kernel using atomic register reads and writes through the AXI4-Lite interface</li><li>The application developer is responsible for knowing the address offset and purpose of each register in the kernel, and using them properly</li><li>There are no checks, high-level controls, or profiling capabilities. The user is responsible for running the simulations for performance analysis/debugging.</li></ul>

## 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, using Vitis HLS, 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 structure the user specifies.

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

**Table 5: Kernel Interface Types**

Register (AXI4-Lite)	Memory Mapped (M_AXI)	Streaming (AXI4-Stream)
<ul style="list-style-type: none"><li>Register interfaces must be implemented using a single AXI4-Lite interface.</li><li>Designed for transferring scalars between the host application and the kernel.</li><li>Register reads and writes are initiated by the host application.</li><li>The kernel acts as a slave.</li></ul>	<ul style="list-style-type: none"><li>Memory mapped interfaces must be implemented using one or more AXI4 Masters interfaces.</li><li>Designed for bi-directional data transfers with global memory (DDR, PLRAM, HBM).</li><li>Introduces additional latency for memory transfers.</li><li>The kernel acts as a master accessing data stored into global memory.</li><li>The host application allocates the buffer for the size of the dataset.</li><li>The base address of the buffer is provided by the host application to the kernel via its AXI4-Lite interface.</li></ul>	<ul style="list-style-type: none"><li>Streaming interfaces must be implemented using one or more AXI4-Stream interfaces.</li><li>Designed for uni-directional data transfers between kernels.</li><li>The access pattern is sequential.</li><li>Does not use global memory.</li><li>Data set is unbounded.</li><li>A sideband signal can be used to indicate the last value in the stream.</li></ul>

## 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](#)).
- 

## Non-Software Controlled Kernels

These kernels are present in the device but are not visible to or directly accessible by the software application. These kernels do not have a programmable register interface. The kernels must have at least one AXI4-Stream interface. The kernel synchronizes with the rest of the system through these streaming interfaces.

Non-software controlled kernels are considered an advanced feature and should only be used when a software controllable kernel cannot be used. Because they do not have a programmable register interface, control-related information needs to be passed through the data interfaces of the kernel.

Non-software controlled kernels do not require a software API, as the host application is not interacting directly with the kernel. The kernel can be developed as either [Packaging RTL Kernels](#), or as [Developing PL Kernels using C++](#) as described in Auto-Restarting Kernels in *Vitis High-Level Synthesis User Guide* ([UG1399](#)).

## HW Interfaces

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

- Programmable interface:** There is no AXI4-Lite interface.
  - Data interfaces:** At least one AXI4-Stream interfaces.
  - Clock and resets:** As described in [Clock and Reset Requirements](#).
- 

## Clock and Reset Requirements

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

**Table 6: Requirements**

C/C++/OpenCL C Kernel	RTL Kernel
<ul style="list-style-type: none"> <li>C kernel does not require any input from user on clock ports and reset ports. The HLS tool always generates RTL with clock port <code>ap_clk</code> and reset port <code>ap_rst_n</code>.</li> <li>HLS kernels can only have one clock/reset.</li> </ul>	<ul style="list-style-type: none"> <li>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, but most embedded platforms can have multiple clocks.</li> <li>An active-Low reset port can optionally be associated with a clock through the ASSOCIATED_RESET parameter on the clock.</li> </ul>

# Developing PL Kernels using C++

In the Vitis™ core development kit, the PL kernel code is generally a compute-intensive part of the system, and intended to run on the programmable logic (PL) region of a Xilinx device. The Vitis core development kit supports PL kernel code written in C or C++, and also written in RTL. For C/C++ based kernels the Vitis HLS tool offers the best place to start.

The Vitis HLS tool, which is part of the Vitis core development kit, is targeted at the development of PL kernels written in C or C++. The environment 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. Vitis HLS is the recommended tool for developing PL kernels for use in the Vivado traditional design flow, or in the Vitis heterogeneous design flow.

Generally, off-the-shelf software cannot be efficiently converted into hardware running on an FPGA. 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 Vitis HLS achieve the desired performance goals. To help, you need to understand the best practices for writing good software for execution on the FPGA as discussed in [Design Principles for Software Programmers](#) in the *Vitis HLS User Guide* (UG1399).

The code for C++ kernels written and optimized in Vitis HLS can be compiled in the Vitis tools either from the `v++` command line as explained in [Compiling C/C++ PL Kernels](#), or from an Application project in the Vitis IDE as explained in [Creating a Vitis IDE Project](#).



**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 Vitis application acceleration development flow, RTL IP from the Vivado® Design Suite can be packaged as kernels (or compiled Xilinx object (.xo) files) that can be linked into an FPGA executable (.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 adhere to 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 [Build Targets](#), but an RTL kernel in its native form does not support software emulation. To support software emulation, you must add a C-model to the packaged RTL kernel, as described in [Adding C-Models to RTL Kernels](#).

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, as well as any additional requirements. Finally, the development flow is described to help you package RTL IP in the Vivado® 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 7: RTL Kernel Interface and Port Requirements**

Port or Interface	Description	Comment
Clock	One or more clock inputs.	<ul style="list-style-type: none"><li>At least one clock is required for the kernel.<sup>1</sup></li><li>Can be named anything, but must be packaged with a bus interface.</li></ul> <p><b>IMPORTANT!</b> <i>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:</i></p> <div style="background-color: #f0f0f0; padding: 5px;"><p>ERROR: UNDEF When packaging for Vitis, pins that are not part of an interface are not supported</p></div>
Reset	Primary active-Low reset input port	<ul style="list-style-type: none"><li>Optional port.</li><li>Can be named anything, but must be associated with a Clock signal through the ASSOCIATED_RESET property on the Clock.</li><li>This signal should be internally pipelined to improve timing.</li><li>The signal is driven by a synchronous reset in the associated Clock domain.</li></ul>
interrupt	Active-High interrupt.	<ul style="list-style-type: none"><li>Optional port.</li><li>When used, the name must be exactly as shown.</li></ul>

Table 7: RTL Kernel Interface and Port Requirements (cont'd)

Port or Interface	Description	Comment
s_axi_control	One (and only one) AXI4-Lite slave control interface	<ul style="list-style-type: none"> <li>Required port. The <code>s_axilite</code> interface is generally required with exception for some cases using AXI4-Stream interfaces. It is not required for non-software controlled kernels.</li> <li>When used, the name must be exactly as shown, and is case-sensitive.</li> </ul> <p><b>Note:</b> The address range of the <code>s_axilite</code> interface can be edited in the <code>kernel.xml</code> file and repackaged using the <code>package_xo</code> command if needed.</p>
AXI4_Memory Mapped Interface (m_axi)	AXI4 memory mapped interfaces for global memory access	<ul style="list-style-type: none"> <li>Optional port.</li> <li>All AXI4 memory mapped interfaces must have 64-bit addresses (32 bits on Zynq-7000 devices).</li> <li>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.</li> <li>AXI4 memory mapped must not use Wrap or Fixed burst types and must not use narrow (sub-size) bursts. This means that AxSIZE should match the width of the AXI data bus.</li> <li>Any user logic or RTL code that does not conform to the requirements above, must be wrapped or bridged to satisfy these requirements.</li> </ul>
AXI4_STREAM (axis)	AXI4-Stream interfaces for one-way data transfers between kernels or between the host application and kernels.	<ul style="list-style-type: none"> <li>Optional port.</li> <li>Cannot be used with bi-directional ports.</li> <li>Use the STREAM interface template in the Vivado Design Suite.</li> <li>Refer to <a href="#">AXI4-Stream Interfaces</a> in the <i>Vitis HLS User Guide</i> (UG1399) for additional information on interface requirements.</li> </ul>

**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 8: 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 the different control protocols as explained in [Supported Kernel Execution Models](#) in the XRT documentation. 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 9: Control Register Signals**

Bit	Name	Description
0	ap_start	Asserted when the kernel can start processing data. Cleared on handshake with <code>ap_done</code> 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 <i>Vitis High-Level Synthesis User Guide</i> ( <a href="#">UG1399</a> ).
31:5	Reserved	Reserved

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

**Table 10: 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 11: 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 12: 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.

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 should 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.

## 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](#), 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 Working with Auto-Restarting Kernels in the Vitis HLS 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 Xilinx 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 should 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.

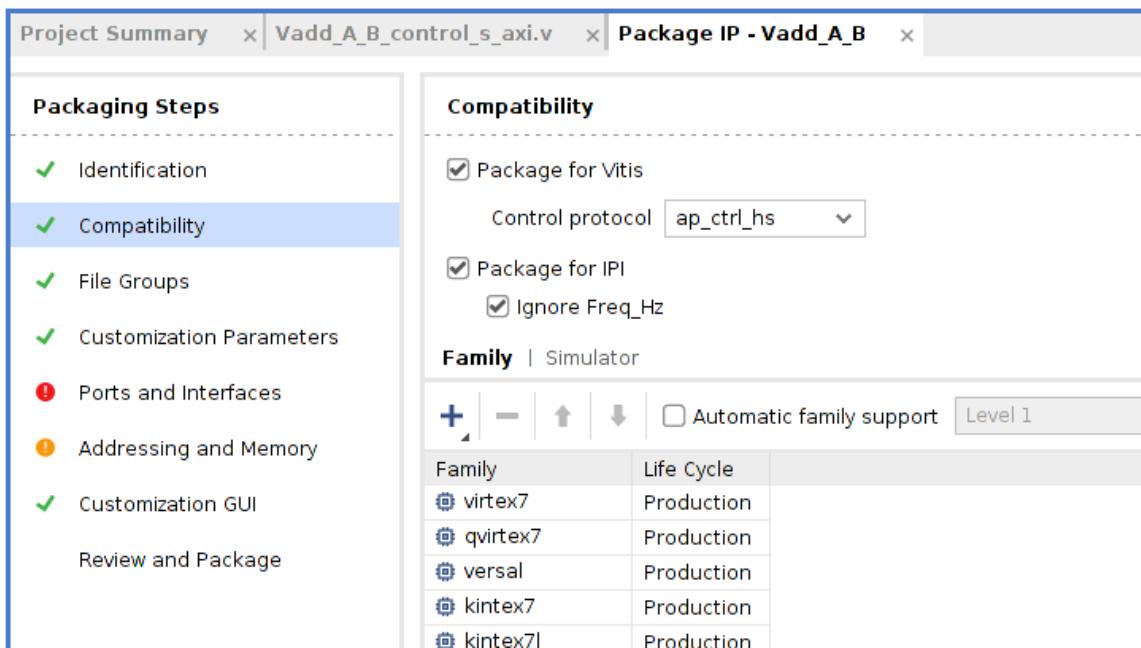


**RECOMMENDED:** Xilinx strongly recommends that AXI4 interfaces be packaged with AXI meta data `HAS_BURST=0` and `SUPPORTS_NARROW_BURST=0`. These properties can be set in an IP-level `bd.tcl` file. This indicates wrap and fixed burst type is not used, and narrow (sub-size burst) is not used.

- 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 [Non-Software Controlled 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 figure above, 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 table below, but the process of assigning names, offsets, and widths is the same.

**Table 13: Address Map**

Register Name	Description	Address Offset	Size
CTRL	Control Signals as described in <a href="#">Control Requirements for XRT-Managed Kernels</a> .	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
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 <code>0x10</code> ; locations below this are reserved.	0x010	32/64 Scalar arguments are 32-bits <code>wide.m_axi</code> and <code>axis</code> 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 should look similar to the following example.

Name	Display Name	Description
s_axi_control		

Name	Display Name	Description	Base Address	Range	Range Dependency
reg0			0	4096	$\text{pow}(2, (\text{C_S_AXI_CONTROL_ADDR_WIDTH} - 1) + 1)$

Name	Display Name	Description	Address Offset	Size
CTRL		Control signals	0x000	32
GIER		Global Interrupt Enable Register	0x004	32
IP_JER		IP Interrupt Enable Register	0x008	32
IP_ISR		IP Interrupt Status Register	0x00C	32
scalar00			0x010	32
A			0x018	64

Name	Description	Display Name	Value	Value Bit String Length	Value Format	Value Source	Value Validation List	Maximum	Minimum	Parameter Types
ASSOCIATED_BUSIF		m00_axi	0	string	default					
B					0x024					64

Name	Description	Display Name	Value	Value Bit String Length	Value Format	Value Source	Value Validation List	Maximum	Minimum	Parameter Types
ASSOCIATED		m01_axi	0	string	default					

Memory Maps (for slaves) Address Spaces (for masters)



**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.

- 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.

- At this point you should be ready to package your IP.

- 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.

- 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 [Section IV: Building and Running the Application](#) 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](#) for details of the command. Some specific reasons why you may 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
  - Include a C-model using the `package_xo -kernel_files` option to enable software emulation for your RTL kernel as described in [Adding C-Models to RTL Kernels](#)

#### 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 should show the various interfaces described above. Examine the IP in the canvas view. The properties of the AXI interface can be viewed by selecting the interface on the canvas. Then in the Block Interface Properties 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>
```

## Adding C-Models to RTL Kernels

RTL kernels support the hardware emulation build, and the hardware build described in [Build Targets](#), but an RTL kernel in its native form does not support software emulation. To support software emulation, you must add a C-model to the packaged RTL kernel using the package\_xo-kernel\_files option.



**TIP:** C-models are not supported for user-managed kernels. They are only supported for ap\_ctrl\_hs and ap\_ctrl\_chain kernels.

The C-model can be a simple C/C++ application defining the RTL function that is packaged with the RTL code for the kernel. The C-model is included in the packaged kernel (.xo) in a folder called `cpu_sources`, which the Vitis compiler uses during software emulation. For the hardware emulation and hardware builds, the Vitis compiler always use the RTL implementation of the kernel when creating the device binary (.xclbin).

The Vitis IDE also provides a mechanism for including a C-model with the RTL kernel as described in [RTL Kernel Wizard](#).

---

## Design Recommendations for RTL Kernels

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

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

- [Memory Performance Optimizations for AXI4 Interface](#)

- 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 must cross between them should be flip-flop to flip-flop timing paths.

## Debug and Verification Considerations

- RTL kernels should be verified in their own test bench using advanced verification techniques including verification components, randomization, and protocol checkers. The AXI Verification IP (VIP) is available in the Vivado IP catalog and can help with the verification of AXI interfaces. The RTL kernel example designs contain an AXI VIP-based test bench with sample stimulus files.
- You can add ILA inside of RTL kernels as described in [Adding Debug IP to RTL Kernels](#).
- Hardware emulation should be used to test the host code software integration or to view the interaction between multiple kernels.

# Programming Versal AI Engines

As described in *AI Engine Kernel and Graph Programming Guide* ([UG1079](#)), an AI Engine kernel is a C/C++ program that is written using the AI Engine API and specialized intrinsic functions that target the VLIW scalar and vector processors of Versal® AI Core devices.

The AI Engine kernel code is compiled using the AI Engine compiler (`aiecompiler`) which is included in the Vitis core development kit. The AI Engine compiler produces ELF files that are run on the AI Engine processors. Multiple AI Engine kernels are combined in an adaptive data flow (ADF) graph that consists of nodes and edges where nodes represent compute kernel functions, and edges represent data connections. The ADF graph is a static flow dataflow graph with kernels operating in parallel on data streams. The ADF graph interacts with the PL kernels of the Vitis flow, global memory, and the host application described here.

Refer to *AI Engine Kernel and Graph Programming Guide* ([UG1079](#)) and *Versal ACAP AI Engine Programming Environment User Guide* ([UG1076](#)) for more information on developing Versal AI Engine applications.

# Building and Running the Application

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, followed by packaging the outputs for use. However, the first step in building the application is to identify the build target, indicating if you are building for test or simulation of the application, or building for the target hardware. After building, both the host program and the FPGA binary, you will be ready to run the application.

This section contains the following chapters:

- [Setting Up the Vitis Environment](#)
- [Build Targets](#)
- [Building the Software Application](#)
- [Building the Device Binary](#)
- [Packaging the System](#)
- [Running the Application](#)

# Setting Up the Vitis Environment

The Vitis™ unified software platform includes three elements that must be installed and configured to work together properly: the Vitis core development kit, the Xilinx® Runtime (XRT), and an accelerator card such as the Alveo™ Data Center accelerator card. The requirements of installation and configuration are described in [Installation](#).

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
```

You can also specify the location of the available platforms for use with your Vitis IDE by setting the following environment variable:

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



---

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

---

# Build Targets

The build target of the Vitis™ tool defines the nature and contents of the FPGA binary (.xclbin) created during compilation and linking. There are three different build targets: two emulation targets used for validation and debugging purposes: software emulation and hardware emulation, and the default system hardware target used to generate the FPGA binary (.xclbin) loaded into the Xilinx® device.

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

**Table 14: Comparison of Emulation Flows with Hardware Execution**

Software Emulation	Hardware Emulation	Hardware Execution
Host application runs with a C/C++ model of the kernels.	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.
Used to confirm functional correctness of the system.	Test the host / kernel integration, get performance estimates.	Confirm that the system runs correctly and with desired performance.
Fastest build time supports quick design iterations.	Best debug capabilities, moderate compilation time with increased visibility of the kernels.	Final FPGA implementation, long build time with accurate (actual) performance results.

## Software Emulation

The main goal of software emulation (`sw_emu`) is to ensure functional correctness of the host program and kernels. Software emulation provides a purely functional execution, without any modeling of timing delays, or latency; it does not give any indication of the accelerator performance.

The kernel code is always compiled and running natively. The application code is either:

- Compiled and running natively on an x86 processor (Data Center platforms)
- Cross-compiler to the Arm® processor and running in an emulator (Embedded platforms)

Thus, software emulation is typically used for algorithm refinement, debugging functional issues, and letting developers iterate quickly through the code to make improvements. The software programming model of fast compilation and run iterations is preserved.

The `v++` compiler does the minimum transformation of the kernel code to create the FPGA binary to run the host program and kernel code together. Software emulation takes the C-based kernel code and compiles it with GCC. It runs each kernel as a separate C-thread. If there are multiple compute units of a single kernel, each CU is run as a separate thread. Therefore, it mimics the parallel execution model of the hardware. However, within each kernel the execution is modeled sequentially although there might be parallelism within a kernel when running on hardware. The software emulation driver implements the XRT API and acts as a bridge between the user application running XRT and the device process modeling the hardware components.



**TIP:** For RTL kernels, software emulation can be supported if a C model is associated with the kernel. The [RTL Kernel Development Flow](#) provides an option to associate C model files with the RTL kernel for support of software emulation flows.

The following describes the software emulation limitations:

- There is a global memory limit of 16 GB which should not be exceeded for simulation purposes.
- Software emulation does not support [AXI4-Stream Interfaces without Side-Channels](#) in the [Vitis HLS User Guide \(UG1399\)](#).

As discussed in [v++ Command](#), the software emulation target is specified in the `v++` command with the `-t` option:

```
v++ -t sw_emu ...
```

You can use the GDB debugger for both the host application and the kernel code, set breakpoints or use `printf()` to print information and checkpoints. For details on how to debug the host application or the kernel during software emulation, refer to [Debugging in Software Emulation](#).

---

## Hardware Emulation

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
- SystemC models of HLS kernels can be used to speed up simulation when needed
- 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 Vivado logic simulator, with a waveform viewer to examine the kernel design. Some third-party simulators are also supported as described in [Simulator Support](#). In addition, hardware emulation provides performance and resource estimates for the hardware implementation.

SystemC models are provided for the key IP used in the hardware platform, like Versal NoC/DDR memory, CIPS, PS block, AI Engine, UltraScale+ MIG DDR memory, and AXI4 SmartConnect. These IP models are used during hardware emulation to improve simulation performance and results.

In hardware emulation, compile and execution times are longer than software emulation, but it provides a detailed, cycle-accurate, view of kernel activity. Xilinx 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 should be used for guidance and for comparing relative performance between different kernel implementations.

As discussed in [v++ Command](#), the hardware emulation target is specified in the v++ command with the -t option:

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

## System Hardware Target

When the build target is the hardware, v++ builds the FPGA binary for the Xilinx 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, or embedded processor platform, and the application can be run in its actual operating environment.

As discussed in [v++ Command](#), the system hardware target 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 or embedded Arm processor.



**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](#) prior to running the tools.

Each source file of the host application is compiled into an object file (`.o`) using the `g++` compiler.

```
g++ ... -c <source_file1> <source_file2> ... <source_fileN>
```

The generated object files (`.o`) are linked with the Xilinx Runtime (XRT) shared library to create the executable host program. Linking is performed using the `-l` option.

```
g++ ... -l <object_file1.o> ... <object_fileN.o>
```

Compiling and linking for x86 follows the standard `g++` flow. The only requirement is to include the XRT header files and link the XRT shared libraries.

The host application can be written in native C++ using the Xilinx runtime (XRT) native C++ API or industry standard OpenCL™ 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:

```
g++ -g -std=c++14 -I$XILINX_XRT/include -L$XILINX_XRT/lib -o host.exe
host.cpp \
-lxrt_coreutil -pthread
```

When compiling the source code, the following `g++` options are required:

- `-I$XILINX_XRT/include/`: XRT include directory.
- `-std=c++14`: Define the C++ language standard. Compiling host code with XRT native C++ API requires C++ standard with `-std=c++14` or newer. However, on GCC versions older than 4.9.0, use `-std=c++1y` instead.

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.

### **Building OpenCL API Host Code**

The 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 \
-lopenccl -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.

## **Compiling and Linking for Arm**



**TIP:** Set up the command shell or window as described in [Setting Up the Vitis Environment](#) prior to running the tools.

For embedded processor-based platforms, the host program (`host.exe`), is cross-compiled and linked for an Arm processor using the GNU Arm cross-compiler version of `g++` in the following two step process:



**TIP:** `aarch64` is used for Zynq® UltraScale+™ (A53) and Versal® (A72) devices. `aarch32` is used for Zynq-7000 SoC (A9) and the tool chain is in a different location.

1. Compile the host .cpp into an object file (.o):

```
$XILINX_VITIS/gnu/aarch64/lin/aarch64-linux/bin/aarch64-linux-gnu-g++ -c
\ -D__USE_XOPEN2K8 -I$SYSROOT/usr/include/xrt -I$XILINX_VIVADO/include \
-I$SYSROOT/usr/include -fmessage-length=0 -std=c++14 --sysroot=$SYSROOT \
-o src/host.o ../src/host.cpp
```

2. Link the object file with required libraries to build the executable application.

```
$XILINX_VITIS/gnu/aarch64/lin/aarch64-linux/bin/aarch64-linux-gnu-g++ -l
\ -lxrt_coreutil -lpthread -lrt -lstdc++ -lgmp -L$SYSROOT/usr/lib/ \
--sysroot=$SYSROOT -o host.exe src/host.o
```

When compiling the application for use with an embedded process, you must specify the `sysroot` for the application. The `sysroot` is part of the platform where the basic system root file structure is defined, and is installed as described in [Installing Embedded Platforms](#).



**IMPORTANT!** The above examples rely on the use of `$SYSROOT` environment variable that must be used to specify the location of the `sysroot` for your embedded platform.

The following are key elements to compiling the host code for an edge platform:

- **Compilation:**

- `-I$SYSROOT/usr/include`
- `-I$SYSROOT/usr/include/xrt`
- `-std=c++14`: Define the C++ language standard. Compiling host code with XRT native C++ API requires C++ standard with `-std=c++14` or newer. However, on GCC versions older than 4.9.0, use `-std=c++1y` instead.

- **Linking:**

- `-L$SYSROOT/usr/lib`: Library paths location.
- `-lxrt_coreutil`: Required library for use with the XRT native API.
- `-pthread`: Required by XRT library for multithreading.

## Building OpenCL API Host Code

The 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 \
-1OpenCL -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.

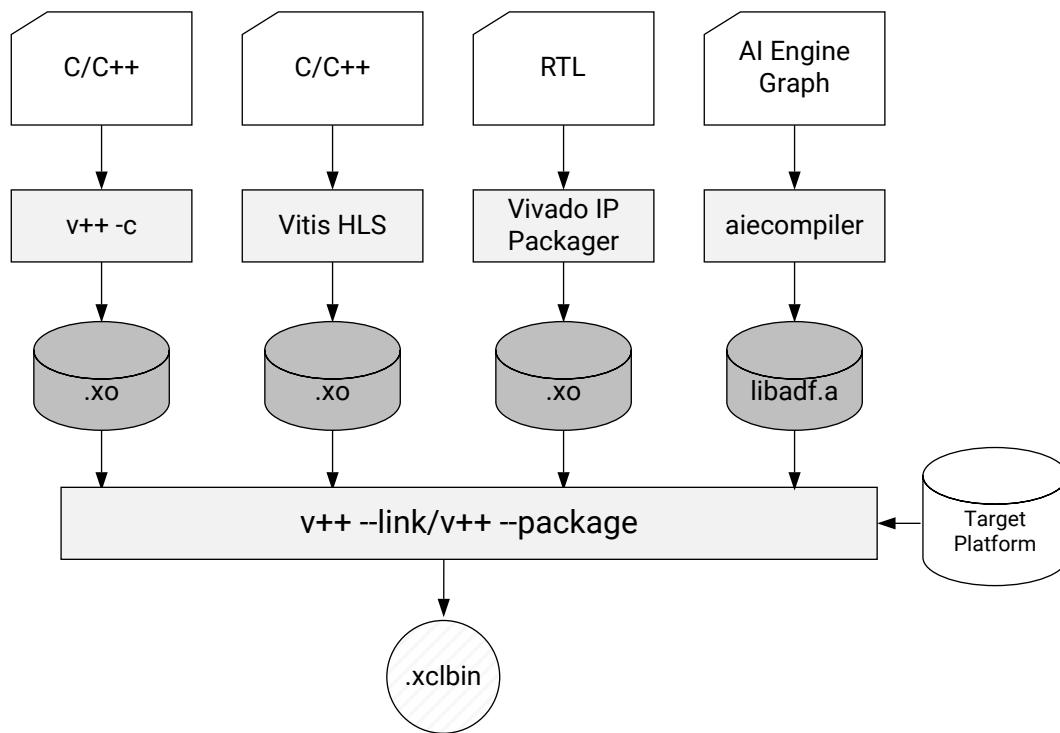
**Note:** In [Vitis Accel\\_Examples](#) you can see the addition of `xc12.cpp` source file, and the `-I ../../xc12` include statement. These additions to the host program and `g++` command provide access to helper utilities used by the example code, but are generally not required for your own code.

# Building the Device Binary

The PL kernel code is written in C++ or RTL, and is built by compiling the kernel code into a Xilinx® object (.XO) file, and linking the XO files into a Xilinx binary (.xclbin) file, as shown in the following figure.

When targeting a Versal AI Core device, the hardware can include an AI Engine graph application that is also written in C++, compiled with the `aiecompiler` tool, and results in a `libadbf.a` file that can also be linked into the device binary as shown below.

Figure 19: Device Build Process



X21155-060321

The process, as outlined above, has two steps:

1. Build the Xilinx object files from the kernel source code.

- For C, C++ kernels, the `v++ -c` command compiles the source code into Xilinx object (XO) files. Multiple kernels are compiled into separate XO files.
  - For RTL kernels, the Vivado IP packager command produces the XO file to be used for linking. Refer to [Packaging RTL Kernels](#) for more information.
  - You can also create kernel object (XO) files working directly in the Vitis™ HLS tool as described in the *Vitis HLS User Guide* ([UG1399](#)).
2. After compilation, the `v++ -l` command links one or multiple kernel objects (XO), together with the hardware platform XSA file, to produce the Xilinx binary `.xclbin` file.



**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 IDE as discussed in [Section IX: Using the Vitis IDE](#).

---

## Compiling C/C++ PL Kernels



**IMPORTANT!** Set up the command shell or window as described in [Setting Up the Vitis Environment](#) prior to running the tools.

The recommended approach for creating C++ PL kernels is using the Vitis HLS tool. The techniques of programming, simulating, and synthesizing the PL kernel objects are described in the *Vitis HLS User Guide* ([UG1399](#)). The synthesized kernel object (`.xo`) can be exported from the HLS tool and used in the `v++ --link` command as described in [Linking the Kernels](#). However, it is better to use the optimized C++ code with any required HLS pragmas to define the PL kernel and compile it using the Vitis compiler (`v++ --compile`), as described below.

To compile the PL kernel code using the Vitis compiler there are multiple `v++` options that need to be used. The following is an example command line to compile the `vadd` kernel:

```
v++ -t sw_emu --platform xilinx_u200_gen3x16_xdma_2_202110_1 -c -k vadd \
-I'./src' -o'vadd.sw_emu.xo' ./src/vadd.cpp
```

The various arguments used are described below. Note that some of the arguments are required.

- `-t <arg>`: Specifies the build target, as discussed in [Build Targets](#). Software emulation (`sw_emu`) is used as an example. Optional. The default is `hw`.
- `--platform <arg>`: Specifies the accelerator platform for the build. This is required because runtime features, and the target platform are linked as part of the FPGA binary. To compile a kernel for an embedded processor application, specify an embedded processor platform: `--platform $PLATFORM_REPO_PATHS/zcu102_base/zcu102_base.xpfm`.
- `-c`: Compile the kernel. Required. The kernel must be compiled (`-c`) and linked (`-l`) in two separate steps.
- `-k <arg>`: Name of the kernel associated with the source files.

- `-o '<output>.xo'`: Specify the shared object file output by the compiler. Optional.
- `<source_file>`: Specify source files for the kernel. Multiple source files can be specified. Required.

The above list is a sample of the extensive options available. Refer to [v++ Command](#) for details of the various command line options. Refer to [Output Directories of the v++ Command](#) to get an understanding of the location of various output files.

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 Vitis analyzer, and includes a Summary report, Kernel Estimate for timing and resource estimates, Kernel Guidance offering any suggestions for compilation, and the HLS Synthesis log from Vitis HLS. Refer to [Section VIII: Using the Vitis Analyzer](#) for additional information.

---

## Compiling AI Engine Graph Applications



**IMPORTANT!** Set up the command shell or window as described in [Setting Up the Vitis Environment](#) prior to running the tools.

As described in *Versal ACAP AI Engine Programming Environment User Guide (UG1076)* Versal AI Engine graph applications are written in C++ and compiled using the Vitis `aiecompiler` command.

An example command follows:

```
aiecompiler -target=hw --platform=xilinx_vck190_base_dfx_202220_1 \
--include=./src/graphs -include=./src/kernels ./src/datamove_app.cpp \
--constraints=./src/datamove_app.aiecst -workdir=./Work
```

The various arguments used are described below.

- `-target=hw`: Specifies the build target as either for simulation purposes (`x86`) or for running on the physical device (`hw`). The default is `hw`.
- `--platform=xilinx_vck190_base_dfx_202220_1`: Specifies the platform to use when compiling the graph application.
- `--include=<dir>`: This option can be used to include additional directories in the include path for the compiler front-end processing.
- `<source_file>`: Specify source files for the kernel. Multiple source files can be specified.
- `--constraints=<file>`: Specify one or more constraints files in JSON format
- `-workdir= ./Work`: Use this option to specify the output directory.

The default output is an archive of the AI Engine graph application called `libadaf.a`. Refer to [Compiling an AI Engine Graph Application](#) in [UG1076](#) for additional information. The archive file can be used by the `v++ --link` command to build a fixed platform of the integrated system design as described in the next section.

## Linking the Kernels



**TIP:** Set up the command shell or window as described in [Setting Up the Vitis Environment](#) prior to running the tools.

The PL kernel compilation process results in a Xilinx object (`.xo`) file whether the kernel is written in C/C++ or RTL. During the linking stage, one or more PL kernels are linked with the platform to create the FPGA binary container file (`.xclbin`). In the case of Versal® ACAP devices, the linking process also includes an AI Engine graph application (`libadaf.a`) and creates a fixed hardware platform (`.xsa`) for use by the Vitis packaging process as described in [Packaging the System](#).



**IMPORTANT!** Starting from the 2022.1 release, the `v++ --link` command generates a hardware definition file (`.xsa`) for Versal® device platforms (i.e. `vck190`, `vck5000`), for use by the `v++ --package` command to generate the `.xclbin` file. For Alveo™ data center accelerator cards, and embedded processor cards the `v++ --link` command generates an `.xclbin` file directly, though the `v++ --package` command might still be needed.

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

```
v++ -t hw_emu --platform xilinx_vck190_base_202210_1 --link vadd.xo
libadaf.a -o"binary_container_1.xsa" \
--config ./system.cfg
```

This command contains the following arguments:

- `-t <arg>`: Specifies the build target. Software emulation (`sw_emu`) is used as an example. When linking, you must use the same `-t` and `--platform` arguments as specified when the input (XO) file was compiled.
- `--platform <arg>`: Specifies the platform to link the kernels with.
- `--link`: Link the kernels and platform into a system design.
- `<input>.xo`: Input object file. Multiple object files can be specified.
- `libadaf.a`: Input AI Engine graph application.
- `-o '<output>.xsa'`: Specify the output file name. The output file in the link stage will be an `.xsa` file. The default output name is `a.xsa`.



**TIP:** For Alveo accelerator cards, or Zynq MPSoC based platforms, the output of the `link` command will be an `.xclbin` file rather than the `.xsa`.

- `--config ./system.cfg`: Specify a configuration file that is used to provide `v++` command options for a variety of uses. Refer to [v++ Command](#) 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 Vitis analyzer, and includes a Summary report, System and Platform Diagrams to illustrate the hardware design, System Estimate providing timing and resources estimates, System Guidance offering any suggestions for improving linking and the performance of the system, and the Vivado Automation Summary providing design details such as interface connections, clocks, resets, and interrupts. Refer to [Section VIII: Using the Vitis Analyzer](#) for additional information.



**TIP:** Refer to [Output Directories of the v++ Command](#) to get an understanding of the location of various output files.

Beyond simply linking the Xilinx object (XO) files, the linking process is also where important architectural details of the design are specified. 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 kernel ports to global memory or between streaming ports. 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 the [--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](#) for more information.

## Creating Multiple Instances of a Kernel

By default, the linker builds a single hardware instance from a kernel. If the host program will execute the same kernel multiple times, due to data processing requirements for instance, then it must execute the kernel on the hardware accelerator in a sequential manner. This can impact overall application performance. However, you can customize the kernel linking stage to instantiate multiple hardware compute units (CUs) from a single kernel. This can improve performance as the host program can now make multiple overlapping kernel calls, executing kernels concurrently by running separate compute units.

Multiple CUs of a kernel can be created by using the `connectivity.nk` option in the `v++` config file during linking. Edit a config file to include the needed options, and specify it in the `v++` command line with the `--config` option, as described in [v++ Command](#).

For example, for the `vadd` kernel, two hardware instances can be implemented in the config file as follows:

```
[connectivity]
#nk=<kernel_name>:<number>:<cu_name>,<cu_name>...
nk=vadd:2
```

Where:

- `<kernel_name>`: Specifies the name of the kernel to instantiate multiple times.
- `<number>`: The number of kernel instances, or CUs, to implement in hardware.
- `<cu_name>,<cu_name>...`: Specifies the instance names for the specified number of instances. This is optional, and the CU name will default to `kernel_1` when it is not specified. Notice that 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](#).

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
```

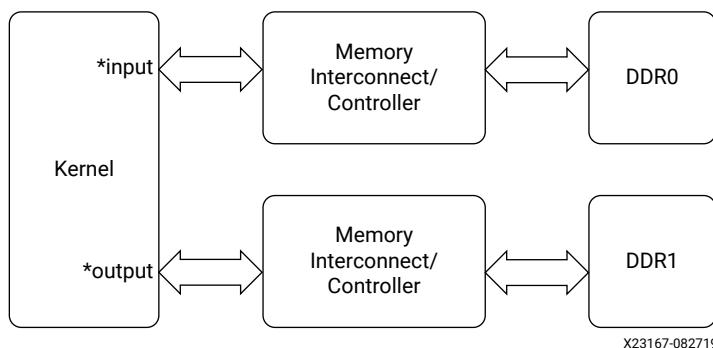
## Mapping Kernel Ports to Memory

The link phase is when the memory ports of the kernels are connected to memory resources which include DDR, HBM, and PLRAM. By default, when the `xclbin` file is produced during the `v++` linking process, all kernel memory interfaces are connected to the same global memory bank (or `gmem`). As a result, only one kernel interface can transfer data to/from the memory bank at one time, limiting the performance of the application due to memory access.

While the Vitis compiler can automatically connect CU to global memory resources, you can also manually 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 shows the [Global Memory Two Banks](#) example in [Vitis Examples](#) on GitHub. This example connects the input pointer interface of the kernel to DDR bank 0, and the output pointer interface to DDR bank 1.

Figure 20: Global Memory Two Banks Example



**IMPORTANT!** Up to 15 separate kernel interfaces can be connected to a single 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 `--connectivity.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 at [HW Interfaces](#).

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
```

Note that the memory interface inputs `pixel` and `weights` are assigned different bundle names in the example above, while `out` is bundled with `pixel`. This creates two separate interface ports: `gmem` and `gmem1`.



**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]
```

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 run time.

Some platforms also provide support for PLRAM, HBM, HP or MIG memory, in which case you would use `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](#) for more information.

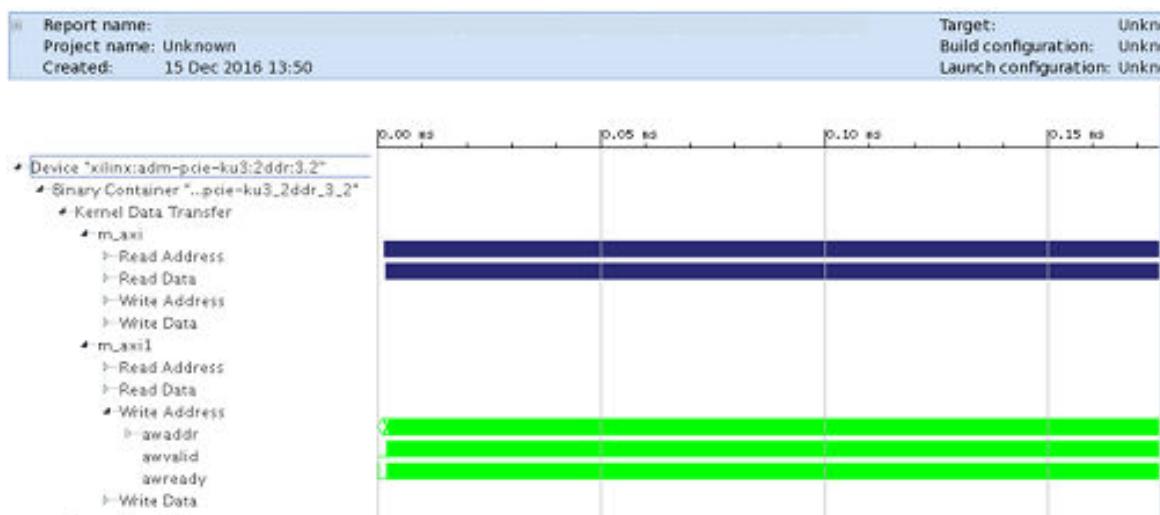
In platforms that include both DDR and HBM memory banks, kernels must use separate AXI interfaces to access the different memories. DDR and PLRAM access can be shared from a single port.



**TIP:** Assigning kernel interfaces to specific memory banks may also require you to specify the SLR to place the kernel into. For more information see [Assigning Compute Units to SLRs on Alveo Accelerator Cards](#).

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

**Figure 21: 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
```

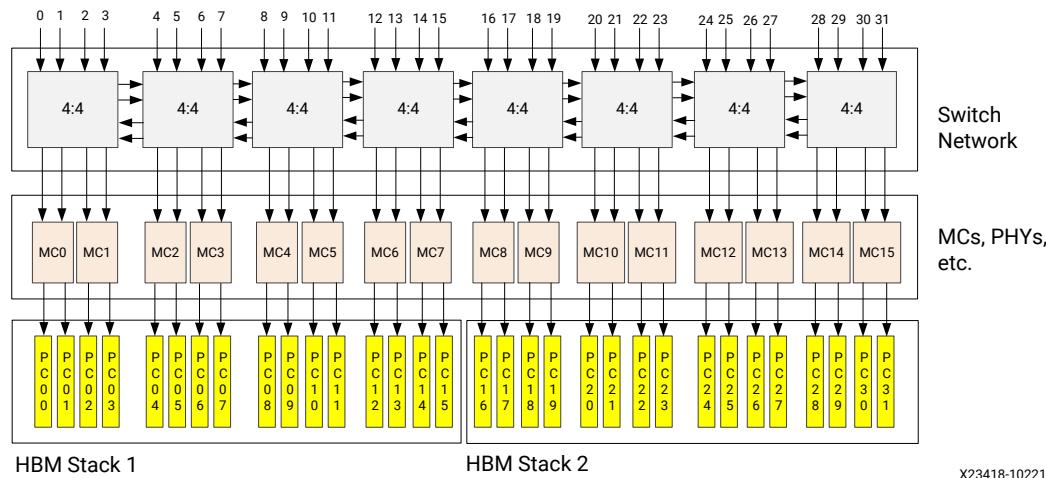


**TIP:** To reduce latency and improve performance the kernel accessing this PLRAM should be placed into SLR3 as well as described in [Assigning Compute Units to SLRs on Alveo Accelerator Cards](#).

## 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 (HBM2 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 22: High-Level Diagram of Two HBM Stacks



- 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 ( $32 \times 14.375$  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 should 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 should 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 should 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](#) 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 2 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 should access memory from multiple HBM PCs and should 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 ACAP devices, and have been used in DSP and image processing applications, as well as in 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](#).



**IMPORTANT!** An error will occur 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](#). Note that 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.



**TIP:** When specifying connections to Versal AI Engine ports, the `<cu_name>` will be `ai_engine_0` as shown above.

- `<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 will automatically insert a data-width converter between the two ports as part of the build process. The inclusion of the data-width converter will 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 best performance, you should 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 utilization values to help distribute CUs across SLRs to reduce congestion in any one SLR. The system estimate report lists the number of resources (LUTs, Flip-Flops, BRAMs, etc.) used by the kernels early in the design cycle. 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 you should 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,...

Xilinx 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.



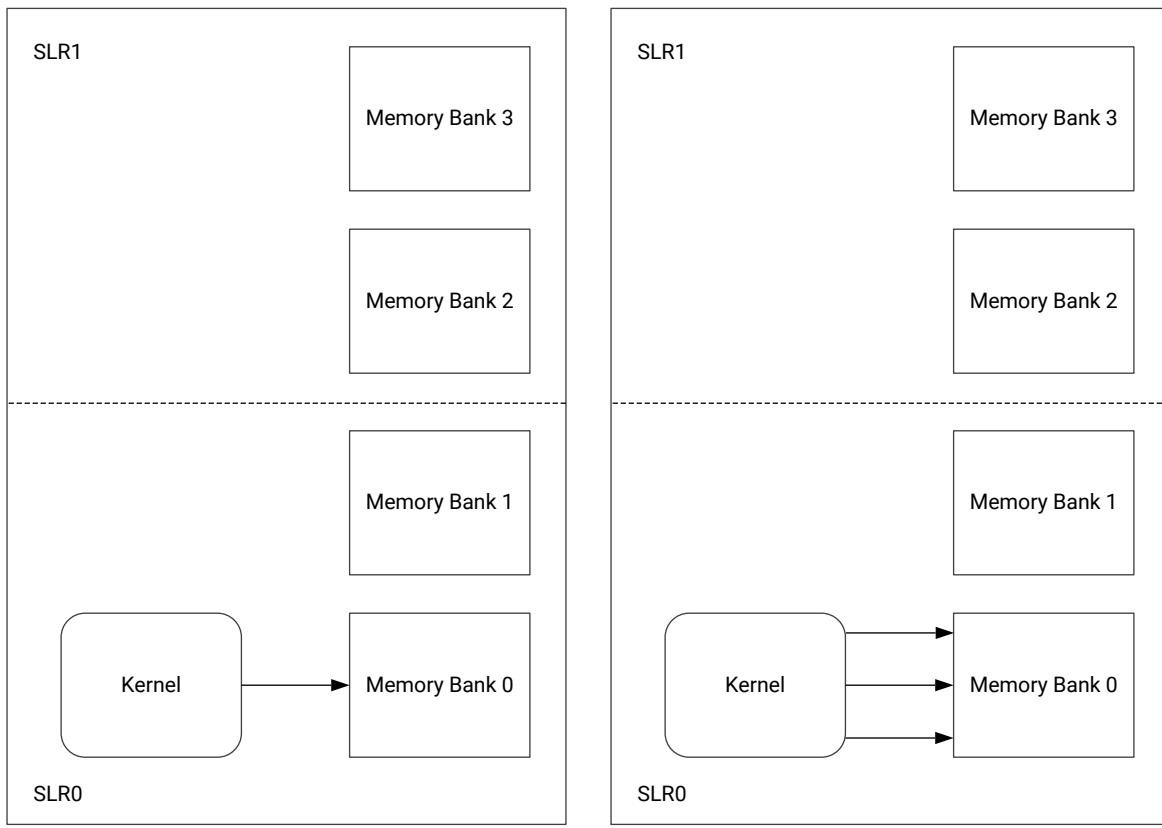
**IMPORTANT!** When building the hardware and specifying trace memory for the `profile` command, as described in [--profile Options](#), you should also be aware of CU placement and assign memory resources according to SLR use. This can improve timing by limiting and managing SLR crossings. The command 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 23: Kernel and Memory in Same SLR



X22194-010919

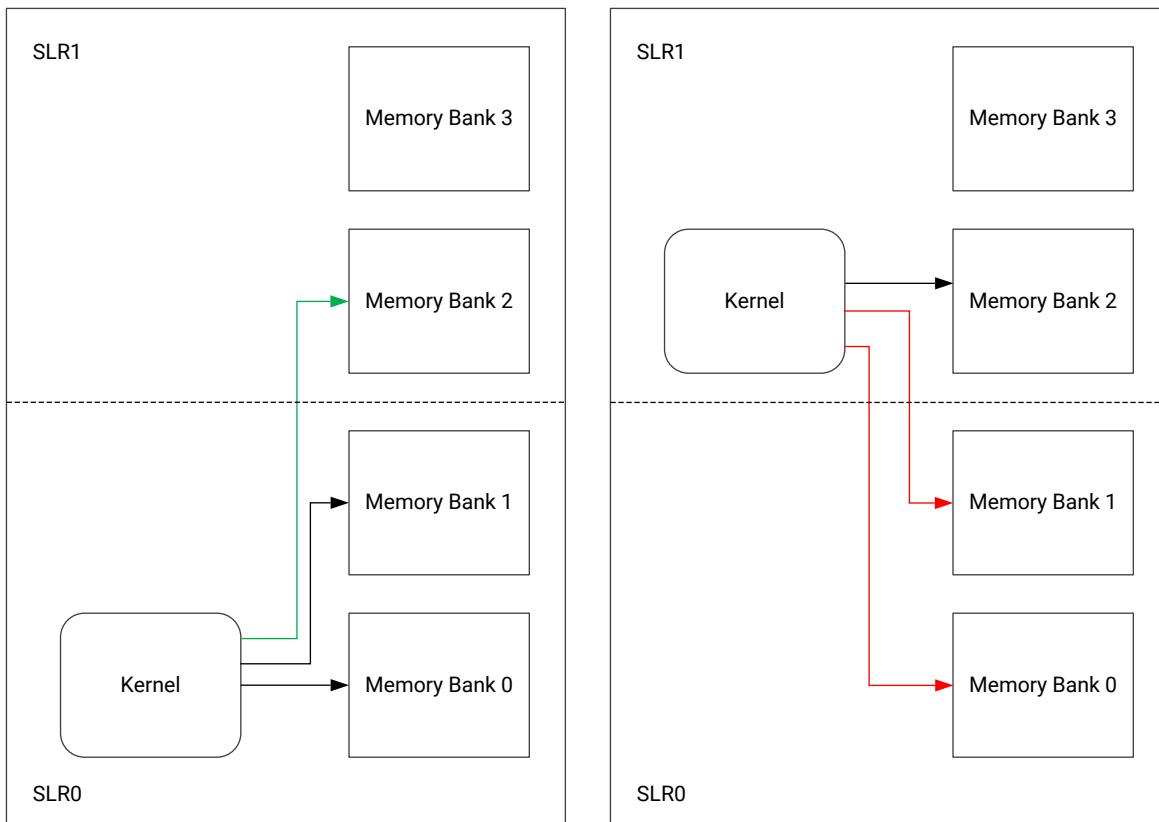
**Note:** The image on the left shows a single AXI interface mapped to a single memory bank. The image on the right shows multiple AXI interfaces mapped to the same memory bank.

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

- If the design contains a large number of kernels accessing the same memory bank.
- A kernel requires some specialized logic resources that are not available in the SLR of the memory bank.

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

Figure 24: Memory Bank in Adjoining SLR



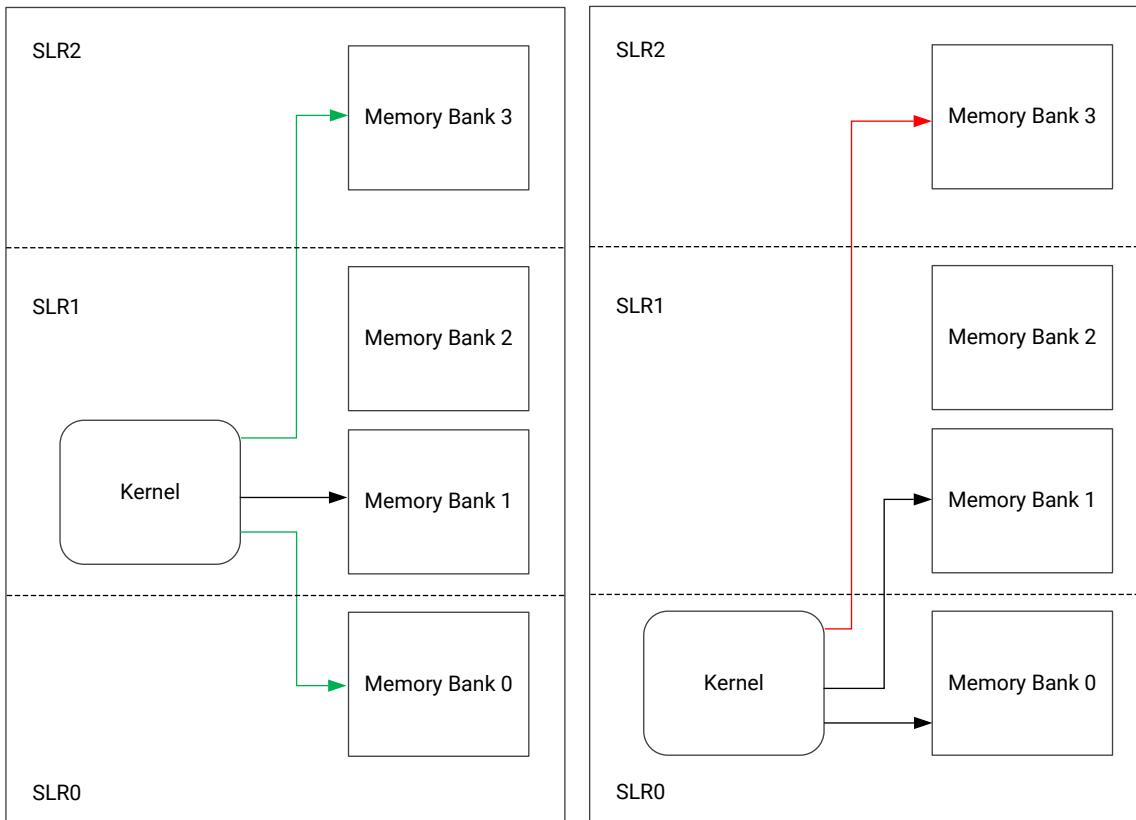
X22195-010919

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

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

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

Figure 25: Memory Banks Two SLRs Away



X22196-010919

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

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

## Managing Clock Frequencies

During the compilation process (`v++ -c`), you can specify a kernel frequency using the `--hls.clock` command. This lets you compile the kernel targeting the specified frequency, and lets the Vitis HLS tool perform validation of the kernel logic at the specified frequency. While this is just a target frequency for compilation, it does provide optimization and feedback.

To implement the kernel using a different frequency from the platform default frequency, you can use the [--clock Options](#) for platforms with fixed clocks as described in [Identifying Platform Clocks](#). When generating the device binary (.xclbin) you can connect multiple kernels to the platform using different clock frequencies. Each kernel, or unique instance (CU) of the kernel can connect to a specified clock frequency, or multiple clocks, and different kernels can use different clock frequencies.



**TIP:** For platforms with scalable clocks use the `v++ --kernel_frequency` option to specify the clock frequency for one or both clocks on the platform, as described in [v++ General Options](#).

Therefore the process for managing clock frequencies for kernels is as follows:

1. Compile the HLS code at a specified frequency using the Vitis compiler:

```
v++ -c -k <krnl_name> --hls.clock freqHz:<krnl_name>
```



**TIP:** `freqHz` must be in Hz (for example, `250000000Hz` is 250 MHz).

2. During linking, specify the clock frequency or clock ID for each clock signal in a kernel with the following command:

```
v++ -l ... --clock.freqHz <freqHz>:kernelName.clk_name
```

You can specify the `--clock` option using either a clock ID from the platform shell, or by specifying a frequency for the kernel clock. When specifying the clock ID, the kernel frequency is defined by the frequency of that clock ID on the platform. When specifying the kernel frequency, the platform attempts to create the specified frequency by scaling one of the available fixed platform clocks.

In some cases, the clock frequency can only be achieved in some approximation, and you can specify the `--clock.tolerance` or `--clock.default_tolerance` to indicate an acceptable range. If the available fixed clock cannot be scaled within the acceptable tolerance, a warning is issued and the kernel is connected to the default clock.

## ***Identifying Platform Clocks***

The handling of clocks in accelerator cards has evolved to support multiple platform clocks and clock frequencies. The kernels can have any number of independent and edge-aligned clocks, and platforms can have multiple kernels running at different clock frequencies under user-control. A platform can have scalable clocks and fixed clocks.

- **Scalable Clocks:**

An Alveo™ platform provides a frequency scalable kernel clock with ID 0 that drives all XRT managed kernels. XRT can set the clock frequency of this clock according to the meta-data contained in the `xclbin` file, when loading the file. An Alveo platform also provides a second scalable clock with ID 1 that is XRT also controllable based on `xclbin` meta-data. 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 `--kernel_frequency` command, as described in [V++ General Options](#). For example, the `v++` linker will automatically wire 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 Vitis HLS does not generate kernels with multiple clocks.

- **Fixed Clocks:**

You can generate any number of derived fixed clocks that are not controllable by XRT. The `v++` linker provides the [--clock Options](#) for users to specify which kernel clock pins should be connected to which clocks, and to define clock frequencies that are not defined on the platform. Fixed clocks are more commonly used for embedded platforms and RTL kernels.

Specifying the `--clock` options directs `v++` to use the fixed clocks of a platform, rather than the scalable clocks. For HLS kernels, the clock ID 0 is always used. For RTL kernels with one clock, clock ID 0 is used by default, but you can select a different clock.

Scalable clocks can be used to scale the clock itself whereas fixed clocks are used to add MMCMs to generate frequencies other than the frequencies defined on the platform. For example, if you choose to specify clock frequencies: 60, 200, and 450, Vitis will add all the necessary logic to generate the required clocks.



**TIP:** You cannot mix fixed and scalable clocks on a single kernel, but these can be mixed across different kernels within a single `.xclbin` file.

If Vivado place and route tools are unable to meet the frequency specification, the tools can scale the clock frequency to an achievable frequency if the scalable clock has been used (`--kernel_frequency`). If a fixed clock is used (`--clock`), then you will need to re-run the implementation to update the frequency target.

You can determine the clocks available in the target platform by using the `platforminfo` command. For example, the following command returns verbose information related to a newer shell for the U250 platform:

```
platforminfo -v -p xilinx_u250_gen3x16_xdma_3_1_202020_1 -o pfmClocks.txt
```

The information reported in the output file includes the following clock details:

```
=====
Clock Information
=====
Default Clock Index: 0
Default Clock Frequency: 300.000000
Default Clock Pretty Name: PL 0
Clock Index: 0
  Frequency: 300.000000
  Name: ss_ucs_aclk_kernel_00
  Pretty Name: PL 0
  Inst Ref: ss_ucs
  Comp Ref: shell_ucs_subsystem
  Period: 3.333333
  Normalized Period: .003333
  Status: scalable
Clock Index: 1
  Frequency: 500.000000
  Name: ss_ucs_aclk_kernel_01
  Pretty Name: PL 1
  Inst Ref: ss_ucs
  Comp Ref: shell_ucs_subsystem
  Period: 2.000000
  Normalized Period: .002000
  Status: scalable
Clock Index: 2
  Frequency: 50.000000
  Name: ii_level1_wire_ulp_m_aclk_ctrl_00
  Pretty Name: PL 2
  Inst Ref: ii_level1_wire
  Comp Ref: ii_level1_wire
  Period: 20.000000
  Normalized Period: .020000
  Status: fixed
Clock Index: 3
  Frequency: 250.000000
  Name: ii_level1_wire_ulp_m_aclk_PCIE_user_00
  Pretty Name: PL 3
  Inst Ref: ii_level1_wire
  Comp Ref: ii_level1_wire
  Period: 4.000000
  Normalized Period: .004000
  Status: fixed
Clock Index: 4
  Frequency: 100.000000
  Name: ii_level1_wire_ulp_m_aclk_freerun_ref_00
  Pretty Name: PL 4
  Inst Ref: ii_level1_wire
  Comp Ref: ii_level1_wire
  Period: 10.000000
  Normalized Period: .010000
  Status: fixed
```

The following are some examples of clock management using the [--clock Options](#) in increasing order of precedence:

- In absence of any `--clock` option, a scalable default clock will be applied. For kernels with two clocks clock ID 0 will be assigned to `ap_clk`, and clock ID 1 will be assigned to `ap_clk_2`.

- Specifying `--clock.defaultId=<id>` overrides the platform default. The specified clock `<id>` will be used as a reference clock for all pins on all CUs, unless additional `--clock` options are specified.
- Specifying `--clock.defaultFreq=<Hz>` overrides platform default with a clock of the specified frequency. The default clock will be assigned to all pins on all CUs, unless additional `--clock` options are specified.
- Specifying `--clock.id=<id>:<cu>` assigns the specified ID to all clock pins on the specified CU.
- Specifying `--clock.id=<id>:<cu>.<clk0>` assigns the specified ID to the specified clock pin on the specified CU.

There are a couple of high-level considerations in practice. Scalable clocks help you to achieve timing without having to regenerate the `.xclbin` file, by allowing the tool to scale the clock as needed to achieve a specific frequency or to meet timing.

Designs requiring different kernels to run at different clock rates, e.g., to close timing or to meet performance targets require the `--clock` directive that targets fixed clocks. Similarly, RTL kernels requiring multiple clocks, in general, require the `--clock` directives. For most serious designs, the expectation is that the user will at some point know what their achievable frequency targets are, and will need to specify these (fixed clocks) when they exceed the scalable clock frequencies (scaling will not increase a scaled clock frequency above the platform default).

---

## Managing Vivado Synthesis and Implementation Results



**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](#)), or the Versal ACAP Design Guide ([UG1273](#)).

---

In most cases, the Vitis environment completely abstracts away the underlying process of synthesis and implementation of the programmable logic region, as the CUs are linked with the hardware platform and the FPGA binary (`xclbin`) is generated. 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 tool automates much of the FPGA implementation process.

However, in some cases you might want to exercise some control over the synthesis and implementation processes deployed by the Vitis compiler, especially when large designs are being implemented. Towards this end, the Vitis tool offers some control through specific options that can be specified in a `v++` configuration file, or from the command line. The following are some of the methods you can interact with and 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](#) 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, `writeIntermediateCheckpoints`, 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-temp` 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_onViolation}
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_onViolation}
```

- **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\_onViolation}**: Specifies various options of the `report_*` Tcl command to be used when generating the report. In this example, the `-warn_onViolation` 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.

## Associating an ELF file with MicroBlaze

You can use the following steps to associate an ELF file with a MicroBlaze processor in your design. Associating the ELF file configures a memory target, such as a set of BRAMs. The information that is needed for ELF file association includes:

- The location of the ELF file to be loaded
- The address space accessible via a master interface to a memory location, which the ELF file will be stored
- The mapped peripheral within that address space representing the memory where ELF file will be stored and from where it will be accessed at run time



**IMPORTANT!** This flow requires you to have access to the level of design hierarchy containing the MicroBlaze processor, and an existing ELF file.

This process uses SCOPED\_TO\_REF and SCOPED\_TO\_CELLS properties on the MicroBlaze processor itself, and not to the BRAMs that are the actual target of the ELF file data.

You can associate the ELF file to the MicroBlaze processor during the `v++ --link` process using the `--advanced.param <param_name>=<param_value>` command as described at [--advanced Options](#). An example for a config file is shown below.

```
[advanced]
param=hw_emu.post_sim_settings=<file_path>/link.tcl
```

The `link.tcl` should add the ELF file to the Vivado Design Suite project, exclude it for use in simulation, and associate it with the MicroBlaze processor, as shown in the example below:

```
add_files <file_path>/executable.elf
set_property used_in_simulation 0 [get_files <file_path>/executable.elf]
set_property SCOPED_TO_REF base_microblaze_design [get_files -all \
-of_objects [get_fileset sources_1] {<file_path>/executable.elf}]
set_property SCOPED_TO_CELLS { microblaze_0 } \
[get_files -all -of_objects [get_fileset sources_1] {<file_path>/
executable.elf}]
```

This information will be used to generate a BMM file which will be used by programs such as `data2mem` to generate a `.mem` file that will populate the BRAMs that are generated from the `block_memory_generator`.

## 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/2021.1/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 complete and pick the best results from the completed runs before proceeding. This would require the use of the `--advanced.compiler` directive as shown:

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

`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.

As discussed in [Link Summary: Multiple Strategies and Timing Reports](#), Vitis analyzer displays the implementation results for the all strategies that have been allowed to run to completion. This includes an overview of the implementation results, as well as a Timing Summary report. 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 should resume 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++ General Options](#).

---

 **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

For example, 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 --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.

---

# 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](#).



**TIP:** This option is useful for the software emulation build (-t sw\_emu).

# Packaging the System

The `v++ --package` command will generate the device binary (`.xclbin`) for Versal® platform designs, and package the required files for all platforms to boot and run the accelerated application for software or hardware emulation, or to run on the hardware device. The `v++ --package` step, or `-p` option, packages the final product at the end of the `v++` compile and link process. As described in [Packaging for Embedded Platforms](#), this is a required step for all Versal platforms, including AI Engine platforms, and embedded processor platforms.

The [--package Options](#) let you package your design and define various files required for booting and configuring the Xilinx® device for use during emulation or in production systems. It collects the various elements to create an SD card, or other means to program the device, to define the operating system, and to load the application and kernel code.



**TIP:** After packaging the design the Vitis compiler generates a `v++ . package_summary` that includes the packaging command and log file. The summary file can be viewed in Vitis analyzer alongside the compile, link, and run summaries as explained in [Section VIII: Using the Vitis Analyzer](#).

## Packaging for Embedded Platforms

For Zynq® UltraScale+™ MPSoC and Zynq®-7000 embedded platforms, the `--package` command line is shown below:

```
v++ --package -t [sw_emu | hw_emu | hw] --platform <platform> input.xclbin  
[ -o output.xclbin ]
```

For Versal® platforms, the `--package` command line is as follows:

```
v++ --package -t [sw_emu | hw_emu | hw] --platform <platform> input.xsa [ -  
o output.xclbin ]
```

**Note:** If the output option (`-o`) is not specified, the tool creates an output file with the default name of `a.xclbin`.

In the case of Versal® platforms, the package process takes the `.xsa` file generated during the `v++ --link` command, and also takes the `libadf.a` file produced by the `aiecompiler` command and integrates it into the output device binary. For more information, refer to the [AI Engine Tools and Flows User Guide \(UG1076\)](#).

The `--package` command has a range of options for use with the different platforms and build targets supported by the Vitis tools. In the Vitis IDE, the package process is automated and the tool creates the required files as needed. However, in the command line flow, you must specify the `v++ --package` command or add the `[package]` tag in the `config` file with the right options for the job. The following is an example command for hardware emulation that runs the package process for a ZCU104 based application:

```
v++ --package -t hw_emu --platform xilinx_zcu104_base_202010_1 --save-temp  
\  
.input.xclbin ./output.xclbin --config package.cfg
```

Where, the `--config package.cfg` option specifies a configuration file for the Vitis compiler with the various options specified for the package process. The following is the content of an example configuration file:

```
[package]  
out_dir=sd_card  
boot_mode=sd  
image_format=ext4  
rootfs=/tmp/platforms/sw/zynqmp/xilinx-zynqmp-common-v2022.2/rootfs.ext4  
sd_file=/tmp/platforms/sw/zynqmp/xilinx-zynqmp-common-v2022.2/Image  
sd_file=host.elf  
sd_file=output.xclbin  
sd_file=xrt.ini  
sd_file=launch_app.sh
```

For software and hardware emulation, the command takes the `.xclbin` file as input, produces a script to launch emulation (`launch_sw_emu.sh` or `launch_hw_emu.sh`), and writes needed support files to a specified output folder, `--package.out_dir`.

Additional files required for running the application, such as data files needed as input or to validate the application, or the `xrt.ini` file for profiling and debug, must be included in the output files, and can be transferred individually using the `sd_file` option, or transferred as a directory using the `sd_dir` option as explained in [--package Options](#).

For hardware builds, the `--package` command creates an `sd_card` folder, or the `QSPI.img` depending on the boot mode specified with the `--package.boot_mode` option.



**TIP:** For bare metal ELF files running on PS cores, you should also add the following option to the command line:

```
--package.ps_elf <elf>,<core>
```

The package command creates an output folder called `sd_card`, that contains all of the files needed to run hardware emulation for the application, modeling the boot process of an `sd_card`. For hardware builds, it contains the files required for creating an SD card to boot the device.

After creating the `sd_card` folder, copy the contents to an SD card to create the boot image.

**Note:** On Windows OS you must use a third-party tool, such as Etcher, to write on the SD card for use in booting the Xilinx device.

After the package process completes you can use the Vitis analyzer tool to visualize and navigate the relevant reports or log files by running the following command:

```
vitis_analyzer ./<output>.package_summary
```

---

## Packaging for Data Center Platforms



**TIP:** The `--package` command is only required for Data Center accelerator cards incorporating the Versal portfolio of devices.

For Versal® Data Center accelerator cards, the `--package` command line is as follows:

```
v++ --package -t [ sw_emu | hw_emu | hw ] --platform <platform> libadf.a  
input.xsa [ -o output.xclbin ]
```

If the output option (`-o`) is not specified, the tool creates an output file with the default name of `a.xclbin`.

For and Data Center platforms, packaging is not required.

# Running the Application

Running the application hardware build allows you to see your application running on an accelerator card, such as the Alveo™ Data Center accelerator card, or an embedded processor platform targeting a Versal® ACAP or Zynq UltraScale+ MPSoC devices. 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. For embedded platforms, boot the SD card.



**TIP:** This step is only required for platforms using Xilinx embedded devices such as Versal ACAP or Zynq UltraScale+ MPSoC.

For an embedded processor platform, copy the contents of the `./sd_card` folder produced by the `v++ --package` command to an SD card as the boot device for your system. Boot your system from the SD card.

4. Run your application.

The specific command line to run the application will depend on your host code. A common implementation used in Xilinx 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 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 [Section VIII: Using the Vitis Analyzer](#) for additional information.

# Simulating the Application with the Emulation Flow

Development of a user application and hardware kernels targeting an FPGA requires a phased development approach. Because FPGA, Versal® ACAP, and Zynq UltraScale+ MPSoC 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 Vitis™ tool provides software emulation targets to perform C-based simulation of the design, and hardware emulation targets to perform C-RTL co-simulation of the software application and PL kernels. Compiling for emulation targets is significantly faster than compiling for the actual hardware. Additionally, emulation targets provide full visibility into the application or accelerator, thus making it easier to perform debugging. Once your design passes in emulation, then in the late stages of development you can compile and run the application on the hardware platform.

The Vitis tool provides two emulation targets:

- **Software emulation (sw\_emu):** The software emulation build compiles and links quickly, and the host program runs either natively on an x86 processor or in the QEMU emulation environment. The PL kernels are natively compiled and running on the host machine. This build target lets you quickly iterate on both the host code and kernel logic.
- **Hardware emulation (hw\_emu):** The host program runs in sw\_emu, natively on x86 or in the QEMU, but the kernel code is compiled into an RTL behavioral model which is run in the 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 either of the emulation targets is seamlessly integrated into the Vitis command line and IDE flows. You can compile your host and kernel source code for either 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 or PS application ELF binary can be used in emulation. Emulation targets 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 both the software and hardware emulation flows.

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 [Section VIII: Using the Vitis Analyzer](#) for additional information.

SW Emulation is an abstract model and does not use any of the petalinux drivers like such as Zynq OpenCL (ZOCL), interrupt controller, or Device Tree Binary (DTB). Hence, the overhead of creating `sd_card.img` and booting petalinux on full QEMU machines can be avoided for SW Emulation. This enables faster SW\_EMU as QEMU is slow and requires petalinux. Thus, for this approach the user is not required to provide fields such as `sysroot`, `rootfs` and `sd_Card Image`.

**Note:** If users are sourcing the environment setup script "xilinx-versal-common-v2022.2/environment-setup-cortexa72-cortexa53-xilinx-linux", they may find a warning or error to unset the `LD_LIBRARY_PATH` (if already set) in order to execute the embedded XRT. The environment setup script sets up the arm gcc tool chain path along with the required additional environment variables.

Installing the x86 XRT automatically sets the `LD_LIBRARY_PATH` variable to point to XRT libraries. For running both the embedded XRT and x86 XRT on the same setup (terminal), you must specify `arm-gcc` and `SYSROOT` paths for embedded systems.

This section contains the following chapters:

- [Running Emulation Targets](#)
- [Data Center vs. Embedded Platforms](#)
- [QEMU](#)
- [Running Emulation on Data Center Accelerator Cards](#)
- [Running Emulation on an Embedded Processor Platform](#)
- [Speed and Accuracy of Hardware Emulation](#)
- [Working with Simulators in Hardware Emulation](#)
- [Working with Functional Model of the HLS Kernel](#)
- [Working with SystemC Models](#)
- [Debug Techniques in Hardware Emulation](#)
- [Working with I/O Traffic Generators](#)

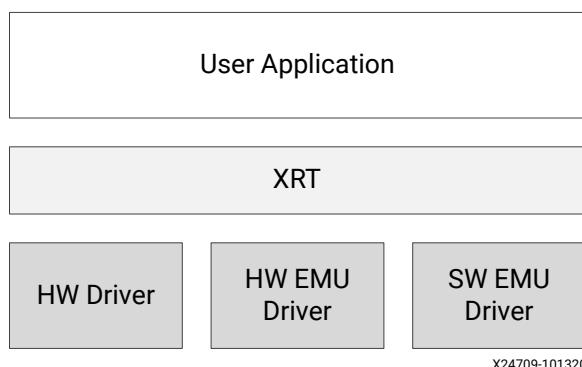
# Running Emulation Targets

The emulation targets have their own target specific drivers which are loaded by XRT. Thus, the same CPU binary can be run as-is without recompiling, by just changing the target mode during runtime. Based on the value of the `XCL_EMULATION_MODE` environment variable, XRT loads the target specific driver and makes the application interface with an emulation model of the hardware. The allowed values of `XCL_EMULATION_MODE` are `sw_emu` and `hw_emu`. If `XCL_EMULATION_MODE` is not set, then XRT will load the hardware driver.



**IMPORTANT!** It is required to set `XCL_EMULATION_MODE` when running emulation.

Figure 26: XRT Drivers



You can also use the `xrt.ini` file to configure various options applicable to emulation. There is an `[Emulation]` specific section in `xrt.ini`, as described in [xrt.ini File](#).

# Data Center vs. Embedded Platforms

Emulation is supported for both data center and embedded platforms. For data center platforms, the host application is compiled for x86 server, while the device is modeled as separate x86 process emulating the hardware. The user host code and the device model process communicate using RPC calls. For embedded platforms, where the CPU code is running on the embedded Arm processor, emulation flows use QEMU (Quick Emulator) to mimic the Arm-based PS-subsystem. In QEMU, you can boot embedded Linux and run Arm binaries on the emulation targets.

For running software emulation (`sw_emu`) and hardware emulation (`hw_emu`) of a data center application, you must compile an emulation model of the accelerator card using the `emconfigutil` command and set the `XCL_EMULATION_MODE` environment variable prior to launching your application. The steps are detailed in [Running Emulation on Data Center Accelerator Cards](#).

For running `sw_emu` or `hw_emu` of an embedded application, you will compile the application for an Arm processor using Arm-GCC and launch the QEMU emulation environment on an x86 processor to model the execution environment of the Arm processor. This requires the use of the `launch_emulator.py` command, or `launch_emulator.sh` shell scripts generated during the build process. The details of this flow are explained in [Running Emulation on an Embedded Processor Platform](#).



**TIP:** You can also compile and run the simulation for an embedded processor application directly on the x86 processor as described in [Running Emulation on an Embedded Processor Platform Using PS on x86](#). This compiles using x86 GCC and does not require QEMU.

# QEMU

QEMU stands for Quick Emulator. It is a generic and open source machine emulator. Xilinx provides a customized QEMU model that mimics the Arm based processing system present on Versal ACAP, Zynq® UltraScale+™ MPSoC, and Zynq-7000 SoC. The QEMU model provides the ability to execute CPU instructions at almost real time without the need for real hardware. For more information, refer to the [Xilinx Quick Emulator User Guide: QEMU](#).

For hardware emulation, the Vitis emulation targets use QEMU and co-simulate it with an RTL and SystemC-based model for rest of the design to provide a complete execution model of the entire platform. You can boot an embedded Linux kernel on it and run the XRT-based accelerator application. Because QEMU can execute the Arm instructions, you can take the Arm binaries and run them in emulation flows as-is without the need to recompile. QEMU also allows you to debug your application using GDB and TCF-based target connections from Xilinx System Debugger (XSDB).

The Vitis emulation flow also uses QEMU to emulate the MicroBlaze™ processor to model the platform management modules (PLM and PMU) of the devices. On Versal devices, the PLM firmware is used to load the PDI to program sections of the PS and AI Engine model.

To ensure that the QEMU configuration matches the platform, there are additional files that must be provided as part of `sw` directory of Vitis platforms. Two common files, `qemu_args.txt` and `pmc_args.txt`, contain the command line arguments to be used when launching QEMU. When you create a custom platform, these two files are automatically added to your platform with default contents. You can review the files and edit as needed to model your custom platform. Refer to a Xilinx embedded platform for an example.

Because QEMU is a generic model, it uses a Linux device tree style DTB formatted file to enable and configure various hardware modules. A default QEMU hardware DTB file is shipped with the Vitis tools in the `<vitis_installation>/data/emulation/dtbs` folder. However, if your platform requires a different QEMU DTB, you can package it as part of your platform.



---

**TIP:** The QEMU DTB represents the hardware configuration for QEMU, and is different from the DTB used by the Linux kernel.

---

# Running Emulation on Data Center Accelerator Cards



**TIP:** Set up the command shell or window as described in [Setting Up the Vitis Environment](#) 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. Create an `emconfig.json` file from the target platform as described in [emconfigutil Utility](#). This is required for running hardware or software 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 `sw_emu` (software emulation) or `hw_emu` (hardware emulation) as appropriate. 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 sw_emu
```

Bash shell:

```
export XCL_EMULATION_MODE=sw_emu
```

---

 **IMPORTANT!** *The emulation targets will not run if the XCL\_EMULATION\_MODE environment variable is not properly set.*

---

4. Run the application.

With the runtime initialization 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 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.*

---

# Running Emulation on an Embedded Processor Platform



**RECOMMENDED:** The file size limit on your machine should either be set to unlimited or a higher value (over 16 GB) because embedded HW Emulation can create files with larger file size for memory.



**TIP:** Set up the command shell or window as described in [Setting Up the Vitis Environment](#) prior to running the builds.

1. Set the desired runtime settings in the `xrt.ini` file.

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. As described in [Enabling Profiling in Your Application](#) this enables the runtime to capture debugging and profile data as your application is running.

The `xrt.ini` file, as well as any additional files required for running the application, must be included in the output files as explained in [Packaging for Embedded Platforms](#).



**TIP:** Be sure to use the `v++ -g` option when compiling your kernel code for emulation mode.

2. Launch the QEMU emulation environment by running the `launch_sw_emu.sh` script or `launch_hw_emu.sh` script.

```
launch_sw_emu.sh -forward-port 1440 22
```

The script is created in the emulation directory during the packaging process, and uses the `launch_emulator.py` command to setup and launch QEMU. When launching the emulation script you can also specify options for the `launch_emulator.py` command. Such as the `-forward-port` option to forward the QEMU port to an open port on the local system. This is needed when trying to copy files from QEMU as discussed in Step 5 below. Refer to [launch\\_emulator Utility](#) for details of the command.

Another example would be to specify `launch_hw_emu.sh -enable-debug` to configure additional XTERMs to be opened for QEMU and PL processes to observe live transcripts of command execution to aid in debugging the application. This is not enabled by default, but can be useful when needed for debug.

3. Mount and configure the QEMU shell with the required settings.

The Xilinx embedded base platforms have `rootfs` on a separate EXT4 partition on the SD card. After booting Linux, this partition needs to be mounted. If you are running emulation manually, you need to run the following commands from the QEMU shell:

```
mount /dev/mmcblk0p1 /mnt
cd /mnt
export LD_LIBRARY_PATH=/mnt:/tmp:$LD_LIBRARY_PATH
export XCL_EMULATION_MODE=hw_emu
export XILINX_XRT=/usr
export XILINX_VITIS=/mnt
```



**TIP:** You can set the `XCL_EMULATION_MODE` environment variable to `sw_emu` for software emulation, or `hw_emu` for hardware emulation. This configures the host application to run in emulation mode.

4. Run the application from within the QEMU shell.

With the runtime initialization (`xrt.ini`), the `XCL_EMULATION_MODE` environment set, run the host executable with the command line as required by the host application. For example:

```
./host.elf kernel.xclbin
```



**TIP:** This command line assumes that the host program is written to take the name of the `xclbin` file as an argument, as most Vitis examples and tutorials do. However, your application can have the name of the `xclbin` file hard-coded into the host program, or can require a different approach to running the application.

5. After the application run has completed, you might have some files that were produced by the runtime, such as `opencl_summary.csv`, `opencl_trace.csv`, and `xrt.run_summary`. These files can be found in the `/mnt` folder inside the QEMU environment. However, to view these files you must copy them from the QEMU Linux system back to your local system. The files can be copied using the `scp` command as follows:

```
scp -P 1440 root@<root-login-ip-address>:/mnt/<file> <dest_path>
```

Where:

- 1440 is the QEMU port to connect to.
- `root@<root-login-ip-address>` is the root login for the PetaLinux running under QEMU on the specified IP address. The default root password is "root".

**Note:** The ip address of the QEMU system can be found with a Linux command such as `hostname -I | awk '{print $1}'`

- `/mnt/<file>` is the path and file name of the file you want to copy from the QEMU environment.
- `<dest_path>` specifies the path and file name to copy the file to on the local system.

For example:

```
scp -P 1440 root@172.55.12.26:/mnt/xrt.run_summary
```

6. When your application has completed emulation and you have copied any needed files, click **Ctrl + a + x** keys to terminate the QEMU shell and return to the Linux shell.

**Note:** If you have trouble terminating the QEMU environment, you can kill the processes it launches to run the environment. The tool reports the process IDs (pids) at the start of the transcript, or you can specify the `-pid-file` option to capture the pids when launching emulation.

---

## Running Emulation on an Embedded Processor Platform Using PS on x86

Running emulation under the QEMU environment for embedded processors is compute-intensive and requires additional setup and configuration. You must cross-compile the embedded application using `arm-gcc`, create an SD card image, and boot Linux under the QEMU environment before running the application. The PS on x86 feature lets you simulate your embedded system design on an x86 processor with much less effort. This feature requires you to compile the PS application using an x86 version of `gcc` or `g++` compiler, and to build your `.xclbin` file to run on an emulation platform for software emulation created by `emconfigutil`.



**IMPORTANT!** PS on x86 compilation is only valid for software emulation and requires GCC 8.3 or later.  
This feature also requires the x86 installation of XRT as described in [Installing Xilinx Runtime and Platforms](#).

The process for building and running software emulation using PS on x86 is as follows:

1. Compile the PS application using the standard `gcc` compiler rather than the Arm cross-compiler. This is described in [Compiling and Linking for x86](#).
2. Compile and link the device binary (`.xclbin`) using standard `v++` commands as described in [Building the Device Binary](#).
3. For the PS on x86 flow the `v++ --package` command is only needed for Versal platforms to convert the `.xsa` produced by the `v++ --link` command into an `.xclbin` file. This step is not required for Zynq MPSoC based platforms.
4. Use the [emconfigutil Utility](#) to create an `emconfig.json` file for software emulation.

```
emconfigutil --platform xilinx_zcu102_base_202220_1
```

5. Set the `XCL_EMULATION_MODE` environment variable to `sw_emu` for software emulation mode.



**IMPORTANT!** The emulation targets will not run if the `XCL_EMULATION_MODE` environment variable is not properly set.

6. Run the application. For example:

```
./host.exe kernel.xclbin
```

The following table describes the differences in building the PS application and device binary (.xclbin) for running software emulation under QEMU and for running PS on x86.

**Table 15: Running SW Emulation for PS on X86**

Process	Running SW_EMU under QEMU	SW_EMU with PS on X86
Host compilation	Requires cross-compilation using arm-gcc and requires SYSROOT definition, and embedded include and library files:  <pre>aarch64-linux-gnu-g++-o hello_world host.cpp -lxrt_coreutil -pthread --sysroot=\${EDGE_COMMON_SW}/sysroots/cortexa72-cortexa53-xilinx-linux-\$({EDGE_COMMON_SW})/sysroots/cortexa72-cortexa53-xilinx-linux/usr/include/xrt -L\$({EDGE_COMMON_SW})/sysroots/cortexa72-cortexa53-xilinx-linux/usr/lib</pre>	Requires x86 gcc and requires x86 installation of XRT:  <pre>g++ -o hello_world host.cpp -lxrt_coreutil -pthread -I\${XILINX_XRT}/include -L\${XILINX_XRT}/lib</pre>
Package	Requires SD card image creation to run under QEMU:  <pre>v++ -p -t sw_emu \${LINK_XCLBIN} \${LIBADDF} --platform \${PLATFORM_PATH} --package.out_dir ./ package.hw_emu --package.rootfs \${EDGE_COMMON_SW}/rootfs.ext4 -- package.kernel_image \${EDGE_COMMON_SW}/Image --package.sd_file xrt.ini --package.sd_file ./run_app.sh -o vadd.xclbin --package.sd_file ./hello_world</pre>	For PS on x86 simulation the --package command is only needed to generate the .xclbin file:  <pre>v++ -p \${LINK_OUTPUT} \${VPP_FLAGS} --package.out_dir \${PACKAGE_OUT} -o \${BUILD_DIR}/kernel.xclbin</pre>
Running Simulation	Requires use of launch_emulation shell script to setup QEMU and run the simulation:  <pre>./launch_sw_emu.sh -run-app \${RUN_APP_SCRIPT}   tee run_app.log;</pre>	Set emulation mode environment variable and launch application directly:  <pre>XCL_EMULATION_MODE=sw_emu ./\$(EXECUTABLE) \$(CMD_ARGS)</pre>

You can of course compile the same PS code using arm-gcc in x86 gcc. However, any ARM-only data types or libraries that are linked in your PS code will not work when compiled for x86. Refer to [ps\\_on\\_x86](#) on GitHub for an example of running software emulation using this feature.



**TIP:** In order to compile and run AI Engine applications for PS on x86, refer to Versal ACAP AI Engine Programming Environment User Guide ([UG1076](#)) for additional information.

# 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 a mix of purely functional models and performance approximate models. Hardware emulation does not mimic hardware accuracy 100%, therefore you should 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 16: 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.
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.
AI Engine	The AI Engine SystemC model is cycle approximate, though it is not intended to be 100% cycle accurate. A common model is used between AI Engine Simulator and hardware emulation, thus enabling a reasonable comparison between the two stages.
Versal NoC and DDR Models	The Versal NoC and DDR SystemC models are cycle approximate.

**Table 16: Hardware Platform (cont'd)**

Hardware Functionality	Description
Arm Processing Subsystem (PS, CIPS)	The Arm PS is modeled using QEMU, which is a purely functional execution model. For more information, see <a href="#">QEMU</a> .
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.
Other I/O Models	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 which mimics the I/O in your design. 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. Xilinx recommends using small data buffers. For example, if you have a configurable vector addition and in hardware you are performing a 1024 element `v.add`, 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.

# Working with Simulators in Hardware Emulation

## Simulator Support

The Vitis tool uses the Vivado logic simulator (`xsim`) as the default simulator for all platforms, including Alveo Data Center accelerator cards, and Versal and Zynq UltraScale+ MPSoC embedded platforms. However, for Versal embedded platforms, like `xilinx_vck190_base` or custom platforms similar to it, the Vitis tool also supports the use of third-party simulators for hardware emulation: Mentor Graphics Questa Advanced Simulator, Xcelium, and VCS. The specific versions of the supported simulators are the same as the versions supported by Vivado Design Suite.



**TIP:** 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 to be implemented during generation of the device binary (`.xclbin`) and supporting Tcl scripts. The specific requirements for each simulator is discussed below. Also, note that you should 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:

```
## Final set of additional options required for running simulation using
Questa Simulator
[advanced]
param=hw_emu.simulator=QUESTA
[vivado]
prop=project...CURRENT...simulator.questa_install_dir=<Questa_install_dir>
prop=project...CURRENT...compxlib.questa_pre-
compiled_library_dir=<Questa_compiled_lib_path>
prop=fileset.sim_1.questa.compile.sccom.cores={16}
```

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=filesset.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=filesset.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
```

You can use the `-user-pre-sim-script` and `-user-post-sim-script` options from the `launch_emulator.py` command to specify Tcl scripts to run before the start of simulation, or after simulation completes. As an example, in these scripts, you can use the `$cwd` command to get the run directory of the simulator and copy any files needed prior to simulation, or copy any output files generated at the end of simulation.

To enable hardware emulation, you must set up the environment for simulation in the Vivado Design Suite. A key step for setup is pre-compiling the RTL and SystemC models for use with the simulator. To do this, you must run the `compile_sim_lib` command in the Vivado tool. For more information on pre-compilation of simulation models, refer to the *Vivado Design Suite User Guide: Logic Simulation (UG900)*.

When creating your Versal platform ready for simulation, the Vivado tool generates a simulation wrapper which must be instantiated in your simulation test bench. So, if the top most design module is `<top>`, then when calling `launch_simulation` in the Vivado tool, it will generate a `<top>_sim_wrapper` module, and also generates `xlnoc.bd`. These files are generated as simulation-only sources and will be overwritten whenever `launch_simulation` is called in the Vivado tool. Platform developers need to instantiate this module in the test bench and not their own `<top>` module.

---

## 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](#), or refer to [Vivado Design Suite User Guide: Logic Simulation \(UG900\)](#).

---

## Generating Test Vectors for Vitis HLS during Hardware Emulation

Now it is possible to instruct the 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> ...
```

Note that 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.

# Working with Functional Model of the HLS Kernel

Using a functional model of the Vitis HLS kernel during hardware emulation is an advanced use case that enables compilation of kernels in functional mode that generates the XO with the SystemC wrapper around the C code.

HW Emulation is mainly targeted for hardware kernel debug with detailed, cycle-accurate view of kernel activity. The functional (TLM) model speeds up emulation by compiling the kernel of interest in functional mode rather than as RTL code. This provides faster compile time for the kernel as it does not need full C to RTL synthesis, and faster execution time as C-code is simulated instead of RTL simulation. You can also mix and match of C and RTL kernels in hardware emulation for faster debug of RTL blocks.

The functional model feature supports modeling AXI4-Stream interfaces (`axis`) and AXI4 memory-mapped interfaces (`m_axi`), as well as register reads and writes of the AXI4-Lite (`s_axilite`) interfaces. However, with this approach, the kernel will be purely functional without latency information, unlike cycle-accurate models.

The user HLS function is wrapped into a SystemC module with TLM interfaces and IP is created out of the generated code which will allow generating HW\_EMU compatible XO that can be used in IP integrator for stitching v++ link designs in HW Emulation flows. This also allows the Wrapper IP to talk to other RTL and SystemC models. So, the HLS C/C++ kernels compiled in functional mode will have TLM transactions during simulation and users can see traffic between the memory models (e.g. DDR) and the TLM kernels.



**TIP:** The functional model uses C-code performing C-simulation for the kernel. The kernel will be purely functional without any latency information unlike cycle-accurate models. Although, you can see boundary transactions via TLM interfaces during HW Emulation.

## XO Generation with Functional Model

During the `v++` compile step, while creating the hardware emulation (`hw_emu`) XO files, you can provide a switch describing the intention to do a functional simulation that will generate XO with the SystemC wrapper on the C code. You need to provide an `--advanced.param` option during compilation. This can be done by adding the compiler option `--advanced.param compiler.emulationMode=func` as described in [--advanced Options](#).

The generated XO is linked using the `v++ --link` command same as the regular XO. For an example refer to [mm\\_stream\\_func\\_mode](#) on GitHub.

## Limitations of the Functional Model

1. Limitations in HLS are applied "as is". For example, HLS does not support double pointers so the functional model does not identify it.
2. HLS designs which operate on multiple data iteration from host with single kernel `ap_start` (for example `ap_ctrl_chain`) may not operate if the restart is triggered from the kernel code. Mailboxing works fine.
3. Application Binary Interface (ABI) changes for FPGA are not available in Functional Mode x86 ABI. For most optimizations where ABI is used, they need to be disabled in functional compiler.
4. Limiting DDR Analysis by Casting/Inter procedural uses:
  - a. Typecasting DDR pointers from scalars will not work.

```
kernel void vadd(size_t a_s,size_t b_s,size_t c){  
    int* a = (size_t)a;  
    int* b = (size_t)b;  
    int* c = (size_t)c;  
    for(int i=0; i < 64; i++){  
        c[i] = a[i] + b[i];  
    }  
}
```

- b. Caching DDR pointers across procedural context will not work.

```
class Cache{  
    int* local;  
    Cache(int *a) : local(a){}  
    int read(){  
    void write(int x){  
    };  
    kernel void vadd(int *a,int *b, int *c){  
        Cache ca(a);  
        for(int i=0; i < 64; i++){  
            c[i] = ca.read() + b[i];  
        }  
    }  
}
```

5. HLS Features implemented in binary and consuming DDR access are not supported and require functional rewrite.

Coding guidelines for working with functional models: For kernel compute units that run multiple times and expect static value reset to zero in each iteration you must initialize all static variables at the entry of the kernel function. The following example shows code that returns an error and also demonstrates the recommended approach:

```
// User code that errors out
static int i = 0;
void hls_kernel_logic(...) {
    ...
}

// Recommended
static int i = 0;
void hls_kernel_logic(...) {
    i = 0;
    ...
}
```

# Working with SystemC Models

SystemC models in the Vitis application acceleration development flow allow you to quickly model an RTL algorithm for rapid analysis in software and hardware emulation. Using this approach you can model portions of your system while the RTL kernel is still in development, but you want to move forward with some system analysis.

The SystemC model feature supports all the XRT-managed kernel execution models using `ap_ctrl_hs` and `ap_ctrl_chain`. It also supports modeling both AXI4 memory mapped interfaces (`m_axi`) and AXI4-Stream interfaces (`axis`), as well as register reads and write of the `s_axilite` interface.

You can model your kernel code in SystemC TLM models, provide interfaces to other kernels and the host application, and use it during emulation. You can create a Xilinx object file (XO) to link the SystemC model to other kernels in your `xclbin`. The sections that follow discuss the creation of SystemC models, the use of the `create_sc_xo` command to create the XO, and generating the `xclbin` using the `v++` command.



**TIP:** Keep in mind that the SystemC model is not cycle accurate, and therefore impacts the timing results of your emulation. It does not reflect the true bandwidth, latency, or throughput of the RTL code.

## Coding a SystemC Model

The process for defining a SystemC model uses the following steps:

1. Include header files `"xtlm_ap_ctrl.h"` and `"xtlm.h"`.
2. Derive your kernel from a predefined class based on the supported kernel types: `ap_ctrl_chain`, `ap_ctrl_hs`, etc.
3. Declare and define the AXI interfaces used on your kernel.
4. Add required kernel arguments with the correct address offset and size.
5. Write the kernel body in `main()` thread.

This process is demonstrated in the code example below.

When creating the SystemC model, you derive the kernel from a class-based on a supported control protocol: `xtlm_ap_ctrl_chain`, `xtlm_ap_ctrl_hs`, and `xtlm_ap_ctrl_none`. Use the following structure to create your SystemC model.



**TIP:** The following example is based on the simple vector addition (VADD) example design found in the [Vitis\\_Accel\\_Examples](#) on GitHub.

```
#include "xtlm.h"
#include "xtlm_ap_ctrl.h"

class vadd : public xsc::xtlm_ap_ctrl_hs
{
public:
    SC_HAS_PROCESS(vadd);
    vadd(sc_module_name name, xsc::common_cpp::properties& _properties):
        xsc::xtlm_ap_ctrl_hs(name)
    {
        DEFINE_XTLM_AXIMM_MASTER_IF(in1, 32);
        DEFINE_XTLM_AXIMM_MASTER_IF(in2, 32);
        DEFINE_XTLM_AXIMM_MASTER_IF(out_r, 32);

        ADD_MEMORY_IF_ARG(in1, 0x10, 0x8);
        ADD_MEMORY_IF_ARG(in2, 0x18, 0x8);
        ADD_MEMORY_IF_ARG(out_r, 0x20, 0x8);
        ADD_SCALAR_ARG(size, 0x28, 0x4);

        SC_THREAD(main_thread);
    }

    //! Declare aximm interfaces..
    DECLARE_XTLM_AXIMM_MASTER_IF(in1);
    DECLARE_XTLM_AXIMM_MASTER_IF(in2);
    DECLARE_XTLM_AXIMM_MASTER_IF(out_r);

    //! Declare scalar args...
    unsigned int size;

    void main_thread()
    {
        wait(ev_ap_start); //! Wait on ap_start event...

        //! Copy kernel args configured by host...
        uint64_t in1_base_addr = kernel_args[0];
        uint64_t in2_base_addr = kernel_args[1];
        uint64_t out_r_base_addr = kernel_args[2];
        size = kernel_args[3];

        unsigned data1, data2, data_r;
        for(int i = 0; i < size; i++) {
            in1->read(in1_base_addr + (i*4), (unsigned
char*)&data1); //! Read from in1 interface
            in2->read(in2_base_addr + (i*4), (unsigned
char*)&data2); //! Read from in2 interface

            //! Add data1 & data2 and write back result
            data_r = data1 + data2; //! Add
            out_r->write(out_r_base_addr + (i*4), (unsigned
char*)&data_r); //! Write the result
        }
    }
}
```

```
        ap_done(); //! completed Kernel computation...
    }
};
```

The include files are available in the Vitis installation hierarchy under the \$XILINX\_Vivado/  
data/systemc/simlibs/ folder.

The kernel name is specified when defining the class for the SystemC model, as shown above,  
inheriting from the `xtlm_ap_ctrl_hs` class.

You must declare and define the AXI interfaces associated with the kernel arguments as shown  
by the following constructs:

```
DECLARE_XTLM_AXIMM_MASTER_IF(in1);
DEFINE_XTLM_AXIMM_MASTER_IF(in1, 32);
```

The declaration associates the interface type with the argument. The definition defines the data  
width of the interface. You must also declare the register offsets and size for the kernel  
arguments as shown in the following:

```
ADD_MEMORY_IF_ARG(in1, 0x10, 0x8);
```

When specifying the kernel arguments, offsets, and size, these values should match the values  
reflected in the AXI4-Lite interface of the XRT-managed kernel as described in [SW-Controllable  
Kernels](#) and [Control Requirements for XRT-Managed Kernels](#).

The addresses below `0x10` are reserved for use by XRT for managing kernel execution. Kernel  
arguments can be specified from `0x10` onwards. Most importantly the arguments, offsets, and  
size specified in the SystemC model should match the values used in the Vitis HLS or RTL kernel.

The the kernel is executed in the SystemC `main_thread`. This thread waits until the `ap_start`  
bit is set by the host application, or XRT, at which time the kernel can process argument values as  
shown:

1. The kernel waits for a signal to begin from XRT (`ev_ap_start`).
2. Kernel arguments are mapped to variables in the SystemC model.
3. The inputs are read.
4. The vectors are added and the result is captured.
5. The output is written back to the host.
6. The finished signal is sent to XRT (`ap_done`).

## Creating the XO

To generate XO file from the SystemC model, use the `create_sc_xo` command. This takes the SystemC kernel source file as input and creates IP that generates the XO, which can be used for linking with the target platform and other kernels with the Vitis compiler. For example:

```
create_sc_xo vadd.cpp
```

Generating an XO file from the source file involves a number of intermediate steps like generating a Package IP script, and running the `package_xo` command. These intermediate commands can be used for debugging if necessary.

The output of the above `create_sc_xo` command is `vadd.xo`.

---

## Linking with the v++ Command

Link the SystemC model XO file by adding the kernel to the `v++ --link` command line:

```
v++ --link --platform <platform> --target hw_emu \
--config ./vadd.cfg --input_files ../vadd.xo --output ../vadd.link.xclbin \
--optimize 0 --save-temps --temp_dir ./hw_emu
```

The SystemC model can be used for both software emulation and hardware emulation, but is not supported for hardware build targets.

When a SystemC model is included in the `xclbin`, the design is no longer clock cycle accurate due to the limitations of the TLM.

---

## Xilinx TLM – SystemC Library for ESL Modelling

Xilinx TLM (XTLM) is a Xilinx extension of Accellera SystemC TLM 2.0 library for modeling AXI protocols. It provides simulation infrastructure between SystemC to SystemC using custom TLM sockets and also provides the co-simulation infrastructure between RTL and SystemC through Transactors. Though TLM 2.0 consists of sockets, payload, and phases, these default elements are not sufficient to support the full functionality of AXI. Therefore, XTLM has defined its own set of sockets, payload, and phases.

A user of XTLM should be familiar with the basics of SystemC and the TLM 2.0 specifications. The following table provides a brief introduction to classes available inside the XTLM library.

**Table 17: XTLM Features and Usage**

S. No.	XTLM Feature/Classes	Usage
1	aximm_payload	Transaction object class to describe data over the AXI4 memory map bus in a single transaction. Based on TLM 2.0 generic payload but not derived from.
2	axis_payload	Transaction object class to describe data over the AXI4-Stream bus in a single transaction. Based on TLM 2.0 generic payload but not derived from it.
3	xtlm_aximm_initiator_socket	Basic socket to be used for AXI4 memory map interface in master/initiator. One socket shall be instantiated for each READ and WRITE socket.
4	xtlm_aximm_simple_initiator_socket_tagged	Basic socket to be used for AXI4 memory map interface in master/initiator to bind to multiple interfaces. One socket shall be instantiated for each READ and WRITE socket. Each interface has to be added with new ID.
5	xtlm_aximm_target_socket	Basic socket to be used for AXI4 memory map interface in slave/target. One socket shall be instantiated for each READ and WRITE socket.
6	xtlm_aximm_passthrough_target_socket_tagged	Basic socket to be used for AXI4 memory map interface in slave/target to bind to multiple interfaces. One socket shall be instantiated for each READ and WRITE socket. Each interface has to be added with new ID.
7	xtlm_axis_initiator_socket	Basic socket to be used for AXI4-Stream interface in master/initiator.
8	xtlm_axis_simple_initiator_socket_tagged	Basic socket to be used for AXI4-Stream interface in master/initiator to bind to multiple interfaces. Each interface has to be added with new ID.
9	xtlm_axis_target_socket	Basic socket to be used for AXI4-Stream interface in slave/target.
10	xtlm_axis_passthrough_target_socket_tagged	Basic socket to be used for AXI4-Stream interface in slave/target to bind to multiple interfaces. Each interface has to be added with new ID.
11	xtlm_aximm_initiator_stub	One Initiator socket to stub XTLM slave interface. This socket is useful where there is no XTLM master. This avoids port binding errors in SystemC.
12	xtlm_aximm_target_stub	One Target socket to stub XTLM master interface. This socket is useful where there is no XTLM slave. This avoids port binding errors in SystemC
13	xtlm_axis_initiator_stub	One Initiator socket to stub XTLM AXI4-Stream slave interface. This socket is useful where there is no XTLM AXI4-Stream master. This avoids port binding errors in SystemC.
14	xtlm_aximm_target_stub	One Target socket to stub XTLM master interface. This socket is useful where there is no XTLM slave. This avoids port binding errors in SystemC.

**Table 17: XTLM Features and Usage (cont'd)**

S. No.	XTLM Feature/Classes	Usage
15	<code>xtlm_aximm_initiator_rd_socket_util</code>	Utility for AXI4 Memory Map Initiator Read Socket. Works only with <code>xtlm_aximm_initiator_socket</code> type.
16	<code>xtlm_aximm_initiator_wr_socket_util</code>	Utility for AXI4 Memory Map Initiator Write Socket. Works only with <code>xtlm_aximm_initiator_socket</code> type.
17	<code>xtlm_aximm_target_rd_socket_util</code>	Utility for AXI4 Memory Map Target Read Socket. Works only with <code>xtlm_aximm_target_socket</code> type.
18	<code>xtlm_aximm_target_wr_socket_util</code>	Utility for AXI4 Memory Map Target Write Socket. Works only with <code>xtlm_aximm_target_socket</code> type.

# Debug Techniques in Hardware Emulation

The following table shows some specific techniques for debugging different situations in hardware emulation.

*Table 18: Techniques for Debugging in Hardware Emulation*

Debug Focus	Description	Steps
x86 host (XRT)	Enable detailed XRT logs by updating <code>xrt.ini</code> .	Add the following to <code>xrt.ini</code> file. Run the test case with the updated <code>xrt.ini</code> . Review the generated <code>xrt_hal.log</code> .  <code>[runtime]hal_log=xrt_hal.log</code>
AXI traffic on SystemC models like AIE, NOC, CIPS	The host transactions are routed through <code>sim_qdma</code> SystemC module. These transactions can be dumped into log file. If PL is also SystemC, then even PL transactions can be viewed there.  If PL is RTL, then boundary of PL can be viewed in waveform.	In <code>xrt.ini</code> add,  <code>[Emulation]xtlm_aximm_log=true xtlm_axis_log=true</code>  View file <code>xsc_report.log</code> .
Viewing DDR content	The DDR model saves its contents in a binary file. In the folder where simulation is run (e.g. <code>package.hw_emu/sim/behav_waveform/xsim dir</code> ), look for files named like below. Each such binary file corresponds to a region of DDR memory at a particular offset.  <code>qemu-memory-addr@0x00000000 or qemu-memory-mem_0xc08000000@0xc080000000ULL</code>  These files represent DDR/LPDDR memory contents in binary form.  There is a backdoor connection between QEMU to NOC_DDR model. Thus, users will not see any transactions in the waveform nor will they see any logs. The shared memory is directly updated. To view memory contents, users can directly dump the memory contents.	The contents of the memory can be seen by using <code>hexdump</code> command. an example command is shown below.  <code>hexdump qemu-memory-mem_0x6000000000@0x60000000000ULL -s 0x80000000 -n 4096 -v -e '1/4 "%02X" "\n"' &gt; dump_600.log</code>

**Table 18: Techniques for Debugging in Hardware Emulation (cont'd)**

Debug Focus	Description	Steps
PS (QEMU)	<p>During firmware and software execution, the ARM APU can run into issues. You see details of various transactions and state on PS, please use these mechanisms.</p>	<ol style="list-style-type: none"> <li>1. Setenv variable <code>ENABLE_RP_LOGS=true</code> in the shell where QEMU is being run. Look for a log file named <code>sim/behav_waveform/xsim/rp_log.txt</code>. This contains transactions from PS to rest of the peripherals (other than DDR).</li> <li>2. Review <code>package.hw_emu/qemu_output.log</code> to see the output. This contains PS output to STDOUT (UART). If there is any error during PLM or other SW execution, those will be captured here. Look at the details of the error (CDO address, u-boot stage etc) to narrow down which CDO is causing the error.</li> <li>3. Run <code>launch_hw_emu.sh -enable_debug</code> mode which will direct QEMU logs in its own xterm window.</li> </ol>
AIE	<p>PDI is downloaded from PS to AIE through fast mode, thus transactions cannot be seen in the waveform. Users can enable logging all transactions at AIE AXI interfaces. There is a separate file per AXI interface</p>	<ol style="list-style-type: none"> <li>1. Enable setenv <code>ENABLE_AIE_DBG_TRACE</code>. View transaction log in the simulation folder at <code>aie_log/S00_AXI.txt</code> file.</li> <li>2. Enable AIE VCD Dump and view contents in VCD Analyzer by adding switch <code>-aie-sim-options</code> to <code>launch_hw_emu.sh</code> cmd line. See UG1076 for details of <code>aie-sim-options</code> file. These are common between <code>aiesim</code> and <code>hw_emu</code>.</li> <li>3. Create a file (<code>aie_sim_config.txt</code>) to enable AIE simulation options.             <ol style="list-style-type: none"> <li>a. In this file, add <code>"AIE_DEBUG_AXIMM=True"</code></li> <li>b. Pass this file on launch emulator cmd line:   <code>./package.hw_emu/launch_hw_emu.sh -aie-sim-options &lt;Absolute path to the options .txt file&gt;</code> </li> </ol> </li> </ol>

**Table 18: Techniques for Debugging in Hardware Emulation (cont'd)**

Debug Focus	Description	Steps
Dumping Waveform in DC flow (Batch Mode)	<p>Make sure the design is linked with the <code>-g</code> option and you have run the design at least once in Xsim GUI mode to save the required <code>.wcfg</code> file and save the list of relevant signals.</p> <p>Dump the waveform and open it in xsim along with saved <code>.wcfg</code> file</p>	<p>Update following options in <code>xrt.ini</code> option</p> <pre>[Emulation] user_pre_sim_script=&lt;use pre sim script absolute path&gt;</pre> <p>Pre-simulation script is needed to enable signal dumping:</p> <pre>log_wave -r *</pre>

# Working with I/O Traffic Generators

Some user applications such as video streaming and Ethernet-based applications make use of I/O ports on the platform to stream data into and out of the platform. For these applications, performing software and hardware emulation of the design, or running AI Engine simulation, requires a mechanism to mimic the hardware behavior of the I/O port, and to simulate data traffic running through the ports. I/O traffic generators let you model traffic through the I/O ports during software and hardware emulation in the Vitis application acceleration development flow, during the AI Engine simulation flows (x86sim, aiesim), or during logic simulation in the Vivado Design Suite.



**IMPORTANT!** Software emulation supports only AXI4-Stream I/O emulation and is not supported on Zynq®-7000 devices. Hardware emulation supports both AXI4-Stream and AXI4 memory map interface I/O emulation.

As described in [AXI4-Stream I/O Model for Streaming Traffic through Python/C++/Verilog](#), traffic generators can be written in Python/C/C++ or RTL (Verilog/SV) modules; they are launched in an external process which communicates with Vitis Emulation process or AI Engine simulation process using Inter Process Communication (IPC). The IPC connections are established using IPC AXI4-Stream master/slave modules as described in [Running Traffic Generators in Python/C++](#).

The following are additional details on ways to integrate traffic generators in Python/C/C++/Verilog with subsequent PL kernels or the AI Engine kernels based on your application.

## Adding Traffic Generators to Your Design

Xilinx devices have rich I/O interfaces. Your platform can have memory interfaces (e.g DDR) which have their own specific model. However, your platforms could also have other I/Os, for example GT-kernel based generic I/O, Video Streams, and Sensor data. I/O Traffic Generator kernels provide a method for platforms and applications to inject traffic onto the I/O during simulation.

This solution requires both the inclusion of streaming I/O kernels (XO) or IP in your design, and the use of a Python/C++/C provided by Xilinx to inject traffic or to capture output data from the emulation process. The Xilinx provided Python/C++/C library can be used to integrate traffic generator code into your application, run it as a separate process, and have it interface with the emulation process. Currently, Xilinx provides a library that enables interfacing at AXI4-Stream level to mimic any Streaming I/O for software and hardware emulation and AXI3/AXI4 memory mapped interface to mimic any memory mapped I/O for hardware emulation.

---

## AXI4-Stream I/O Model for Streaming Traffic through Python/C++/Verilog

The following section is specific to AXI4-Stream. The streaming I/O model can be used to emulate streaming traffic on the platform, and also support delay modeling. You can add streaming I/O to your application when targeted either for software emulation or hardware emulation, or add them to your custom platform design in the context of hardware emulation as described below:

- Streaming I/O kernels can be added to the device binary (.xclbin) file like any other compiled kernel object (XO) file, using the `v++ --link` command. Using these pre-compiled XO files reduces `v++` compile time. The Vitis installation provides kernels for AXI4-Stream interfaces of various data widths. The standard bit widths supported are 8, 16, 32, 64, 128, 256, 512. These can be found in the software installation at `$XILINX_VITIS/data/emulation/XO`.

Add these to your designs using the following example command:

```
v++ -t hw_emu --link $XILINX_VITIS/data/emulation/XO/  
sim_ipc_axis_master_32.xo $XILINX_VITIS/data/emulation/XO/  
sim_ipc_axis_slave_32.xo ...
```

In the example above, the `sim_ipc_axis_master_32.xo` and `sim_ipc_axis_slave_32.xo` provide 32-bit master and slave kernels that can be linked with the target platform and other kernels in your design to create the `.xclbin` file for the emulation build.

- In case of hardware emulation, IPC modules can also be added to a platform block design using the Vivado IP integrator for Versal and Zynq UltraScale+ MPSoC custom platforms. The tool provides `sim_ipc_axis_master_v1_0` and `sim_ipc_axis_slave_v1_0` IP to add to your platform design. These can be found in the software installation at `$XILINX_VIVADO/data/emulation/hw_em/ip_repo`.

The following is an example Tcl script used to add IPC IP to your platform design, which will enable you to inject data traffic into your simulation from an external process written in Python or C++:

```
#Update IP Repository path if required
set_property ip_repo_paths $XILINX_VIVADO/data/emulation/hw_em/ip_repo
[current_project]
## Add AXIS Master
create_bd_cell -type ip -vlnv xilinx.com:ip:sim_ipc_axis_master:1.0
sim_ipc_axis_master_0
#Change Model Property if required
set_property -dict [list CONFIG.C_M00_AXIS_TDATA_WIDTH {64}]
[get_bd_cells sim_ipc_axis_master_0]

##Add AXIS Slave
create_bd_cell -type ip -vlnv xilinx.com:ip:sim_ipc_axis_slave:1.0
sim_ipc_axis_slave_0
#Change Model Property if required
set_property -dict [list CONFIG.C_S00_AXIS_TDATA_WIDTH {64}]
[get_bd_cells sim_ipc_axis_slave_0]
```

## Writing Traffic Generators in Python

You must also include a traffic generator process while simulating your application to generate data traffic on the I/O traffic generators, or to capture output data from the emulation process. The Xilinx provided Python or C++ library can be used to create the traffic generator code as described below. Also, an application can communicate to multiple I/O interface. It is not necessary to have each instance of I/O utilities to be in a separate process/thread. In case your application demands it, you might consider the non-blocking version APIs (details provided in the following section).

- For Python, set \$PYTHONPATH on the command terminal:

```
setenv PYTHONPATH $XILINX_VIVADO/data/emulation/hw_em/lib/python:\$XILINX_VIVADO/data/emulation/python/xtlm_ipc
```

- Sample Python code to connect with the gt\_master instance would look like the following:

```
Blocking Send
from xilinx_xtlm import ipc_axis_master_util
from xilinx_xtlm import xtlm_ipc
import struct

import binascii

#Instantiating AXI Master Utilities
master_util = ipc_axis_master_util("gt_master")

#create payload
payload = xtlm_ipc.axi_stream_packet()
payload.data = "BINAY_DATA" # One way of getting "BINAY_DATA" from
integer can be like payload.data = bytes(bytearray(struct.pack("i",
int_number))) More info @ https://docs.python.org/3/library/struct.html
payload.tlast = True #AXI Stream Fields
#Optional AXI Stream Parameters
payload.tuser = "OPTIONAL_BINAY_DATA"
```

```

payload.tkeep = "OPTIONAL_BINARY_DATA"

#Send Transaction
master_util.b_transport(payload)
master_util.disconnect() #Disconnect connection between Python & Emulation

```

- Sample Python code to connect with the `gt_slave` instance would look like the following:

```

Blocking Receive
from xilinx_xtlm import ipc_axis_slave_util
from xilinx_xtlm import xtlm_ipc

#Instantiating AXI Slave Utilities
slave_util = ipc_axis_slave_util("gt_slave")

#Sample payload (Blocking Call)
payload = slave_util.sample_transaction()
slave_util.disconnect() #Disconnect connection between Python & Emulation

```

- For non-blocking version APIs in Python, it can be found at:

```
$XILINX_VIVADO/data/emulation/ip_utils/xtlm_ipc/xtlm_ipc_v1_0/python/
xilinx_xtlm.py
```

## Writing Traffic Generators in C++

- For C++, the APIs are available at:

```
$XILINX_VIVADO/data/emulation/cpp/inc/xtlm_ipc/axis/
```

You can build the executable with include path:

```
-I $XILINX_VIVADO/data/emulation/cpp/inc/xtlm_ipc/axis/
```

and linked against library as:

```
-L $XILINX_VIVADO/data/emulation/cpp/lib/
```

And use `-lxtlm_ipc` with a `gcc` compiler.

The C++ API provides both blocking and non-blocking function support. The following snippets show the usage.

- Blocking send:

A simple API is available if you prefer not to have fine granular control (recommended):

```

#include "xtlm_ipc.h" //Include file
void send_data()
{
    //! Instantiate IPC socket with name matching in IPI diagram...
    xtlm_ipc::axis_initiator_socket_util<xtlm_ipc::BLOCKING>
    socket_util("gt_master");
    const unsigned int NUM_TRANSACTIONS = 8;
    std::vector<char> data;
    std::cout << "Sending " << NUM_TRANSACTIONS << " data transactions..." 
    << std::endl;
}

```

```

for(int i = 0; i < NUM_TRANSACTIONS; i++) {
    data = generate_data();
    print(data);
    socket_util.transport(data.data(), data.size());
}
}
}

```

For advanced users who need fine granular control over AXI4-Stream can use the following:

```

#include "xtlm_ipc.h" //Include file

void send_packets()
{
    //! Instantiate IPC socket with name matching in IPI diagram...
    xtlm_ipc::axis_initiator_socket_util<xtlm_ipc::BLOCKING>
    socket_util("gt_master");

    const unsigned int NUM_TRANSACTIONS = 8;
    xtlm_ipc::axi_stream_packet packet;

    std::cout << "Sending " << NUM_TRANSACTIONS << " Packets..." 
    << std::endl;
    for(int i = 0; i < NUM_TRANSACTIONS; i++) {
        xtlm_ipc::axi_stream_packet packet;
        // generate_data() is your custom code to generate traffic
        std::vector<char> data = generate_data();
        //! Set packet attributes...
        packet.set_data(data.data(), data.size());
        packet.set_data_length(data.size());
        packet.set_tlast(1);
        //Additional AXIS attributes can be set if required
        socket_util.transport(packet); //Blocking transport API to send
        the transaction
    }
}

```

- Blocking receive:

A simple API is available if you prefer not to have fine granular control (recommended):

```

#include "xtlm_ipc.h" //Include file
void receive_data()
{
    //! Instantiate IPC socket with name matching in IPI diagram...
    xtlm_ipc::axis_target_socket_util<xtlm_ipc::BLOCKING>
    socket_util("gt_slave");
    const unsigned int NUM_TRANSACTIONS = 100;
    unsigned int num_received = 0;
    std::vector<char> data;
    std::cout << "Receiving " << NUM_TRANSACTIONS << " data transactions..." 
    << std::endl;
    while(num_received < NUM_TRANSACTIONS) {
        socket_util.sample_transaction(data);
        print(data);
        num_received += 1;
    }
}

```

For advanced users who need fine granular control over AXI4-Stream can use the following:

```
#include "xtlm_ipc.h"

void receive_packets()
{
    //! Instantiate IPC socket with name matching in IPI diagram...
    xtlm_ipc::axis_target_socket_util<xtlm_ipc::BLOCKING>
socket_util("gt_slave");

    const unsigned int NUM_TRANSACTIONS = 8;
    unsigned int num_received = 0;
    xtlm_ipc::axi_stream_packet packet;

    std::cout << "Receiving " << NUM_TRANSACTIONS << " packets..." 
<< std::endl;
    while(num_received < NUM_TRANSACTIONS) {
        socket_util.sample_transaction(packet); //API to sample the
transaction
        //Process the packet as per requirement.
        num_received += 1;
    }
}
```

- Non-Blocking send:

```
#include <algorithm>      // std::generate
#include "xtlm_ipc.h"

//A sample implementation of generating random data.
xtlm_ipc::axi_stream_packet generate_packet()
{
    xtlm_ipc::axi_stream_packet packet;
    // generate_data() is your custom code to generate traffic
    std::vector<char> data = generate_data();

    /// Set packet attributes...
    packet.set_data(data.data(), data.size());
    packet.set_data_length(data.size());
    packet.set_tlast(1);
    //packet.set_tlast(std::rand()%2);
    //! Option to set tuser tkeep optional attributes...

    return packet;
}
//Simple Usage

void send_data()
{
    //! Instantiate IPC socket with name matching in IPI diagram...
    xtlm_ipc::axis_initiator_socket_util<xtlm_ipc::NON_BLOCKING>
socket_util("gt_master");

    const unsigned int NUM_TRANSACTIONS = 8;
    std::vector<char> data;

    std::cout << "Sending " << NUM_TRANSACTIONS << " data
transactions..." << std::endl;
    for(int i = 0; i < NUM_TRANSACTIONS/2; i++) {
        data = generate_data();
        print(data);
        socket_util.transport(data.data(), data.size());
    }
}
```

```

        std::cout<< "Adding Barrier to complete all outstanding
transactions..." << std::endl;
        socket_util.barrier_wait();
        for(int i = NUM_TRANSACTIONS/2; i < NUM_TRANSACTIONS; i++) {
            data = generate_data();
            print(data);
            socket_util.transport(data.data(), data.size());
        }
    }
void send_packets()
{
    //! Instantiate IPC socket with name matching in IPI diagram...
    xtlm_ipc::axis_initiator_socket_util<xtlm_ipc::NON_BLOCKING>
socket_util("gt_master");
    // Instantiate Non Blocking specialization

    const unsigned int NUM_TRANSACTIONS = 8;
    xtlm_ipc::axi_stream_packet packet;

    std::cout << "Sending " << NUM_TRANSACTIONS << " Packets..." 
<<std::endl;
    for(int i = 0; i < NUM_TRANSACTIONS; i++) {
        packet = generate_packet(); // Or user's test patter / live data
etc.
        socket_util.transport(packet);
    }
}

```

- **Non-Blocking receive:**

```

#include <unistd.h>
#include "xtlm_ipc.h"
//Simple Usage
void receive_data()
{
    //! Instantiate IPC socket with name matching in IPI diagram...
    xtlm_ipc::axis_target_socket_util<xtlm_ipc::NON_BLOCKING>
socket_util("gt_slave");

    const unsigned int NUM_TRANSACTIONS = 8;
    unsigned int num_received = 0, num_outstanding = 0;
    std::vector<char> data;

    std::cout << "Receiving " << NUM_TRANSACTIONS << " data
transactions..." <<std::endl;
    while(num_received < NUM_TRANSACTIONS) {
        num_outstanding = socket_util.get_num_transactions();
        num_received += num_outstanding;

        if(num_outstanding != 0) {
            std::cout << "Outstanding data transactions = "<<
num_outstanding <<std::endl;
            for(int i = 0; i < num_outstanding; i++) {
                socket_util.sample_transaction(data);
                print(data);
            }
        }
        usleep(100000);
    }
}

void receive_packets()
{

```

```

//! Instantiate IPC socket with name matching in IPI diagram...
xtlm_ipc::axis_target_socket_util<xtlm_ipc::NON_BLOCKING>
socket_util("gt_slave");

const unsigned int NUM_TRANSACTIONS = 8;
unsigned int num_received = 0, num_outstanding = 0;
xtlm_ipc::axi_stream_packet packet;

std::cout << "Receiving " << NUM_TRANSACTIONS << " packets..." 
<< std::endl;
while(num_received < NUM_TRANSACTIONS) {
    num_outstanding = socket_util.get_num_transactions();
    num_received += num_outstanding;

    if(num_outstanding != 0) {
        std::cout << "Outstanding packets = " << num_outstanding
    << std::endl;
        for(int i = 0; i < num_outstanding; i++) {
            socket_util.sample_transaction(packet);
            print(packet);
        }
    }
    usleep(100000); //As transaction is non-blocking we would like to
give some delay between consecutive samplings
}
}

```

- For C APIs, it can be found at:

```
$XILINX_VIVADO/data/emulation/c/inc/xtlm_ipc/axis/c_axis_socket.h
```

It can be linked against pre-compiled library at:

```
$XILINX_VIVADO/data/emulation/c/lib/
```

A full system-level example is available at [https://github.com/Xilinx/Vitis\\_Accel\\_Examples/tree/master/emulation](https://github.com/Xilinx/Vitis_Accel_Examples/tree/master/emulation).

## Running Traffic Generators in Python/C++

After generating an external process binary as shown above using the headers and sources available at \$XILINX\_VIVADO/data/emulation/ip\_utils/xtlm\_ipc/xtlm\_ipc\_v1\_0/<supported\_language>, you can run the emulation using the following steps:

1. Launch the Vitis emulation or Vivado simulation using the standard process and wait for the simulation to start.
2. From another terminal(s), launch the external process such as Python/C++/C.

**Note:** If you are running multiple I/O or traffic generator-based solutions on the same machine, then set XTLM\_IPC\_SOCK\_DIR unique to each test case on both emulation terminal as well as external process terminal. For example, `setenv XTLM_IPC_SOCK_DIR <test_case_dir>` (same environment on both emulation process and external process).



**WARNING!** Xilinx provides an `end_of_simulation()` API to terminate emulation from master utilities of memory mapped AXI4 and AXI4-Stream interfaces. However, you are warned not to use this method unless there is no way to terminate emulation from host. In a normal course of emulation, external process is not expected to terminate emulation. Use this in exceptional scenarios.

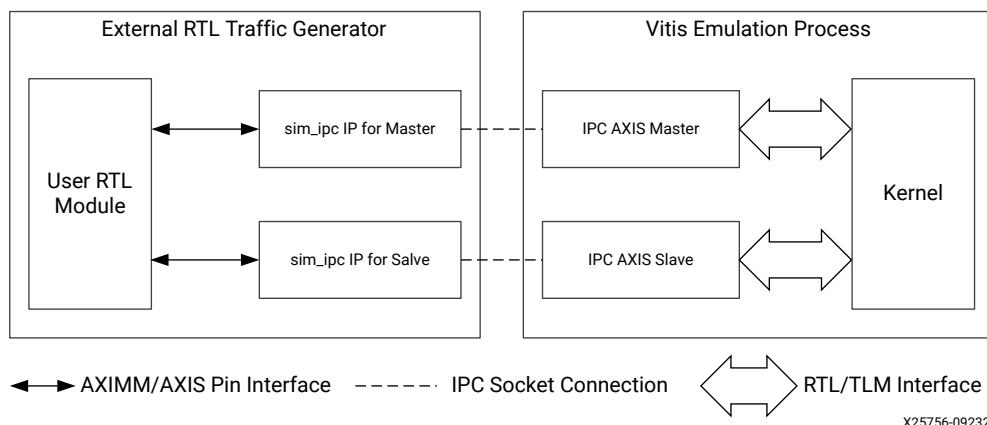
## External RTL Traffic Generators using SV/Verilog

Generate traffic using the existing test bench written in the System Verilog/Verilog with slight modification to your test bench hierarchy, as explained below.

### ***External RTL Traffic Generator and Emulation Process***

External RTL traffic generators are used to drive traffic to Vitis emulation process or AI Engine simulation process using SystemVerilog or Verilog modules.

*Figure 27: Test Bench Hierarchy*



As shown in the figure above, the external test bench (on the left) and the Vitis emulation (on the right), both run as separate simulation processes. To establish communication between two processes using IPC, you must instantiate SIM\_IPC Master/Slave modules.

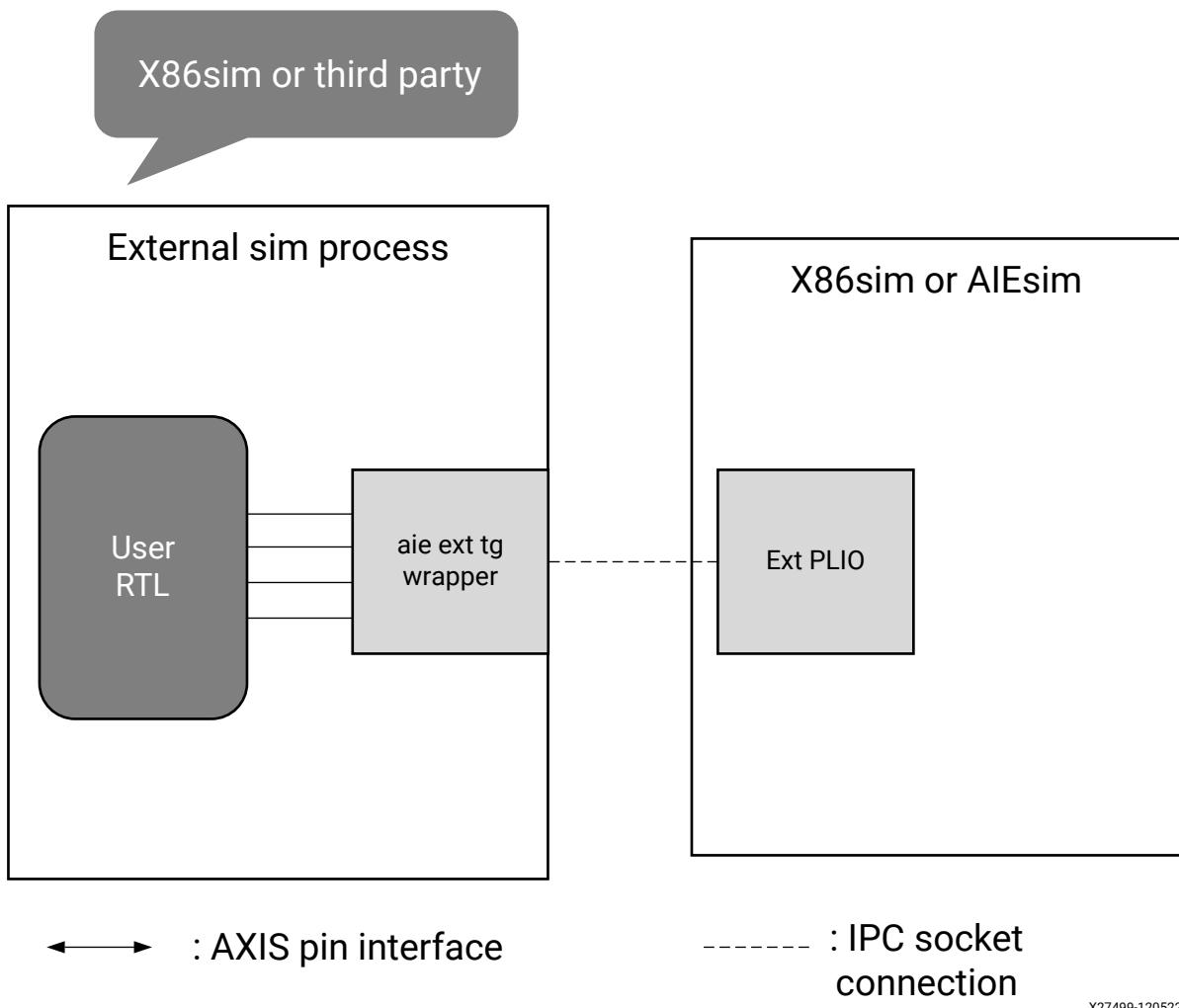
Perform the following modifications:

1. You need to create a project in Vivado simulator. For details on how to create a project, refer [Vivado Design Suite User Guide: Design Flows Overview \(UG892\)](#)
2. Once the project is created, you need to instantiate `sim_ipc` IP in the external SV/Verilog testbench.
3. Then run the `export_simulation` command in Vivado to generate the scripts for the simulation
4. Run the simulation in Vivado simulator. For details running simulation refer to [Vivado Design Suite User Guide: Logic Simulation \(UG900\)](#)

## External RTL Traffic Generator and AI Engine Simulation

The same technique can be deployed to drive traffic from the external System Verilog/Verilog traffic generators/test benches to the AI Engine simulator or x86-simulator.

Figure 28: XTLM Test Bench Hierarchy



To generate the AI Engine wrapper stub module (`aie_wrapper_ext_tb.v`) use the following steps:

1. The wrapper stubs will be generated based on the external PLIO declarations in the ADF graph. You need to perform the ADF graph compilation to generate `scsim_config.json` file that resides in `./Work/config/scsim_config.json` directory. This config file contains information on the PLIOs declared in the graph. For more details on how to perform ADF graph compilation and external PLIOs declaration, refer to *Versal ACAP AI Engine Programming Environment User Guide (UG1076)*.

- Using this config file as argument to the `gen_aie_wrapper.py` script, you can auto-generate Verilog stub modules based on ext PLIO declared in ADF Graph:

```
python3 ${XILINX_VITIS}/data/emulation/scripts/gen_aie_wrapper.py \
    -json Work/config/scsim_config.json --mode <wrapper/vivado>
```



**TIP:** The python script is available in the Vitis installation area as shown in the example above. There are two modes for the script: wrapper and Vivado mode. By default, the script runs in Vivado mode.

The name of the instance stubs must be identical to the name of the corresponding external PLIOs in the graph and will be reflected in the generated `aie_wrapper_ext_tb.v` file.

After running the `gen_aie_wrapper.py` script you can see `aie_wrapper_ext_tb.v` is generated that has instances of `sim_ipc_axis` modules that can be directly instantiated in your external test bench.

**Note:** The module used to send data to/from external traffic generator to AI Engine simulator/x86sim are the XTLM IPC SystemC modules which are present inside the wrapper stub module which includes all the XTLM IPC modules. This wrapper needs to be instantiated in the external test bench to establish the connection as shown in the figure above.

## Instantiating AI Engine Wrapper in the Test Bench

The aie wrapper module (`aie_wrapper_ext_tb.v`) needs to be instantiated in the external test bench. The aiesim expects data to be in beats instead of transaction, so you need to keep `tlast` at high (`1'b1`) all the time.

**Note:** You can add timescale directive as per your requirement in `aie_wrapper_ext_tb.v`.

## Generating sim\_ipc\_axis IP for Vivado Project

By default, the python script generates `aie_wrapper_ext_tb_ip.tcl` and `aie_wrapper_ext_tb_proj.tcl` along with the wrapper Verilog file as mentioned in the previous section.

There are two ways to proceed based on the existence of a Vivado project:

- If you have already created Vivado project, use the IP flow described here. From the **Tcl console** source the `aie_wrapper_ext_tb_ip.tcl` script:

```
source <absolute_path>/aie_wrapper_ext_tb_ip.tcl
```

This Tcl script can be used for generating required `sim_ipc_axis` IP. After sourcing the Tcl file you will see hierarchy created under `simulation_sources`. You can add the required files and directories for your project.

2. If a Vivado project is not already created, use the project script `aie_wrapper_ext_tb_proj.tcl` to create one. From a terminal use the following command:

```
vivado -mode batch -source aie_wrapper_ext_tb_proj.tcl
```

**Note:** To use third party simulators, you need to update the required paths for `SIMULATOR_GCC_PATH`, `SIMULATOR_LIBS_PATH` and `INSTALL_BIN_PATH`. For more details on how to set the third party simulators, please refer [Vivado Design Suite User Guide: Logic Simulation \(UG900\)](#).

After sourcing `aie_wrapper_ext_tb_proj.tcl`, the tool will generate the `export_sim` directory with sub-directories and scripts required for use with other simulators. This Tcl script sources the `aie_wrapper_ext_tb_ip.tcl` script.



**TIP:** The scripts mentioned above only contain the `sim_ipc_axis` modules, so you must add any additional required RTL modules and options to the script. You can modify and directly include required RTL in the needed script.

## Running the RTL Traffic Generator with AI Engine Simulation

You can launch the external process using the following steps:

1. If already inside the Vivado project, once the project hierarchy is updated after adding the required sources, you can run the simulation from Vivado. Refer to [Vivado Design Suite User Guide: Logic Simulation \(UG900\)](#) for more information.
2. If outside the Vivado project, once the `export_sim` directory is generated with required simulation scripts, you can traverse inside the appropriate simulator directory and run the `<top_module_name>.sh` script to launch the RTL simulation.
3. Also, simultaneously launch the AI Engine simulator process as already mentioned. For details on how to run the `aiesimulator` command, refer to [Versal ACAP AI Engine Programming Environment User Guide \(UG1076\)](#).

After simulation is launched you can see the traffic propagating to and from the user RTL.



**TIP:** For more details on how to integrate and launch the external RTL traffic generator with AI simulation process refer to the tutorial

## AXI4 Memory Map External Traffic through Python/C++

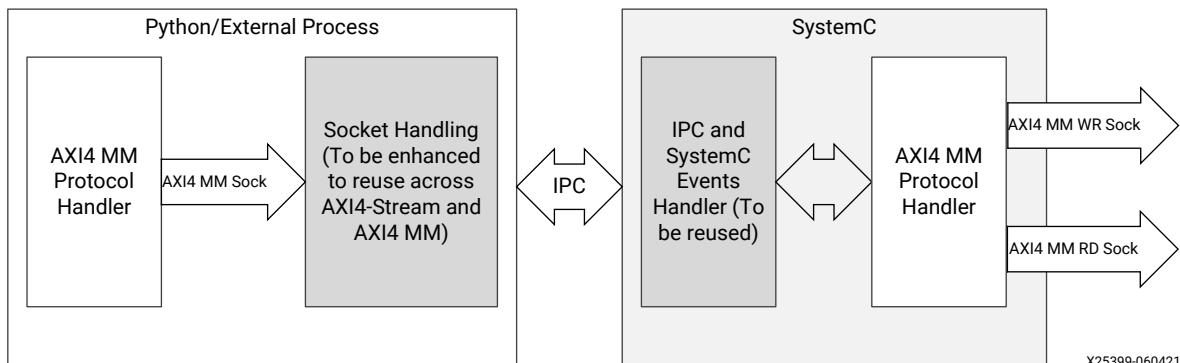
The AXI4 memory map external traffic which is supported only for hardware emulation has the following specifications:

- Only transaction-level granularity is supported.

- Re-ordering of transactions is not supported.
- Parallel Read, Write transactions are not supported (transactions will be serialized).
- Unaligned transactions are not supported.

The following figure shows the high-level design.

**Figure 29: AXI4 Memory Map External Traffic Design**



## Use Cases

The use cases include the following:

- Emulate AXI4 memory map Master/Slave through an external process such as Python/C++. This can help you with emulating design with quick design time of AXI4 Master/Slave without investing resources in developing AXI4 Master.
- Chip-to-chip connection between two FPGAs can be emulated with AXI4 memory map Interprocess communication.

## API/Pseudo Code

For API/pseudo code, a single instance of AXI4 memory map transaction is used for the complete transaction. This is in line with how payload is used in the Xilinx SystemC modules. For AXI4 Master, there is a `b_transport(aximm_packet)` API. After the call, `aximm_packet` is updated with a response given by AXI4 Slave. For AXI4 Slave, there are `sample_transaction()` and `send_response(aximm_packet)` APIs.

The following code snippets show the API usage in the context of C++.

- Code snippet for C++ Master:

```
auto payload = generate_random_transaction(); //Custom Random transaction
generator. Users can configure AXI properties on the payload.
/* Or User can set the AXI transaction properties as follows
payload->set_addr(std::rand() * 4);
payload->set_len(1 + (std::rand() % 255));
payload->set_size(1 << (std::rand() % 3));
```

```
/*
master_utl.b_transport(*payload.get(), std::rand() % 0x10); //A blocking
call. Response will be updated in the same payload. Each AXI MM
transaction will use same payload for whole transaction
std::cout << "-----Transaction Response-----" << std::endl;
std::cout << *payload << std::endl; //Prints AXI transaction info
```

- **Code snippet for C++ Slave:**

```
auto& payload = slave_utl.sample_transaction(); // Sample the transaction

//If it is read transaction, give read data
if(payload.cmd() == xt1m_ipc::aximm_packet_command_READ)
{
    rd_resp.resize(payload.len()*payload.size());
    std::generate(rd_resp.begin(), rd_resp.end(), []()
    {   return std::rand()%0x100;});

}

//Set AXI response (for Read & Write)
payload.set_resp(std::rand()%4);
slave_utl.send_response(payload); //Send the response to the master
```

The following code snippets show the API usage in the context of Python.

You need to set PYTHONPATH as follows:

- For example, on C Shell:

```
setenv PYTHONPATH $XILINX_VIVADO/data/emulation/hw_em/lib/python:
$XILINX_VIVADO/data/emulation/ip_utils/xt1m_ipc/xt1m_ipc_v1_0/python
```

- **Code snippet of Python Master:**

```
aximm_payload = xt1m_ipc.aximm_packet()
random_packet(aximm_payload) # Custom function to set AXI Properties
randomly
#Or user can set AXI properties as required
#aximm_payload.addr = int(random.randint(0, 1000000)*4)
#aximm_payload.len = random.randint(1, 64)
#aximm_payload.size = 4

master_utl.b_transport(aximm_payload)
#After this call aximm_payload will have updated response as set by the
AXI Slave.
```

- **Code snippet of Python Slave:**

```
aximm_payload = slave_utl.sample_transaction()
aximm_payload.resp = random.randint(0,3)
if not aximm_payload.cmd: #if it is a read transaction set Random data
    tot_bytes = aximm_payload.len * aximm_payload.size
    for i in range(0, int(tot_bytes/SIZE_OF_EACH_DATA_IN_BYTES)):
        aximm_payload.data += bytes(bytarray(struct.pack(">I",
random.randint(0,60000)))) # Binary data should be aligned with C struct

slave_utl.send_resp(aximm_payload)
```

## AXI4 Memory Map I/O Limitations in the Platform

The following shows the AXI4 memory map I/O limitations in the platform:

- During platform development, AXI4 memory map I/O can be connected to any memory/slave.
- Master AXI4 memory map I/O cannot connect to kernel as kernel cannot provide an additional slave interface.
- AXI4 memory map Slave I/O can be used without any restrictions.
- AXI4 memory map Master I/O can be used where data needs to be driven from external process to memory/slave.

## XO Usage

The use cases of AXI4 memory map I/O XO differs from AXI4-Stream I/O XO. AXI4 memory map XOs have few limitations on usage during the link stage of Vitis listed as below:

- Only AXI4 memory map Master I/O can be used.
- AXI4 memory map Master I/O can connect only with available slaves in the platform.
- AXI4 memory map Master I/O cannot communicate with kernel in the design.

For XO usage during link stage:

- To generate XO, developers can use the script available at \$XILINX\_VITIS/data/emulation/XO/scripts/aximm\_xo\_creation.sh
- Required configuration of XO can be generated using the above script.

```
$XILINX_VITIS/data/emulation/XO/scripts/aximm_xo_creation.sh --  
address_width <adr_width> --data_width <data_width> --id_width <id_width>  
--output_path <output_path>.xo  
$XILINX_VITIS/data/emulation/XO/scripts/aximm_xo_creation.sh --  
address_width 64 --data_width 64 --id_width 4 --output_path  
sim_ipc_aximm_master.xo
```

- After generating XO, it can be used in the design with configuration as shown below (sample usage, actual connection to be done based on the requirement):

```
[connectivity]  
nk=sim_ipc_aximm_master:1:aximm_master  
sp=aximm_master.M_AXIMM:HBM[0]
```

# 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.

This section contains the following chapters:

- [Profiling the Application](#)
- [Debugging Applications and Kernels](#)

# Profiling the Application

The 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
- AI Engine profiling and event trace
- 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 should be optimized for performance and resource usage.

There are four distinct areas to be considered when performing algorithm optimization in Vitis: System resource usage and performance, kernel optimization, host optimization, and data transfer optimization. The following Vitis reports and graphical tools support your efforts to profile and optimize these areas:

- [Guidance](#)
- [System Estimate Report](#)
- [HLS Report](#)
- [Profile Summary Report](#)
- [Timeline Trace](#)
- [Waveform View and Live Waveform Viewer](#)

When properly enabled as described in [Enabling Profiling in Your Application](#), reports are automatically generated while running the active build, either from the command line as described in [Section IV: Building and Running the Application](#), or from the Vitis integrated design environment (IDE). Separate reports are generated for the different build targets and can be found in the respective report directories. Refer to [Output Directories of the v++ Command](#) or [Output Directories from the Vitis IDE](#) for more information on locating these reports.

Reports can be viewed in Vitis analyzer, or in some cases from the Vitis IDE. To access these reports from Vitis analyzer, open the `run_summary` report as explained in [Section VIII: Using the Vitis Analyzer](#).

---

## Enabling Profiling in Your Application

To enable profiling and capturing 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. You should be sure 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 19: 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 <code>opencl_trace</code> option in the <code>xrt.ini</code> file.	Generates the <code>opencl_trace.csv</code> file and the <code>xrt.run_summary</code> for viewing in Vitis analyzer.
Host Application XRT Native API	Specified by the use of the <code>native_xrt_trace</code> option in the <code>xrt.ini</code> file.	Generates profile summary and trace events for the XRT API as described in <a href="#">Writing the Software Application</a> .

Table 19: Profiling Host and Kernels (cont'd)

Profile/Trace	Description	Comments
Host Application User-Event Profiling	Requires additional code in the host application as described in <a href="#">Custom Profiling of the Host Application</a> .	Generates user range data and user events for the host application.  <b>TIP:</b> <i>Can be used to capture event data for user-managed kernels as described in <a href="#">Working with User-Managed Kernels</a>.</i>
Low Overhead Profiling	Specified by the use of the <code>lop_trace</code> option in the <code>xrt.ini</code> file.	Generates the <code>lop_trace.csv</code> file as described in <a href="#">Enabling Low Overhead Profiling</a> .
Device Side Profiling	Enabled by the use of <code>--profile</code> options during <code>v++</code> compilation and linking, as described in <a href="#">--profile Options</a> , and the use of <code>device_trace</code> in the <code>xrt.ini</code> file.	Enables capturing data traffic between the host and kernel, kernel stalls, the execution times of kernels and compute units (CUs), as well as monitoring activity in Versal AI Engines.
AI Engine Graph and Kernels	Specified by the use of the <code>aie_profile</code> and <code>aie_trace</code> options in the <code>xrt.ini</code> file. These options can be specified together or separately.	Generates the <code>aie_profile_&lt;device&gt;.csv</code> and <code>aie_trace_##_&lt;stream id&gt;.txt</code> reports.
Power Profile	Specified by the use of the <code>power_profile</code> option in the <code>xrt.ini</code> file.	Generates the <code>power_profile_&lt;device&gt;.csv</code> report.  <b>TIP:</b> <i>This feature is not supported on certain platforms including AWS.</i>
Vitis AI Profiling	Specified by the use of the <code>vitis_ai_profile</code> option in the <code>xrt.ini</code> file.	Enables counter profiling of DPUs to generate the <code>opencl_summary.csv</code> file and the <code>xrt.run_summary</code> for viewing in Vitis analyzer.

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 may 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 are 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 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 `data_transfer_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 should 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 30: 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](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.



**TIP:** An example of `user_range` and `user_event` can be seen in the host code at [https://github.com/Xilinx/Vitis\\_Accel\\_Examples/blob/master/host/debug\\_profile/src/host.cpp](https://github.com/Xilinx/Vitis_Accel_Examples/blob/master/host/debug_profile/src/host.cpp).

## 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.

The [debug\\_profile](#) example of the [Vitis\\_Accel\\_Examples](#) demonstrates user event profiling in a host application. With your host application properly instrumented, XRT can capture profile data from these user-defined ranges and events, as well as 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     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
159     q.enqueueMigrateMemObjects({buffer_result},CL_MIGRATE_MEM_OBJECT_HOST);
160
161     xrtUEMark("Starting verification") ;
162
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 all three flows (hardware, hardware emulation, and software 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
```



**TIP:** The `lop_trace` parameter can be enabled alongside other profiling parameters, but doing so eliminates any benefit of low overhead profiling by capturing all profiling data as well.

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:

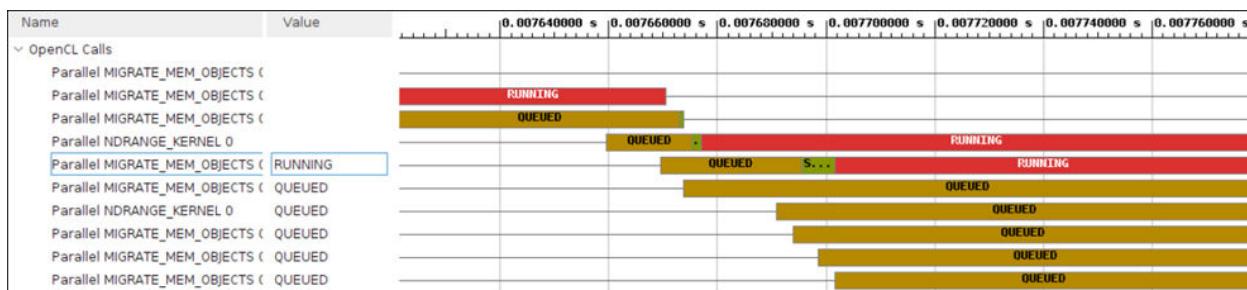
```
[Debug]
xocl_debug=true
#xocl_event_begin= 0 (default)
#xocl_event_end=1000 (default)
```

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 31: No Overhead Timeline*



*Table 20: 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.	<code>opencl_trace = true</code>
Partial	Low	When profile overhead is High and unexpected delay is experienced.	<code>lop_trace = true</code>
Minimum	No	Only to confirm the cause of delay between events.	<code>xocl_debug = true</code>

# 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 Vivado Design Suite when invoked by the `v++` compiler. The generated design guidance can have several severity levels; warning messages, informational messages and design rule checks are provided during software emulation, hardware emulation, and system builds. The profile design guidance helps you interpret the profiling results which allows you to focus on improving performance.

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 IDE.

You can open the Guidance report as discussed in [Section VIII: Using the Vitis Analyzer](#). To access the Guidance report, open the Compile Summary, the Link Summary, or the Run Summary, and open the Guidance report.

- 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 as well as 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 kernel 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, 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 32: Design Guidance Example**

Name	Threshold	Actual	Details	Resolution	Impact
DATAFLOW_ACCELERATION	> 1.500	1.000	Compute unit <a href="#">runOnFpga_1</a> had dataflow acceleration of <a href="#">1.000</a> .	Improve dataflow acceleration to maximize performance. Click <a href="#">here</a> .	Medium
KERNEL_COUNT (1)	> 1				
KERNEL_COUNT	> 1	1	Kernel <a href="#">runOnFpga</a> was executed <a href="#">17</a> time(s) with 1 compute unit(s).	Ensure kernel utilizes multiple compute units. Click <a href="#">here</a> .	Medium
KERNEL_PORT_DATA_WIDTH (1)					
KERNEL_PORT_DATA_WIDTH	= 512	32	Port <a href="#">m_axi_maxiport1</a> has a data width of 32.	Utilize the entire memory data width. Click <a href="#">here</a> .	Medium
System (2)					
Device: xilinx_u200_xdma_201830_2-0 (1)					
KERNEL_READ_TRANSFER_AMOUNT_MIN (1)					
KERNEL_READ_TRANSFER_AMOUNT_MIN					
Host Data Transfers (1)					
HOST_READ_TRANSFER_UTIL (1)	> 70.0				
HOST_READ_TRANSFER_UTIL	> 70.0	0.569	Host read bandwidth utilization was 0.569%.	Improve efficiency of host read transfers. Click <a href="#">here</a> .	Medium



**TIP:** As described in [Setting Guidance Thresholds](#), you can manually edit the values in the Threshold column of the Run Guidance report to customize the report.

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. The [Guidance Messaging](#) web page lists all of the current messages for your review.

Figure 33: Guidance Messaging Web Page

The screenshot shows a web page titled "XILINX® Guidance Messaging". The main content area is titled "Kernel Port Data Width". On the left, there is a sidebar with a list of items, many of which have a blue triangle icon followed by text. Some items are collapsed (indicated by a triangle) and some are expanded (indicated by a plus sign). The expanded items include:

- Block-RAM Resource Utilization
- DSP Resource Utilization
- Synchronous Storage Resource Utilization
- System Clock Timing Violation
- Look-up Table Resource Utilization
- Kernel clock minimum frequency
- Kernel clock maximum frequency
- Kernel timing failure
- System clock minimum frequency
- System clock maximum frequency
- System timing failure
- Higher Frequency Possible
- Runtime controllable clock domains achieved clock frequency (MHz)
- Compute Unit Utilization
- Device Utilization
- Migrate Memory API
- Average Read Size
- Average Write Size

The main content area has sections for "Description", "Explanation", and "Recommendation".

Kernel and Compute Unit objects, as well as profile reported data values, can also be cross-probed to other views like the System Diagram or Profile Report. Refer to [Working with Summary Reports](#) for more information.

## 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 [Section VIII: Using the Vitis Analyzer](#) 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.
- **Name:** The Name column displays a mnemonic name uniquely identifying the guidance rule. These names are designed to assist in memorizing specific guidance rules in the view.
- **Severity:** The Severity column allows the easy identification of the importance of a guidance rule.
- **Full Name:** The Full Name provides a less cryptic name compared to the mnemonic name in the Name column.
- **Categories:** Most messages are grouped within different categories. This allows the GUI to display groups of messages within logical categories under common tree nodes in the Guidance view.
- **Threshold:** The Threshold column displays an expected threshold value, which determines whether or not a rule is met. The threshold values are determined from many applications that follow good design and coding practices.
- **Actual:** The Actual column displays the values actually encountered on the specific design. This value is compared against the expected value to see if the rule is met.
- **Details:** The Details column provides a brief message describing the specifics of the current rule.
- **Resolution:** The Resolution column provides a pointer to common ways the model source code or tool transformations can be modified to meet the current rule. Clicking the link brings up a 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 Xilinx devices. Build time is also affected by the target device and the number of compute units instantiated onto the FPGA fabric. Therefore, it is useful to estimate the performance of an application without needing to build it for the system hardware.

The System Estimate report provides estimates of FPGA resource usage and the 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 34: System Estimate

The screenshot shows the Vitis Analyzer 2019.2 interface with the 'System Estimate' report open. The report details for the design 'mmult\_hw.xilinx\_u200\_xdma\_201830\_2' are displayed. Key sections include:

- Kernel Summary:** Shows the kernel name (mmult), target (fpga0:OCL\_REGION\_0), and OpenCL Library (mmult\_hw.xilinx\_u200\_xdma\_201830\_2).
- Timing Information (MHz):** Lists the target clock frequency (300.000000MHz) and estimated frequency (411.015198MHz) for the mmult kernel.
- Latency Information (clock cycles):** Provides start interval, best case, avg case, worst case, and real-time values for the mmult kernel.
- Area Information:** Lists FF, LUT, DSP, BRAM, and URAM usage for the mmult kernel.

## 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. Refer to [Section VIII: Using the Vitis Analyzer](#) for more information.

## Interpreting the System Estimate Report

The System Estimate report generated by the `v++` command provides information on every binary container in the application, as well as every compute unit in the design. The report is structured as follows:

- Target device information
- Summary of every kernel in the application
- Detailed information on every binary container in the solution

The following example report file represents the information generated for the estimate report:

```
-----  
Design Name: mmult.hw_emu.xilinx_u200_xdma_201830_2  
Target Device: xilinx:u200:xdma:201830.2  
Target Clock: 300.000000MHz  
Total number of kernels: 1  
-----  
  
Kernel Summary  
Kernel Name Type Target OpenCL Library Compute Units  
-----  
mmult c fpga0:OCL_REGION_0 mmult.hw_emu.xilinx_u200_xdma_201830_2 1  
-----  
  
OpenCL Binary: mmult.hw_emu.xilinx_u200_xdma_201830_2  
Kernels mapped to: clc_region  
  
Timing Information (MHz)  
Compute Unit Kernel Name Module Name Target Frequency Estimated Frequency  
-----  
mmult_1 mmult mmult 300.300293 411.015198  
-----  
  
Latency Information (clock cycles)  
Compute Unit Kernel Name Module Name Start Interval Best Case Avg Case Worst Case  
-----  
mmult_1 mmult mmult 826 ~ 829 825 827 828
```

Area Information							
Compute Unit	Kernel Name	Module Name	FF	LUT	DSP	BRAM	URAM
mmult_1	mmult	mmult	81378	35257	1036	2	0

## Design and Target Device Summary

All design estimate reports begin with an application summary and information about the target device. The device information is provided in the following section of the report:

Design Name:	mmult.hw_emu.xilinx_u200_xdma_201830_2
Target Device:	xilinx:u200:xdma:201830.2
Target Clock:	300.000000MHz
Total number of kernels:	1

For the design summary, the information provided includes the following:

- **Target Device:** Name of the Xilinx device on the target platform that runs the FPGA binary built by the Vitis compiler.
- **Target Clock:** Specifies the target operating frequency for the compute units (CUs) mapped to the FPGA fabric.

## Kernel Summary

This section lists all of the kernels defined for the application project. The following example shows the kernel summary:

Kernel Summary			OpenCL Library	Compute Units
Kernel Name	Type	Target		
mmult	c	fpga0:OCL_REGION_0	mmult.hw_emu.xilinx_u200_xdma_201830_2	1

In addition to the kernel name, the summary also provides the execution target and type of the input source. Because there is a difference in compilation and optimization methodology for OpenCL™, C, and C/C++ source files, the type of kernel source file is specified.

The Kernel Summary section is the last summary information in the report. From here, detailed information on each compute unit binary container is presented.

## Timing Information

For each binary container, the detail section begins with the execution target of all compute units (CUs). It also provides timing information for every CU. As a general rule, if the estimated frequency for the FPGA binary is higher than the target frequency, the CU will be able to run in the device. If the estimated frequency is below the target frequency, the kernel code for the CU needs to be further optimized to run correctly on the FPGA fabric. This information is shown in the following example:

```
OpenCL Binary : mmult.hw_emu.xilinx_u200_xdma_201830_2
Kernels mapped to: clc_region
```

Timing Information (MHz)				
Compute Unit	Kernel Name	Module Name	Target Frequency	Estimated Frequency
mmult_1	mmult	mmult	300.300293	411.015198

It is important to understand the difference between the target and estimated frequencies. CUs are not placed in isolation into the FPGA fabric. CUs are placed as part of a valid FPGA design that can include other components defined by the device developer to support a class of applications.

Because the CU custom logic is generated one kernel at a time, an estimated frequency that is higher than the target frequency indicates that the CU can run at the higher estimated frequency. Therefore, CU should meet timing at the target frequency during implementation of the FPGA binary.

## Latency Information

The latency information presents the execution profile of each CU in the binary container. When analyzing this data, it is important to recognize that all values are measured from the CU boundary through the custom logic. In-system latencies associated with data transfers to global memory are not reported as part of these values. Also, the latency numbers reported are only for CUs targeted at the FPGA fabric. The following is an example of the latency report:

Latency Information (clock cycles)						
Compute Unit	Kernel Name	Module Name	Start Interval	Best Case	Avg Case	Worst Case
mmult_1	mmult	mmult	826 ~ 829	825	827	828

The latency report is divided into the following fields:

- Start interval
- Best case latency
- Average case latency
- Worst case latency

The start interval defines the amount of time that has to pass between invocations of a CU for a given kernel.

The best, average, and worst case latency numbers refer to how much time it takes the CU to generate the results of one ND Range data tile for the kernel. For cases where the kernel does not have data dependent computation loops, the latency values will be the same. Data dependent execution of loops introduces data specific latency variation that is captured by the latency report.

The interval or latency numbers will be reported as "undef" for kernels with one or more conditions listed below:

- OpenCL kernels that do not have explicit `reqd_work_group_size(x, y, z)`
- Kernels that have loops with variable bounds

**Note:** The latency information reflects estimates based on the analysis of the loop transformations and exploited parallelism of the model. These advanced transformations such as pipelining and data flow can heavily change the actual throughput numbers. Therefore, latency can only be used as relative guides between different runs.

## Area Information

Although the FPGA can be thought of as a blank computational canvas, there are a limited number of fundamental building blocks available in each FPGA. These fundamental blocks (FF, LUT, DSP, block RAM) are used by the Vitis compiler to generate the custom logic for each CU in the design. The quantity of fundamental resources needed to implement the custom logic for a single CU determines how many CUs can be simultaneously loaded into the FPGA fabric. The following example shows the area information reported for a single CU:

Area Information							
Compute Unit	Kernel Name	Module Name	FF	LUT	DSP	BRAM	URAM
mmult_1	mmult	mmult	81378	35257	1036	2	0

---

## HLS Report

The HLS report provides details about the high-level synthesis (HLS) process of a user kernel and is generated during the compilation process for hardware emulation and system builds. This process translates the C/C++ and OpenCL kernel into the hardware description language used for implementing the kernel logic on the FPGA. The report provides estimated FPGA resource usage, operating frequency, latency, and interface signals of the custom-generated hardware logic. These details provide many insights to guide kernel optimization.

When running from the Vitis IDE, this report can be found in the following directory: `_x/<kernel_name>.<target>.<platform>/<kernel_name>/<kernel_name>/solution/syn/report`

The HLS report can be opened from the Vitis analyzer by opening the Compile Summary, or the Link Summary as described in [Section VIII: Using the Vitis Analyzer](#). An example of the HLS report is shown.

**Figure 35: HLS Report**

Name	Type	Latency	Latency (absolute) [us]	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	BRAM (%)	DSF
dct_2d		220	733.000				no	3	~0	4
dct_1d4	II Violation	12	39.996				function	0	0	4
Row_DCT_Loop				14		8	yes			
Xpose_Row_Outer_Loop_Xpose_Row_Inner_Loop				3		64	yes			
Col_DCT_Loop				14		8	yes			
Xpose_Col_Outer_Loop_Xpose_Col_Inner_Loop				3		64	yes			
read_data		105	350.000				no	0	0	0
RD_Loop_Row				6		8	yes			
write_data	II Violation	102	340.000			6	no	0	0	0
WR_Loop_Row						8	yes			

## Generating and Opening the HLS Report



**IMPORTANT!** You must specify the `--save-temp` 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. They are not generated for software emulation or RTL 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 [Section VIII: Using the Vitis Analyzer](#) 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
- Latency in clock cycles

- Latency in absolute time (ns)
- Iteration latency
- Iteration Interval
- Loop Tripcount
- Pipelined
- Utilization Estimates of BRAM, DSP, FF, and LUT
- Slack

If this information is part of a hierarchical block, it will sum up the information of the blocks contained in the hierarchy. Therefore, the hierarchy can also be navigated from within the report when it is clear which instance contributes to the overall design.



**CAUTION!** *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.

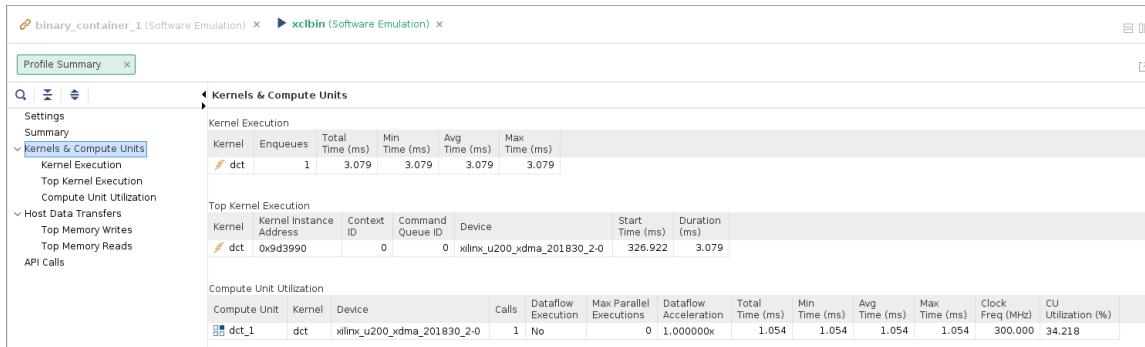
After the application finishes running, the Profile Summary report is saved as `.csv` files in the directory where the compiled host code is executed. The Profile Summary provides annotated details regarding the overall application performance. All data generated during the execution of the application 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. However, with the software emulation build, the report will not include any data transfer details under kernel execution efficiency and data transfer efficiency. This information is only generated in hardware emulation or system builds.*

An example of the Profile Summary report is shown below.

Figure 36: 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`:

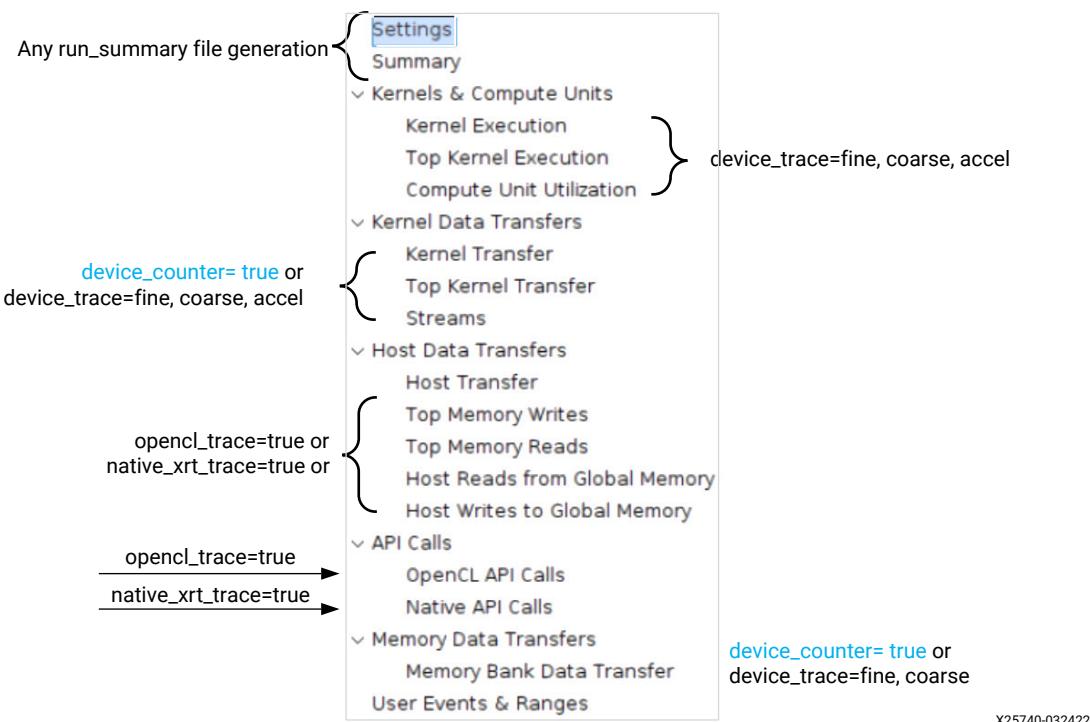
```
[Emulation]
debug_mode = batch
```

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 [Section VIII: Using the Vitis Analyzer](#) 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 the different data tables and the options to enable them.

Figure 37: Profile Summary Tables



X25740-032422

## Related Information

[Running the Application](#)

[Section VIII: Using the Vitis Analyzer](#)

# 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. The following tables show the profile summary descriptions.

## Settings

This displays the report and XRT configuration settings.

## Summary

This displays summary statistics including device execution time and device power.

## Kernels & Compute Units

The following table displays the profile summary data for all kernel functions scheduled and executed.

**Table 21: Kernel Execution**

Name	Description
Kernel	Name of kernel
Enqueues	Number of times kernel is enqueued. When the kernel is enqueued only once, the following stats are all the same.
Total Time	Sum of runtimes of all enqueues (measured from START to END in OpenCL execution model) (in ms)
Minimum Time	Minimum runtime of all enqueues
Average Time	Average kernel runtime (in ms) (Total time) / (Number of enqueues)
Maximum Time	Maximum runtime of all enqueues (in ms)

The following table displays the profile summary data for top kernel functions.

**Table 22: Top Kernel Execution**

Name	Description
Kernel	Name of kernel
Kernel Instance Address	Host address of kernel instance (in hex)
Context ID	Context ID on host
Command Queue ID	Command queue ID on host
Device	Name of device where kernel was executed (format: <device>-<ID>)
Start Time	Start time of execution (in ms)
Duration	Duration of execution (in ms)

This following table displays the profile summary data for all compute units on the device.

**Table 23: Compute Unit Utilization**

Name	Description
Compute Unit	Name of compute unit
Kernel	Kernel this compute unit is associated with
Device	Name of the device (format: <device>-<ID>)
Calls	Number of times the compute unit is called
Dataflow Execution	Specifies whether the CU is executed with dataflow
Max Parallel Executions	Number of executions in the dataflow region
Dataflow Acceleration	Shows the performance improvement due to dataflow execution
CU Utilization (%)	Shows the percent of the total kernel runtime that is consumed by the CU
Total Time	Sum of the runtimes of all calls (in ms)
Minimum Time	Minimum runtime of all calls (in ms)
Minimum runtime of all calls	(Total time) / (Number of work groups)
Maximum Time	Maximum runtime of all calls (in ms)
Clock Frequency	Clock frequency used for a given accelerator (in MHz)

This following table displays the profile summary data for running times and stalls for compute units on the device.

**Table 24: Compute Unit Running Times & Stalls**

Name	Description
Compute Unit	Name of compute unit
Execution Count	Execution count of the compute unit
Running Time	Total time compute unit was running (in $\mu$ s)
Intra-Kernel Dataflow Stalls (%)	Percent time the compute unit was stalling from intra-kernel streams
External Memory Stalls (%)	Percent time the compute unit was stalling from external memory accesses
Inter-Kernel Pipe Stalls (%)	Percent time the compute unit was stalling from inter-kernel pipe accesses

## Kernel Data Transfers

This following table displays the data transfer for kernels to the global memory.

**Table 25: Data Transfer**

Name	Description
Compute Unit Port	Name of compute unit/port
Kernel Arguments	List of kernel arguments attached to this port
Device	Name of device (format: <device>-<ID>)
Memory Resources	Memory resource accessed by this port
Transfer Type	Type of kernel data transfers
Number of Transfers	Number of kernel data transfers (in AXI transactions)  <b>Note:</b> This might contain printf transfers.
Transfer Rate	Rate of kernel data transfers (in MB/s): Transfer Rate = (Total Bytes) / (Total CU Execution Time) Where total CU execution time is the total time the CU was active
Bandwidth Utilization with regard to Current Port Configuration	Application bandwidth usage on this port with respect to the current configuration: Bandwidth Utilization (%) = (100 * Transfer Rate) / (Max Achievable BW) where Max Achievable BW is based on the bit-width of the port and the clock speed of the kernel in the design
Maximum Bandwidth with regard to Current Port Configuration	Maximum achievable bandwidth on the current port configuration: Bandwidth MB/s = (Current port bit width/8) * (Running PL clock rate in MHz)
Bandwidth Utilization with regard to Ideal Port Configuration	Application bandwidth usage against the maximum possible with ideal conditions: Bandwidth Utilization (%) = (100 * Transfer Rate) / (Max Possible BW) where Max Possible BW is based on the max bit-width of a port (512 bits) and the max clock speed of a kernel on this platform
Maximum Bandwidth with regard to Ideal Port Configuration	Maximum theoretical bandwidth on an ideal port configuration: Bandwidth MB/s = (Maximum possible port bit width/8) * (Highest possible PL clock rate in MHz)
Avg Size	Average size of kernel data transfers (in KB): Average Size = (Total KB) / (Number of Transfers)
Avg Latency	Average latency of kernel data transfers (in ns)

This following table displays the top data transfer for kernels to the global memory.

**Table 26: Top Data Transfer**

Name	Description
Compute Unit	Name of compute unit
Device	Name of device
Number of Transfers	Number of write and read data transfers
Avg Bytes per Transfer	Average bytes of kernel data transfers: Average Bytes = (Total Bytes) / (Number of Transfers)
Transfer Efficiency (%)	Efficiency of kernel data transfers: Efficiency = (Average Bytes) / min((Memory Byte Width * 256), 4096)
Total Data Transfer	Total data transferred by kernels (in MB): Total Data = (Total Write) + (Total Read)
Total Write	Total data written by kernels (in MB)
Total Read	Total data read by kernels (in MB)
Total Transfer Rate	Average total data transfer rate (in MB/s): Total Transfer Rate = (Total Data Transfer) / (Total CU Execution Time) Where total CU execution time is the total time the CU was active

This following table displays the data transfer streams.

**Note:** This table is only shown if there is stream data

**Table 27: Data Transfer Streams**

Name	Description
Master Port	Name of master compute unit and port
Master Kernel Arguments	List of kernel arguments attached to this port
Slave Port	Name of slave compute unit and port
Slave Kernel Arguments	List of kernel arguments attached to this port
Device	Name of device (format: <device>-<ID>)
Number of Transfers	Number of stream data packets
Transfer Rate	Rate of stream data transfers (in MB/s): Transfer Rate = (Total Bytes) / (Total CU Execution Time) Where total CU execution time is the total time the CU was active
Avg Size	Average size of kernel data transfers (in KB): Average Size = (Total KB) / (Number of Transfers)
Link Utilization (%)	Link utilization (%): Link Utilization = 100 * (Link Busy Cycles - Link Stall Cycles - Link Starve Cycles) / (Link Busy Cycles)
Link Starve (%)	Link starve (%): Link Starve = 100 * (Link Starve Cycles) / (Link Busy Cycles)
Link Stall (%)	Link stall (%): Link Stall = 100 * (Link Stall Cycles) / (Link Busy Cycles)

## Host Data Transfers

This following table displays profile data for all write transfers between the host and device memory through PCI Express® link.

**Table 28: Top Memory Writes**

Name	Description
Buffer Address	Specifies the address location for the buffer
Context ID	OpenCL Context ID on host
Command Queue ID	OpenCL Command queue ID on host
Start Time	Start time of write operation (in ms)
Duration	Duration of write operation (in ms)
Buffer Size	Amount of data being transferred (in KB)
Writing Rate	Data transfer rate (in MB/s): $(\text{Buffer Size}) / (\text{Duration})$

This following table displays profile data for all read transfers between the host and device memory through PCI Express® link.

**Table 29: Top Memory Reads**

Name	Description
Buffer Address	Specifies the address location for the buffer
Context ID	Context ID on host
Command Queue ID	Command queue ID on host
Start Time	Start time of read operation (in ms)
Duration	Duration of read operation (in ms)
Buffer Size	Amount of data being transferred (in KB)
Reading Rate	Data transfer rate (in MB/s): $(\text{Buffer Size}) / (\text{Duration})$

This following table displays the data transfer for host to the global memory.

**Table 30: Data Transfer**

Name	Description
Context:Number of Devices	Context ID and number of devices in context
Transfer Type	Type of kernel host transfers
Number of Buffer Transfers	Number of host buffer transfers  <i>Note:</i> This might contain printf transfers.
Transfer Rate	Rate of host buffer transfers (in MB/s): $\text{Transfer Rate} = (\text{Total Bytes}) / (\text{Total Time in } \mu\text{s})$

**Table 30: Data Transfer (cont'd)**

Name	Description
Avg Bandwidth Utilization (%)	Average bandwidth of host buffer transfers: Bandwidth Utilization (%) = $(100 * \text{Transfer Rate}) / (\text{Max. Theoretical Rate})$
Avg Size	Average size of host buffer transfers (in KB): Average Size = $(\text{Total KB}) / (\text{Number of Transfers})$
Total Time	Sum of host buffer transfer durations (in ms)
Avg Time	Average of host buffer transfer durations (in ms)

## API Calls

This following table 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.

**Table 31: API Calls**

Name	Description
API Name	Name of the API function (for example, <code>clCreateProgramWithBinary</code> , <code>clEnqueueNDRangeKernel</code> )
Calls	Number of calls to this API made by the host application
Total Time	Sum of runtimes of all calls (in ms)
Minimum Time	Minimum runtime of all calls (in ms)
Average Time	Average Time (in ms) (Total time) / (Number of calls)
Maximum Time	Maximum runtime of all calls (in ms)

## Device Power

This following table displays the profile data for device power.

**Table 32: Device Power**

Name	Description
Power Used By Platform	Shows a line graph of the three power rails on a Data Center acceleration card: <ul style="list-style-type: none"><li>• 12V Auxiliary</li><li>• 12V PCIe</li><li>• Internal power</li></ul> These show the power (W) usage of the card over time.
Temperature	One chart is created for each device that has non-zero temperature readings. Displays one line for each temperature sensor with readouts in (°C).
Fan Speed	One chart is created for each device that has non-zero fan speed readings. The fan speed is measured in RPM.

## Kernel Internals

This following table displays the running time for compute units in microseconds ( $\mu\text{s}$ ) and reports stall time as a percent of the running time.



**TIP:** The Kernel Internals tab reports time in  $\mu\text{s}$ , while the rest of the Profile Summary reports time in milliseconds (ms).

**Table 33: CU Runtime and Stalls**

Name	Description
Compute Unit	Indicates the compute unit instance name
Running Time	Reports the total running time for the CU (in $\mu\text{s}$ )
Intra-Kernel Dataflow Stalls (%)	Reports the percentage of running time consumed in stalls when streaming data between kernels
External Memory Stalls (%)	Reports the percentage of running time consumed in stalls for memory transfers outside the CU
Inter-Kernel Pipe Stalls (%)	Reports the percentage of running time consumed in stalls when streaming data to or from outside the CU

This following table displays the data transfer for specific ports on the compute unit.

**Table 34: CU Port Data Transfers**

Name	Description
Port	Indicates the port name on the compute unit
Compute Unit	Indicates the compute unit instance name
Write Time	Specifies the total data write time on the port (in $\mu\text{s}$ )
Outstanding Write (%)	Specifies the percentage of the runtime consumed in the write process
Read Time	Specifies the total data read time on the port (in $\mu\text{s}$ )
Outstanding Read (%)	Specifies the percentage of the runtime consumed in the read process

This following table displays the functional port data transfers on the compute unit.

**Table 35: Functional Port Data Transfers**

Name	Description
Port	Name of port
Function	Name of function
Compute Unit	Name of compute unit
Write Time	Total time the port had an outstanding write (in $\mu\text{s}$ )
Outstanding Write (%)	Percent time the port had an outstanding write
Read Time	Total time the port had an outstanding read (in $\mu\text{s}$ )
Outstanding Read (%)	Percent time the port had an outstanding read

This following table displays the running time and stalls on the compute unit.

**Table 36: Functions**

Name	Description
Compute Unit	Name of compute unit
Function	Name of function
Running Time	Total time function was running (in ms)
Intra-Kernel Dataflow Stalls	Percent time the function was stalling from intra-kernel streams (in ms)
External Memory Stalls	Percent time the function was stalling from external memory accesses (in ms)
Inter-Kernel Pipe Stalls	Percent time the function was stalling from inter-kernel pipe accesses (in ms)

## Shell Data Transfers

This following table displays the DMA data transfers.

**Table 37: DMA Data Transfer**

Name	Description
Device	Name of device (format: <device>-<ID>)
Transfer Type	Type of data transfers
Number of Transfers	Number of data transfers (in AXI transactions)
Transfer Rate	Rate of data transfers (in MB/s): $\text{Transfer Rate} = (\text{Total Bytes}) / (\text{Total Time in } \mu\text{s})$
Total Data Transfer	Total amount of data transferred (in MB)
Total Time	Total duration of data transfers (in ms)
Avg Size	Average size of data transfers (in KB): $\text{Average Size} = (\text{Total KB}) / (\text{Number of Transfers})$
Avg Latency	Average latency of data transfers (in ns)

For DMA bypass and Global Memory to Global Memory data transfers, see the DMA Data Transfer table above.

## NoC Counters



**TIP:** This data is not displayed unless it has been specifically generated during implementation.

NoC Counters display the NoC Counters Read and NoC Counters Write. These sections are only displayed if there is a non-zero NoC counter data.

Each section has a table containing summary data with line graphs for transfer rate and latency. The graphs can have multiple NoC counters, so you can toggle the counters ON/OFF through check boxes in the Chart column of the table.

Depending on the design, it can be possible to correlate NoC counters to CU ports. In this case, the CU port appears in the table, and selecting it cross-probes to the system diagram, profile summary, and any other views that include CU ports as selectable objects.

**Table 38: NoC Counters Read or Write**

Name	Description
Compute Unit Port	Name of compute unit/port
Name	Name of NoC port
Traffic Class	Traffic class type
Requested QoS	QoS (MB/s): Requested quality of service (in MB/s)
Min Transfer Rate	Rate of minimum data transfers (in MB/s)
Avg Transfer Rate	Rate of average data transfers (in MB/s)
Max Transfer Rate	Rate of maximum data transfers (in MB/s)
Avg Size	Average size of data transfers (in KB): Average Size = (Total KB) / (Number of Transfers)
Min Latency	Minimum latency of data transfers (in ns)
Avg Latency	Average latency of data transfers (in ns)
Max Latency	Maximum latency of data transfers (in ns)

## AI Engine Counters

AI Engine counters display if there is a non-zero AI Engine counter data. If there is an incompatible configuration of the AI Engine counters, this section displays a message stating that the configuration does not support performance profiling.

This section has a table containing summary data with line graphs for active time and usage. The usage chart is only available if stall profiling is enabled.

The graphs can have multiple AI Engine counters, so you can toggle the counters ON/OFF through check boxes in the Chart column of the table.

It is possible to cross-probe tiles to the AI Engine array and graph views.

**Note:** Depending on how the AI Engine counters are configured, one or more metric columns might appear. These include memory stall, stream stall, call inst time, group error time, etc. For more information, see *AI Engine Tools and Flows User Guide* ([UG1076](#)).

**Table 39: AI Engine Counters**

Name	Description
Tile	AI Engine Tile [Column, Row]
Clock Frequency (MHz)	Frequency (in MHz) of clock used for AI Engine tiles

## Timeline Trace

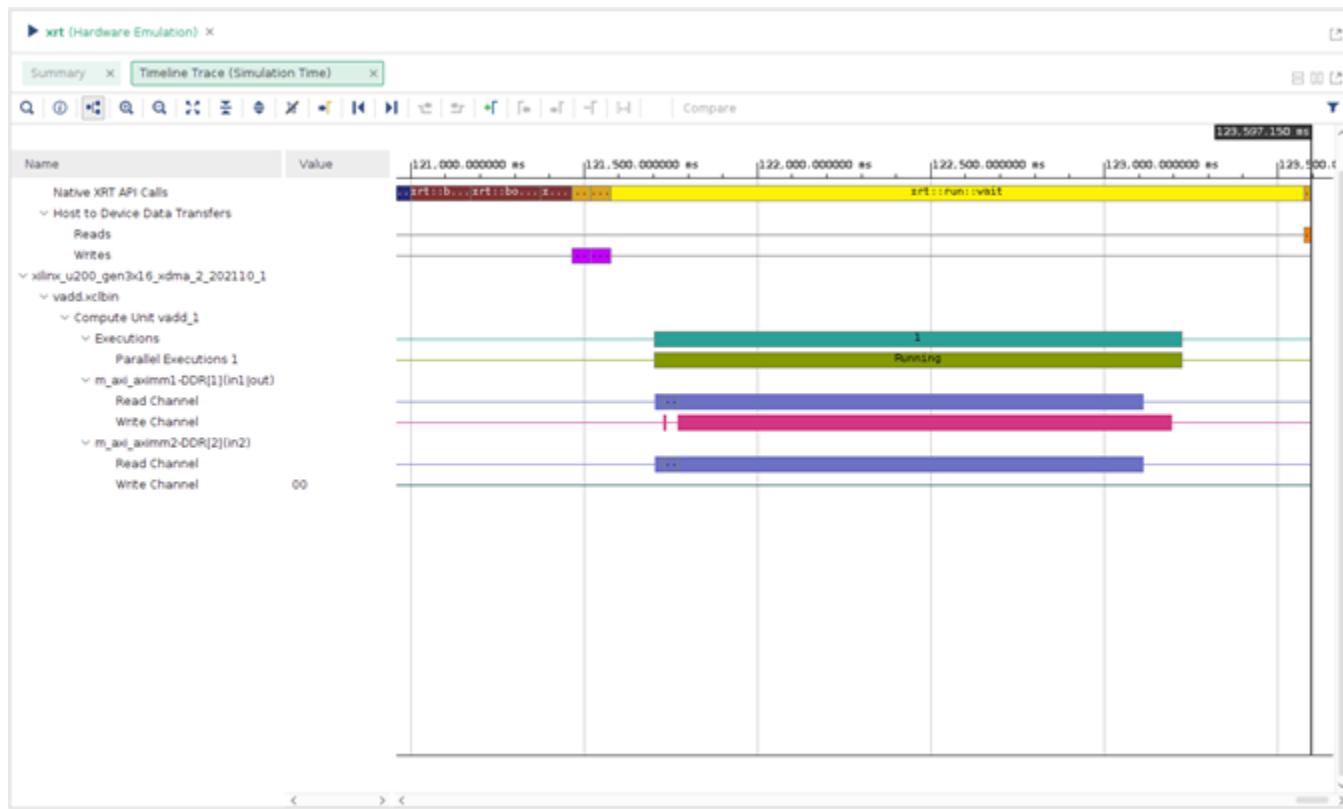
The Timeline Trace collects and displays host and kernel events on a common timeline to help you understand and visualize the overall health and performance of your systems. The graphical representation lets you see issues regarding kernel synchronization and efficient concurrent execution. The displayed 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 38: 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 [Section VIII: Using the Vitis Analyzer](#).

## 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 should not be confused with the kernels/CUs on the device. Here kernel refers to the `NDRangeKernels` and tasks created by the OpenCL commands `clEnqueueNDRangeKernels` and `clEnqueueTask`. These are plotted against the time measured from the host's perspective. Multiple kernels can be scheduled to be executed at the same time, and they are traced from the point they are scheduled to run until the end of the kernel execution. This is the reason for multiple entries. The number of rows depend on the number of overlapping kernel executions.

**Note:** Overlapping of the kernels should not be mistaken for actual parallel execution on the device as the process might not be ready to execute right away.

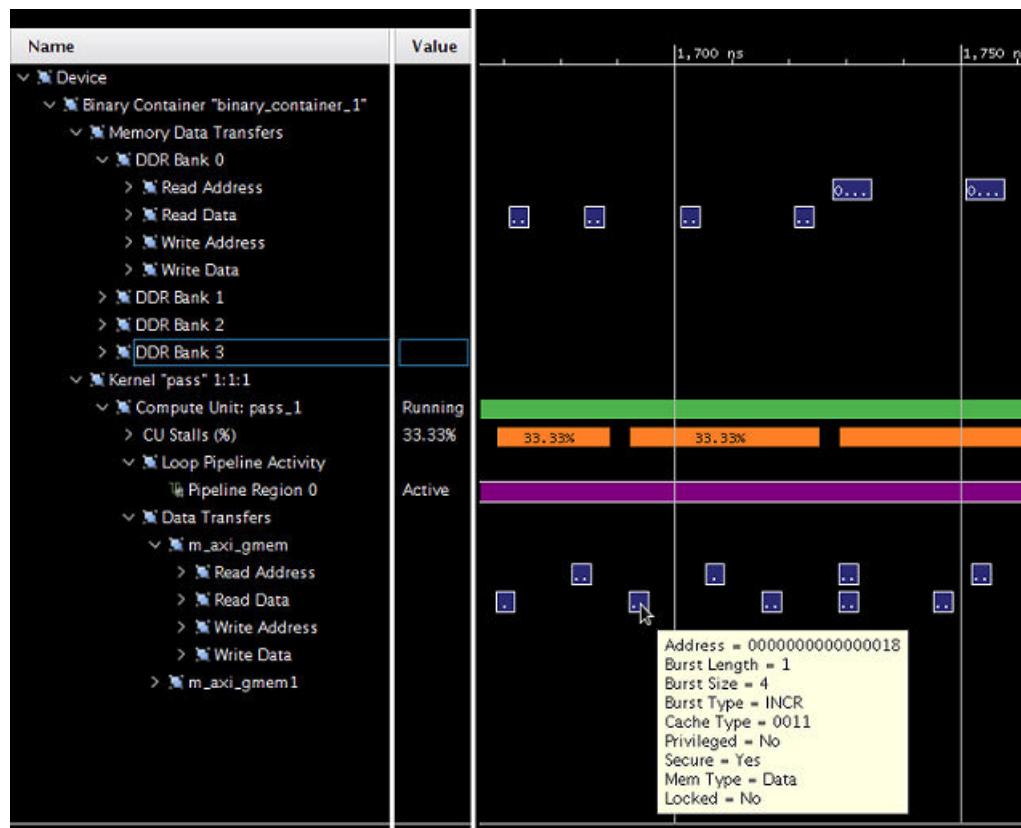
- **Device "name"**
  - **Binary Container "name":** Binary container name.
  - **Compute Unit "name":** Name of the compute unit on the FPGA.
    - **User Functions:** In the case of the Vitis HLS tool kernels, functions that are implemented as data flow processes are traced here. The trace for these 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, as well as the Kernel side (Kernel "pass" 1:1:1) of the AXI interconnect. The following figure illustrates a typical kernel trace of a newly accelerated algorithm.

Figure 39: Accelerated Algorithm Kernel Trace



Most interesting with respect to performance are the fields:

- **Burst Length:** Describes how many packages are sent within one transaction.
- **Burst Size:** Describes the number of bytes being transferred as part of one package.

Given a burst length of 1 and just 4 bytes per package, it will require many individual AXI transactions to transfer any reasonable amount of data.

**Note:** The Vitis core development kit never creates burst sizes less than 4 bytes, even if smaller data is transmitted. In this case, if consecutive items are accessed without AXI bursts enabled, it is possible to observe multiple AXI reads to the same address.

Small burst lengths, as well as burst sizes, considerably less than 512 bits are therefore good opportunities to optimize interface performance.

## Using Burst Data Transfers

Transferring data in bursts hides the memory access latency and improves bandwidth usage and efficiency of the memory controller. Also, check the HLS report for bursting information.



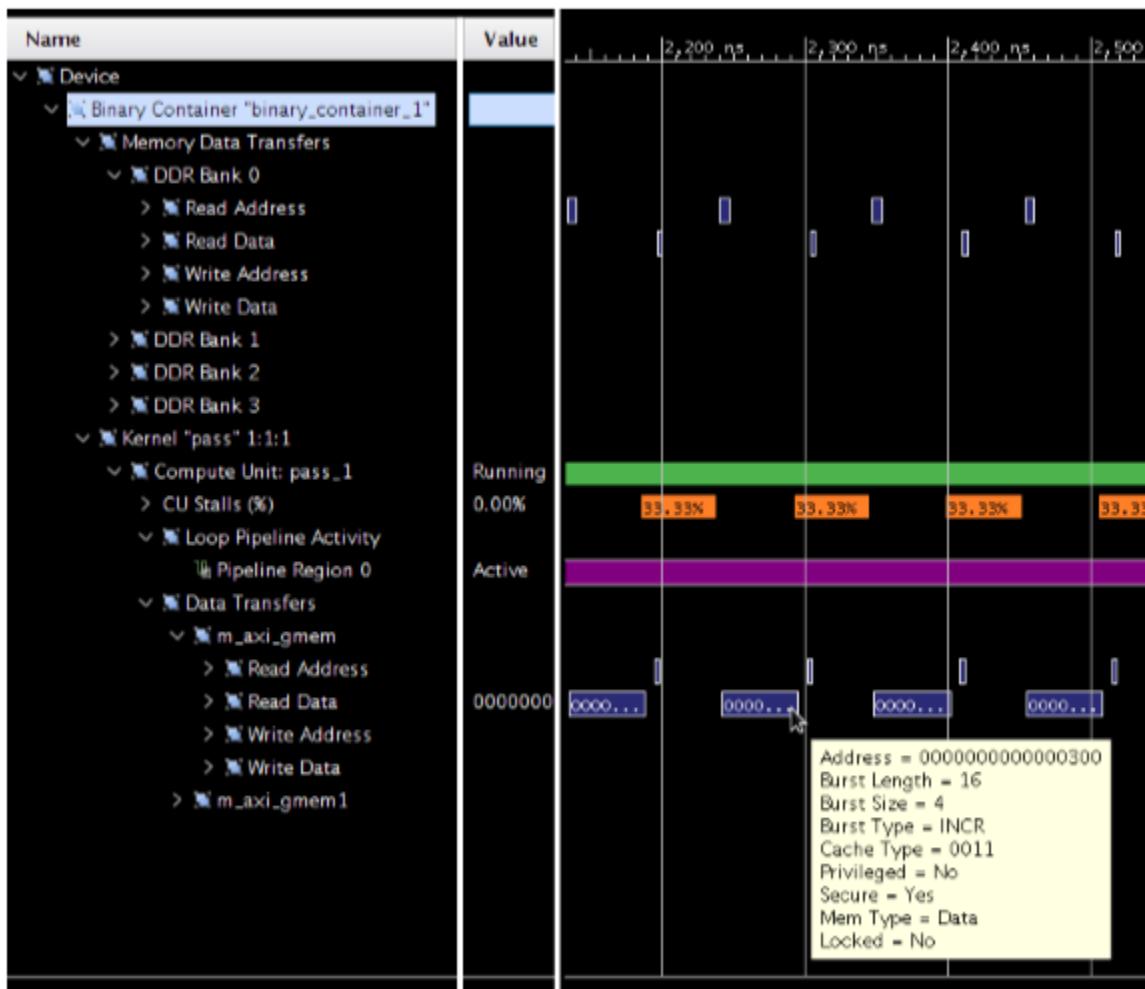
---

**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 40: 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 tmpOut[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 the [AXI Burst Transfers](#) in the *Vitis HLS User Guide* (UG1399).

The isolation of the read and write function from the computation results in:

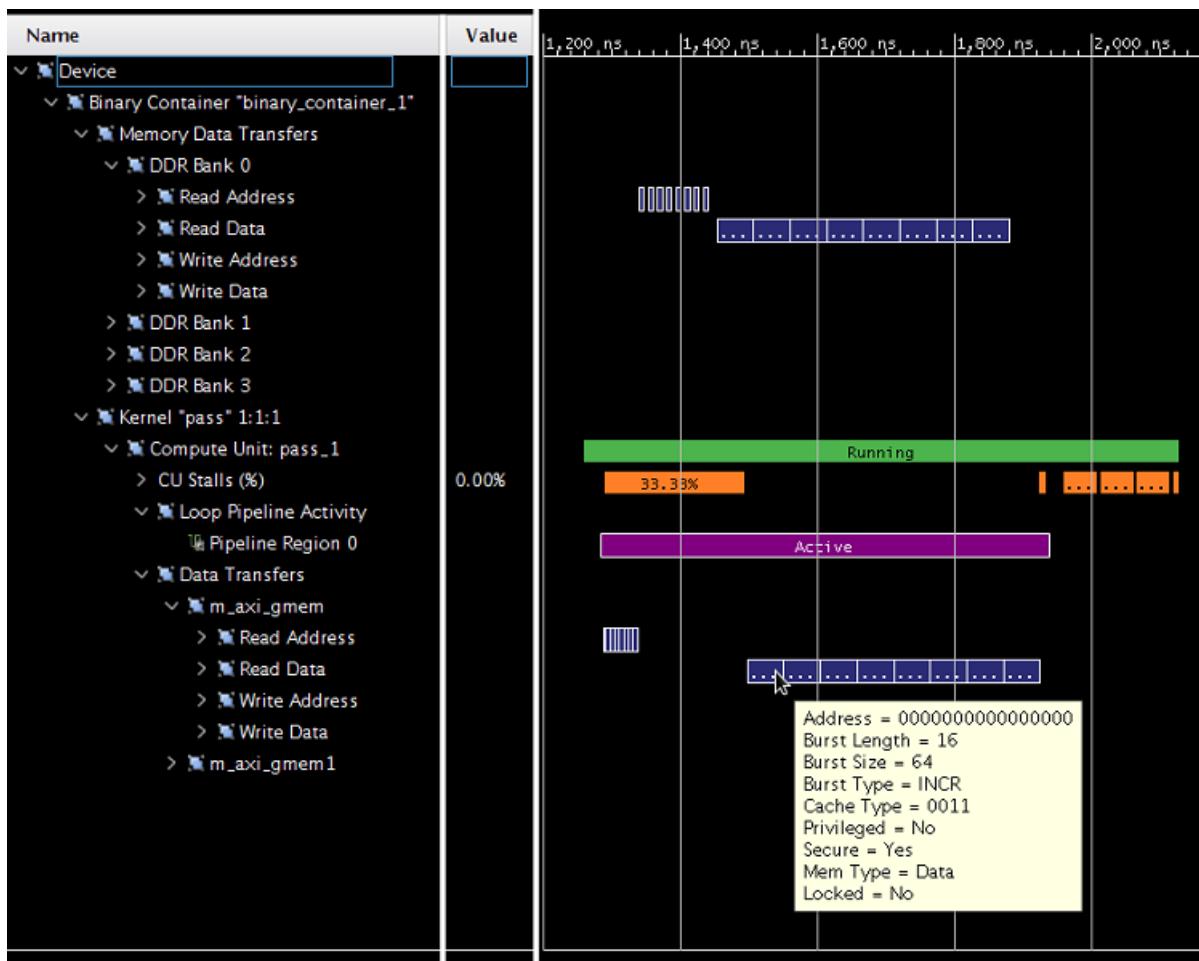
- Simple control structures (loops) in the read/write function which makes burst detection simpler.
- The isolation of the computational function away from the AXI interfaces, simplifies potential kernel optimization.
- The internal variables are mapped to on-chip memory, which allow faster access compared 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, Xilinx recommends that you choose data types map to the full data width on the memory controller. The memory controller in all supported acceleration cards supports 512-bit user interface, which can be mapped to 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 41: Burst AXI Transactions



This example shows good interface configuration as it maximizes AXI data width as well as actual burst transactions.

Complex structs or classes, used to declare interfaces, can lead to very complex hardware interfaces due to memory layout and data packing differences. This can introduce potential issues that are very difficult to debug in a complex system.



**RECOMMENDED:** Use simple structs for kernel arguments that can be packed to 32-bit boundary.

# Waveform View and Live Waveform Viewer

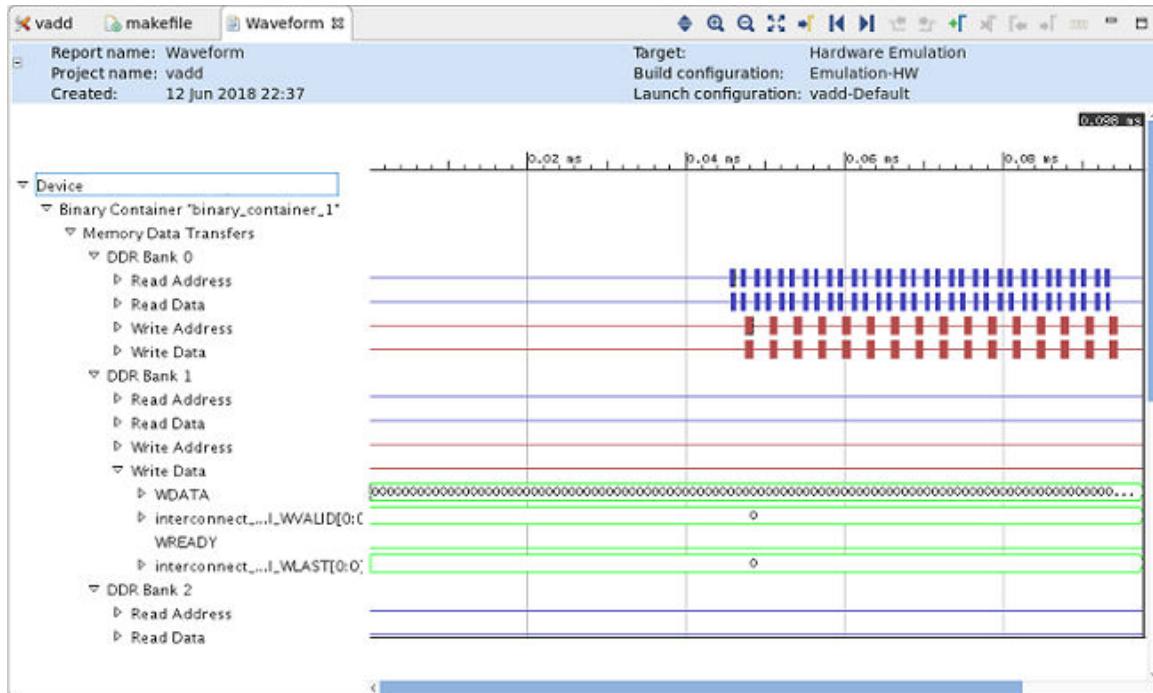
The Vitis core development kit can generate a Waveform view when running hardware emulation. It displays in-depth details at the system-level, CU level, and at the function level. The details include data transfers between the kernel and global memory and data flow through inter-kernel pipes. These details provide many insights into performance bottlenecks from the system-level down to individual function calls to help optimize your application.

The Live Waveform Viewer is similar to the Waveform view, however, it provides even lower-level details with some degree of interactivity. The Live Waveform Viewer can also be opened using the Vivado logic simulator, `xsim`.

**Note:** The Waveform view allows you to examine the device transactions from within the Vitis analyzer, as described in [Section VIII: Using the Vitis Analyzer](#). 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 42: 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 [Section V: Simulating the Application with the 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](#) or [Output Directories from the Vitis IDE](#) 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.

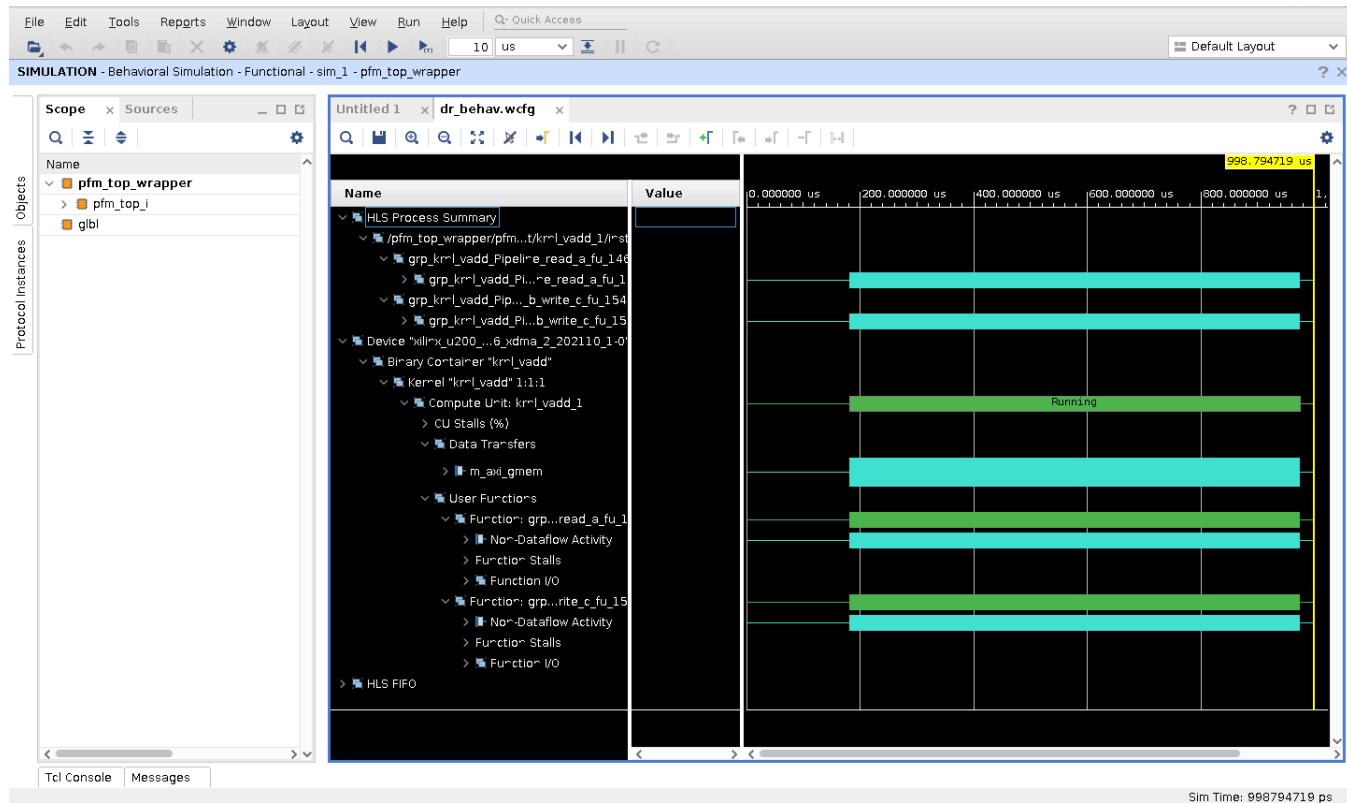
```
vitis_analyzer xrt.run_summary
```

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 43: 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-pipeline processes
- The transactions on the processes including stalls, are shown using the corresponding protocol analyzer instance
- Allows you to get an overview about the utilization 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 the Vitis HLS tool to inform you when a portion of the circuit is stalling because of external memory accesses, internal streams (that is, dataflow), or external streams (that is, OpenCL pipes). The stall bus shown in detailed kernel trace compiles all of the lowest level stall signals and reports the percentage that are stalling at any point in time. This provides a factor of how much of the kernel is stalling at any point in the simulation.

For example, if there are 100 lowest level stall signals and 10 are active on a given clock cycle, then the CU Stall percentage is 10%. If one goes inactive, then it is 9%.

- **Data Transfers:** This shows the read/write data transfer accesses originating from each Master AXI port of the compute unit to the DDR.
- **User Functions:** This information is available for the HLS kernels and shows the user functions.
  - **Function: "name":** Function name.
    - **Dataflow/Pipeline Activity:** This shows the number of parallel executions of the function if the function is implemented as a dataflow process.
    - **Active Iterations:** This shows the currently active iterations of the dataflow. The number of rows is dynamically incremented to accommodate the visualization of any concurrent execution.
    - **StallNoContinue:** This is a stall signal that tells if there were any output stalls experienced by the dataflow processes (function is done, but it has not received a continue from the adjacent dataflow process).

- **RTL Signals:** These are the underlying RTL control signals that were used to interpret the above transaction view of the dataflow process.
  - **Function Stalls:** Shows the different types of stalls experienced by the process.
    - **External Memory:** Stalls experienced while accessing the DDR memory.
    - **Internal-Kernel Pipe:** If the compute units communicated between each other through pipes, then this 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 instance
    - **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 Applications and Kernels

The Vitis™ unified software platform provides application-level debug features and techniques that allow the host code, kernel code, and the interactions between them to be debugged. These features and techniques are split between software debugging and hardware debugging flows.

For software debugging, the host and kernel code can be debugged using the Vitis IDE, or using GDB from the command line as a standard debug tool.

For hardware debugging, kernels running on hardware can be debugged using Xilinx® virtual cable (XVC) running over the PCIe® bus, for Alveo™ Data Center accelerator cards, and debugged using USB-JTAG cables for both Alveo cards and embedded processor platforms.

---

## Debugging Flows

The Vitis unified software platform provides application-level debug features which allow the host code, the kernel code, and the interactions between them to be efficiently debugged in either the Vitis IDE, or from the command line. The recommended debugging flow consists of three levels of debugging:

- [Debugging in Software Emulation](#) to confirm the algorithm functionality of the application as represented in both your host program and kernel code.
- [Debugging in Hardware Emulation](#) to compile the kernel into RTL, confirm the behavior of the generated logic, and evaluate the simulated performance of the hardware.
- [Debugging During Hardware Execution](#) to implement the FPGA binary and debug the application running in hardware.

This three-tiered approach allows debugging the host and kernel code, and the interactions between them at different levels of abstraction. Each provides specific insights into the design and makes debugging easier. All flows are supported through an integrated GUI flow as well as through a batch flow using basic compile time and runtime setup options.

In the case of applications running on embedded processor platforms, some additional setup is required as described in [Debugging on Embedded Processor Platforms](#).

# Debugging in Software Emulation

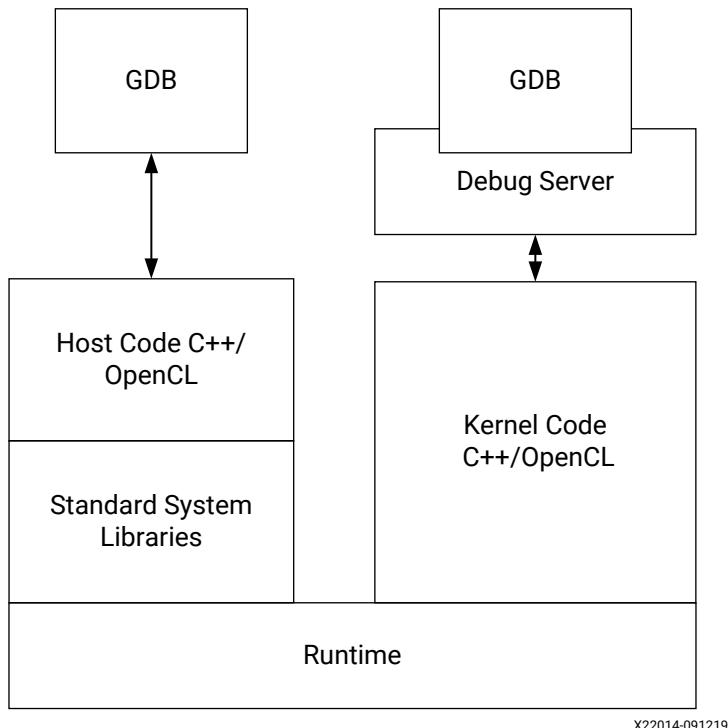


**IMPORTANT!** The following steps describe debugging from the command line. However, the Vitis IDE offers a standalone debug environment for use with the Vitis application acceleration projects created from the command line. Refer to [Using the Standalone Debug Flow](#) for more information.

The Vitis unified software platform supports typical software debugging for the host code at all times, the kernel code when running in software emulation mode, and at points during hardware emulation mode. This is a standard software debug flow using breakpoints, stepping through code, analyzing variables, and forcing the code into specific states.

The following figure shows the debug flow during software emulation for the host and kernel code (written in C/C++ or OpenCL™) using the GNU debugging (GDB) tool. Notice the two instances of GDB to separately debug the host and kernel processes, and the use of the debug server (xrt\_server).

Figure 44: Software Emulation



Xilinx recommends iterating the design as much as possible in Software Emulation, which takes little compile time and executes quickly. For more detailed information on software emulation, see [Software Emulation](#).

## GDB-Based Debugging



**IMPORTANT!** Both the host and kernel code must be compiled for debugging using the `-g` option.

For the GNU debugging (GDB), you can debug the kernel or host code, adding breakpoints, and inspecting variables. This familiar software debug flow allows quick design, compile, and debug to validate the functionality of your application. The Vitis debugger also provides extensions to GDB to let you examine the content of the Xilinx Runtime (XRT) library from the host program. These extensions can be used to debug protocol synchronization issues between the host and the kernel.

The Vitis core development kit supports GDB host program debugging in all flows, but kernel debugging is limited to the software emulation mode. Debugging features need to be enabled in your host and kernel code by using the `-g` option during compilation and linking.

This section shows how host and kernel debugging can be performed with the help of GDB. Because this flow should be familiar to most software developers, this section focuses on the extensions of host code debugging capabilities for the XRT library and the requirements of kernel debug.

### Xilinx Runtime Library GDB Extensions

The Vitis debugger (`xgdb`) enables new GDB commands that give you visibility from the host application into the XRT library.

**Note:** If you launch GDB outside of the Vitis debugger, the command extensions need to be enabled using the `appdebug.py` script as described in [Launching Host and Kernel Debug](#).

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

1. `xprint` commands that give visibility into XRT library data structures (`cl_command_queue`, `cl_event`, and `cl_mem`). These commands are explained below.
2. `xstatus` commands that give visibility into IP running on the Vitis target platform when debugging during hardware execution.

You can get more information about the `xprint` and `xstatus` commands by using the `help <command>` from the `gdb` command prompt.

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

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

## xprint Commands

The arguments to `xprint queue` and `xprint mem` are optional. The application debug environment keeps track of all the XRT library objects and automatically prints all valid queues and `cl_mem` objects if the argument is not specified. In addition, the commands do a proper validation of supplied command `queue`, `event`, and `cl_mem` arguments.

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

## xstatus Commands

This functionality is only available in the system flow (hardware execution) and not in any of the emulation flows.

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

## GDB Kernel-Based Debugging

GDB kernel debugging is supported for the software emulation flow. When the GDB executable is connected to the kernel in the IDE or command line flows, you can set breakpoints and query the content of variables in the kernel, similar to normal host code debugging. This is fully supported in the software emulation flow because the kernel GDB processes attach to the spawned software processes.

## Command Line Debug Flow



**TIP:** Set up the command shell or window as described in [Setting Up the Vitis Environment](#) prior to running the tools.

The following describes the steps required to run the debug flow in software emulation from the command line. Refer to [Section IX: Using the Vitis IDE](#) for information on debugging in the IDE. Debugging in the Vitis core development kit uses the following steps:

1. Compiling and linking the host code for debugging by adding the `-g` option to the `g++` command line as described in [Building the Software Application](#).

2. Compiling and linking the kernel code for debugging by adding the `-g` option to the `v++` command line as described in [Building the Device Binary](#).  
**Note:** When debugging OpenCL kernels, there are additional steps that you can take during compiling and linking as described in [Debugging OpenCL Kernels](#).
3. Launching GDB to debug the application. There are currently two flows supporting debug during software emulation:
  - a. Debugging PL or AI Engine kernels using the `kernel-dbg` option as explained in [Software Emulation Debug for Embedded Processors](#), or [Software Emulation Debug for Alveo Accelerators](#). This is a simple flow, but only enables debugging of the kernels in the `.xclbin`, and not the host application.
  - b. Debugging the host application and PL kernels using three command terminals and `xrt-server` as described in [Launching Host and Kernel Debug](#). This is a more complex flow, but supports concurrent debug of both the host and kernels.

## ***Software Emulation Debug for Embedded Processors***

You can use the following process to debug the system in software emulation for an embedded processor platform, such as a Versal® VCK190 or Zynq® UltraScale+™ MPSoC ZCU102 or ZCU102. This process launches a new terminal that allows for `gdb` commands to be used, as well as visual of the files for code stepping.

1. After completing the `sw_emu` build process, launch the QEMU emulation environment with debug using the following command from your build directory:

```
./emulation/launch_sw_emu.sh -kernel-dbg true
```

Where:

- `./emulation` is the output directory of the `v++ --package` command.
- `-kernel-dbg true` sets up the emulator to run `gdb` at the execution of the application kernels (PL, AI Engine).

2. As described in [Running Emulation on an Embedded Processor Platform](#), run the following commands in the QEMU shell once you see the `qemu%` prompt:

```
mount /dev/mmcblk0p1 /mnt
cd /mnt
export LD_LIBRARY_PATH=/mnt:/tmp:$LD_LIBRARY_PATH
export XCL_EMULATION_MODE=sw_emu
export XILINX_XRT=/usr
```



**TIP:** There might be additional steps required if your design incorporates AI Engine kernels. For more information, refer to the AI Engine Tools and Flows User Guide ([UG1076](#)).

3. Run the PS application from the QEMU environment. For example: `./host.exe a.xclbin`

Launching the host application also launches `gdb` in a separate terminal to let you debug the PL and AI Engine kernels in the `.xclbin`. In `gdb`, you can perform all the typical activities to set up your debug environment, such as breakpoint insertion for PL and AI Engine kernels, and step or continue commands to walk through the code.

4. You can set a breakpoint on either function name or line number in `gdb` using this syntax. During execution, when it reaches that breakpoint, `gdb` automatically opens the file with the right line number, as long as it can locate the sources. For example,

```
break <filename>:<function name>
break <filename>:<line_num>
```

Also, you can set a breakpoint for a kernel using a command such as `b Vadd_A_B`. This command pauses `.xclbin` execution in `gdb` at the invocation of the specified kernel, in this example the `Vadd_A_B` kernel. In an `.xclbin` file with multiple kernels, you can add breakpoints for all or specific kernels.

**Note:** When you set the breakpoint in `gdb`, you will see a note that the function is not defined and prompting you to make the breakpoint pending on future load. Enter `y` to proceed.

5. In the `gdb` terminal, press `r` to run the application and step through the code, set variable values, and continue.



**TIP:** To use a Textual User Interface (TUI) in `gdb`, use the following keystrokes:

```
Ctrl+x Ctrl+a
```

6. If the TUI is displayed, the source code for the kernel is displayed, as well as the path to the source code defined in the `.xclbin` by the software emulation build process.
7. Step through the code, or press `c` to let the code run through to completion. The results of your software emulation are displayed in the original command terminal.
8. Press `q` to quit `gdb`.
9. Close the `gdb` command terminal.

## ***Software Emulation Debug for Alveo Accelerators***

You can use the following process to debug the system in software emulation for an Alveo™ accelerator card. This process launches a new terminal that allows for `gdb` commands to be used, as well as displaying the source files for code stepping.

1. Complete the `sw_emu` build process, using the `-g` or `--debug` options when building the host and kernels.
2. Add the `kernel-dbg=true` option to the `xrt.ini` file prior to running the application:

```
[Emulation]
kernel-dbg=true
```

3. As described in [Running Emulation on Data Center Accelerator Cards](#), set the XCL\_EMULATION\_MODE to sw\_emu:

```
setenv XCL_EMULATION_MODE sw_emu
```

4. Launch the host application with the .xclbin file: ./host.exe a.xclbin

Launching the host application also launches `gdb` in a separate terminal to let you debug the PL kernels in the .xclbin. In `gdb`, you can perform all the typical activities to set up your debug environment, such as breakpoint insertion for PL kernels, and step or continue commands to walk through the code.

5. You can set a breakpoint on either function name or line number in `gdb` using this syntax. During execution, when it reaches that breakpoint, `gdb` automatically opens the file with the right line number, as long as it can locate the sources. For example,

```
break <filename>:<function name>
break <filename>:<line_num >
```

Also, you can set a breakpoint for a kernel using a command such as `b Vadd_A_B`. This command pauses .xclbin execution in `gdb` at the invocation of the specified kernel, in this example the `Vadd_A_B` kernel. In an .xclbin file with multiple kernels, you can add breakpoints for all or specific kernels.

**Note:** When you set the breakpoint in `gdb`, you will see a note that the function is not defined and prompting you to make the breakpoint pending on future load. Enter `y` to proceed.

6. In the `gdb` terminal, press `r` to run the application and step through the code, set variable values, and continue.



**TIP:** To use a Textual User Interface (TUI) in `gdb`, use the following keystrokes:

```
Ctrl+x Ctrl+a
```

7. If the TUI is displayed, the source code for the kernel is displayed, as well as the path to the source code defined in the .xclbin by the software emulation build process.
8. Step through the code, or press `c` to let the code run through to completion. The results of your software emulation are displayed in the original command terminal.
9. Press `q` to quit `gdb`.
10. Close the `gdb` command terminal.

## 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
```

Then 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](#) for more information on the `--config` option.

## Launching Host and Kernel Debug

In software emulation, to better model the hardware accelerator, the execution of the FPGA binary is spawned as a separate process. If you are using GDB to debug the host code, breakpoints set in kernel code are not encountered because the kernel code is not run within the host code process. To support the concurrent debugging of the host and kernel code, the Vitis debugger provides a system to attach to spawned kernels through the use of the debug server (`xrt_server`). To connect the host and kernel code to the debug server, you must open three terminal windows using the following process.



**TIP:** This flow should also work while using a graphical front-end for GDB, such as the data display debugger (DDD) available from GNU. The following steps are the instructions for launching GDB.

1. Open three terminal windows, and set up each window as described in [Setting Up the Vitis Environment](#). The three windows are for:
  - Running `xrt_server`
  - Running GDB (`xgdb`) on the Host Code
  - Running GDB (`xgdb`) on the Kernel Code
2. In the first terminal, after setting up the terminal environment, start the Vitis debug server using the following command:

```
xrt_server --sdx-url
```

The debug server listens for debug commands from the host and kernel, connecting the two processes to create a single debug environment. The `xrt_server` returns a listener port `<num>` on standard out. To control this process, you must start new GDB instances and connect to the `xrt_server` through the listener port. This is done in the next steps.



**TIP:** The initial listener port should not be connected to as it is used for connection by TCF agents such as the Vitis IDE. In this command-line flow the first listener port should be ignored.



**IMPORTANT!** With the `xrt_server` running, all spawned GDB processes wait for control from you. If no GDB ever attaches to the `xrt_server`, or provides commands, the kernel code appears to hang.

3. In a second terminal, after setting up the terminal environment, launch GDB for the host code as described in the following steps:

- a. Set the `ENABLE_KERNEL_DEBUG` environment variable. For example, in a C-shell use the following:

```
setenv ENABLE_KERNEL_DEBUG true
```

- b. Set the `XCL_EMULATION_MODE` environment variable to `sw_emu` mode as described in [Running the Application](#). For example, in a C-shell use the following:

```
setenv XCL_EMULATION_MODE sw_emu
```

- c. The runtime debug feature must be enabled using an entry in the `xrt.ini` file, as described in [xrt.ini File](#). Create an `xrt.ini` file in the same directory as your host executable, and include the following lines:

```
[Debug]
app_debug=true
```

This informs the runtime library that the kernel has been compiled for debug, and that XRT library should enable debug features.

- d. Start `gdb` through the Xilinx wrapper:

```
xgdb --args <host> <xclbin>
```

Where `<host>` is the name of your host executable, and `<xclbin>` is the name of the FPGA binary. For example:

```
xgdb --args host.exe vadd.xclbin
```

Launching GDB from the `xgdb` wrapper performs the following setup steps for the Vitis debugger:

- Loads GDB with the specified host program.
- Sources the Python script from the GDB command prompt to enable the Vitis debugger extensions:

```
gdb> source ${XILINX_XRT}/share/appdebug/appdebug.py
```

When you run the host application in `gdb`, it spawns off the software emulation process. The software emulation process detects that `xrt_server` is running and connects to it. At that point, a listener port for the kernel `gdb` is created and the software emulation process waits until it receives commands. Now, you should connect the kernel `gdb` to the `xrt_server` listener port as described in the next step.

4. In a third terminal, after setting up the terminal environment, launch the `xgdb` command, and run the following commands from the (`gdb`) prompt:

- For software emulation:

```
file <Vitis_path>/data/emulation/unified/cpu_em/generic_pcie/model/  
genericpciemodel
```

Where `<Vitis_path>` is the installation path of the Vitis core development kit. Using the `$XILINX_VITIS` environment variable does not work inside GDB.

- Connect to the kernel process:

```
target remote :<num>
```

Where `<num>` is the listener port number returned by the `xrt_server`.

With the three terminal windows running the `xrt_server`, GDB for the host, and GDB for the kernels, you can set breakpoints on your host or kernels as needed, run the `continue` command, and debug your application. When the all kernel invocations have finished, the host code continues and the `xrt_server` connection drops.

## Using `printf()` or `cout` to Debug Kernels

The basic approach to debugging algorithms is to verify key code steps and key data values throughout the execution of the program. For application developers, printing checkpoint statements, and outputting current values in the code is a simple and effective method of identifying issues within the execution of a program. This can be done using the `printf()` function, or `cout` for standard output.

### C/C++ Kernel

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

```
#ifndef __SYNTHESIS__  
    printf("Checkpoint 1 reached");  
#endif
```

For C++ kernels, you can also use `cout` in your code to add checkpoints or messages used for debugging the code. For example, you might add the following:

```
std::cout << "TEST " << (match ? "PASSED" : "FAILED") << std::endl;
```

## OpenCL Kernel

The Xilinx Runtime (XRT) library supports the OpenCL™ `printf()` built-in function within kernels in all build configurations: software emulation, hardware emulation, and during hardware execution.



**TIP:** The `printf()` function is only supported in all build configurations for OpenCL kernels. For C/C++ kernels, `printf()` is only supported in software emulation.

The following is an example of using `printf()` in the kernel, and the output when the kernel is executed with `global size` of 8:

```
__kernel __attribute__((reqd_work_group_size(1, 1, 1)))
void hello_world(__global int *a)
{
    int idx = get_global_id(0);

    printf("Hello world from work item %d\n", idx);
    a[idx] = idx;
}
```

The output is as follows:

```
Hello world from work item 0
Hello world from work item 1
Hello world from work item 2
Hello world from work item 3
Hello world from work item 4
Hello world from work item 5
Hello world from work item 6
Hello world from work item 7
```



**IMPORTANT!** `printf()` messages are buffered in the global memory and unloaded when kernel execution is completed. If `printf()` is used in multiple kernels, the order of the messages from each kernel display on the host terminal is not certain. Note, especially when running in hardware emulation and hardware, the hardware buffer size might limit `printf` output capturing.

## Debugging in Hardware Emulation

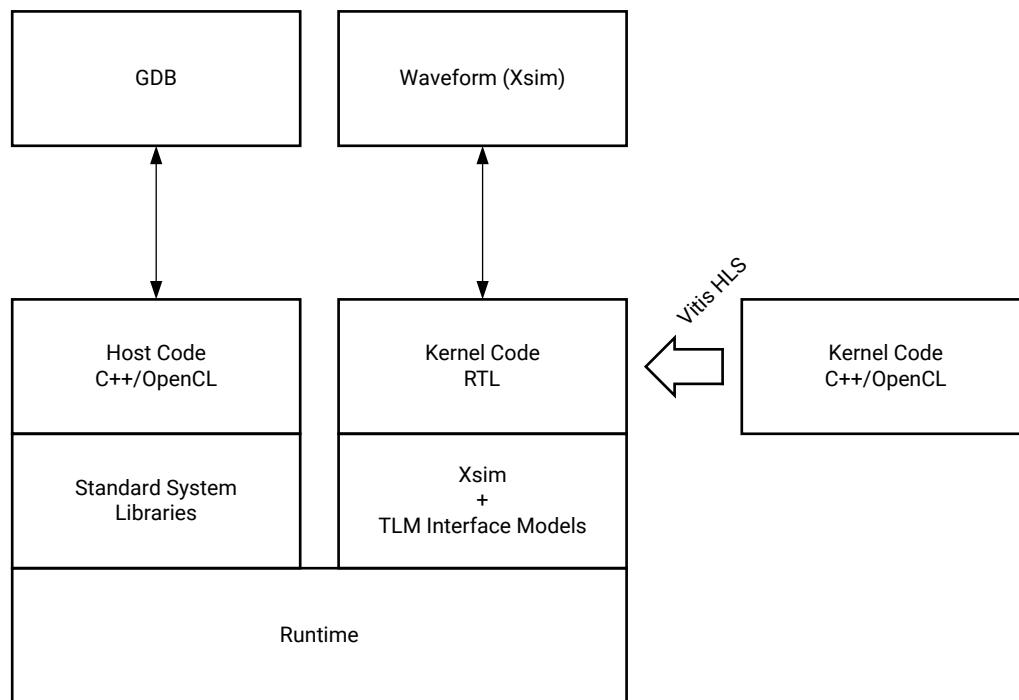


**IMPORTANT!** The following steps describe debugging from the command line. However, the Vitis IDE offers a standalone debug environment for use with the Vitis application acceleration projects created from the command line. Refer to [Using the Standalone Debug Flow](#) for more information.

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 Xilinx 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](#).

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 a Vivado simulator or third-party RTL simulator. GDB is used for more traditional software-style debugging of the host code.

**Figure 45: Hardware Emulation**



X21159-042221

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. The execution time for hardware emulation is longer than software emulation.



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

During 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.

## Waveform-Based Kernel Debugging

Because the C/C++ and OpenCL kernel code is synthesized into RTL code using Vitis HLS in the hardware emulation build configuration, you can also use RTL behavioral simulation to analyze the kernel logic. Hardware designers are likely to be familiar with this approach. This waveform-based HDL debugging is supported by the Vitis core development kit using both the command line flow, or through the IDE flow during hardware emulation.



**TIP:** Waveform-based debugging is considered an advanced feature. In most cases, the RTL Logic does not need to be analyzed.

### Enable Waveform Debugging with the Vitis Compiler Command

The waveform debugging process can be enabled through the `v++` command using the following steps:

1. Enable debug features in the kernel code during compilation and linking, as described in [Building the Device Binary](#).

```
v++ -g ...
```

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
```

3. Run the application, host and kernel, in hardware emulation mode. The waveform database, reflecting the hardware transaction data, is collected in a file named `<hardware_platform>-<device_id>-<xclbin_name>.wdb`. This file can directly be opened in the Vitis analyzer as described in [Section VIII: Using the Vitis Analyzer](#).



**TIP:** If `debug_mode=gui` in the `xrt.ini`, a live waveform viewer is launched when the application is run, as described in [Waveform View and Live Waveform Viewer](#). 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.

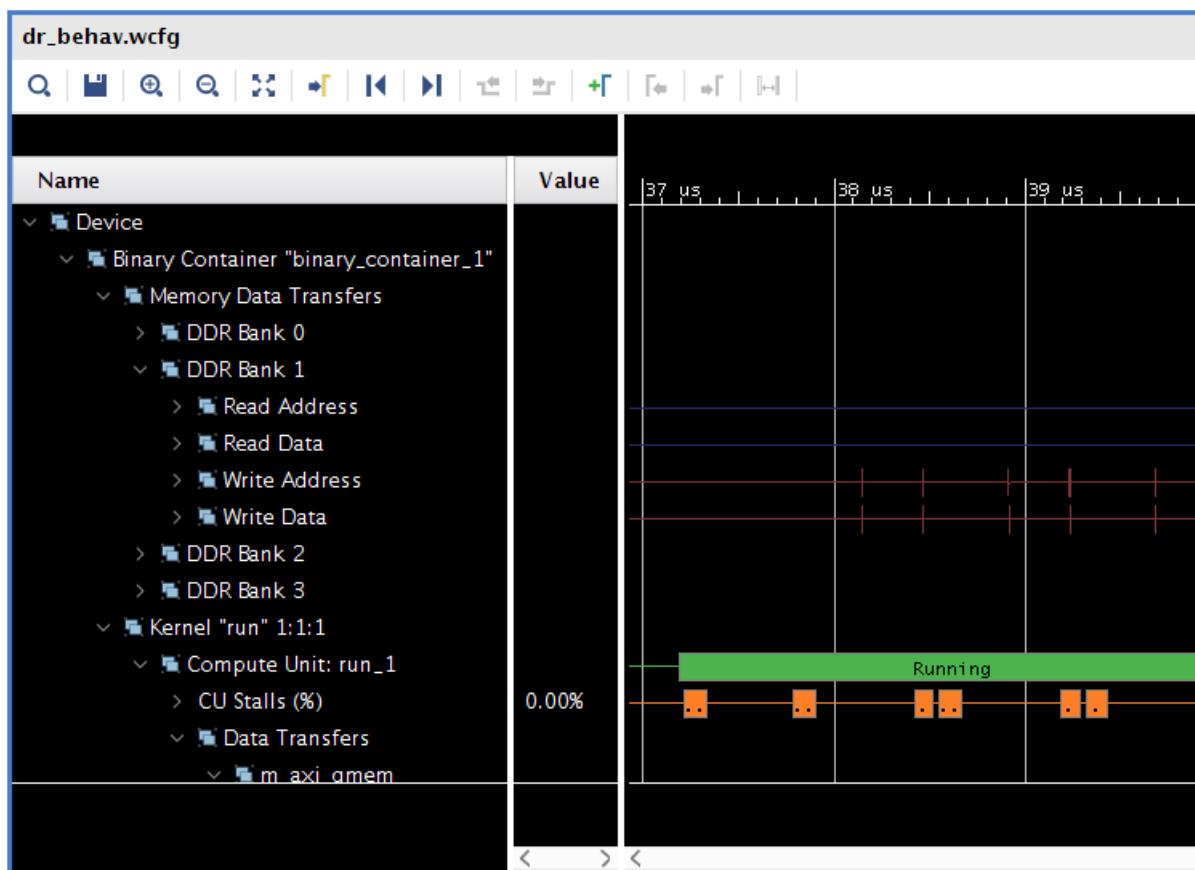
## Run the Waveform-Based Kernel Debugging Flow

The Vitis IDE provides waveform-based HDL debugging in the hardware emulation mode. The waveform is opened in the Vivado waveform viewer which should be familiar to Vivado logic simulation users. The Vitis IDE lets you display kernel interfaces, internal signals, and includes debug controls such as restart, HDL breakpoints, as well as HDL code lookup and waveform markers. In addition, it provides top-level DDR data transfers (per bank) along with kernel-specific details including compute unit stalls, loop pipeline activity, and data transfers.

For details, see [Waveform View and Live Waveform Viewer](#).

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

Figure 46: 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 there 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, the Vitis IDE offers a standalone debug environment for use with the Vitis application acceleration projects created from the command line. Refer to [Using the Standalone Debug Flow](#) for more information.

---

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 just 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.

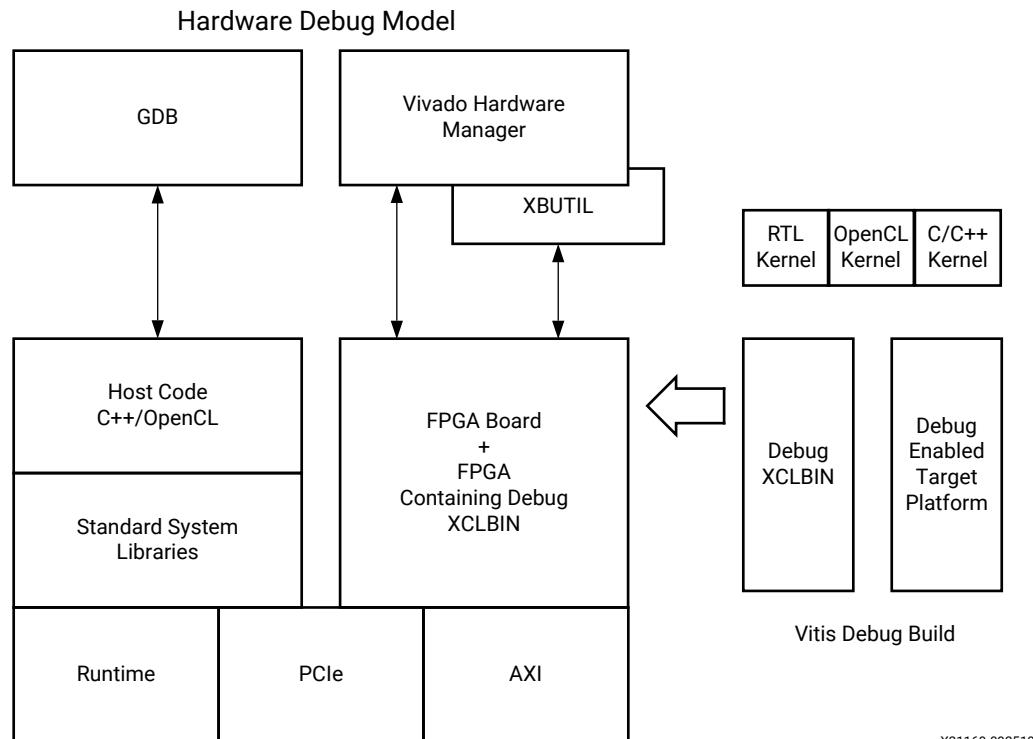


**TIP:** The additional logic required for debugging the hardware should be removed from the final production build.

---

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 47: Hardware Execution



X21160-092519

With the system hardware build configured for debugging, the host program running on the CPU and the Vitis accelerated kernels running on the Xilinx 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, you should 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 Xilinx 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_ilal_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(addr_tvalid),   // input wire [0:0] probe4
    .probe5(addr_tready_n), // input wire [0:0] probe5
    .probe6(addr_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:

```
// 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 must have been 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 a Xilinx platform is used, debug availability can be verified using the `platforminfo` utility to query the platform. Debug capabilities are listed under the `chipscope_debug` objects.

For example, to query the a platform for hardware debug support, the following `platforminfo` command can be used:

```
$ platforminfo --json="hardwarePlatform.extensions.chipscope_debug"
xilinx_u200_xdma_201830_2
{
    "debug_networks": {
        "user": {
            "name": "User Debug Network",
            "pcie_pf": "1",
            "bar_number": "0",
            "axi_baseaddr": "0x000C0000",
            "supports_jtag_fallback": "false",
            "supports_microblaze_debug": "true",
            "is_user_visible": "true"
        },
        "mgmt": {
            "name": "Management Debug Network",
            "pcie_pf": "0",
            "bar_number": "0",
            "axi_baseaddr": "0x001C0000",
            "supports_jtag_fallback": "true",
            "supports_microblaze_debug": "true",
            "is_user_visible": "false"
        }
    }
}
```

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](#).
2. Use the `debug_hw` script to launch the `xvc_PCIE` and `hw_server` apps as follows:

```
debug_hw --xvc_PCIE /dev/xfgpa/xvc_pub.<driver_id> --hw_server
```

The `debug_hw` script returns the following:

```
launching xvc_PCIE...
xvc_PCIE -d /dev/xfgpa/xvc_pub.<driver_id> -s TCP::10200
launching hw_server...
hw_server -sTCP::3121
```



**TIP:** The `/dev/xfgpa/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.

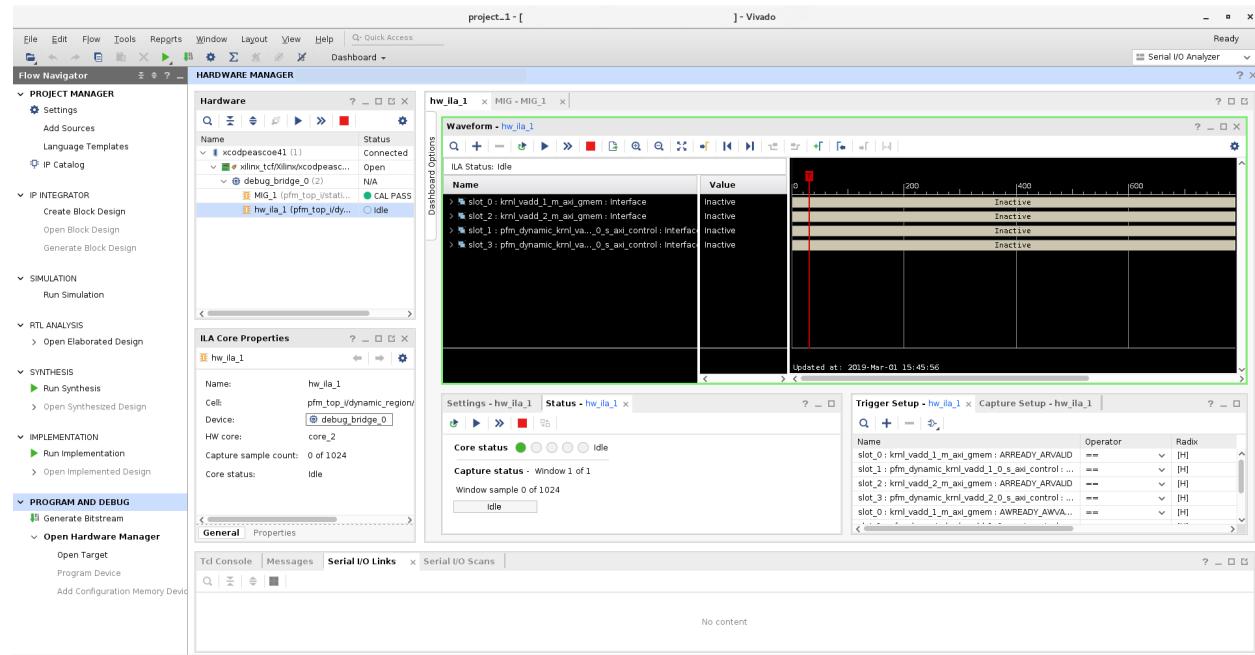
As an example, the command window displays the following results:

```
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.

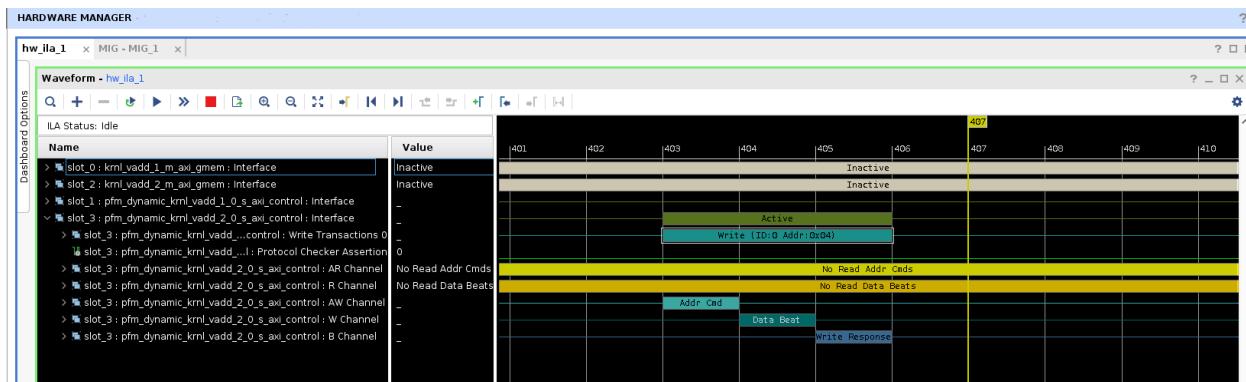
start_gui
```

## 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](#).
2. Launch the `xvc_pcnie` server. The file name passed to `xvc_pcnie` must match the character driver file installed with the kernel device driver, where `<driver_id>` can be found by examining the `/dev` folder.

```
>xvc_pcnie -d /dev/xfpga/xvc_pub.<device_id>
```



**TIP:** The `xvc_pcnie` server has many useful command line options. You can issue `xvc_pcnie -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

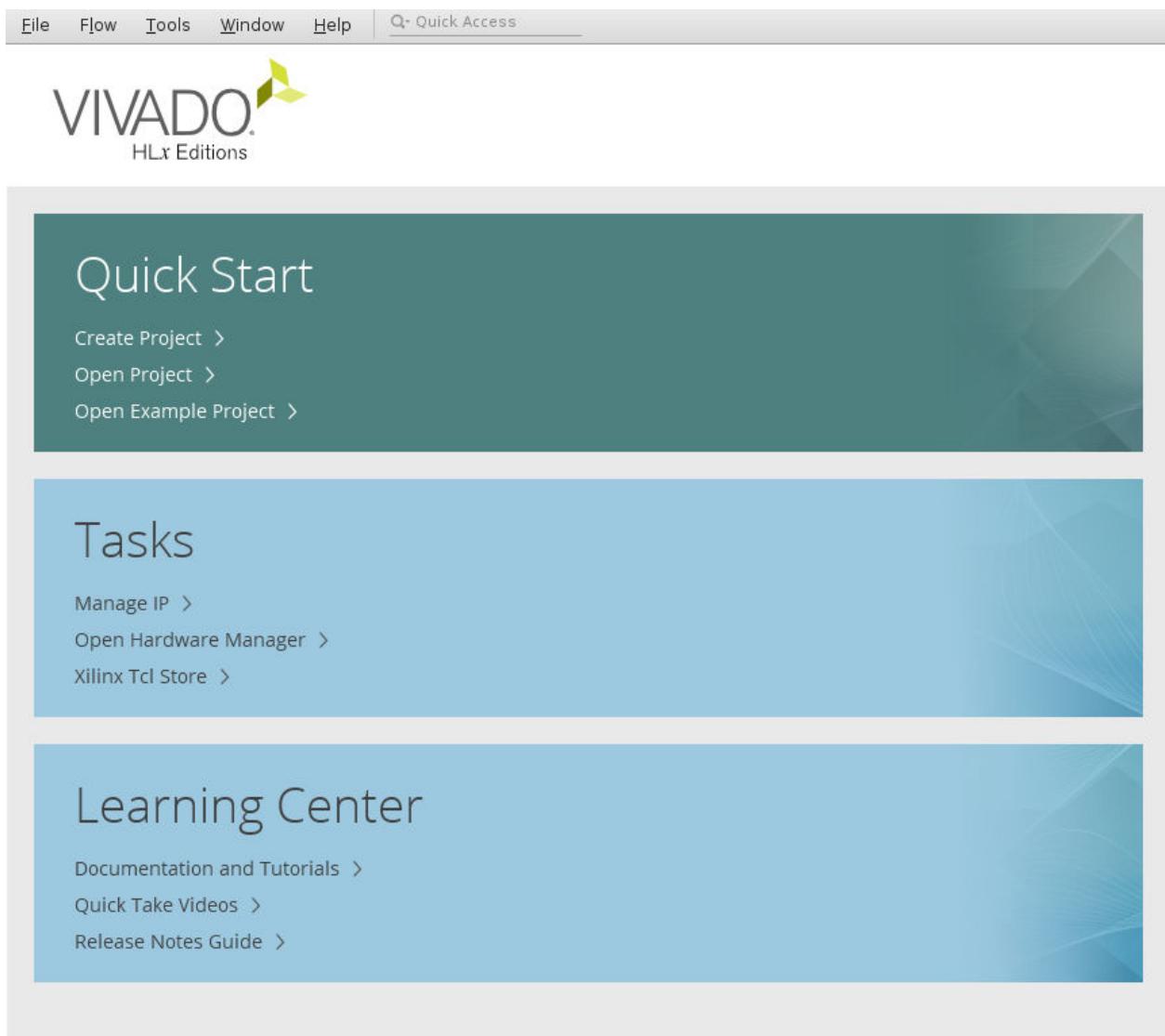
Instructions to start the debug servers on an Amazon F1 instance can be found here: [https://github.com/aws/aws-fpga/blob/master/hdk/docs/Virtual\\_JTAG\\_XVC.md](https://github.com/aws/aws-fpga/blob/master/hdk/docs/Virtual_JTAG_XVC.md).

## **Debugging Designs Using Vivado Hardware Manager**

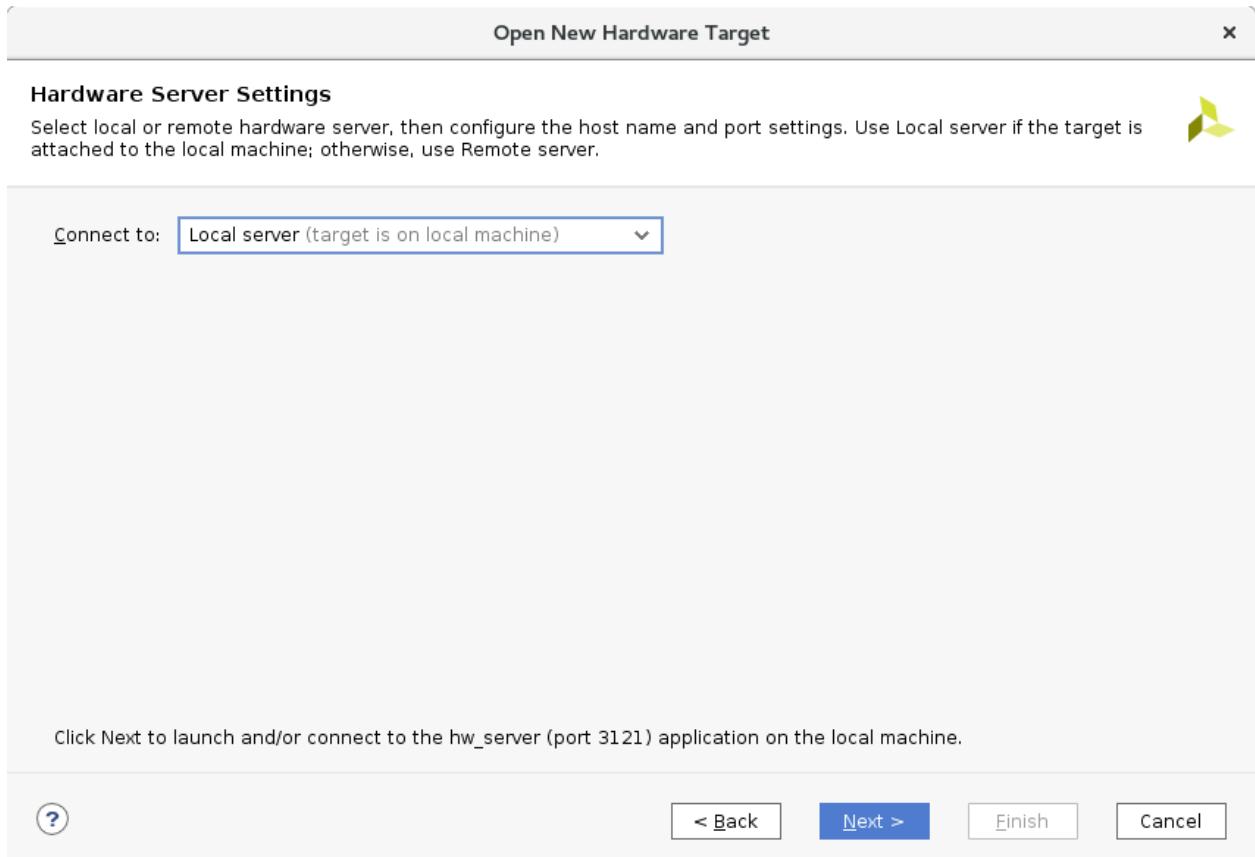
Traditionally, a physical JTAG connection is used to perform hardware debug for Xilinx devices with the Vivado hardware manager. The Vitis unified software platforms also makes use of the Xilinx virtual cable (XVC) for hardware debugging on remote accelerator cards. To take advantage of this capability, 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 Xilinx 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 should be followed on the machine hosting the Vivado tools:

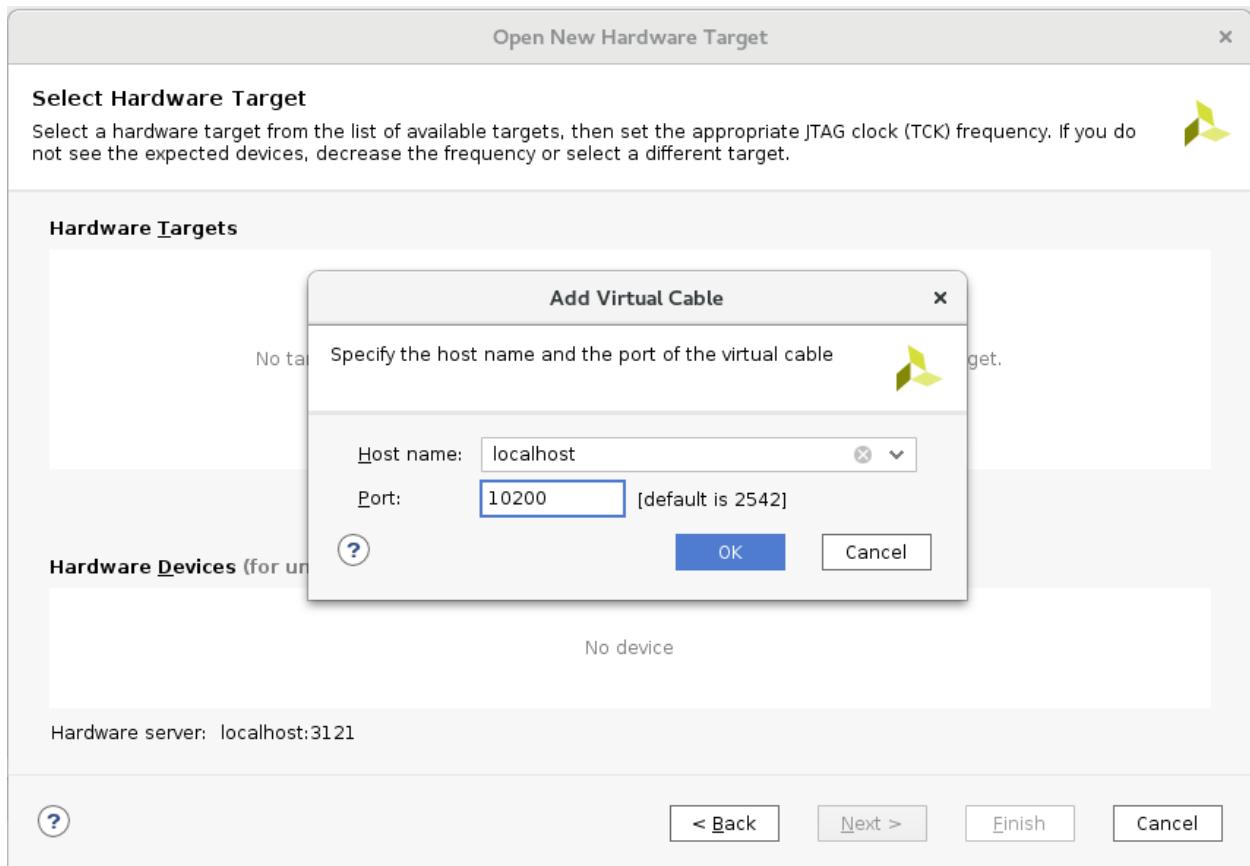
1. Launch the Vivado debug feature, or the full Vivado Design Suite.
2. Select **Open Hardware Manager** from the Tasks menu, as shown in the following figure.



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.



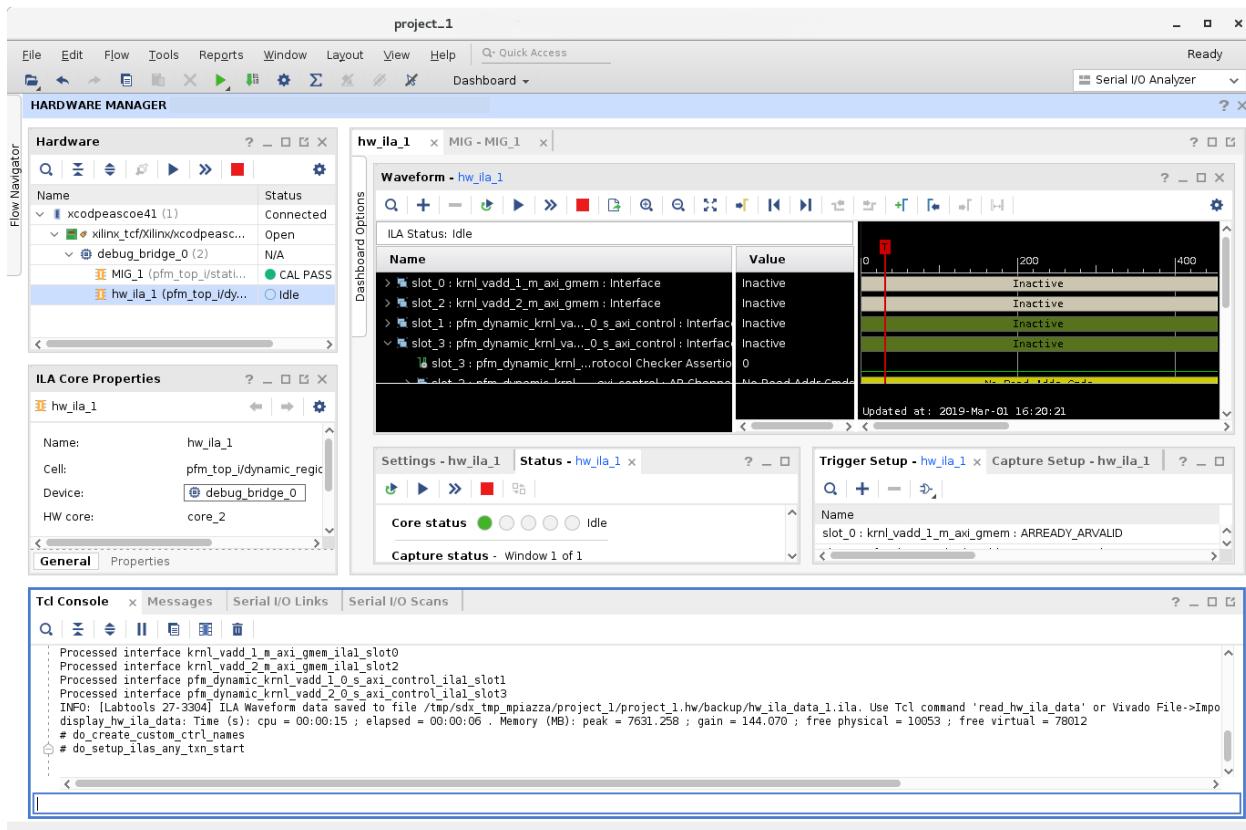
5. Select the `debug_bridge` instance from the Hardware window in the Vivado hardware manager.

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 Hardware window should now show 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.



---

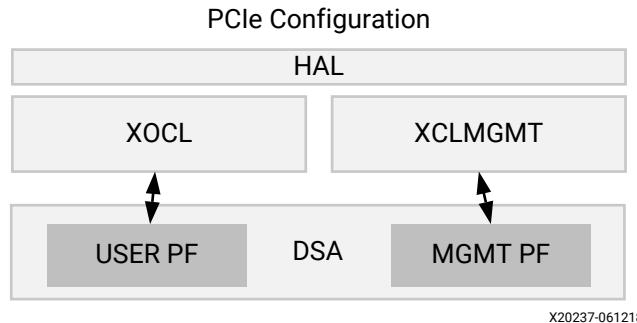
**TIP:** In most cases, it is sufficient to work with the less verbose `xbutil` feature to localize an issue. Refer to [Using the Xilinx xbutil Utility](#) for more information on using this tool for debug.

---

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.), the `dmesg` tool should be used to check the ring buffer.

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

Figure 48: Software Platform Layers



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

```
sudo dmesg -c
```

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

```
sudo dmesg
```

The `dmesg` utility prints a record shown in the following example:

Figure 49: `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(0xfffffc900064e0000) = 0x1fc00006 (id).
[ 9902.318410] xocl:engine_reg_dump: 0-H2C0-MM: ioread32(0xfffffc900064e0040) = 0x00000001 (status).
[ 9902.318895] xocl:engine_reg_dump: 0-H2C0-MM: ioread32(0xfffffc900064e0004) = 0x00f83elf (control)
[ 9902.319370] xocl:engine_reg_dump: 0-H2C0-MM: ioread32(0xfffffc900064e4080) = 0xa7a30000 (first_desc_lo)
[ 9902.319848] xocl:engine_reg_dump: 0-H2C0-MM: ioread32(0xfffffc900064e4084) = 0x00000000 (first_desc_hi)
[ 9902.320336] xocl:engine_reg_dump: 0-H2C0-MM: ioread32(0xfffffc900064e4088) = 0x0000000f (first_desc_adjacent).
[ 9902.320802] xocl:engine_reg_dump: 0-H2C0-MM: ioread32(0xfffffc900064e0048) = 0x00000000 (completed_desc_count).
[ 9902.321279] xocl:engine_reg_dump: 0-H2C0-MM: ioread32(0xfffffc900064e0090) = 0x00f83ele (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 `xbutil` utility.

## Using the Xilinx `xbutil` Utility

The Xilinx board utility (`xbutil`) is a powerful standalone command line utility that can be used to debug lower level hardware/software interaction issues. A full description of this utility can be found in [xbutil Utility](#).

With respect to debugging, the following `xbutil` 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 Xilinx device.
- **xbutil examine -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 `xbutil` commands as shown here.

If the process of hardware debugging does not resolve the problem, it is necessary to perform hardware debugging using the ChipScope feature.

### AXI Firewall Trips

The AXI firewall should prevent host hangs. This is why 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 `xbutil` to program the FPGA:

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



**TIP:** Refer to [xbutil Utility](#) for more information on `xbutil`.

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

```
xbutil 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 should indicate an issue. In this case, it is necessary to check kernel and host code expectations. If the host code is using the Xilinx OpenCL extensions, it is necessary to check which DDR banks should be used by the kernel. These should match the `connectivity.sp` options specified as discussed in [Mapping Kernel Ports to 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 utilization it might be necessary to limit instrumentation.
  - a. Add Lightweight AXI Protocol Checkers (`laptc`). 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 just `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 `xbutil 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.

```
xbutil 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.

```
xbutil 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:

```
xbutil 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 `xbutil` query shows that the CU is stuck in a `BUSY` state and is not making progress.
  2. The AXI Firewall might trip if a PCIe® DMA request is made to the affected MIG, because the DMA engine is unable to complete the request. AXI Firewall trips result in the Linux kernel driver killing all processes which have opened the device node with the `SIGBUS` signal. The `xbutil` query shows if an AXI Firewall has indeed tripped and includes a timestamp.

If the above hang does not occur, the host code might not read back the correct data. This incorrect data is typically 0s and is located in the last part of the data. It is important to review the host code carefully. One common example is compression, where the size of the compressed data is not known up front, and an application might try to migrate more data to the host than was produced by the kernel.

## Defensive Programming

The Vitis 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.

# Debugging on Embedded Processor Platforms

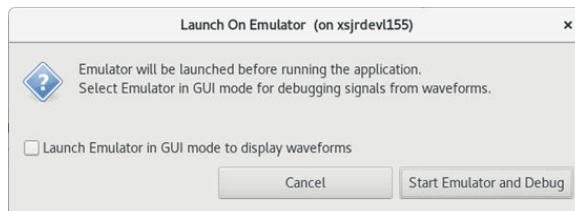
Debugging on embedded processor platforms, such as the `xilinx_zcu104_base_202010_1` platform, requires the use of the QEMU emulation environment to model the Arm processor and operating system for the device. As described in the next sections, running or debugging the application requires the additional step of launching the emulator, or connecting to the hardware platform through a TCF agent.

## Emulation Debug for Embedded Processors

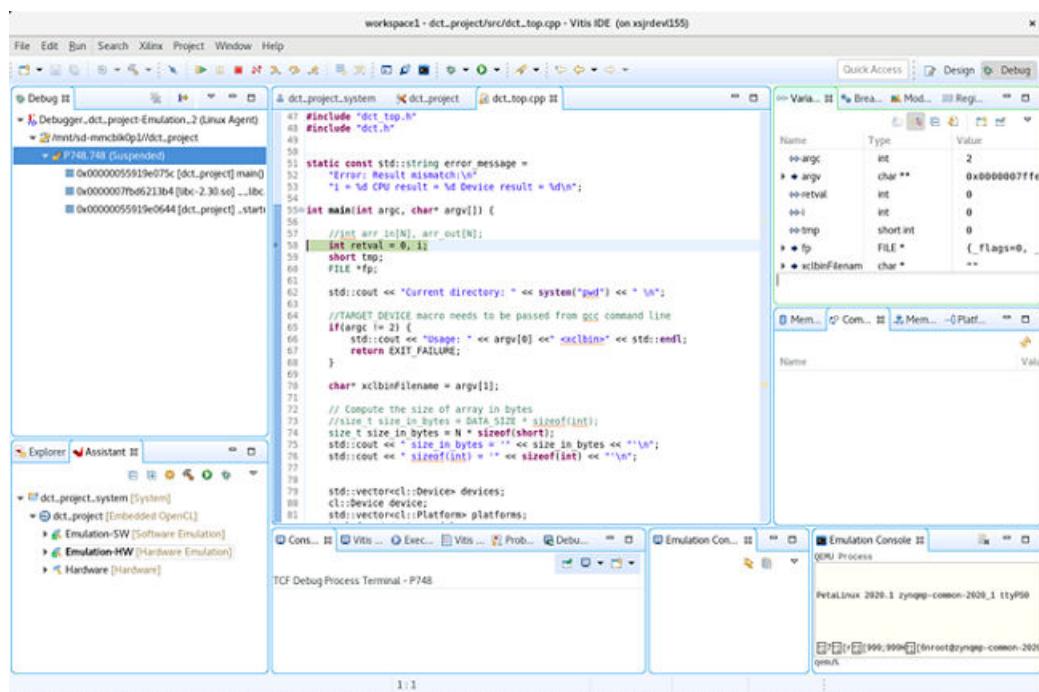
From within the Vitis IDE, launching debug for the software and hardware emulation builds include the following steps:

1. In the Assistant view, right-click the **Emulation-SW** or **Emulation-HW** build and select **Set Active** to make the build active.
2. From the Assistant view menu, select the **Debug** (⚙️) command, and select the **Launch on Emulator** command to launch the debug environment.

This will open the Launch on Emulator dialog box as shown in the following figure. This prompts you to confirm launching the emulation environment and connecting to it using a Linux TCF agent. Select **Start Emulator and Debug** to continue.



This launches the emulation environment (QEMU), and loads the application in preparation for debugging. The application is paused as it enters the `main()` function. The Debug perspective is opened in the Vitis IDE, and you are ready to begin debugging your application.



## Hardware Debug for Embedded Processors

For hardware builds the setup involves the following steps:

1. Copy the contents of the <project>/Hardware/sd\_card/sd\_card folder to a physical SD card. This creates a bootable medium for your target platform.
2. Insert the SD card into the card reader of your embedded processor platform.
3. Change the boot-mode settings of the platform to SD boot mode, and power up the board.
4. After the device is booted, enter the `mount` command at the command prompt to get a list of mount points. As shown in the following figure, the `mount` command displays mounting information for the system.



**TIP:** Be sure to capture the proper path for the `cd` command in the next step, and subsequent commands, based on the results of the `mount` command.

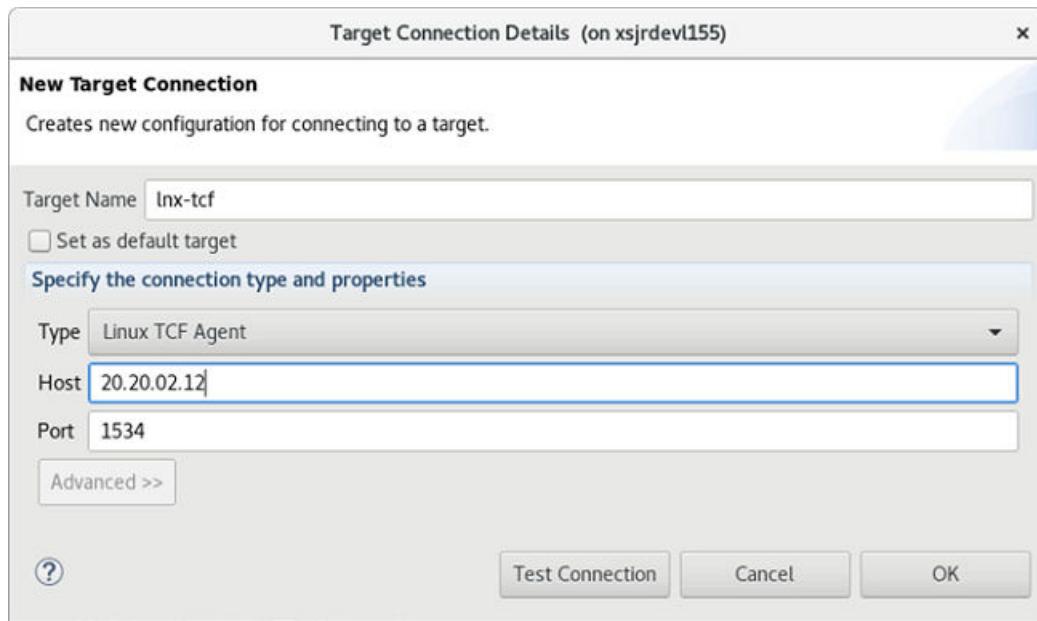
```
root@versal-rootfs-common-2020_1:~# mount
/dev/mmcblk0p2 on / type ext4 (rw,relatime)
devtmpfs on /dev type devtmpfs (rw,relatime,size=893208k,nr_inodes=223302,mode=75
5)
proc on /proc type proc (rw,relatime)
sysfs on /sys type sysfs (rw,relatime)
debugfs on /sys/kernel/debug type debugfs (rw,relatime)
configfs on /sys/kernel/config type configfs (rw,relatime)
tmpfs on /run type tmpfs (rw,nosuid,nodev,mode=755)
tmpfs on /var/volatile type tmpfs (rw,relatime)
/dev/mmcblk0p1 on /run/media/mmcblk0p1 type vfat (rw,relatime,gid=6,fmask=0007,dm
ask=0007,allow_utime=0020,codepage=437,iocharset=iso8859-1,shortname=mixed,errors
=remount-ro)
devpts on /dev/pts type devpts (rw,relatime,gid=5,mode=620,ptmxmode=000)
root@versal-rootfs-common-2020_1:~#
```

5. Execute the following commands, for example:

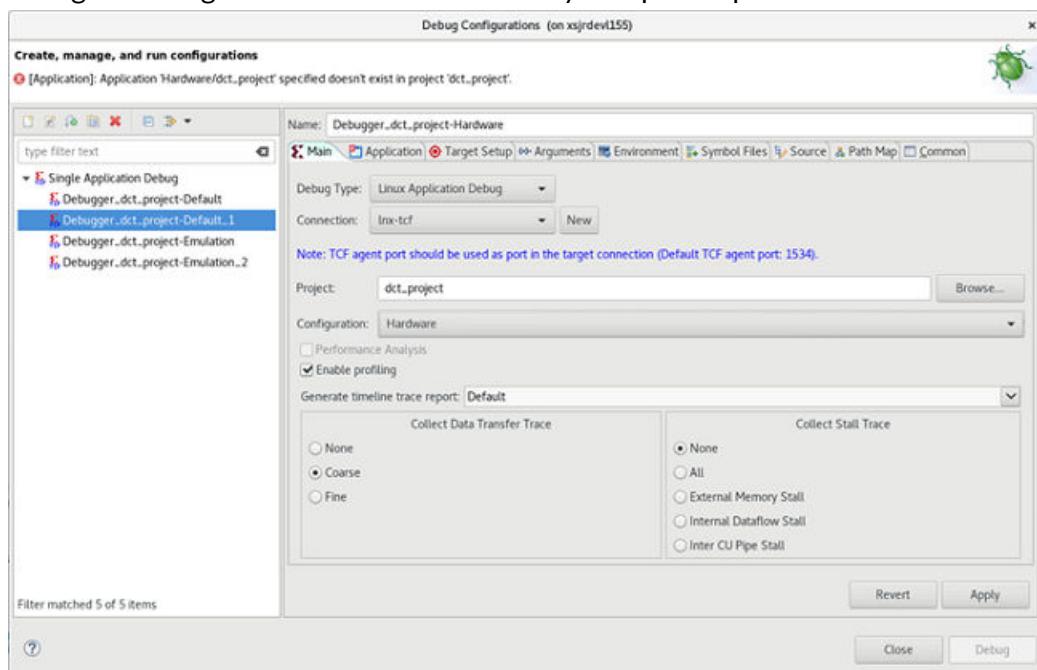
```
cd /run/media/mmcblk0p1
source init.sh
cat /etc/xocl.txt
```

The `cat` command will display the platform name `xilinx_vck190_base_202010_1` to let you confirm it is the same as your specified platform and that your setup is correct.

6. Run `ifconfig` to get the IP address of the target card. You will use the IP address to set up a TCF agent connection in Vitis IDE to connect to the assigned IP address of the embedded processor platform.
7. Create a target connection to the remote accelerator card. Use the **Window**→**Show view**→**Xilinx**→**Target connections** command to open the Target Connections view.
8. In the Target Connections view, right-click the **Linux TCF Agent** and select the **New Target** command to open the New Target Connection dialog box.
9. Specify the **Target Name**, enable the **Set as default target** check box, and specify the **Host IP** address of the accelerator card that you obtained in an earlier step.



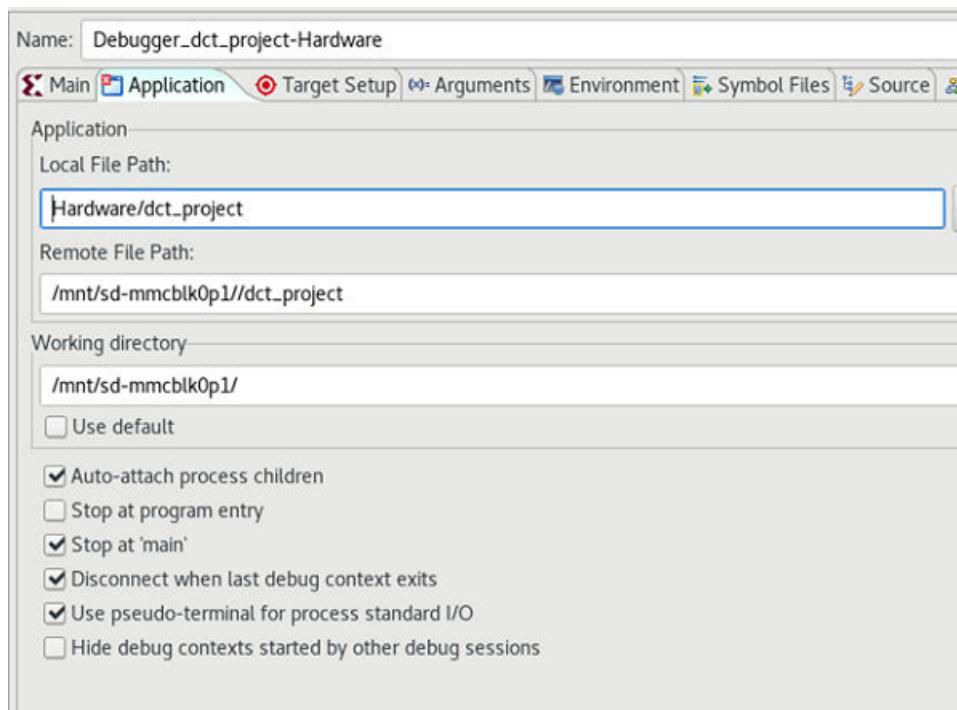
10. Click **OK** to close and continue.
11. In the Assistant view, right-click the Hardware build and select **Set Active** to make it the active build.
12. From the Assistant view menu, select the Debug (bug icon) command, and select the **Debug Configurations** command. This opens the Debug Configurations dialog box to let you configure debug for the Hardware build on your specific platform.



Set the following fields on the **Main** tab of the dialog box:

- **Name:** Specifies a name for your Hardware debug configuration.
- **Linux TCF Agent:** Selects the new agent you built with the specified IP address for the accelerator card.
- **Configuration:** Ensure you have selected the Hardware configuration.
- **Enable Profiling:** If you want to capture trace data from events.

Select the **Application** tab in the Debug Configuration dialog box to see the following fields:



Set the following fields on the Application tab:

- **Local File Path:** Specifies where the files created on the target platform will be written back into your local disk.
- **Remote File Path:** Specifies the remote mount location from the accelerator card as determined in an earlier step.
- **Working directory:** Specifies the location to write files created on the target platform.

13. Select **Apply** to save your changes, and **Debug** to start the process.

This opens the Debug perspective in the Vitis IDE, and connects to the PS application on your hardware platform. The application automatically breaks at the `main()` function to let you set up and configure the debug environment.

# Example of Command Line Debugging

To help you get familiar with debugging using the command line flow, this example walks you through building and debugging the [hello\\_world example](#) available from the Xilinx GitHub.

1. In a terminal, set up your environment as described in [Setting Up the Vitis Environment](#).
2. If you have not already done it, clone the [Vitis Examples](#) GitHub repository to acquire all of the Vitis examples:

```
git clone https://github.com/Xilinx/Vitis_Accel_Examples.git
```

This creates a `Vitis_Examples` directory which includes the IDCT example.

3. CD to the IDCT example directory:

```
cd Vitis_Accel_Examples/hello_world/
```

The host code is fully contained in `src/host.cpp` and the kernel code is part of `src/vadd.cpp`.

4. Build the kernel software for software emulation as discussed in [Building the Device Binary](#).
  - a. Compile the kernel object file for debugging using the `v++` compiler, where `-g` indicates that the code is compiled for debugging:

```
v++ -t sw_emu --platform <DEVICE> -g -c -k vadd \
-o vadd.xo ./src/vadd.cpp
```

- b. Link the kernel object file, also specifying `-g`:

```
v++ -g -l -t sw_emu --platform <DEVICE> -o vadd.xclbin vadd.xo
```

5. Compile and link the host code for debugging using the GNU compiler chain, `g++` as described in [Building the Software Application](#):

**Note:** For embedded processor target platforms, use the GNU Arm cross-compiler as described in [Compiling and Linking for Arm](#).

- a. Compile host code C++ files for debugging using the `-g` option:

```
g++ -c -I${XILINX_XRT}/include -g -o host.o src/host.cpp
```

- b. Link the object files for debugging using `-g`:

```
g++ -g -lOpenCL -lpthread -lrt -lstdc++ -L${XILINX_XRT}/lib/ -o host
host.o
```

6. As described in [emconfigutil Utility](#), prepare the emulation environment using the following command:

```
emconfigutil --platform <device>
```

The actual emulation mode (`sw_emu` or `hw_emu`) then needs to be set through the `XCL_EMULATION_MODE` environment variable. In C-shell this would be as follows:

```
setenv XCL_EMULATION_MODE sw_emu
```

7. As described in [xrt.ini File](#), you must setup the runtime for debug. In the same directory as the compiled host application, create an `xrt.ini` file with the following content:

```
[Debug]
app_debug=true
```

8. Run GDB on the host and kernel code. The following steps guide you through the command line debug process which requires three separate command terminals, setup as described in [Setting Up the Vitis Environment](#).

- a. In the first terminal, start the XRT debug server, which handles the transactions between the host and kernel code:

```
`${XILINX_VITIS}/bin/xrt_server --sdx-url
```

- b. In a second terminal, set the emulation mode:

```
setenv XCL_EMULATION_MODE sw_emu
```

Run GDB by executing the following:

```
xgdb --args host vadd.xclbin
```

Enter the following on the `gdb` prompt:

```
run
```

- c. In the third terminal, attach the software emulation model to GDB to allow stepping through the design. Start up another `xgdb`:

```
xgdb
```

- For debugging in software emulation:

- Type the following on the `gdb` prompt:

```
file <XILINX_VITIS>/data/emulation/unified/cpu_em/generic_pcie/
model/genericpciemodel
```

**Note:** Because GDB does not expand the environment variable, you must specify the path to the Vitis software platform installation as represented by `<XILINX_VITIS>`

- Connect to the kernel process:

```
target remote :NUM
```

Where `NUM` is the number returned by the `xrt_server` as the GDB listener port.

At this point, debugging the host and kernel code can be done as usual with GDB, with the host code and the kernel code running in two different GDB sessions. This is common when dealing with different processes.



---

**IMPORTANT!** Be aware that the application might hit a breakpoint in one process before the next breakpoint in the other process is hit. In these cases, the debugging session in one terminal appears to hang, while the second terminal is waiting for input.

---

# Vitis Commands and Utilities

The reference materials contained here include the following:

- [v++ Command](#): A description of the compiler options (-c), the linking options (-l), options common to both compile and linking, and a discussion of the --config options.
- Various Xilinx® utilities are provided for the Vitis tools and Xilinx Runtime (XRT) to provide detailed information about the platform resources, including SLR and memory resource availability, to help you construct the v++ command line, and manage the build and run process.
  - [emconfigutil Utility](#)
  - [kernelinfo Utility](#)
  - [launch\\_emulator Utility](#)
  - [manage\\_ipcache Utility](#)
  - [package\\_xo Command](#)
  - [platforminfo Utility](#)
  - [xbutil Utility](#)
  - [xbmgmt Utility](#)
  - [xclbinutil Utility](#)



**TIP:** The Xilinx Runtime (XRT) Architecture reference material is available on the [Xilinx Runtime GitHub repository](#).

- The [xrt.ini file](#) is used to initialize XRT to produce reports, debug, and profiling data as it transacts business between the host and kernels. This file is used when the application is run, for emulation or hardware builds, and must be created manually when the build process is run from the command line.
- [HLS Pragmas](#): A description of pragmas used by the Vitis HLS tool in synthesizing C/C++ kernels.

# v++ Command

This section describes the Vitis compiler command, v++, and the various options it supports for building the device binary. v++ is a standalone command line utility with three command modes:

- `--compile (-c)`: For launching the Vitis HLS tool to compile C/C++ code into PL kernel object files (.xo) as described in [Compiling C/C++ PL Kernels](#)
- `--link (-l)`: For linking PL kernels(.xo), AI Engine graph applications (`libadbf.a`), and a target hardware platform (.xpfm) into a Xilinx device binary (.xclbin) or hardware design (.xsaa) as described in [Linking the Kernels](#)
- `--package (-p)`: For packaging an AI Engine `libadbf.a` file into the `xclbin`, and generating an SD card file or QSPI/OSPI file as needed to initialize and boot the accelerated system as described in [Packaging the System](#)

Beyond these three command modes there are many options available to customize the build process as described in the following sections. Some of the options are supported for all three command modes, and some options are specific to compilation, linking, or packaging.

---

## v++ General Options

The v++ command supports many options for the compilation, linking, and packaging processes as described below.



**TIP:** All v++ command-line options can be specified in a configuration file for use with the `--config` option, as discussed in the [Vitis Compiler Configuration File](#). For example, the `--platform` option can be specified in a configuration file using the following syntax:

```
platform=xilinx_u50_gen3x16_xdma_5_202210_1
```

### --advanced

- **Applies to:** Compile, Link, Package

Specify parameters and properties for use by the v++ command. See [--advanced Options](#) for more information.

### --board\_connection

- Applies to:: Link

```
--board_connection
```

Specifies a dual in-line memory module (DIMM) board file for each DIMM connector slot. The board is specified using the Vendor:Board:Name:Version (vbnv) attribute of the DIMM card as it appears in the board repository.

For example:

```
<DIMM_connector>:<vbnv_of_DIMM_board>
```

### -c | --compile

- Applies to:: Compile

```
--compile
```

Required for compilation, but mutually exclusive with `--link` and `--package`. Run `v++ -c` to generate XO files from kernel source files.

### --clock

- Applies to:: Link

Provide a method for assigning clocks to kernels during the linking process. See [--clock Options](#) for more information.

### --config

- Applies to:: Compile, Link, Package

```
--config <config_file> ...
```

Specifies a configuration file containing `v++` command options. The configuration file can be used to capture compilation, linking, or packaging strategies, that can be easily reused by referring to the config file on the `v++` command line. In addition, the config file allows the `v++` command line to be shortened to include only the options that are not specified in the config file. Refer to the [Vitis Compiler Configuration File](#) for more information.



**TIP:** Multiple configuration files can be specified on the `v++` command line. A separate `--config` switch is required for each file used. For example:

```
v++ -l --config system.cfg --config vivado.cfg ...
```

### --connectivity

- Applies to:: Link

Used to specify important architectural details of the device binary during the linking process. See [--connectivity Options](#) for more information.

### --custom\_script

- Applies to:: Compile, Link

```
--custom_script <kernel_name>:<file_name>
```

This option lets you specify custom Tcl scripts to be used in the build process during compilation or linking. Use with the `--export_script` option to create, edit, and run the scripts to customize the build process.

When used with the `v++ --compile` command, this option lets you specify a custom HLS script to be used when compiling the specified kernel. The script lets you modify or customize the Vitis HLS tool. Use the `--export_script` option to extract a Tcl script Vitis HLS uses to compile the kernel, modify the script as needed, and resubmit using the `--custom_script` option to better manage the kernel build process.

The argument lets you specify the kernel name, and path to the Tcl script to apply to that kernel. For example:

```
v++ -c -k kernel1 -export_script ...
*** Modify the exported script to customize in some way, then resubmit. ****
v++ -c --custom_script kernel1:./kernel1.tcl ...
```

When used with the `v++ --link` command for the hardware build target (`-t hw`), this option lets you specify the absolute path to an edited `run_script_map.dat` file. This file contains a list of steps in the build process, and Tcl scripts that are run by the Vitis and Vivado tools during those steps. You can edit `run_script_map.dat` to specify custom Tcl scripts to run at those steps in the build process. You must use the following steps to customize the Tcl scripts:

1. Run the build process specifying the `--export_script` option as follows:

```
v++ -t hw -l -k kernel1 -export_script ...
```

2. Copy the Tcl scripts referenced in the `run_script_map.dat` file for any of the steps you want to customize. For example, copy the Tcl file specified for the synthesis run, or the implementation run. You must copy the file to a separate location, outside of the project build structure.
3. Edit the Tcl script to add or modify any of the existing commands to create a new custom Tcl script.
4. Edit the `run_script_map.dat` file to point a specific implementation step to the new custom script.

5. Relaunch the build process using the `--custom_script` option, specifying the absolute path to the `run_script_map.dat` file as shown below:

```
v++ -t hw -l -k kernel1 -custom_script /path/to/run_script_map.dat
```

 **IMPORTANT!** When editing a custom synthesis run script, you must either comment out the lines related to the `dont_touch.xdc` file, or edit the lines to point to a new user-specified `dont_touch.xdc` file. The specific lines to comment or edit are shown below:

```
read_xdc dont_touch.xdc
set_property used_in_implementation false [get_files dont_touch.xdc]
```

The synthesis run returns an error related to a missing `dont_touch.xdc` file if this is not done.

### --debug

- **Applies to::** Link

Specify the addition of debug IP core insertion into the hardware design. See [--debug Options](#) for more information.

### -D | --define

- **Applies to::** Compile, Link

```
--define <arg>
```

Valid macro name and definition pair: `<name>=<definition>`.

Predefine name as a macro with definition. This option is passed to the `v++` preprocessor.

### --export\_script

- **Applies to::** Compile, Link, Package

```
--export_script
```

This option runs the build process up to the point of exporting a script file, or list of script files, and then stops execution. The build process must be completed using the `--custom_script` option. This lets you edit the exported script, or list of scripts, and then rerun the build using your custom scripts.

When used with the `v++ --compile` command, this option exports a Tcl script for the specified kernel, `<kernel_name>.tcl`, that can be used to execute Vitis HLS, but stops the build process before actually launching the HLS tool. This lets you interrupt the build process to edit the generated Tcl script, and then restart the build process using the `--custom_script` option, as shown in the following example:

```
v++ -c -k kernel1 -export_script ...
```



**TIP:** This option is not supported for software emulation (-t sw\_emu) of OpenCL kernels.

When used with the `v++ --link` command for the hardware build target (-t hw), this option exports a `run_script_map.dat` file in the current directory. This file contains a list of steps in the build process, and Tcl scripts that are run by the Vitis and Vivado tools during those steps. You can edit the specified Tcl scripts, customizing the build process in those scripts, and relaunch the build using the `--custom_script` option. Export the `run_script_map.dat` file using the following command:

```
v++ -l -t hw -export_script ...
```

### **--from\_step**

- **Applies to:** Compile, Link, Package

```
--from_step <arg>
```

Specifies a step name for the Vitis compiler build process, to start the build process from that step. If intermediate results are available, the link process fast forwards and begins execution at the named step if possible. This allows you to run the build through a `--to_step`, and then resume the build process at the `--from_step`, after interacting with your project in some method. You can use the `--list_step` option to determine the list of valid steps.



**IMPORTANT!** `--to_step`/`--from_step` are sequential build options that require you to use `--from_step` to resume the build in the same project directory that you used when starting the build with `--to_step`.

For example:

```
v++ --link --from_step vpl.update_bd
```

### **-g**

- **Applies to:** Compile, Link

```
-g
```

Generates code for debugging the kernel during software emulation. Using this option adds features to facilitate debugging the kernel as it is compiled.

For example:

```
v++ -g ...
```

**-h | --help**

- **Applies to:** Compile, Link, Package

```
-h
```

Prints the help contents for the v++ command. For example:

```
v++ -h
```

**--hls**

- **Applies to:** Compile

Specify options for the Vitis HLS synthesis process during kernel compilation. See [--hls Options](#) for more information.

**-I | --include**

- **Applies to:** Compile, Link

```
--include <arg>
```

Add the specified directory to the list of directories to be searched for header files. This option is passed to the Vitis compiler pre-processor.

**--input\_files <input\_file>**

- **Applies to:** Compile, Link, Package

```
--input_files <input_file1> <input_file2> ...
```

Specifies an OpenCL or C/C++ kernel source file for v++ compilation, or Xilinx object (XO) files for v++ linking.

For example:

```
v++ -l --input_files kernel1.xo kernelRTL.xo ...
```



**TIP:** Input files can also be specified positionally without the --input\_files option. For example:

```
v++ -l kernel1.xo kernelRTL.xo ...
```

**--interactive**

- **Applies to:** Link

```
--interactive [ impl ]
```

v++ configures the needed environment and launches the Vivado tool with the implementation project.

Because you are interactively launching the Vivado tool, the linking process is stopped after the vpl step, which is the equivalent of using the --to\_step vpl option in your v++ command.

When you are done interactively working with the Vivado tool, and you save the design checkpoint (DCP), you can resume the Vitis compiler linking process using the v++ --from\_step rtsgen, or use the --reuse\_impl or --reuse\_bit options to read in the implemented DCP file or bitstream.

For example:

```
v++ --interactive impl
## Interactively use the Vivado tool
v++ --from_step rtsgen
```

### -k | --kernel

- Applies to:: Compile

```
--kernel <arg>
```

Compile only the specified kernel from the input file. Only one -k option is allowed per v++ command. Valid values include the name of the kernel to be compiled from the input .cl or .c/.cpp kernel source code.

This is required for C/C++ kernels, but is optional for OpenCL kernels. OpenCL uses the kernel keyword to identify a kernel. For C/C++ kernels, you must identify the kernel by -k or --kernel.

When an OpenCL source file is compiled without the -k option, all the kernels in the file are compiled. Use -k to target a specific kernel.

For example:

```
v++ -c --kernel vadd
```

### --kernel\_frequency

---

 **IMPORTANT!** This command is used to specify kernel frequencies for Alveo platforms with scalable clocks. Platforms using fixed clocks, including both Alveo and embedded platforms, use the [--clock Options](#) for clock management. Refer to [Managing Clock Frequencies](#) for more information.

---

- Applies to:: Link

```
--kernel_frequency <freq> | <clockID>:<freq>[<clockID>:<freq>]
```

Specifies a user-defined clock frequency (in MHz) for the kernel, overriding the default clock frequency defined on the hardware platform. The <freq> specifies a single frequency for kernels with only a single clock, or can be used to specify the <clockID> and the <freq> for kernels that support two clocks.

The syntax for overriding the clock on a platform with only one kernel clock, is to simply specify the frequency in MHz:

```
v++ --kernel_frequency 300
```

To override a specific clock on a platform with two clocks, specify the clock ID and frequency:

```
v++ --kernel_frequency 0:300
```

To override both clocks on a multi-clock platform, specify each clock ID and the corresponding frequency. For example:

```
v++ --kernel_frequency 0:300|1:500
```



---

**TIP:** During implementation of the design, if Vivado place and route tools are unable to meet the frequency specification, the tools can scale the clock frequency to an achievable frequency.

---

## -l | --link

- Applies to:: Link

```
--link
```

This is a required option for the linking process, which follows compilation, but is mutually exclusive with --compile or --package. Run v++ in link mode to link XO input files and generate an `xclbin` or and `.xsa` output file.



---

**IMPORTANT!** As discussed in [Linking the Kernels](#), the `--link` option generates an `.xclbin` file for most platforms, but generates an `.xsa` file for Versal platforms.

---

## --linkhook

- Applies to:: Link

Lets you customize the build process for the device binary by specifying Tcl scripts to be run at specific steps in the implementation flow. See [--linkhook Options](#) for more information.

## --list\_steps

- Applies to:: Compile, Link, Package

```
--list_steps
```

List valid run steps for a given target. This option returns a list of steps that can be used in the `--from_step` or `--to_step` options. The command must be specified with the following options:

- `-t` | `--target [sw_emu | hw_emu | hw ]:`
- `[ --compile | --link ]:` Specifies the list of steps from either the compile or link process for the specified build target.

For example:

```
v++ -t hw_emu --link --list_steps
```

#### **--log\_dir**

- **Applies to:** Compile, Link, Package

```
--log_dir <dir_name>
```

Specifies a directory to store log files into. If `--log_dir` is not specified, the tool saves the log files to `./_x/logs`. Refer to [Output Directories of the v++ Command](#) for more information.

For example:

```
v++ --log_dir /tmp/myProj_logs ...
```

#### **--message\_rules**

- **Applies to:** Compile, Link, Package

```
--message_rules <file_name>
```

Specifies a message rule file with rules for controlling messages. Refer to [Using the Message Rule File](#) for more information.

For example:

```
v++ --message_rules ./minimum_out.mrf ...
```

#### **--no\_ip\_cache**

- **Applies to:** Link

```
--no_ip_cache
```

Disables the IP cache for out-of-context (OOC) synthesis for Vivado Synthesis. Disabling the IP cache repository requires the tool to regenerate the IP synthesis results for every build, and can increase the build time. However, it also results in a clean build, eliminating earlier results for IP in the design.

For example:

```
v++ --no_ip_cache ...
```

### **-O | --optimize**

- **Applies to::** Link

```
--optimize <arg>
```

This option specifies the optimization level of the Vivado implementation results. Valid optimization values include the following:

- 0: Default optimization. Reduces compilation time.
- 1: Optimizes to reduce power consumption by running Vivado implementation strategy Power\_DefaultOpt. This takes more time to build the design.
- 2: Optimizes to increase kernel speed. This option increases build time, but also improves the performance of the generated kernel by adding the PHYS\_OPT\_DESIGN step to implementation.
- 3: This optimization provides the highest level performance in the generated code, but compilation time can increase considerably. This option specifies retiming during synthesis, and enables both PHYS\_OPT\_DESIGN and POST\_ROUTE\_PHYS\_OPT\_DESIGN during implementation.
- s: Optimizes the design for size. This reduces the logic resources of the device used by the kernel by running the Area\_Explore implementation strategy.
- quick: Reduces Vivado implementation time, but can reduce kernel performance, and increases the resources used by the kernel. This enables the Flow\_RuntimeOptimized strategy for both synthesis and implementation.

For example:

```
v++ --link --optimize 2
```

### **-o | --output**

- **Applies to::** Compile, Link, Package

```
-o <output_name>
```

Specifies the name of the output file generated by the v++ command. The compilation (-c) process output name must end with the XO file suffix, for Xilinx object file. The linking (-l) process output file must end with the xclbin file suffix, for Xilinx executable binary.

For example:

```
v++ -o krnl_vadd.xo
```

If `--o` or `--output` are not specified, the output file names default to the following:

- **Compilation:** `a.o`
- **Linking:** `a.xclbin` (`a.xsa` for Versal platforms)
- **Packaging:** `a.xclbin`

### **-p | --package**

- **Applies to::** Package

Specify options for the Vitis compiler to package your design for either running emulation or running on hardware. See [--package Options](#) for more information.

### **-f | --platform**

- **Applies to::** Compile, Link, Package

```
--platform <platform_name>
```

Specifies the name of a supported acceleration platform as specified by the `$PLATFORM_REPO_PATHS` environment variable, or the full path to the platform `.xpfm` file. For a list of supported platforms for the release, see the [Vitis Software Platform Release Notes](#).

This is a required option for both compilation and linking, to define the target Xilinx platform of the build process. The `--platform` option accepts either a platform name, or the path to a platform file `xpfm`, using the full or relative path.



**IMPORTANT!** The specified platform and build targets for compiling, linking, and packaging must match.

The `--platform` and `-t` options specified when the `XO` file is generated by compilation, must be the `--platform` and `-t` used during linking, and packaging. For more information, see [platforminfo Utility](#).

For example:

```
v++ --platform xilinx_u50_gen3x16_xdma_5_202210_1 ...
```



**TIP:** All Vitis compiler options can be specified in a configuration file for use with the `--config` option.

For example, the `platform` option can be specified in a configuration file without a section head using the following syntax:

```
platform=xilinx_u50_gen3x16_xdma_5_202210_1
```

### **--profile**

- **Applies to::** Compile, Link

Specify options to configure the Xilinx runtime environment to capture application performance information. See [--profile Options](#) for more information.

**--remote\_ip\_cache**

- Applies to:: Link

```
--remote_ip_cache <dir_name>
```

Specifies the location of the remote IP cache directory for Vivado Synthesis to use during out-of-context (OOC) synthesis of IP. OOC synthesis lets the Vivado synthesis tool reuse synthesis results for IP that have not been changed in iterations of a design. This can reduce the time required to build your .xclbin files, due to reusing synthesis results.

When the --remote\_ip\_cache option is not specified the IP cache is written to the current working directory from which v++ was launched. You can use this option to provide a different cache location, used across multiple projects for instance.

For example:

```
v++ --remote_ip_cache /tmp/IP_cache_dir ...
```

**--report\_dir**

- Applies to:: Compile, Link, Package

```
--report_dir <dir_name>
```

Specifies a directory to store report files into. If --report\_dir is not specified, the tool saves the report files to ./\_x/reports. Refer to [Output Directories of the v++ Command](#) for more information.

For example:

```
v++ --report_dir /tmp/myProj-reports ...
```

**-R | --report\_level**

- Applies to:: Compile, Link, Package

```
--report_level <arg>
```

Valid report levels: 0, 1, 2, estimate.

These report levels have mappings kept in the optMap.xml file. You can override the installed optMap.xml to define custom report levels.

- -R0 specification turns off all intermediate design checkpoint (DCP) generation during Vivado implementation. Turns on post-route timing report generation.

- The `-R1` specification includes everything from `-R0`, plus `report_failfast pre-opt_design`, `report_failfast post-opt_design`, and enables all intermediate DCP generation.
- The `-R2` specification includes everything from `-R1`, plus `report_failfast post-route_design`.
- The `-Restimate` specification forces Vitis HLS to generate a `design.xml` file if it does not exist and then generates a System Estimate report, as described in [System Estimate Report](#).



**TIP:** This option is useful for the software emulation build (`-t sw_emu`), when `design.xml` is not generated by default.

For example:

```
v++ -R2 ...
```

#### --reuse\_bit

```
--reuse_bit <arg>
```

- **Applies to::** Link

Specifies the path and file name of generated bitstream file (`.bit`) to use when generating the device binary (`xclbin`) file. As described in [Using -to\\_step and Launching Vivado Interactively](#), you can specify the `--to_step` option to interrupt the Vitis build process and manually place and route a synthesized design to generate the bitstream.



**IMPORTANT!** The `--reuse_bit` option is a sequential build option that requires you to use the same project directory when resuming the Vitis compiler with `--reuse_bit` that you specified when using `--to_step` to start the build.

For example:

```
v++ --link --reuse_bit ./project.bit
```

#### --reuse\_impl

```
--reuse_impl <arg>
```

- **Applies to::** Link

Specifies the path and file name of an implemented design checkpoint (DCP) file to use when generating the device binary (`xclbin`) file. The link process uses the specified implemented DCP to extract the FPGA bitstream and generates the `xclbin`. You can manually edit the Vivado project created by a previously completed Vitis build, or specify the `--to_step` option to interrupt the Vitis build process and manually place and route a synthesized design, for instance. This allows you to work interactively with Vivado Design Suite to change the design and use DCP in the build process.



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

For example:

```
v++ --link --reuse_impl ./manual_design.dcp
```

### -s | --save-temp

- **Applies to::** Compile, Link, Package

```
--save-temp
```

Directs the `v++` command to save intermediate files/directories created during the compilation and link process. Use the `--temp_dir` option to specify a location to write the intermediate files to.



**TIP:** This option is useful for debugging when you encounter issues in the build process.

For example:

```
v++ --save-temp ...
```

### -t | --target

- **Applies to::** Compile, Link, Package

```
-t [ sw_emu | hw_emu | hw ]
```

Specifies the build target, as described in [Build Targets](#). The build target determines the results of the compilation and linking processes. You can choose to build an emulation model for debug and test, or build the actual system to run in hardware. The build target defaults to `hw` if `-t` is not specified.



**IMPORTANT!** The specified platform and build targets for compiling and linking must match. The `--platform` and `-t` options specified when the XO file is generated by compilation must be the `--platform` and `-t` used during linking.

The valid values are:

- `sw_emu`: Software emulation
- `hw_emu`: Hardware emulation
- `hw`: Hardware

For example:

```
v++ --link -t hw_emu
```

### --temp\_dir

- **Applies to:** Compile, Link, Package

```
--temp_dir <dir_name>
```

This allows you to manage the location where the tool writes temporary files created during the build process. The temporary results are written by the v++ compiler, and then removed, unless the `--save-temp`s option is also specified.

If `--temp_dir` is not specified, the tool saves the temporary files to `./_x/temp`. Refer to [Output Directories of the v++ Command](#) for more information.

For example:

```
v++ --temp_dir /tmp/myProj_temp ...
```

### --to\_step

- **Applies to:** Compile, Link, Package

```
--to_step <arg>
```

Specifies a step name, for either the compile or link process, to run the build process through that step. You can use the `--list_step` option to determine the list of valid compile or link steps.

The build process terminates after completing the named step. At this time, you can interact with the build results. For example, manually accessing the HLS project or the Vivado Design Suite project to perform specific tasks before returning to the build flow, launch the v++ command with the `--from_step` option.




---

**IMPORTANT!** `--to_step`/`--from_step` are sequential build options that require you to use `--from_step` to resume the build in the same project directory that you used when starting the build with `--to_step`.

---

You must also specify `--save-temp`s when using `--to_step` to preserve the temporary files required by the Vivado tools. For example:

```
v++ --link --save-temp --to_step vpl.update_bd
```

### --user\_board\_repo\_paths

- **Applies to:** Link

```
--user_board_repo_paths
```

Specifies an existing user board repository for DIMM board files. This value is pre-pended to the `board_part_repo_paths` property of the Vivado project.

### --user\_ip\_repo\_paths

- Applies to:: Link

```
--user_ip_repo_paths <repo_dir>
```

Specifies the directory location of one or more user IP repository paths to be searched first for IP used in the kernel design. This value is appended to the start of the `ip_repo_paths` used by the Vivado tool to locate IP cores. IP definitions from these specified paths are used ahead of IP repositories from the hardware platform (`.xsa`) or from the Xilinx IP catalog.



**TIP:** Multiple `--user_ip_repo_paths` can be specified on the `v++` command line.

The following lists show the priority order in which IP definitions are found during the build process, from high to low.

**Note:** All of following entries can possibly include multiple directories in them.

- For the system hardware build (-t hw):
  1. IP definitions from `--user_ip_repo_paths`.
  2. Kernel IP definitions (`vpl --iprepo` switch value).
  3. IP definitions from the IP repository associated with the platform.
  4. IP cache from the installation area (for example, `<Install_Dir>/Vitis/2019.2/data/cache/`).
  5. Xilinx IP catalog from the installation area (for example, `<Install_Dir>/Vitis/2019.2/data/ip/`)
- For the hardware emulation build (-t hw\_emu):
  1. IP definitions and User emulation IP repository from `--user_ip_repo_paths`.
  2. Kernel IP definitions (`vpl --iprepo` switch value).
  3. IP definitions from the IP repository associated with the platform.
  4. IP cache from the installation area (for example, `<Install_Dir>/Vitis/2019.2/data/cache/`).
  5. `$::env(XILINX_VITIS)/data/emulation/hw_em/ip_repo`
  6. `$::env(XILINX_VIVADO)/data/emulation/hw_em/ip_repo`
  7. Xilinx IP catalog from the installation area (for example, `<Install_Dir>/Vitis/2019.2/data/ip/`)

For example:

```
v++ --user_ip_repo_paths ./myIP_repo ...
```

### -v | --version

```
-v
```

Prints the version and build information for the v++ command. For example:

```
v++ -v
```

### --vivado

- **Applies to:** Link

Specify properties and parameters to configure the Vivado synthesis and implementation environment prior to building the device binary. See [--vivado Options](#) for more information.

---

## --advanced Options

The --advanced.param and --advanced.prop options specify parameters and properties for use by the v++ command. When compiling or linking, these options offer fine-grain control over the hardware generated by the Vitis core development kit, and the hardware emulation process.

The arguments for the --advanced.xxx options are specified as

<param\_name>=<param\_value>. For example:

```
v++ --link --advanced.param compiler.enableXSACheck=true  
--advanced.prop kernel.foo.kernel_flags="-std=c++0x"
```



**TIP:** All Vitis compiler options can be specified in a configuration file for use with the --config option, as discussed in [Vitis Compiler Configuration File](#). For example, the --platform option can be specified in a configuration file without a section head using the following syntax:

```
platform=xilinx_u200_gen3x16_xdma_2_202110_1
```

### --advanced.param

```
--advanced.param <param_name>=<param_value>
```

Specifies advanced parameters as described in the table below.

## Param Options

Parameter Name	Valid Values	Description
compiler.acceleratorBinaryContent	Type: String Default Value: <empty>	<p>Design content to insert in the generated <code>xclbin</code> file. Valid options include <code>bitstream</code>, <code>pdi</code>, or <code>dcp</code>. <code>bitstream</code> and <code>pdi</code> are mutually exclusive. <code>pdi</code> applies to Versal platforms, <code>bitstream</code> applies to non-Versal platforms.</p> <p><b>TIP:</b></p> <p>You can specify two values to have <code>v++</code> generate two <code>xclbin</code> files: one containing a DCP file, and the other containing either a bitstream or PDI file. For example:</p> <pre>--advanced.param compiler.acceleratorBinaryContent=dcp,bitstream --advanced.param compiler.acceleratorBinaryContent=dcp,pdi</pre> <p>This parameter is used while building the hardware target, this option applies to:</p> <ul style="list-style-type: none"> <li>• <code>v++ --link</code></li> <li>• <code>vpl.impl</code></li> <li>• <code>xclbinutil</code></li> </ul>
compiler.addOutputTypes	Type: String Default Value: <empty>	<p>Additional output types produced by the Vitis compiler. Valid values include: <code>xclbin</code> and <code>hw_export</code>. Use <code>hw_export</code> to create a fixed XSA from dynamic hardware platforms for use in the Embedded Software Development Flow.</p> <p>Applies to:</p> <ul style="list-style-type: none"> <li>• <code>v++ --link</code></li> <li>• <code>vpl.impl</code></li> <li>• XSA generation</li> </ul>
compiler.axiDeadLockFree	Type: Boolean Default Value: TRUE	Avoid dead locks. This option is enabled by default for Vitis HLS.
compiler.deadlockDetection	Type: Boolean Default Value: FALSE	<p>Enables detection of kernel deadlocks during the simulation run as part of hardware emulation. The tool posts an Error message to the console and the log file when the application is deadlocked:</p> <pre>// ERROR!!! DEADLOCK DETECTED at 42979000 ns! SIMULATION WILL BE STOPPED! //</pre> <p>The message is repeated until the deadlock is terminated. You must manually terminate the application to end the deadlock condition.</p> <p><b>TIP:</b> When deadlocks are encountered during simulation, you can open the kernel code in Vitis HLS for additional deadlock detection and debug capability.</p> <p>Applies to:</p> <ul style="list-style-type: none"> <li>• <code>v++ --compile</code></li> <li>• Vitis HLS</li> <li>• <code>config_export</code></li> </ul>

Parameter Name	Valid Values	Description
compiler.emulationMode =<mode>	Type: String func   rtl	Indicates that the kernel should be compiled as RTL code for use in hardware emulation and hardware design, or as a C functional model with a SystemC wrapper for use in hardware emulation as described in <a href="#">Working with Functional Model of the HLS Kernel</a> . This option applies to <code>v++ --compile</code> . The default is to compile the kernel as RTL code.
compiler.enableIncrHwemu	Type: Boolean Default Value: FALSE	Use to enable incremental compilation of the hardware emulation <code>xclbin</code> when there are minor changes made to the platform. This enables a quick rebuild of the device binary for hardware emulation when the platform has been updated. Applies to: <ul style="list-style-type: none"><li>• <code>v++ --link</code></li><li>• <code>vpl.impl</code></li></ul>
compiler.errorOnHoldViolation	Type: Boolean Default Value: TRUE	After the last step of Vivado implementation, during timing analysis check, and clock scaling if needed. If hold violations are found, <code>v++</code> quits and returns an error by default, and does not generate an <code>xclbin</code> . This parameter lets you over ride the default behavior. Applies to: <ul style="list-style-type: none"><li>• <code>v++ --link</code></li><li>• <code>vpl.impl</code></li></ul>
compiler.fsanitize	Type: String Default Value: <empty>	Enables additional memory access checks for OpenCL kernels as described in <a href="#">Debugging OpenCL Kernels</a> . Valid values include: address, memory. Applies to Software Emulation and Debug.
compiler.interfaceRdBurstLen	Type: Int Range Default Value: 0	Specifies the expected length of AXI read bursts on the kernel AXI interface. This is used with option <code>compiler.interfaceRdOutstanding</code> to determine the hardware buffer sizes. Values are 1 through 256. Applies to: <ul style="list-style-type: none"><li>• <code>v++ --compile</code></li><li>• Vitis HLS</li><li>• config_interface</li></ul>
compiler.interfaceWrBurstLen	Type: Int Range Default Value: 0	Specifies the expected length of AXI write bursts on the kernel AXI interface. This is used with option <code>compiler.interfaceWrOutstanding</code> to determine the hardware buffer sizes. Values are 1 through 256. Applies to: <ul style="list-style-type: none"><li>• <code>v++ --compile</code></li><li>• Vitis HLS</li><li>• config_interface</li></ul>
compiler.interfaceRdOutstanding	Type: Int Range Default Value: 0	Specifies how many outstanding reads to buffer are on the kernel AXI interface. Values are 1 through 256. Applies to: <ul style="list-style-type: none"><li>• <code>v++ --compile</code></li><li>• Vitis HLS</li><li>• config_interface</li></ul>

Parameter Name	Valid Values	Description
compiler.interfaceWrOutstanding	Type: Int Range Default Value: 0	<p>Specifies how many outstanding writes to buffer are on the kernel AXI interface. Values are 1 through 256.</p> <p>Applies to:</p> <ul style="list-style-type: none"> <li>• <code>v++ --compile</code></li> <li>• Vitis HLS</li> <li>• <code>config_interface</code></li> </ul>
compiler.maxComputeUnits	Type: Int Default Value: -1	<p>Maximum compute units allowed in the system. The default is 60 compute units, or is specified in the hardware platform (.xsa) with the <code>numComputeUnits</code> property.</p> <p>The specified value overrides the default value or the hardware platform. The default value of -1 preserves the default.</p> <p>Applies to <code>v++ --link</code>.</p>
compiler.skipTimingCheckAndFrequencyScaling	Type: Boolean Default Value: FALSE	<p>This parameter causes the Vivado tool to skip the timing check and optional clock frequency scaling that occurs after the last step of implementation process, which is either <code>route_design</code> or <code>post-route phys_opt_design</code>.</p> <p>Applies to:</p> <ul style="list-style-type: none"> <li>• <code>v++ --link</code></li> <li>• <code>vpl.impl</code></li> </ul>
compiler.userPreCreateProjectTcl	Type: String Default Value: <empty>	<p>Specifies a Tcl script to run before creating the Vivado project in the Vitis build process.</p> <p>Applies to:</p> <ul style="list-style-type: none"> <li>• <code>v++ --link</code></li> <li>• <code>vpl.create_project</code></li> </ul>
compiler.userPreSysLinkOverlayTcl	Type: String Default Value: <empty>	<p>Specifies a Tcl script to run after opening the Vivado IP integrator block design, before running the compiler-generated <code>dr.bd.tcl</code> script in the Vitis build process.</p> <p>Applies to:</p> <ul style="list-style-type: none"> <li>• <code>v++ --link</code></li> <li>• <code>vpl.create_bd</code></li> </ul>
compiler.userPostSysLinkOverlayTcl	Type: String Default Value: <empty>	<p>Specifies a Tcl script to run after running the compiler-generated <code>dr.bd.tcl</code> script.</p> <p>Applies to:</p> <ul style="list-style-type: none"> <li>• <code>v++ --link</code></li> <li>• <code>vpl.update_bd</code></li> </ul>
compiler.userPostDebugProfileOverlayTcl	Type: String Default Value: <empty>	<p>Specifies a Tcl script to run after debug profile overlay insertion in Vivado IP integrator block design in the <code>vpl.update_bd</code> step.</p> <p>Applies to:</p> <ul style="list-style-type: none"> <li>• <code>v++ --link</code></li> <li>• <code>vpl.updated_bd</code></li> </ul>
compiler.worstNegativeSlack	Type: Float Default Value: 0	<p>During timing analysis check, this specifies the worst acceptable negative slack for the design, specified in nanoseconds (ns). When negative slack exceeds the specified value, the tool might try to scale the clock frequency to achieve timing results. This specifies an acceptable negative slack value instead of zero slack.</p> <p>Applies to:</p> <ul style="list-style-type: none"> <li>• <code>v++ --link</code></li> <li>• <code>vpl.impl</code></li> </ul>

Parameter Name	Valid Values	Description
compiler.xclDataflowFifoDepth	Type: Int Default Value: -1	Specifies the depth of FIFOs used in kernel data flow region. Applies to: <ul style="list-style-type: none"><li>• <code>v++ --compile</code></li><li>• Vitis HLS</li><li>• <code>config_dateflow</code></li></ul>
hw_emu.aie_shim_sol_path	Type: String Default Value: <empty>	For use by Versal platforms, this option specifies the path to the AI Engine SHIM Solution constraints file which is generated by the <code>aiecompiler</code> . Used during simulation, compilation, and elaboration, the file provides a logical mapping to the physical interface. This is needed for third-party simulators like Mentor Graphics Questa Advanced Simulator or Cadence Xcelium Logic Simulation.
hw_emu.compiledLibs	Type: String Default Value: <empty>	Uses mentioned <code>libs</code> for the specified simulator. Applies to Hardware Emulation and Debug.
hw_emu.debugMode	wdb Default Value: wdb	The default value is WDB and runs simulation in waveform mode. This option only works in combination with the <code>-g</code> or <code>--debug</code> options. Applies to Hardware Emulation and Debug.
hw_emu.enableProtocolChecker	Type: Boolean Default Value: FALSE	Enables the lightweight AXI protocol checker (lapc) during HW emulation. This is used to confirm the accuracy of any AXI interfaces in the design. Applies to Hardware Emulation and Debug.
hw_emu.json_device_file_path	Type: String Default Value: <empty>	For use by Versal platforms, this option specifies the path to the AI Engine JSON Device file located in the Vitis software installation area. Used during simulation, compilation, and elaboration, the file specifies the size of the AI Engine array. This is needed for third-party simulators like Mentor Graphics Questa Advanced Simulator or Cadence Xcelium Logic Simulation.
hw_emu.platformPath	Type: String Default Value: <empty>	Specifies the path to the custom platform directory. The <code>&lt;platformPath&gt;</code> directory should meet the following requirements to be used in platform creation: <ul style="list-style-type: none"><li>• The directory should contain a subdirectory called <code>ip_repo</code>.</li><li>• The directory should contain a subdirectory called <code>scripts</code> and this <code>scripts</code> directory should contain a <code>hw_em_util.tcl</code> file. The <code>hw_em_util.tcl</code> file should have the following two procedures defined in it:<ul style="list-style-type: none"><li>◦ <code>hw_em_util::add_base_platform</code></li><li>◦ <code>hw_em_util::generate_simulation_scripts_and_compile</code></li></ul></li></ul> Applies to Hardware Emulation and Debug.
hw_emu.post_sim_settings	Type: String	Specifies the path to a Tcl script that is used to configure the settings of the Vivado simulator prior to running hardware emulation. This script is run after the default configuration of the tool, but prior to launching simulation. You can use the Tcl script to override specific settings, or to custom configure the simulator as needed. Applies to Hardware Emulation and Debug.
hw_emu.reduceHwEmuCompileTime	Type: Boolean Default Value: FALSE	Move the generation of the top-level block design into the Generate Targets step of <code>v++ --link</code> . Applies to Hardware Emulation and Debug.

Parameter Name	Valid Values	Description
hw_emu.scDebugLevel	none   waveform   log   waveform_and_log Default Value: waveform_and_log	Sets the TLM transaction debug level of the Vivado logic simulator (xsim). <ul style="list-style-type: none"> <li>• NONE to disable TLM debug</li> <li>• LOG to dump TLM transaction log info into report file</li> <li>• WAVEFORM for enabling the TLM transaction waveform view</li> <li>• WAVEFORM_AND_LOG for both the Log Messages and Waveform view</li> </ul> Applies to Hardware Emulation and Debug.
hw_emu.simulator	XSIM   QUESTA Default Value: XSIM	Uses the specified simulator for the hardware emulation run. Applies to Hardware Emulation and Debug.

For example:

```
--advanced.param compiler.addOutputTypes="hw_export"
```



**TIP:** This option can be specified in a configuration file under the `[advanced]` section head using the following format:

```
[advanced]
param=compiler.addOutputTypes="hw_export"
```

### --advanced.prop

```
--advanced.prop <arg>
```

Specifies advanced kernel or solution properties for kernel compilation where `<arg>` is one of the values described in the table below.

**Table 40: Prop Options**

Property Name	Valid Values	Description
kernel.<kernel_name>.kernel_flags	Type: String Default Value: <empty>	Sets specific compile flags on the kernel <kernel_name>.
solution.device_repo_path	Type: String Default Value: <empty>	Specifies the path to a repository of hardware platforms. The <code>--platform</code> option with full path to the .xpm platform file should be used instead.
solution.kernel_compiler_margin	Type: Float Default Value: 12.5% of the kernel clock period.	The clock margin (in ns) for the kernel. This value is subtracted from the kernel clock period prior to synthesis to provide some margin for place and route delays.

### --advanced.misc

```
--advanced.misc <arg>
```

Specifies advanced tool directives for kernel compilation.

## --clock Options



**IMPORTANT!** The `--clock` options described here are supported on embedded processor platforms and Alveo™ platforms with fixed clocks, as described in [Managing Clock Frequencies](#).

You can specify the `--clock` option using either a clock ID from the platform shell, or by specifying a frequency for the kernel clock. When specifying the clock ID, the kernel frequency is defined by the frequency of that clock ID on the platform. When specifying the kernel frequency, the platform attempts to create the specified frequency by scaling one of the available fixed platform clocks. In some cases, the clock frequency can only be achieved in some approximation, and you can specify the `--clock.tolerance` or `--clock.default_tolerance` to indicate an acceptable range. If the available fixed clock cannot be scaled within the acceptable tolerance, a warning is issued and the kernel is connected to the default clock.

The `--clock.XXX` options provide a method for assigning clocks to kernels from the `v++` command line and locating the required kernel clock frequency source during the linking process. There are a number of options that can be used with increasing specificity. The order of precedence is determined by how specific a clock option is. The rules are listed in order from general to specific, where more specific rules take precedence over general rules:

- When no `--clock.XXX` option is specified, the platform default clock is applied to each compute unit (CU). For kernels with two clocks, clock ID 0 from the platform is assigned to `ap_clk`, and clock ID 1 is assigned to `ap_clk_2`.
- Specifying `--clock.defaultId=<id>` defines a specific clock ID for all kernels, overriding the platform default clock assignments.
- Specifying `--clock.defaultFreqHz=<Hz>` defines a specific clock frequency for all kernels that overrides a user specified default clock ID, and the platform default clock.
- Specifying `--clock.id=<id>:<cu_0>[.<clk_pin_0>] [,<cu_n>[.<clk_pin_n>]]` assigns a clock ID to a list of associated CUs, and optionally the clock pin for the CU.
- Specifying `--clock.freqHz=<Hz>:<cu_0>[.<clk_pin_0>] [,<cu_n>[.<clk_pin_n>]]` assigns the specified clock frequency to a list of associated CUs, and optionally the clock pin for the CU.

### --clock.defaultFreqHz

```
--clock.defaultFreqHz <arg>
```

Specifies a default clock frequency in Hz to use for all kernels. This lets you override the default platform clock and assign the specified clock frequency as the default. `<arg>` is specified as the clock frequency in Hz.

For example:

```
v++ --link --clock.defaultFreqHz 300000000
```



**TIP:** This option can be specified in a configuration file under the `[clock]` section head using the following format:

```
[clock]
defaultFreqHz=300000000
```

### --clock.defaultId

```
--clock.defaultId <arg>
```

Specifying `--clock.defaultId=<id>` defines a specific clock ID for all kernels, overriding the platform default clock. `<arg>` is specified as the clock ID from one of the clocks defined on the target platform, other than the default clock ID.



**TIP:** You can determine the available clock IDs and clock status for a target platform using the `platforminfo -v` command as described in [platforminfo Utility](#).

For example:

```
v++ --link --clock.defaultId 1
```



**TIP:** This option can be specified in a configuration file under the `[clock]` section head using the following format:

```
[clock]
defaultId=1
```

### --clock.defaultTolerance

```
--clock.defaultTolerance <arg>
```

Specifies a default clock tolerance as a value, or as a percentage of the default clock frequency. When specifying `clock.defaultFreqHz`, you can also specify the tolerance with either a value or percentage. This updates the timing constraints to reflect the accepted tolerance.

The tolerance value, `<arg>`, can be specified as a whole number, indicating the `clock.defaultFreqHz ± the specified tolerance`; or as a percentage of the default clock frequency specified as a decimal value.



**IMPORTANT!** The default clock tolerance is 5% when this option is not specified.

For example:

```
v++ --link --clock.defaultFreqHz 300000000 --clock.defaultTolerance 0.10
```



**TIP:** This option can be specified in a configuration file under the `[clock]` section head using the following format:

```
[clock]
defaultTolerance=0.10
```

### --clock.freqHz

```
--clock.freqHz <arg>
```

Specifies a clock frequency in Hz and assigns it to a list of associated compute units (CUs) and optionally specific clock pins on the CU. `<arg>` is specified as

```
<frequency_in_Hz>:<cu_0>[.<clk_pin_0>] [,<cu_n>[.<clk_pin_n>]]:
```

- `<frequency_in_Hz>`: Defines the clock frequency specified in Hz.
- `<cu_0>[.<clk_pin_0>] [,<cu_n>[.<clk_pin_n>]]`: Applies the defined frequency to the specified CUs, and optionally to the specified clock pin on the CU.

For example:

```
v++ --link --clock.freqHz 300000000:vadd_1,vadd_3
```



**TIP:** This option can be specified in a configuration file under the `[clock]` section head using the following format:

```
[clock]
freqHz=300000000:vadd_1,vadd_3
```

### --clock.id

```
--clock.id <arg>
```

Specifies an available clock ID from the target platform and assigns it to a list of associated compute units (CUs) and optionally specific clock pins on the CU. `<arg>` is specified as

```
<reference_ID>:<cu_0>[.<clk_pin_0>] [,<cu_n>[.<clk_pin_n>]]:
```

- `<reference_ID>`: Defines the clock ID to use from the target platform.



**TIP:** You can determine the available clock IDs for a target platform using the `platforminfo` utility as described in [platforminfo Utility](#).

- `<cu_0>[.<clk_pin_0>] [,<cu_n>[.<clk_pin_n>]]`: Applies the defined frequency to the specified CUs and optionally to the specified clock pin on the CU.

For example:

```
v++ --link --clock.id 1:vadd_1,vadd_3
```



**TIP:** This option can be specified in a configuration file under the `[clock]` section head using the following format:

```
[clock]
id=1:vadd_1,vadd_3
```

### --clock.tolerance

```
--clock.tolerance <arg>
```

Specifies a clock tolerance as a value, or as a percentage of the clock frequency. When specifying `--clock.freqHz`, you can also specify the tolerance with either a value or percentage. This updates the timing constraints to reflect the accepted tolerance. `<arg>` is specified as `<tolerance>:<cu_0>[.<clk_pin_0>] [,<cu_n>[.<clk_pin_n>]]`

- **<tolerance>:**

- Can be specified either as a whole number, indicating the `clock.freqHz` ± the specified tolerance value; or as a percentage of the clock frequency specified as a decimal value.
- `<cu_0>[.<clk_pin_0>] [,<cu_n>[.<clk_pin_n>]]`: Applies the defined clock tolerance to the specified CUs, and optionally to the specified clock pin on the CU.
- `<cu_0>[.<clk_pin_0>] [,<cu_n>[.<clk_pin_n>]]`: Applies the defined clock tolerance to the specified CUs, and optionally to the specified clock pin on the CU.



**IMPORTANT!** The default clock tolerance is 5% of the `clock.FreqHz` when this option is not specified.

For example:

```
v++ --link --clock.tolerance 0.10:vadd_1,vadd_3
```



**TIP:** This option can be specified in a configuration file under the `[clock]` section head using the following format:

```
[clock]
tolerance=0.10:vadd_1,vadd_3
```

## --connectivity Options

As discussed in [Linking the Kernels](#), there are a number of `--connectivity.XXX` options that let you define the topology of the FPGA binary, specifying the number of CUs, assigning them to SLRs, connecting kernel ports to global memory, and establishing streaming port connections. These commands are an integral part of the build process, critical to the definition and construction of the application.

## --connectivity.nk

```
--connectivity.nk <arg>
```

Where **<arg>** is specified as

```
<kernel_name>:#:<cu_name1>,<cu_name2>,...<cu_name#>.
```

This instantiates the specified number of CU (#) for the specified kernel (`kernel_name`) in the generated FPGA binary (.xclbin) file during the linking process. The `cu_name` is optional. If the `cu_name` is not specified, the instances of the kernel are simply numbered:

`kernel_name_1`, `kernel_name_2`, and so forth. By default, the Vitis compiler instantiates one compute unit for each kernel.

For example:

```
v++ --link --connectivity.nk vadd:3:vadd_A,vadd_B,vadd_C
```



**TIP:** This option can be specified in a configuration file under the `[connectivity]` section head using the following format:

```
[connectivity]
nk=vadd:3:vadd_A,vadd_B,vadd_C
```

## --connectivity.sc

```
--connectivity.sc <arg>
```

Create a streaming connection between two compute units through their AXI4-Stream interfaces. Use a separate `--connectivity.sc` option for each streaming interface connection. The order of connection must be from a streaming output port of the first kernel to a streaming input port of the second kernel. Valid values include:

```
<cu_name>.<streaming_output_port>:<cu_name>.<streaming_input_port>[:<fifo_depth>]
```

Where:

- `<cu_name>` is the compute unit name specified in the `--connectivity.nk` option. Generally this is `<kernel_name>_1` unless a different name was specified.
- `<streaming_output_port>/<streaming_input_port>` is the function argument for the compute unit port that is declared as an AXI4-Stream.
- `[:<fifo_depth>]` inserts a FIFO of the specified depth between the two streaming ports to prevent stalls. The value is specified as an integer.



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

For example, to connect the AXI4-Stream port `s_out` of the compute unit `mem_read_1` to AXI4-Stream port `s_in` of the compute unit `increment_1`, use the following:

```
--connectivity.sc mem_read_1.s_out:increment_1.s_in
```



**TIP:** This option can be specified in a configuration file under the `[connectivity]` section head using the following format:

```
[connectivity]
sc=mem_read_1.s_out:increment_1.s_in
```

The inclusion of the optional `<fifo_depth>` value lets the `v++` linker add a FIFO between the two kernels to help prevent stalls. This uses BRAM resources from the device when specified, but eliminates the need to update the HLS kernel to contain FIFOs. The tool also instantiates a Clock Converter (CDC) or Datawidth Converter (DWC) IP if the connections have different clocks, or different bus widths.

### --connectivity.slr

```
--connectivity.slr <arg>
```

Use this option to assign a CU to a specific SLR on the device. The option must be repeated for each kernel or CU being assigned to an SLR.



**IMPORTANT!** If you use `--connectivity.slr` to assign the kernel placement, then you must also use `--connectivity.sp` to assign memory access for the kernel.

Valid values include:

```
<cu_name>:<SLR_NUM>
```

Where:

- `<cu_name>` is the name of the compute unit as specified in the `--connectivity.nk` option. Generally this is `<kernel_name>_1` unless a different name was specified.
- `<SLR_NUM>` is the SLR number to assign the CU to. For example, SLR0, SLR1.

For example, to assign CU `vadd_2` to SLR2, and CU `fft_1` to SLR1, use the following:

```
v++ --link --connectivity.slr vadd_2:SLR2 --connectivity.slr fft_1:SLR1
```



**TIP:** This option can be specified in a configuration file under the `[connectivity]` section head using the following format:

```
[connectivity]
slr=vadd_2:SLR2
slr=fft_1:SLR1
```

## --connectivity.sp

```
--connectivity.sp <arg>
```

Use this option to specify the assignment of kernel arguments to system ports within the platform. A primary use case for this option is to connect kernel arguments to specific memory resources. A separate `--connectivity.sp` option is required to map each argument of a kernel to a particular memory resource. Any argument not explicitly mapped to a memory resource through the `--connectivity.sp` option is automatically connected to an available memory resource during the build process.



**RECOMMENDED:** Xilinx recommends specifying argument names when using the `--connectivity.sp` option as this provides the greatest connection flexibility. However, you can also specify kernel interface ports with this option.

Valid values include:

```
<cu_name>.<kernel_argument_name>:<sptag[min:max]>
```

Where:

- `<cu_name>` is the name of the compute unit as specified in the `--connectivity.nk` option. Generally this is `<kernel_name>_1` unless a different name was specified.
- `<kernel_argument_name>` is the name of the function argument for the kernel, or the compute unit interface port.
- `<sptag>` represents a system port tag, such as for memory controller interface names from the target platform. Valid `<sptag>` names include DDR, PLRAM, and HBM.
- `[min:max]` enables the use of a range of memory, such as DDR[0:2]. A single index is also supported: DDR[2].



**TIP:** The supported `<sptag>` and range of memory resources for a target platform can be obtained using the `platforminfo` command. Refer to [platforminfo Utility](#) for more information.

The following example maps the input argument (A) for the specified CU of the VADD kernel to DDR[0:3], input argument (B) to HBM[0:31], and writes the output argument (C) to PLRAM[2]:

```
v++ --link --connectivity.sp vadd_1.A:DDR[0:3] --connectivity.sp
vadd_1.B:HBM[0:31] \
--connectivity.sp vadd_1.C:PLRAM[2]
```



**TIP:** This option can be specified in a configuration file under the `[connectivity]` section head using the following format:

```
[connectivity]
sp=vadd_1.A:DDR[0:3]
sp=vadd_1.B:HBM[0:31]
sp=vadd_1.C:PLRAM[2]
```

### --connectivity.noc.connect

```
--connectivity.noc.connect <arg>
```

Where **<arg>** is in the form of

**<compute\_unit\_name>. <kernel\_interface\_name>:<noc\_interface>**, and specifies a connection between the PL kernel interface and the Versal NoC. Valid values are internal memory controllers, or master interfaces on the Versal NoC cell.

The Vitis compiler estimates kernel bandwidth requirements based on NoC connectivity and M\_AXI properties (datawidth \* clock freq) across the dynamic region, and automatically sets NoC configuration settings for read and write bandwidth, scaling as needed to avoid exceeding the available bandwidth.

For example:

```
[connectivity]
noc.read_bw=mm2s.M_AXI:2000.16
noc.write_bw=mm2s.M_AXI:2010.16
noc.connect=mm2s.M_AXI:M00_INI
```

### --connectivity.noc.read\_bw

```
--connectivity.noc.read_bw <arg>
```

Where **<arg>** is in the form

**<compute\_unit\_name>. <kernel\_interface\_name>:<Bandwidth>. <Avg\_burst\_length>**, and specifies both the bandwidth and burst length of the connection. The bandwidth is specified in MB/s.

This option specifies expected read traffic characteristics on M\_AXI interfaces to let you override the automatic Versal NoC configuration.

### --connectivity.noc.write\_bw

```
--connectivity.noc.write_bw <arg>
```

Where **<arg>** is in the form

**<compute\_unit\_name>. <kernel\_interface\_name>:<Bandwidth>. <Avg\_burst\_length>**, and specifies both the bandwidth and burst length of the connection. The bandwidth is specified in MB/s.

This option specifies expected write traffic characteristics on M\_AXI interfaces to let you override the automatic Versal NoC configuration.

### --connectivity.connect

```
--connectivity.connect <X:Y>
```

This option can be used to make connections through the Vivado IP integrator, but `v++` does not perform any error checking on the specified connections. Use this to specify general connections between kernels and non-AXI elements of the target platform, such as connections to GT ports.

The X and Y connections must be specified as arguments compatible with either the IP integrator `connect_bd_net` or `connect_bd_intf_net` commands. The specific format of `<X:Y>` is:

```
src/hierarchy_name/cell_name/pin_name:dst/hierarchy_name/cell_name/pin_name
```

These cannot include connections between AXI4-Stream interfaces which require the use of `--connectivity.sc`, or M\_AXI interfaces which require the use of `--connectivity.sp` as described above.



**TIP:** This option can be specified in a configuration file under the `[connectivity]` section head using the following format:

```
[connectivity]
connect=<X:Y>
```

## --debug Options

This option enables debug IP core insertion in the device binary (`.xclbin`) for hardware debugging. This option lets you specify the type of debug core to add, and which compute unit and interfaces to monitor with ChipScope™ as described in [Debugging During Hardware Execution](#). The `--debug .xxx` options lets you attach AXI protocol checkers and System ILA cores at the interfaces to kernels or specific compute units (CUs) for debugging and performance monitoring purposes:

- The System Integrated Logic Analyzer (ILA) provides transaction level visibility into an accelerated kernel or function running on hardware. AXI traffic of interest can also be captured and viewed using the [System ILA](#) core.
- The AXI Protocol Checker debug core is designed to monitor AXI interfaces on the accelerated kernel. When attached to an interface of a CU, the [AXI Protocol Checker](#) actively checks for protocol violations and provides an indication of which violation occurred.

The `--debug .xxx` commands can be specified in a configuration file under the `[debug]` section head using the following format as an example:

```
protocol=all:all          # Protocol analyzers on all CUs
protocol=cu2:port3        # Protocol analyzer on port3 of cu2
chipscope=cu2              # ILA on cu2
```

The various options of `--debug` include the following:

**--debug.aie.chipscope**

```
--debug.aie.chipscope <interface_name> | <adf_graph_arg_name>
```

Enables hardware debug for the Versal AI Engine through ChipScope. The `<interface_name>` argument applies to non-PL kernel interfaces such as AI Engine PLIO interfaces, or AXIS interfaces. The `<adf_graph_arg_name>` specifies arguments of the graph.

**--debug.chipscope**

```
--debug.chipscope <cu_name>[:<interface_name>]
```

Adds the System Integrated Logic Analyzer debug core to the specified CUs in the design.



**IMPORTANT!** The `--debug.chipscope` option requires the `<cu_name>` to be specified and does not accept the keyword `all`. You can optionally specify an `<interface_name>`.

For example, the following command adds an ILA core to the `vadd_1` CU:

```
v++ --link --debug.chipscope vadd_1
```

**--debug.list\_ports**

Shows a list of valid compute units and port combinations in the current design. This is informational to help you with crafting a command line or config file for the `--debug` command.

This option needs to be specified during linking, but does not run the linking process. The required elements of the command line are shown in the following example, which returns the available ports when linking the specified kernels with the listed platform:

```
v++ --platform <platform> --link --debug.list_ports <kernel.xo>
```

**--debug.protocol**

```
--debug.protocol all|<cu_name>[:<interface_name>]
```

Adds the AXI Protocol Checker debug core to the design. This can be specified with the keyword `all`, or the `<cu_name>` and optional `<interface_name>` to add the protocol checker to the specified CU and interface.

For example:

```
v++ --link --debug.protocol all
```

## --drc Options

This option lets you manage the Design Rule Check (DRC) messages returned by the Vivado Design Suite during implementation of the design. Messages can be disabled or enabled, reduced in severity, or waived to allow the design to proceed.

### --drc.disable

```
--drc.disable <arg>
```

Lets you disable the DRC message specified by the DRC ID. Disabled DRCs are not checked.

For example:

```
v++ --link --drc.disable TIMING-18
```

### --drc.enable

```
--drc.enable <arg>
```

Lets you enable the DRC message specified by the DRC ID. DRC checks that have been disabled can be re-enabled as needed..

For example:

```
v++ --link --drc.enable TIMING-18
```

### --drc.severity

```
--drc.severity <arg>
```

Change the severity of a DRC check. For example, instead of a {CRITICAL WARNING} a violation is only reported as a WARNING. The DRC must be specified by ID, and the new severity to apply.

For example:

```
v++ --link --drc.severity TIMING-18:Warning
```

### --drc.waive

```
--drc.waive <arg>
```

Lets you waive the specified DRC ID. This lets the Vivado tool proceed through implementation, ignoring DRCs that might otherwise prevent completion of the design. A waived rule is still checked, but it is marked as waived.

For example:

```
v++ --link --drc.waive TIMING-18
```

## --hls Options

The `--hls.XXX` options described below are used to specify options for the Vitis HLS synthesis process invoked during kernel compilation.

### --hls.clock

```
--hls.clock <arg>
```

Specifies a frequency in Hz at which the listed kernel(s) should be compiled by Vitis HLS.

Where `<arg>` is specified as:

```
<frequency_in_Hz>:<cu_name1>,<cu_name2>,...<cu_nameN>
```

- `<frequency_in_Hz>`: Defines the kernel frequency specified in Hz.
- `<cu_name1>,<cu_name2>,...`: Defines a list of kernels or kernel instances (CUs) to be compiled at the specified target frequency.

For example:

```
v++ -c --hls.clock 300000000:mmult,mmadd --hls.clock 100000000:fifo_1
```



**TIP:** This option can be specified in a configuration file under the `[hls]` section head using the following format:

```
[hls]
clock=300000000:mmult,mmadd
clock=100000000:fifo_1
```

### --hls.export\_mode

```
--hls.export_mode <file_type>:<file_path>
```

Specifies the RTL export mode for Vitis HLS and the path and name of the exported file. As a `v++` compiler option, the only supported `<file_type>` is `XO`.

For example:

```
v++ --hls.export_mode xo:./kernel.xo
```



**TIP:** This option can be specified in a configuration file under the `[hls]` section head using the following format:

```
[hls]
export_mode=xo:./kernel.xo
```

### --hls.export\_project

```
--hls.export_project <arg>
```

Specifies a directory where the Vitis HLS project setup script is exported.

For example:

```
v++ --hls.export_project ./hls_export
```



**TIP:** This option can be specified in a configuration file under the `[hls]` section head using the following format:

```
[hls]
export_project=./hls_export
```

### --hls.jobs

```
--hls.jobs <arg>
```

Specifies the number of jobs for launching HLS runs.

This option specifies the number of parallel jobs Vitis HLS uses to synthesize the RTL kernel code. Increasing the number of jobs allows the tool to spawn more parallel processes and complete faster.

For example:

```
v++ --hls.jobs 4
```



**TIP:** This option can be specified in a configuration file under the `[hls]` section head using the following format:

```
[hls]
jobs=4
```

### --hls.lsf

```
--hls.lsf <arg>
```

Specifies a `bsub` command to submit a job to LSF for HLS runs.

Specifies the `bsub` command line as a string to pass to an LSF cluster. This option is required to use the IBM Platform Load Sharing Facility (LSF) for Vitis HLS synthesis.

For example:

```
v++ --compile --hls.lsf '{bsub -R \"select[type=X86_64]\\" -N -q medium}'
```



**TIP:** This option can be specified in a configuration file under the `[hls]` section head using the following format:

```
[hls]
lsf='{bsub...
```

### --hls.post\_tcl

```
--hls.post_tcl <arg>
```

Specifies a Tcl file containing Tcl commands for `vitis_hls` to source after `csynth_design`.

For example:

```
v++ --hls.post_tcl ./runPost.tcl
```



**TIP:** This option can be specified in a configuration file under the `[hls]` section head using the following format:

```
[hls]
post_tcl=./runPost.tcl
```

### --hls.pre\_tcl

```
--hls.pre_tcl <arg>
```

Specifies a Tcl file containing Tcl commands for `vitis_hls` to source before running `c synth_design`.

For example:

```
v++ --hls.pre_tcl ./runPre.tcl
```

Where `runPre.tcl` contains the following commands to configure `m_axi` interfaces in Vitis HLS:

```
config_interface -m_axi_auto_max_ports=1
config_interface -m_axi_max_bitwidth 512
```



**TIP:** This option can also be specified in a configuration file under the `[hls]` section head using the following format:

```
[hls]  
pre_tcl=./runPre.tcl
```

## --linkhook Options

The `--linkhook.XXX` options described below are used to specify Tcl scripts to run at specific steps during the Vitis linking process. Valid steps can be determined using the `--linkhook.list_steps` command as described below.

### **--linkhook.custom**

```
--linkhook.custom <step name, path to script file>
```

Specifies a Tcl script to execute at a predefined point in the build process. The path to specify the script can be an absolute path, or partial path relative to the build directory.

For example, the following command runs the specified Tcl script before the SysLink step in the build:

```
v++ -l --linkhook.custom preSysLink ./runScript.tcl
```

### **-linkhook.do\_first**

```
--linkhook.do_first <step name, path to script file>
```

Specifies a Tcl script to execute before the given step name. The path to specify the script can be an absolute path, or partial path relative to the build directory.

For example, the following command runs the specified Tcl script before the `place_design` step in the build:

```
v++ -l --linkhook.do_first vpl.impl.place_design,runScript.tcl
```

### **-linkhook.do\_last**

```
--linkhook.do_last <step name, path to script file>
```

Specifies a Tcl script to execute immediately after the given step completes. The path to specify the script can be an absolute path, or partial path relative to the build directory.

For example, the following command runs the specified Tcl script after the `place_design` step in the build:

```
v++ -l --linkhook.do_last vpl.impl.place_design,runScript.tcl
```

### -linkhook.list\_steps

```
--linkhook.list_steps
```

List default and optional build steps that support script hooks for a specified target. This command requires the `--target` to be specified as well as the `--link` option.

For example:

```
v++ --target hw -l --linkhook.list_steps
```

The command returns both default steps that are always enabled during the build process, and optional steps that you can enable if needed. Refer to [Managing Vivado Synthesis and Implementation Results](#) for directions on enabling optional steps.

---

## --package Options

### Introduction

As described in [Packaging the System](#), the `v++ --package`, or `-p` step, generates and packages the final product at the end of the `v++` compile and link build process. This is a required step for all embedded platforms, including Versal devices, AI Engine, and Zynq UltraScale+ MPSoC devices.

The syntax of the `--package` command for Versal platforms is as follows:

```
v++ --package -t <sw_emu | hw_emu | hw> --platform <platform> input.xsa \
[ -o output.xclbin --package.<options> ]
```



**TIP:** Package command options can be specified in a configuration file for use with the `--config` option, as discussed in the [Vitis Compiler Configuration File](#).

For non-Versal platforms the syntax for the `--package` command is:

```
v++ --package -t <sw_emu | hw_emu | hw> --platform <platform> input.xclbin \
[ -o output.xclbin --package.<options> ]
```

The various options to specify as `--package.<options>` as shown in the syntax above include the following:

**--package.aie\_debug\_port**

```
--package.aie_debug_port <arg>
```

Where `<arg>` specifies a TCP port where emulator listens for incoming connections from the debugger to debug Versal AI Engine cores. The default port value is 10100.

For example:

```
v++ -l --package.aie_debug_port 1440
```

**--package.bl31\_elf**

```
--package.bl31_elf <arg>
```

Where `<arg>` specifies the absolute or relative path to Arm trusted FW ELF that executes on A72 #0 core. If this option is not specified, then the Vitis compiler searches for the bl31 in the platform.

For example:

```
v++ -l --package.bl31_elf ./arm_trusted.elf
```

**--package.boot\_mode**

```
--package.boot_mode <arg>
```

Where `<arg>` specifies `<ospi | qspi | sd>` Boot mode used for running the application in emulation or on hardware. For embedded platforms, the default boot mode is `sd`. For Data Center platforms, it is either `qspi` or `ospi` as appropriate.



**TIP:** `ospi` is for use with Versal Data Center platforms only.

For example:

```
v++ -l --package.boot_mode sd
```

**--package.defer\_aie\_run**

```
--package.defer_aie_run
```

Where this option specifies that the Versal AI Engine cores are enabled by an embedded processor (PS) application. When not specified, the tool generates CDO commands to enable the AI Engine cores during PDI load instead. By default this option is disabled, or FALSE.

For example:

```
v++ -l --package.defer_aie_run
```

### --package.domain

```
--package.domain <arg>
```

Where `<arg>` specifies a domain name. If this option is not specified, then the Vitis compiler picks up the default domain from the software platform (SPFM) file.

For example:

```
v++ -l --package.domain xrt
```

### --package.dtb

```
--package.dtb <arg>
```

Where `<arg>` specifies the absolute or relative path to device tree binary (DTB) used for loading Linux on the APU. If this options is not specified, then Vitis compiler searches for the `dtb` in the platform.

For example:

```
v++ -l --package.dtb ./device_tree.image
```

### --package.emu\_ps

```
--package.emu_ps <x86 | qemu>
```

Specifies that the PS application code has been compiled for the x86 processor instead of an Arm® processor, and will run under the system OS for C-simulation instead of PetaLinux/QEMU. The default setting is x86.




---

**IMPORTANT!** This option is only valid for the `sw_emu` target.

---

For example:

```
v++ -l --package.emu_ps x86
```

### --package.enable\_aie\_debug

```
--package.enable_aie_debug
```

When enabled, the tool generates CDO commands to stop the AI Engine cores during PDI load. By default this option is disabled, or FALSE.

For example:

```
v++ -l --package.enable_aie_debug
```

### --package.image\_format

```
--package.image_format <arg>
```

Where `<arg>` specifies `<ext4 | fat32>` output image file format used on the SD card. For embedded platforms with a Linux domain, the default image format is `ext4`. For all others, the image format is `fat32`.

- `ext4`: Linux file system
- `fat32`: Windows file system



**IMPORTANT!** EXT4 format is not supported on Windows.

For example:

```
v++ -l --package.image_format fat32
```

### --package.kernel\_image

```
--package.kernel_image <arg>
```

Where `<arg>` specifies the absolute or relative path to a Linux kernel image file. Overrides the existing image available in the platform. The platform image file is available for download from [xilinx.com](http://xilinx.com). Refer to the [Vitis Software Platform Installation](#) for more information. If this option is not specified, then the Vitis compiler copies the Linux image from the platform to the SD card folder.

For example:

```
v++ -l --package.kernel_image ./kernel_image
```

### --package.no\_image

```
--package.no_image
```

Bypass SD card image creation. Valid for `--package.boot_mode sd`. By default this option is disabled, or FALSE.

### --package.out\_dir

```
--package.out_dir <arg>
```

Where <arg> specifies the absolute or relative path to the output directory of the --package command. The default output directory is the directory from which the Vitis compiler is launched.

For example:

```
v++ -l --package.out_dir ./out_dir
```

### --package.ps\_debug\_port

```
--package.ps_debug_port <arg>
```

Where <arg> specifies the TCP port where emulator listens for incoming connections from the debugger to debug PS cores.

For example:

```
v++ -l --package.debug_port 3200
```

### --package.ps\_elf

```
--package.ps_elf <arg>
```

Where <arg> specifies <path\_to\_elf\_file,core>.

- **path\_to\_elf\_file:** Specifies the ELF file for the PS core.
- **core:** Indicates the PS core it should run on.

Used when a baremetal ELF file is running on a device processor core. This option specifies an ELF file and processor core pair to be included in the boot image. The available processors for supported devices are listed below:

- Versal processor core values include: a72-0, a72-1, a72-2, and a72-3.
- Zynq UltraScale+ MPSoC processor core values include: a53-0, a53-1, a53-2, a53-3, r5-0, and r5-1.
- Zynq-7000 processor core values include: a9-0 and a9-1.



*TIP: Specify the option separately for each ELF/Core pair.*

For example:

```
v++ -l --package.ps_elf a53_0.elf,a53-0 --package.ps_elf r5_0.elf,r5-0
```

### --package.rootfs

```
--package.rootfs <arg>
```

Where <arg> specifies the absolute or relative path to a processed Linux root file system file. The platform RootFS file is available for download from Xilinx.com. Refer to the [Vitis Software Platform Installation](#) for more information. If this option is not specified, then the Vitis compiler picks up the default `rootfs` path from the software platform (SPFM) file.

For example:

```
v++ -l --package.rootfs ./rootfs.ext4
```

#### --package.sd\_dir

```
--package.sd_dir <arg>
```

Where <arg> specifies a folder to package into the `sd_card` directory/image. The contents of the directory are copied to a sub-folder of the `sd_card` folder.

For example:

```
v++ -l --package.sd_dir ./test_data
```

#### --package.sd\_file

```
--package.sd_file <arg>
```

Where <arg> specifies an ELF or other data file to package into the `sd_card` directory/image. This option can be used repeatedly to specify multiple files to add to the `sd_card`.

For example:

```
v++ -l --package.sd_file ./arm_trusted.elf
```

#### --package.uboot

```
--package.uboot <arg>
```

Where <arg> specifies a path to U-Boot ELF file which overrides a platform U-Boot. If this option is not specified, then the Vitis compiler searches for the `uboot` in the platform.

For example:

```
v++ -l --package.uboot ./uboot.elf
```

## --profile Options

As discussed in [Enabling Profiling in Your Application](#), there are a number of `--profile` options that let you enable profiling of the application and kernel events during runtime execution. This option enables capturing transaction details for data traffic between the kernel and host, kernel stalls, the execution times of kernels and compute units (CUs), as well as monitoring activity in Versal AI Engines.



**IMPORTANT!** Using the `--profile` option in `v++` also requires the addition of one of the `profile` or `trace` options in the `xrt.ini` file. Refer to [xrt.ini File](#) for more information.

The `--profile` commands can be specified in a configuration file under the `[profile]` section head using the following format, for example:

```
[profile]
data=all:all:all          # Monitor data on all kernels and CUs
data=k1:all:all           # Monitor data on all instances of kernel k1
data=k1:cu2:port3         # Specific CU master
data=k1:cu2:port3:counters # Specific CU master (counters only, no trace)
memory=all                # Monitor transfers for all memories
memory=<sptag>           # Monitor transfers for the specified memory
stall=all:all              # Monitor stalls for all CUs of all kernels
stall=k1:cu2               # Stalls only for cu2
exec=all:all               # Monitor execution times for all CUs
exec=k1:cu2                # Execution times only for cu2
aie=all                   # Monitor all AIE streams
aie=DataIn1               # Monitor the specific input stream in the SDF
graph                     # Monitor specific stream interface
aie=M02_AXIS
```

The various options of the command are described below:

### --profile.aie:<arg>

Enables profiling of AI Engine streams in adaptive data flow (ADF) applications, where `<arg>` is:

```
<ADF_graph_argument | pin_name | all>
```

- `<ADF_graph_argument>`: Specifies an argument name from the ADF graph application.
- `<pin_name>`: Indicates a port on an AI Engine kernel.
- `<all>`: Indicates monitoring all stream connections in the ADF application.

For example, to monitor the `DataIn1` input stream use the following command:

```
v++ --link --profile.aie:DataIn1
```

**--profile.data:<arg>**

Enables monitoring of data ports through monitor IP that are added into the design. This option needs to be specified during linking.

Where <arg> is:

```
[<kernel_name>|all] : [<cu_name>|all] : [<interface_name>|all] (:[counters|all])
```

- [<kernel\_name>|all] defines either a specific kernel to apply the command to. However, you can also specify the keyword all to apply the monitoring to all existing kernels, compute units, and interfaces with a single option.
- [<cu\_name>|all] when <kernel\_name> has been specified, you can also define a specific CU to apply the command to, or indicate that it should be applied to all CUs for the kernel.
- [<interface\_name>|all] defines the specific interface on the kernel or CU to monitor for data activity, or monitor all interfaces.
- [<counters|all] is an optional argument, as it defaults to all when not specified. It allows you to restrict the information gathering to just counters for larger designs, while all will include the collection of actual trace information.

For example, to assign the data profile to all CUs and interfaces of kernel k1 use the following command:

```
v++ --link --profile.data:k1:all:all
```

**--profile.exec:<arg>**

This option records the execution times of the kernel and provides minimum port data collection during the system run. This option needs to be specified during linking.



**TIP:** The execution time of a kernel is collected by default when --profile.data or --profile.stall is specified. You can specify --profile.exec for any CUs not covered by data or stall.

The syntax for exec profiling is:

```
[<kernel_name>|all] : [<cu_name>|all] (:[counters|all])
```

For example, to profile to execution of cu2 for kernel k1 use the following command:

```
v++ --link --profile.exec:k1:cu2
```

**--profile.stall:<arg>**

**IMPORTANT!** This option must be specified during both v++ compilation and linking.

Adds stall monitoring logic to the device binary (.xclbin) which requires the addition of stall ports on the kernel interface. To facilitate this, the `stall` option must be specified during both compilation and linking.

The syntax for `stall` profiling is:

```
[<kernel_name>|all] : [<cu_name>|all] (:[counters|all])
```

For example, to monitor stalls of `cu2` for kernel `k1` use the following command:

```
v++ --compile -k k1 --profile.stall ...  
v++ --link --profile.stall:k1:cu2 ...
```

#### --profile.trace\_memory:<arg>



**TIP:** This option applies to hardware build targets (`-t=hw`) only, and should not be used for software or hardware emulation flows.

When building the hardware target (`-t=hw`), use this option to specify the type and amount of memory to use for capturing trace data. You can specify the argument as follows:

```
<FIFO>:<size>|<MEMORY>[<n>] [ :<SLR>]
```

This argument specifies memory type to use for capturing trace data. Use the `--profile.trace_memory` command to define the type or memory to use, with the `trace_buffer_size` switch in the `xrt.ini` file to define the amount of memory to use as described in [xrt.ini File](#). The default memory type used is the first memory defined in the platform, and the default buffer size is 1 MB.

When `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.

- **FIFO:<size>:** Specified in KB. The maximum is 128K, although 64K is the maximum recommended.
- **Memory:** Specifies the type and number of memory resource on the platform. Memory resources for the target platform can be identified with the `platforminfo` command. Supported memory types include HBM, DDR, PLRAM, HP, ACP, MIG, and MC\_NOC. For example, `DDR[1]`.
- **[:<SLR>]:** Optionally indicates that CUs assigned to the specified `<SLR>` should use the DDR or HBM resources specified in the `<MEMORY>` field. Note that this syntax can only be used with DDR or HBM memory banks.

You can specify the `--profile.trace_memory` command with just the memory size and unit such as `FIFO:8k`, or specify the memory bank such as `DDR[0]` or `HBM[3]`. In this case, the profile data for all CUs are captured in the specified memory.

Or you can specify the memory to use to capture profile data, and the SLR assignment for that memory. In this case, the SLR assignment indicates that any CUs assigned to the specified SLR should have profile data captured in the specified memory. This is shown in the following config file example:

```
[profile]
trace_memory=DDR[1]:SLR0
trace_memory=DDR[2]:SLR1
```

In the example above, profile data for CUs assigned to SLR0 are captured in DDR bank 1, and CUs assigned to SLR1 are captured in DDR bank 2. CUs are assigned to SLRs using the `--connectivity.slr` command as described in [Assigning Compute Units to SLRs on Alveo Accelerator Cards](#).

 **IMPORTANT!** You cannot mix DDR and HBM memory banks in a single design, and when you specify the `<SLR>` syntax you must use that syntax for all `trace_memory` commands in the design.

## --vivado Options

The `--vivado.XXX` options are used to configure the Vivado tools for synthesis and implementation of your device binary (`.xclbin`). For instance, you can specify the number of jobs to spawn, LSF commands to use for implementation runs, or the specific implementation strategies to use. You can also configure optimization, placement, timing, or specify which reports to output.

 **IMPORTANT!** Familiarity with the Vivado Design Suite is required to make the best use of these options.  
See the Vivado Design Suite User Guide: Implementation ([UG904](#)) for more information.

### --vivado.impl.jobs

```
--vivado.impl.jobs <arg>
```

Specifies the number of parallel jobs the Vivado Design Suite uses to implement the device binary. Increasing the number of jobs allows the Vivado implementation step to spawn more parallel processes and complete faster jobs.

For example:

```
v++ --link --vivado.impl.jobs 4
```

### --vivado.impl.lsf

```
--vivado.impl.lsf <arg>
```

Specifies the `bsub` command line as a string to pass to an LSF cluster. This option is required to use the IBM Platform Load Sharing Facility (LSF) for Vivado implementation.

For example:

```
v++ --link --vivado.impl.lsf '{bsub -R \\"select[type=X86_64]\\\" -N -q  
medium}'
```

### --vivado.impl.strategies

```
--vivado.impl.strategies <arg>
```

Specifies a comma-separated list of strategy names for Vivado implementation runs. Use `ALL` to run all available implementation strategies. This lets you run a variety of implementation strategies at the same time during the build process and allows you to more quickly resolve the timing and routing issues of the design.



**IMPORTANT!** *Running ALL implementation strategies might launch 30 or more runs in the Vivado tool. This can be a tremendous drain on resources, and is not advised. You can prevent this by defining a set of 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.*

### --vivado.param

```
--vivado.param <arg>
```

Specifies parameters for the Vivado Design Suite to be used during synthesis and implementation of the FPGA binary (`xclbin`).



**TIP:** You can use the `report_param` Tcl command in the Vivado tool to identify the available parameters.

### --vivado.prop

```
--vivado.prop <arg>
```

Specifies properties for the Vivado Design Suite to be used during synthesis and implementation of the FPGA binary (`xclbin`).

Table 41: Prop Options

Property Name	Valid Values	Description
vivado.prop <object_type>.<object_name>.<prop_name>	Type: Various	<p>This allows you to specify any property used in the Vivado hardware compilation flow.</p> <p>&lt;object_type&gt; is run fileset file project.</p> <p>The &lt;object_name&gt; and &lt;prop_name&gt; values are described in <i>Vivado Design Suite Properties Reference Guide</i> (<a href="#">UG912</a>).</p> <p>Examples:</p> <pre>vivado.prop run.impl_1. {STEPS.PLACE_DESIGN.ARGS.MORE OPTIONS}={-no_bufg_opt}</pre> <pre>vivado.prop fileset. current.top=foo</pre> <p>If &lt;object_type&gt; is set to file, current is not supported.</p> <p>If &lt;object_type&gt; is set to run, the special value of __KERNEL__ can be used to specify run optimization settings for ALL kernels, instead of the need to specify them one by one.</p>

For example, from the command line:

```
v++ --link --vivado.prop run.impl_1.STEPS.PHYS_OPT_DESIGN.IS_ENABLED=true  
--vivado.prop run.impl_1.STEPS.PHYS_OPT_DESIGN.ARGS.DIRECTIVE=Explore  
--vivado.prop run.impl_1.STEPS.PLACE_DESIGN.TCL.PRE=/.../xxx.tcl
```

The example above enables the optional PHYS\_OPT\_DESIGN step as part of the Vivado implementation process, sets the Explore directive for that step, and specifies a Tcl script to run before the PLACE\_DESIGN step.



**TIP:** As described in [Managing Vivado Synthesis and Implementation Results](#), each step in the Vivado synthesis and implementation process can have a Tcl prescript to run before the step, and a Tcl postscript to run after the step. This lets you customize the build process by inserting pre-processing or post-processing Tcl commands around the different steps. These scripts can be specified as shown in the example above.

These options can also be specified in a configuration file under the [vivado] section head using the following format:

```
[vivado]  
prop=run.impl_1.STEPS.PHYS_OPT_DESIGN.IS_ENABLED=true  
prop=run.impl_1.STEPS.PHYS_OPT_DESIGN.ARGS.DIRECTIVE=Explore  
prop=run.impl_1.STEPS.PLACE_DESIGN.TCL.PRE=/.../xxx.tcl
```

---

 **IMPORTANT!** Some Vivado properties have spaces in their name, such as MORE OPTIONS and Tcl syntax requires these properties to be enclosed in braces, {}. However, the placement of the braces in the --vivado options is important. You must surround the complete property name with braces, rather than just a portion of it. For instance, the correct placement would be:

```
--vivado_prop run.impl_1.{STEPS.PLACE_DESIGN.ARGS.MORE_OPTIONS}={ -  
no_bufg_opt}
```

While the following would result in an error during the build process:

```
--vivado.prop "run.impl_1.{STEPS.PLACE_DESIGN.ARGS.MORE_OPTIONS}={ -  
no_bufg_opt}"
```

---

### --vivado.synth.jobs

```
--vivado.synth.jobs <arg>
```

Specifies the number of parallel jobs the Vivado Design Suite uses to synthesize the device binary. Increasing the number of jobs allows the Vivado synthesis to spawn more parallel processes and complete faster jobs.

For example:

```
v++ --link --vivado.synth.jobs 4
```

### --vivado.synth.lsf

```
--vivado.synth.lsf <arg>
```

Specifies the bsub command line as a string to pass to an LSF cluster. This option is required to use the IBM Platform Load Sharing Facility (LSF) for Vivado synthesis.

For example:

```
v++ --link --vivado.synth.lsf '{bsub -R \"select[type=X86_64]\" -N -q  
medium}'
```

---

## Vitis Compiler Configuration File

A configuration file can also be used to specify the Vitis compiler options. A configuration file provides an organized way of passing options to the compiler by grouping similar switches together, and minimizing the length of the v++ command line. Some of the features that can be controlled through config file entries include:

- HLS options to configure kernel compilation

- Connectivity directives for system linking such as the number of kernels to instantiate or the assignment of kernel ports to global memory
- Package directives to configure the [--package Options](#).
- Directives for the Vivado Design Suite to manage hardware synthesis and implementation.

In general, any `v++` command option can be specified in a configuration file. However, the configuration file supports defining sections containing groups of related commands to help manage build options and strategies. The following table lists the defined sections.



**TIP:** While both compilation (`v++ -c`) and linking (`v++ -l`) commands can be put into a single configuration file, error checking may return errors related to linking during the compilation process. Therefore you are recommended to keep separate configuration files for compilation and linking.

**Table 42: Section Tags of the Configuration File**

Section Name	Compiler/Linker	Description
[advanced]	either	<b>--advanced Options:</b> <ul style="list-style-type: none"> <li>• misc</li> <li>• param</li> <li>• prop</li> </ul>
[clock]	compiler	<b>--clock Options:</b> <ul style="list-style-type: none"> <li>• defaultFreqHz</li> <li>• defaultID</li> <li>• defaultTolerance</li> <li>• freqHz</li> <li>• id</li> <li>• tolerance</li> </ul>
[connectivity]	linker	<b>--connectivity Options:</b> <ul style="list-style-type: none"> <li>• nk</li> <li>• sc</li> <li>• slr</li> <li>• sp</li> <li>• connect</li> </ul>
[debug]	linker	<b>--debug Options</b> <ul style="list-style-type: none"> <li>• chipscope</li> <li>• list_ports</li> <li>• protocol</li> </ul>
[hls]	compiler	HLS directives <b>--hls Options:</b> <ul style="list-style-type: none"> <li>• clock</li> <li>• export_mode</li> <li>• export_project</li> <li>• jobs</li> <li>• lsf</li> <li>• post_tcl</li> <li>• pre_tcl</li> </ul>

Table 42: Section Tags of the Configuration File (cont'd)

Section Name	Compiler/Linker	Description
[linkhook]	linker	<b>--linkhook Options</b> <ul style="list-style-type: none"> <li>• custom</li> <li>• do_first</li> <li>• do_last</li> <li>• list_steps</li> </ul>
[package]	packager	<b>--package Options</b> <ul style="list-style-type: none"> <li>• aie_debug_port</li> <li>• bl31_elf</li> <li>• boot_mode</li> <li>• defer_aie_run</li> <li>• domain</li> <li>• dtb</li> <li>• enable_aie_debug</li> <li>• image_format</li> <li>• kernel_image</li> <li>• no_image</li> <li>• out_dir</li> <li>• ps_debug_port</li> <li>• ps_elf</li> <li>• rootfs</li> <li>• sd_dir</li> <li>• sd_file</li> <li>• uboot</li> </ul>
[profile]	linker	<b>--profile Options</b> <ul style="list-style-type: none"> <li>• aie</li> <li>• data</li> <li>• exec</li> <li>• stall</li> <li>• trace_memory</li> </ul>
[vivado]	linker	<b>--vivado Options:</b> <ul style="list-style-type: none"> <li>• impl.jobs</li> <li>• impl.lsf</li> <li>• impl.strategies</li> <li>• param</li> <li>• prop</li> <li>• synth.jobs</li> <li>• synth.lsf</li> </ul>



**TIP:** Comments can be added to the configuration file by starting the line with a "#". The end of a section is specified by an empty line at the end of the section.

Because the `v++` command supports multiple config files on a single `v++` command line, you can partition your configuration files into related options that define compilation and linking strategies or Vivado implementation strategies, and apply multiple config files during the build process.

Configuration files are optional. There are no naming restrictions on the files and the number of configuration files can be zero or more. All `v++` options can be put in a single configuration file if desired. However, grouping related switches into separate files can help you organize your build strategy. For example, group `[connectivity]` related switches in one file, and `[Vivado]` options into a separate file.

The configuration file is specified through the use of the `v++ --config` option as discussed in the [v++ General Options](#). An example of the `--config` option follows:

```
v++ --config ./src/system.cfg
```

Switches are read in the order they are encountered. If the same switch is repeated with conflicting information, the first switch read is used. The order of precedence for switches is as follows, where item one takes highest precedence:

1. Command line switches.
2. Config files (on command line) from left-to-right.
3. Within a config file, precedence is from top-to-bottom.

---

## Using the Message Rule File

The `v++` command executes various Xilinx tools during kernel compilation and linking. These tools generate many messages that provide build status to you. These messages might or might not be relevant to you depending on your focus and design phase. The Message Rule file (`.mrf`) can be used to better manage these messages. It provides commands to promote important messages to the terminal or suppress unimportant ones. This helps you better understand the kernel build result and explore methods to optimize the kernel.

The Message Rule file is a text file consisting of comments and supported commands. Only one command is allowed on each line.

### Comment

Any line with “#” as the first non-white space character is a comment.

## Supported Commands

By default, `v++` recursively scans the entire working directory and promotes all error messages to the `v++` output. The `promote` and `suppress` commands below provide more control on the `v++` output.

- `promote`: This command indicates that matching messages should be promoted to the `v++` output.
- `suppress`: This command indicates that matching messages should be suppressed or filtered from the `v++` output. Note that errors cannot be suppressed.

Enter only one command per line.

## Command Options

The Message Rule file can have multiple `promote` and `suppress` commands. Each command can have one and only one of the options below. The options are case-sensitive.

- `-id [<message_id>]`: All messages matching the specified message ID are promoted or suppressed. The message ID is in format of nnn-mmm. As an example, the following is a warning message from HLS. The message ID in this case is 204-68.

```
WARNING: [V++ 204-68] Unable to enforce a carried dependence constraint
          (II = 1, distance = 1, offset = 1)
          between bus request on port 'gmem'
          (/matrix_multiply_cl_kernel/mmult1.cl:57) and bus request on port 'gmem' -
          severity [severity_level]
```

For example, to suppress messages with message ID 204-68, specify the following:

```
suppress -id 204-68.
```

- `-severity [<severity_level>]`: The following are valid values for the severity level. All messages matching the specified severity level will be promoted or suppressed.
  - `info`
  - `warning`
  - `critical_warning`

For example, to promote messages with severity of 'critical-warning', specify the following:

```
promote -severity critical_warning.
```

## Precedence of Message Rules

The `suppress` rules take precedence over `promote` rules. If the same message ID or severity level is passed to both `promote` and `suppress` commands in the Message Rule file, the matching messages are suppressed and not displayed.

### Example of Message Rule File

The following is an example of a valid Message Rule file:

```
# promote all warning, critical warning
promote -severity warning
promote -severity critical_warning
# suppress the critical warning message with id 19-2342
suppress -id 19-2342
```

# emconfigutil Utility

When running software or hardware emulation in the command line flow, it is necessary to create an emulation configuration file, `emconfig.json`, used by the runtime library during emulation. The emulation configuration file defines the device type and quantity of devices to emulate for the specified platform. A single `emconfig.json` file can be used for both software and hardware emulation.

**Note:** When running on real hardware, the runtime and drivers query the installed hardware to determine the device type and quantity installed, along with the device characteristics.

To use the `emconfigutil` utility to automate the creation of the emulation file, specify the target platform and additional options in the `emconfigutil` command line:

```
emconfigutil --platform <platform_name> [options]
```

At a minimum, you must specify the target platform through the `-f` or `--platform` option to generate the required `emconfig.json` file. The specified platform must be the same as specified during the host and hardware builds.

The `emconfigutil` options are provided in the following table.

*Table 43: emconfigutil Options*

Option	Valid Values	Description
<code>-f</code> or <code>--platform</code>	Target device	Required. Defines the target device from the specified platform. For a list of supported devices, refer to <a href="#">Supported Platforms</a> .
<code>--nd</code>	Any positive integer	Optional. Specifies number of devices. The default is 1.
<code>--od</code>	Valid directory	Optional. Specifies the output directory. When running emulation, the <code>emconfig.json</code> file must be in the same directory as the host executable. The default is to write the output in the current directory.
<code>-s</code> or <code>--save-temps</code>	N/A	Optional. Specifies that intermediate files are not deleted and remain after the command is executed. The default is to remove temporary files.
<code>--xp</code>	Valid Xilinx parameters and properties.	Optional. Specifies additional parameters and properties. For example:  <code>--xp prop:solution.platform_repo_paths=&lt;xsa_path&gt;</code>  This example sets the search path for the target platforms.
<code>-h</code> or <code>--help</code>	N/A	Prints command help.

The `emconfigutil` command generates the `emconfig.json` configuration file in the output directory or the current working directory.



**TIP:** When running emulation, the location of the `emconfig.json` file can be specified by the `$EMCONFIG_PATH` variable, or must be found in the same directory as the host executable.

The following example creates a configuration file targeting two `xilinx_u200_qdma_201910_1` devices.

```
$emconfigutil --xilinx_u200_qdma_201910_1 --nd 2
```

# kernelinfo Utility

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

The following command options are available:

*Table 44: kernelinfo Commands*

Option	Description
<code>-h [ --help ]</code>	Print help message.
<code>-x [ --xo_path ] &lt;arg&gt;</code>	Absolute path to file including file name and <code>.xo</code> extension.
<code>-l [ --log ] &lt;arg&gt;</code>	By default, information is displayed on the screen. Otherwise, you can use the <code>--log</code> option to output the information as a file.
<code>-j [ --json ]</code>	Output the file in JSON format.
<code>[input_file]</code>	XO file. Specify the XO file positionally or use the <code>--xo_path</code> option.
<code>[output_file]</code>	Output from Xilinx OpenCL Compiler. Specify the output file positionally, or use the <code>--log</code> option.

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

```
kernelinfo <filename.o>
```

The output is divided into three sections:

- Kernel Definitions
- Arguments
- Ports

The report generated by the following command is reviewed to help better understand the report content. Note that the report is broken down into specific sections for better understandability.

```
kernelinfo krnl_vadd.xo
```

Where `krnl_vadd.xo` is a compiled kernel.

# Kernel Definition

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

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

# Arguments

Reports kernel function arguments.

In the following example, there are four arguments: `a`, `b`, `c`, and `n_elements`.

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

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

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

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

---

## Ports

Reports the memory and control ports used by the kernel.

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

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

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

# launch\_emulator Utility

For embedded platforms that have an Arm subsystem, the Vitis tool uses QEMU to emulate the PS subsystem. The QEMU processes must be run along with the RTL simulator process to emulate the entire system in hardware emulation. The `launch_emulator.py` is a utility which launches QEMU and manages the synchronization of the PL simulator processes. It launches QEMU and the simulation process with provided arguments. The Vitis IDE also calls this command when starting and stopping the emulator.



**TIP:** For help inside QEMU, press **Ctrl + a h** while in the emulator shell. To terminate the QEMU command, press **Ctrl + a x** while in the emulator shell.

For embedded platforms, the `--package Options` command generates scripts, `launch_hw_emu.sh`, or `launch_sw_emu.sh` to call the `launch_emulator.py` command with the required arguments based on the platform and the target application.

You can pass additional arguments to the `launch_emulator` utility from the command line when using the `launch_hw_emu.sh` or `launch_sw_emu.sh` wrapper scripts. Simply append the option to the command line when running the script. This allows you to customize the `launch_emulator` utility as needed to support your specific platform or application.

The following table shows the list of available options.

**Table 45: Common Options for launch\_emulator**

Option	Accepted Value	Description
<code>-add-env ADD_ENV_CMD</code>	N/A	Specify additional Environment Variables for the emulation shell.
<code>-aie-sim-options</code>	AIE_SIM_OPTIONS file	Points to an AI Engine sim options file that has various AI Engine debug flags that are required for debugging the AI Engine SystemC module. Refer to <a href="#">Reusing AI Engine Simulator Options</a> for more information. The options file should be specified with a relative path with respect to <code>package.hw_emu/sim/beav_waveform/xsim/</code> .  <b>TIP:</b> This is optional and only applies to AI Engine designs.

Table 45: Common Options for launch\_emulator (cont'd)

Option	Accepted Value	Description
-enable-debug	N/A	Debug mode opening two different XTERMs for QEMU and PL.  <b>IMPORTANT!</b> This is very useful for the batch mode users to understand the flow and handshake between the QEMU and PL process.
-graphic-qemu	N/A	Start the Quick Emulator(QEMU) in GUI mode
-help	N/A	Prints help message.
-kernel-dbg	true, false	Enable debug in SW_EMU. This is used only for software emulation.
-pl-kernel-debug	true, false	Enable debug for the PL kernel.
-run-app	<application_script_name>	Ensure that the application script is packaged using <code>--package.sd_file</code> option during package step. Only when it is packaged in <code>sd_card</code> , the application script is available to run after QEMU is up running and mounted.  <b>TIP:</b> When using the <code>-run-app</code> option, all QEMU messages are initially written to a file called <code>qemu_output.log</code> inside the <code>package.hw_emu</code> or <code>package.sw_emu</code> folder, based on your emulation target, and then re-written to the console after some delay. This delay can cause you to think there is a problem with QEMU, but you can examine the contents of the <code>qemu_output.log</code> if needed.
-timeout	<n>	Terminates emulation after <n> seconds. The default value when -run-app is used is 4000 seconds. This means the application terminates after running for 4000 seconds without user intervention.
-user-post-sim-script	Path to Tcl script required to be done post simulation before quit	Creates Tcl for any post operations into a Tcl file and pass the Tcl script to this switch.
-user-pre-sim-script	Path to Tcl script	For first run, <code>launch_emulator.py</code> in GUI mode and add the signals that you want to observe. Copies the commands from the Tcl console and save into a Tcl script. From the next run, pass the Tcl script in batch mode, <code>launch_emulator.py -user-pre-sim-script &lt;path_to_saved_tcl_script&gt;</code> . Only supports the Vivado simulator (xsim).
-verbose	N/A	Enable additional debug messages
-wcfg-file-path	N/A	Specify the wcfg file created by the XSIM to open during GUI simulation
-wdb-File	Path to WDB file	Specify the wdb file to load. Please provide complete absolute path
-x86-sim-options	N/A	Points to the x86 simulation options file which has various AI Engine debug flags that are required for debugging the AI Engine Model. Used only for software emulation.

**Table 45: Common Options for launch\_emulator (cont'd)**

Option	Accepted Value	Description
-xtlm-aximm-log	N/A	<p>This switch generates xTLM AXI4 transaction logs for interface connection between two SystemC models (with information like address/data/size, etc.).</p> <p>While running the emulation log is available at (directory structure can vary based on v++ options and simulator used):</p> <pre>package.hw_emu/sim/beav_waveform/xsim/xsc_report.log</pre>
-xtlm-axis-log	N/A	<p>This switch generates xTLM AXI4-Stream transaction logs for interface connection between two SystemC models.</p> <p>While running emulation log is available at (directory structure can vary based on v++ options and simulator used):</p> <pre>package.hw_emu/sim/beav_waveform/xsim/xsc_report.log</pre>

**Table 46: Advanced Options for launch\_emulator**

Option	Accepted Value	Description
-disable-host-completion-check	N/A	<p>Skip the check for host/test completion. Generally used in applications where python scripts check for the test completion status PASS/FAIL.</p> <p>By default, you search for "TEST PASSED" string when -run-app switch is used.</p>
-enable-tcp-sockets	N/A	Enables TCP Sockets
-kill	<pid>	Kills a specified emulator process.
-kill-pid-file	N/A	Specifies the file to be used to kill the process. This file stores the group PID of the process. This might have been created using -pid-file.
-no-reboot	N/A	Exit QEMU instead of rebooting. Used to exit gracefully from QEMU by executing command <code>reboot -f</code> at the embedded Linux prompt.
-no_build	N/A	Enables a check of the build command without running the build process.
-no_run	N/A	Build but don't run the emulation.
-ospi-image	OSPI Image file	Specify an OSPI image file for booting.
-pl-sim-args	Arguments to simulator	These arguments gets appended to simulator command line. Alternative to <code>pm-sim-args-file</code> .

Table 46: Advanced Options for launch\_emulator (cont'd)

Option	Accepted Value	Description
-pmc-args	Arguments to PMC	<p>The PMC/PMU is emulated by <code>qemu-system-microblazeel</code>. Most common command line switches of the PMC are captured in <code>pmc_args.txt</code>. Instead of writing into a file called <code>pmc_args.txt</code>, you can directly provide all the arguments that need to be appended to the PMC command line. This is an alternative to <code>-pmc-args-file</code>.</p> <p>PMC/PMU arguments for specific devices can be found in <a href="#">Versal PS and PMC Arguments for QEMU</a> and <a href="#">Zynq UltraScale+ MPSoC PS and PMU Arguments for QEMU</a>.</p> <p><b>TIP:</b> This option is not supported for Zynq-7000 devices.</p>
-pmc-args-file	PMC QEMU arguments file name	<p>Any options to be passed to PMU/PMC can be given in this file. The specific format is determined by the base file on your chosen platform.</p> <p>This is auto passed in the <code>v++</code> package generated script.</p> <p>PMC/PMU arguments for specific devices can be found in <a href="#">Versal PS and PMC Arguments for QEMU</a> and <a href="#">Zynq UltraScale+ MPSoC PS and PMU Arguments for QEMU</a>.</p> <p><b>TIP:</b> This option is not supported for Zynq-7000 devices.</p>
-print-qemu-version	N/A	Prints the version of QEMU being used.
-qemu-args	Arguments to QEMU	<p>The PS is emulated by <code>qemu-system-aarch64</code>. Most common command line switches of the PS are captured in <code>qemu_args.txt</code>.</p> <p>Instead of writing into a file called <code>qemu_args.txt</code>, you can directly provide all the arguments that need to be appended to the QEMU command line. This is an alternative to <code>qemu-args-file</code>.</p> <p>PS arguments for specific devices can be found in <a href="#">Versal PS and PMC Arguments for QEMU</a> and following sections for Zynq UltraScale+ MPSoC devices.</p>
-qemu-dtb	<path_to_DTB_file>	<p><code>v++ --package</code> automatically creates a DTB file based on the addressing in the design and passes it to the <code>launch_emulator</code> command. This option lets you specify the DTB file to override the defaults.</p> <p><b>Note:</b> Ensure the DTB is compatible with the <code>noc_memory_config.txt</code> file used.</p>
-qspi-high-image	Specify QSPI high image file	<p>The image file which is passed as a QEMU argument in the form of boot mode. This is auto passed in the <code>v++</code> package generated script.</p> <p>Required only when DUAL QSPI mode is used.</p>

**Table 46: Advanced Options for launch\_emulator (cont'd)**

Option	Accepted Value	Description
-qspi-image	Specify qspi.bin	The image file is passed as a QEMU argument in the form of boot mode. This is auto passed in the <code>V++</code> package generated script. Required only when you opt for QSPI mode.
-qspi-low-image	Specify QSPI low image file	The image file is passed as a QEMU argument in the form of boot mode. This is auto passed in the <code>V++</code> package generated script. Required only when DUAL QSPI mode is used.
-result-string	N/A	Result string searches for the status of test completion. Default = "TEST PASSED."
-use-qemu-version-v4	N/A	Use QEMU version 4.2

**Table 47: Auto-Populated by V++ in Emulation Script**

Option	Accepted Value	Description
-aie-sim-config	N/A	Points to the AI Engine sim config file that provides various AI Engine files that are required for the SystemC Model of AI Engine. This is auto passed by the <code>v++</code> package. Required for AI Engine designs.
-boot-bh	Path to BH file	Specify the boot BH file path
-device-family	7Series   UltraScale   Versal	Required to specify the device family for the platform. This is auto passed by the <code>v++</code> package generated scripts <code>launch_hw_emu.sh</code> or <code>launch_sw_emu.sh</code> based on the target chosen. This needs to be passed manually for direct usage of the <code>launch_emulator</code> command.
-enable-prep-target	N/A	Enable pre-process target.
-forward-port	<target> <host>	Forwards TCP port from target to host.
-gdb-port	Port number	QEMU waits for GDB connection on <port>.
-noc-memory-config <path/to/ noc_memory_config.txt>	N/A	By default, <code>v++ --package</code> creates the NoC memory configuration based on the design configuration, and you can see this file parallel to simulation binaries. You can override this file by replacing the file specified in the simulation binary folder. Use the <code>-user-pre-sim-script</code> option to copy your <code>noc_memory_config.txt</code> file to the simulation binary area and to get the configuration applied.
-pid-file	File name	Write process ID to <file> for later use with <code>-kill</code> . Used by the Vitis software platform to kill once emulation is successful.
-pl-sim-dir	Simulation directory	Start the Programmable Logic Simulator by launching the scripts from this directory. This is auto passed in the <code>v++</code> package generated script. The tool expects a file called <code>simulate.sh</code> in the specified directory and will execute it to launch the PL simulator (for example, XSIM).

Table 47: Auto-Populated by V++ in Emulation Script (cont'd)

Option	Accepted Value	Description
-pl-sim-script	Simulation script location	Advanced users can have one direct script to launch simulation (for example, Vivado users). When this option is given, run the script, other options are of no value.
-platform-name	NAME	The platform name
-pmc-args-file	PMC QEMU arguments file name	<p>Any options to be passed to PMU/PMC can be given in this file. The specific format is determined by the base file on your chosen platform.</p> <p>This is auto passed in the <code>v++</code> package generated script.</p> <p>PMC/PMU arguments for specific devices can be found in <a href="#">Versal PS and PMC Arguments for QEMU</a> and <a href="#">Zynq UltraScale+ MPSoC PS and PMU Arguments for QEMU</a>.</p> <p><b>TIP:</b> This option is not supported for Zynq-7000 devices.</p>
-pmc-dtb	<path_to_DTB_file>	<p><code>v++ --package</code> automatically creates a device-tree binary (DTB) file based on the addressing in the design and passes it to the <code>launch_emulator</code> command. This option lets you specify the DTB file to override the defaults.</p> <p><b>Note:</b> Ensure the DTB is compatible with the <code>noc_memory_config.txt</code> file used.</p> <p>PMC/PMU arguments for specific devices can be found in <a href="#">Versal PS and PMC Arguments for QEMU</a> and <a href="#">Zynq UltraScale+ MPSoC PS and PMU Arguments for QEMU</a>.</p> <p><b>TIP:</b> This option is not supported for Zynq-7000 devices.</p>
-protoinst-File	Path to ProtoInst file	Specify the protoinst file to load. Please provide complete absolute path
-qemu-args-file	PS QEMU Arguments file name	Any options to be passed to QEMU can be given in this file. This is specific format where you fetch the base file from the platform chosen. This is auto passed in the <code>v++</code> package generated script.
-qemu-dtb	<path_to_DTB_file>	<p><code>v++ --package</code> automatically creates a DTB file based on the addressing in the design and passes it to the <code>launch_emulator</code> command. This option lets you specify the DTB file to override the defaults.</p> <p><b>Note:</b> Ensure the DTB is compatible with the <code>noc_memory_config.txt</code> file used.</p>
-sd-card-image	Specify <code>sd_card.img</code>	<p>The image file is passed as a QEMU argument in the form of boot mode. This is auto passed in the <code>V++</code> package generated script.</p> <p>Required only when SD mode is used.</p>

Table 47: Auto-Populated by V++ in Emulation Script (cont'd)

Option	Accepted Value	Description
-t   -target	sw_emu or hw_emu	Specify to run sw_emu or hw_emu. Based on the target chosen in the v++, respective script is generated by the v++ package. For sw_emu target, launch_sw_emu.sh is generated and for hw_emu target, launch_hw_emu.sh is generated.
-xtlm-log-state	WAVEFORM   LOG   BOTH	Option to specify what the XTLM Log should contain. It can contain the waveform, the text log, or both. This option is used only for hardware emulation.

## Versal PS and PMC Arguments for QEMU

In the Versal® device, the PS(a72) is emulated by `qemu-system-aarch64` and PMC is emulated by `qemu-system-microblazeel`. Most common command line switches of PS are captured in `qemu_args.txt` and PMC command line switches are captured in `pmc_args.txt`.

Table 48: Versal Options for `qemu_args.txt`

Arg Name	Value	Description	Source	How to Extract the Info
-boot	mode=<boot-number> Ex. for sd1 -boot mode=5	Specify boot mode on your platform: <ul style="list-style-type: none"><li>• qspi24 = 1</li><li>• qspi32 = 2</li><li>• sd0 = 3</li><li>• sd1 = 5</li><li>• emmc0 = 6</li><li>• ospi = 8</li></ul>	v++ -p	DRC needed between CIPS enabled boot modes and v++ -p selection
-display	none	By default QEMU creates display for user I/O. This option disables the display	Static	Specify none to disable the display
-hw-dtb	<ps-dtb-file>	Device tree file which describes the PS (a72). The Vitis compiler --package command generates the dtb file and appends it to <code>qemu_args.txt</code> .	v++ -p	
-M	arm-generic-fdt	This specifies the QEMU machine to create. <code>arm-generic-fdt</code> machine option tells QEMU to parse dtb for machine generation, passes by <code>-hw-dtb user.dtb</code> .	Device specific	Hard coded for Versal

Table 48: Versal Options for qemu\_args.txt (cont'd)

Arg Name	Value	Description	Source	How to Extract the Info
-serial	-serial null -serial null -serial mon:stdio	<p>By default, QEMU connects invoking terminal to UART0 for user I/O operations. You can override this behavior by specifying this option. Versal platforms have four UARTs specified using positional arguments: the first two are for debug and the last two are UART0 and UART1.</p> <p>To connect UART0 to the terminal, specify <code>-serial null -serial null -serial mon:stdio</code> which ignores debug related UARTS and connects to UART0 to terminal.</p> <p>Similarly to connect only UART1 to terminal specify <code>-serial null -serial null -serial null -serial mon:stdio</code></p>	Based on UARTs enabled on CIPS configuration.	<p>The Versal device has four UARTs. The first two are debug related UARTs.</p> <p>When enabling UART0:</p> <pre>CONFIG.PS_UART0_P_ERIPHERAL_ENABLE = 1 CONFIG.PS_UART1_P_ERIPHERAL_ENABLE = 0 or 1</pre> <p>Then specify <code>-serial null -serial null -serial mon:stdio</code></p> <p>When enabling only UART1:</p> <pre>CONFIG.PS_UART0_P_ERIPHERAL_ENABLE = 0 CONFIG.PS_UART1_P_ERIPHERAL_ENABLE = 1</pre> <p>Then specify: <code>-serial null -serial null -serial null -serial mon:stdio</code></p>
-sync-quantum	Time in milliseconds	Specifies how frequently QEMU will sync with the RTL simulator. Modifying this can have an impact on the speed of simulation.	Static	Hard coded for Versal devices (user need to change)

Versal CIPS has two Ethernet interfaces. Most of Xilinx Versal CIPS board has eth0 enabled. If no `-net` or `-netdev` is specified then QEMU by default enables eth0 and maps to user mode backend.

Table 49: Versal Options for pmc\_args.txt

Switch Name	Value	Description	Source of the Config	How to Extract the Info
-M	microblaze-fdt	Specifies the QEMU machine to create. QEMU creates MicroBlaze with nodes from dtb.	Static	Hard coded for Versal Devices
-display	none	By default, QEMU creates display for user I/O. This option instructs QEMU that there is no need for display.	Static	Hard coded for Versal Devices

Table 49: Versal Options for pmc\_args.txt (cont'd)

Switch Name	Value	Description	Source of the Config	How to Extract the Info
-device	loader,file=<BOOT_bh.bin>,addr=0xF201e000,force-raw	Specifies Boot header file with load address as 0xF201E000 (<BOOT_bh.bin> is loaded at address 0xF201e000). This is fixed argument in pmc_args.txt which is processed by v++ -p for final argument which has absolute path of BOOT_bh.bin file. BOOT_bh.bin is generated by v++ -p from final PDI. BOOT_bh.bin is directly loaded onto QEMU because there is no BOOT ROM access for QEMU to load boot header from PDI.	v++ -pack	v++ pack extracts BOOT_bh.bin and generates this switch
-device	loader,file=<pmc_cdo.bin>,addr=0xF2000000,force-raw	Specifies pmc_cdo.bin with load address as 0xF2000000. This is fixed argument in pmc_args.txt. This is processed by v++ -p for final argument which has absolute path of pmc_cdo.bin file.	v++ -pack	v++ pack extracts pmc_cdo.bin and generates this switch
-device	loader,file=<plm.bin>,addr=0xF0200000,force-raw	Specifies plm binary firmware with load address as 0xF0200000. This is a fixed argument in pmc_args.txt which is processed by v++ -p for final argument. This has an absolute path of plm.bin file. This plm is executed by PPU1 when it is out of reset.	v++ -pack	v++ pack extracts plm.bin and generates this switch
-device	loader,addr=0xf0000000,data=a=0xba020004,data-len=4 -device loader,addr=0xf0000004,data=a=0xb800fffc,data-len=4	Specifies PPU0 process to be in boot loop. As there is no BOOTROM access for QEMU, PPU0 is put in bootloop which generally loads BOOT header from PDI to memory.	Static	Hard coded for Versal Devices
-device	loader,addr=0xF1110624,data=0x0,data-len=4 -device loader,addr=0xF1110620,data=0x1,data-len=4	Makes PPU1 out of reset and puts in executing mode.	Static	Hard coded for Versal Devices
-hw-dtb	<ps-dtb-file>	dtb file which describes aout PS(a72). v++ pack generates this dtb file and appends to qemu-args.txt.	v++ pack	v++ pack generates dtb files based on DDR config on device

# Zynq UltraScale+ MPSoC PS and PMU Arguments for QEMU

The Zynq UltraScale+ MPSoC PS(a53) is emulated by `qemu-system-aarch64` and PMU is emulated by `qemu-system-microblazeel`. Most common command line switches of PS are captured in `qemu_args.txt` and PMC command line switches are captured in `pmu_args.txt`.

*Table 50: Zynq UltraScale+ MPSoC Options for qemu\_args.txt*

Switch Name	Value	Description	Source of the Config	How to Extract the Info
-M	arm-generic-fdt	This specifies the QEMU machine to create. <code>arm-generic-fdt</code> machine option tells QEMU to parse <code>dtb</code> for machine generation, passes by <code>-hw-dtb user.dtb</code> .	Static	Hard-coded for Zynq UltraScale+ MPSoC devices
-serial	mon:stdio	-serial is a positional argument. Redirect the serial port to specified char dev (i.e., stdio, tcp port, file, etc.).	Based on UART configuration on Zynq UltraScale+ MPSoC	<p>Zynq UltraScale+ MPSoC has two UARTs. When enabling UART0:</p> <pre>CONFIG.PSU__UART0__PERIPHERAL__ENA BLE = 1 CONFIG.PSU__UART1__PERIPHERAL__ENA BLE = 0 or 1</pre> <p>Then specify: -serial mon:stdio</p> <p>When enabling only UART1:</p> <pre>CONFIG.PSU__UART0__PERIPHERAL__ENA BLE = 0 CONFIG.PSU__UART1__PERIPHERAL__ENA BLE = 1</pre> <p>Then specify: -serial null -serial mon:stdio</p>
-global	xlnx,zynqmp-boot.cpu-num=0	Make the specified CPU come out of reset.	Static	Hard coded for Zynq UltraScale+ MPSoC devices

Table 50: Zynq UltraScale+ MPSoC Options for qemu\_args.txt (cont'd)

Switch Name	Value	Description	Source of the Config	How to Extract the Info
-net	-net nic -net nic -net nic -net nic -net user	<p>-net is positional argument. Initialize network interfaces gem3. Connect the specified network adapter to user mode network.</p> <p><b>TIP:</b> <i>-net none will disable all Ethernet interfaces.</i></p>	Static	<p>Based on Ethernet configurations: If gem0(eth0) is enabled:</p> <pre>CONFIG.PSU__ENET0__PERIPHERAL_ENA_BLE = 1</pre> <p>Then specify -net nic -net user If gem1 is enabled:</p> <pre>CONFIG.PSU__ENET1__PERIPHERAL_ENA_BLE = 1</pre> <p>Then specify -net nic -net nic -net user If gem2 is enabled:</p> <pre>CONFIG.PSU__ENET2__PERIPHERAL_ENA_BLE = 1</pre> <p>Then specify -net nic -net nic -net nic -net user If gem 3 is enabled:</p> <pre>CONFIG.PSU__ENET3__PERIPHERAL_ENA_BLE = 1</pre> <p>Then specify -net nic -net nic -net nic -net nic -net user</p> <p><b>TIP:</b> <i>If no -net (and/or -netdev) is mentioned then by default QEMU will enable the first Ethernet (gem0) and map it to user mode backend.</i></p>
-m	4G	Enabling 4 GB DDR on Zynq UltraScale+ MPSoC.	Static	Emulating full DDR on Zynq UltraScale+ MPSoC
-device	loader,file=<bl31.elf>,cpu-num=0	Load bl31.elf file on A53 core 0.	Static	v++ --package should replace bl31.elf with absolute path of bl31.elf

Table 50: Zynq UltraScale+ MPSoC Options for qemu\_args.txt (cont'd)

Switch Name	Value	Description	Source of the Config	How to Extract the Info
-device	loader,file=<u-boot.elf>	Loading u-boot.elf.	Static	v++ --package should replace b131.elf with absolute path of u- boot.elf
-hw-dtb	<ps-dtb-file>	dtb file which describes PS which is emulated by QEMU can be specified using -hw- dtb.	Static	Hard coded for Zynq UltraScale+ MPSoC devices: <ps-dtb-file>=/ proj/xbuilds/ HEAD_daily_latest/ installs/lin64/ Vitis/HEAD//data/ emulation/dtbs/ zynqmp/zynqmp- arm-cosim.dtb

Table 51: Zynq UltraScale+ MPSoC Options for pmu\_args.txt

Switch Name	Value	Description	Source of the Config	How to Extract the Info
-M	microblaze-fdt	This specifies the QEMU machine to create. microblaze-fdt tells QEMU to parse dtb for machine generation, passes by -hw- dtb user.dtb.	Static	Hard coded for Zynq UltraScale+ MPSoC devices
-device	loader,file=<pmufw.elf>	Load pmufw.elf file on PMU RAM.	Static	Hard coded for Zynq UltraScale+ MPSoC devices
-machine-path	<path-to-xsim-dir>	Point -machine-path to folder to create shared RAM and remote-port sockets.	Static	launch_emulator command will set this machine path
-display	none	By default, QEMU creates display for user I/O. This option disables the display.	Static	Hard coded for Zynq UltraScale+ MPSoC devices
-hw-dtb	<pmu-dtb-file>	dtb file which describes PMU which is emulated by QEMU can be specified using -hw-dtb.	Static	<pmu-dtb-file>=/proj/ xbuilds/ HEAD_daily_latest/ installs/lin64/Vitis/ HEAD//data/ emulation/dtbs/ zynqmp/zynqmp- pmu.dtb



**TIP:** Although the file is called pmu\_args.txt here, the file is specified for launch\_emulator using the -pmc-args-file command.

## Zynq-7000 PS Arguments for QEMU

Zynq-7000 PS(a9) is emulated by `qemu-system-aarch64` QEMU binary. Most common command line switches of PS are captured in `qemu_args.txt`.

*Table 52: Zynq-7000 Options for qemu\_args.txt*

Switch Name	Value	Description	Source of the Config	How to Extract the Info
-M	arm-generic-fdt-7series	Indicates the QEMU machine to create. arm-generic-fdt-7series tells QEMU to parse <code>dtb</code> for machine generation, passes by <code>-hw-dtb user.dtb</code> .	Static	Hard coded for Zynq-7000 devices
-serial	-serial /dev/null -serial mon:stdio	Redirect the serial port to specified char dev (i.e., stdio, tcp port, file, etc.)	Based on the UART configuration of the Zynq IP.	Zynq-7000 has two UARTs. When enabling UART0:  CONFIG.PCW_UART0_PERIPHERAL_ENABLE = 1 CONFIG.PCW_UART1_PERIPHERAL_ENABLE = 0 or 1  Then specify: -serial mon:stdio When enabling only UART1:  CONFIG.PCW_UART1_PERIPHERAL_ENABLE = 1  Then specify: -serial null -serial mon:stdio
-device	loader,addr=0xf8000008,data-len=4 - device loader,addr=0xf8000140,data-len=4 - device loader,addr=0xf800012c,data-len=4 - device loader,addr=0xf8000108,data-len=4 - device loader,addr=0xF800025C,data=0x00000005,data-len=4 - device loader,addr=0xF8000240,data=0x00000000,data-len=4	Register writes to SLCR block, to set PLL and CLK_CTRL regs (required for Linux).	Static	Hard coded for Zynq-7000 devices
-boot	mode=5	Boot mode 5 is for SD boot.	v++ -p	

**Table 52: Zynq-7000 Options for qemu\_args.txt (cont'd)**

Switch Name	Value	Description	Source of the Config	How to Extract the Info
-kernel	<u-boot.elf>	Guest software to load during boot up.	Static	<u-boot.elf> is replaced with the absolute path of <code>u-boot.elf</code> from the target platform
-machine	linux=on	Make QEMU itself a loader of the Linux image.	Static	Hard coded for Zynq-7000 devices

# manage\_ipcache Utility

To provide better performance during synthesis of kernels in your application designs, the Vitis compiler uses an IP cache to store and reuse synthesis results. This lets the build process for the `.xclbin` file avoid having to repeat synthesis for kernels and CUs that have not changed. The IP cache stores the synthesis results and applies them for unchanged kernels in the design.

By default, the IP cache is stored inside the Vitis IDE workspace for a project, or at the level of your builds when running `v++` from the command line. You can customize the location for the IP cache using `--remote_ip_cache` to specify a new location, or disable the use of the IP cache using `--no_ip_cache`. See [v++ General Options](#) for information on these options.

The `manage_ipcache` utility is a standalone utility to help you manage the contents of your IP cache repository. It lets you report statistics on the IP cache repository and remove entries based on a variety of criteria.

*Table 53: manage\_ipcache Options*

Option	Description
<code>-c   --cache</code>	Required. Specifies the IP Cache directory to work on.
<code>-d   --disk_space &lt;size&gt;</code>	Delete all but the most recently used entries that fit in the disk space specified in MB.
<code>-h   --help</code>	Prints help for the <code>manage_ipcache</code> command.
<code>-k   --keep_top &lt;N&gt;</code>	Delete all but the top N most recently used entries (N is an integer).
<code>-o   --outfile &lt;file&gt;</code>	Report stats for the IP cache to the specified file.
<code>-p   --purge</code>	Delete ALL cache entries.
<code>-r   --report</code>	Report stats for the IP cache to stdout.
<code>-u   --unused</code>	Delete cache entries that have never been used (no cache hits).

The following example reports on the entries of the specified IP cache:

```
manage_ipcache --cache ./ip_cache --report
```

The `manage_ipcache` command returns 0 if successful, or returns -1 if an error occurs.

# package\_xo Command

## Syntax

```
package_xo -kernel_name <arg> [-force] [-kernel_xml <arg>]
[-output_kernel_xml <arg>] [-design_xml <arg>]
[-ip_directory <arg>] [-parent_ip_directory <arg>]
[-kernel_files <args>] [-kernel_xml_args <args>]
[-kernel_xml_pipes <args>] [-kernel_xml_connections <args>]
[-ctrl_protocol <arg>] -xo_path <arg> [-quiet] [-verbose]
```

## Description

The `package_xo` command is a Tcl command within the Vivado Design Suite. Kernels written in RTL are compiled in the Vivado tool using the `package_xo` command line utility which generates a Xilinx object (XO) file which can subsequently used by the `v++` command, during the linking stage.

**Table 54: Arguments**

Argument	Description
<code>-kernel_name &lt;arg&gt;</code>	(Required) Specify the name of the RTL kernel.
<code>-force</code>	(Optional) Overwrite an existing XO file if one exists.
<code>-kernel_xml &lt;arg&gt;</code>	(Optional) Specify the path to an existing kernel XML file. The Vivado tool will create a <code>kernel.xml</code> file for the XO file if one is not specified.
<code>-output_kernel_xml</code>	(Optional) Specify the path to write the kernel XML file. The Vivado tool will create a <code>kernel.xml</code> file to include in the XO file, and also write it to the specified output file.  <b>TIP:</b> You can use this option to generate a <code>kernel.xml</code> file which you can edit and use as an input in the <code>package_xo</code> command.
<code>-design_xml &lt;arg&gt;</code>	(Optional) Specify the path to an existing design XML file
<code>-ip_directory &lt;arg&gt;</code>	(Optional) Specify the path to the packaged IP directory.
<code>-parent_ip_directory</code>	(Optional) If the kernel IP directory specified contains multiple IPs, specify a directory path to the parent IP where its <code>component.xml</code> is located directly below.
<code>-kernel_files</code>	(Optional) Kernel file name(s). Can be used to add a C-model to your kernel XO to enable software emulation for your kernel.

Table 54: Arguments (cont'd)

Argument	Description
-kernel_xml_args <args>	(Optional) Generate the kernel.xml with the specified function arguments. Each argument value should use the following format:  <code>{name:addressQualifier:id:port:size:offset:type:memSize}</code>  <b>Note:</b> memSize is optional.
-kernel_xml_pipes <args>	(Optional) Generate the kernel.xml with the specified pipe(s). Each pipe value use the following format:  <code>{name:width:depth}</code>
-kernel_xml_connections <args>	(Optional) Generate the kernel.xml file with the specified connections. Each connection value should use the following format:  <code>{srcInst:srcPort:dstInst:dstPort}</code>
-ctrl_protocol	Kernel control protocol as described in <a href="#">PL Kernel Properties</a> . Valid values: ap_ctrl_hs, ap_ctrl_chain, ap_ctrl_none, user_managed.  <b>TIP:</b> The default ap_ctrl_hs is written to the kernel.xml file when -ctrl_protocol is not specified.
-xo_path <arg>	(Required) Specify the path and file name of the compiled object (XO) file.
-quiet	(Optional) Execute the command quietly, returning no messages from the command. The command also returns TCL_OK regardless of any errors encountered during execution.  <b>Note:</b> Any errors encountered on the command-line, while launching the command, will be returned. Only errors occurring inside the command will be trapped.
-verbose	(Optional) Temporarily override any message limits and return all messages from this command.  <b>Note:</b> Message limits can be defined with the set_msg_config command.

## Examples

The following example creates the specified XO file containing an RTL kernel of the specified name using the ap\_ctrl\_chain control protocol, and creates the kernel.xml file because one has not been specified:

```
package_xo -xo_path Vadd_A_B.xo -kernel_name Vadd_A_B -ctrl_protocol
ap_ctrl_chain -ip_directory ./ip
```

The following example creates the XO file using the specified `kernel.xml` file:

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



**TIP:** The control protocol will be defined in the specified `kernel.xml` file.

## RTL Kernel XML File



**TIP:** The `package_xo` command will create a `kernel.xml` file from the `component.xml` of a packaged IP, so you do not need to manually provide one, or generate one using the RTL Kernel wizard.

An XML kernel description file, called `kernel.xml`, must be created for each RTL kernel, so that it can be used in the Vitis application acceleration development flow. The `kernel.xml` file specifies kernel attributes like the register map and ports needed by the runtime and Vitis tool flows. The following code shows is an example of a `kernel.xml` file.

```
<?xml version="1.0" encoding="UTF-8"?>
<root versionMajor="1" versionMinor="6">
    <kernel name="vitis_kernel_wizard_0" language="ip_c"
        vlnv="mycompany.com:kernel:vitis_kernel_wizard_0:1.0"
        attributes="" preferredWorkGroupSizeMultiple="0" workGroupSize="1"
        interrupt="true">
        <ports>
            <port name="s_axi_control" mode="slave" range="0x1000" dataWidth="32"
                portType="addressable" base="0x0"/>
            <port name="m00_axi" mode="master" range="0xFFFFFFFFFFFFFF"
                dataWidth="512" portType="addressable"
                base="0x0"/>
        </ports>
        <args>
            <arg name="axi00_ptr0" addressQualifier="1" id="0" port="m00_axi"
                size="0x8" offset="0x010" type="int*"
                hostOffset="0x0" hostSize="0x8"/>
        </args>
    </kernel>
</root>
```

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

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

**Table 55: Kernel XML File Content**

Tag	Attribute	Description
<root>	versionMajor	For the current release of Vitis software platform, set to 1.
	versionMinor	For the current release of Vitis software platform, set to 6.

**Table 55: Kernel XML File Content (cont'd)**

Tag	Attribute	Description
<kernel>	name	Kernel name
	language	Always set to <code>ip_c</code> for RTL kernels.
	vlnv	Must match the vendor, library, name, and version attributes in the <code>component.xml</code> of an IP. For example, if <code>component.xml</code> has the following tags: <code>&lt;spirit:vendor&gt;xilinx.com&lt;/spirit:vendor&gt;</code> <code>&lt;spirit:library&gt;hls&lt;/spirit:library&gt;</code> <code>&lt;spirit:name&gt;test_sincos&lt;/spirit:name&gt;</code> <code>&lt;spirit:version&gt;1.0&lt;/spirit:version&gt;</code> The <code>vlnv</code> attribute in kernel XML must be set to <code>xilinx.com:hls:test_sincos:1.0</code>
	attributes	Reserved. Set it to empty string: ""
	preferredWorkGroupSizeMultiple	Reserved. Set it to 0.
	workGroupSize	Reserved. Set it to 1.
	interrupt	Set to "true" ( <code>interrupt="true"</code> ) if the RTL kernel has an interrupt, otherwise omit.
	hwControlProtocol	Specifies the control protocol for the RTL kernel. <ul style="list-style-type: none"> <li><code>ap_ctrl_hs</code>: Default control protocol for RTL kernels.</li> <li><code>ap_ctrl_chain</code>: Control protocol for chained kernels that support dataflow. Adds <code>ap_continue</code> to the control registers to enable <code>ap_done/ap_continue</code> completion acknowledgment.</li> <li><code>ap_ctrl_none</code>: Control protocol (none) applied for data driven kernels.</li> <li><code>user_managed</code>: Specifies a kernel that meets the interface requirements for Vitis compiler to link the kernel with other kernels and a target platform, but does not adhere to the requirements for execution management by XRT. Refer to <a href="#">Creating User-Managed RTL Kernels</a> for more information.</li> </ul>

Table 55: Kernel XML File Content (cont'd)

Tag	Attribute	Description
<port>	name	<p>Specifies the port name.</p> <p><b>IMPORTANT!</b> The AXI4-Lite interface must be named <code>S_AXI_CONTROL</code>.</p>
	mode	<p>At least one AXI4 master port and one AXI4-Lite slave control port are required.</p> <p>AXI4-Stream ports can be specified to stream data between kernels.</p> <ul style="list-style-type: none"> <li>For AXI4 master port, set to "master."</li> <li>For AXI4 slave port, set to "slave."</li> <li>For AXI4-Stream master port, set to "write_only."</li> <li>For AXI4-Stream slave port, set it "read_only."</li> </ul>
	range	The range of the address space for the port.
	dataWidth	The width of the data that goes through the port, default is 32-bits.
	portType	Indicate whether or not the port is addressable or streaming. <ul style="list-style-type: none"> <li>For AXI4 master and slave ports, set it to "addressable."</li> <li>For AXI4-Stream ports, set it to "stream."</li> </ul>
	base	For AXI4 master and slave ports, set to <code>0x0</code> . This tag is not applicable to AXI4-Stream ports.
<arg>	name	Specifies the kernel software argument name.
	addressQualifier	Valid values: 0: Scalar kernel input argument 1: global memory 2: local memory 3: constant memory 4: pipe
	id	Only applicable for AXI4 master and slave ports. The ID needs to be sequential. It is used to determine the order of kernel arguments. Not applicable for AXI4-Stream ports.
	port	Specifies the <port> name to which the <code>arg</code> is connected.
	size	Size of the argument in bytes. The default is 4 bytes.
	offset	Indicates the register memory address.
	type	The C data type of the argument. For example, <code>uint*</code> , <code>int*</code> , or <code>float*</code> .
	hostOffset	Reserved. Set to <code>0x0</code> .
	hostSize	Size of the argument. The default is 4 bytes.
	memSize	For AXI4-Stream ports, <code>memSize</code> sets the depth of the created FIFO.  <b>TIP:</b> Not applicable to AXI4 ports.

**Table 55: Kernel XML File Content (cont'd)**

Tag	Attribute	Description
The following tags specify additional tags for AXI4-Stream ports. They do not apply to AXI4 ports.		
<connection>	The connection tag describes the actual connection in hardware, either from the kernel to the FIFO inserted for the PIPE, or from the FIFO to the kernel.	
	srcInst	Specifies the source instance of the connection.
	srcPort	Specifies the port on the source instance for the connection.
	dstInst	Specifies the destination instance of the connection.
	dstPort	Specifies the port on the destination instance of the connection.

# platforminfo Utility

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

The following command options are available to use with `platforminfo`:

**Table 56: platforminfo Commands**

Option	Description
<code>-f [ --force ]</code>	Overwrite an existing output file.
<code>-h [ --help ]</code>	Print help message and exit.
<code>-k [ --keys ]</code>	Get keys for a given platform. Returns a list of JSON paths.
<code>-l [ --list ]</code>	List platforms. Searches the user repo paths \$PLATFORM_REPO_PATHS and then the install locations to find <code>.xpfm</code> files.
<code>-e [ --extended ]</code>	List platforms with extended information. Use with '--list'.
<code>-d [ --hw ] &lt;arg&gt;</code>	Specify the platform definition (*.dsa) on which to operate. The value must be a full path, including file name and <code>.dsa</code> extension.
<code>-s [ --sw ] &lt;arg&gt;</code>	Specify the software platform definition (*.spfm) on which to operate. The value must be a full path, including file name and <code>.spfm</code> extension.
<code>-p [ --platform ] &lt;arg&gt;</code>	Xilinx® platform definition (*.xpfm) on which to operate. The value for <code>--platform</code> can be a full path including file name and <code>.xpfm</code> extension, as shown in example 1 below. If supplying a file name and <code>.xpfm</code> extension without a path, this utility will search only the current working directory. You can also specify just the base name for the platform. When the value is a base name, this utility will search the \$PLATFORM_REPO_PATHS, and the install locations, to find a corresponding <code>.xpfm</code> file, as shown in example 2 below.  <div style="background-color: #f0f0f0; padding: 5px;"> Example 1: <code>--platform /opt/xilinx/platforms/xilinx_u50_gen3x16_xdma_201920_3.xpfm</code>  Example 2: <code>--platform xilinx_u200_gen3x16_xdma_2_202110_1</code> </div>
<code>-o [ --output ] &lt;arg&gt;</code>	Specify an output file to write the results to. By default the output is returned to the terminal (stdout).

Table 56: platforminfo Commands (cont'd)

Option	Description
-j [ --json ] <arg>	<p>Specify JSON format for the generated output. When used with no value, the <code>platforminfo</code> utility prints the entire platform in JSON format. This option also accepts an argument that specifies a JSON path, as returned by the <code>-k</code> option. The JSON path, when valid, is used to fetch a JSON subtree, list, or value.</p> <p>Example 1:  <code>platforminfo --json="hardwarePlatform" --platform &lt;platform base name&gt;</code></p> <p>Example 2: Specify the index when referring to an item in a list.  <code>platforminfo --json="hardwarePlatform.devices[0].name" --platform &lt;platform base name&gt;</code></p> <p>Example 3: When using the short option form (-j), the value must follow immediately.  <code>platforminfo -j"hardwarePlatform.systemClocks[]" -p &lt;platform base name&gt;</code></p>
-v [ --verbose ]	Specify more detailed information output. The default behavior is to produce a human-readable report containing the most important characteristics of the specified platform.

**Note:** Except when using the `--help` or `--list` options, a platform must be specified. You can specify the platform using the `--platform` option, or using either `--hw`, `--sw`. You can also simply insert the platform name or full path into the command line positionally.

To understand the generated report, condensed output logs, based on the following command are reviewed. Note that the report is broken down into specific sections for better understandability.

```
platforminfo -p $PLATFORM_REPO_PATHS/
xilinx_u200_gen3x16_xdma_2_202110_1.xpfm
```



**TIP:** See [Platforminfo for xilinx\\_zcu104\\_base\\_202010\\_1](#) for an example of embedded processor platforms.

## Basic Platform Information

Platform information and high-level description are reported.

```
Platform:          xdma
File:              /proj/xbuilds/2020.2_daily_latest/internal_platforms/
xilinx_u200_xdma_201830_3/xilinx_u200_xdma_201830_3.xpfm
Description:
```

## Hardware Platform Information

General information on the hardware platform is reported. For the Software Emulation and Hardware Emulation field, a "1" indicates this platform is suitable for these configurations.

Vendor:	xilinx
Board:	U200 (xdma)
Name:	xdma
Version:	201830.3
Generated Version:	2018.3
Hardware:	1
Software Emulation:	1
Hardware Emulation:	1
Hardware Emulation Platform:	0
FPGA Family:	virtexuplus
FPGA Device:	xcu200
Board Vendor:	xilinx.com
Board Name:	xilinx.com:au200:1.0
Board Part:	xcu200-fsgd2104-2-e

## Interface Information

The following shows the reported PCIe interface information.

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

## Clock Information

Reports the maximum kernel clock frequencies available. The Clock Index is the reference used in the `--kernel_frequency v++` directive when overriding the default value.

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

## Valid SLRs

Reports the valid SLRs in the platform.

```
SLR0, SLR1, SLR2
```

## Resource Availability

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

```
=====
Total
=====
LUTs: 1047139
FFs: 2186064
BRAMs: 1896
DSPs: 6833

=====
Per SLR
=====
SLR0:
LUTs: 354690
FFs: 723308
BRAMs: 638
DSPs: 2265
SLR1:
LUTs: 159739
FFs: 331654
BRAMs: 326
DSPs: 1317
SLR2:
LUTs: 354839
FFs: 723294
BRAMs: 638
DSPs: 2265
```

# Memory Information

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

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

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

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

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

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

This memory index is used when specifying memory resources in the **v++** command as shown below:

```
v++ ... --connectivity.sp vadd.m_axi_gmem:DDR[3]
```

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

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

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

---

## Feature ROM Information

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

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

---

## Software Platform Information

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

```
Number of Runtimes:        1
Default System Configuration: config0_0
System Configurations:
  System Config Name:          config0_0
  System Config Description:    config0_0 Linux OS on x86_0
  System Config Default Processor Group: x86_0
  System Config Default Boot Image:  0
  System Config Is QEMU Supported:  0
```

```
System Config Processor Groups:  
    Processor Group Name:      x86_0  
    Processor Group CPU Type:  x86  
    Processor Group OS Name:   Linux OS  
System Config Boot Images:  
Supported Runtimes:  
    Runtime: OpenCL
```

## Platforminfo for xilinx\_zcu104\_base\_202010\_1

Use the following command to return the platforminfo for the xilinx\_zcu104\_base\_202010\_1 platform:

```
platforminfo -p xilinx_zcu104_base_202010_1
```

The results returned are as follows:

```
=====  
Basic Platform Information  
=====  
Platform:          xilinx_zcu104_base_202010_1  
File:              /platforms/xilinx_zcu104_base_202010_1/  
xilinx_zcu104_base_202010_1.xpfm  
Description:  
A basic static platform targeting the ZCU104 evaluation board, which  
includes 2GB DDR4, GEM, USB, SDIO interface and UART of the Processing  
System. It reserves most of the PL resources for user to add acceleration  
kernels  
  
=====  
Hardware Platform (Shell) Information  
=====  
Vendor:            xilinx  
Board:             zcu104_base  
Name:              zcu104_base  
Version:           1.0  
Generated Version: 2020.1  
Software Emulation: 1  
Hardware Emulation: 0  
FPGA Family:       zynquplus  
FPGA Device:        xczu7ev  
Board Vendor:       xilinx.com  
Board Name:         xilinx.com:zcu104:1.1  
Board Part:         xczu7ev-ffvc1156-2-e  
Maximum Number of Compute Units: 60  
  
=====  
Clock Information  
=====  
Default Clock Index: 0  
Clock Index:          0  
    Frequency:        150.000000  
Clock Index:          1
```

```

    Frequency:          300.000000
Clock Index:          2
    Frequency:          75.000000
Clock Index:          3
    Frequency:          100.000000
Clock Index:          4
    Frequency:          200.000000
Clock Index:          5
    Frequency:          400.000000
Clock Index:          6
    Frequency:          600.000000

=====
Memory Information
=====
    Bus SP Tag: HPO
    Bus SP Tag: HP1
    Bus SP Tag: HP2
    Bus SP Tag: HP3
    Bus SP Tag: HPC0
    Bus SP Tag: HPC1

=====
Feature ROM Information
=====

=====
Software Platform Information
=====
Number of Runtimes:      1
Default System Configuration: xilinx_zcu104_base_202010_1
System Configurations:
    System Config Name:           xilinx_zcu104_base_202010_1
    System Config Description:    xilinx_zcu104_base_202010_1
    System Config Default Processor Group: xrt
    System Config Default Boot Image:   standard
    System Config Is QEMU Supported:  1
    System Config Processor Groups:
        Processor Group Name:       xrt
        Processor Group CPU Type:  cortex-a53
        Processor Group OS Name:   linux
    System Config Boot Images:
        Boot Image Name:           standard
        Boot Image Type:          -
        Boot Image BIF:            xilinx_zcu104_base_202010_1/boot/linux.bif
        Boot Image Data:           xilinx_zcu104_base_202010_1/xrt/image
        Boot Image Boot Mode:      sd
        Boot Image RootFileSystem: -
        Boot Image Mount Path:    /mnt
        Boot Image Read Me:       xilinx_zcu104_base_202010_1/boot/
generic.readme
        Boot Image QEMU Args:     xilinx_zcu104_base_202010_1/qemu/
pmu_args.txt:xilinx_zcu104_base_202010_1/qemu/qemu_args.txt
        Boot Image QEMU Boot:    -
        Boot Image QEMU Dev Tree: -
Supported Runtimes:
    Runtime: OpenCL

```

# xbutil Utility

The Xilinx® Board Utility (`xbutil`) is a standalone command line utility that is included with the Xilinx Runtime (XRT) installation package. Details of the `xbutil` command can be found at <https://xilinx.github.io/XRT/master/html/xbutil.html>.

`xbutil` includes multiple commands to validate and identify the installed accelerator card(s) along with additional card details including on card memory, host interface, target platform name, and system information. This information can be used for both card administration and application debugging.

Accelerator cards are partitioned into a user function and a management function to provide different levels of card access. The user function lets you load and run applications, while the management function is for system administrators to manage the card. The `xbutil` utility interacts with the user function. The `xbmgmt` utility, which requires root privilege, is for interacting with the management function. The reason for splitting the function access between the two utilities is to provide some security for the management features of the tool.

You can use the `help` command to list the available `xbutil` commands and options:

```
xbutil --help
```

Set up the `xbutil` command as part of the XRT installation using the following scripts:

- For csh shell:

```
$ source /opt/xilinx/xrt/setup.csh
```

- For bash shell:

```
$ source /opt/xilinx/xrt/setup.sh
```

# xbmgmt Utility

Xilinx® Board Management (`xbmgmt`) utility is a standalone command line tool that is included with the Xilinx Runtime (XRT) installation package. Details of the `xbmgmt` command can be found at <https://xilinx.github.io/XRT/master/html/xbmgmt.html>.

Accelerator cards are partitioned into a user function and a management function to provide different levels of card access. The user function lets you load and run applications, while the management function is for system administrators to manage the card. The `xbutil` utility interacts with the user function.

The `xbmgmt` utility is used for card installation and administration, and requires `sudo` privileges when running it. The `xbmgmt` supported tasks include flashing the card firmware, and scanning the current device configuration.

You can use the `help` command to list the available `xbmgmt` commands and options, and access help for individual commands by using the following:

```
xbmgmt --help <command>
```

For detailed help of each sub-command, use the following:

```
xbmgmt help <subcommand>
```

Set up the `xbmgmt` command as part of the XRT installation using the following scripts:

- For csh shell:

```
$ source /opt/xilinx/xrt/setup.csh
```

- For bash shell:

```
$ source /opt/xilinx/xrt/setup.sh
```

# xclbinutil Utility

The `xclbinutil` utility can create, modify, and report `xclbin` content information.

The available command options are shown in the following table.

**Table 57: xclbinutil Commands**

Option	Description
<code>-h [ --help ]</code>	Print help messages.
<code>-i [ --input ]&lt;arg&gt;</code>	Input file name. Reads <code>xclbin</code> into memory.
<code>-o [ --output ]&lt;arg&gt;</code>	Output file name. Writes in memory <code>xclbin</code> image to a file.
<code>--target &lt;arg&gt;</code>	Target flow for this image. Valid values: <code>hw</code> , <code>hw_emu</code> , and <code>sw_emu</code> .
<code>----private-key &lt;arg&gt;</code>	Private key used in signing the <code>xclbin</code> image.
<code>--ceritifcate &lt;arg&gt;</code>	Certificate used in signing and validating the <code>xclbin</code> image.
<code>--digest-algorithm &lt;arg&gt;</code>	Digest algorithm. Default: <code>sha512</code>
<code>--validate-signature</code>	Validates the signature for the given <code>xclbin</code> archive.
<code>-v [ --verbose ]</code>	Display verbose/debug information
<code>-q [ --quiet ]</code>	Minimize reporting information.
<code>--migrate-forward</code>	Migrate the <code>xclbin</code> archive forward to the new binary format.
<code>--add-section &lt;arg&gt;</code>	Section name to add to the <code>xclbin</code> image. Format: <code>&lt;section&gt;:&lt;format&gt;:&lt;file&gt;</code>
<code>--add-replace-section &lt;arg&gt;</code>	Replace an existing section or add the section of the <code>xclbin</code> image if it does not exist. Format: <code>&lt;section&gt;:&lt;format&gt;:&lt;file&gt;</code>
<code>--add-merge-section &lt;arg&gt;</code>	Add the section if it does not exist, or merge content with an existing section. Format: <code>&lt;section&gt;:&lt;format&gt;:&lt;file&gt;</code>
<code>--remove-section&lt;arg&gt;</code>	Section name to remove from the <code>xclbin</code> image.
<code>--dump-section&lt;arg&gt;</code>	Section to dump. Format: <code>&lt;section&gt;:&lt;format&gt;:&lt;file&gt;</code>
<code>--replace-section&lt;arg&gt;</code>	Section to replace.
<code>--key-value&lt;arg&gt;</code>	Key value pairs. Format: <code>[USER SYS] :&lt;key&gt;:&lt;value&gt;</code>
<code>--remove-key&lt;arg&gt;</code>	Removes the given user key from the <code>xclbin</code> archive.
<code>--add-signature&lt;arg&gt;</code>	Adds a user defined signature to the given <code>xclbin</code> image.
<code>--remove-signature</code>	Removes the signature from the <code>xclbin</code> image.
<code>--get-signature</code>	Returns the user defined signature (if set) of the <code>xclbin</code> image.
<code>--info</code>	Report accelerator binary content. Including: generation and packaging data, kernel signatures, connectivity, clocks, sections, etc
<code>--list-sections</code>	List all possible section names (standalone option).

Table 57: **xclbinutil Commands** (cont'd)

Option	Description
--version	Version of this executable.
--force	Forces a file overwrite.

The following are various use examples of the tool.

- **Reporting xclbin information:** xclbinutil --info --input binary\_container\_1.xclbin
- **Extracting the bitstream image:** xclbinutil --dump-section BITSTREAM:RAW:bitstream.bit --input binary\_container\_1.xclbin
- **Extracting the build metadata:** xclbinutil --dump-section BUILD\_METADATA:HTML:buildMetadata.json --input binary\_container\_1.xclbin
- **Removing a section:** xclbinutil --remove-section BITSTREAM --input binary\_container\_1.xclbin --output binary\_container\_modified.xclbin

For most users, details about the contents and how the `xclbin` was created is desired. This information can be obtained through the `--info` option and reports information on the `xclbin`, hardware platform, clocks, memory configuration, kernel, and how the `xclbin` was generated.

The output of the `xclbinutil` command using the `--info` option is shown below divided into sections.

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

## xclbin Information

```
Generated by:          v++ (2020.1) on Mon Apr 13 20:19:40 MDT 2020
Version:              2.6.436
Kernels:              CopyKernel
Signature:             Bitstream
Content:               d081de98-3fd3-4e9b-bab3-108b42c73101
UUID (xclbin):        862c7020a250293e32036f19956669e5
UUID (IINTF):          Sections: DEBUG_IP_LAYOUT, BITSTREAM, MEM_TOPOLOGY,
IP_LAYOUT,             CONNECTIVITY, CLOCK_FREQ_TOPOLOGY,
BUILD_METADATA,        EMBEDDED_METADATA, SYSTEM_METADATA,
PARTITION_METADATA
```

## Hardware Platform Information

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

## Clocks

Reports the maximum kernel clock frequencies available. Both the clock names and clock indexes are provided. The clock indexes are identical to those reported in [platforminfo Utility](#).

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

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

## Memory Configuration

```
Name:      bank0
Index:    0
Type:      MEM_DDR4
Base Address: 0x0
Address Size: 0x4000000000
Bank Used: No

Name:      bank1
Index:    1
Type:      MEM_DDR4
Base Address: 0x4000000000
Address Size: 0x4000000000
Bank Used: Yes

Name:      bank2
Index:    2
```

```
Type:           MEM_DDR4
Base Address: 0x800000000
Address Size: 0x400000000
Bank Used:    No

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

Name:          PLRAM[0]
Index:         4
Type:          MEM_DDR4
Base Address: 0x100000000
Address Size: 0x20000
Bank Used:    No

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

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

## Kernel Information

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

The following is an example of the reported function definition.

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

The following is an example of the reported ports.

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

Port:          M_AXI_GMEM1
```

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

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

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

```
Instance: krnl_vadd_1
Base Address: 0x0

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

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

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

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

---

## Tool Generation Information

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

```
Generated By
-----
Command: v++
Version: 2018.3 - Tue Nov 20 19:42:42 MST 2018 (SW BUILD: 2394611)
Command Line: v++ -t hw_emu --platform /opt/xilinx/platforms/
xilinx_u200_xdma_201830_1/xilinx_
u200_xdma_201830_1.xpfm --save-temps -l --connectivity.nk
krnl_vadd:1
-g --messageDb binary_container_1.mdb
--temp_dir binary_container_1
--report_dir binary_container_1/reports --log_dir
binary_container_1/logs
--remote_ip_cache /wrk/tutorials/ip_cache
```

```
krnl_vadd.o          -obinary_container_1.xclbin binary_container_1/
Options:           -t hw_emu
                  --platform /opt/xilinx/platforms/xilinx_u200_xdma_201830_1/
xilinx_u200_xdma_201830_1.xpfm
                  --save-temps
                  -l
                  --connectivity.nk krnl_vadd:1
                  -g
                  --messageDb binary_container_1.mdb
                  --temp_dir binary_container_1
                  --report_dir binary_container_1/reports
                  --log_dir binary_container_1/logs
                  --remote_ip_cache /wrk/tutorials/ip_cache
                  -obinary_container_1.xclbin binary_container_1/krnl_vadd.o
=====
== User Added Key Value Pairs
-----
<empty>
=====
```

# xrt.ini File

The Xilinx runtime (XRT) library uses various control parameters to specify debugging, profiling, and message logging when running the host application and kernel execution. These control parameters are specified in a runtime initialization file, `xrt.ini` and used to configure features of XRT at start-up.

If you are a command line user, the `xrt.ini` file needs to be created manually and saved to the same directory as the host executable.

The runtime library checks if `xrt.ini` exists in the same directory as the host executable and automatically reads the file to configure the runtime. You can also specify the location of an `xrt.ini` file at runtime by setting the `XRT_INI_PATH` environment variable to point to the file, for example:

```
export XRT_INI_PATH=/path/to/xrt.ini
```



---

**TIP:** The Vitis IDE creates an `xrt.ini` file automatically based on your run configuration and saves it in the run configuration folder.

---

## Runtime Initialization File Format

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

The following is an example `xrt.ini` file that enables the timeline trace feature, and directs the runtime log messages to the Console view.

```
#Start of Debug group
[Debug]
native_xrt_trace = true
device_trace = fine

#Start of Runtime group
[Runtime]
runtime_log = console
```

There are three groups of initialization keys:

- Runtime

- Debug
  - AIE\_profile\_settings
  - AIE\_trace\_settings
- Emulation

The following tables list all supported keys for each group, the supported values for each key, and a short description of the purpose of the key.

## **Runtime Group**

The Runtime group of switches lets you configure elements of the runtime operation as described below.

**Table 58: Runtime Group Keys and Values**

Key	Valid Values	Description
api_checks	[true false]	Enables or disables OpenCL API checks. <ul style="list-style-type: none"> <li>• true: Enable. This is the default value.</li> <li>• false: Disable.</li> </ul>
cpu_affinity	{N,N,...}	Pins all runtime threads to specified CPUs. Example: <code>cpu_affinity = {4,5,6}</code>
exclusive_cu_context	[true false]	This allows the host application to direct OpenCL to acquire exclusive CU access, so that low-level AXI read/write (xclRegRead and xclRegWrite) can be used for regular kernels.
runtime_log	[null   console   syslog   <filename>]	Specifies where the runtime logs are printed <ul style="list-style-type: none"> <li>• null: Do not print any logs. This is the default value.</li> <li>• console: Print logs to stdout</li> <li>• syslog: Print logs to Linux syslog.</li> <li>• &lt;filename&gt;: Print logs to the specified file. For example, <code>runtime_log=my_run.log</code>.</li> </ul>
verbosity	[0   1   2   3]	Verbosity of the log messages. The default value is 0.

## **Debug Group**

The Debug group of switches define key options for the enabling profiling of the application during runtime, or tracing data transfers and execution. These switches apply to both AI Engine and PL kernels in the Vitis acceleration flow, and let you configure aspects of the runtime to control the frequency of data capture, the events to capture, and the amount of memory to reserve or use for recording trace and profile data.

**Table 59: Debug Options**

Key	Valid Values	Description
aie_profile	[true false]	Enables the runtime configuration and polling of AI Engine hardware performance counters. Available on VCK190 hardware and hardware emulation runs. <ul style="list-style-type: none"> <li>• true: Enable.</li> <li>• false: Disable. This is the default value.</li> </ul>
aie_trace	[true false]	Enables the runtime configuration and collection of AI Engine event trace. Available on VCK190 hardware runs only. <ul style="list-style-type: none"> <li>• true: Enable.</li> <li>• false: Disable. This is the default value.</li> </ul>
aie_status	[true false]	Enables the polling of AI Engine status information. Available on VCK190 hardware and hardware emulation runs.
aie_status_interval_us	integer (default=1000us)	Controls the interval at which AI Engine status information is captured. Specified in microseconds.
app_debug	[true false]	Enables the OpenCL application debug for the host code when debugging with GDB. <ul style="list-style-type: none"> <li>• true: Enable.</li> <li>• false: Disable. This is the default value.</li> </ul>
continuous_trace	[true false]	Enables the continuous dumping of files for trace and the continuous reading of device data into the host. <ul style="list-style-type: none"> <li>• true: Enable.</li> <li>• false: Disable. This is the default value.</li> </ul> <p><b>Note:</b> This switch only has an effect if device_trace is enabled.</p>
device_counters	[true false]	Enables device counter offload only, without enabling trace functionality.
device_trace	[off fine coarse accel]	Enables the collection of data from monitors inserted on the PL to add to summary and trace. <ul style="list-style-type: none"> <li>• accel: Traces compute unit starts/stops.</li> <li>• coarse: Lumps all reads/writes together under each execution of a compute unit.</li> <li>• fine: Tracks everything as it happens.</li> <li>• off: Turns off reading and reporting of device-level trace during runtime. This is the default value.</li> </ul>

Table 59: Debug Options (cont'd)

Key	Valid Values	Description
host_trace	[true false]	<p>Enables trace of host code based on the first protocol encountered.</p> <p><b>TIP:</b> If your host application uses both OpenCL and XRT native API you should manually specify both <code>opencl_trace</code> and <code>native_xrt_trace</code> to capture all events.</p>
lop_trace	[true false]	<p>Enables generation of lower overhead OpenCL API host trace. Should not be used with other OpenCL options.</p> <ul style="list-style-type: none"> <li>• true: Enable.</li> <li>• false: Disable. This is the default value.</li> </ul>
native_xrt_trace	[true false]	<p>Enables generation of the Native C/C++ API trace. This also generates the tables for "Host Data Transfer from/to Global memory" in the Profile Summary.</p> <ul style="list-style-type: none"> <li>• true: Enable.</li> <li>• false: Disable. This is the default value.</li> </ul>
opencl_trace	[true false]	<p>Enables generation of OpenCL API host trace.</p> <ul style="list-style-type: none"> <li>• true: Enable.</li> <li>• false: Disable. This is the default value.</li> </ul>
pl_deadlock_detection	[true false]	Enables deadlock detection for PL kernels.
power_profile	[true false]	<p>Enables the polling of power data during the execution of the application.</p> <ul style="list-style-type: none"> <li>• true: Enable.</li> <li>• false: Disable. This is the default value.</li> </ul> <p><b>Note:</b> This feature is not supported on certain platforms including AWS.</p>
power_profile_interval_ms	<int>(default=20)	<p>Controls the interval of reading the power counters in milliseconds. The default interval is 20 ms.</p> <p><b>Note:</b> This switch only has an effect if <code>power_profile = true</code>.</p>
profile_api	[true false]	<p>Enables access to HAL API directly from the host application to read counters on device profiling monitors during execution.</p> <ul style="list-style-type: none"> <li>• true: Enable.</li> <li>• false: Disable. This is the default value.</li> </ul>

Table 59: Debug Options (cont'd)

Key	Valid Values	Description
stall_trace	[off all dataflow memory pipe]	<p>Specifies the type of device-side stalls to capture and report in the timeline trace. The default is off.</p> <ul style="list-style-type: none"> <li>off: Turn off stall trace information.</li> <li>all: Record all stall trace information.</li> <li>dataflow: Intra-kernel streams (for example, writing to full FIFO between dataflow blocks).</li> <li>memory: External memory stalls (for example, AXI4 read from the DDR memory).</li> <li>pipe: Inter-kernel pipe for OpenCL kernels (for example, writing to full pipe between kernels).</li> </ul> <p><b>Note:</b> This switch only has an effect if <code>device_trace</code> is enabled.</p>
trace_buffer_offload_interval_ms	<int>	<p>Controls the reading of device data from the device to the host in milliseconds (ms). The default is 10 ms.</p> <p><b>Note:</b> This switch only has an effect if <code>device_trace</code> is enabled.</p>
trace_buffer_size	<string>	<p>If the <code>.xclbin</code> was created with memory offload of trace specified, as described in <a href="#">Profile Options</a>, this switch determines the size of the buffer to allocate in memory to capture trace data. The default is 1M.</p> <p><b>Note:</b> This switch only has an effect if <code>device_trace</code> is enabled.</p>
trace_file_dump_intervals	<int>	<p>Controls the time between dumping of trace files in seconds (s). The default is 5s.</p> <p><b>Note:</b> This switch only has an effect if <code>device_trace</code> is enabled.</p>
vitis_ai_profile	[true false]	<p>Profile summary and other files come from Vitis AI application layer.</p> <ul style="list-style-type: none"> <li>true: Enable.</li> <li>false: Disable. This is the default value.</li> </ul>
xocl_debug	[true false]	<p>Generates the <code>xocl.log</code> file when enabled.</p> <p>When any trace options are also enabled, the debug log is added to the <code>xrt.run_summary</code> to view in Vitis Analyzer.</p>

Table 59: Debug Options (cont'd)

Key	Valid Values	Description
xrt_trace	[true false]	<p>Enables generation of low-level HW shim function trace during HW runs. This will be disabled when used with native_xrt_trace.</p> <ul style="list-style-type: none"> <li>• true: Enable.</li> <li>• false: Disable. This is the default value.</li> </ul>

### AIE\_profile\_settings Group

The options specified in this group are applied only if `aie_profile=true` under the [Debug] group.

Table 60: AI Engine Profile Options

Key	Valid Values	Description
graph_based_aie_metrics	<graph name all>:<kernel name all>:<off heat_map stalls execution floating_point write_bandwidths read_bandwidths aie_trace>	<p>Specify the metric sets reported by the AI Engine module of AI Engine tiles on a graph-by-graph basis.</p> <p><b>IMPORTANT!</b> Currently, only <code>all</code> is supported for kernel specification.</p> <p>Controls the configuration of the statistics read from the AIE core performance counters for the entire AI Engine graph application.</p> <p><u>heat_map</u>: profile active/stall cycles and vector instruction usage</p> <p><u>stalls</u>: profile the different types of stalls (i.e., memory, stream, lock, and cascade)</p> <p><u>execution</u>: profile the AI Engine instructions</p> <p><u>floating_point</u>: profile floating point exceptions</p> <p><u>write_bandwidths</u>: profile the write bandwidth of streams and cascades</p> <p><u>read_bandwidths</u>: profile the read bandwidths of streams and cascades</p> <p><u>aie_trace</u>: profile amount and stalls of event trace from core and memory modules</p>

Table 60: AI Engine Profile Options (cont'd)

Key	Valid Values	Description
graph_based_aie_memory_metrics	<graph name all>:<kernel name all>:<off conflicts dma_locks dma_stalls_s2mm dma_stalls_mm2s write_bandwidths read_bandwidths>	<p>Specify the metric sets reported by the memory module of AI Engine tiles on a graph-by-graph basis.</p> <p><b>IMPORTANT!</b> Currently, only <code>a11</code> is supported for kernel specification.</p>
tile_based_aie_metrics	<{<column>,<row>} all>:<off heat_map stalls execution floating_point write_bandwidths read_bandwidths aie_trace> ; {<mincolumn,<minrow>}:<{<maxcolumn>,<maxrow>}>:<off heat_map stalls execution floating_point write_bandwidths read_bandwidths aie_trace>	<p>Specify the metric sets reported by the AI Engine module of AI Engine tiles on a tile-by-tile basis. This can be used in conjunction with graph-by-graph selection and will take priority on the specified tiles.</p> <p>Refer to descriptions from <code>graph_based_aie_metrics</code></p>
tile_based_aie_memory_metrics	<{<column>,<row>} all>:<off conflicts dma_locks dma_stalls_s2mm dma_stalls_mm2s write_bandwidths read_bandwidths> ; {<mincolumn,<minrow>}:<{<maxcolumn>,<maxrow>}>:<off conflicts dma_locks dma_stalls_s2mm dma_stalls_mm2s write_bandwidths read_bandwidths>	<p>Specify the metric sets reported by the memory module of AI Engine tiles on a tile-by-tile basis. This can be used in conjunction with graph-by-graph selection and will take priority on the specified tiles.</p> <p>Refer to descriptions from <code>graph_based_aie_memory_metrics</code></p>
tile_based_interface_tile_metrics	<column all>:<off input_bandwidths output_bandwidths packets>[:<channel>] ; <mincolumn>:<maxcolumn>:<off input_bandwidths output_bandwidths packets>[:<channel>]	<p>Specify the metric sets reported by the AI Engine interface tiles on a tile-by-tile basis.</p> <p><b>Note:</b> Interface tiles are separate from the AI Engine tiles and have different metric sets.</p>
interval_us	<int>	<p>Controls the interval of reading the AI Engine counter values in microseconds (μs). The default interval is 1000 μs.</p> <p><b>Note:</b> This switch only has an effect if <code>aie_profile = true</code>.</p>

## AIE\_trace\_settings Group

The options specified in this group are applied only if `aie_trace=true` under the [Debug] group.

**Table 61: AI Engine Trace Options**

Key	Valid Values	Description
<code>buffer_size</code>	<code>&lt;string&gt;</code> (default=8M)	Controls the total size of the buffers allocated for AI Engine event trace. This size is partitioned evenly into the number of different trace streams coming out of the AI Engine. The default is 8M.  <b>Note:</b> This switch only has an effect if <code>aie_trace = true</code> .
<code>buffer_offload_interval_us</code>	integer (default=10ms)	Interval, in milliseconds, between reading of PLIO mode AI Engine trace from device to Host memory.
<code>periodic_offload</code>	true/false (default=true)	Enables continuous offload of PLIO mode AI Engine trace. Generated AI Engine trace output files (one per stream) gets appended with new trace data.
<code>file_dump_interval_s</code>	integer (default=5s)	Interval, in seconds, between writing (appending) of raw AI Engine trace data to output files.
<code>graph_based_aie_tile_metrics</code>	<code>string("")</code> <code>&lt;graph name all&gt;:&lt;kernel name all&gt;:&lt;off functions functions_partial_stalls functions_all_stalls&gt;</code>	Specify the metric sets reported by the AI Engine module of AI Engine tiles on a graph-by-graph basis.  <b>IMPORTANT!</b> Currently, only <code>a11</code> is supported for kernel specification.
<code>tile_based_aie_tile_metrics</code>	<code>string("")</code> <code>&lt;{&lt;column&gt;,&lt;row&gt;} all&gt;:&lt;off functions functions_partial_stalls functions_all_stalls&gt;[:&lt;memory_stalls stream_stalls cas cascade_stalls lock_stalls&gt;]</code> <code>{&lt;mincolumn,&lt;minrow&gt;}:</code> <code>{&lt;maxcolumn,&lt;maxrow&gt;}:&lt;off functions functions_partial_stalls functions_all_stalls&gt;</code>	Specify the metric sets reported by the AI Engine module of AI Engine tiles on a tile-by-tile basis.  <b>IMPORTANT!</b> Currently, only <code>a11</code> is supported for kernel specification.
<code>reuse_buffer</code>	true/false (false)	

## Emulation Group

The Emulation group of switches apply to the emulation environments and the Vivado simulator.

**Table 62: Emulation Group Keys and Values**

Key	Valid Values	Description
aliveness_message_interval	Any integer	Specifies the interval in seconds that aliveness messages need to be printed. The default is 300.
debug_mode	[off batch gui]	<p>Specifies how the waveform is saved and displayed during emulation.</p> <ul style="list-style-type: none"> <li>off: Do not launch simulator waveform GUI, and do not save <code>wdb</code> file. This is the default value.</li> <li>batch: Do not launch simulator waveform GUI, but save <code>wdb</code> file</li> <li>gui: Launch simulator waveform GUI, and save <code>wdb</code> file</li> </ul> <p><b>Note:</b> The kernel needs to be compiled with debug enabled (<code>v++ -g</code>) for the waveform to be saved and displayed in the simulator GUI.</p>
kernel-dbg	[true false]	<p>Enables kernel debug functionality during software emulation as described in <a href="#">Command Line Debug Flow</a>.</p> <ul style="list-style-type: none"> <li>true: Enable.</li> <li>false: Disable. This is the default value.</li> </ul>
print_infos_in_console	[true false]	<p>Controls the printing of emulation info messages to user's console. Emulation info messages are always logged into a file called <code>emulation_debug.log</code></p> <ul style="list-style-type: none"> <li>true: Print in user's console. This is the default value.</li> <li>false: Do not print in user console.</li> </ul>
print_warnings_in_console	[true false]	<p>Controls the printing emulation warning messages to user's console. Emulation warning messages are always logged into a file called <code>emulation_debug.log</code>.</p> <ul style="list-style-type: none"> <li>true: Print in user's console. This is the default value.</li> <li>false: Do not print in user console.</li> </ul>
print_errors_in_console	[true false]	<p>Controls printing emulation error messages in user's console. Emulation error messages are always logged into the <code>emulation_debug.log</code> file.</p> <ul style="list-style-type: none"> <li>true: Print in user's console. This is the default value.</li> <li>false: Do not print in user's console.</li> </ul>
user_pre_sim_script	Path to Tcl file	<p>For the first run, run simulation in GUI mode. Add signals that you want to add. Copy the commands from the Tcl console and save into a Tcl script.</p> <p>For the next run, pass the Tcl script in batch mode.</p>

Table 62: Emulation Group Keys and Values (cont'd)

Key	Valid Values	Description
user_post_sim_script	Path to Tcl file	Any post operations can be specified in the Tcl and pass to the switch. All the command provided in the Tcl gets executed after simulation is completed.
xtlm_aximm_log	[true false]	Enables the XTLM AXI4 Memory Map transaction logging at runtime and you could see all the transactions in the <code>xsc_report.log</code> file.
xtlm_axis_log	[true false]	Enables the XTLM AXI4-Stream transaction logging at runtime and you could see all the transactions in the <code>xsc_report.log</code> file.
timeout_scale	na/ms/sec/min	Timeout support for <code>c1PollStream</code> API in emulation. Provides a scale for the timeout specified in <code>c1PollStream</code> API. The timeout specified in the code is specified in ms, and might not work for emulation. Therefore use the <code>timeout_scale</code> to map ms to another scale if needed for emulation.  <b>IMPORTANT!</b> Timeout is not enabled in emulation by default. Use this option to enable <code>c1PollStream</code> timeout.

# HLS Pragmas

## Optimizations in Vitis HLS

In the Vitis software platform, a kernel defined in the C/C++ language, or OpenCL™ C, must be compiled into the register transfer level (RTL) that can be implemented into the programmable logic of a Xilinx device. The `v++` compiler calls the Vitis High-Level Synthesis (HLS) tool to synthesize the RTL code from the kernel source code.

The HLS tool is intended to work with the Vitis IDE project without interaction. However, the HLS tool also provides pragmas that can be used to optimize the design, reduce latency, improve throughput performance, and reduce area and device resource usage of the resulting RTL code. These pragmas can be added directly to the source code for the kernel.

The HLS pragmas include the optimization types specified in the following table.

For detailed pragma information, refer to the *Vitis High-Level Synthesis User Guide (UG1399)*.

*Table 63: Vitis HLS Pragmas by Type*

Type	Attributes
Kernel Optimization	<ul style="list-style-type: none"> <li>• <code>pragma HLS aggregate</code></li> <li>• <code>pragma HLS bind_op</code></li> <li>• <code>pragma HLS bind_storage</code></li> <li>• <code>pragma HLS expression_balance</code></li> <li>• <code>pragma HLS latency</code></li> <li>• <code>pragma HLS reset</code></li> <li>• <code>pragma HLS top</code></li> </ul>
Function Inlining	<ul style="list-style-type: none"> <li>• <code>pragma HLS inline</code></li> </ul>
Interface Synthesis	<ul style="list-style-type: none"> <li>• <code>pragma HLS interface</code></li> </ul>
Task-level Pipeline	<ul style="list-style-type: none"> <li>• <code>pragma HLS dataflow</code></li> <li>• <code>pragma HLS stream</code></li> </ul>
Pipeline	<ul style="list-style-type: none"> <li>• <code>pragma HLS pipeline</code></li> <li>• <code>pragma HLS occurrence</code></li> </ul>
Loop Unrolling	<ul style="list-style-type: none"> <li>• <code>pragma HLS unroll</code></li> <li>• <code>pragma HLS dependence</code></li> </ul>
Loop Optimization	<ul style="list-style-type: none"> <li>• <code>pragma HLS loop_flatten</code></li> <li>• <code>pragma HLS loop_merge</code></li> <li>• <code>pragma HLS loop_tripcount</code></li> </ul>

**Table 63: Vitis HLS Pragmas by Type (cont'd)**

Type	Attributes
Array Optimization	<ul style="list-style-type: none"><li>• <code>pragma HLS array_partition</code></li><li>• <code>pragma HLS array_reshape</code></li></ul>
Structure Packing	<ul style="list-style-type: none"><li>• <code>pragma HLS aggregate</code></li><li>• <code>pragma HLS dataflow</code></li></ul>

# Using the Vitis Analyzer

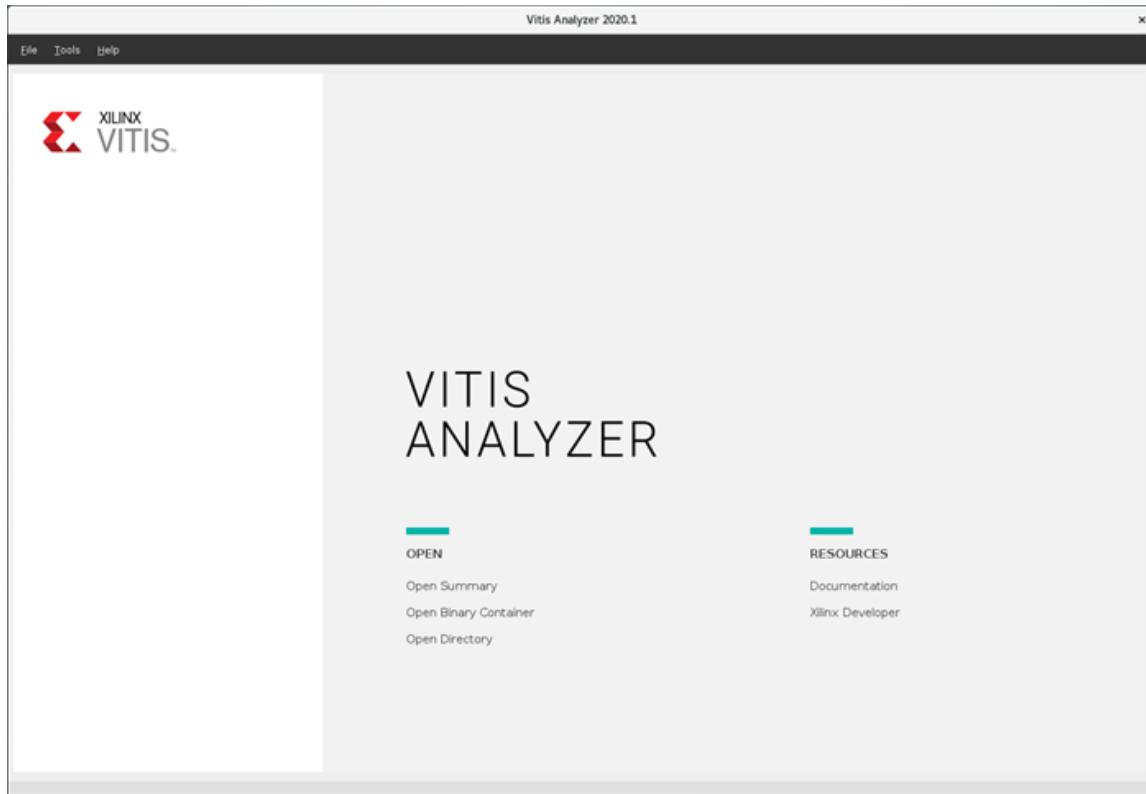
This section contains the following chapters:

- [Working with Summary Reports](#)
- [Vitis Analyzer GUI and Window Manager](#)
- [Platform and System Diagrams](#)
- [AI Engine Graphs and Arrays](#)
- [Link Summary: Multiple Strategies and Timing Reports](#)
- [Setting Guidance Thresholds](#)
- [Creating an Archive File](#)

The Vitis™ analyzer is a utility that allows you to view and analyze the reports generated while building and running the application. It is intended to let you review reports generated by both the Vitis compiler when the application is built, and the Xilinx® Runtime (XRT) library when the application is run. The Vitis analyzer can be used to view reports from both the `v++` command line flow, and the Vitis integrated design environment (IDE). You will launch the tool using the `vitis_analyzer` command (see [Setting Up the Vitis Environment](#)).

When first launched, the Vitis analyzer opens with a home screen that lets you open summary files, binary containers, or directories. Clicking any of these links opens a file browser that allows you to select a specific file of the type described.

Figure 50: Vitis Analyzer – Home Screen



- **Open Summary:** Report Summaries are collections of reports related to specific stages of application development in the Vitis tool. There are summaries created for the two steps of the build process, compile and link, and from the run process when the application is executed. Selecting **Open Summary** lets you open one of the following:
  - **Compile Summary:** The Compile Summary report is generated by the `v++` command during compilation, provides the status of the kernel compilation process. This collection of reports includes a Summary report, Kernel Estimate for timing and resource estimates, Kernel Guidance offering any suggestions for compilation, and the HLS Synthesis log from Vitis HLS.
  - **Link Summary:** The Link Summary report is created by the `v++` command during linking and creation of the `.xclbin` file. This collection of reports includes a Summary report, System and Platform Diagrams to illustrate the hardware design, System Estimate providing timing and resources estimates, System Guidance offering any suggestions for improving linking and the performance of the system, and the Vivado Automation Summary providing design details such as interface connections, clocks, resets, and interrupts.



**TIP:** *Timing Summary and Utilization are only generated when the build targets Hardware (as opposed to emulation).*

When you open a Link Summary, the Vitis analyzer will automatically open the associated Compile Summaries of kernels that were linked into the `.xclbin` file.

- **Package Summary:** The Package Summary report is generated by the `v++ --package` command, and provides information related to the generation of emulation scripts and SD card output. When viewing the Package Summary report the tool also references the configuration file used on the command line, and the log file generated.
- **Run Summary:** The Run Summary report is created by the XRT library during the application execution, and provides a summary of the run process. When viewing the Run Summary report the tool also references the following reports generated during the application run: Guidance, Profile Summary, Timeline Trace, Platform and System Diagrams, and Simulation Waveforms when enabled.



**IMPORTANT!** Reports generated during the runtime execution of an application generally require some prior setup as described in [Profiling the Application](#).

When you run the application after the `v++` build process, the ID from the Link Summary is assigned to the Run Summary. When you open Run Summary and Link Summary, Vitis Analyzer links them based on the shared ID.

- **AI Engine Compile and Run Summaries:** Vitis analyzer also lets you view reports from the Versal AI Engine compile and run processes. These reports are generated by the `aiecompiler` and `aiesimulator` as described in the [AI Engine Tools and Flows User Guide \(UG1076\)](#).



**TIP:** In the Vitis IDE, the project hierarchy consists of a top-level system project with sub-projects which contain elements of the design: the processor application, the hardware kernels, the hardware link project, and the AI Engine project. The various summary reports described above can be found in the appropriate sub-project: Compile Summary will be in the specific hardware kernel projects, Link Summary will be in the hardware link project, Package Summary will be found in the system project, the Run Summary will be found in the processor application, and AI Engine summaries will be found in the `./Work` folder or the `./aiesimulator` output.

- **Open Binary Container:** Opens the selected `.xclbin` file to display the Platform Diagram and the System Diagram for the build.
- **Open Directory:** Specifies a directory to open. The tool recursively examines the contents of the directory and displays a dialog box allowing you to select which type of files to open and which individual files to open.



**TIP:** The Open Recent section of the home screen provides a list of recently opened summaries and reports to let you quickly reopen them.

The `vitis_analyzer` command lets you open the tool to the home screen, as discussed above, or specify a file to load when opening the tool. You can open a file by specifying the name of the file to open. You can open the Vitis analyzer with any of the files supported by the tool, as described in [Working with Summary Reports](#). For example:

```
vitis_analyzer xrt.run_summary
```

You can access the command help for `vitis_analyzer` command by typing the following:

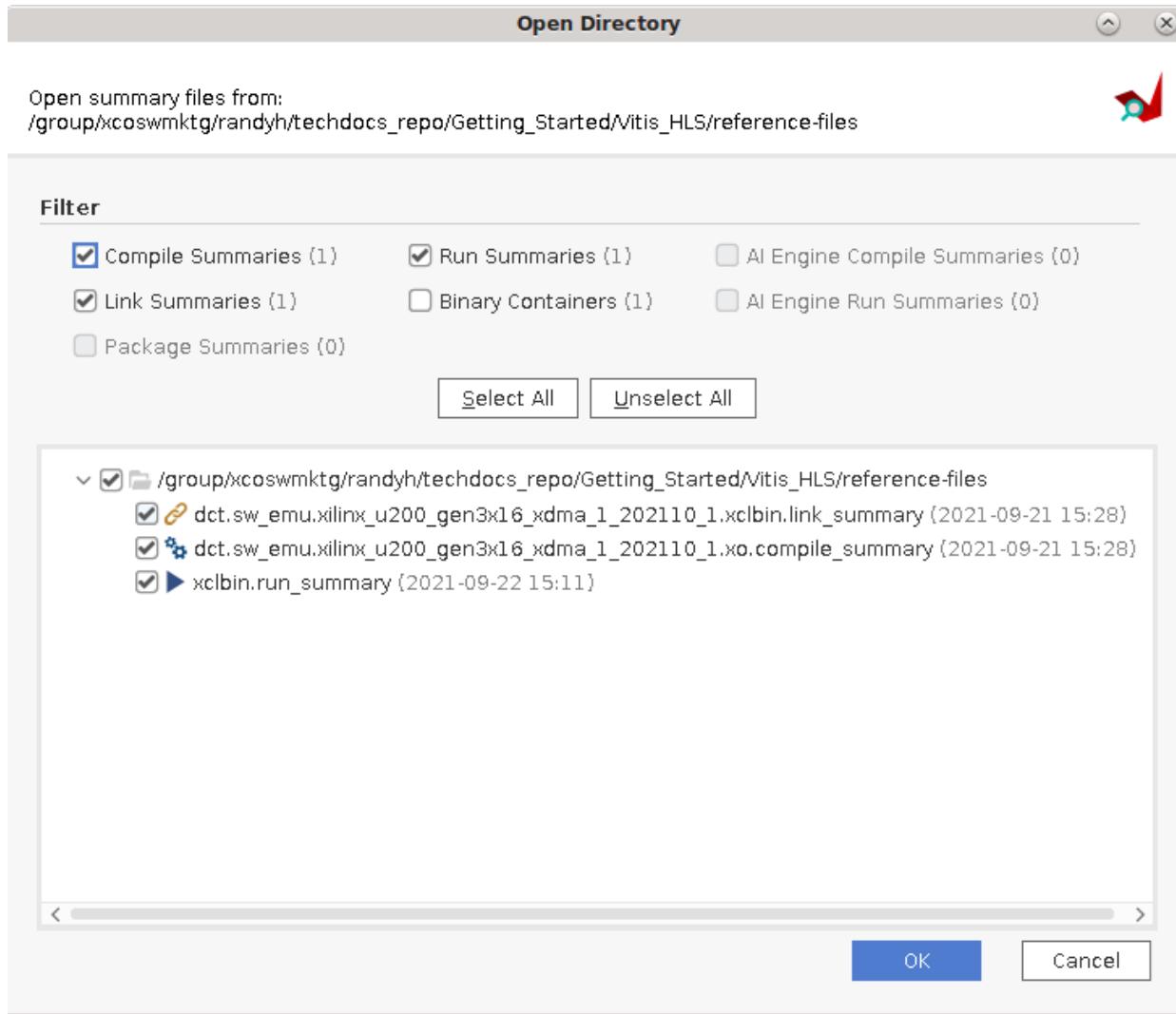
```
vitis_analyzer -help
```

# Working with Summary Reports

Generally, the Summary reports provide a great overview of the specific steps in building and profiling the application to get a good view of where the application is with regard to performance and optimization. For individual kernels, review the Compile Summary. For the device binary (`xclbin`), start with the Link Summary, which also loads the Compile Summaries for linked kernels. For embedded processor and Versal® AI Engine systems, review the Package Summary. For profiling data related to the application execution, start with the Run Summary.

In addition, the **File** menu offers commands to let you open individual reports, and directories of reports.

Figure 51: Open Directory



- **Open Directory:** Specifies a directory to open. The tool recursively examines the contents of the directory and displays a dialog box allowing you to select which type of files to open and which individual files to open.
- **Open Binary Container:** FPGA binary, <name>.xclbin, created by the compilation and linking process as described in [Section IV: Building and Running the Application](#).
- **Open Report:** Opens one of the report files generated by the Vitis™ core development kit during compilation, linking, or running the application. The reports you can open include:
  - **Profile Summary:** Refer to [Profile Summary Report](#).
  - **Waveform:** Waveform database and waveform config file as described in [Waveform View and Live Waveform Viewer](#).

- **Utilization:** A resource utilization report generated by the Vivado® tool when you build the system hardware (HW) target.

The Vitis analyzer can also open Kernel Estimate, Operation Trace, AI Engine Trace, Timeline Trace, System Estimate, Log, and Timing Summary reports. Refer to [Profiling the Application](#) for more information on the individual reports generated by the build and run processes.

The Vitis analyzer displays log files rendered for improved readability. These improvements include line wrapping, message severity tagging (Error, Critical Warning, Warning, Info, Status), added hyperlinks to referenced files, search capability, and live log monitoring. This last feature lets you open a log file in-process and see it rendered in real time.

## Viewing Report Contents

The Vitis analyzer features depend on the specific report you are viewing. When the report is structured like a spreadsheet you interact with the report like a spreadsheet, selecting rows or cells of data, and sorting columns by clicking the column header. When the report is graphical in nature, you can interact with the report by zooming into the report to view details, and zooming out to view more information. The Vitis analyzer supports the following mouse strokes to let you quickly zoom into and out of a graphical report:

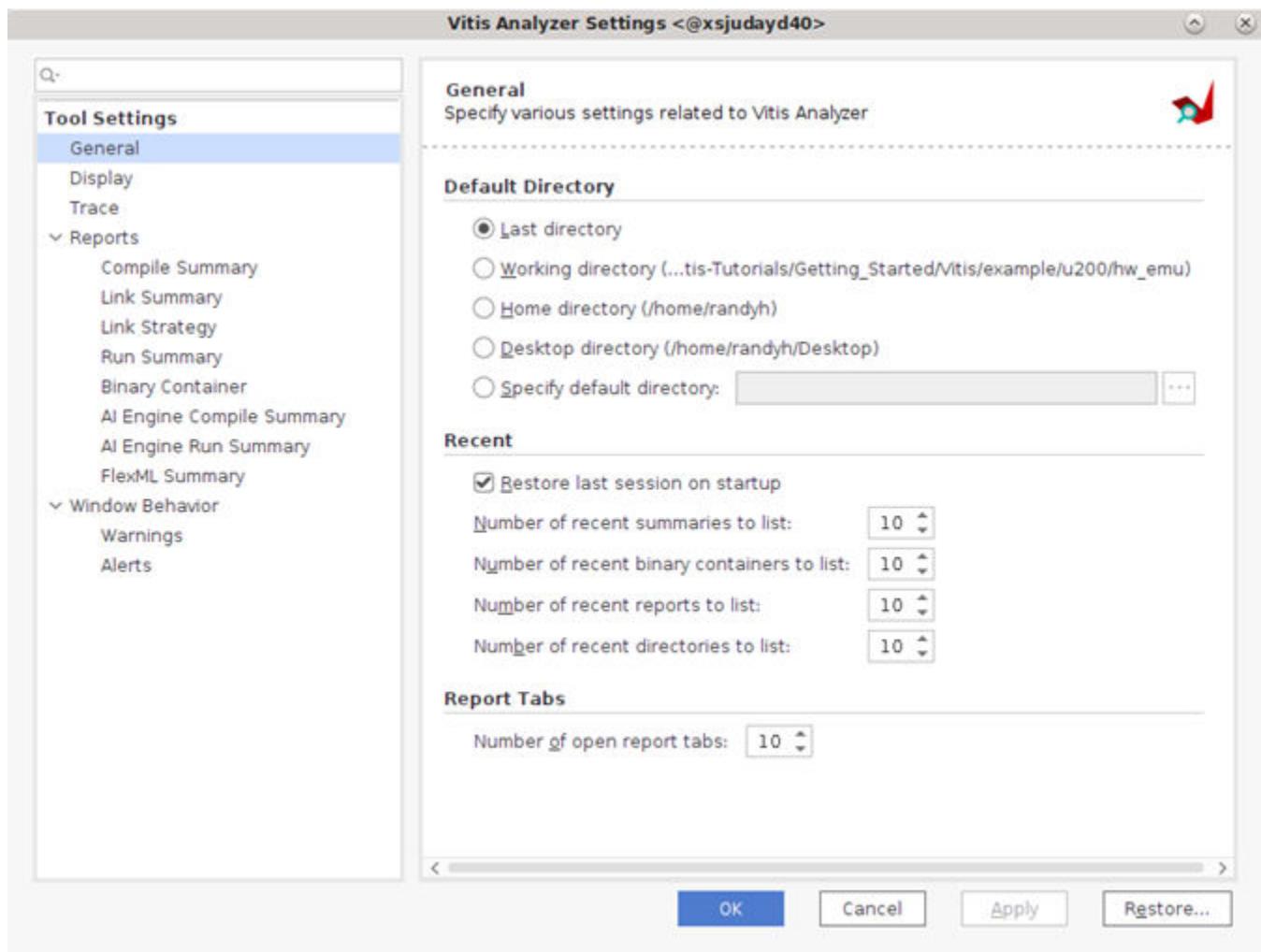
- **Zoom In:** Press and hold the left mouse button while dragging the mouse from top left to bottom right to define an area to zoom into.
- **Zoom Out:** Press and hold the left mouse button while drawing a diagonal line from lower left to upper right. This zooms out the window by a variable amount. The length of the line drawn determines the zoom factor applied. Alternatively, press **Ctrl** and scroll the wheel mouse button down to zoom out.
- **Zoom Fit:** Press and hold the left mouse button while drawing a diagonal line from lower right to upper left. The window zooms out to display the entire device.
- **Horizontal scrolling:** In a report such as the Timeline Trace, you can hold the shift button down while scrolling the middle mouse roller to scroll across the timeline.
- **Panning:** Press and hold the wheel mouse button while dragging to pan.

---

## Configuring the Vitis Analyzer

The **Tools**→**Settings** command opens the Vitis Analyzer Settings dialog box as shown below.

Figure 52: Settings Dialog Box



1. In the General settings, the following can be configured:
  - **Default Directory:** Specifies the default directory used by the Vitis™ analyzer when it is opened.
  - **Recent:** Configures the tool to restore the workspace when reopening the Vitis analyzer, and specify the number of entries to display for **File → Open Recent** commands.
  - **Report Tabs:** Defines the number of reports and views that can be opened in the main Reports window.
2. In the Display settings you can configure the following features of the display:
  - **Scaling:** Sets the font scaling to make the display easier to read on high resolution monitors. Use OS font scaling uses the value set by the OS for your primary monitor. User-defined scaling allows you to specify a value specific to the Vitis analyzer.

- **Spacing:** Sets the amount of space used by the Vitis IDE. Comfortable is the default setting. Compact reduces the amount of space between elements to fit more elements into a smaller space.
3. In the Trace settings you can configure the temporary directory to write trace data to as it is being captured.
  4. The Reports section configures the Vitis analyzer to also open specified reports when opening the Compile Summary, Link Summary, Run Summary, or Binary Container reports:
    - **Compile Summary:** Select which reports are listed in the Report Navigator view, and opened with the Compile Summary.
    - **Link Summary:** Select which reports are listed and opened with the Link Summary.
    - **Run Summary:** Select which reports are listed and opened with the Run Summary. Also, select which reports are dynamically updated while the application is running. Timeline Trace reports can be automatically reloaded when continuous trace data is captured as described in [Continuous Trace Capture](#). This lets you observe the trace data in Vitis analyzer as it is captured from your running application.
    - **Binary Container:** Select which reports are listed and opened with the Binary Container.
    - **AI Engine Compile Summary:** Select which reports are listed and opened with the AI Engine Compile Summary.
    - **AI Engine Run Summary:** Select which reports are listed and opened with the AI Engine Run Summary.
5. For Window Behavior settings, the following can be configured:
    - **Warnings:** Shows warning when exiting or just exits the Vitis analyzer.
    - **Alerts:** Issues an alert when you are running the tool on an unsupported operating system.

After configuring the settings, click **OK**, **Apply**, or **Cancel**. You can also use the **Restore** command to restore the defaults settings of the tool.

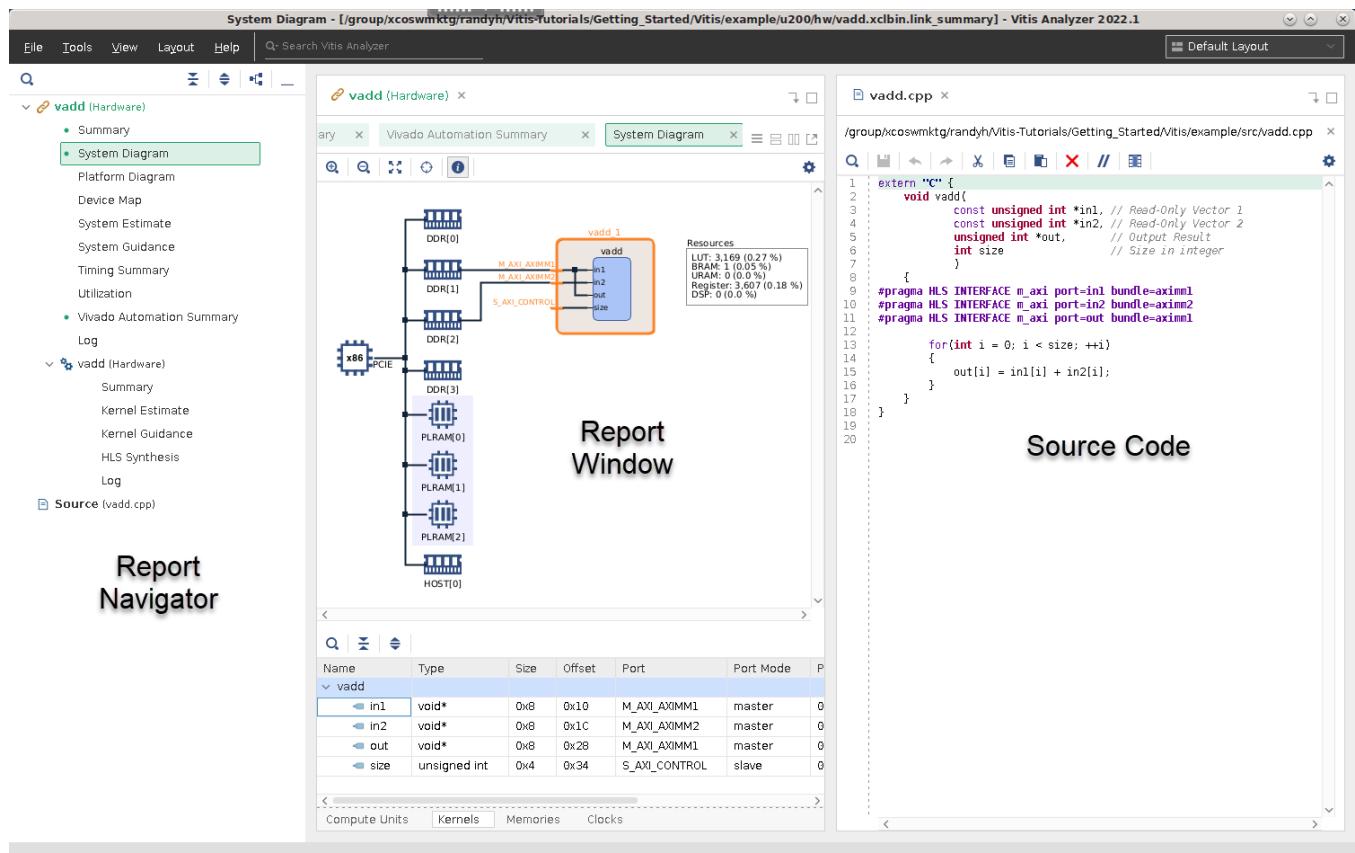
## Layout

The Vitis analyzer also lets you arrange reports in different layouts, and save those layouts for reuse. This allows you to arrange two reports for comparison, or to check one report against the other. The **Layout** menu in the main menu lets you switch between existing layouts, or save a custom report layout using the **Save Layout As** command. You can also **Reset Layout** to return it to its saved configuration, or **Remove Layout** to eliminate saved layout.

# Vitis Analyzer GUI and Window Manager

As shown below, the Vitis™ analyzer workspace is arranged into three views, which includes the Report Navigator, Reports, and the Source Code view. The different views can be opened, closed, and rearranged as needed.

*Figure 53: Vitis Analyzer Workspace*



- **Report Navigator:** On the left side, this view lists all open summary files and associated reports. You can use this view to quickly find and open a report. In the figure above you can see the Link Summary is opened, and the Compile Summary is contained within it, and all their related reports are listed in the Report Navigator.

When you click any file in Report Navigator, it opens as a new tab in the Report view. Opening a file adds a green dot next to the report name in Report Navigator to let you quickly determine if a report is already open in the tool.



**TIP:** You can right-click the Compile Summary file to **Open HLS Project**, or right-click the Link Summary or Run Summary to **Open Vivado Project** if needed. The Vivado project cannot be opened for the Software Emulation build.

- **Report Window:** The center area displays the contents of the summary files and open reports. You can have multiple reports open in the Reports view, and quickly change from one report to another by selecting the window tab at the top of the view.

All the reports related to the Compile Summary, Link Summary, or Run Summary are grouped together within a single container. You can arrange the reports for a container in different ways, using the New Horizontal Group, New Vertical Group, or Float commands for the reports in a container. Multiple Summary reports can be opened and the contents managed as collections.

Reports that are currently opened in Vitis analyzer will display an "out-of-date" banner if the summary files or reports have been updated on the disk, due to recompiling or rerunning the application for instance. You can keep working in the open report, or reload the updated files.

The screenshot shows the Vitis Analyzer Report window with the following details:

- Toolbar:** Summary, Profile Summary, Run Guidance, System Diagram, Platform Diagram, Application Tip.
- Status Bar:** Report is out of date because files have been modified on disk. Reload, Reload all.
- Filter Bar:** Includes icons for search, filter, and sort, followed by 1 warning, 7 Infos, 36 Met, and Show All.
- Table Headers:** Name, Threshold, Actual, Details, Resolution.
- Data Rows:**
  - Profiling (1)
    - Kernel: runOnfpga (1)
      - Argument: bloom\_filter (1)
        - KERNEL\_BUFFER\_PLRAM\_USAGE > 128KB 65536 Buffer for argument [bloom\\_filter](#) on kernel [runOnfpga](#) did not use PLRAMs. Use PLRAMs for sm buffers. Click [here](#).

By default, some graphical reports such as the Timeline Trace are displayed in a hierarchical view, which presents the information according to the design hierarchy but consumes significant real estate in the display. As an alternative, you can select the **Flatten Signal** command on the toolbar to display to display the report with flattened signals and eliminate unnecessary spacing between lines. This feature is useful when there is less display area to work with, or when comparing multiple files.

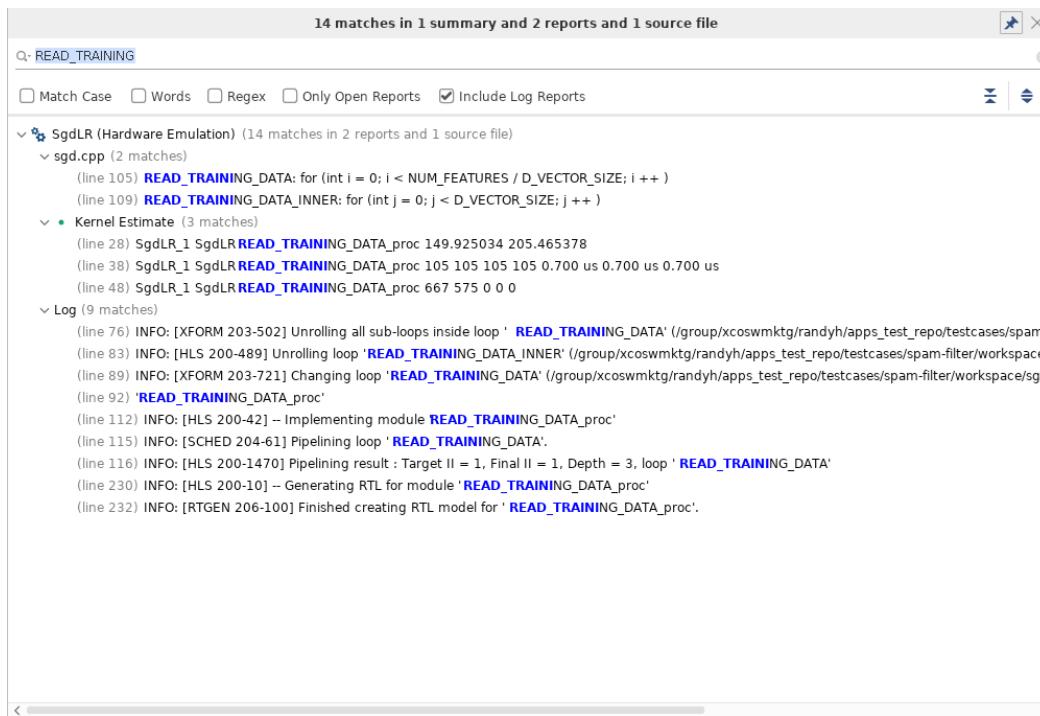
You can also click the **Filter** command on the top right of the report window to specify the elements of the report to display. This allows you to limit the information displayed in the report window, and lets you focus on the information you are interested in for review. You can save the filters using the **Save Filter As** command, give each filter a meaningful name, and reuse the saved filters in other reports. This is useful when users create multiple filters, e.g. pipe stalls, memory stalls, etc. The saved filters are associated with the specific report. They cannot be accessed from other reports or summaries.



**TIP:** The flattened or hierarchical view of the report is saved when the report is closed, but any filters are disabled to avoid confusion as to the actual content of the reports. You can simply reapply any saved filters after reopening the reports.

- **Source Code:** The optional Source Code view is opened on the right side of the workspace. This lets you view and edit kernel source code, based on feedback from the System Guidance report for instance. You can open the Source Code window by selecting a link in the Guidance report, or by right-clicking the Compile Summary in the Report Navigator and clicking **Open Source**.
- **Global Search:** Vitis analyzer provides a search field next to the **Help** command in the main menu at the top of the display. You can use it to search the loaded Summary, associated reports, and source files. The search dialog box provides options to limit the scope of the search of the search terms, as shown in the following figure.

Figure 54: Global Search



The Report Navigator and Source Code views can be collapsed by clicking the **Minimize** button in the toolbar, and restored by clicking the tab for the collapsed view.

To close all the open source code views, select the **File → Close All Sources** command.

To close all the open reports associated with a Summary report, such as the Link Summary, right-click the Summary in the Report Navigator view and select **Close Tabs**. This closes all open reports associated with the summary in the Report view.

To close a Summary file, such as the Link Summary, right-click the file in the Report Navigator area and select **Close File**. Closing the Summary file closes all associated reports and files. Therefore, closing the Link Summary also ends the Compile Summary for the build.

To close all files displayed in the Report Navigator select the **File→Close All Files** command. This returns Vitis analyzer to the home screen.

## Diff Two Text Files

The Vitis analyzer lets you compare two reports of the same type. This opens a report window similar to the one shown below.

*Figure 55: Compare Text Files*

```

File Compare: link.steps.log - dct.steps.log (on xsjrdv105)
link.steps.log                                     dct.steps.log
wrk... /workspace2/dct_project/Emulation-HW/binary_container_1.b...
^_ /proj/xbuilds/SWIP/2020.1_0331_2005/install... 111
XILINX_CD_CONNECT_ID=41572                         112 XILINX_CD_CONNECT_ID=41347
XILINX_CD_SESSION=95908fce-097c-4841-920e-a24b7e2b6f69 113 XILINX_CD_SESSION=c391222c-e395-48f7-acda-9bacfb7e5865
114
115
116 V++ command line:
117 -----
118 /proj/xbuilds/SWIP/2020.1_0331_2005/install... 118 /proj/xbuilds/SWIP/2020.1_0331_2005/install... 119
119 FINAL PROGRAM OPTIONS
120 --advanced.msc_solution_name=dct
121 --compile
122 --config common-config.cfg
123 --config binary_container_1-link.cfg
124 --config binary_container_1-dct-compile.cfg
125 --debug
126 --include ../src
127 --input_files .../src/dct_inline.cpp
128 --kernel dct
129 --log_dir binary_container_1.build/logs
130 --messageDb binary_container_1.mdb
131 --optimize 0
132 --output binary_container_1.xclbin
133 --platform /proj/xbuilds_released_2018/2018.3/AR/2018.3_0629_0404/intel
134 --remote_ip_cache /wrk/xsjhdmobup2/randyh/workspace2/ip_cache
135 --report_dir binary_container_1.build/reports
136 --report_level 0
137 --save-temps
138 --target hw_emu
139 --temp_dir binary_container_1.build
140 PARSED COMMAND LINE OPTIONS
141 --target hw_emu
142 --compile
143 -I../src
144 --config common-config.cfg
145 --config binary_container_1-dct-compile.cfg
146 -Obinary_container_1.build/dct.xo
147 ..//src/dct_inline.cpp
148 PARSED CONFIG FILE (1) OPTIONS
149 file: common-config.cfg
150 platform /proj/xbuilds_released_2018/2018.3/AR/2018.3_0629_0404/internal
151 save-temps 1
152
153 PARSED CONFIG FILE (2) OPTIONS
154 file: binary_container_1-link.cfg
155 debug 1
156 messageDb binary_container_1.mdb
157 temp_dir binary_container_1.build
158 < ----->
    
```

18 changes

Changed Inserted Deleted

The text-based comparison reports include:

- Kernel estimate
- System estimate
- Timing summary

- Log files

To compare reports, you must have two of the same type of reports listed in the Report Navigator window, or opened in the Reports view. Right-click a supported report in the Report Navigator and select the **Diff Against >** command. This command lets you specify another report of the same type to compare with the currently selected report.

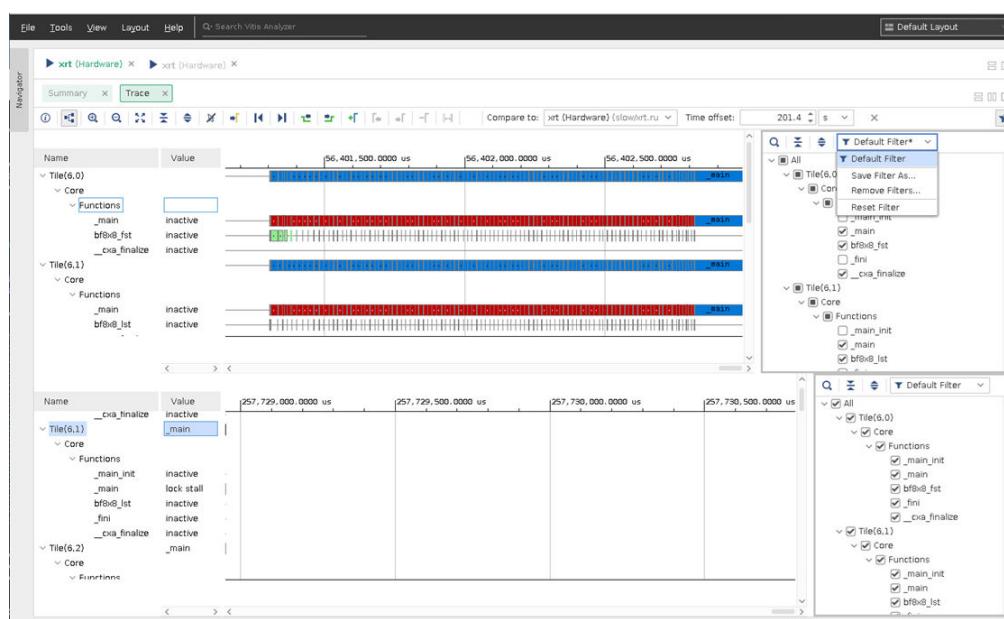
## Compare Two Reports

Vitis analyzer provides the capability of comparing two graphical reports when multiple reports are available, such as Timeline Trace reports, Waveform reports, or AI Engine Trace reports.

For example, in an open Timeline Trace report click the **Compare** link in the report toolbar menu to display a drop-down list of other Timeline Trace reports which can be compared with the current report. Vitis analyzer synchronizes the timelines with each other, as shown in the following figure, and when you scroll horizontally in one trace window both trace reports are scrolled together.



**TIP:** You can also right-click the **Timeline Trace** report listed in the Report Navigator pane on the left and select the **Compare to** command, which is enabled when there are multiple timelines to compare.



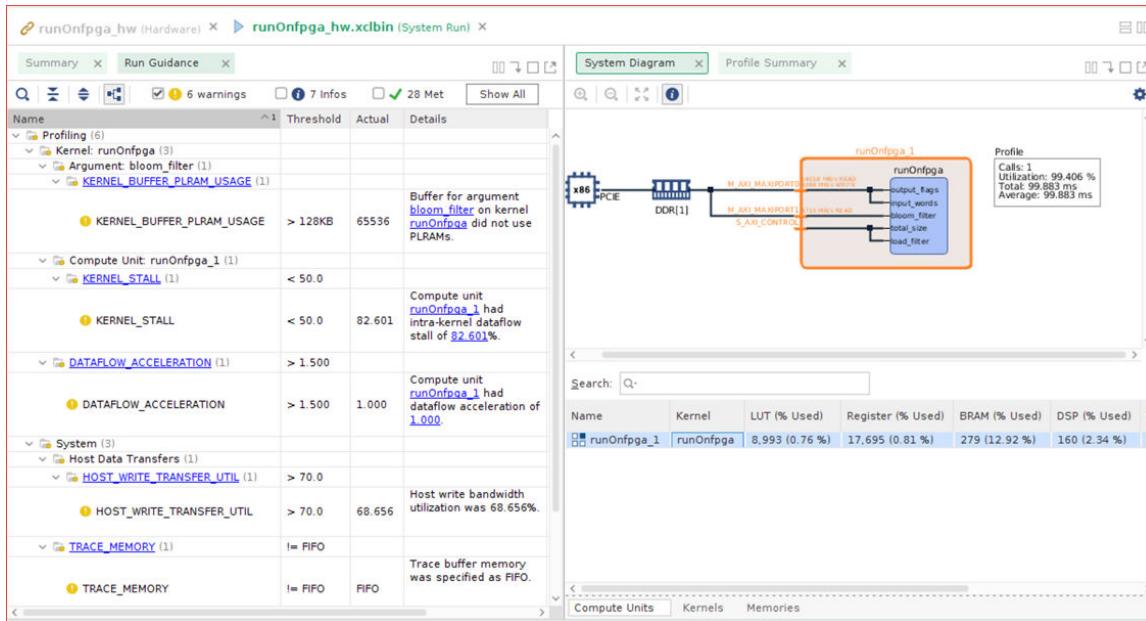
Events in the Timeline reports can be triggered at different times between the two different application runs. You can use the **Time offset** option report toolbar menu, as shown in the figure above, to synchronize the cursors in both windows so that a visual comparison can be carried out. When you zoom in or zoom out in the top trace window, the bottom window also zooms in or zooms out automatically.

# Cross-Probing between Reports

The Vitis analyzer supports a variety of selectable objects within different reports and views:

- **Compute Units (CU):** Selectable in the System Diagram and associated Compute Units table. Selecting the kernel selects associated compute units and vice versa. The CU is found in the Utilization report, the Profile Summary, the Timeline Trace, and the Waveform view.
- **CU ports:** Selectable in the System Diagram.
- **Kernels:** Selectable in the System Diagram and associated Kernels table. Note that selecting the kernel also selects the CUs and vice versa. Kernels are found in the Link Summary, Utilization report, System Guidance (under Accelerators), Profile Summary, and the Waveform view.
- **Kernel ports:** Selectable in the System Diagram.
- **Function arguments:** Selectable in the System Diagram and Kernels table.
- **AXI interconnects:** Selectable in the System Diagram. This selects all connections to a memory bank.
- **AXI ports:** Selectable in the System Diagram. These are "flattened", for example, they are the same for all kernels. Shown in the Profile Summary, and the Waveform view (data transfers).
- **Memory resources:** Selectable in the Platform and System Diagrams, and associated Memories table. Shown in the Profile Summary (data transfers: kernels to global memory).
- **Host CPU:** Selectable in the Platform and System Diagrams.

Figure 56: Cross-Probing Reports



The Vitis analyzer supports cross-probing between reports, such as within the System Diagram and from the Guidance View to other views. The Guidance view will provide an actionable resolution for a violation reported, and you can use cross-probing from the violation to quickly navigate to other reports and views.

Cross-probing can be bidirectional or unidirectional, depending on the report. The Guidance report lets you select objects in other reports, but does not support cross-probing from other reports or views.

- Bidirectional cross-probing between the System Diagram and Profile Summary report. Selecting a kernel, compute unit or compute unit port in one selects it in the other. Selecting a kernel also selects associated CUs in the reports.
- Unidirectional cross-probing from the Guidance to the System Diagram and Profile Summary report. The Details column of the guidance report displays hyperlinks that correspond to design objects such as kernels, CUs, etc.
  - Clicking a kernel, compute unit, or compute unit port hyperlink in Guidance selects it in the System Diagram and Profile Summary.
  - Clicking a memory or kernel argument hyperlink selects it in the System Diagram, but not the Profile Summary.
  - Clicking a kernel port hyperlink in Guidance selects a CU port in the System Diagram.
  - In some cases, the Details column displays a hyperlink for a value, for example, 82.601%.
    - Clicking a value hyperlink selects the corresponding design object and navigates to the associated section in the Profile Summary report.

- If the report is already open but is hidden behind another tab, it will be brought to the front.
  - If the report is not open, clicking a value hyperlink will open the report.
  - Guidance hyperlinks also have tooltips explaining what the click action does.
  - Additionally, selecting other objects such as the host, memories, AXI interconnects, and kernel arguments in the System Diagram does not cross-probe to the Profile Summary because the report does not represent these as selectable objects.
- 

## Using the Floating Ruler

The floating ruler assists with time measurements using a time base other than the absolute simulation time shown on the standard ruler at the top of the Wave window.

You can display (or hide) the floating ruler and drag it to change the vertical position in the Wave window. The time base (time 0) of the floating ruler is the secondary cursor, or, if there is no secondary cursor, the selected marker.

The floating ruler is visible only when the secondary cursor or a marker is present.

1. Do either of the following to display or hide a floating ruler:
  - Place the secondary cursor.
  - Select a marker.
2. In the Waveform Settings window, enable (check) the **Floating Ruler** option.

You only need to follow this procedure the first time. The floating ruler displays each time you place the secondary cursor or select a marker.

Uncheck/disable the **Floating Ruler** option to hide the floating ruler.

# Platform and System Diagrams

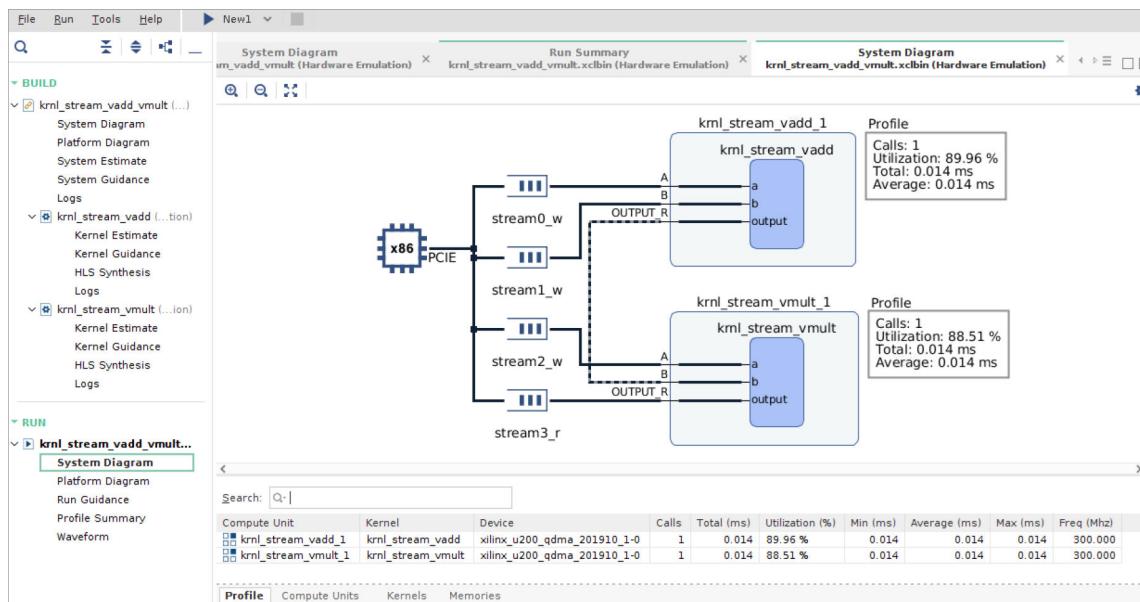
The Platform and System Diagrams display a representation of the platform resources, and the kernel code integrated onto the platform. They can be viewed in the Vitis™ analyzer from the Link Summary, the Run Summary, or the .xclbin for a project.

The Platform Diagram is a block diagram of the target platform, before the .xclbin is loaded. This diagram shows all DDR banks and PLRAM available, and their available connections. A table at the bottom displays details of bank names with types of memories, their sizes and which SLR region these are available.

The System Diagram shows memory banks or PLRAMs used by the .xclbin. You can also see how the function arguments of Compute Units are connected to AXI4 interfaces. A table at the bottom of the System Diagram displays information for each Compute Unit, Kernels, and Memories. The System Diagram also contains a table of clocks including default, requested and achieved frequencies. For designs that include AI Engine kernels, the System Diagram also displays information related to those kernels. Features of the System Diagram include the following:

- Name of the kernel with an indication which SLR this is available.
- LUT%
- Register %
- BRAM % used
- URAM % used
- DSP % used

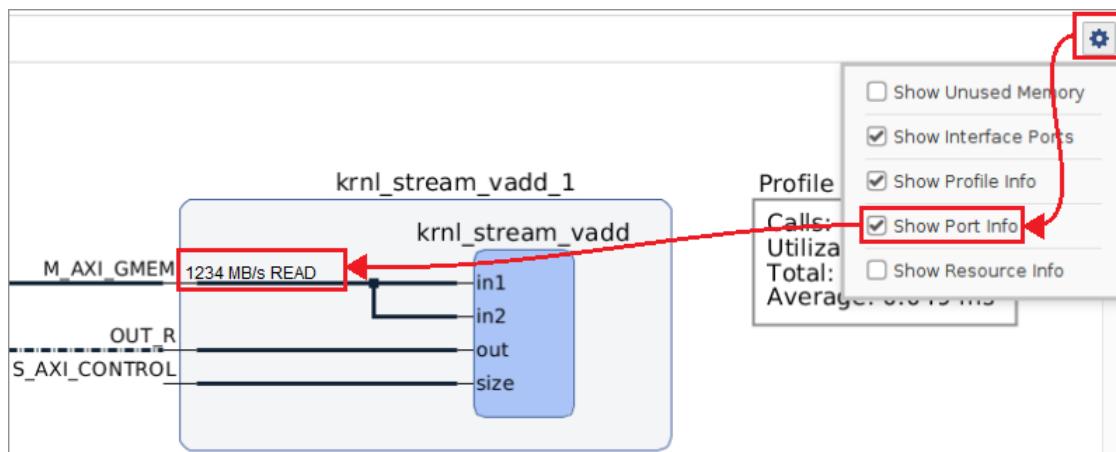
Figure 57: System Diagram with Profile Data



When Run Summary is loaded, the System Diagram includes profile data from the run. The Vitis analyzer automatically runs `vp_analyze` when the generated `xrt.run_summary` file has any of the profiling files available. The profile data is added to the table at the bottom of the System Diagram and can also be displayed in the diagram, as shown in the figure above.

The resource information from the table can also be displayed in a box next to each kernel or CU in the System Diagram. The **Settings** command ( ) lets you display or hide Unused Memory, Interface Ports, Profile Info, and Resource info.

Figure 58: Show Port Info



The ports on a Compute Unit can display the transfer rates on the system diagram, as well as CU Utilization percentage. CU port transfer rates are taken from the Kernel Transfer section of the Profile Summary report. CU utilization statistics are taken from the Compute Unit Utilization section of Profile Summary. The performance data is available as long as Profiling was enabled for Hardware and Hardware Emulation run, using the Vitis compiler `--profile` option as described in [--profile Options](#).

## Device Map

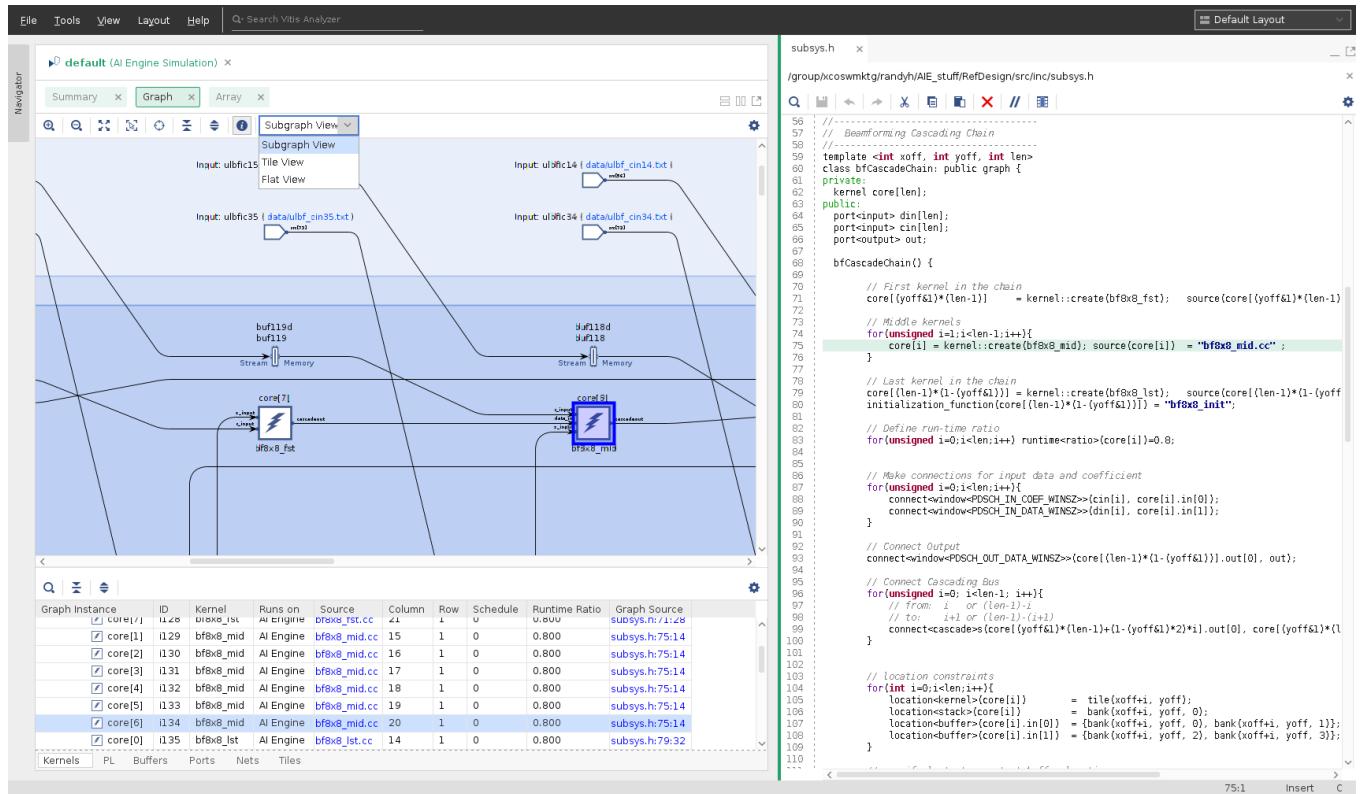
Vitis™ analyzer also provides a Device Map with the Link Summary for a project which can be opened from the Report Navigator pane. The Device Map shows an abstract view of the static and dynamic regions of the target platform, and shows the placement of kernels on the device and the SLRs. You can also see where each Compute Unit is placed inside the dynamic region.

The table at the bottom of the Device Map shows information similar to the table in System Diagram. You can select the rows in this table and the corresponding section is highlighted in the Device Map. The highlight color for any of the objects in the table can be changed by right-clicking the object in the table and selecting the **Highlight** color. This updated color is used to highlight the selected object on the Device Map. Additionally, the Device Map and System Diagram object support highlighting across multiple views. For example, you can also select the object in the System Diagram or Device Map table, both views highlight the object selected from the table.

# AI Engine Graphs and Arrays

The AI Engine Graph and Array diagrams provide a quick overview of the structure of the ADF graph application implemented in the AI Engine tiles.

*Figure 59: AI Engine Graph Application*

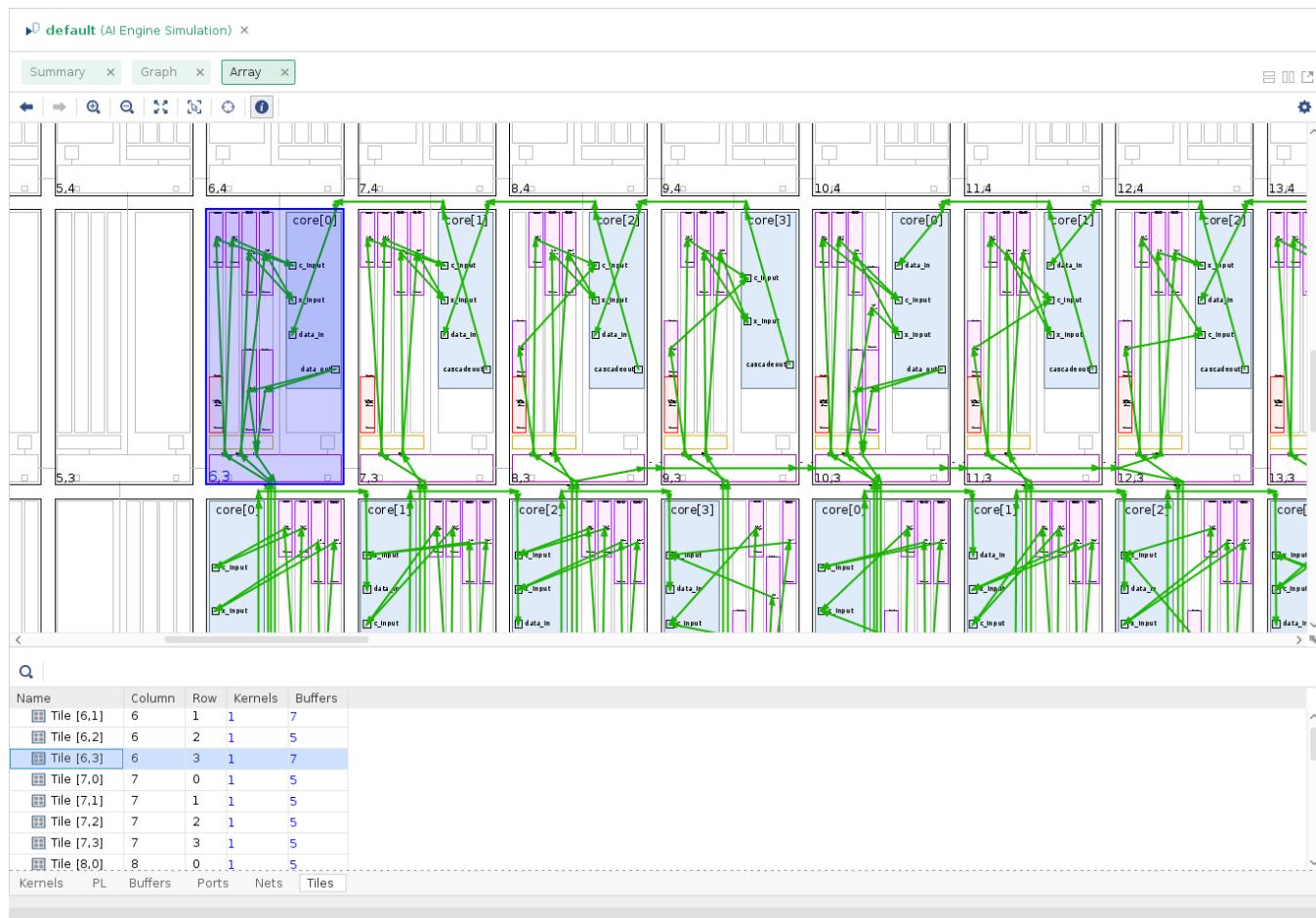


The AI Engine Graph view shows the connectivity of the ADF graph as seen by the AI Engine compiler. The canvas shows nodes representing kernels, kernel arguments, memory buffers, and primary inputs and outputs, with edges drawn to represent the connectivity between these elements.

To the right of the canvas, a Settings panel that can be used to customize the view of the graph, by letting you show or hide elements of the displayed graph.

Below the graphical view is a set of tables containing all of the objects drawn in the graphical view: kernels, connections, paths, memories, etc. Selecting an element in one of these tables will cross-probe to the graphical view and vice versa.

**Figure 60: AI Engine Array Diagram**



The Array view shows how the ADF graph is spatially placed on the AI Engine tile array. The array canvas contains all the core and memory components of the array. Kernels are drawn in the core where they are programmed, and connectivity between cores and/or memory are shown with connectivity lines. There is an abstract representation of the PL area as needed to model PL cores and interfaces to the programmable logic region.

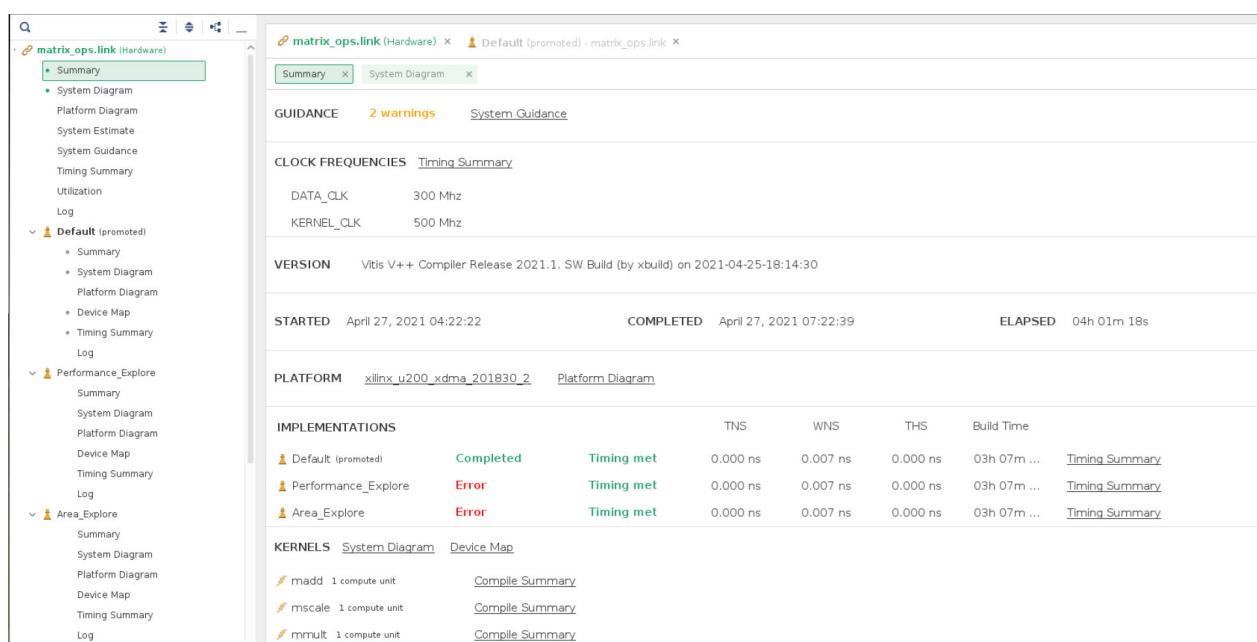
To the right of the canvas, a Settings panel that can be used to customize the view of the array, by letting you show or hide elements of the resources and/or tile array.

# Link Summary: Multiple Strategies and Timing Reports

As discussed in [Running Multiple Implementation Strategies for Timing Closure](#), the `v++` linking step can run multiple iterations of Vivado implementation using different strategies to achieve timing closure. Vitis analyzer can visualize and compare these results from multiple implementation attempts. As explained, the Vitis compiler automatically picks the first completed run that meets timing to proceed with the build and generate the device binary (`xclbin`). This includes logging the report files for that one run only. However, you can have the Vitis compiler wait until all of the implementation runs are complete before proceeding to generate an `xclbin` file. In this case, results for each implementation run are available in the `link_summary` for your review and comparison in Vitis analyzer.

Link Summaries with multiple implementation strategies display each strategy in the Report Navigator pane, as shown in the figure below. Each of these strategies include a Timing Summary report. Even though the `xclbin` is generated for only one strategy, the Link Summary reports are available for all implementation runs to extract useful resource and timing information.

Figure 61: Link Summary



The Timing Summary report in Vitis analyzer is a simplified version of the complete Timing Summary report generated by Vivado place and route. The simplified report only displays the worst setup, hold, and pulse width paths for the design. You can view this report without needing to open a Vivado project or netlist, so you can quickly navigate to failing timing paths. The details are displayed in the report viewer as shown below. The Timing Summary is displayed, as well as detailed path reports for the worst Setup, Hold, and Pulse Width observed in the design. To see the complete timing report, click the **T** toolbar button, which displays the text version of the report.

Figure 62: Timing Summary

The screenshot shows the Vitis Analyzer interface with the 'matrix\_ops.link (Hardware)' project selected. The 'Timing Summary' tab is active, indicated by a green border. The report displays the 'Design Timing Summary' for the 'matrix\_ops.link' hardware. It includes sections for 'Setup', 'Hold', and 'Pulse Width' with their respective values and counts. A note at the bottom states 'All user specified timing constraints are met.'

Design Timing Summary		
Setup	Hold	Pulse Width
Worst Negative Slack (WNS): <a href="#">0.007 ns</a>	Worst Hold Slack (WHS): <a href="#">0.010 ns</a>	Worst Pulse Width Slack (WPWS): <a href="#">0.000 ns</a>
Total Negative Slack (TNS): 0.000 ns	Total Hold Slack (THS): 0.000 ns	Total Pulse Width Negative Slack (TPWS): 0.000 ns
Number of Failing Endpoints: 0	Number of Failing Endpoints: 0	Number of Failing Endpoints: 0
Total Number of Endpoints: 625158	Total Number of Endpoints: 621165	Total Number of Endpoints: 257581

# Setting Guidance Thresholds

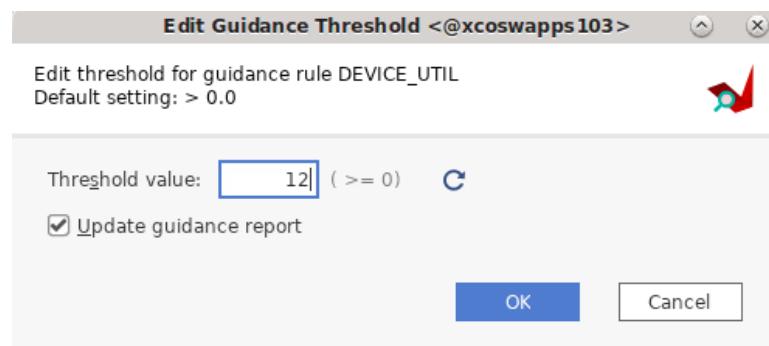
Guidance messages reported in the Run Guidance report are triggered by specific rules and value thresholds that are defined within the tool. Some of these rules and value thresholds are user-modifiable within Vitis™ analyzer. When the Run Guidance report is opened, the modifiable values will appear as links in the Threshold column of the report, as shown in the following figure.

*Figure 63: Run Guidance Threshold*

Name	Threshold	Actual	Details
Profiling (4)			
Kernel: SgdLR_1 (2)			
Compute Unit: SgdLR_1 (1)			
UNUSED_CUS	> 0	0	Compute unit <a href="#">SgdLR_1</a> on device xilinx_u280_xdma_201920_3-0 was used <a href="#">0</a> time(s).
KERNEL_PORT_DATA_WIDTH	= 512	32	Port <a href="#">m_axi_gmem1</a> has a data width of 32.
System (2)			
Device: xilinx_u280_xdma_201920_3-0 (1)			
DEVICE_UTIL	> <a href="#">0.000</a>	0.000	Device xilinx_u280_xdma_201920_3-0 was utilized for 0.000 msec.
Host Data Transfers (1)			
HOST_MIGRATE_MEM	> <a href="#">0</a>	0	Migrate Memory OpenCL APIs were used <a href="#">0</a> time(s).

Clicking a link in the Threshold column will display the Edit Guidance Threshold dialog box, as shown below. The top part of the dialog box displays the current operator and threshold for the selected rule, and the lower part lets you edit the value.

*Figure 64: Edit Guidance Threshold*



The figure above shows the Threshold value being redefined with a user-specified value. In some cases, a range of accepted values is displayed and the tool checks that the provided value falls within that range. In the example above, the specified value must be greater than or equal to 0.

The Edit Guidance Threshold dialog box also provides a **Reset threshold** command that allows you to reset the user-defined value to the hard-coded value provided by the tool. This is the  seen in the figure above.



**TIP:** The menu bar for the Run Guidance report also has a **Reset Guidance Thresholds** command to let you clear all user-modified thresholds from the report.

The **Update guidance report** check box lets you specify if the guidance report will be rerun after the value is updated. If you deselect this check box, the threshold value will be changed as specified, and the report will be marked out-of-date due to the new user-specified threshold. You will need to manually **Reload** the report to see the impact of the user values.

Click **OK** to change the value, or **Cancel** to close the dialog box without change.

## Import/Export Rule Thresholds

The rules as to which threshold values can be modified, and the default values of all rules, are stored within the tool installation. You can override these rules with a custom rule file, which you can export and import into different projects in Vitis analyzer.

When you have customized one or more rules in the Run Guidance report, an **Export User Guidance Thresholds** command becomes active in the toolbar menu of the report. You can see this command displayed in the top figure above. This lets you export the customized value thresholds to reuse them in other designs.

A sample file is shown below:

```
profile_rules =
(
{
    id = "HOST_MIGRATE_MEM";
    value = "1";
},
{
    id = "DEVICE_UTIL";
    value = "3";
}
);
version = "1";
```

The user threshold file does not need to contain all guidance rules, only the rules where the value has changed, and only the rule id and value fields are needed:

- `id` is the name of the rule as displayed in the **Name** column of the Run Guidance report.
- `value` is the user-specified value, displayed in the **Threshold** column.

The Vitis analyzer IDE also lets you **Import User Guidance Thresholds** to reuse the custom thresholds in other projects. You can export a `user-thresholds.cfg` file, edit the values, and import the file as needed. Importing a user-thresholds file regenerates the Run Guidance report to use the imported values.



**IMPORTANT!** *Importing a user-guidance threshold file will overwrite any existing user-modified thresholds.*

# Creating an Archive File

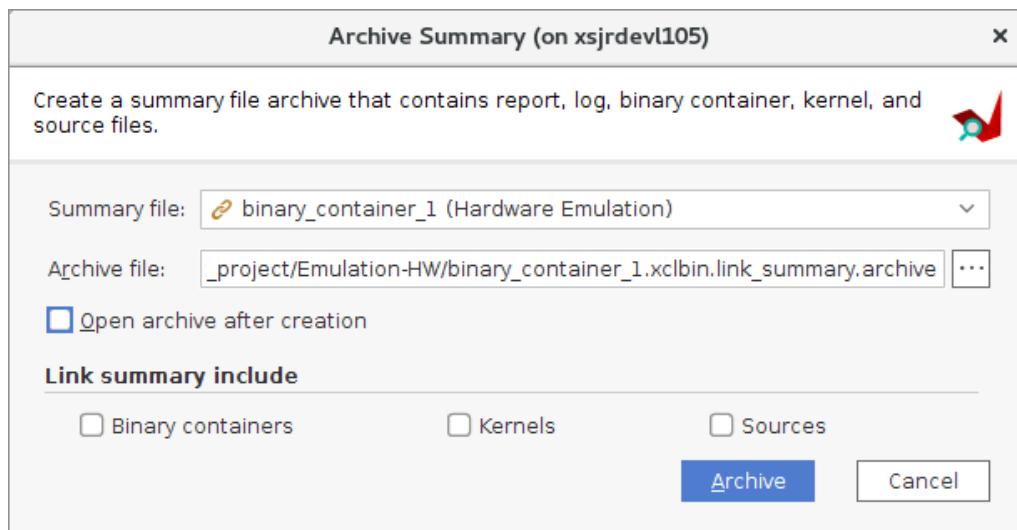
At times while developing your project, you might introduce changes in your host or kernel code that completely alters the contents or quality of the various build and run summaries produced by the tool. Your existing reports and analysis data are overwritten unless you manually save the relevant files. The Archive Summary command lets you save all relevant files with an open Link Summary or Run Summary.



**TIP:** This feature lets you quickly share design reports with other team members by sharing the archive summary.

Select **File→Archive Summary** menu command, or right-click a summary in the Report Navigator and select **Archive Summary**. This opens the Archive Summary dialog box as shown below.

*Figure 65: Archive Summary Dialog Box*



Archive file names must have the extension of `link_summary.archive`, or `run_summary.archive` to be recognized by the Vitis analyzer. The contents of the archive depend on the summary report being archived.

The contents of the Link Summary include these file extensions:

- **Binary container(s):** `.xclbin`
- **Kernel(s):** `.xo`

- **System/platform diagram:** .json
- **System estimate:** .xtxt
- **Guidance:** .html, .pb
- **Timing summary:** .rpt, .rpv
- **Utilization(s):** .xutil

The contents of the Run Summary include these file extensions:

- **System diagram:** .run\_summary
- **Platform diagram:** .run\_summary
- **Guidance:** Not required, generated using vp\_analyze and timeline
- **Profile summary:** .csv
- **Timeline Trace:** .csv
- **Waveform report:** .wdb



**IMPORTANT!** For AI Engine designs, when a *run\_summary* contains a trace report but does not have an entry for the AI Engine work directory, Vitis analyzer cannot run *hwanalyze* to display the trace report. In this case, the tool prompts you for the location of the AI Engine *compile\_summary*, and updates the *run\_summary* file on disk. If you archive the *run\_summary* before providing this location, you cannot view the trace report from the archived *run\_summary*.

You can also choose to save the .xclbin file extension, any compiled kernel object file extensions (.o and .xo), and the original source file extensions (.cpp, .c, and .cl) used to generate the summary reports.



**TIP:** Guidance is not saved because this is dynamically generated by Vitis™ analyzer from *opencl\_summary.csv*, and optionally *profile\_kernels.csv*.

To open an existing archive file, use the **File → Open Summary** command and browse for the archive file. You can also open an archive file when launching Vitis analyzer:

```
vitis_analyzer design.archive
```

There is also a command line form of Archive Summary that you learn more about with:

```
archive_summary -help
```

**Note:** The Vitis analyzer archive is a compressed archive that can be uncompressed to access the individual, including the xclbin and xo file when they are included in the archive.

# Using the Vitis IDE

This section contains the following chapters:

- [Launching Vitis IDE](#)
- [Creating a Vitis IDE Project](#)
- [Building the System](#)
- [Vitis IDE Debug Flow](#)
- [Configuring the Vitis IDE](#)
- [Project Export and Import](#)
- [Getting Started with Examples](#)
- [RTL Kernel Wizard](#)

# Launching Vitis IDE

In the Vitis™ integrated design environment (IDE), you can create a new application project or platform development project.

The `vitis` command launches the Vitis IDE with your defined options. It provides options for specifying the workspace and options of the project. The following sections describe the `vitis` command options.

## Display Options

The following options display the specified information intended for review.

- `-help`: Displays help information for the Vitis core development kit command options.
- `-debug`: Launches the Vitis IDE to run debug on a command-line project.



---

**TIP:** To view the help for the `vitis -debug` command, use `-debug -help`.

---

- `-version`: Displays the Vitis core development kit release version.

## Command Options

The following command options specify how the `vitis` command is configured for the current workspace and project.

- `-workspace <workspace location>`: Specify the workspace directory for Vitis IDE projects.
- `{-lp <repository_path>}`: Add `<repository_path>` to the list of Driver/OS/Library search directories.
- `-eclipseargs <eclipse arguments>`: Eclipse-specific arguments are passed to Eclipse.
- `-vmargs <java vm arguments>`: Additional arguments to be passed to Java VM.

# Creating a Vitis IDE Project

In the Vitis™ IDE, you can create a new application project, or platform development project. The following section shows you how to set up a workspace, create a new Vitis IDE project, and use key features of the IDE.

## Launch a Vitis IDE Workspace

1. Launch the Vitis IDE directly from the following command line.

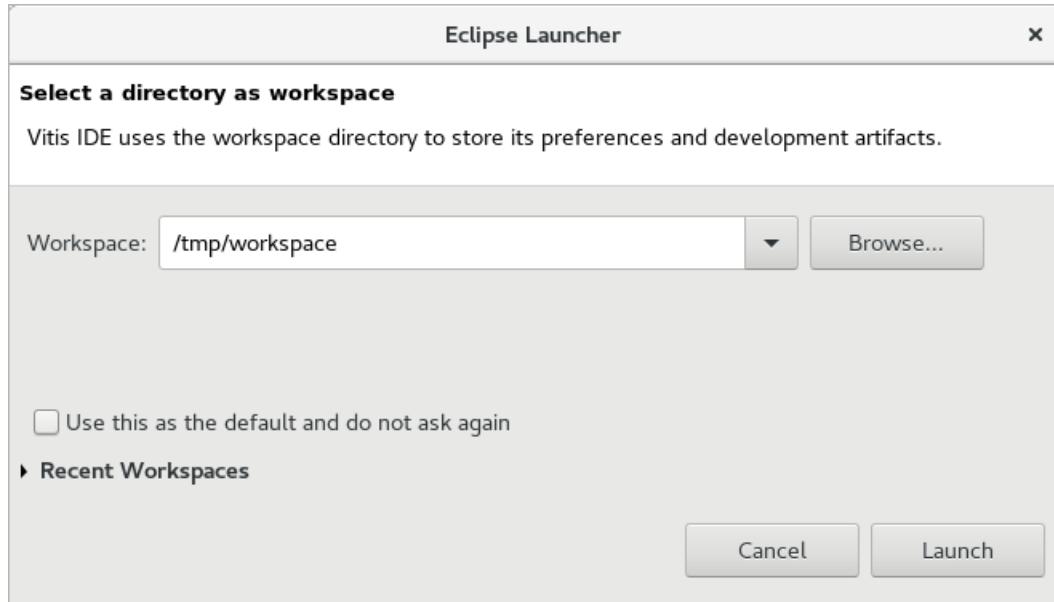
```
$vitis
```



**IMPORTANT!** When opening a new target platform, to enter a Vitis core development kit command, ensure that you set it up as described in [Setting Up the Vitis Environment](#).

The Vitis IDE opens.

2. Select a workspace as shown in the following figure.



The workspace is the folder that stores your projects, source files, and results while working in the IDE. You can define separate workspaces for each project, or have a single workspace with multiple projects and types. The following instructions show you how to define a workspace for a Vitis IDE project.

3. Click **Browse** to navigate to and specify the workspace, or type the appropriate path in the **Workspace** field.
4. Optionally enable **Use this as the default and do not ask again** to set the specified workspace as your default choice and eliminate this dialog box in subsequent uses of the IDE.  
**Note:** To restore the dialog box, navigate to **Window → Preference → Additional → General → Startup and Shutdown → Workspaces**, and select **Prompt for workspace on startup**.
5. Click **Launch**.



**TIP:** To change the current workspace from within the Vitis IDE, select **File → Switch Workspace**.

You have now created a workspace and can populate it with projects.

---

## Create an Application Project

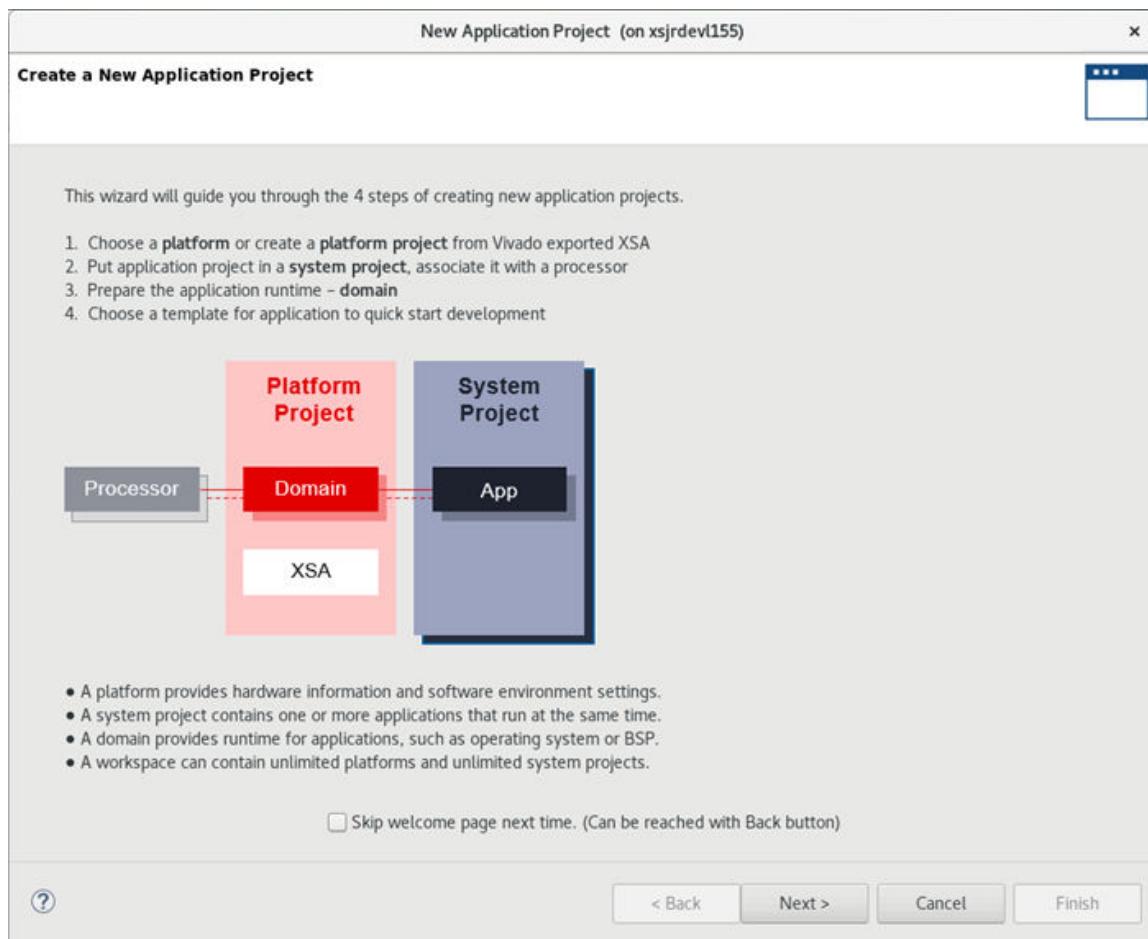


**TIP:** Example designs are provided with the Vitis core development kit installation and also on the Xilinx [Vitis Examples](#) GitHub repository. For more information, see [Getting Started with Examples](#).

---

After launching the Vitis IDE, you can create a new Application Project.

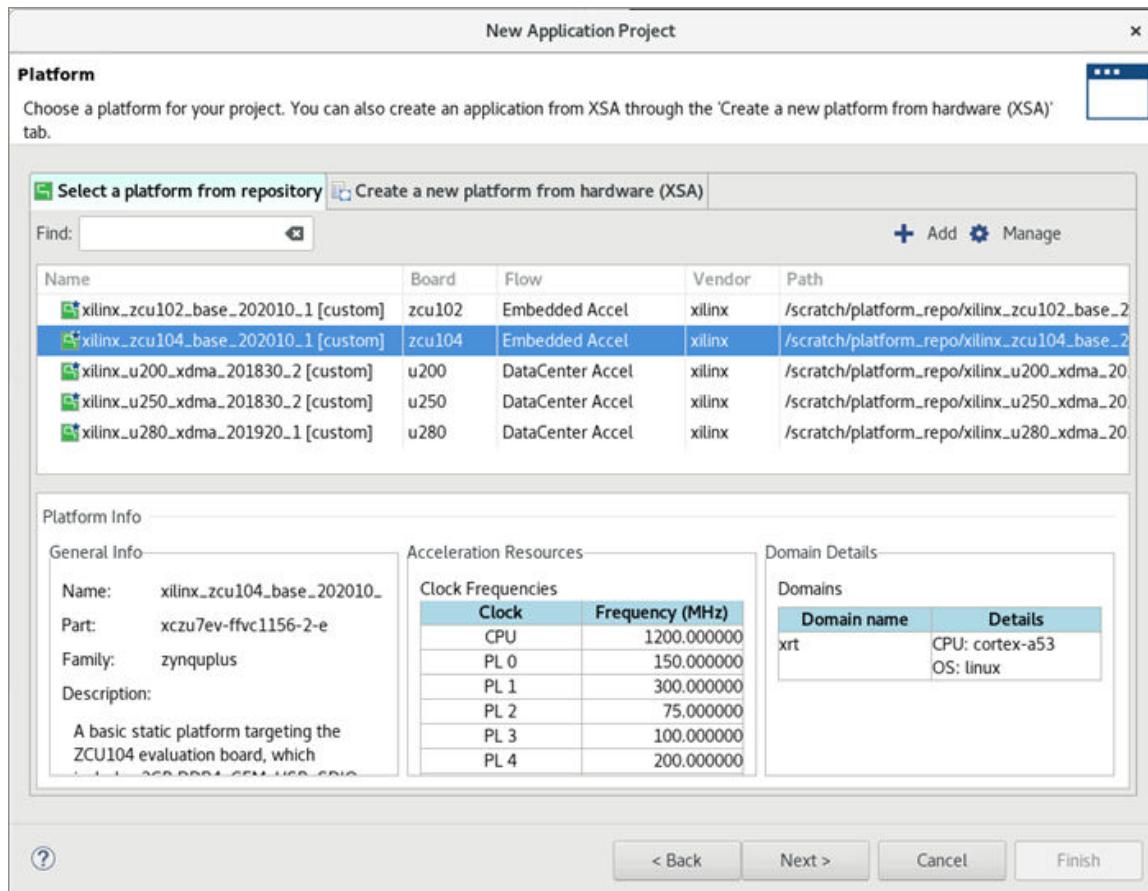
1. Select **File → New → Vitis Application Project**, or if this is the first time the Vitis IDE has been launched, you can select **Create Application Project** on the Welcome screen.



The New Application Project wizard opens displaying a Welcome page that explains the process for new users. You can disable this from being shown again by enabling **Skip welcome page next time**.

2. Click **Next** to open the Platform page of the New Application Project wizard to specify a target platform.

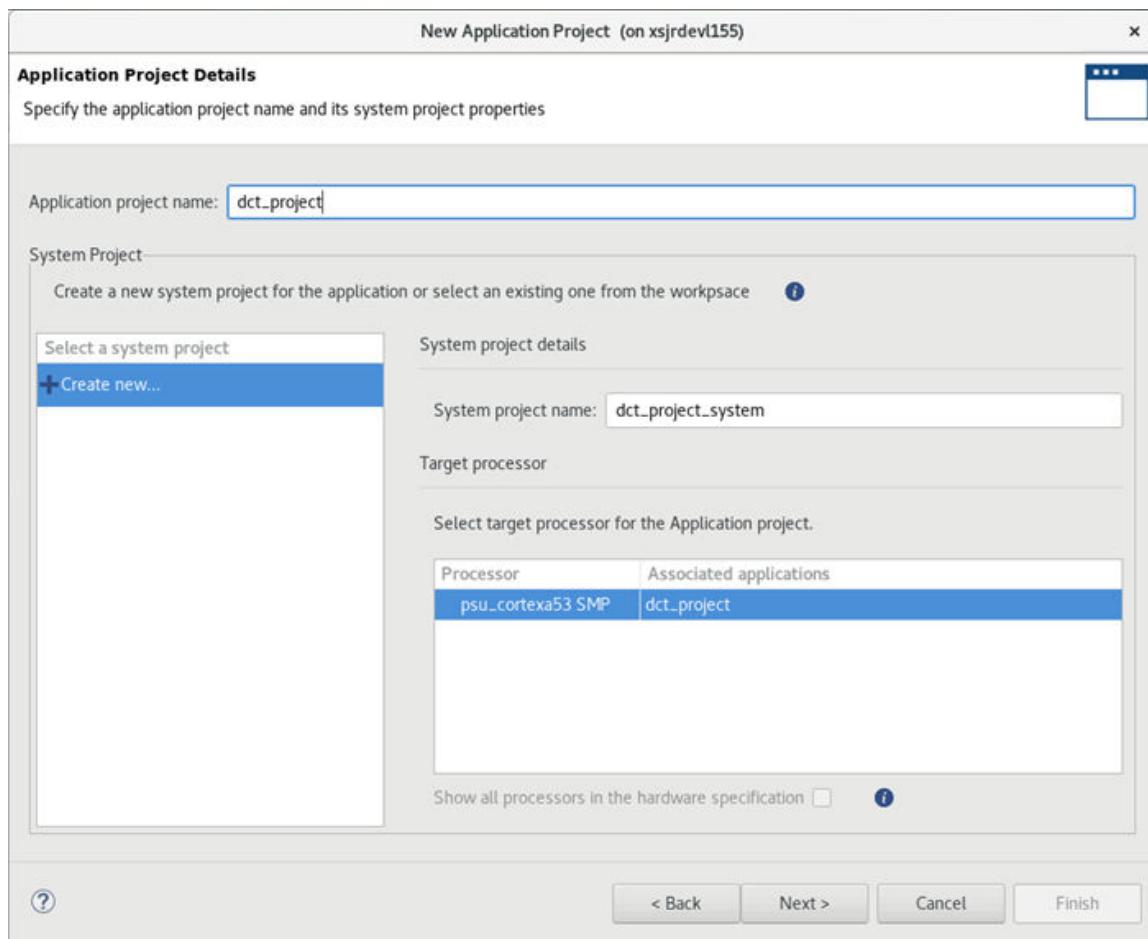
A target platform is composed of a base hardware design and the meta-data used in attaching accelerators to declared interfaces. Use the Select a platform from repository tab to choose a platform for your project. You can enter a value in the Find field to limit the choices displayed to make it easier to locate the required platform. The bottom portion displays information related to the currently selected platform, as shown in the following figure.



**Note:** For platforms supported by a specific release refer to the Release Notes in the [Section I: Getting Started with Vitis](#).

You can also add custom defined or third-party platforms into a repository. For more information, see [Managing Platforms and Platform Repositories](#).

- In the Application Project Details page, specify the name in the Application project name field, as shown in the following figure.

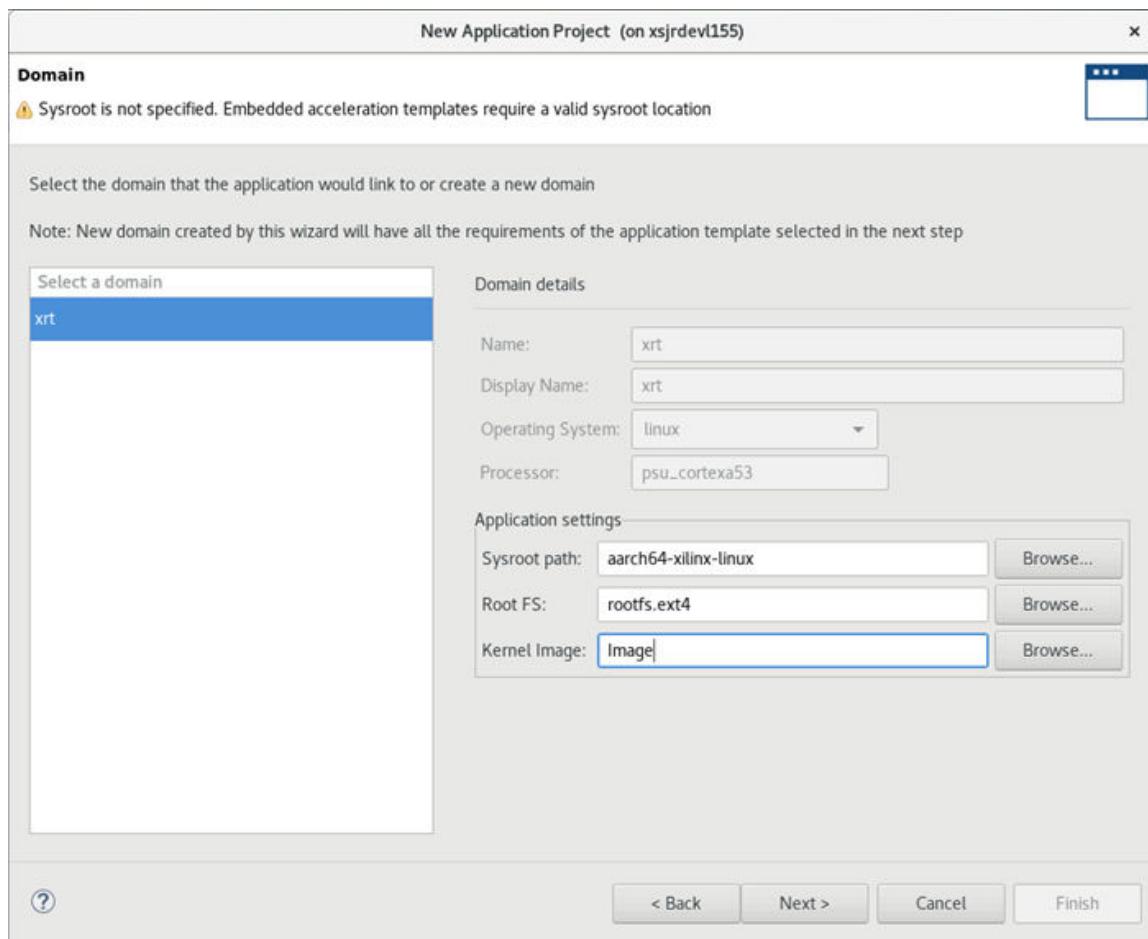


By default, the tool creates a new system project for your application project. However, you can also add your application project to an existing system project, if one exists. The system project is a top-level manager for different projects that combine to create the system view.

4. Click **Next** to proceed.

**Note:** If you selected Data Center accelerator card as your project platform in Step 2, the following page is not displayed and you can skip to Step 6.

5. If you select an **Embedded Acceleration** target platform on the Platform page, as displayed in the Flow column, the Domain page opens next as shown in the following figure.

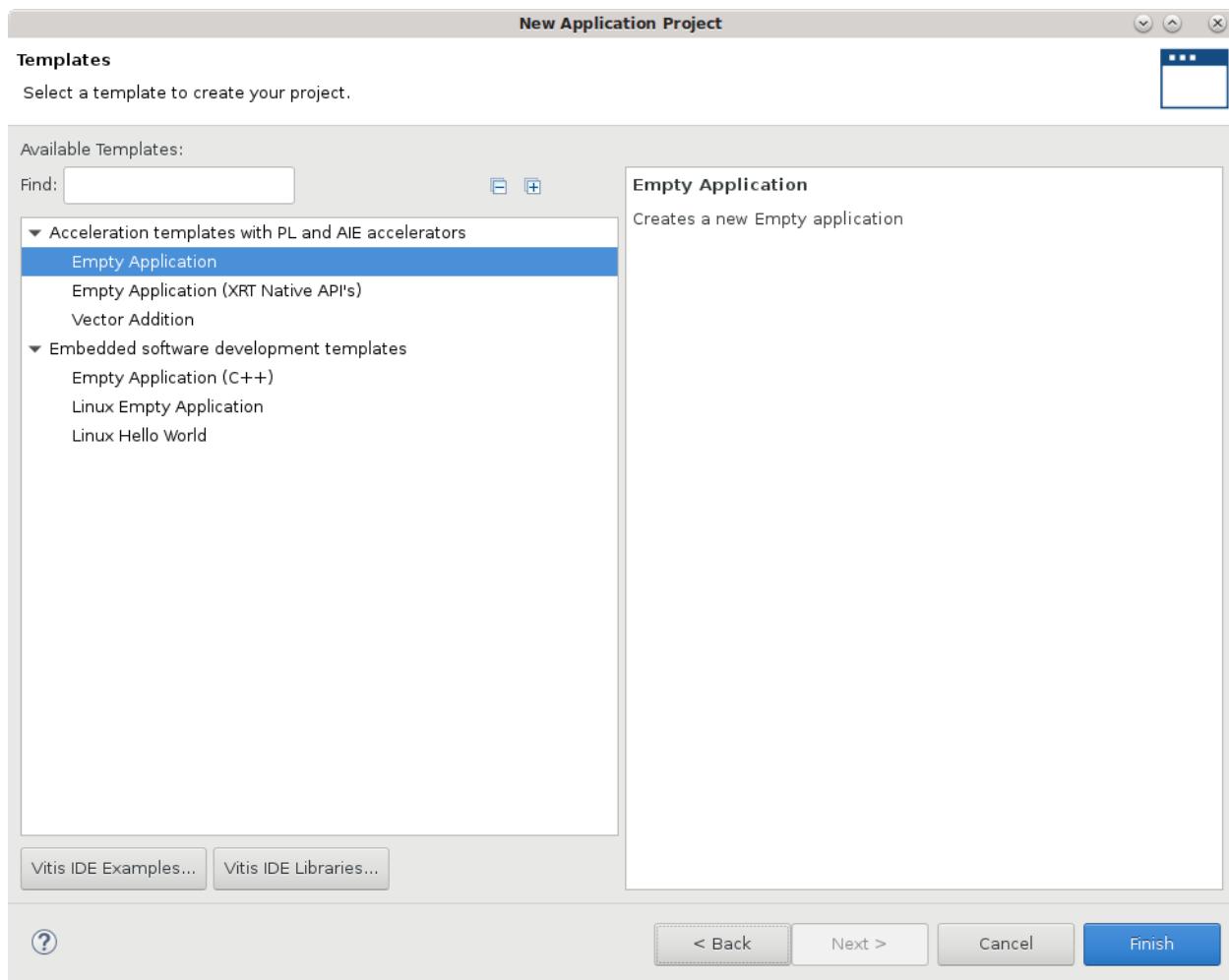


Select a **Domain** from the list of existing domains on the platform, and Domain details are populated from your selection. The Domain defines the processor and operating used for running the host program on the target platform. You must also set the following Application Settings for the project to build correctly on the embedded platform:

- **Sysroot path:** The `sysroot` is part of the platform where the basic system root file structure is defined. The Sysroot path lets you define a new `sysroot` for your application.
- **Root FS:** Specify the location of the root file system.
- **Kernel Image:** Specify the location of the operating system kernel.

These options can be changed after the project is created from the **System Project Settings** in the Project Editor window.

6. Click **Next** to open the Templates page letting you select an application acceleration template for your new project.



Select an **Empty Application (XRT Native API)** to create a blank project to import XRT API source files into as described in [Writing the Software Application](#), and build the project from scratch. Also, use one of the provided template projects as a foundation for your new application project to help start your project, or help you learn the tool.



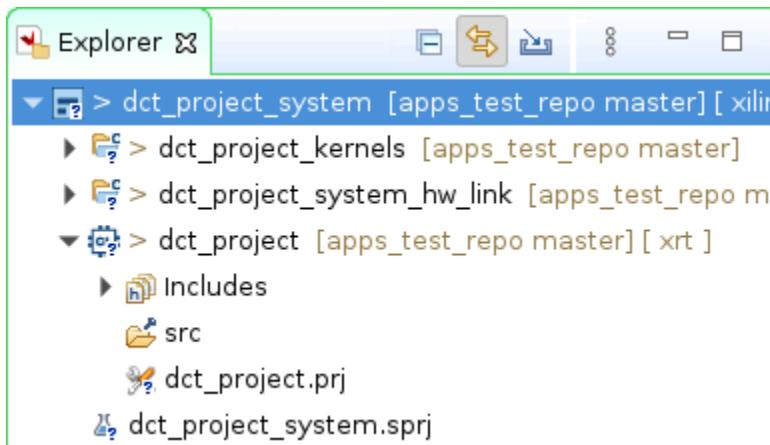
**TIP:** Click the *Vitis IDE Examples* button, or the *Vitis IDE Libraries* button to install additional examples as discussed in [Getting Started with Examples](#).

7. Click **Finish** to close the New Application Project wizard and open the project in the IDE.



**TIP:** The Vitis IDE opens in the Design perspective as described in [Understanding the Vitis IDE](#). Review this information if you are unfamiliar with the display.

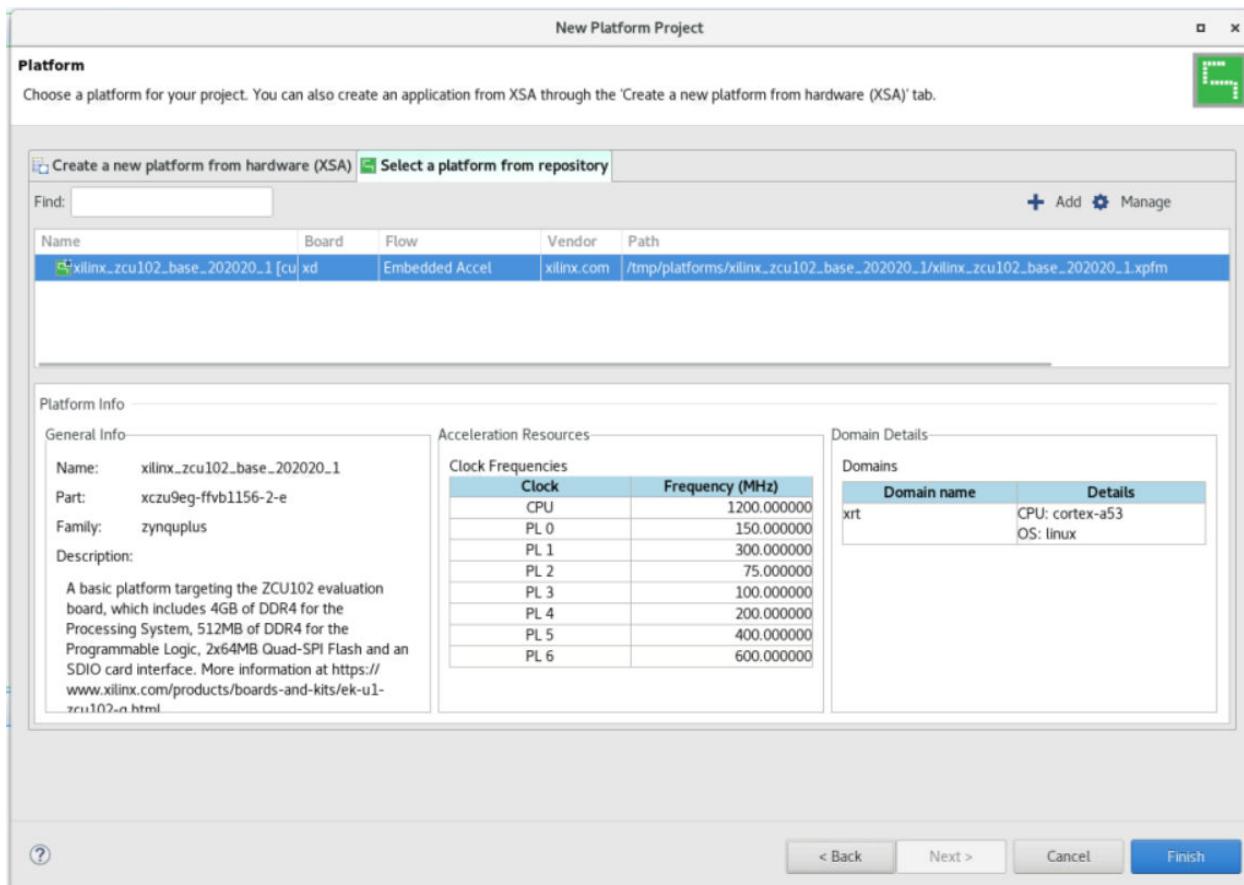
When a new application acceleration project is created in the Vitis IDE, it includes a top-level system project, and nested within an application project for the host-code, a hardware kernels project for compiling kernel objects, and a `hw_link` project that is used for linking hardware kernels to the target platform and to each other. These projects are displayed in the Explorer view as shown in the following figure.



## Managing Platforms and Platform Repositories

You can manage the platforms that are available for use in Vitis IDE projects, from **Xilinx→Add Custom Platform** in the main menu of an open project, or from the Platform page present on both New Application and New Platform wizards.

*Figure 66: New Platform Project*



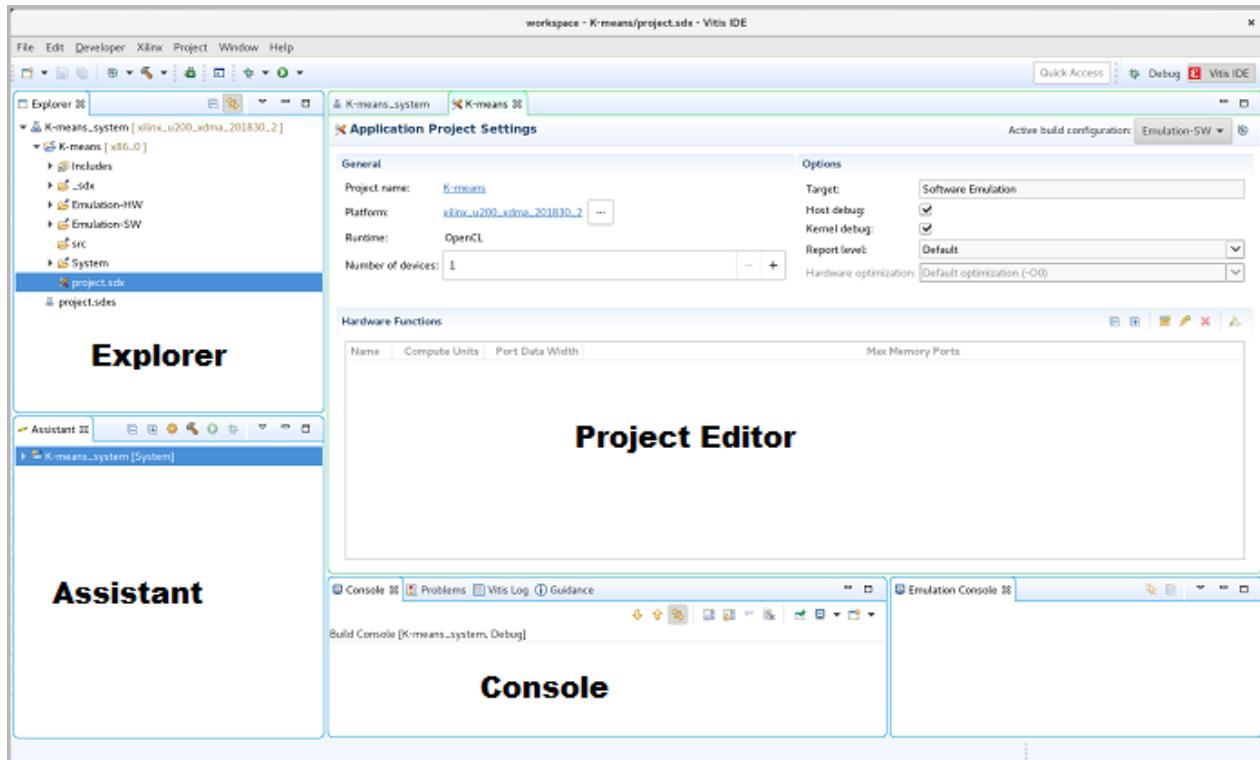
From the Platform page, manage the available platforms and platform repositories using one of the following options:

- **Add** (+): Add your own platform to the list of available platforms. To add a new platform, navigate to the top-level directory of the custom platform, select it, and click **OK**. The custom platform is immediately available for selection from the list of available platforms.
- **Manage** (⚙️): Add or remove standard and custom platforms. If a custom platform is added, the path to the new platform is automatically added to the repositories. When a platform is removed from the list of repositories, it no longer displays in the list of available platforms.

## Understanding the Vitis IDE

When you open a project in the Vitis IDE, the workspace is arranged in a series of different views and editors, also known as a *perspective* in the Eclipse-based IDE. The tool opens with the Design perspective shown in the following figure.

*Figure 67: Vitis IDE – Default Perspective*



Some key views and editors in the default perspective include:

- **Explorer view:** Displays a file-oriented tree view of the project folders and their associated source files, plus the build files, and reports generated by the tool. You can use this to explore your project file hierarchy.
- **Assistant view:** Provides a central location to view and manage the projects of the workspace, and the build and run configurations of the project. You can interact with the various project settings and reports of the different configurations. From this view, you can build and run your Vitis IDE application projects, and launch the Vitis analyzer to view reports and performance data as explained in [Section VIII: Using the Vitis Analyzer](#).
- **Project Editor view:** Displays the current project, the target platform, the active build configuration, and specified hardware functions; allows you to directly edit project settings.
- **Console view:** Presents multiple views including the command console, design guidance, project properties, logs, and terminal views.

The Vitis IDE includes several predefined perspectives, such as the Vitis IDE perspective, the Debug perspective, and the Performance Analysis perspective. To quickly switch between perspectives, click the perspective name in the upper right of the Vitis IDE.

You can arrange views to suit your needs by dragging and dropping them into new locations in the IDE, and the arrangement of views is saved in the current perspective. You can close windows by selecting the **Close (X)** button on the View tab. You can open new windows by using the **Window → Show View** command and selecting a specific view.

To restore a perspective to the default arrangement of views, make the perspective active and select **Window → Reset Perspective**.

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

---

## Adding Sources

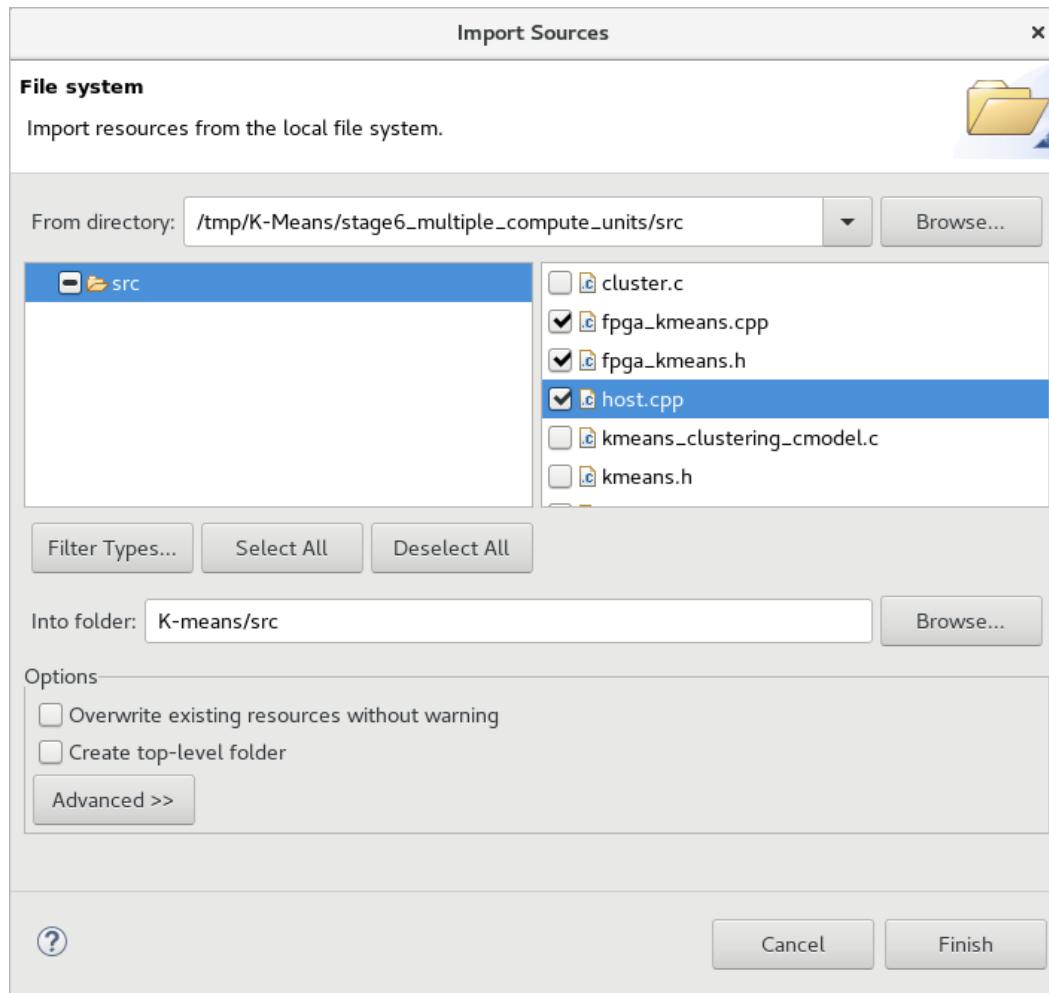
A project consists of many different source files, including C/C++ files and headers for the [Writing the Software Application](#), [Developing PL Kernels using C++](#), compiled Xilinx object (XO) files containing RTL kernels as discussed in [Packaging RTL Kernels](#), or HLS kernels. You can add these source files as needed to support your application.

Each individual project within a system project requires its own source and data files. The host application code (`host.cpp`) is added to the `./src` folder in the processor application project, and the kernel code (`kernel.cpp`), or the compiled `kernel.xo` files are added to the `./src` folder of the kernel application project. You can add these files using the **Import Sources** command as explained in the next section.

## Add Source Files

- With the project open in the Vitis IDE, to add source files, right-click the `src` folder in the Project Explorer, and click **Import Sources**.

This displays the Import Sources dialog box shown in the following figure.



- In the dialog box, for the From directory field, click the **Browse** button to select the directory from which you will import sources.
- In the Into folder field, make sure the folder specified is the `src` folder of the project.
- Select the desired source files by enabling the check box next to the file name, and click **Finish**.



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

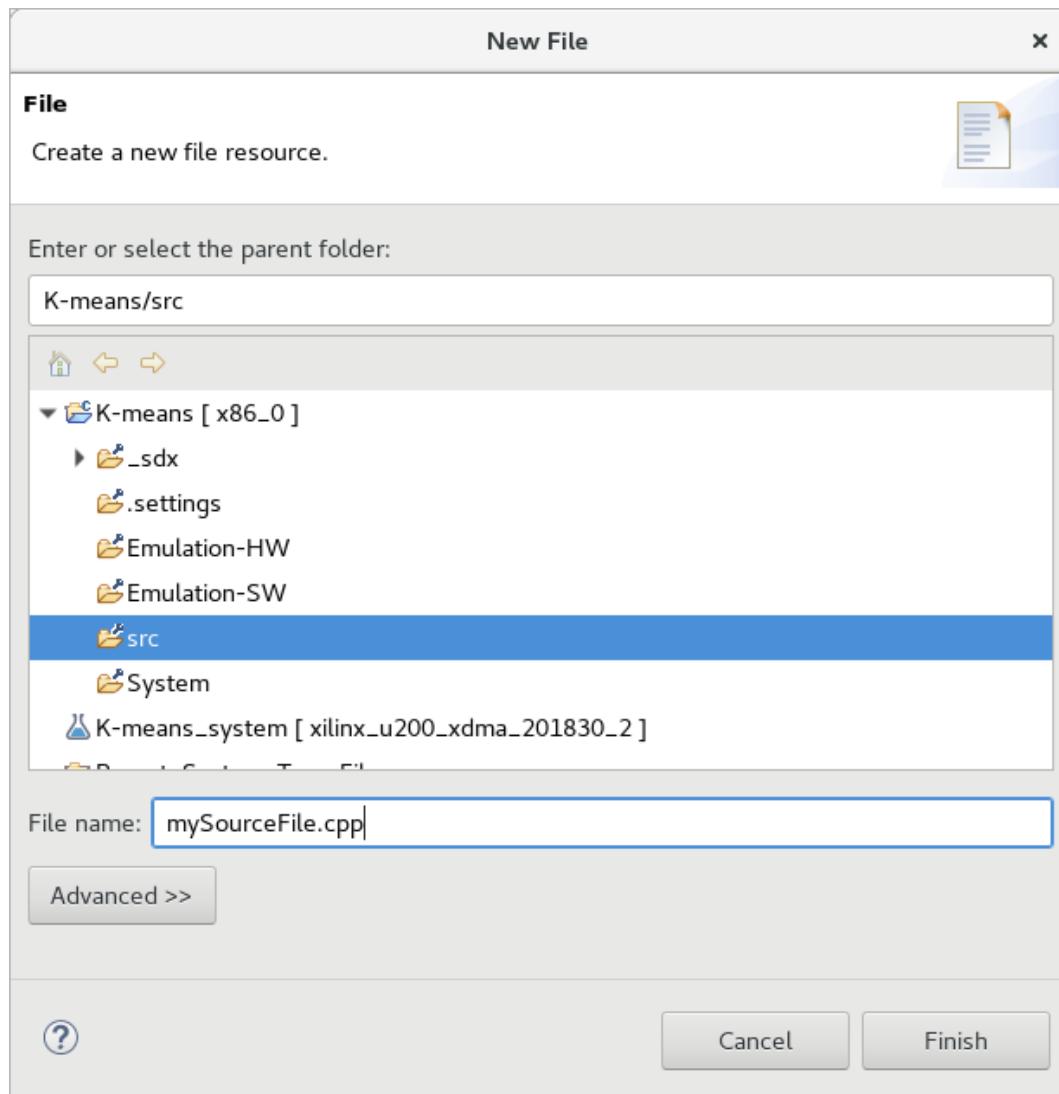
After adding source files to your project, you are ready to begin configuring, building, and running the application. To open a source file in the built-in text editor, expand the `src` folder in the Project Explorer and double-click on a specific file.

## Create and Edit New Source Files

You can also create and edit new source files directly in the Vitis IDE.

1. From the open project, right-click the `src` folder and select **New → File**.

The New File dialog box is displayed as shown in the following figure.



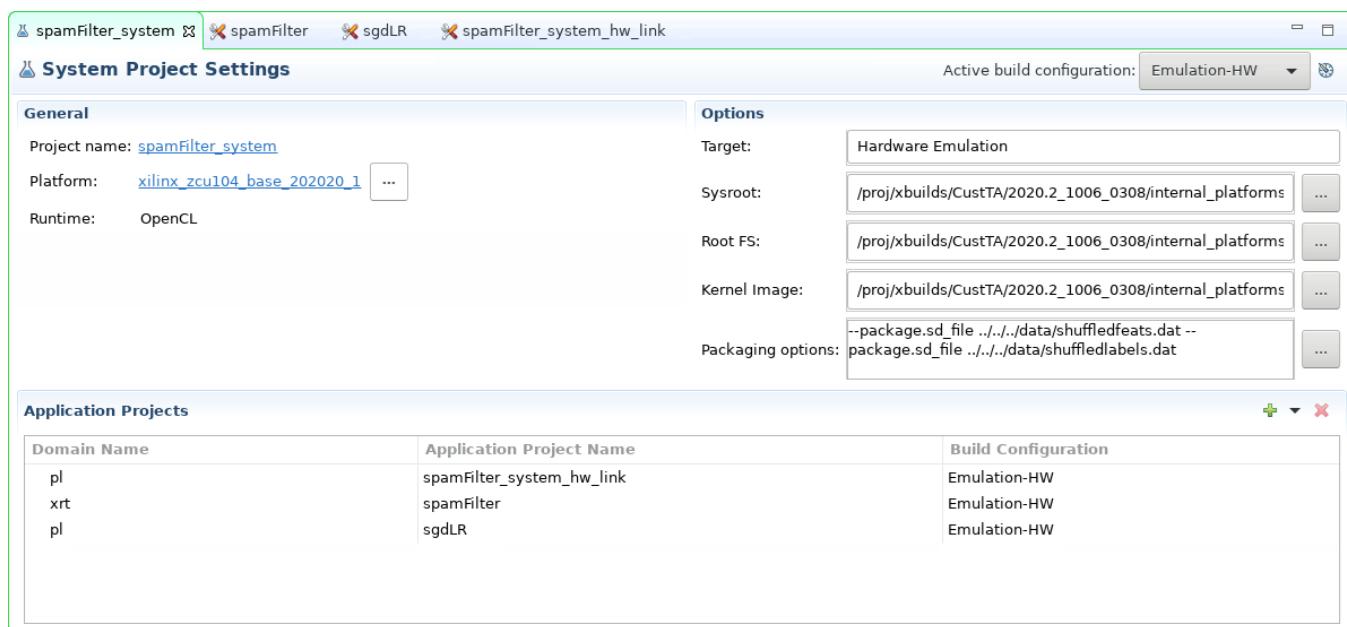
2. Select the folder in which to create the new file and enter a file name.
3. Click **Finish** to add the file to the project.

After adding source files to your project, you are ready to begin configuring, building, and running the application. To open a source file in the built-in text editor, expand the `src` folder in the Project Explorer and double-click on a specific file.

# Working in the Project Editor View

Building the system requires compiling and linking both the host program and the FPGA binary (`xclbin`). Your defined application project includes a top-level system project, host processor project, hardware kernel project, and `hw_link` project. Both the host and kernel projects contain source code in the `src` folder, as imported or created in the project. Any of these projects can be opened in the Project Editor view, shown in the following figure, which gives a top-level view of the project and its various build configurations.

Figure 68: Project Editor View



Depending on the type of project you are viewing, system project, host, kernel, or link, the Project Editor provides the following details:

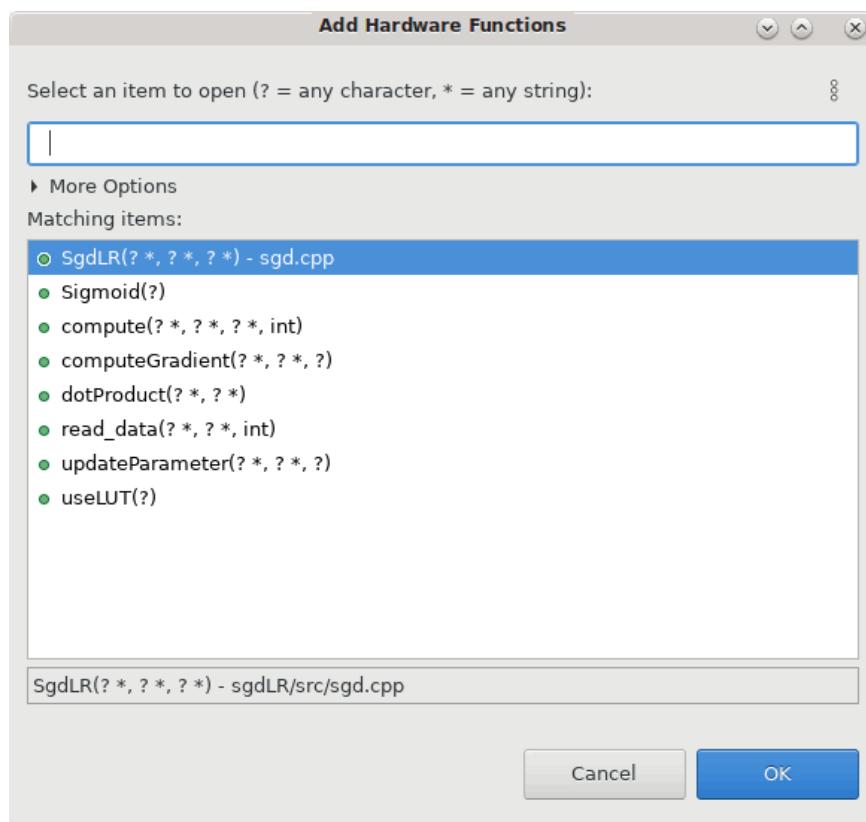
- General information about the project name
- Target platform
- Active build configuration
- Several configuration options related to the selected project

These include boot files for system project, debug options for the host or kernel projects, and a menu to select the report level of the hardware kernel project as discussed in [Controlling Report Generation](#).

The bottom portion of the Project Editor view displays Application Projects contained in the top-level system project, as shown in the figure above, or displays the Hardware Functions that will be compiled in a hardware kernel project, or assigned to the binary container in the `hw_link` project to be built into the `xclbin`.

To specify a function to be compiled in a hardware kernel project, click the **Add Hardware Function** (💡) button in the upper right of the Hardware Functions pane. This opens the Add Hardware Functions dialog box displaying a list of functions defined in the source code of the current project, as shown below.

**Figure 69: Adding Hardware Functions to a Binary Container**

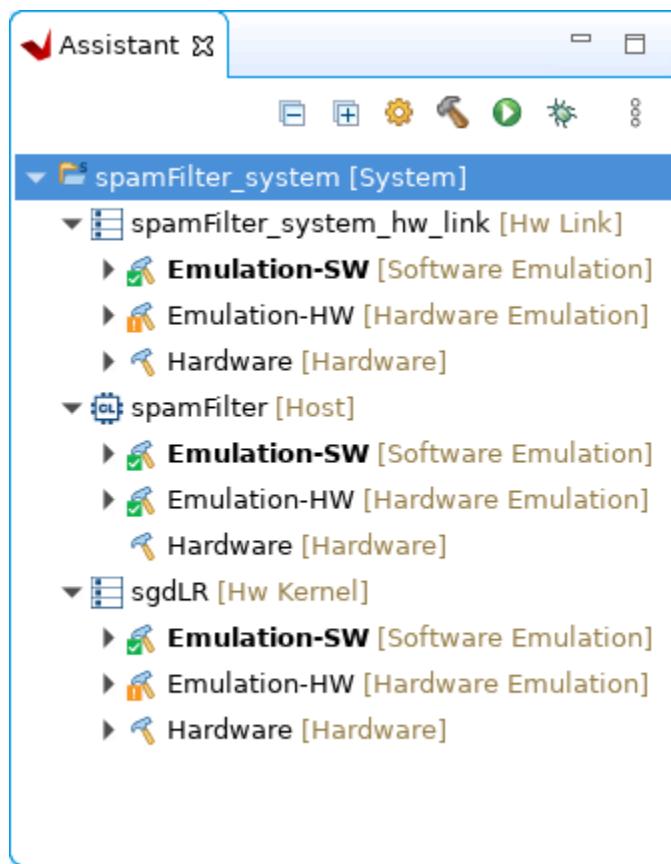


Select a function from the list to specify the hardware function, and click **OK**. The selected function becomes the target of the build process for the hardware kernel project and is also added to the device binary in the `hw_link` project.

# Working in the Assistant View

The Assistant view provides a project tree to manage build configurations, run configurations, and set the attributes of these configurations. It is a companion view to the Explorer view and displays directly below it in the default Vitis IDE perspective. The following figure shows an example Assistant view and its tree structure.

Figure 70: Assistant View



The objects displayed in the Assistant view hierarchy include the top-level system project, host project, hardware kernel projects, and the `hw_link` project. For each of these projects the different build configurations are also displayed: the software emulation and hardware emulation build configurations, and the hardware build configuration. The build configurations define the build target as described in [Build Targets](#), and specify options for the compilation and linking process.



**TIP:** The status of build configurations can be quickly determined from the icons displayed in the figure above:

- Emulation-SW builds are complete as indicated by the green check box

- Emulation-HW builds need to be updated for the hardware kernel and `hw_link` projects, as indicated by the yellow exclamation (!)
- Hardware builds are not built

When you select a build configuration, such as Emulation-HW build and click the **Settings** button (⚙), the [Vitis Build Configuration Settings](#) dialog box opens. You will use this Settings dialog box to configure the build process for the specific emulation or hardware target.

Within the hierarchy of these build configuration is the binary container (or `.xclbin`), the hardware function or functions, the run configuration, and any reports or summaries generated by the build or run process. When you select the hardware function for a specific build configuration and click the **Settings** button, the [Vitis Hardware Function Settings](#) dialog box is displayed. You will use this dialog box to specify the number of compute units for each kernel, assign compute units to SLRs, and assign kernel ports to global memory.

After a specific build configuration has been built, launch configurations become available for the project. There are two types of launch configurations:

- Run configurations specify the profile used for running the compiled and linked application; it defines the environment and options for running the host application and kernel code. Use the Run command (▶) on the toolbar menu to access run configurations. This opens the [Vitis Run and Debug Configuration Settings](#) dialog box, where you can configure the run before launching it.
- Debug configurations specify the profile for debugging the application. It launches the environment needed to interactively debug both the host and kernel code. You can access debug configurations through the Debug command (🐞) on the toolbar menu. Refer to [Vitis IDE Debug Flow](#) for more information.

Figure 71: Assistant View Menu



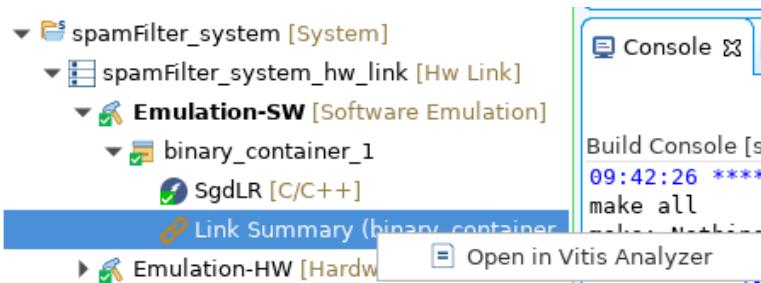
Within the Assistant view, the View menu includes options that affect what the Assistant view displays, or affect how the Assistant view interacts with other views. Open the View menu by left-clicking the menu command to display the following options:

- **Show Active Build Configurations Only:** When enabled, the Assistant view will only show the active build configuration for each project. This option can be useful to reduce the clutter in the Assistant view. To change the active configuration, select **Active build configuration** in the Project Editor view.

- **Link with Console:** When enabled, the build console in the Console view switches automatically to match the currently selected build configuration in the Assistant view. If not enabled, the build console does not automatically change to match the Assistant view.
- **Link with Guidance:** When enabled, the Guidance tab of the Console view automatically switches to match the current selection in the Assistant view.

For each of the build configurations, reports are generated during the build and run process and are displayed in the Assistant view. The different reports are grouped into Compile Summary, Link Summary, Package Summary, and Run Summary which can be viewed in the Vitis analyzer tool as described in [Section VIII: Using the Vitis Analyzer](#). You can right-click one of these summary reports in the Assistant view and select **Open in Vitis Analyzer** as shown below.

Figure 72: Open in Vitis Analyzer



# Building the System

When building the system, it best practice to use the three available build targets described in [Build Targets](#). Each build target is represented in a separate build configuration in the Assistant view. Work through these build configurations in the following order:

- **Emulation-SW:** Build for software emulation (`sw_emu`) to confirm the algorithm functionality of both the host program and kernel code working together.
- **Emulation-HW:** Build for hardware emulation (`hw_emu`) to compile the kernel into a hardware description language (HDL), confirm the correctness of the generated logic, and evaluate its simulated performance.
- **Hardware:** Perform a system hardware build (`hw`) to implement the application running on the target platform.

Before launching the build command, configure each of these build configurations to ensure it meets your needs. Select the specific build configuration, and click the **Settings** button to open the Build Configuration Settings dialog box. For more information on using this dialog box, refer to [Vitis Build Configuration Settings](#).

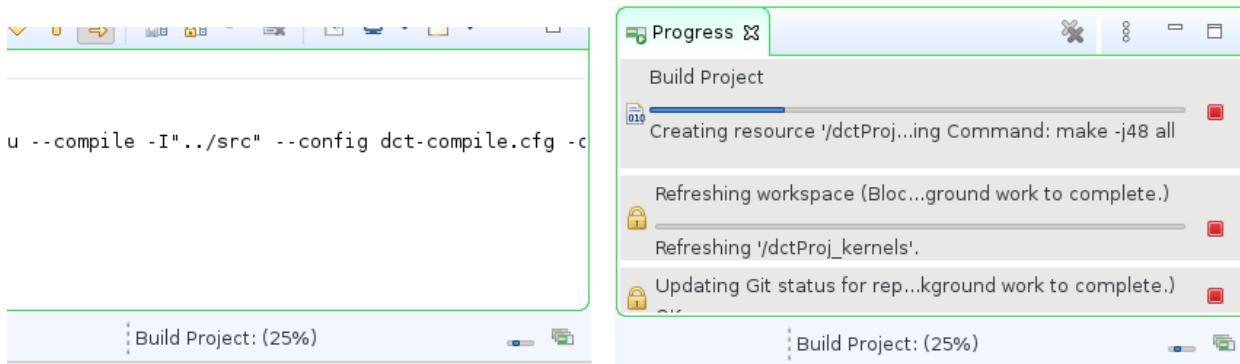
Beyond the build configuration settings, many of the settings that will affect your application are contained in the Hardware Function, accessed through the Vitis Hardware Function Settings dialog box. It is a good idea to review each of the Settings dialog boxes as discussed in [Configuring the Vitis IDE](#).

From the Assistant view, with the various options of the build configuration specified, you can start the build process by selecting a build configuration and clicking the **Build** (🔨) button. The Vitis core development kit uses a two part build process that generates the FPGA binary (`.xclbin`) for the hardware kernels using the Vitis™ compiler `v++` command, and compiles and links the host program code using the `g++` compiler.



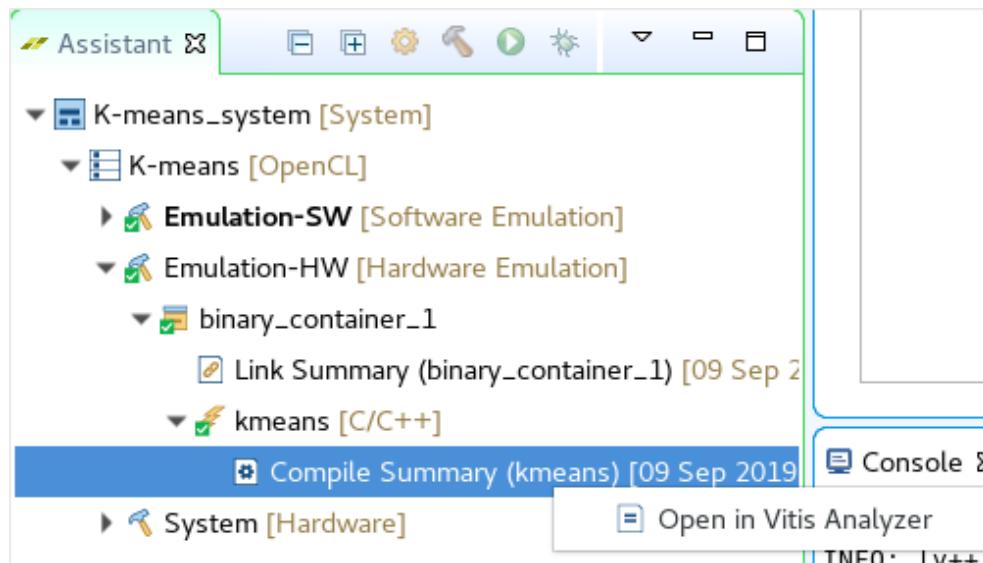
**TIP:** In the lower-right corner of the Vitis IDE the status of the in-progress build is displayed, as shown in the figure below on the left. If you click the button next to the status, the Progress view is opened as shown on the right. You can cancel the build by clicking the red square next to the build step, as shown in the figure below.

Figure 73: Cancel Build



After the build process is complete, the Assistant view shows the specific build configuration with a green check mark to indicate it has been successfully built, as shown in the following figure. You can open any of the build reports, such as the Compile Summary in the hardware function, or the Link Summary in the binary container. Right-click the report in the Assistant view and select **Open in Vitis Analyzer**.

Figure 74: Assistant View - Successful Builds



With the build complete, you can now run the application in the context provided by the specific build configuration. For instance, exercise a C-model of the host program and FPGA binary working together in the Emulation-SW build, or review the host program and the RTL kernel code in simulation in the Emulation-HW build, or run the application on the target platform in the Hardware build.

To run the application from within the Vitis IDE, select the build configuration, and click the **Run** button (▶) to launch the default run configuration. You can also right-click the build configuration and use the Run menu to select a specific run configuration, or edit a run configuration as described in [Vitis Run and Debug Configuration Settings](#).



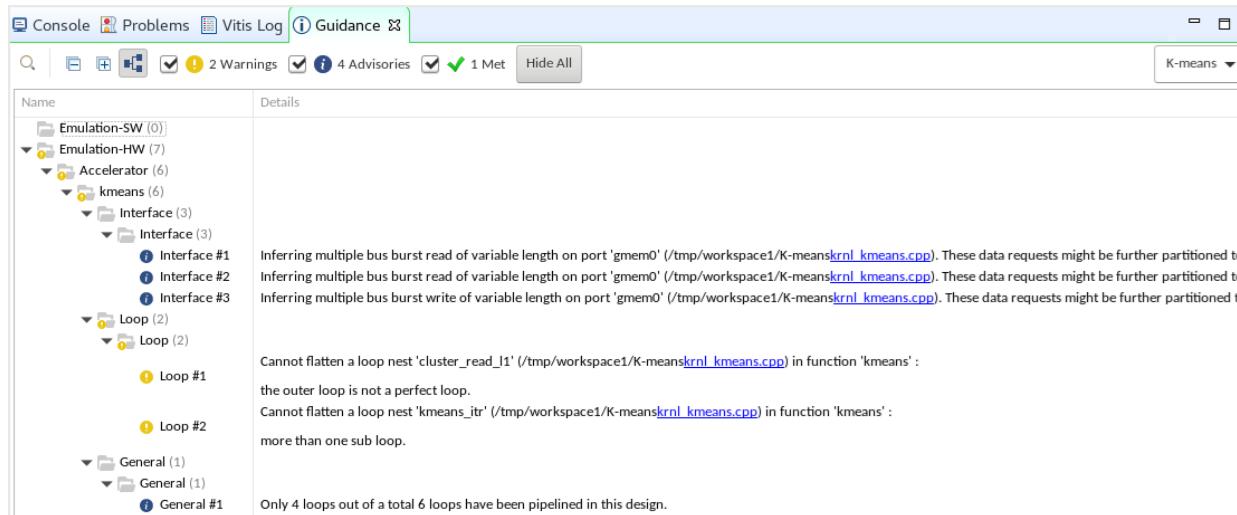
**TIP:** The Vitis IDE creates a folder named after the run configuration in the specific build configuration being run. For instance `./project/Emulation-HW/run_config`. The output files and logs from the application run are written to this folder. All arguments passed to the host program should be written relative to this folder.

## Vitis IDE Guidance View

After building or running a specific build configuration, the Guidance tab of the Console view displays a list of errors, warnings, and suggestions related to the build and run process. The Guidance view is automatically populated and displayed in the tabs located in the Console view. You can review the guidance messages to make any changes that might be needed in your code or build process.

After running hardware emulation, the Guidance view might look like the following figure.

**Figure 75: Guidance for the Build**



**TIP:** The Guidance report can also be viewed in Vitis analyzer as discussed in the [Section VIII: Using the Vitis Analyzer](#).

To simplify sorting through the Guidance view information, the Vitis IDE lets you search, and filter the Guidance view to locate specific guidance rule entries. You can collapse or expand the tree view, or even suppress the hierarchical tree representation and visualize a condensed representation of the guidance rules. Finally, you can select what is shown in the Guidance view by enabling or disabling the display of warnings, as well as rules that have been met, and also restrict the specific content based on the source of the messages such as build and emulation.

By default, the Guidance view shows all guidance information for the project selected in the drop down. To restrict the content to an individual build or run step, do the following:

1. Select **Window→Preferences**
2. Select the category **Guidance**.
3. Deselect **Group guidance rule checks by project**.

---

## Working with Vivado Tools from the Vitis IDE

The Vitis core development kit calls the Vivado Design Suite during the linking process to automatically run RTL synthesis and implementation when generating the FPGA binary (.xclbin). You also have the option of launching the Vivado tool directly from within the Vitis IDE to interact with the project for synthesizing and implementing the FPGA binary. There are three commands to support interacting with the Vivado tool from the Vitis IDE, accessed through the **Xilinx→Vivado Integration** menu:



**TIP:** The *hw\_link* project must be opened and be the current project in the IDE for these options to be available.

- **Open Vivado Project:** This automatically opens the Vivado project (.xpr) associated with the Hardware build configuration. In order for this feature to work, you must have previously completed the Hardware build so that a Vivado project exists for the build.

Opening the Vivado project launches the Vivado IDE and opens the implementation design checkpoint (DCP) file to edit the project, to let you manage the results of synthesis and implementation more directly. You can then use the results of this effort for generating the FPGA binary by selecting **Import Design Checkpoint**.

- **Import Design Checkpoint:** Lets you specify a Vivado DCP file to use as the basis for the Hardware build, and for generating the FPGA binary.
- **Import Vivado Settings:** Lets you specify a configuration file used by the Vivado tools, as described in [Vitis Compiler Configuration File](#), for use during the linking process.

Using the Vivado IDE in standalone mode enables the exploration of various synthesis and implementation options for further optimizing the kernel for performance and area. There are additional options available to let you interact with the FPGA build process. See [Managing Vivado Synthesis and Implementation Results](#) for more information.



**IMPORTANT!** *The optimization switches applied in the standalone project are not automatically incorporated back into the Vitis IDE build configurations. You need to ensure that the various synthesis and implementation properties are specified for the build using the `v++ --config` file options. For more information, refer to [v++ Command](#).*

# Vitis IDE Debug Flow

The Vitis™ IDE provides easy access to the debug capabilities. When performed manually, setting up an executable for debugging requires many steps. When you use the debug flow, these steps are handled automatically by the Vitis IDE.

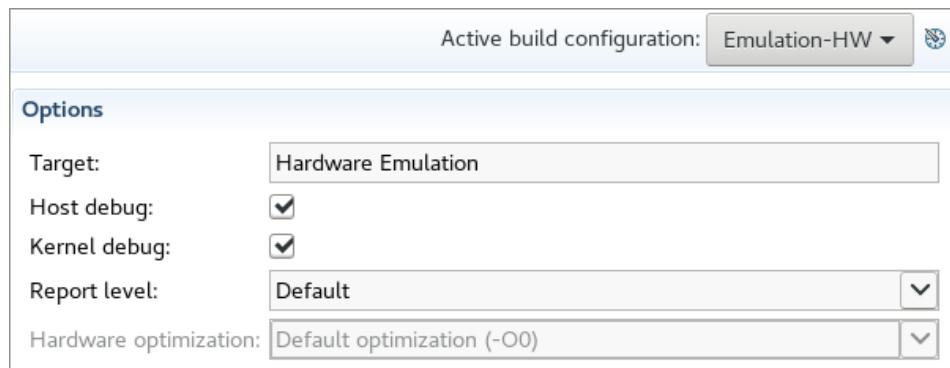
**Note:** The debug flow in the Vitis IDE relies on shell scripts during debugging. This requires that the setup files, such as `.bashrc` or `.cshrc`, do not interfere with the environment setup, such as the `LD_LIBRARY_PATH`.

To prepare the executable for debugging, you must change the build configurations to enable **Host debug** and **Kernel debug**. Set these options in the Project Editor view in the Vitis IDE, as shown in the following figure. There are two check boxes provided in the Options section for the Active build configuration:

- Host debug enables debugging constructs in the host compilation, and is available for all build types.
- Kernel debug enables debugging of the kernels, but is only available in software and hardware emulation builds. To enable debug in hardware builds, use the Chipscope Debug settings as described in [Vitis Hardware Function Settings](#).

These check boxes enable the `-g`, or `--debug` options in the `g++` and Vitis compilers.

**Figure 76: Project Editor View Debug Options**



You can also enable the debug features from the Build Configuration Settings dialog box, as shown in [Vitis Build Configuration Settings](#), by selecting the build configuration in the Assistant view and clicking the **Settings** button. Alternatively, you can double-click the build configuration. The same two check boxes are presented. While you can enable host debug on all targets, kernel debug is only supported for software emulation and hardware emulation build targets.

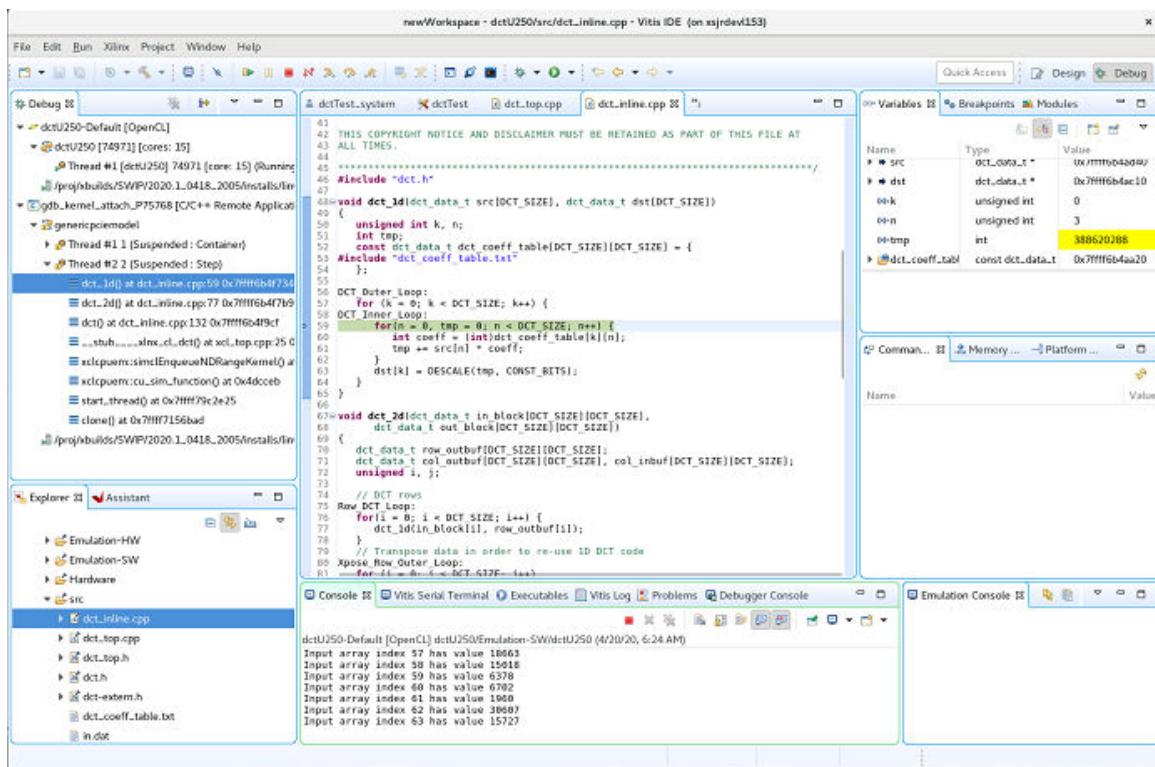
Running a GDB session from the Vitis IDE takes care of all the required setup. It automatically manages the environment setup for software emulation. It configures XRT to ensure debug support when the application is running, as described in [xrt.ini File](#), and manages the different consoles required for the execution of the host code, the kernel code, and the debug server.

When running on an embedded platform, the Vitis IDE also configures and launches the QEMU system mode, the logic simulator for the PL kernel, and manages synchronization between them. For more information, refer to [launch\\_emulator Utility](#).

After setting up the build configuration for debug, clean the build directory, and rebuild the application to ensure that the project is ready to run in the GDB debug environment.

To launch a debug session, select the build configuration in the Assistant view, and click **Debug** ( ). When launching the debug session in the Vitis IDE, the perspective switches to the Debug perspective, which is configured to present additional windows to manage the different debug views and source code windows. The following figure shows the Debug perspective.

**Figure 77: Debug Perspective**



After launching the debug environment, by default, the application is stopped at the beginning of the `main` function body in the host code. As with any GDB graphical front end, you can now set breakpoints and inspect variables in the host code. The Vitis IDE enables the same capabilities for the accelerated kernel implementation in a transparent way. For more information, refer to [Debugging Applications and Kernels](#).

# Using the Standalone Debug Flow

The Vitis IDE lets you open the debug tool for projects that have been built using the command line flow.

## Launching Standalone Debug for Embedded Platforms

The standalone debug flow supports both the embedded processor application acceleration flow (`embedded_accel`) or the embedded processor software development flow (`embedded`). For embedded platforms, the application is running on the Arm processor of the device, the files that are required to boot the system, and load the application and kernel, are on a remote system, but the debug tools are running on the local system, and the data and reports generated need to be moved from the embedded system to the local system. The process for debugging in that environment requires more setup and configuration.

Running standalone debug in the Vitis IDE for the `embedded_accel` flow is a two-step process.

1. You must first launch the QEMU emulator environment using the `launch_sw_emu.sh` or the `launch_hw_emu.sh` script, that is generated during the `--package` process.
2. Then you must launch the Vitis IDE in standalone debug mode using the `-debug` option.

To run standalone debug in the Vitis IDE for the `embedded` flow, you must first launch the QEMU emulator environment using the `launch_hw_emu.sh` script, that is generated during the `--package` process.

The files required for emulation of the system are also defined by the `--package` command. This means that launching the standalone debug process for embedded platforms is reliant on the output of the package process, including the emulation script. An example command to launch the emulation environment would include the following.

```
launch_hw_emu.sh -pid-file emulation.pid -no-reboot -forward-port 1440 1534
 \
 -enable-debug
```

Where:

- **`-enable-debug`:** Opens two different command shells to launch QEMU and XSIM, and enables the GDB connection to the QEMU shell.
- **`-forward-port`:** Forwards the TCP port from target to host for connecting to the QEMU shell. The QEMU port default is 1440. You can change it if necessary, for example, to 1446, but you must specify it for both the `launch_emulation` command or script and in the `vitis -debug` command line. Also, there is support for multiple forward ports enabled. For example, `launch_sw_emu.sh -forward-port 1440 1534 -forward-port 9455 1560`.
- **`-no-reboot`:** Exit the QEMU environment when done.

- **-pid-file:** Write the process ID to the specified file, used to kill the process, if necessary.

For hardware emulation, this launches two terminal windows running the QEMU system mode, and the Vivado simulator for simulating the PL kernel.

After the terminals and emulation are up and running, you can launch the Vitis IDE in standalone debug mode in a separate command shell:

```
vitis -debug -flow embedded_accel -target hw_emu -exe vadd.elf \
-program-args vadd.xclbin -kernels vadd
```

Where:

- **vitis -debug:** Launches the Vitis IDE in standalone debug mode.
- **-flow embedded\_accel:** Specifies the application acceleration flow on an embedded processor platform.
- **-target hw\_emu:** Indicates the target build being debugged.
- **-exe vadd.elf:** Indicates the executable application to run and debug.
- **-program-args vadd.xclbin:** Specifies the .xclbin file to be loaded as an argument to the executable.

There are more options that can be specified as described in [vitis -debug Command Line](#), and these options might be needed depending on the configuration of your application and build environment.

The default for embedded systems searches for the executable and the .xclbin file, and any other required input files, on the /mnt folder of the emulation environment, or the embedded system. You can change this by specifying the `-target-work-dir` when launching the tool. This launches the Vitis IDE with the Debug perspective enabled, running a debug configuration for the specified executable application and kernel code. From this point you can do all the debug activities like step in/step over/viewing variables/adding break points within the GUI-based debug environment.

## Launching Standalone Debug for Data Center Platforms

Launching standalone debug for Data Center applications is a bit simpler. In this case, you need to identify the build target and the executable to run and debug. The Data Center platforms do not require an emulation environment.

The following example launches the Vitis standalone debug for the `data_center` flow targeting the software emulation build. It specifies the executable, `host.exe`, which is looked for in the current directory, and specifies the kernel to debug.

```
vitis -debug -flow data_center -target sw_emu -exe host.exe -kernels
krnl_vadd
```

By default, the standalone debug flow looks in the current directory for specified files and to write results. You can specify the `-work-dir` option to indicate a different working directory from the default. This might be necessary when the `.xclbin` file is built in a different directory.

This launches the Vitis IDE with the Debug perspective enabled, letting you perform debug activities like step in/step over/viewing variables/adding break points within the GUI-based debug environment.

---

## vitis -debug Command Line

### Command Line Usage

The Vitis software platform standalone debug feature lets you launch the Vitis IDE for debugging an existing command line project. In the following sections, an explanation of each of the command line options is described with examples of launching the standalone debug environment for different platforms and target builds.

#### **-debug**

```
vitis -debug
```

Launches the Vitis IDE in standalone debug mode.

#### **-flow**

```
-flow [ data_center | embedded_accel | embedded ]
```

Specifies the type of application project being debugged. This configures the Vitis IDE for debugging Data Center applications running on Alveo cards; for example, application acceleration projects running on embedded platforms, such as the `zcu104_base` platform, or embedded software projects.



**IMPORTANT!** For `embedded` and `embedded_accel` flows, you must launch the QEMU system emulator using the `launch_hw_emu.sh` or `launch_sw_emu.sh` script generated during the `--package` step as described in [Packaging for Embedded Platforms](#), or using the `launch_emulator.py` command.

---

#### **-workspace**

```
-workspace <workspace>
```

Specifies the Vitis IDE workspace to use when opening the application project in debug mode. If this option is not specified, the tool will create a directory named `workspace` in the current working directory. If a directory named `workspace` already exists, the tool will use that as the workspace.

**-exe**

```
-exe <path_to_executable>
```

Specifies the file name and path to the application (host) executable.

For example:

```
vitis -debug -exe ./host.elf
```

**-target**

```
-target [ sw_emu | hw_emu | hw ]
```

Specifies the build target to use for debugging.



**TIP:** This only applies for `data_center` and `embedded_accel` flows.

For example:

```
vitis -debug -target hw_emu
```

**-program-args**

```
-program-args <program arguments>
```

Specifies the command line arguments to be passed to the host application at runtime. If not specified, the tool will pass the `.xclbin` as a program argument when the `data_center` or `embedded_accel` flows are selected.

For example:

```
vitis -debug -program-args ./xclbin in.dat
```

**-kernels**

```
-kernels <list of kernels>
```

Specifies the list of kernels to debug. Multiple kernel names can be specified, separated by commas. Listed kernels are defined as function-level breakpoints, so the debugger stops when kernel execution starts. If a kernel is not specified, no function-level debugging is provided.

This is valid only for `data_center` flows and is not supported for `embedded` or `embedded_accel` flows.

For example:

```
vitis -debug -kernels mmult madd
```

### **-work-dir**

```
-work-dir <path_to_working_directory>
```

Specifies the working directory to save generated output files and reports. This is valid for `data_center` and `embedded_accel` flows.

For the `data_center` flow, this is the directory where the specified .exe will be launched. For `embedded_accel` flow, the launch directory will be defined by `-target-work-dir`.



**TIP:** If not specified, the current working directory is used as the working directory.

### **-target-work-dir**

```
-target-work-dir <Target working directory>
```

This is the directory on the target board OS, and the QEMU environment, where the executable will be launched. This is valid for `embedded_accel` and also for `embedded` flows using a Linux OS.



**TIP:** If not specified, the target working directory is `/mnt`.

### **-xrt-ini**

```
-xrt-ini <path_to_xrt.ini>
```

Specifies the location of the `xrt.ini` file. Valid for `data_center` and `embedded_accel` flows.

If the location is specified, it will be looked for in the same directory as the application .exe or in the working directory.

### **-os**

```
-os [ linux | baremetal ]
```

Specifies the OS running on the target board. This is valid for the `embedded` flows.

**-host**

```
-host <host_name or ip_address>
```

Specifies the name or IP address of the host system where the TCF agent or `hw_server` is running. Valid for `embedded_accel` and `embedded` flows. If not specified, it the default host name is `localhost` for bare metal, and the default IP address is 192.168.0.1 for Linux target OS.

**-port**

```
-port <port number>
```

Port for TCF agent running on target Linux, or the port for `hw_server` running on local host for bare metal target. If not specified, the port is 1534 for `tcf-agent` and 3121 for `hw_server`.

**-launch-script**

```
-launch-script <path_to_tcl_script>
```

Specify a Tcl script to be sourced before attaching the application to the debugger. This is valid only for `embedded` flow with bare metal OS. The Tcl script can contain commands to initialize the board, download the application, add breakpoints and make the target ready for the debugger to attach.

# Configuring the Vitis IDE

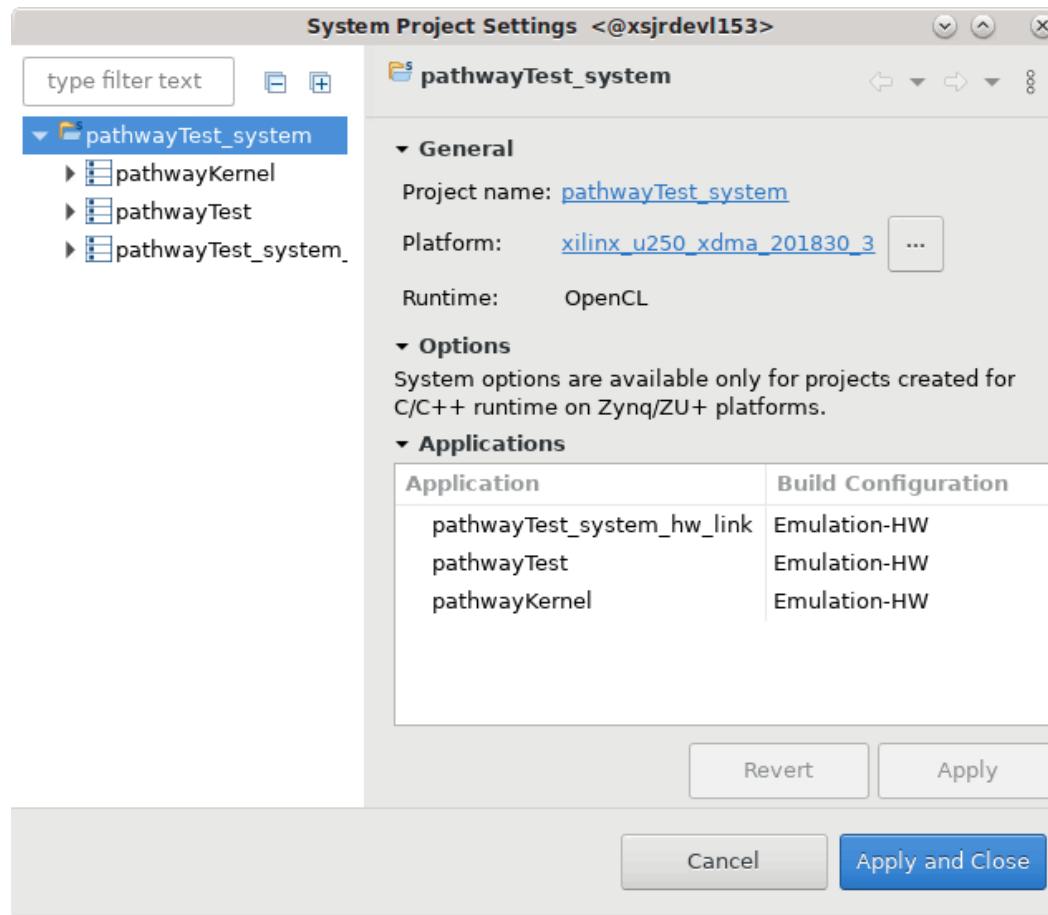
From the Assistant view, use the Settings button (⚙) to configure a selected project or configuration object. For more information, refer to the following topics:

- [Vitis Project Settings](#)
  - [Vitis Build Configuration Settings](#)
  - [Vitis Hardware Function Settings](#)
  - [Vitis Binary Container Settings](#)
  - [Vitis Toolchain Settings](#)
  - [Vitis Run and Debug Configuration Settings](#)
- 

## Vitis Project Settings

To edit the Vitis™ project settings, select the top-level system project in the Assistant view and click the **Settings** button (⚙) to bring up the Project Settings dialog box. This dialog box lets you specify both linking and compile options for the Vitis compiler `v++` command to let you customize the project build process.

Figure 78: Project Settings

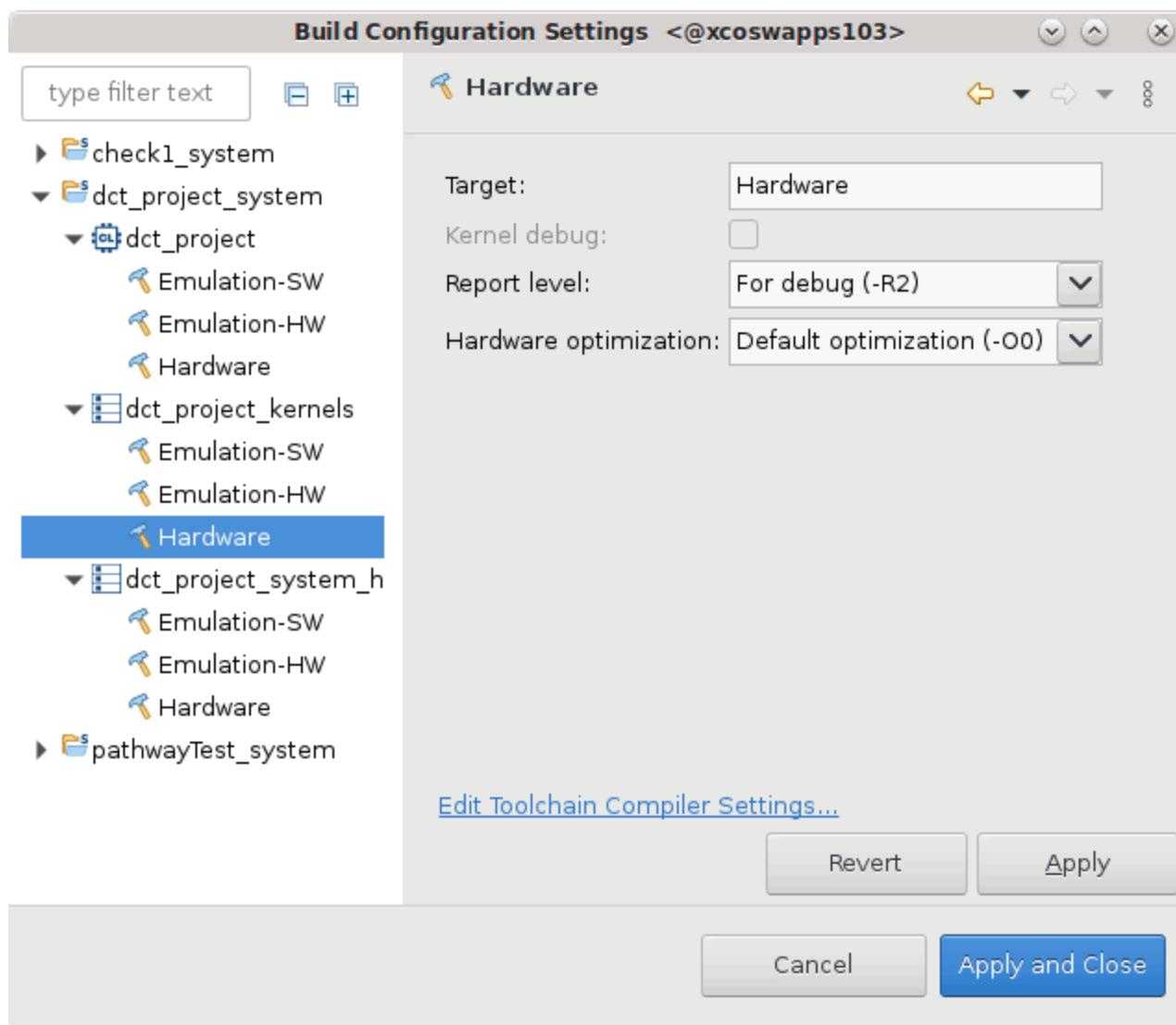


- **Project name:** Name of the project. Click the link to open the Properties dialog box for the project.
- **Platform:** Target platform for this project. Click the link to open the Platform Description dialog box. Click **Browse** to change the platform.
- **Runtime:** Displays the runtime used in this project.
- **Options:** Options are displayed when the Host application has been written using the XRT native C++ API.
- **Applications:** Lists the different sub-projects included with the system project, and also displays the current build target.

# Vitis Build Configuration Settings

To edit the settings for any of the build configurations under a project, select the build configuration in the Assistant view and click the **Settings** button (⚙) to bring up the Build Configuration Settings dialog box. The specific features of the dialog box vary depending on the type of project and the build target you have selected. In this dialog box, you can enable host and kernel debug, specify the level of information to report during the build process, and specify the level of optimization for the hardware build.

Figure 79: Build Configuration Settings



The various options displayed on this dialog box include:

- **Target:** The build configuration target as described in [Build Targets](#).
- **Host debug:** Select to enable debug of the host code.
- **Kernel debug:** Select to enable debug of the kernel code.
- **Report level:** Specify what report level to generate as described in [Controlling Report Generation](#).
- **Hardware optimization:** Specify how much effort to use on optimizing the hardware. Hardware optimization is a compute intensive task. Higher levels of optimization might result in more optimal hardware but with increased build time. This option is only available in the Build Configuration System.

The Build Configuration dialog box also contains links to Edit Toolchain Compiler Settings and Edit Toolchain Linker Settings. These links provide access to all the settings in the standard Eclipse environment, and can be used to configure the G++ compiler, the V++ compiler, and the emconfigutil command as described in [Vitis Toolchain Settings](#).

---

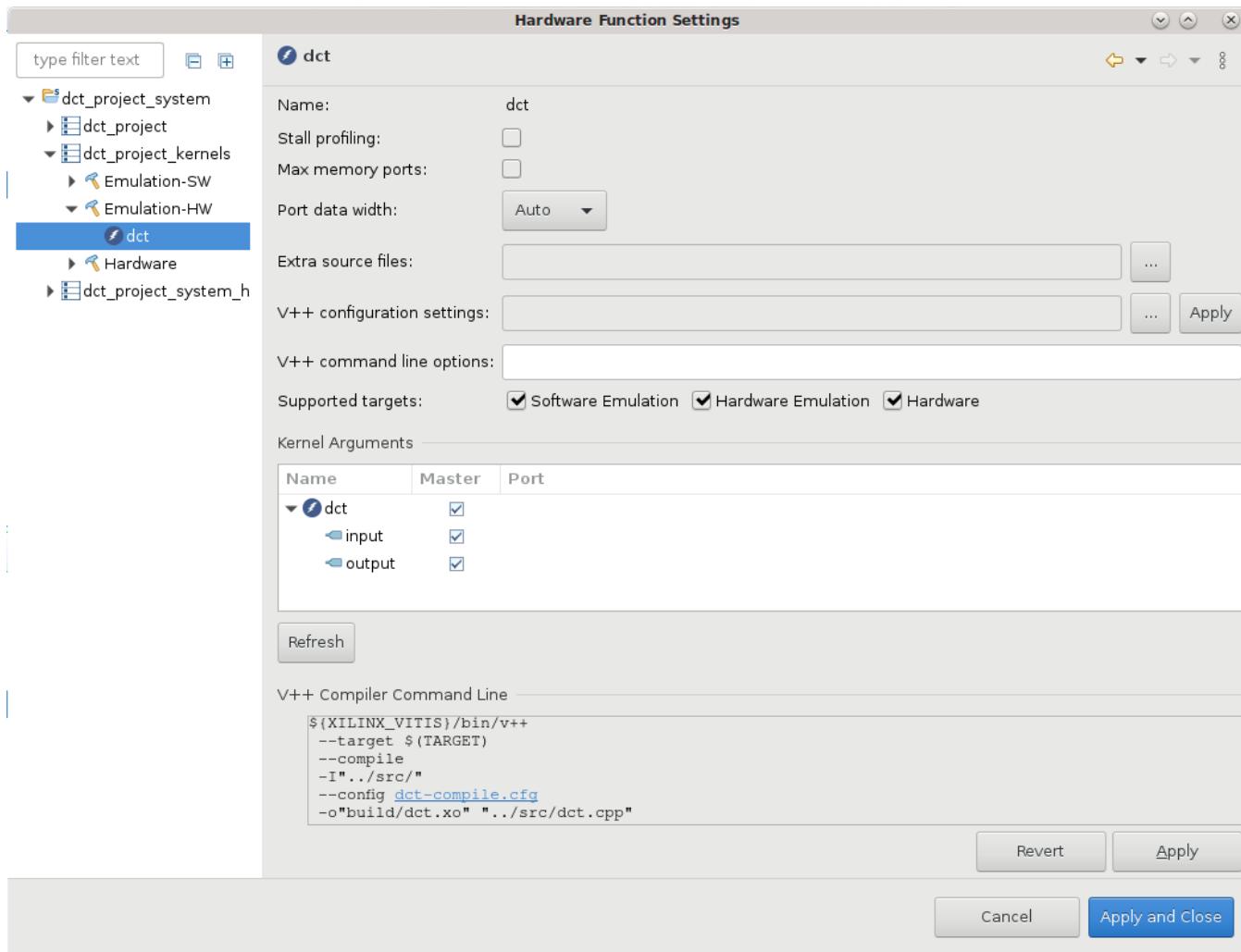
## Vitis Hardware Function Settings

To edit the settings for the hardware functions, expand the build target for the kernels project in the Assistant view, select the hardware function, and click the **Settings** button (⚙). This displays the Hardware Function Settings dialog box as shown in the following figure.



**TIP:** Notice in the figure below that the `dct` kernel (or hardware function) is selected for the `Emulation-HW` build target of the hardware kernels project (`dct-project-kernels`).

Figure 80: Hardware Function Settings



This dialog box lets you set options related to the hardware function, and the `v++` compilation process for the selected build target. Specific options include:

- **Stall Profiling:** Enables the `--profile.stall` option for the kernel as explained in [--profile Options](#).
- **Max memory ports:** For OpenCL kernels, when enabled, generates a separate physical memory interface (`m_axi`) for every global memory buffer declared in the kernel function signature. If not enabled, a single physical memory interface is created for the memory mapped kernel ports.
- **Port data width:** For OpenCL kernels, specify the width of the data port.
- **Extra source files:** Define any additional source files required by this hardware function, such as input data files.

- **V++ configuration settings:** Specify Vitis compiler options to be added to the compiler configuration file. Select the **Edit** command (...) to edit the options to add to the config file. Specified options will be added to the `compile.cfg` file that is linked in the V++ Compiler Command Line displayed at the bottom of the dialog box.
- **V++ compiler options:** Specify Vitis compiler options to be added to the V++ Compiler Command Line displayed at the bottom of the dialog box.
- **Supported targets:** Specify which of the three build targets you are defining in the Hardware Function Settings dialog box. You can select one or all build targets.
- **Kernel Arguments:** Displays the names and attributes of the arguments of the hardware kernel.
- **V++ Compiler Command Line:** Displays the current `v++` command line with any compilation options you have specified.



**TIP:** The settings specified by the Hardware Function Settings dialog box are written to a configuration file used by the Vitis compiler with the `--config` option as described in [Vitis Compiler Configuration File](#). The configuration file is a link; when you place your mouse over the link, it displays the contents of the configuration file.

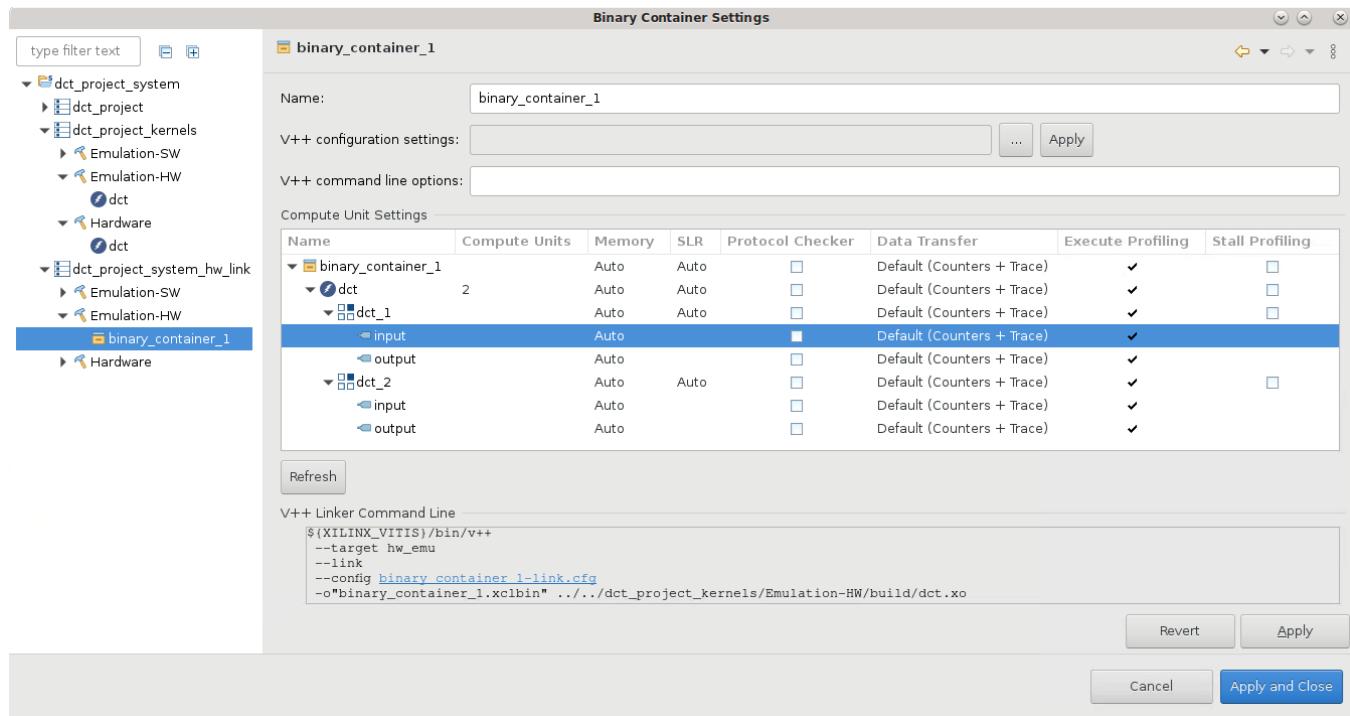
## Vitis Binary Container Settings

To edit the settings for a binary container, expand the build target for the `hw_link` project in the Assistant view, select the `binary_container`, and click the **Settings** button (⚙). This displays the Binary Container Settings dialog box as shown in the following figure.



**TIP:** The options displayed in the Binary Container Settings dialog box depend on the specific build target selected, and will vary between software emulation, hardware emulation, and hardware.

Figure 81: Binary Container Settings



This dialog box lets you specify a new name for the binary container, and to specify link options for the `v++` command. Specific options include:

- **Name:** Specify the name of the binary container.
- **V++ configuration settings:** Specify Vitis linker options to be added to the configuration file. Select the **Edit** command (...) to edit the options to add to the config file. Specified options will be added to the `binary_container-link.cfg` file that is linked in the V++ Linker Command Line displayed at the bottom of the dialog box.
- **V++ command line:** Enter link options for the selected binary container to be added to the V++ Linker Command Line displayed at the bottom of the dialog box. For more information on the available options, refer to [V++ Command](#).
- **Compute Units:** Specify the number of kernels to add to the device binary as explained in [--connectivity Options](#). The field can be edited, and when you have entered the value, additional kernels will be displayed in the dialog box.
- **Memory:** Specify global memory assignments for each argument of a compute unit as discussed in [Mapping Kernel Ports to Memory](#).
- **SLR:** Define the SLR placement for each compute unit of the kernel as discussed in [Assigning Compute Units to SLRs on Alveo Accelerator Cards](#).
- **ChipScope Debug:** Add monitors to capture hardware trace debug information. This specifies the `--debug.chipscope` option.

- **Protocol Checker:** Add AXI Protocol Checker to your design. This relates to the `--debug.protocol` option.
- **Data Transfer:** Add performance monitors to capture information related to data transferred between compute unit and global memory. Captured data includes counters, trace, or both. This relates to the `--profile.data` option, and to `xrt.ini` file settings as explained in [Enabling Profiling in Your Application](#).
- **Execute Profiling:** Add an accelerator monitor to capture the start and end of compute unit executions. This relates to the `--profile.exec` option.
- **Stall Profiling:** Add an accelerator monitor with functionality to capture stalls in the flow of data inside a kernel, between two kernels, or between the kernel and external memory. This relates to the `--profile.stall` option.
- **V++ Linker Command Line:** Displays the current `v++` command line with any link options you have specified.



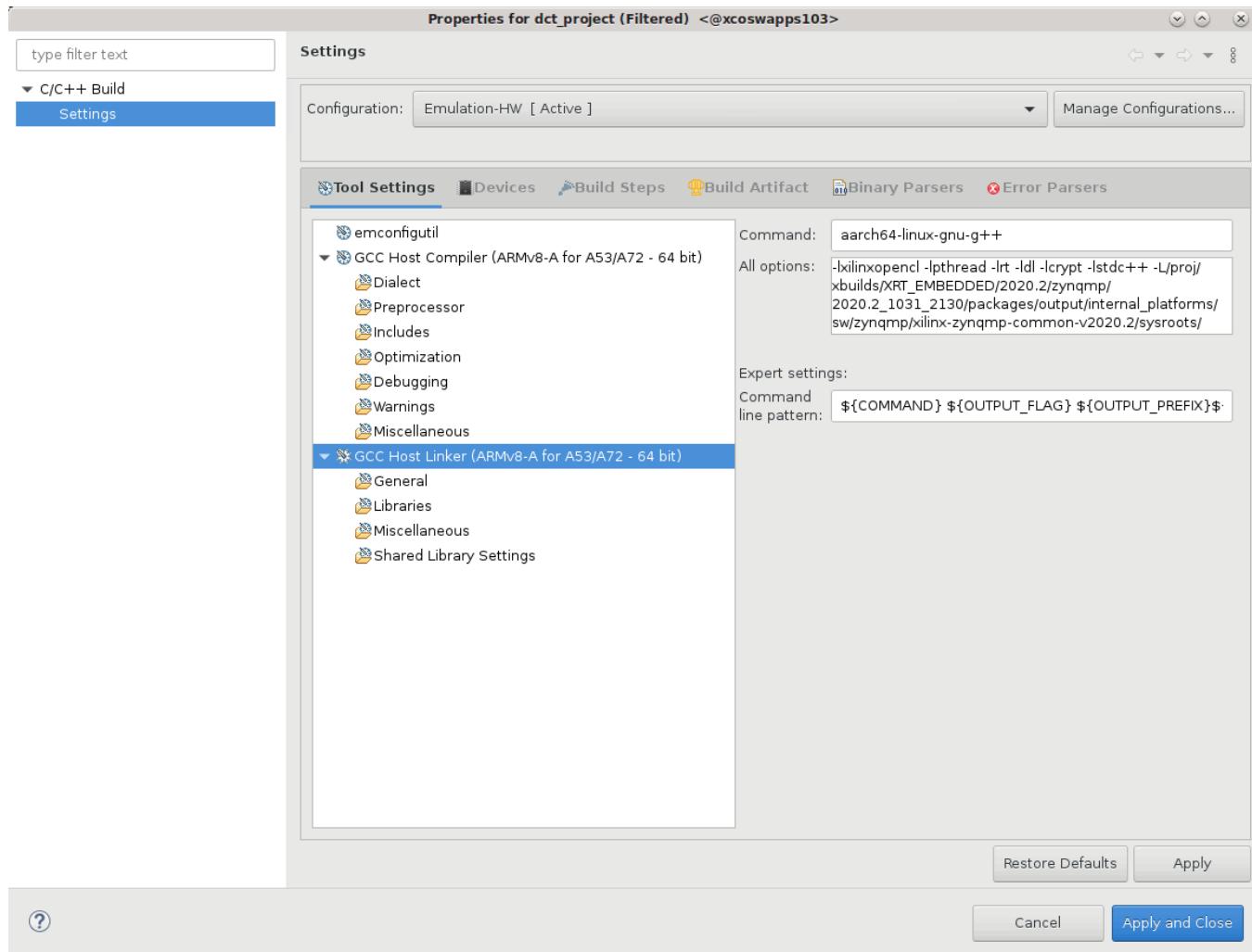
**TIP:** The settings specified by the Binary Container Settings dialog box are written to a configuration file used by the Vitis compiler with the `--config` option as described in [Vitis Compiler Configuration File](#). The configuration file is a link; when you place your mouse over the link, it displays the contents of the configuration file.

## Vitis Toolchain Settings

The toolchain settings provide a standard Eclipse-based view of the project, providing all options for the C/C++ build in the Vitis IDE.

From the Build Configuration Settings dialog box, click **Edit Toolchain Compiler Settings** or **Edit Toolchain Linker Settings** to bring up the compiler and linker Settings dialog box containing all of the C/C++ build settings. This dialog box lets you set standard C++ paths, include paths, libraries, project wide defines, and host defines.

Figure 82: Toolchain Settings



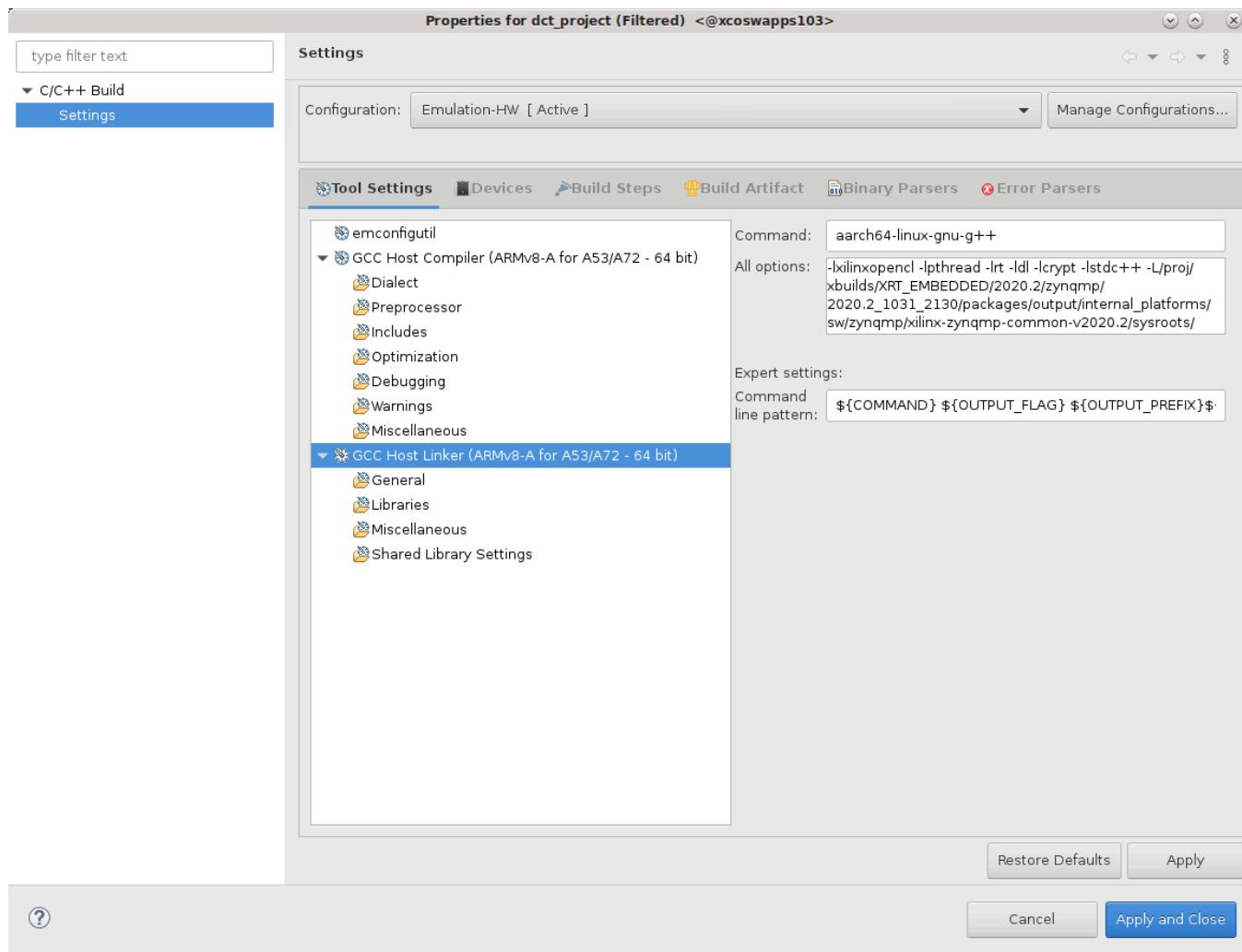
The contents of the Toolchain Settings dialog box depends on the specific Build Configuration dialog box you launched it from. You can launch it from the host application project, the hardware kernels project, or the `hw_link` project. The specific settings available include the following:

- **emconfigutil:** Specify the command line options for [emconfigutil Utility](#). See [emconfigutil Settings](#).
- **GCC Host Compiler:** Specify `g++` linker arguments that must be passed during the host compilation process. See [G++ Host Compiler and Linker Settings](#).
- **GCC Host Linker:** Specify `g++` linker arguments that must be passed during the host linking process. See [G++ Host Compiler and Linker Settings](#).

- **V++ Kernel Compiler:** Specify the `v++` command and any additional options that must be passed when calling the `v++` command for the kernel compilation process. See [V++ Compiler and Linker Settings](#).
- **V++ Kernel Linker:** Specify the `v++` command and any additional options to be passed when calling the `v++` command for the kernel linking process. See [V++ Compiler and Linker Settings](#).

## G++ Host Compiler and Linker Settings

Figure 83: GCC Compiler and Linker Settings



You can open the **GCC Compiler and Linker Settings** dialog box from the Build Configuration Settings dialog box of a host application project. The arguments for the `g++` compiler used by the Vitis core development kit can be accessed under the **GCC Host Compiler** section of the Toolchain Settings.

- **Dialect:** Specify the command options that select the C++ language standard to use. Standard dialect options include C++ 98, C++ 2011, and C++ 2014 (1Y).

 **IMPORTANT!** Host applications using the XRT native API must be compiled with `-std=c++17`.

- **Preprocessor:** Specify preprocessor arguments to the host compiler such as symbol definitions. The default symbols already defined include the platform so that the host code can check for the specific platform.
- **Includes:** Specify the include paths and include files.
- **Optimization:** Specify the compiler optimization flags and other optimization settings.
- **Debugging:** Specify the debug level and other debugging flags.
- **Warnings:** Specify options related to compiler warnings.
- **Miscellaneous:** Specify any other flags that are passed to the `g++` compiler.

### GCC Linker Options

The linker arguments for the Vitis technology G++ Host Linker are provided through the options available here. Specific sections include general options, libraries and library paths, miscellaneous linker options, and shared libraries.

## V++ Compiler and Linker Settings

Figure 84: V++ Compiler Settings



You can open the **V++ Compiler Settings** dialog box from a hardware kernel project Build Configuration Settings dialog box. The V++ Kernel Compiler settings shows the `v++` command used during compilation, and any additional options that must be passed when calling the `v++` command for the kernel compilation process. The `v++` command options can be symbols, include paths, or miscellaneous valid options.

- **Symbols:** Click **Symbols** under Vitis compiler to define any symbols that are passed with the `-D` option when calling the `v++` command.

- **Includes:** To add include paths to the Vitis compiler, select **Includes** and click the **Add** (  ) button.
- **Miscellaneous:** Vitis specific settings, such as the Vitis compiler and linker flags, which are not part of the standard C/C++ tool chain, can be added as flags in the Miscellaneous section. For more information on the available compiler options, refer to [v++ Command](#).

Figure 85: V++ Linker Settings



You can open the **V++ Linker Settings** dialog box from a `hw_link` project Build Configuration Settings dialog box. The **V++ Linker Settings** shows the `v++` command used during linking, as well as any additional options to be passed when calling the `v++` command for the kernel linking process.

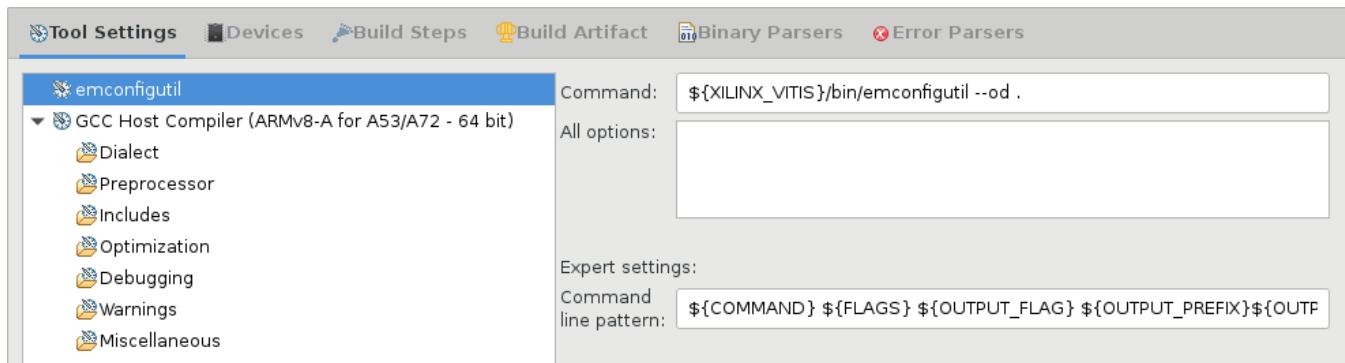
Any additional options that need to be passed to the Vitis compiler can be added as flags in the Miscellaneous section. For more information, refer to [v++ Command](#) for the available options in the linking process.

## emconfigutil Settings

Select the `emconfigutil` command options in the **GCC Compiler and Linker Settings** dialog box, which you can open from the Build Configuration Settings dialog box of a host application project. The **Command** field specifies the `emconfigutil` command that will launch the Vitis IDE, as described in [emconfigutil Utility](#). You can also specify any options for the command line as needed for your project.

The Vitis IDE creates the `emconfig.json` file by running the specified `emconfigutil` command before launching an emulation run configuration.

Figure 86: emconfigutil Settings



## Vitis Run and Debug Configuration Settings

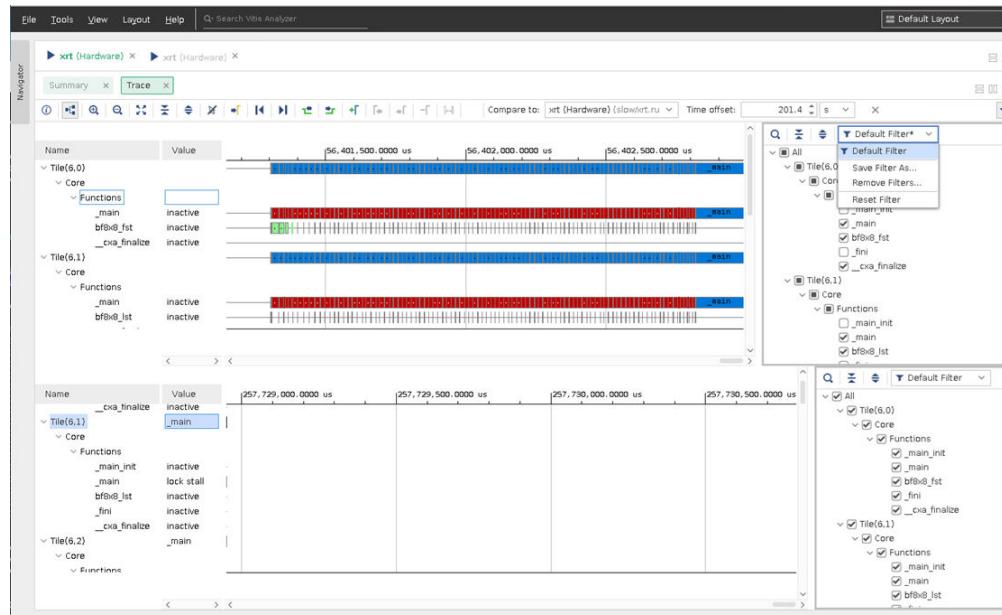
Launching the compiled, linked, and packaged application, to run or to debug within the Vitis IDE requires the use of the Launch Configuration dialog box, as shown in the figure below. When the build process has completed, the tool enables both the **Run** button and the **Debug** button in the Assistant view to let you specify a Launch Configuration to use.



**TIP:** The Launch Configuration is the same for both running and debugging the application. There are differences in the steps that the tool follows when running the application or launching the debugger, but you can use the same configuration for both purposes.

To edit the settings for a launch configuration, select a build target and click the **Run** button to open the Run Configurations dialog box. The Run Configuration dialog box, as shown below, lets you specify debug options, enable profiling of the running application, and specify the types of profiling data to collect.

Figure 87: Run Configuration Settings

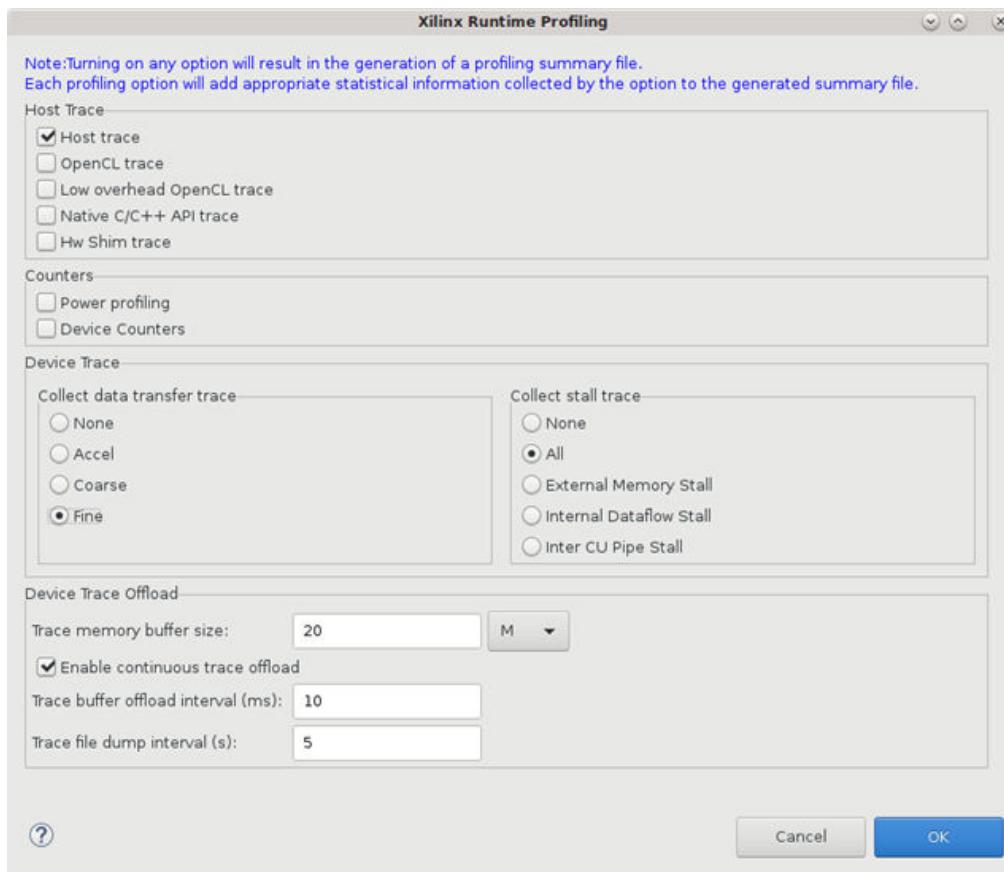


**TIP:** The options displayed in the Launch Configuration dialog box vary for Data Center and Embedded Processor platforms, and also for the software and hardware emulation builds, and the hardware build.

- **Name:** Specify the name of the run configuration. The Vitis IDE creates a folder named after the run configuration in the specific build configuration being run. For instance `./project/Emulation-HW/run_config`. The output files and logs from the application run are written to this folder. All arguments passed to the host program should be written relative to this folder.
- **Project:** Display the current project, but can be changed to other open projects.
- **Build Configuration:** Select the build target that the launch configuration applies to, or it applies to the active build configuration.
- **Disable Build Before Launch:** When enabled this check box prevents the tool from rebuilding the project before running or debugging it.
- **Target:** Specify the run or debug target for the configuration. Note that emulation builds targets the Linux TCF agent, while the hardware build requires the `hw_server`, as described in [Debugging Applications and Kernels](#).
- **Remote Working Directory:** For embedded processor systems, specifies the mount disk for the QEMU environment, or for the physical device.
- **Program Arguments:** Display the Programs Argument dialog box. This lets you specify command line arguments for your application if any are needed. Enabling the **Automatically update arguments** check box allows the tool to automatically specify the `xclbin` file as an argument for the application. The `xclbin` file is appended at the end of the command line, after any other specified arguments.

**Note:** Program arguments should be specified relative to the run configuration folder that is created during the run. You can refer to the Vitis log window in the IDE to review a copy of the command line used to launch the application.

- **Override Application Options:** Generate an options file for the `launch_emulator.py` command, that you can manually edit to customize as needed for your purposes. Click the **Generate** button to create a new `launch_options.cfg` file, or use the **Browse** button to locate an existing one.
- **Xilinx Runtime Profiling:** This specifies the profiling and event trace features that are enabled during the application run. The specified options are stored in a configuration file and written to the `xrt.ini` file to use while running the application. Click the **Edit** button to open the Xilinx Runtime Profiling dialog box as shown below.



**Note:** The options on the Xilinx Runtime Profiling window are displayed based on the Build Configuration selected in the Run Configuration window.

- **Host Trace:** Specify the type of host profiling you want enabled for the run. The **Host trace** option is enabled by default, but can be overridden by adding one or more of the other selections as described in [Enabling Profiling in Your Application](#).



**TIP:** For hardware builds you can also enable **Power profiling** or **Device Counters** for the accelerator card.

- **Device Trace:** Specify level of profiling performed on the hardware kernels. This option relates to the `device_trace` as described in [xrt.ini File](#). Coarse device profiling shows data transfer activity of the CU. Fine device profiling shows all AXI-level transactions on the ports.
- **Collect Stall Trace:** Indicate the capture of stall data for a variety of conditions, as described in [--profile Options](#), and [xrt.ini File](#).
- **Device Trace Offload:** Specify the amount of global memory to allocate to the capture of trace data, and also to enable continuous trace offload at the specified interval to prevent the loss of Timeline Trace data in the case of an interrupted process, as described in [Enabling Profiling in Your Application](#).

The Launch Configuration dialog box has additional tabs to help configure the runtime environment for your application. The three tabs are:

- **Target Setup:** Mainly for embedded platforms with a bare-metal application. It provides options to initialize, manage, and reset the board, device, and program.
- **Environment:** Lets you set up and manage environment variables needed for the Vitis IDE.
- **Common:** This tab is inherited from Eclipse. These are settings and feature common to the Eclipse environment.

# Project Export and Import

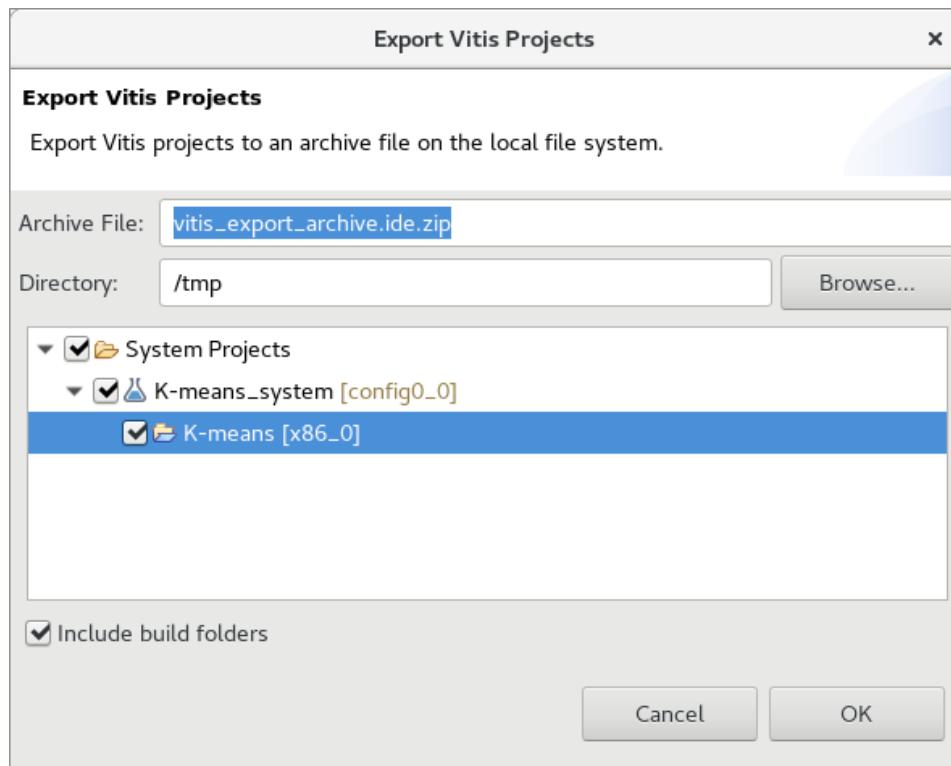
The Vitis™ IDE provides a simplified method for exporting or importing one or more Vitis IDE projects within your workspace, or import from GIT repositories. You can optionally include associated project build folders.

## Export a Vitis Project

When exporting a project, the project is archived in a zip file with all the relevant files needed to import to another workspace.

1. To export a project, select **File → Export** from the main menu.

The Export Vitis Projects page opens, where you select the project or projects in the current workspace to export as shown in the following figure.



2. To change the name for the archive, edit the Archive File field.

3. To include the current build configurations, enable **Include build folders** at the bottom of the dialog box.



**TIP:** This can significantly increase the size of the archive, but might be necessary in some cases.

4. To create the archive with your selected files, click **OK** to create the archive.

The selected Vitis IDE projects are archived in the specified file and location, and can be imported into the Vitis IDE under a different workspace, on a different computer, by a different user.

---

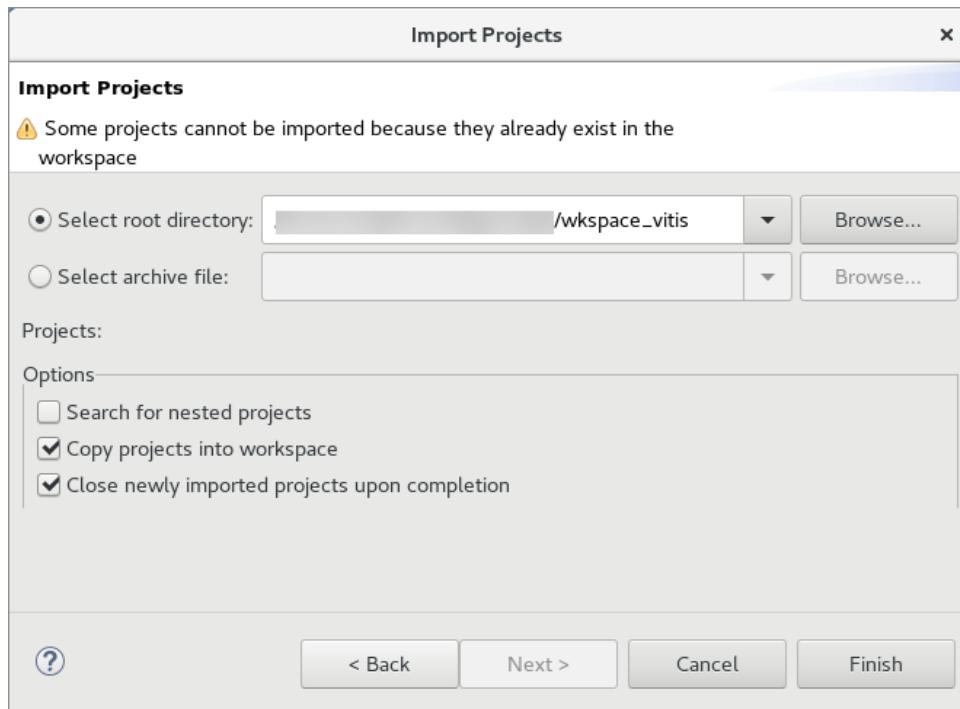
## Import a Vitis Project

1. To import a project, select **File → Import** from the top menu.

This opens the Import Projects page to select the import file type. There are two types of files you can select to import:

- **Vitis project exported zip files:** Lets you import projects previously exported from the Vitis IDE as discussed in [Export a Vitis Project](#).
- **Eclipse workspace or zip file:** Lets you import projects from another Vitis IDE workspace.
- **Import projects from Git:** Lets you import projects from either a local previously cloned Git repository, or from a specified Git URL, as described in the next topic.

2. The following figure shows the page that is opened when you select **Eclipse workspace or zip file** and click **Next**.



3. For Select root directory, point to a workspace for the Vitis IDE, and specify the following options as needed:
  - **Search for nested projects:** Looks for projects inside other projects in the workspace.
  - **Copy projects into workspace:** Creates a physical copy of the project in the current open workspace.
  - **Close newly created imported projects upon completion:** Closes the projects in the open workspace after they are created.
4. Click **Finish** to import the projects into the open workspace in the Vitis IDE.

---

## Import Projects from Git

1. From the Import Projects wizard, select the **Import projects from Git** option, and select **Next** to proceed.

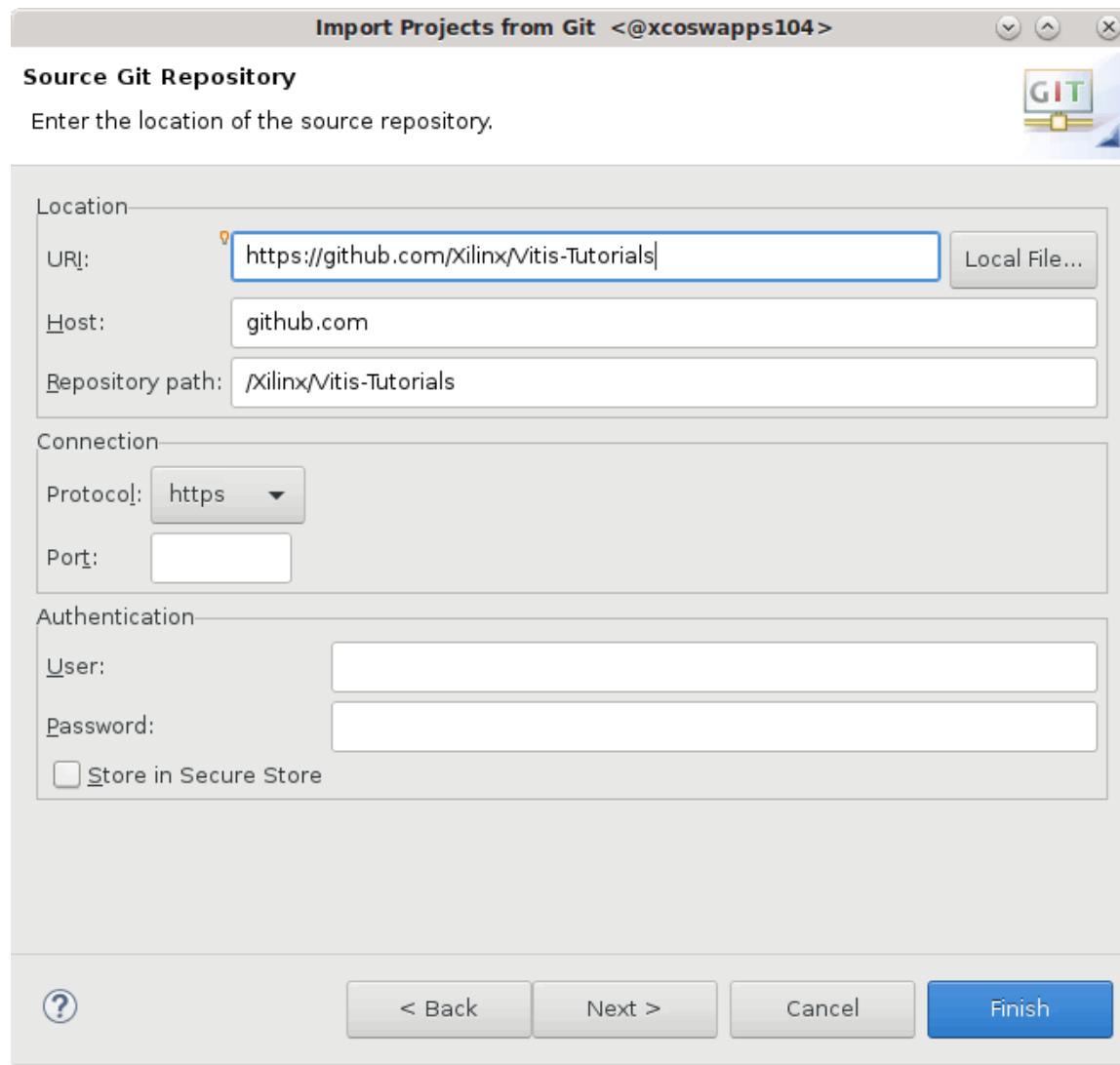
This opens the Select Repository Source page. There are two types of repositories you can select:

- **Existing local repository:** Selects an existing Git repository that has already been cloned locally. When you select this option, the Select a Git Repository page displays the list of currently cloned local repositories found by the Vitis IDE. Click **Next** to continue.



**TIP:** You can create a local Git repository by cloning the URL as described below, or by using the `git clone <url>` command from the Linux shell.

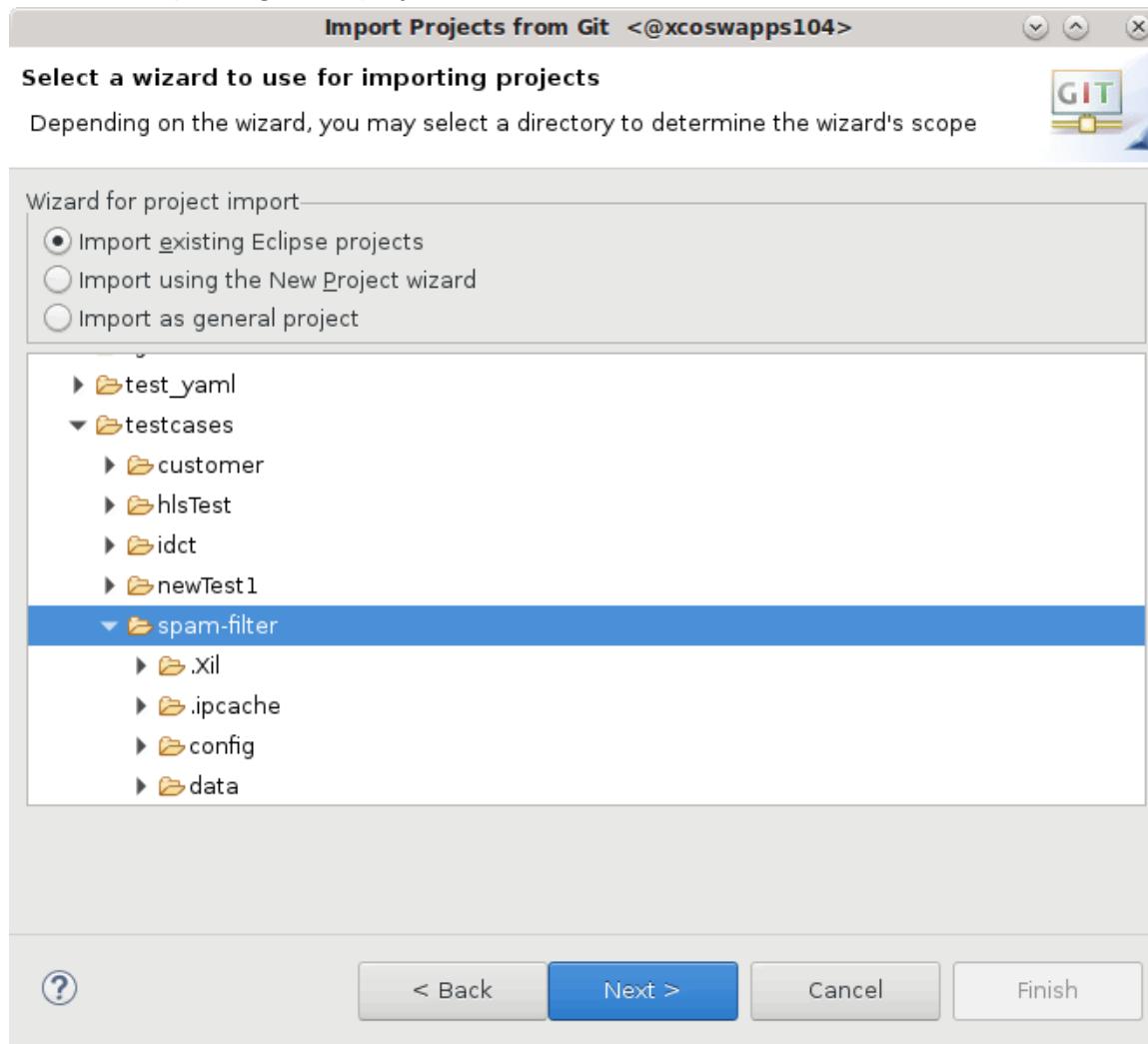
- **Clone URL:** Lets you specify a Git URL to clone to the specified location. When you select this option the Source Git Repository page of the Import Projects wizard is displayed as shown below.



2. On the Source Git Repository page, specify the following and click **Next** to continue:

- **Location:** Specify the URL for the repository. The host and repository path are extracted from the provided URL. Some URLs that might be of interest include:
  - Vitis Acceleration Examples: [https://github.com/Xilinx/Vitis\\_Accel\\_Examples](https://github.com/Xilinx/Vitis_Accel_Examples)
  - Vitis Libraries: [https://github.com/Xilinx/Vitis\\_Libraries](https://github.com/Xilinx/Vitis_Libraries)
  - Vitis Tutorials: <https://github.com/Xilinx/Vitis-Tutorials>
- **Connection:** Specifies the connection protocol used to connect. Use these fields to customize the connection if necessary.
- **Authentication:** Specifies the User ID and Password to access the repository if one is required.

3. In the Branch Selection page, you can select one or more branches to clone. Click **Next** to proceed. In the Local Destination page, specify the **Destination Directory** where the repository will be cloned. Click **Next** to proceed.
4. After opening the local repository or cloning the URL to create a new local repository, the Select a wizard to import projects page of the Import Project wizard is given. As shown below, this page lets you import an Eclipse project, import a project using the New Project wizard, or import a general project.



5. Select the method for importing the project and click **Next** to continue. Depending on which method you chose, you will be directed to the New Project wizard as described in [Creating a Vitis IDE Project](#), or you will be guided through the process of importing an Eclipse project or general project.

# Getting Started with Examples

The Vitis™ core development kit is provided with example designs. These examples can:

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

Every target platform provided within the Vitis IDE contains sample designs to get you started, and are accessible through the project creation flow as described in [Create an Application Project](#).

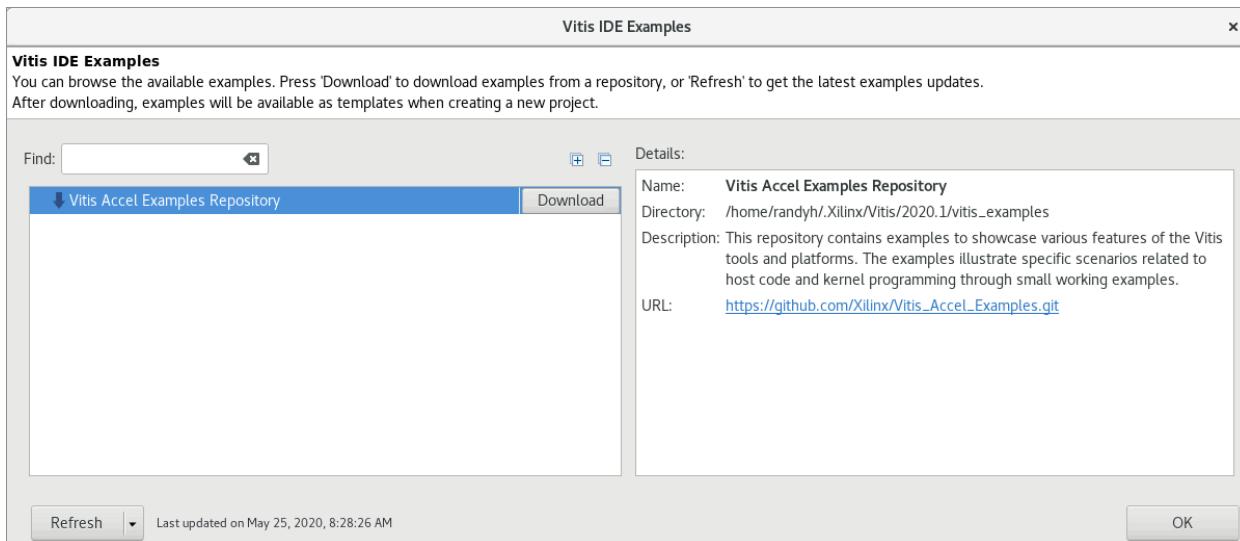
A limited number of sample designs are available in the `<vitis_root>/samples` folder, and many examples are also available for download from the Xilinx® GitHub repository. Each of these designs is provided with a Makefile, so you can build, emulate, and run the code entirely on the command line if preferred.

---

## Installing Examples and Libraries

You can download and install sample applications from the Templates page when working through the New Application Project wizard, or from within an existing project by selecting **Xilinx→Examples**. This displays the Vitis IDE Examples dialog box as shown in the following figure.

Figure 88: Vitis IDE Examples Dialog Box



The left side of the dialog box shows Vitis IDE Examples Repository, and has a download command for each category. The right side of the dialog box shows the directory to where the examples downloaded and the URL from where the examples are downloaded. Click **Download** next to Vitis IDE Examples to download the examples and populate the dialog box.

The command menu at the bottom left of the Vitis IDE Examples dialog box provides two commands to manage the repository of examples:

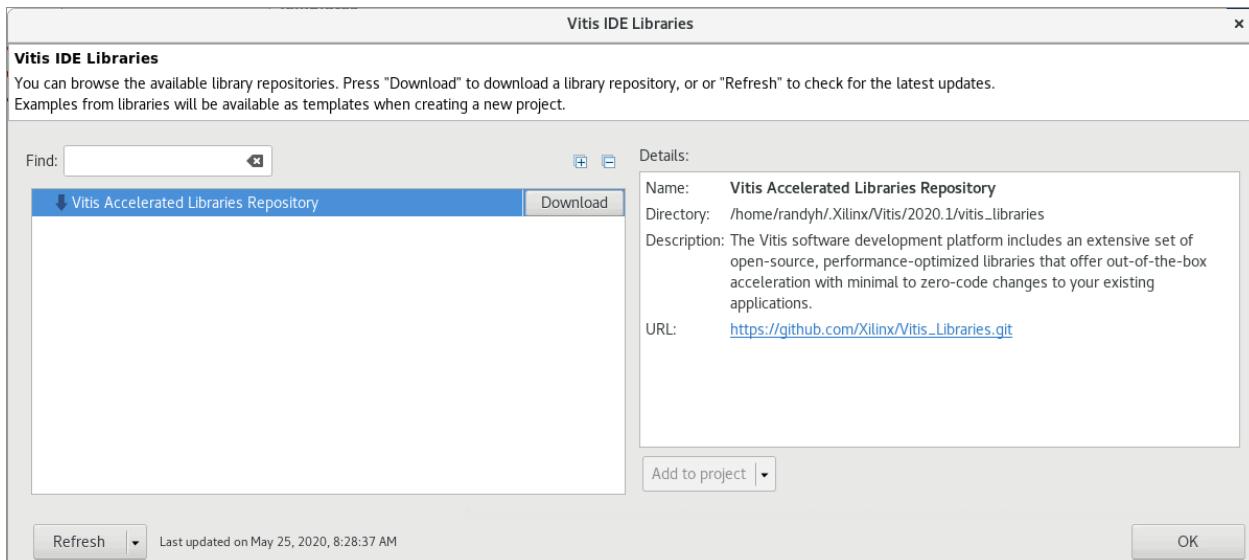
- **Refresh:** Refreshes the list of downloaded examples to download any updates from the [Vitis Examples GitHub](#) repository.
- **Reset:** Deletes the downloaded examples from the `.Xilinx` folder.



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

You can also download Vitis Accelerated Libraries from the **New Application Project** wizard, or by selecting the **Xilinx → Libraries** menu command. For more information on the available libraries and their uses, refer to [Vitis Libraries](#).

Figure 89: Vitis IDE Libraries Dialog Box



## Using Local Copies

While you must download the examples to add templates when you create new projects, the Vitis IDE always downloads the examples into your local `.Xilinx/vitis/<version>` folder:

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

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

```
git clone https://github.com/Xilinx/Vitis_Examples  
<workspace>/examples
```

When you clone the examples using the `git` command as shown above, you can use the example files as a resource for application and kernel code to use in your own projects. However, many of the files use include statements to include other example files that are managed in the makefiles of the various examples. These include files are automatically populated into the `src` folder of a project when the template is added through the New Vitis Project wizard. To make the files local, locate the files and manually make them local to your project.

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

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

# RTL Kernel Wizard

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

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

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

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

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

---

## Launch the RTL Kernel Wizard

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



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

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

To launch the RTL Kernel Wizard from the Vivado IDE:

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

## Using the RTL Kernel Wizard

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

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

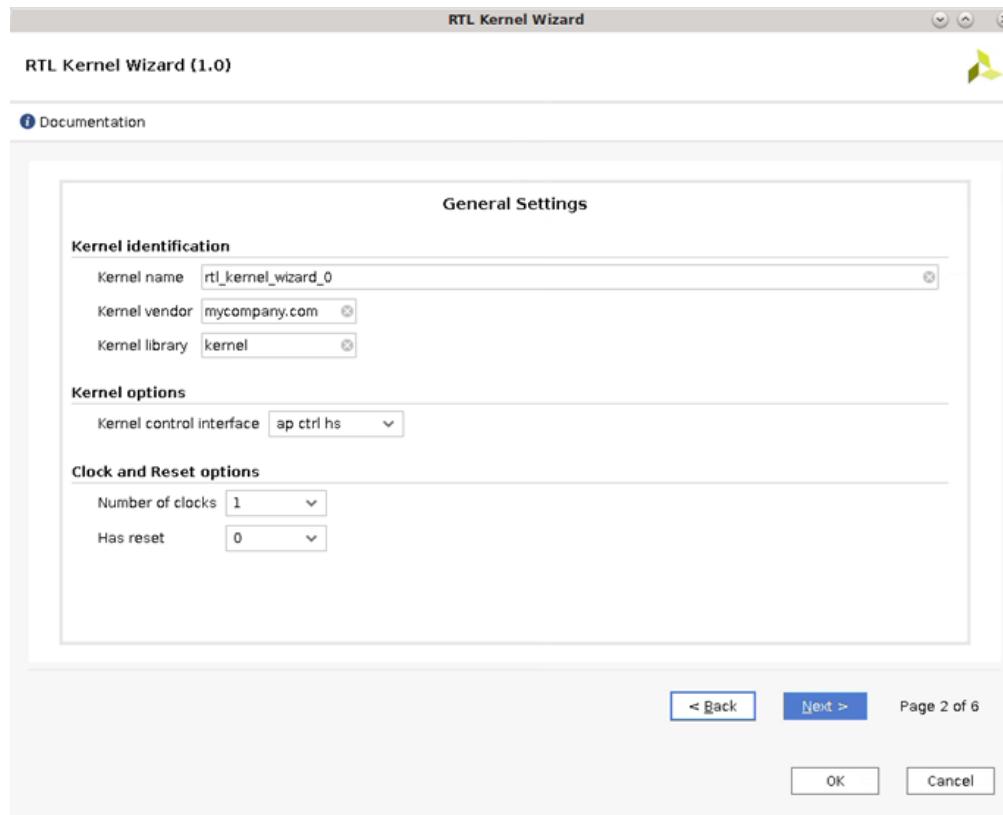
To navigate between pages, click **Next** and **Back** as needed.

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

## General Settings

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

Figure 90: RTL Kernel Wizard General Settings Page



The following are three settings in the General Settings page.

### Kernel Identification

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

### Kernel options

- **Kernel control interface:** There are three types of control interfaces available for the RTL kernel. `ap_ctrl_hs`, `ap_ctrl_chain`, and `ap_ctrl_none`. This defines the `hwControlProtocol` for the `<kernel>` tag as described in [RTL Kernel XML File](#).

## Clock and Reset options

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

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



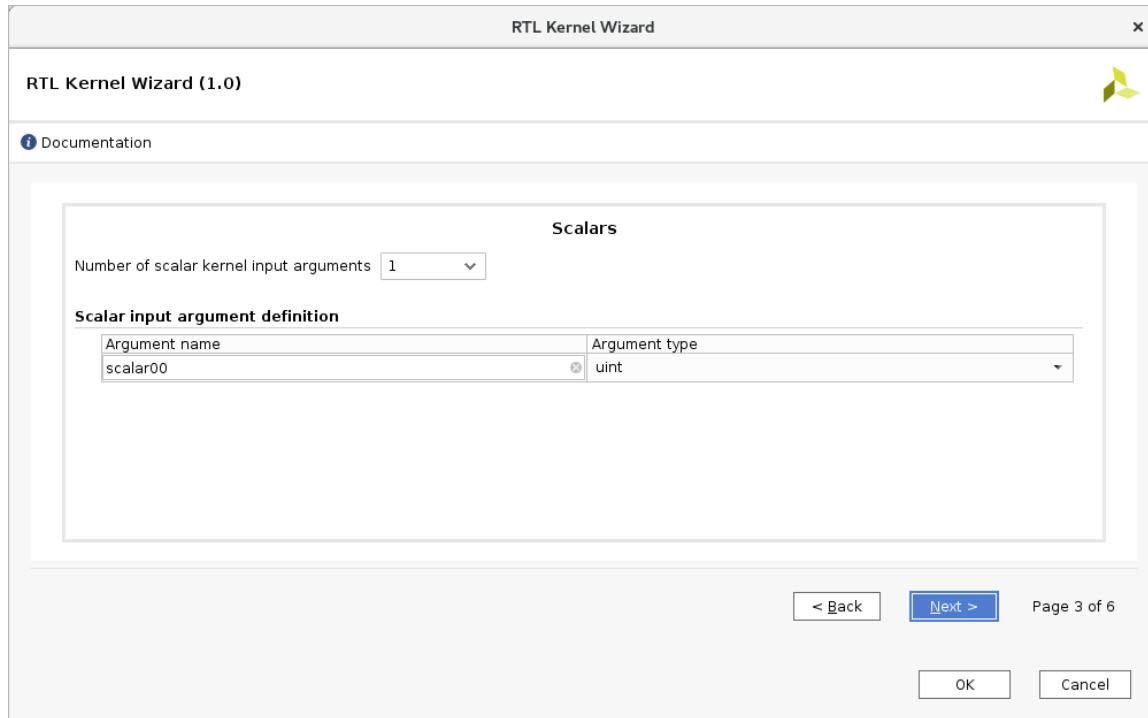
**IMPORTANT!** When designing with multiple clocks, proper clock domain crossing techniques must be used to ensure data integrity across all clock frequency scenarios. Refer to UltraFast Design Methodology Guide for FPGAs and SOCs ([UG949](#)) for more information.

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

## Scalars

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

Figure 91: RTL Kernel Wizard Scalars Page



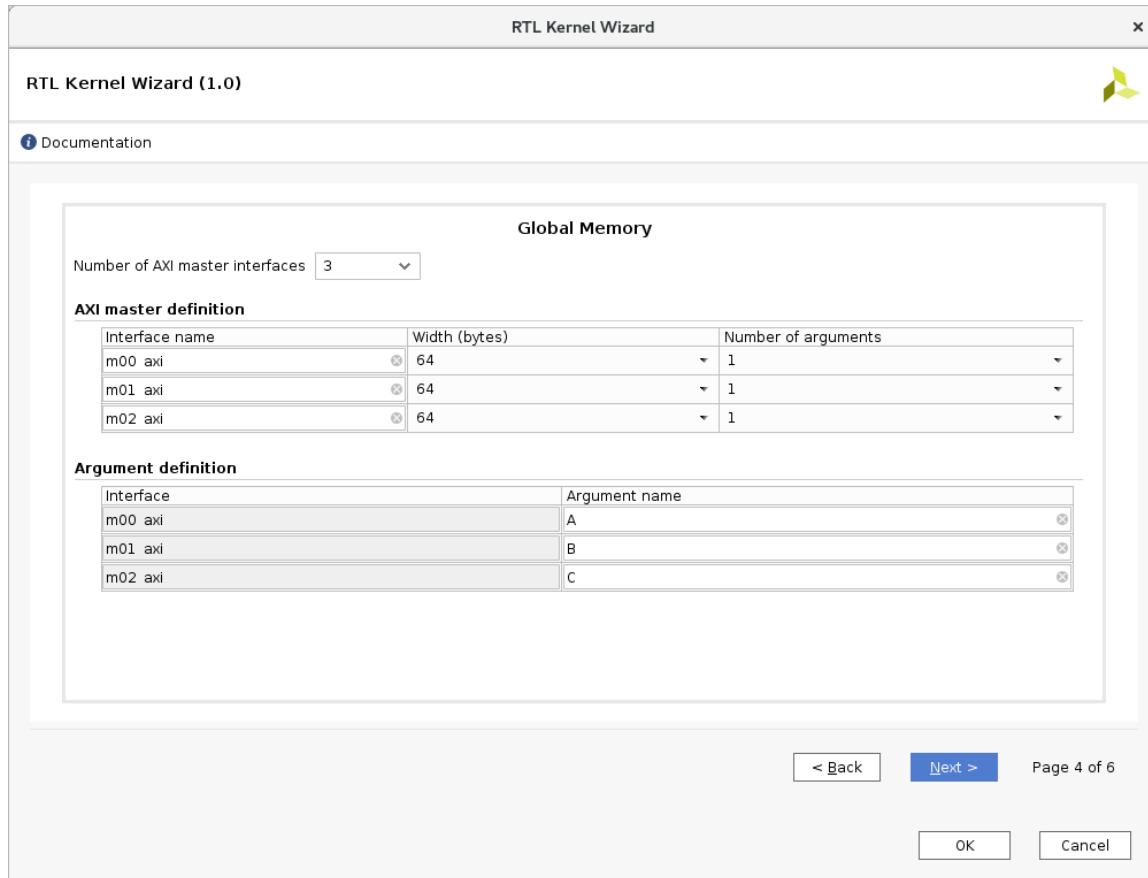
- **Number of scalar kernel input arguments:** Specifies the number of scalar input arguments to pass to the kernel. For each number specified, a table row is generated that allows customization of the argument name and argument type. There is no required minimum number of scalars and the maximum allowed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

The following is the scalar input argument definition:

- **Argument name:** The argument name is used in the generated Verilog control register module as an output signal. Each argument is assigned an ID value. This ID value is used to access the argument from the host software. The ID value assignments can be found on the summary page of this wizard. To ensure maximum compatibility, the argument name follows the same identifier rules as the kernel name.
- **Argument type:** Specifies the data type, and hence bit-width, of the argument. This affects the register width in the generated RTL kernel module. The data types available are limited to the ones specified by the [OpenCL C Specification Version 2.0](#) in "6.1.1 Built-in Scalar Data Types" section. The specification provides the associated bit-widths for each data type. The RTL wizard reserves 64 bits for all scalars in the register map regardless of their argument type. If the argument type is 32 bits or less, the RTL Wizard sets the upper 32 bits (of the 64 bits allocated) as a reserved address location. Data types that represent a bit width greater than 32 bits require two write operations to the control registers.

## Global Memory

Figure 92: RTL Kernel Wizard Global Memory Page



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



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

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

### AXI master definition (table columns)

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

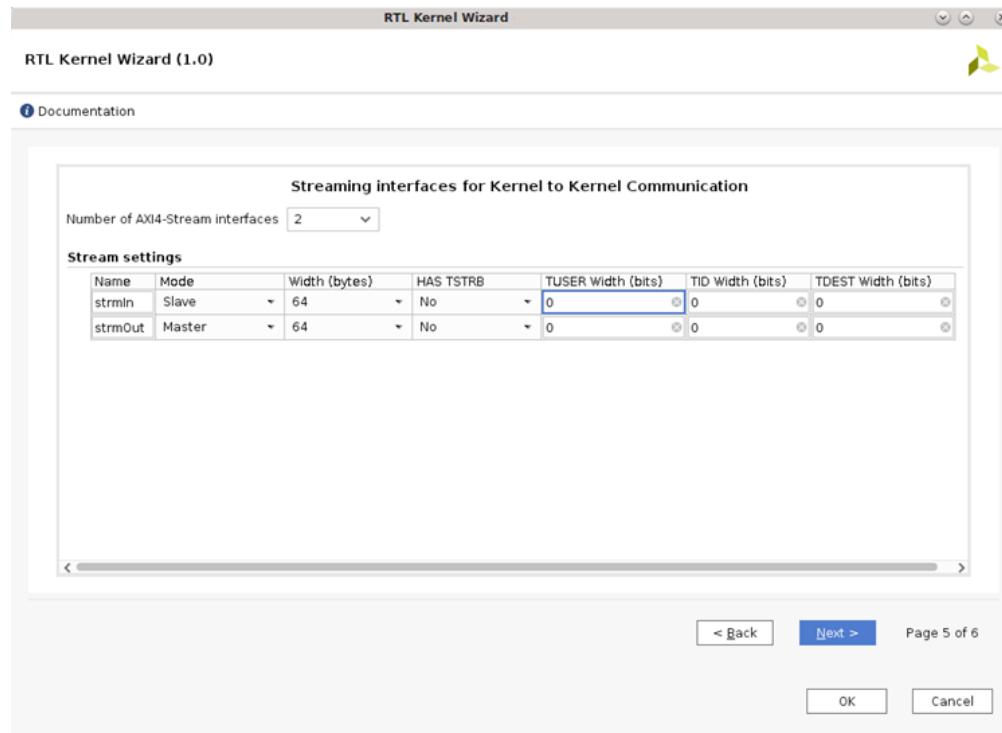
### Argument definition

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

## Streaming Interfaces

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

Figure 93: RTL Kernel Wizard Streaming Interfaces Page



- **Number of AXI4-Stream interfaces:** Specifies the number of AXI4-Stream interfaces that exist on the kernel. A maximum of 32 interfaces can be enabled per kernel. Xilinx recommends keeping the number of interfaces as low as possible to reduce the amount of area consumed.
- **Name:** Specifies the name of the interface. To ensure maximum compatibility, the argument name follows the same identifier rules as the kernel name.
- **Mode:** Specifies whether the interface is a master or slave interface. An AXI4-Stream slave interface is a read-only interface, and an AXI4-Stream master interface is a write-only interface.
- **Width (bytes):** Specifies the TDATA width (in bytes) of the AXI4-Stream interface. This interface width is limited to 1 to 64 bytes in powers of 2.
- **Has TSTRB:** Specifies the TSTRB signal of the AXI4-Stream interface is present on the kernel.
- **TUSER Width (bits):** Specifies the TUSER signal width (in bits) of the AXI4-Stream interface. This interface width is limited to 0 to 512 bits.
- **TID (bits):** Specifies the TID signal width (in bits) of the AXI4-Stream interface. This interface width is limited to 0 to 32 bits.
- **TDEST (bits):** Specifies the TDEST signal width (in bits) of the AXI4-Stream interface. This interface width is limited to 0 to 32 bits.

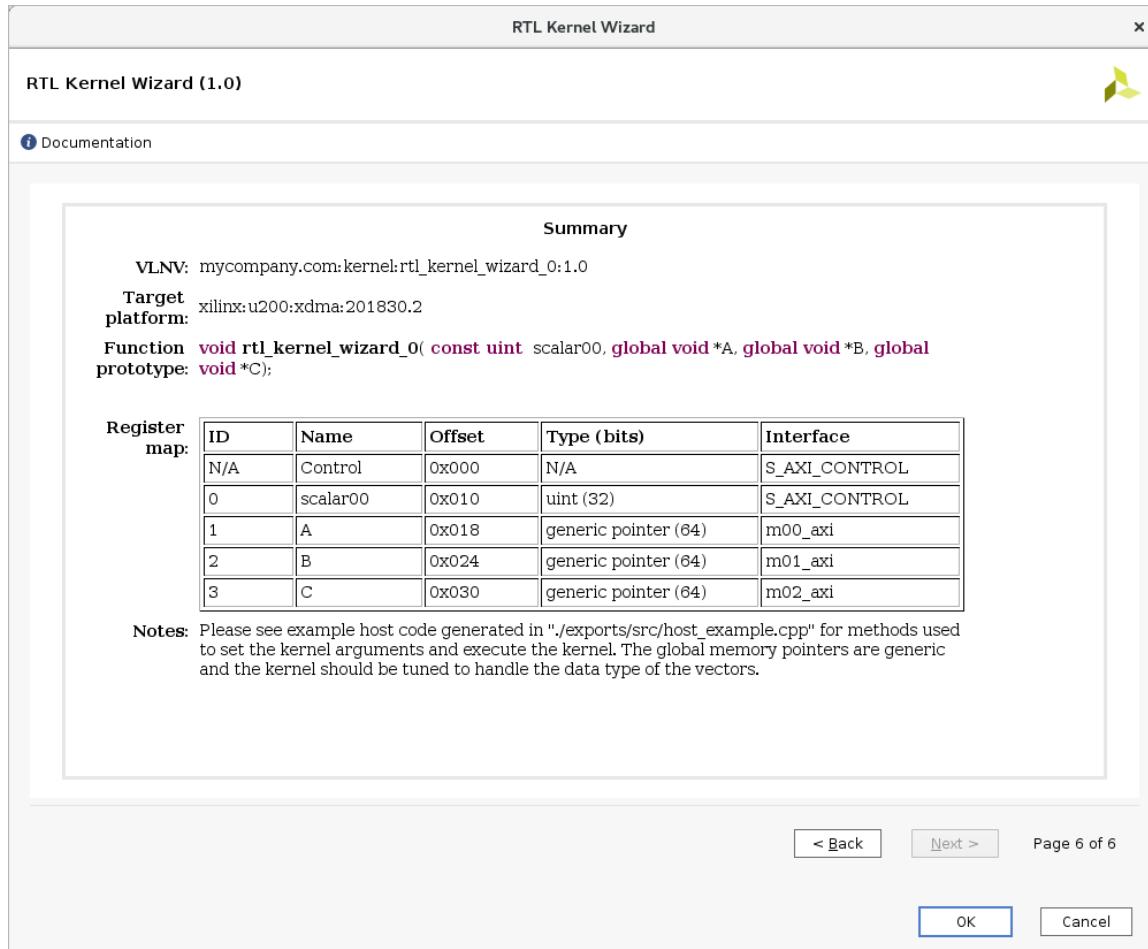
The streaming interface uses the TDATA/TKEEP/TLAST signals of the AXI4-Stream protocol. Stream transactions consists of a series of transfers where the final transfer is terminated with the assertion of the TLAST signal. Stream transfers must adhere to the following:

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

## Summary

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

Figure 94: Kernel Wizard Summary Page



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

## Using the RTL Kernel Project in Vivado IDE

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

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

1. When the Generate Output Products dialog box appears, click **Skip** to close it.
2. Right-click the <kernel\_name>.xci file that is added to the Sources view, and select **Open IP Example Design**.
3. In the Open Example Design dialog box, specify the **Example project directory**, or accept the default value, and click **OK**.



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

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

The example IP project is populated with a top-level RTL kernel file that contains a Verilog example and control registers as described in [RTL Type Kernel Project](#). The top-level Verilog file contains the expected input/output signals and parameters. These top-level ports are matched to the kernel specification file (`kernel.xml`) and can be combined with your RTL code to complete the RTL kernel.

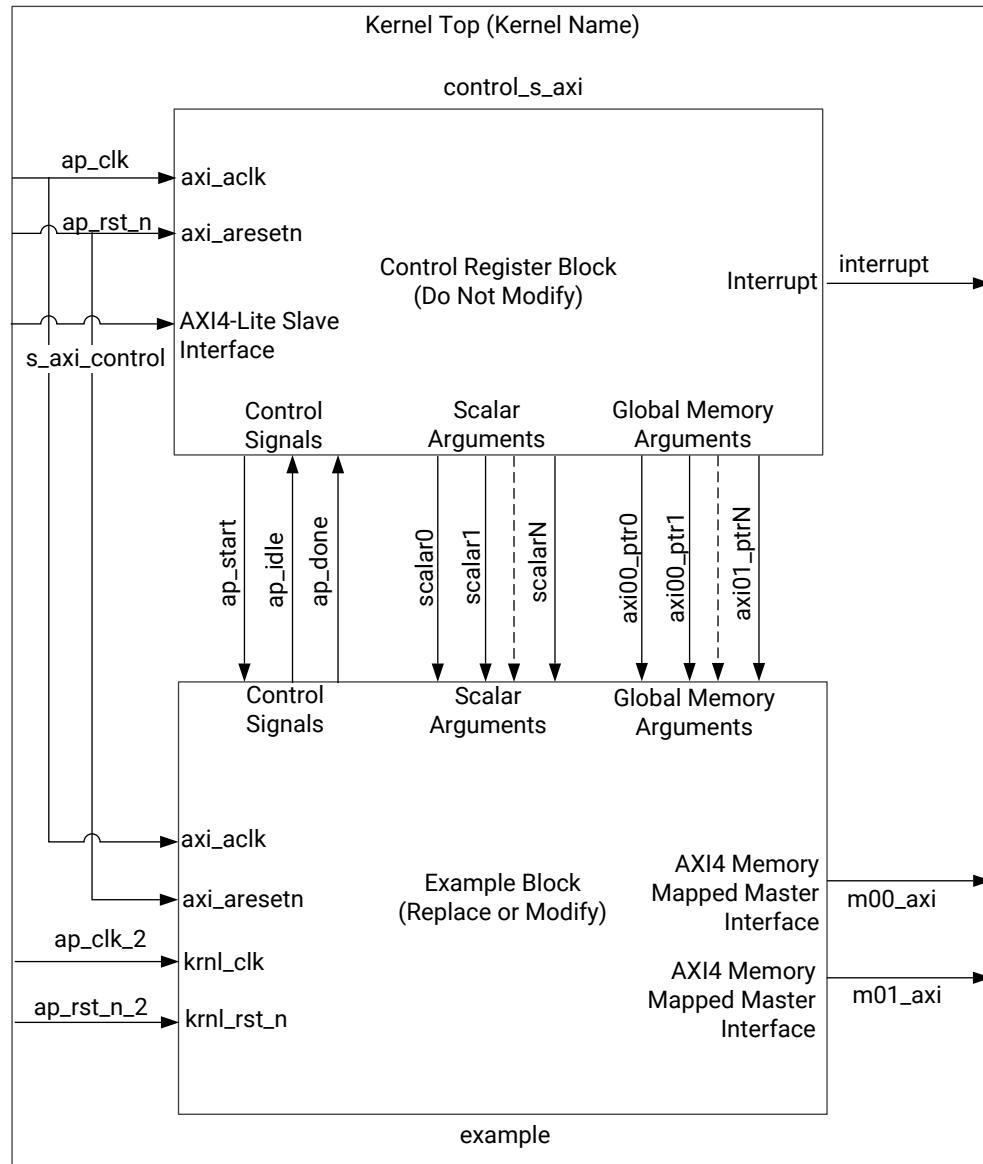
The AXI4 interfaces defined in the top-level file contain a minimum subset of AXI4 signals required to generate an efficient, high throughput interface. Signals that are not present inherit optimized defaults when connected to the rest of the AXI system. These optimized defaults allow the system to omit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

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

## RTL Type Kernel Project

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

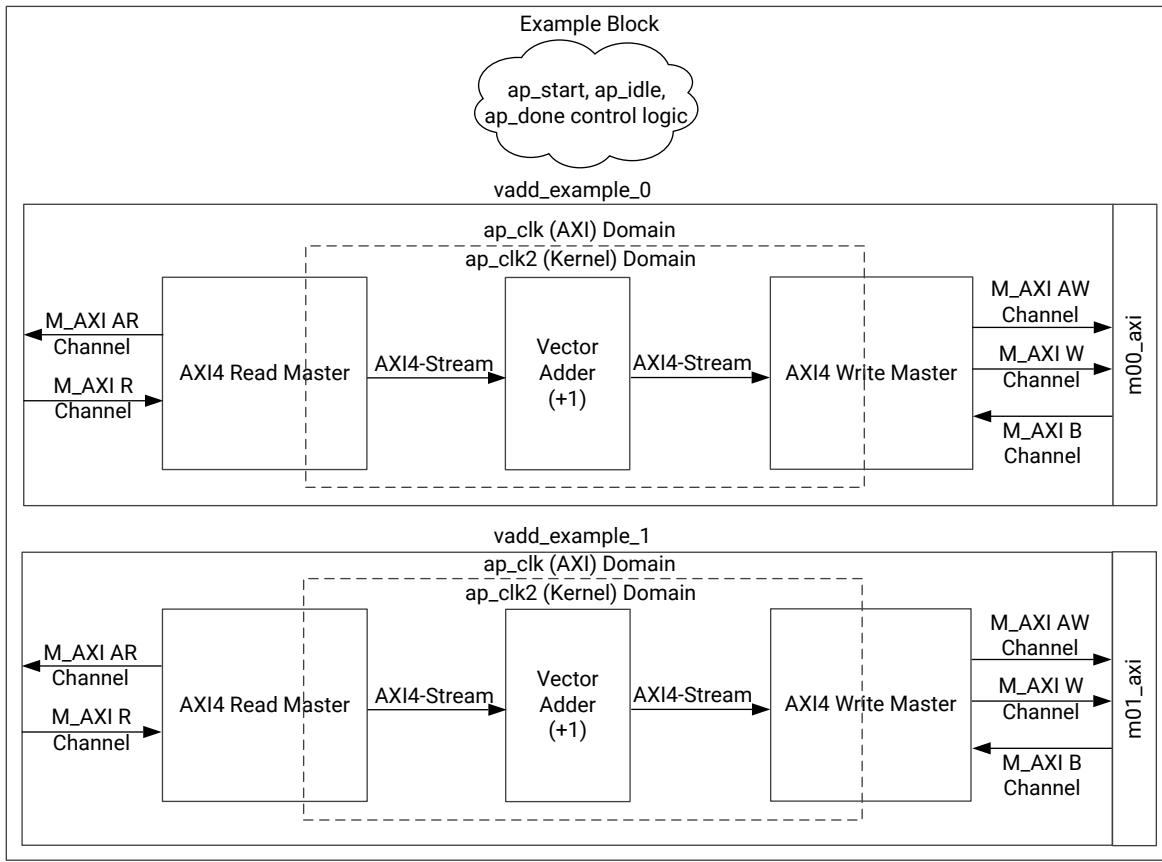
Figure 95: Kernel Type RTL Top



X22079-011019

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

Figure 96: Kernel Type RTL Example



X22080-011019

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

Table 64: RTL Kernel Wizard Source and Test Bench File

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

Table 64: RTL Kernel Wizard Source and Test Bench File (cont'd)

Filename	Description	Delivered with Kernel Type
<kernel_name>_exdes_tb_basic.sv	Simulation test bench	All
<kernel_name>_cmodel.cpp	Software C-Model example for software emulation.	All
<kernel_name>_ooc.xdc	Out-of-context Xilinx constraints file	All
<kernel_name>_user.xdc	Xilinx constraints file for kernel user constraints.	All
kernel.xml	Kernel description file	All
package_kernel.tcl	Kernel packaging script proc definitions	All
post_synth_impl.tcl	Tcl post-implementation file	All
<b>exports directory</b>		
src/host_example.cpp	Host code example	All
makefile	Makefile example	All

## Simulation Test Bench

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

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

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

## Out-of-Context Synthesis

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

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

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

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



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

## Software Model and Host Code Example

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

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

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

## Generate RTL Kernel

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

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

- **Sources only kernel:** Packages the kernel using the RTL design sources directly.
- **Pre-synthesized kernel:** Packages the kernel with the RTL design sources with a synthesized cached output that can be used later on in the flow to avoid re-synthesizing. If the target platform changes, the packaged kernel might fall back to the RTL design sources instead of using the cached output.

- **Netlist (DCP) based kernel:** Packages the kernel as a block box, using the netlist generated by the synthesized output of the kernel. This output can be optionally encrypted if necessary. If the target platform changes, the kernel might not be able to re-target the new device and it must be regenerated from the source. If the design contains a block design, the netlist (DCP) based kernel is the only packaging option available.

Optionally, the **Software Emulation Sources** field lets you specify a software model for your kernel that can be used during software emulation. If the software model contains multiple files, provide a space in between each file in the Source files list, or use the GUI to select multiple files using the **CTRL** key when selecting the file.

After you click **OK**, the kernel output products are generated. If the pre-synthesized kernel or netlist kernel option is chosen, then synthesis can run. If synthesis has previously run, it uses those outputs, regardless if they are stale. The kernel Xilinx Object (XO) file is generated in the `exports` directory of the Vivado kernel project.

At this point, you can close the Vivado kernel project. If the Vivado kernel project was invoked from the Vitis IDE, you can add the example host code called `host_example.cpp` and the kernel object (XO) file into the `./src` folder of the appropriate sub-project of the application project in the Vitis IDE. Refer to [Adding Sources](#) for more information.

## Modifying an Existing RTL Kernel Generated from the Wizard

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



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

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



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

# Using Vitis Embedded Platforms

This section contains the following chapters:

- [Vitis Embedded Platforms](#)
- [Using Vitis Embedded Platforms](#)
- [Creating Embedded Platforms in Vitis](#)

# Vitis Embedded Platforms

---

## Introduction

The Vitis™ unified software platform provides a Platform+Kernel structure to help developers focus on the applications. The decoupling of platform and kernel helps make the platform reusable with multiple kinds of kernels and vice versa.

Xilinx provides pre-built platforms for Alveo™ and embedded evaluation boards. You are free to create your own embedded platforms or customize the Xilinx embedded platforms.

The Vitis software platform is an environment for creating embedded software and accelerated applications on heterogeneous platforms based on FPGAs, Zynq®-7000 SoCs, and Zynq® UltraScale+™ MPSoCs. This document focuses on using embedded platform for Zynq UltraScale+ MPSoC.

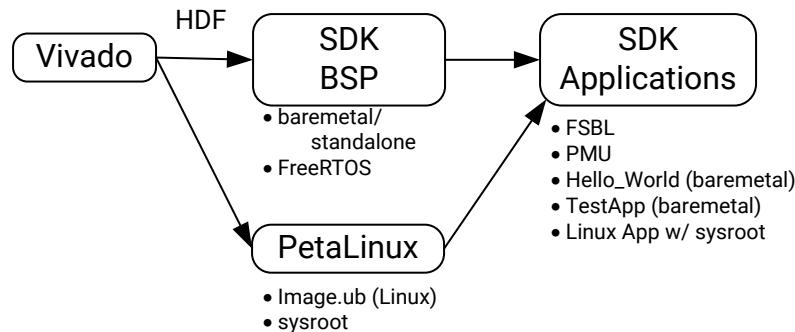
---

## Platform Types

The Vitis target platforms can be customized with unique hardware and software components. There are two general types of platforms: fixed platforms and extensible platforms. Fixed platforms support embedded software development and are a direct analog to the hardware definition file that was previously used for software development with the Xilinx SDK tool. Extensible platforms support the application acceleration development flow, and includes hardware for supporting acceleration kernels, controlling AI Engine for Versal® ACAP, and software for a target running Linux and the Xilinx Runtime (XRT) library. For more information on the XRT library, see <https://github.com/Xilinx/XRT>.

The following figure shows the traditional SDK flow for embedded software application development. A Xilinx Hardware Design File (HDF) is exported from the Vivado® Design Suite. It is used by SDK for board support package (BSP) generation and creating software applications that apply the BSP.

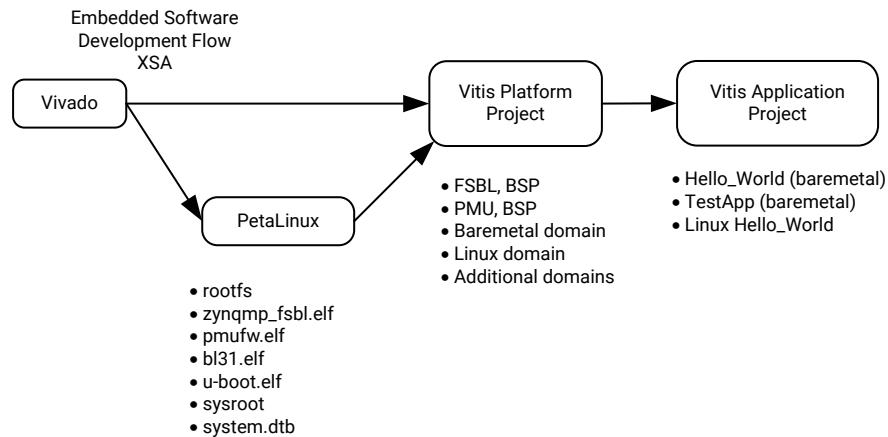
Figure 97: Pre-2019.2 SDK Flow



X233443-102119

The following figure shows the Vitis embedded software development flow that replaces the SDK flow beginning with the 2019.2 release. The hardware specification is now contained in the Xilinx Shell Archive (XSA) and is exported from a Vivado design, but is formatted differently from HDF and uses the .xsa filename extension.

Figure 98: Vitis Embedded Software Development Flow

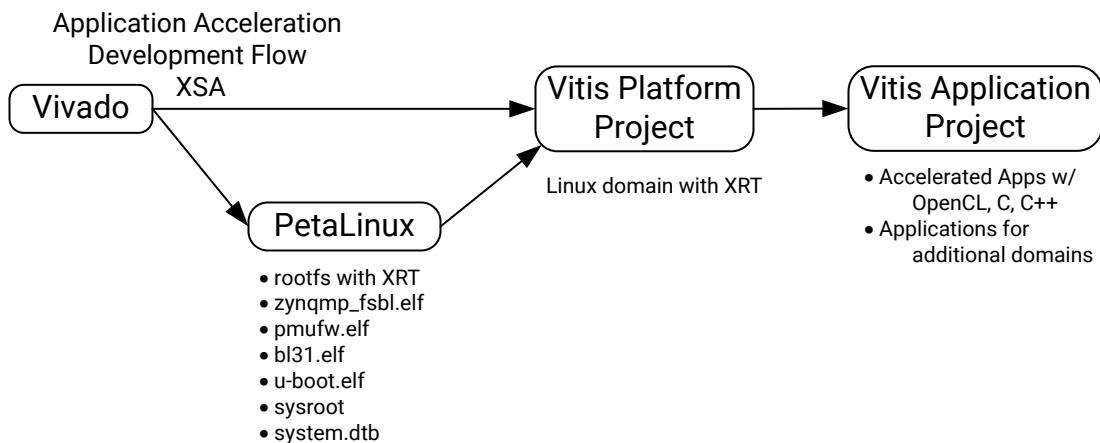


X27497-120522

The Vitis core tools create a platform, BSP, and software boot components such as the FSBL and PMU firmware for the fixed-XSA, and are associated with the Vitis platform. Software applications targeting the fixed-platform can be developed with the Vitis Embedded Software Development flow. Fixed-platforms do not require Linux and the XRT library, but can target processor domains running Baremetal and RTOS operating systems as well. See the [Vitis Embedded Software Development Flow Documentation](#) in the [Vitis Unified Software Platform Documentation \(UG1416\)](#) for more information.

In the Vitis Application Acceleration Development flow, the Vivado Design Suite is also used to generate and write an extensible-XSA, containing additional IP blocks and metadata to support kernel connectivity. The following figure shows the acceleration software development flow.

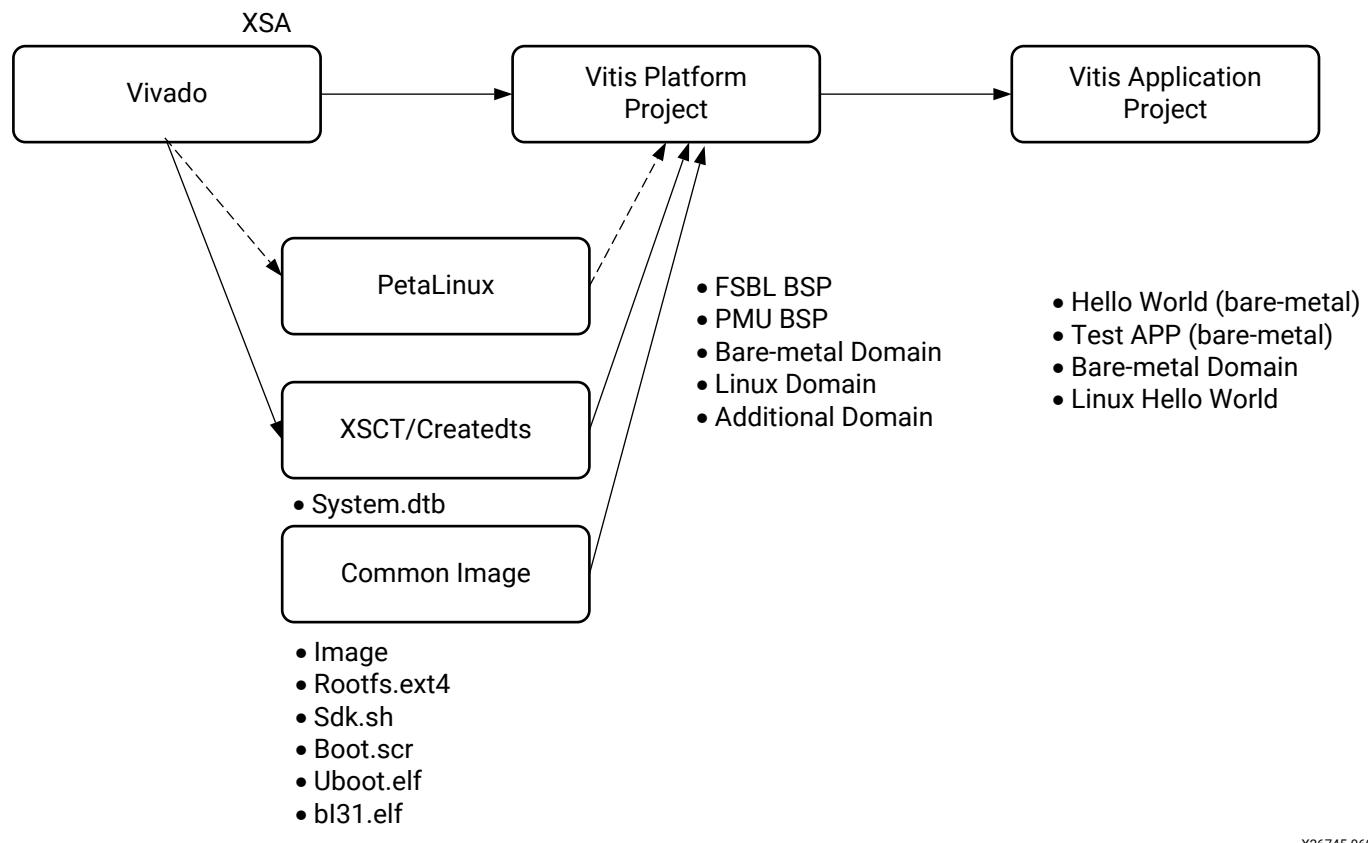
Figure 99: Vitis Acceleration Kernel Flow



X23345-062320

The following figure shows the Vitis embedded software development flow that use common image and createdts to generate software components and leverage PetaLinux as an alternative beginning with the 2022.1 release.

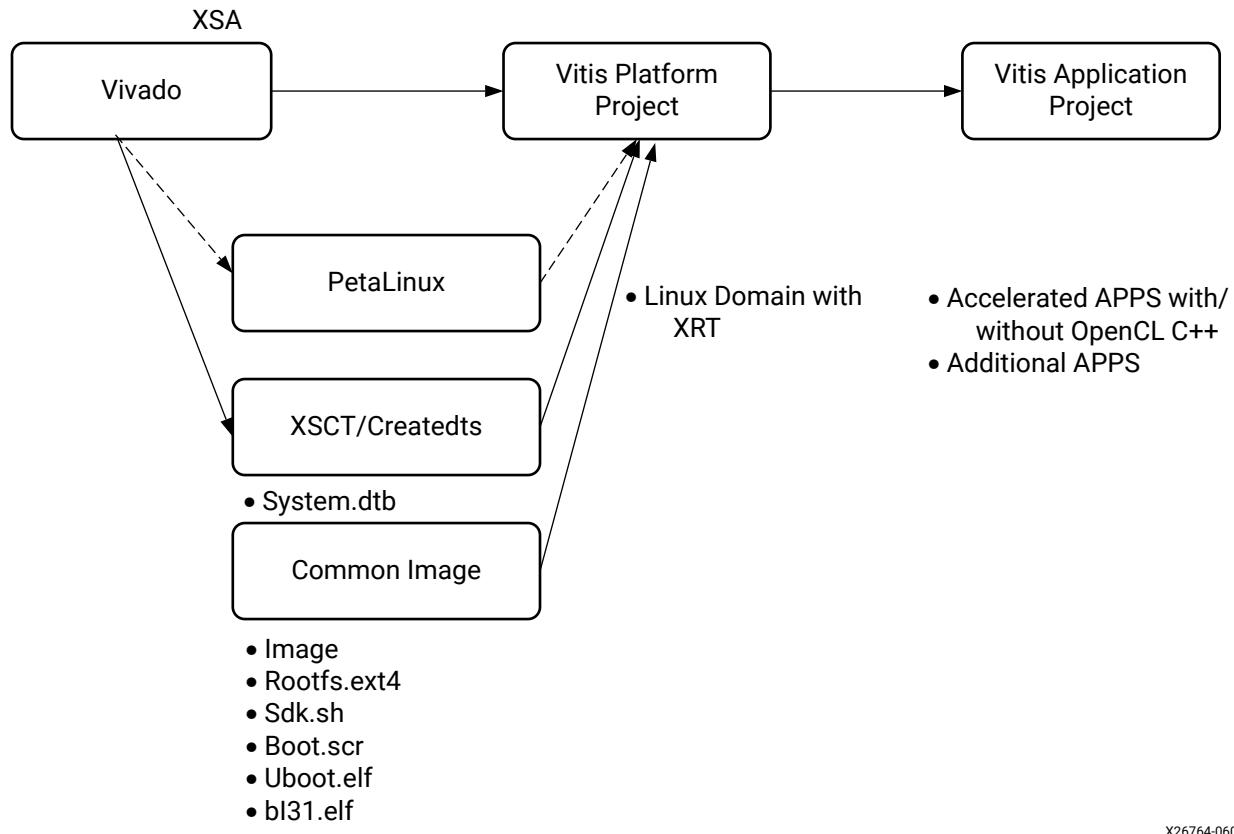
Figure 100: Vitis Embedded Software Development Flow



X26745-060222

In the Vitis Application Acceleration Development flow, the Vivado Design Suite is also used to generate and write an extensible-XSA, containing additional IP blocks and metadata to support kernel connectivity. You start to use common image and createdts to generated DTS for software components and use PetaLinux as an alternative. The following figure shows the acceleration software development flow beginning with the 2022.1 release.

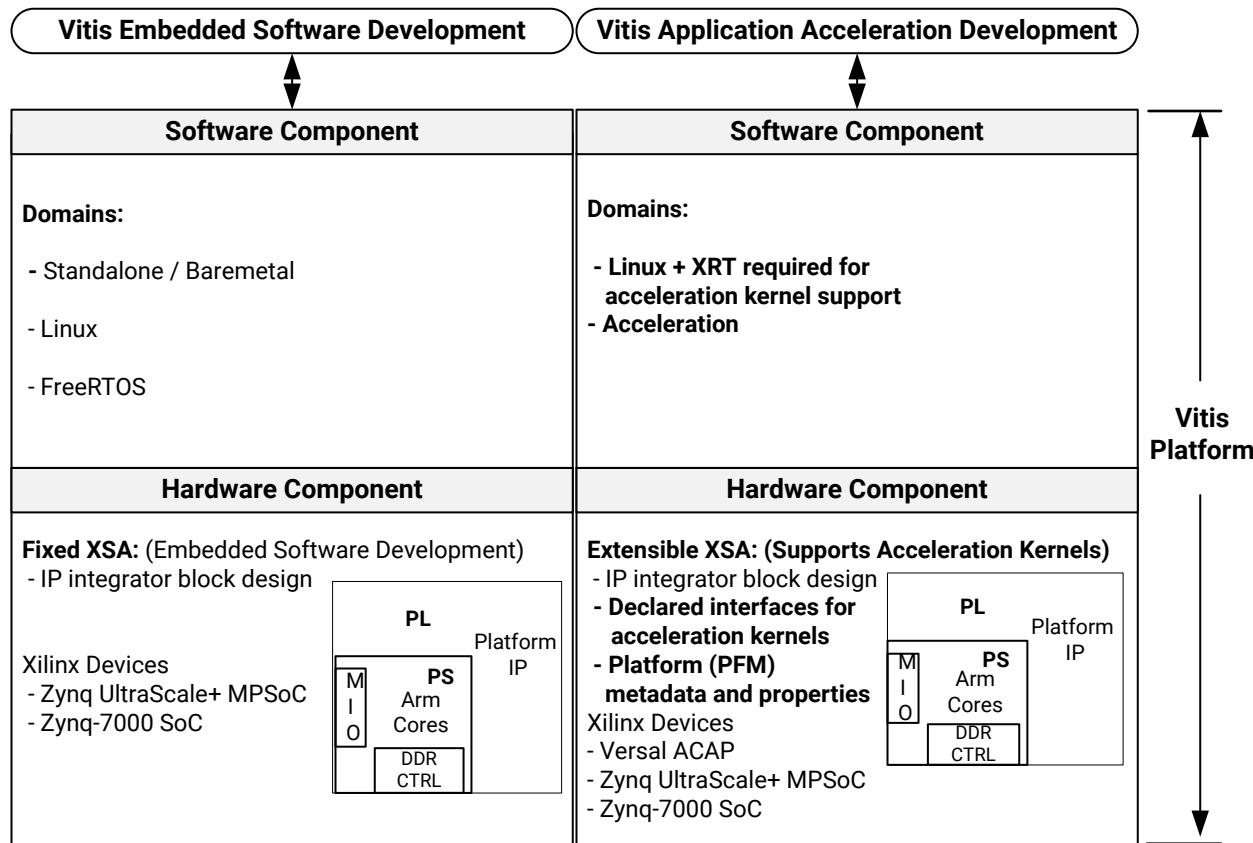
Figure 101: Vitis Acceleration Kernel Flow



The Vitis core tool supports application development in multiple languages (OpenCL™, C, C++) but the applications must target a Vitis platform. A target platform consists of hardware and software components as shown in the following figure. The target platform view on the left side of the page is for the Vitis embedded software development flow, whereas the right side of page shows a platform that supports acceleration kernels. The differences include acceleration kernel requirements of a target with Linux + XRT, metadata, and kernel interface declarations.

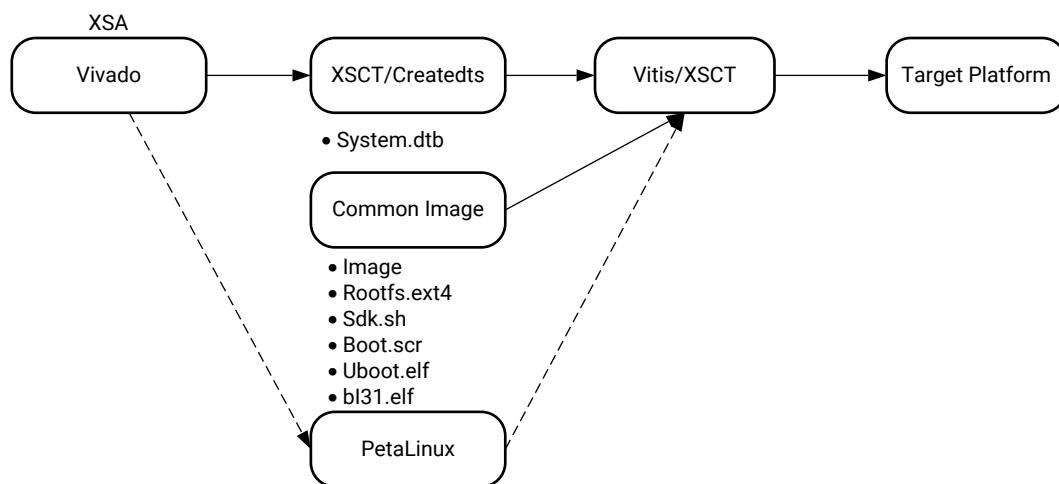
**Note:** Custom platform generation sources are available in [https://github.com/Xilinx/Vitis\\_EMBEDDED\\_Platform\\_Source](https://github.com/Xilinx/Vitis_EMBEDDED_Platform_Source).

Figure 102: Vitis Target Platforms



X23346-110920

Figure 103: Vitis Platform Project Flow



X26749-060222

# Platform Naming Convention

The platform name is used when creating acceleration applications in Vitis or targeting platforms when using the Vitis compiler (v++). Pre-built platform images also use the same file name and directory name.

Pre-built Vitis embedded platforms use the following naming convention.

```
<Vendor>_<Board>_<Feature>_<Supported Vitis Tool Version>_<Release  
Version>
```

Where:

- **<Vendor>**: The board vendor. For all Xilinx-created pre-built platforms, use `xilinx`.
- **<Feature>**: The special function of this platform. For example:
  - `base` indicates that it connects all possible resources for you to use in an acceleration application.
  - `DFX` indicates that it supports Xilinx Dynamic Function eXchange (DFX).
- **<Supported Vitis Tool Version>**: The specific version of the Vitis development platform that the platform is designed for. This also indicates the version of the Vivado® Design Suite tools that the pre-built platform is created by.
- **<Release Version>**: The release version of the platform. The first version is 1.

For example, the following platform names follow the naming convention:

- `xilinx_zcu102_base_202020_1`
- `xilinx_zcu104_base_202020_1`
- `xilinx_zc706_base_202020_1`
- `xilinx_zcu102_base_dfx_202020_1`

**Note:** Platform source code uses a git branch for versioning. The directory name is `<Board>_<Feature>` (for example, `zcu102_base`). The platform generated from the source in [https://github.com/Xilinx/Vitis\\_Embedded\\_Platform\\_Source](https://github.com/Xilinx/Vitis_Embedded_Platform_Source) has the name `xilinx_zcu102_base_202020_1`.

# Embedded Platform Components and Architecture

A platform is the starting point of your Vitis design. Vitis applications are built on top of the platforms.

An embedded platform includes a hardware platform and a software platform.

## Hardware Platform

The hardware platform is the static, unchanging portion of your hardware design. It includes the Xilinx Support Archive (XSA) file exported from the Vivado Design Suite.

The hardware platform describes platform hardware setup and the acceleration resources that can be used by acceleration applications, for example, Input and output interfaces, clocks, AXI buses, and interrupts. Vitis adds kernels and infrastructure modules to the hardware design as needed to facilitate data movement. Acceleration kernels can share data with platform IPs, but cannot change or modify them. For information about setting up the hardware platform, refer to [Installing Xilinx Runtime and Platforms](#).

## Software Platform

The software platform is the environment that runs the software to control acceleration kernels for acceleration applications. It includes the domain setup and boot components setup.

By default, all Xilinx pre-built platforms have a Linux domain that has enabled Xilinx Runtime (XRT) so that acceleration applications can run on this environment. The pre-built binaries for Linux kernel image and rootfs are located in a separate download file on the PetaLinux download page. See the "Common images for Embedded Vitis platforms" section of the [Xilinx download center](#). Because the device tree is unique to each platform, it is provided as a component with the Linux XRT domain inside the platform.

Linux Domain Components must be provided when there is a Linux domain in the embedded platform. These components can be generated by PetaLinux, Yocto, or third-party frameworks. Because these components can be shared across all Xilinx demo boards for the given FPGA family, a common Linux component image generated by PetaLinux is provided for Zynq-7000 SoC and Zynq UltraScale+ MPSoC devices.

The following Linux images can be downloaded from the PetaLinux download page:

- **Root File System (RFS):** Includes binaries, libraries, and setups for a Linux file system. In the Xilinx-provided common rootfs, XRT has been installed so that acceleration application can run on this Linux environment.
- **Kernel Image:** A compiled Linux kernel. The common kernel image provided by Xilinx includes most Xilinx peripheral drivers.
- **Sysroot:** Used for cross compilation. It provides the libraries to be linked when compiling applications for a target system.

**Note:** Optionally, you can pack Linux domain components into embedded platforms. When creating a Linux application in the Vitis IDE, the Linux domain components in the platform setting will be the default and initial settings if they have been set in the platform. You can overwrite these settings with components installed elsewhere.

Xilinx pre-built embedded platforms and pre-built common Linux components are provided in separate download files. You can regenerate the common Linux components from the platform source files hosted on the [Vitis Embedded Platform GitHub repository](#) by setting the environment variable `COMMON_RFS_KRNL_SYSROOT=FALSE` before running `make`.

---

## Installing Embedded Platforms

Pre-built Vitis embedded platforms must be downloaded from the Xilinx website:

- Download required Vitis platforms from the [Vitis Embedded Platforms download page](#).
- Download common Linux components from the "Common images for Embedded Vitis platforms" section of the [PetaLinux download page](#).

After you download the embedded platforms, there are three ways to include them in your project:

- Extract the pre-built platforms to `/opt/xilinx/platforms`.
- Extract the pre-built platforms to any folder. Set the `PLATFORM_REPO_PATHS` environment variable to the folder path.

```
export PLATFORM_REPO_PATHS=/path/to/platforms
```

- Extract the pre-built platforms to any folder. In the Vitis IDE, select **Xilinx → Add Custom Platform** and select the folder path.

Common Linux components can be extracted to any folder. When you create a Linux application, you will specify the path for the components in the Vitis New Application Project wizard.

# Using Vitis Embedded Platforms

---

## Packaging Images

A new packaging stage is added to the Vitis™ compiler (v++) in 2021.2.

In 2021.2, v++ has three stages:

- `-c` or `--compile` to compile acceleration kernels
- `-l` or `--link` to link acceleration kernels with platform logic.
- `-p` or `--package`, the command `v++ --package` generates both `boot.bin` and the `sd_card` image files

The `--package` command supports both initramfs and Ext4 rootfs images.

In the Vitis IDE, the package stage is automatically called during the build process. You can add additional package options in the system project detail page by double-clicking the `.sprj` file. Package log files, command configuration files, and output files are stored in the `package` directory under the `Emulation-SW`, `Emulation-HW` or `Hardware` directories.

In command line mode, you can pass in package options as v++ options or configuration files. For more detailed information about the `v++ --package` option, refer to the [v++ Command](#) or the `v++ -help` command, and the Xilinx [Vitis Acceleration Examples GitHub repository](#).

## Packaging Images with Ext4 rootfs in the Vitis IDE

When `ext4 rootfs` is provided to the Vitis IDE, the generated `sd_card.img` file includes the following:

- The `xclbin` file for PL kernel
- The host application
- The Linux kernel image
- The device tree
- The u-boot configuration file: `boot.scr`
- The `ext4 rootfs` in the Ext4 partition

To package an image with Ext4 rootfs in the Vitis IDE:

1. Select **File→New→Application Project** to create a new application project in the Vitis IDE.
2. Select the platform (for example, `xilinx_zcu102_base_202120_1`), and click **Next**.
3. Provide a name for the application project (for example, `vadd`)
4. For the System Project selection, select **Create New**.
5. For the Target processor, select the processor that can run the Linux domain (for example, `psu_cortexa53 SMP`), and click **Next**.
6. In the Domain page, select **xrt** and provide the following application settings:
  - Sysroot path (for example, `xilinx-zynqmp-common-v2021.2/sysroots/cortexa72-cortexa53-xilinx-linux`)
  - Root FS (for example, `xilinx-zynqmp-common-v2021.2/rootfs.ext4`)
  - Kernel Image (for example, `xilinx-zynqmp-common-v2021.2/Image`)
7. Click **Next**.
8. Select an application template (for example, **Vector Addition**) and click **Finish**.
9. Select the system project and click the Build () button to build the project.
10. Verify that the `sd_card.img` file was created in the package directory under the Emulation-SW, Emulation-HW or Hardware directory.



**TIP:** To change the path for `sysroot`, `rootfs`, or `kernel` after the application project has been created, double-click the `.sprj` file and change the path in the Options dialog box.

## Packaging Images with initramfs rootfs in the Vitis IDE

When `initramfs rootfs(rootfs.cpio)` is provided to the Vitis IDE, the generated `sd_card.img` includes the following:

- The `xclbin` file for PL kernel
- The host application
- The Linux kernel image
- The device tree
- The `boot.scr`
- The `init.sh`, `platform_desc.txt`, and `initramfs rootfs` in FAT32 partition

**Note:** The `sd_card.img` file does not contain the ext4 partition.

1. In the Vitis IDE, select **File→New→Application Project** to create a new application project.

2. Select the platform (for example, `xilinx_zcu102_base_202010_1`), and click **Next**.
3. Provide the application project name (for example, `vadd`).
4. Select **Create New**.
5. For the target processor, select the processor that can run the Linux domain (for example, `psu_cortexa53 SMP`), and click **Next**.
6. In the Domain page, select the `xrt` domain and provide the application settings as follows:
  - a. Sysroot path (for example, `your_linux_component_dir/sysroots/aarch64-xilinx-linux`)
  - b. Root FS (for example, `your_linux_component_dir/rootfs.cpio.gz.u-boot`)
  - c. Kernel Image (for example, `your_linux_component_dir/Image`)
7. Click **Next**.
8. Select the application template (for example, **Vector Addition**).
9. Select the system project and click the **Build** ( ) button to build the project.
10. Verify that the `sd_card.img` file was created in the package directory under the Emulation-SW, Emulation-HW or Hardware directory.

**Note:** The common Linux components package does not provide initramfs rootfs. For more information about generating initramfs rootfs, refer to *PetaLinux Tools Documentation: Reference Guide* ([UG1144](#)).



**TIP:** To change the path for `sysroot`, `rootfs`, or `kernel` after the application project has been created, double-click the `.sprj` file and change the path in the Options dialog box.

## Writing Images to the SD Card

You can use the Vitis unified software platform accelerated flow to target an embedded platform. This facilitates packaging and creating an SD image with RootFS as an EXT4 partition, because initramfs uses Double Data Rate SDRAM (DDR SDRAM) for file system storage. It limits the real usable DDR memory for Linux kernel and applications when the file system size increases. It cannot retain RootFS changes after reboot.

To write EXT4 RootFS to an SD Card:

1. Prepare an SD card binary image file with FAT32 partition for boot and EXT4 partition for RootFS.
2. Write SD card images to the SD card. You can use various tools to do this, such as [Etcher](#) on Windows or `dd` command on Linux.

**Note:** Refer to Xilinx [Answer Record 73711](#) for detailed information about these tools.

There are various ways to prepare an SD card image. You can use the v++ package tool to generate it, or use an open source tool. The v++ package tool generated `sd_card.img` has two partitions:

- **FAT32 partition:** 1 GB size, initialized with the kernel image provided by common Linux components.
- **EXT4 partition:** 2 GB size, initialized with RootFS provided by common Linux components.

To make the pre-built SD card image boot, you must copy the following boot components to the FAT32 partition:

- `pre-built/BOOT.BIN`
- `boot.scr`, `system.dtb`, `init.sh`, and `platform_desc.txt` in the `xrt/image` directory

The pre-built SD card image can be used for evaluation usage and by Windows users. It does not require Vitis or PetaLinux to be installed.

**Note:** The `v++ --package` with Ext4 partition is not supported on Windows.

**Note:** `init.sh` sets up the environment variable `XILINX_XRT` and copies the `platform_desc.txt` file to `/etc/xocl.txt`. You must manually run this after Linux boots up before running any acceleration applications.

---

## Configuring the PL Kernel in DFX Platforms and Non-DFX Platforms

The Xilinx Dynamic Function eXchange (DFX) feature can change some blocks of PL function while keeping other areas of PL working, allowing you to configure PL kernels on the fly. To use the DFX feature, when the `xclbin` file is generated, configure it with your host application. The new kernels in the `xclbin` take effect immediately without requiring a reboot.

For platforms without DFX features, PL kernel must be packed into `boot.bin`. Copy it to the FAT32 partition on your SD card and reboot the system. Then, configure the `xclbin` file with your host application.

The `xclbin` file contains both bit files for PL kernel and metadata to describe these kernel features and connections. Programming the `xclbin` file on DFX platforms loads the bit file and metadata; programming on non-DFX platforms only loads the metadata.

---

# Running an Acceleration Application on the Board

If you are using the common Linux components that are provided by Xilinx perform the following to run an acceleration application on the platform:

1. To the SD card, write the `sd_card.img` generated by the Vitis compiler command `v++ -- package`.
2. Boot the board.
3. Run the command `cd /mnt/sd-mmcbblk0p1/`.
4. Run the command `source init.sh`.
5. Run acceleration application. For example, for vector addition, run `./vadd ./binary_container_1.xclbin`.

Acceleration application uses Xilinx Runtime (XRT) to communicate with acceleration kernels. To set up the environment for XRT, run `init.sh`. This command does the following:

- It sets the environment variable `XILINX_XRT` to `/usr` to allow the application to find the XRT environment.
- It copies `platform_desc.txt` to `/etc/xocl.txt` to inform XRT which platform it is running on.

**Note:** This was done automatically for embedded platforms in the 2019.2 release of Vitis. Because automatically running `init.sh` may introduce security breaches, common Linux rootfs did no run `init.sh` by default.

**Note:** If the `sd_card.img` file has already been written to the SD card and you are only updating the application, you can save time in the debugging phase by copying all files from <Vitis System Project>/Hardware/package/`sd_card` to FAT32 partition on SD card to replace existing files. The Ext4 partition does not change in `sd_card.img`.

---

# Software Package Management in PetaLinux rootfs

The package management feature is new for the Vitis 2020.1 release. All PetaLinux rootfs software packages are hosted on <https://petalinux.xilinx.com/sswreleases/rel-v2020/feeds>. You can install these software packages to rootfs when running Linux on the target board as long as the board has Internet access.

To use this feature, you must enable package manager DNF in rootfs. The rootfs in Xilinx-provided pre-built Linux components provides the DNF package management features by default.

To set up a package feed URL:

1. Visit the package feed repo directory <https://petalinux.xilinx.com/sswreleases/rel-v2020/generic/rpm/repos/>.
2. Download the repo file that matches your SoC device to target board.

```
# Example: ZCU102 uses ZU9EG devices
wget http://petalinux.xilinx.com/sswreleases/rel-v2020/generic/rpm/repos/
zynqmp-generic_eg.repo
# Example: ZCU104 uses ZU7EV devices
wget http://petalinux.xilinx.com/sswreleases/rel-v2020/generic/rpm/repos/
zynqmp-generic_ev.repo
```

3. Copy the downloaded repo file to `/etc/yum.repos.d/`.
4. Clean the cache.

```
dnf clean all
```

To manage packages, use the DNF package manager:

- **Listing available packages:** Use the command `dnf repoquery`.
- **Installing packages from a Xilinx repository:** Use the command `dnf install <pkg name>`.
- **Installing packages from a local package file:** Use the command `dnf install <pkg file name>`.
- **Installing packages to sysroot:**

When packages are installed on the rootfs of a running board, target has the latest binaries and libraries. When cross compiling on host is needed, these libraries must be added to host side sysroot.

A `sysroot_overlay` script is provided in XRT to extract RPM and update sysroot. This script will extract RPM libraries and include a file update in sysroot.

Besides XRT, this script supports all RPMs for various software packages.

- **Getting the `sysroot_overlay.sh`:** Use the command `wget https://github.com/Xilinx/XRT/blob/master/src/runtime_src/tools/scripts/sysroots_overlay.sh`.

The `sysroot` command description is:

```
./sysroots_overlay.sh --sysroot --rpms-file
```

Where:

- `--sysroot` is the sysroot to be overlaid.
- `--rpms-file` is the RPMs file that contains the RPM file paths to be overlaid.

## Examples

The following example is a command to install updated XRT to the common sysroot:

```
./sysroots_overlay.sh -s sysroots/aarch64-xilinx-linux/ -r $PWD/rpm.txt
```

This example shows the contents of an `rpm.txt` file:

```
./xrt-dev-202010.2.6.0-r0.aarch64.rpm  
./xrt-202010.2.6.0-r0.aarch64.rpm
```

**Note:** This script works only for the local RPMs. You must download RPMs to your host machine to install them to the common sysroot.

# Creating Embedded Platforms in Vitis

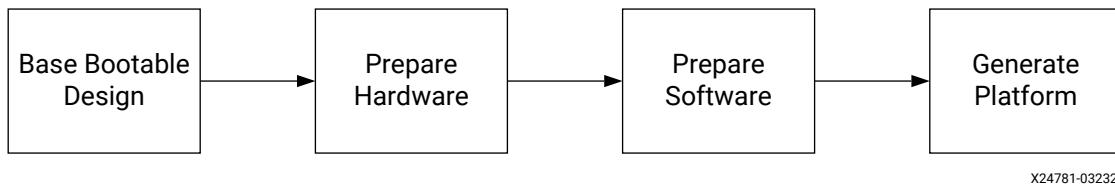
---

## Platform Creation Basics

In the Vitis™ environment acceleration application development flow, the project is divided into two distinct elements: the platform and the processing subsystem. The platform contains essential IP blocks (such as PS for SoCs, NoC and AI Engine for Versal® ACAPs) and board interface IP blocks (such as high-speed I/Os and memory controllers). The processing subsystem contains the application-specific part of the system and can be composed of both programmable logic and AI Engine blocks. This approach promotes separation of concerns, facilitates concurrent development, and encourages reusability. The application developer is insulated from the low-level details of the platform and can focus on the specifics of the processing subsystem. The platform developer can focus on system bring-up and tuning I/O performance without having to worry about the processing subsystem. This means that the application developer can integrate the subsystem on different platforms, and a platform can be reused with different processing subsystems.

Xilinx provides pre-built platforms for Alveo™ cards and embedded evaluation boards. You can download these platforms from the [Xilinx Download Center](#). Efficiently leveraging the decoupling of platforms and subsystems is central to the methodology and the productivity gains offered by the Vitis environment. For embedded designs, Xilinx recommends a parallel development process where the application team starts working on the subsystem using a Xilinx pre-built platform while the platform team works independently on bringing-up the custom platform. Rapid progress can be made by working in this manner. Using a pre-built platform means that the subsystem can be developed, integrated, and tested independently using a pre-verified, known-good foundation. After the subsystem is in a sufficiently advanced and stable state, the subsystem can be integrated with appropriate versions of the custom platform. Overall, this approach greatly streamlines the system integration process.

The following figure shows how to create a customized embedded platform.

**Figure 104: Platform Creation**

To create a platform, you must have a base bootable design as a starting point. This design can be a Xilinx base platform design, an existing working design, or a design created from scratch. The following base components must be included in your base bootable design:

- A base hardware design exported from Vivado® Design Suite
- A base software design that includes Linux kernel, root file system, and device tree

After you have working hardware and board through a Vivado® Design Suite design, converting it into a Vitis environment platform requires adding properties to the base components to meet the requirements of the Vitis environment. In general, platform creation consists of the following steps:

1. Add hardware interface parameters and interrupt support in your Vivado® Design Suite project and export the XSA.
2. Update the software platform components to enable application acceleration software stacks (enable XRT, update device tree, and so on).
3. Package and generate the platform using XSCT commands or the Vitis IDE.

**Note:** The platform creation process is usually iterative, and multiple versions of the platform are created throughout the course of the project. In the early phases of the project, you can create platforms with a reduced set of features to facilitate testing of the processing subsystem on the board. In the later phases of the project, you might need to iterate on the platform to respond to specification changes or to improve overall QoR.

The Vitis environment uses the properties in the hardware project to recognize the resources in the platforms and link kernels to the platforms. The Vitis environment uses the software stacks to take control of the kernels.

For details on Vitis environment embedded platform creation, see the [Vitis Unified Software Platform Documentation](#). For step-by-step instructions, see the [Vitis Platform Creation tutorial](#).

---

## Platform Creation Requirements

The base design you created in a Vitis platform is static after the platform creation process is complete.

Vitis does modify parameters based on certain IPs (for example, SmartConnect, NoCs) by adding additional master/slave interfaces. In some situations, PS/CIPS interfaces can also be modified and Versal® ACAP and AI Engine IP is instantiated in the platform.

The following table shows the workflows to validate the base system on your board.

**Table 65: Platform Workflows**

Workflow	Development	Validation
Basic board bring-up	Processor basic parameter setup.	Standalone Hello world and Memory Test application run properly.
Advanced hardware setup	Enable advanced I/O in Processing System (such as USB, Ethernet, Flash, PCIe®, or RC). Add I/O related IP in PL (such as MIPI, EMAC, or HDMI_TX). Add non-Vitis IP (such as AXI BRAM Controller, or Video Processing Subsystem (VPSS) IP).	If these IP have standalone drivers, test them.
Base software setup	Create PetaLinux project based on hardware platform. Enable kernel drivers. Configure boot mode. Configure rootfs.	Linux boots up successfully. Peripherals work properly in Linux.

## Base Component Requirements

Every hardware platform design must contain a Processing System IP block from the IP catalog.

- Versal ACAP, Zynq® UltraScale+™ MPSoC, and Zynq-7000 SoC devices are supported.
- MicroBlaze™ processors are not supported for controlling acceleration kernels, but can be part of the base hardware.

---

# Creating an Embedded Platform

## Generate XSA file

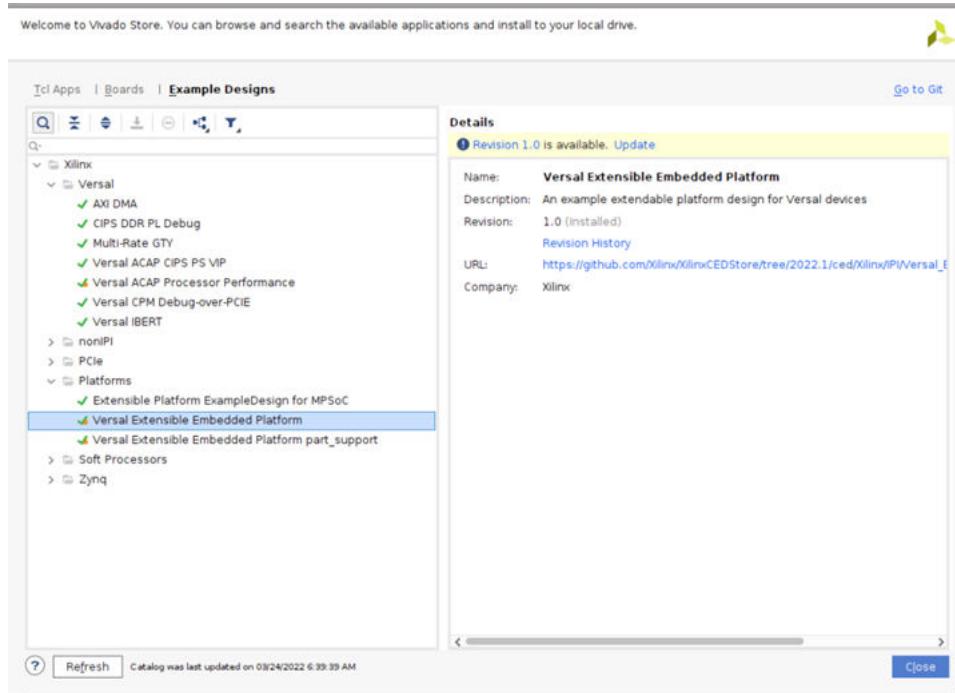
1. Leverage CED example design to create hardware platform and export XSA file.
2. Add hardware interface based on fixed platform and export XSA file.

### **CED Example Usage**

#### **Download Example Platform**

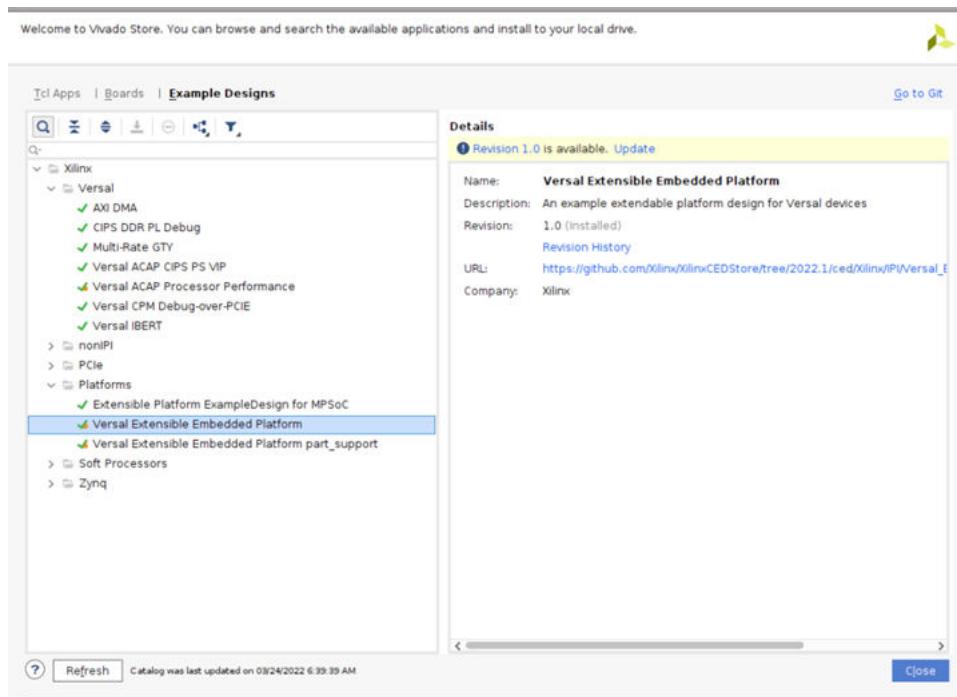
1. Launch Vivado.

2. Click **Tools**→**Vivado Store**.
3. Click **OK** to agree to download open source examples from web.
4. Select **Platform**→**Versal Extensible Embedded Platform** and click the download button on the tool bar.
5. Click **Close** after the installation is complete.



**Note:** In this example, Versal Extensible Embedded Platform is used. Select the appropriate example design you need.

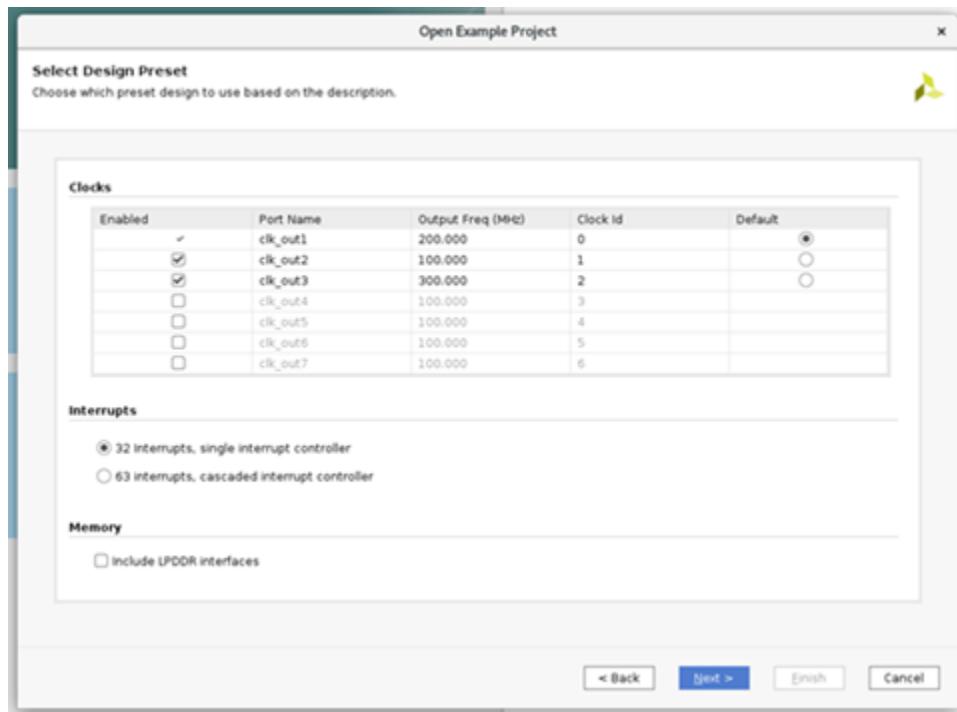
6. Click **Close** after the installation is completed.



**Note:** In this example Versal Extensible Embedded Platform is used. Select the appropriate example design you need.

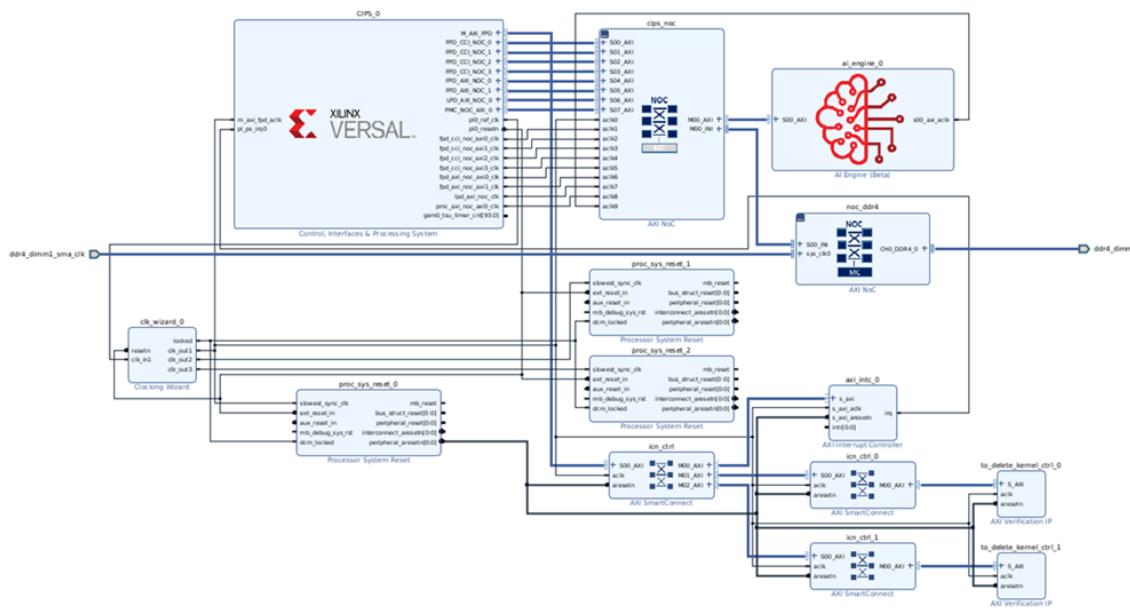
### Create CED Example Design

1. Click **File → Project → Open Example**.
2. Select **Versal Extensible Embedded Platform** in Select Project Template window.
3. Input **project name** and **project location**. Keep **Create project subdirectory** checked. Click **Next**.
4. Select target board in Default Part window. In this example **Versal VCK190 Evaluation Platform** has been selected. Click **Next**.



5. Configure Clocks Settings. You can enable more clocks, update output frequency and define default clock in this view. The default settings are being used in this example.
6. Configure Interrupt Settings. You can choose how many interrupt should this platform support. 63 interrupts mode will use two AXI\_INTC in cascade mode. The default settings are being used in this example.
7. Configure Memory Settings. By default the example design will only enable DDR4. If you enable LPDDR4, it will enable both DDR4 and LPDDR4. The default settings are being used in this example.
8. Click **Next**.
9. Review the new project summary and click **Finish**.
10. After a while, you will see the design example has been generated.

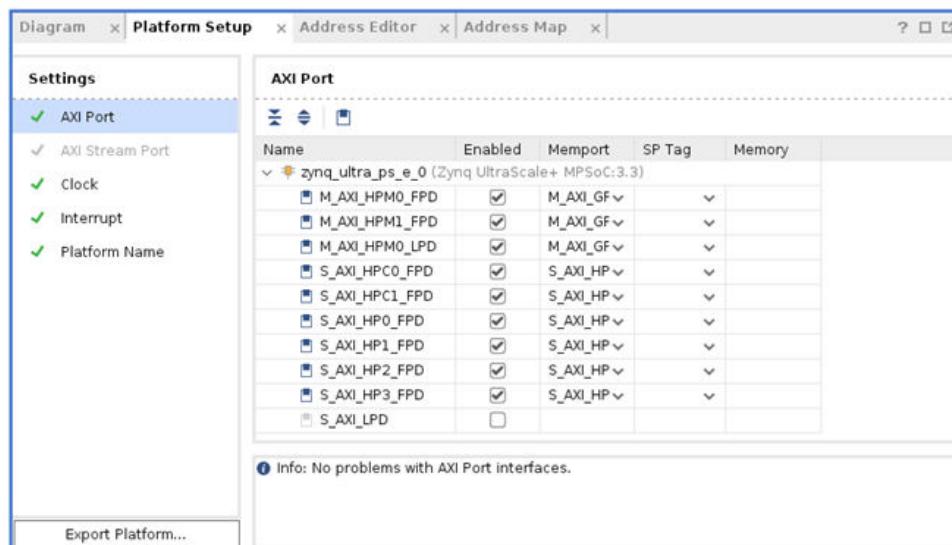
The generated design instantiated AI Engine, enabled DDR4 controller and connected them to CIPS. It also provides one interrupt controller, three clocks and the associated synchronous reset signals.



## Review the Versal Extensible Platform Example Platform Setup

### 1. Review the AXI port settings.

- In **axi\_noc\_ddr4**, **S01\_AXI** to **S27\_AXI** are enabled. **SP Tag** is set to **DDR**.



- In **icn\_ctrl\_0** and **icn\_ctrl\_1**, **M01\_AXI** to **M15\_AXI** are enabled. In **icn\_ctrl**, **M03\_AXI** and **M04\_AXI** are enabled. **Memport** is set to **M\_AXI\_GP**. **SP Tag** is empty. These ports provide the AXI master interfaces to control PL kernels. In the block diagram, **icn\_ctrl\_0** and **icn\_ctrl\_1** connects to an AXI Verification IP because the AXI SmartConnect IP requires a load. The AXI Verification IP is used here as a dummy.

The screenshot shows the Vitis IDE interface. On the left, the 'Settings' tab is selected, displaying a list of checked items under the 'AXI Port' section: 'AXI Stream Port', 'Clock', 'Interrupt', and 'Platform Name'. To the right is a table titled 'AXI Port' with columns for 'Name', 'Enabled', 'Memport', 'SP Tag', and 'Memory'. The table lists various AXI ports, including S01\_AXI through S15\_AXI, M01\_AXI through M04\_AXI, and several CIPS ports like cips\_noc, noc\_ddr4, and icn\_ctrl.

Name	Enabled	Memport	SP Tag	Memory
> cips_noc (AXI NoC:1.0)				
> noc_ddr4 (AXI NoC:1.0)				
> icn_ctrl (AXI SmartConnect:1.0)				
icn_ctrl_0 (AXI SmartConnect:1.0)				
S01_AXI				
S02_AXI				
S03_AXI				
S04_AXI				
S05_AXI				
S06_AXI				
S07_AXI				
S08_AXI				
S09_AXI				
S10_AXI				
S11_AXI				
S12_AXI				
S13_AXI				
S14_AXI				
S15_AXI				
M01_AXI	✓	M_AXI_GP	▼	▼
M02_AXI	✓	M_AXI_GP	▼	▼
M03_AXI	✓	M_AXI_GP	▼	▼
M04_AXI	✓	M_AXI_GP	▼	▼

## 2. Review the Clock settings.

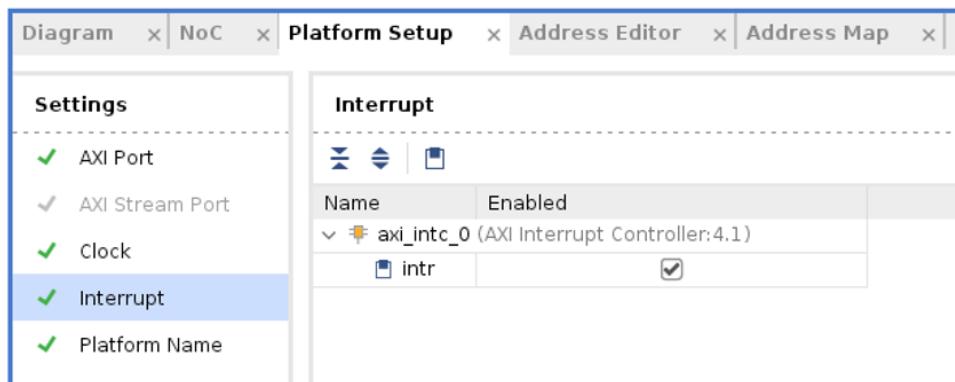
- In Clock tab, clk\_out1, clk\_out2, clk\_out3 from clk\_wizard\_0 are enabled with id {0,1,2}, frequency {200 MHz, 100 MHz, 300 MHz}. clk\_out1 is the default clock. V++ linker will use this clock to connect the kernel if there aren't any clocks specified in the link configuration. The Proc Sys Reset property is set to the synchronous reset signal associated with each clock.

The screenshot shows the 'Clock' table in the Vitis IDE. It lists various clock sources, primarily from the CIPS\_0 system. The table includes columns for 'Name', 'Enabled', 'ID', 'Is Default', 'Proc Sys Reset', 'Status', and 'Frequ...'. The 'clk\_wizard\_0' entry shows three output clocks: clk\_out1 (ID 0, Proc Sys Reset /proc\_sys\_reset\_0, 200 MHz), clk\_out2 (ID 1, Proc Sys Reset /proc\_sys\_reset\_1, 100 MHz), and clk\_out3 (ID 2, Proc Sys Reset /proc\_sys\_reset\_2, 300 MHz). The 'p10\_ref\_clk' entry is also listed.

Name	Enabled	ID	Is Default	Proc Sys Reset	Status	Frequ...
CIPS_0 (Control, Interfaces & Processing System:3.0)						
p10_ref_clk						
clk_wizard_0 (Clocking Wizard:1.0)						
clk_out1	✓	0	●	/proc_sys_reset_0	fixed	▼ 200 MHz
clk_out2	✓	1	○	/proc_sys_reset_1	fixed	▼ 100 MHz
clk_out3	✓	2	○	/proc_sys_reset_2	fixed	▼ 300 MHz

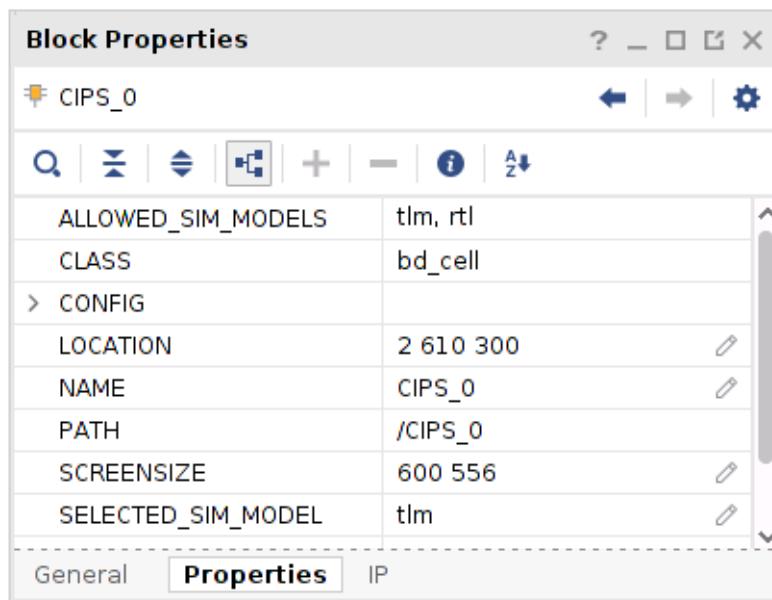
## 3. Review the Interrupt Tab.

- In Interrupt tab, In0 to In31 port of xlconcat is enabled.



4. Review the Simulation Model.

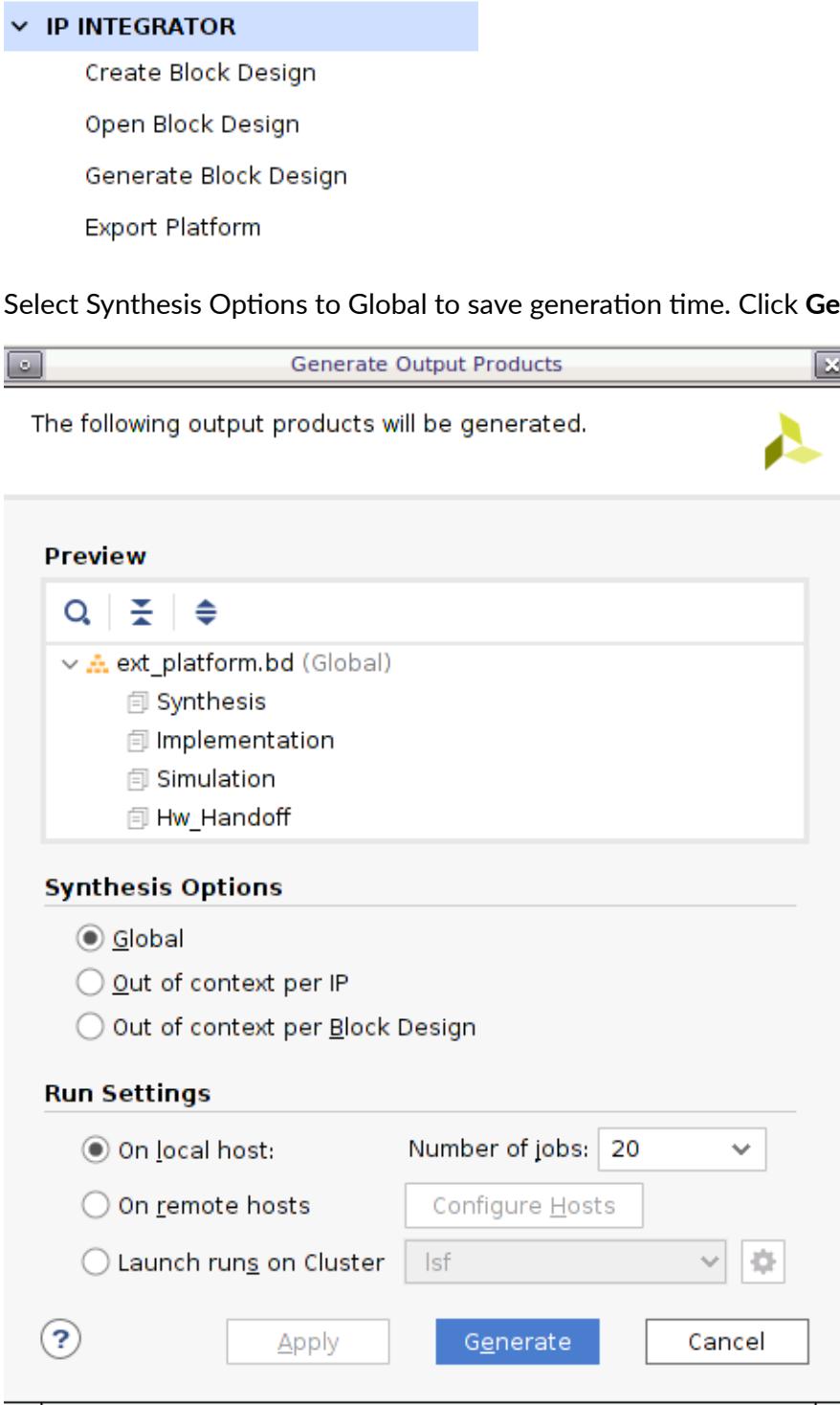
- In Vivado Integrated Design Environment (IDE), select the **CIPS** instance. Check the Block Properties window. In Properties tab, it shows **ALLOWED\_SIM\_MODELS** is **tlm**, and **rtl**, **SELECTED\_SIM\_MODEL** is **tlm**. This means this block supports two simulation models. In this example, **tlm** model was selected.



- Review the simulation model property for NoC and AI Engine in the block diagram.

### Export Hardware XSA

- Generate Block Diagram.
  - Click **Generate Block Diagram** from Flow Navigator window.



2. Export hardware platform with the following scripts.
  - a. Click **File** → **Export** → **Export Platform**. Alternative ways are: Flow Navigator window: **IP Integrator** → **Export Platform**, or the **Export Platform** button on the bottom of Platform Setup tab.

- b. Click **Next** on Export Hardware Platform page.
- c. Select **Hardware**. If there are any IP that don't support simulation, the Hardware XSA and Hardware Emulation XSA need to be generated separately. Click **Next**.
- d. Select **Pre-synthesis**, as a DFX platform is not being created. Click **Next**.
- e. Input the Name and click **Next**.
- f. Update the file name and click **Next**.
- g. Review the summary and click **Finish**.

### **Export Hardware Emulation XSA**

A change in Vitis 2021.2 requires hardware and hardware emulation provide their own XSA files during platform creation step in XSCT. One XSA with both hardware and hardware emulation content is still supported in 2021.2. It will be deprecated in the future.

1. Click **File→Export→Export Platform**. Alternative ways are: Flow Navigator window: **IP Integrator→Export Platform**, or the **Export Platform** button on the bottom of Platform Setup tab.
2. Click **Next** on Export Hardware Platform page.
3. Select **Hardware Emulation**. If there are any IP that don't support simulation, the Hardware XSA and Hardware Emulation XSA need to be generated separately. Click **Next**.
4. Select **Pre-synthesis**, because a DFX platform is not being created. Click **Next**.
5. Input the name and click **Next**.
6. Update the file name and click **Next**.
7. Review the summary and click **Finish**.

**Note:** Using the same hardware and hardware emulation design for a simple project is okay, but if your project is complicated, platform developers should keep the two designs logically identical. Otherwise your emulation result cannot represent your hardware design.

### **Adding Hardware Interfaces**

The following table shows the possible Vitis inputs and the minimal requirements for an acceleration embedded platform.

**Table 66: Available Interfaces for Vitis**

Inputs	Types Vitis Can Use	Minimum Requirements for AXI MM Kernels
Control Interfaces	AXI Master Interfaces from PS or from AXI Interconnect IP or SmartConnect IP	One AXI4-Lite Master for kernel control
Memory Interfaces	AXI Slave Interfaces	One memory interface for data exchange

**Table 66: Available Interfaces for Vitis (cont'd)**

Inputs	Types Vitis Can Use	Minimum Requirements for AXI MM Kernels
Streaming Interfaces	AXI4-Stream Interfaces	Not required
Clock	Multiple clock signals	One clock
Interrupt	Multiple interrupt signals	One Interrupt

## General Requirements



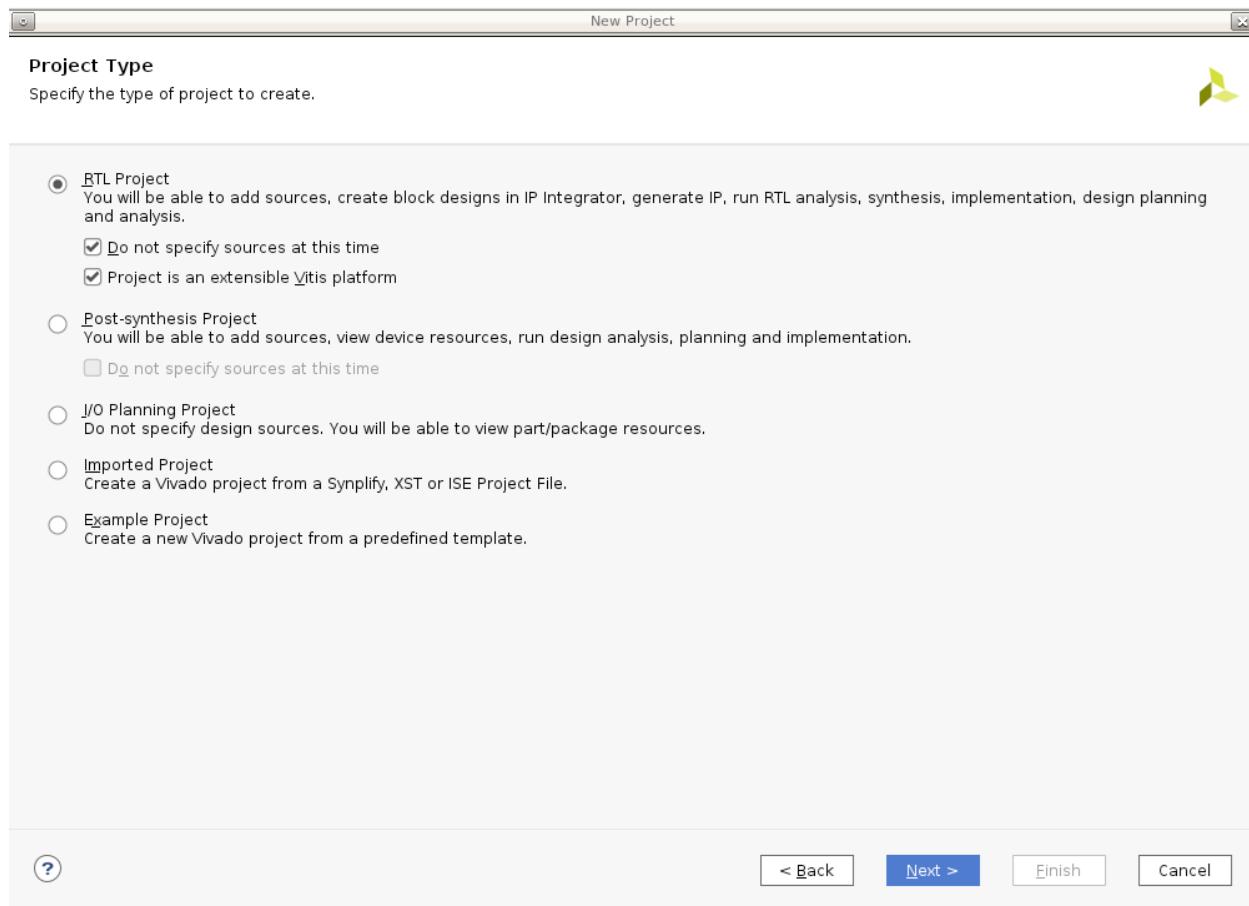
**IMPORTANT!** The source files for all elements of the Vivado project must be local to the project prior to exporting it as an XSA, or an error can be returned when using the platform in the Vitis tool.

- Every IP used in the platform design that is not part of the standard Vivado IP catalog must be local to the Vivado Design Suite project. References to IP repository paths external to the project are not supported when creating extensible XSA.
- Any platform interface, used for linking to kernels by the Vitis compiler, must be an AXI4, AXI4-Lite, AXI4-Stream, interrupt, clock, or reset type of interface.
- Any platform IP that has an AXI interface for linking to kernels by the Vitis compiler must also have associated clock pins to enable `v++` to correctly infer and insert clock domain crossing logic when needed.
- Custom bus type and hardware interfaces on the platform or on kernels are not supported through `v++ linker --connectivity.sp` and `--connectivity.sc` directives. If a data bus with a custom bus type needs to be connected to kernels by the Vitis compiler, it must be converted to an AXI4, AXI4-Lite, or AXI4-Stream interface.

## Project Type

To create a new XSA platform for the Vivado project type, select **RTL Project** and enable **Project is an extensible Vitis platform** check box.

Figure 105: Project Type



**TIP:** You will see these settings in the New Project wizard.

When creating a new project, select **Project is an extensible Vitis platform**.

To change an existing Vivado project to an extensible Vitis platform project, select **Project Manager** → **Settings** in the Flow Navigator and enable **Project is an extensible Vitis platform**.

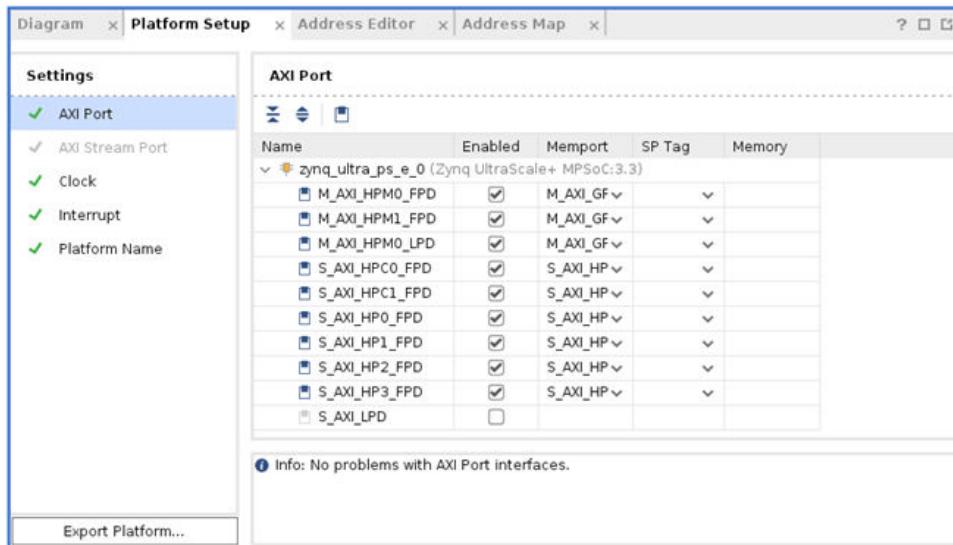
```
set_property platform.extensible true [current_project]
```

## Adding Platform Interfaces

If a component in block design has a PFM property, this component can be recognized by `v++` linker and can be used by the acceleration kernel.

In Vivado IDE, the Platform interface (PFM) properties can be set in the Platform Setup window if the project is created as an extensible platform project. Click **Window menu** → **Platform Setup** to open the settings.

Figure 106: Platform Setup



**TIP:** Platform interfaces can be defined manually in the Tcl Console, or by a Tcl script as well.

The four Platform Interface Tcl APIs include:

- AXI memory-mapped interfaces:

```
set_property PFM.AXI_PORT { <port_name> {parameters} <port2>
{parameters} ... } [get_bd_cells <cell_name>]
```

- AXI4-Stream interfaces:

```
set_property PFM.AXIS_PORT { <port_name> {parameters} <port2>
{parameters} ... } [get_bd_cells <cell_name>]
```

- Clocks and resets:

```
set_property PFM.CLOCK { <port_name> {parameters} <port2>
{parameters} ... } [get_bd_cells <cell_name>]
```

- Interrupts:

```
set_property PFM.IRQ {pin_name {id id_number range irq_count}}
[get_bd_cells <cell_name>]
```

The requirements for the PFM Properties are:

- The value of the PFM interface properties must be specified as a Tcl dictionary, a list of name/"value" pairs.



**IMPORTANT!** The "value" must be quoted, and both the name and value are case-sensitive.

- A bd\_cell can have multiple PFM interface definitions. However, for each type of PFM interface, all ports are required to be set in a single set\_property Tcl command.

- For each PFM interface property, the name specified for the port object must match the name of an external port or interface on a `bd_cell`. Each external port or interface object can only have one PFM interface definition.
- Each different type of PFM interface can have different parameters.
- Setting the PFM property with a NULL ("") string will delete previously defined PFM interfaces.

## Adding AXI Interfaces

To support AXI memory mapped kernels, the platform needs to declare at least one AXI control interface with AXI memory-mapped master port (M\_AXI) and one memory interface with AXI Slave port (S\_AXI). They can be exported from the PS block directly or have an interconnect IP connected. If the platform does not work with AXI memory mapped kernels, these interfaces are not required.

Due to security DFX decoupler IP is required to control the kernel if it is a DFX platform. The DFX decoupler can turn off the channels during reconfiguration to prevent unexpected requests from the static region cause invalid status on AXI interface and prevent random toggles generated by RP reconfiguration cause unexpected side-effects in static region. XRT will turn on DFX decoupler before the reconfiguration process and turn off after the reconfiguration completes.

NOC stub is required in a DFX platform in Vitis Region to export memory interface for the platform. V++ linker can connect the PL kernel memory interfaces to the NOC stub to access memory.

The following is the Tcl command syntax:

```
set_property PFM.AXI_PORT { <port_name> {parameters} <port2>
{parameters} ... } [get_bd_cells <cell_name>]
```

The AXI control interfaces and AXI memory interfaces share the same `PFM.AXI` property. They have different `memport` types.

- **Memport:** AXI control interface can be defined as M\_AXI\_GP. Memory interfaces use other types: S\_AXI\_HP, S\_AXI\_ACP, S\_AXI\_HPC, or MIG.
- **SP Tag ID:** (Optional) A user-defined ID that should start with an alphabetic character. The ID is case-sensitive. The system port tag (`sptag`) is a symbolic identifier that represents a class of platform port connections, such as S\_AXI\_HP, S\_AXI\_ACP,... Multiple block design platform ports can share the same `sptag`. For more information on how `sptags` are used, see [Mapping Kernel Ports to Memory](#).



**TIP:** The `sptag` property for is not supported M\_AXI\_GP ports.

- **Memory:** (Optional) Specify the associated MIG IP instance and `address_segment`. The memory tag is a unique identifier that combines the Cell Name and Base Name columns in the IP integrator Address Editor. This tag will be associated with connections to the Memory Subsystem HIP, where multiple block design platform ports can share the same memory tag.

Exporting AXI interconnect master and slave ports involves the following requirements:

- All ports on the interconnect used within the platform must precede in index order any declared platform interfaces.
- There can be no gaps in the port indexing.
- The maximum number of master IDs for the S\_AXI\_ACP port is 8, so on a connected AXI interconnect, available ports to declare must be one of {S00\_AXI, S01\_AXI, ..., S07\_AXI}. Do not declare any ports that are used within the platform itself. Declaring as many as possible will allow `sds++` to avoid cascaded `axi_interconnects`.
- The maximum number of master IDs for an S\_AXI\_HP or MIG port is 16, so on an connected AXI interconnect, available ports to declare must be one of {S00\_AXI, S01\_AXI, ..., S15\_AXI}. Do not declare any ports that are used within the platform itself. Declaring as many as possible will allow `v++` to avoid cascaded `axi_interconnects` in generated user systems.
- The maximum number of master ports declared on an interconnect connected to an M\_AXI\_GP port is 64, so on an connected AXI interconnect, available ports to declare must be one of {M00\_AXI, M01\_AXI, ..., M63\_AXI}. Do not declare any ports that are use within the platform itself. Declaring as many as possible will allow `v++` to avoid cascaded `axi_interconnects` in generated user systems.

The following shows an example of defining an AXI master ports on AXI Interconnect IP:

```
set parVal []
for {set i 2} {$i < 64} {incr i} {
lappend parVal M[format %02d $i]-AXI \
{memport "M_AXI_GP"}
}
set_property PFM.AXI_PORT $parVal [get_bd_cells /axi_interconnect_0]
```

The following shows an example of defining AXI memory ports with MIG on SmartConnect IP:

```
set parVal []
for {set i 1} {$i < 16} {incr i} {
lappend parVal S[format %02d $i]-AXI
{memport "MIG" sptag "Bank0"}
}
set_property PFM.AXI_PORT $parVal [get_bd_cells /smartconnect_0]
```

The following is an example of the PFM.AXI\_PORT setting for control interface and memory interface.

```
set_property PFM.AXI_PORT {
M_AXI_HPM1_FPD {memport "M_AXI_GP"}
S_AXI_HPC0_FPD {memport "S_AXI_HPC" sptag "HPC0" memory "zynq-ultra-ps-e_0"
HPC0_DDR_LOW}
S_AXI_HPC1_FPD {memport "S_AXI_HPC" sptag "HPC1" memory "zynq-ultra-ps-e_0"
}
```

```
HPC1_DDR_LOW" }  
S_AXI_HP0_FPD {memport "S_AXI_HP" sptag "HP0" memory "zynq_ultra_ps_e_0  
HP0_DDR_LOW" }  
S_AXI_HP1_FPD {memport "S_AXI_HP" sptag "HP1" memory "zynq_ultra_ps_e_0  
HP1_DDR_LOW" }  
S_AXI_HP2_FPD {memport "S_AXI_HP" sptag "HP2" memory "zynq_ultra_ps_e_0  
HP2_DDR_LOW" }  
} [get_bd_cells /ps_e]
```



**TIP:** In the examples above, `zynq_ultra_ps_e_0` is the instance name of the Zynq UltraScale+ MPSoC module, and `HPC0_DDR_LOW` is the address range name.

## Adding AXI4-Stream Interfaces

To support AXI4-Stream stream kernels, the platform needs to declare the corresponding master or slave AXI4-Stream interfaces.

AXI4-Stream kernel interfaces are specified with the PFM.AXIS\_PORT sptag interface property and a matching `connectivity.sc` command argument to the `v++` linker.

The following is the Tcl command syntax:

```
set_property PFM.AXIS_PORT { <port_name> {parameters} <port2>  
{parameters} ... } [get_bd_cells <cell_name>]
```

### Argument Description

- **Port\_name:** AXI4-Stream port name.
- **Parameters:** type value: Streaming interface port type. Valid values for type include:
  - **M\_AXIS:** A general-purpose AXI master port
  - **S\_AXIS:** A high-performance AXI slave port

### Example

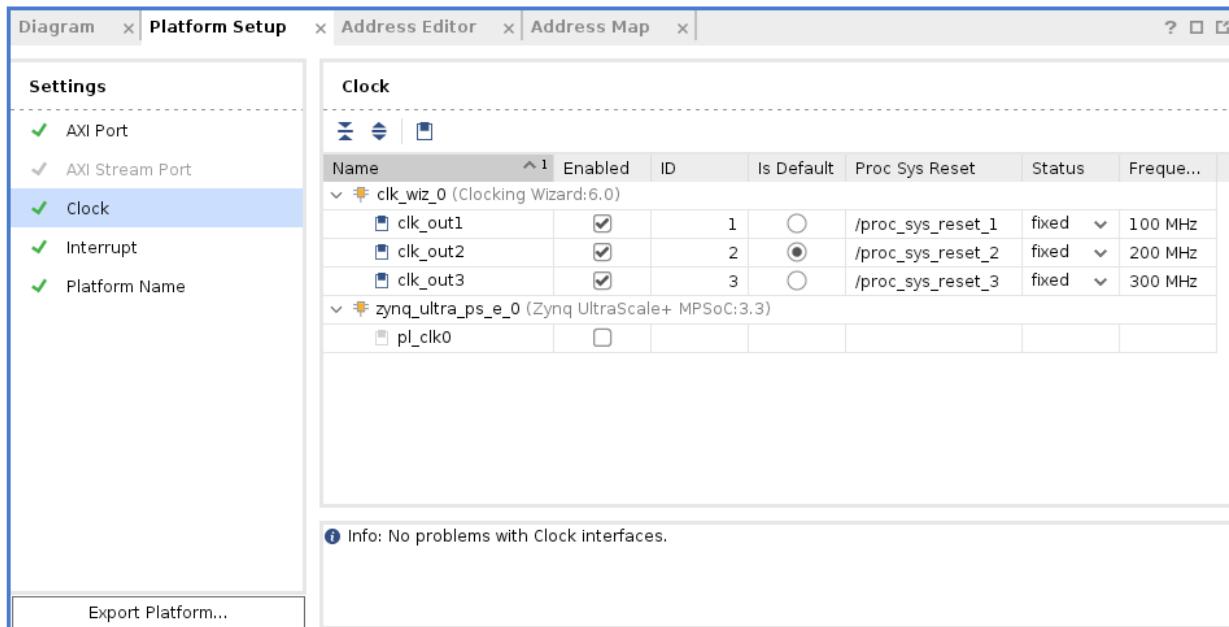
```
set_property PFM.AXIS_PORT {AXIS_P0 {type "S_AXIS"} } [get_bd_cells /  
zynq_ultra_ps_e_0]
```

**Note:** For more information on linking AXI4-Stream interfaces between kernels and platforms, see [Specifying Streaming Connections](#).

## Adding Clock and Resets

If it is a DFX platform static region and dynamic region can have their own clock and reset signals. The Clock Wizard in static region is required so that device tree generator (DTG) can generate correct device tree to describe this clock topology.

Figure 107: Platform Setup - Clock



You can export any clock source with the platform, but for each clock you must also export synchronized reset signals using a Processor System Reset IP block in the platform. For details of defining clocks and resets, see the [Vitis-Tutorials/Vitis\\_Platform\\_Creation](#). The PFM.CLOCK property can be set on a BD cell, external port, or external interface.

In the figure above you can see the details of the platform clocks. There must be at least one enabled clock for the platform and one clock must be specified as the default.

The following is the Tcl command for setting the PFM.CLOCK property:

```
set_property PFM.CLOCK { <port_name> {parameters} \
<port2> {parameters} ... } [get_bd_cells <cell_name>]
```

## Adding Interrupts

DFX decoupler IP is required when it is a DFX platform. It is used to connect the interrupt controller in the static region and the concat IP which is used to export the Interrupt signals in the Vitis region. Interrupt Controller outputs to CIPS IRQ.

Vitis provides a way to automatically connect the kernel output IRQ signal to an IRQ in the platform during the v++ link stage. The following shows the Tcl command syntax:

```
set_property PFM.IRQ {pin_name {id id_number}} bd_cell
set_property PFM.IRQ {port_name {id id_number range irq_count}}
[get_bd_cell <cell_name>]
```

## Argument Description

- **Port\_name:** IRQ port name of bd\_cell.
- **id\_number:** Integer from 0 to 127 to specify the IRQ number or the starting number if range is specified.
- **irq\_count:** Used for labeling interfaces that are otherwise subject to parameter propagation for specifying sizing of a bus (for example, interrupt controller intr interface).

The example shows how to enable 32 IRQ inputs to axi\_intc\_0 intr port.

```
set_property PFM.IRQ {intr {id 0 range 32}} [get_bd_cells /axi_intc_0]
```

The example shows how to enable 63 IRQ with cascaded interrupt controller in VCK190 base platform.

```
set_property PFM.IRQ {intr {id 0 range 32}} [get_bd_cells /  

axi_intc_cascaded_1]  

set_property PFM.IRQ {In0 {id 32} In1 {id 33} In2 {id 34} In3 {id 35} In4  

{id 36} In5 {id 37} In6 {id 38} In7 {id 39} In8 {id 40} \  

In9 {id 41} In10 {id 42} In11 {id 43} In12  

{id 44} In13 {id 45} In14 {id 46} In15 {id 47} In16 {id 48} In17 {id 49}  

In18 {id 50} \  

In19 {id 51} In20 {id 52} In21 {id 53} In22  

{id 54} In23 {id 55} In24 {id 56} In25 {id 57} In26 {id 58} In27 {id 59}  

In28 {id 60} \  

In29 {id 61} In30 {id 62}} [get_bd_cells /  

xlconcat_0]
```

## Platform settings for DFX only

- **Address aperture setting:** Setting the address range for the dynamic region is required so that the CIPS and the SmartConnect can access kernels in the dynamic region after linking stage by V++. Setting the apertures to DDR interfaces is required to export the accessible DDR range for the kernel. Lock the aperture settings by typing the following command in the Tcl console.

```
set_property HDL_ATTRIBUTE.LOCKED TRUE [get_bd_intf_pins /VitisRegion/  

PL_CTRL_S_AXI] set_property HDL_ATTRIBUTE.LOCKED TRUE [get_bd_intf_pins /  

VitisRegion/DDR_0] set_property HDL_ATTRIBUTE.LOCKED TRUE  

[get_bd_intf_pins /VitisRegion/DDR_1] set_property HDL_ATTRIBUTE.LOCKED  

TRUE [get_bd_intf_pins /VitisRegion/DDR_2] set_property  

HDL_ATTRIBUTE.LOCKED TRUE [get_bd_intf_pins /VitisRegion/DDR_3]
```

- **Setup Block Design Container (BDC) for DFX:** Update VitisRegion BDC properties for DFX to freeze the boundary of this container and Enable Dynamic Function eXchange on this container. Configure Dynamic Function eXchange Wizard and add a configuration.

- **Setup the DFX platforms properties:**

```
# Specify that this platform supports
DFX set_property platform.uses_pr true [current_project]
# Specify the dynamic region instance path for hardware run
set_property platform.dr_inst_path {design_1_i/VitisRegion}
[current_project]
# Specify the dynamic region instance path for emulation
set_property platform.emu.dr_bd_inst_path {design_1_wrapper_sim_wrapper/
design_1_wrapper_i/design_1_i/VitisRegion} [current_project]
```

## Exporting the Extensible Platforms

Hardware platforms are encapsulated in XSA file format. There are two kinds of XSA formats: fixed XSA for embedded software development and extensible XSA for Vitis application acceleration projects. To create an embedded platform for the Vitis application acceleration flow, you must use an extensible XSA.

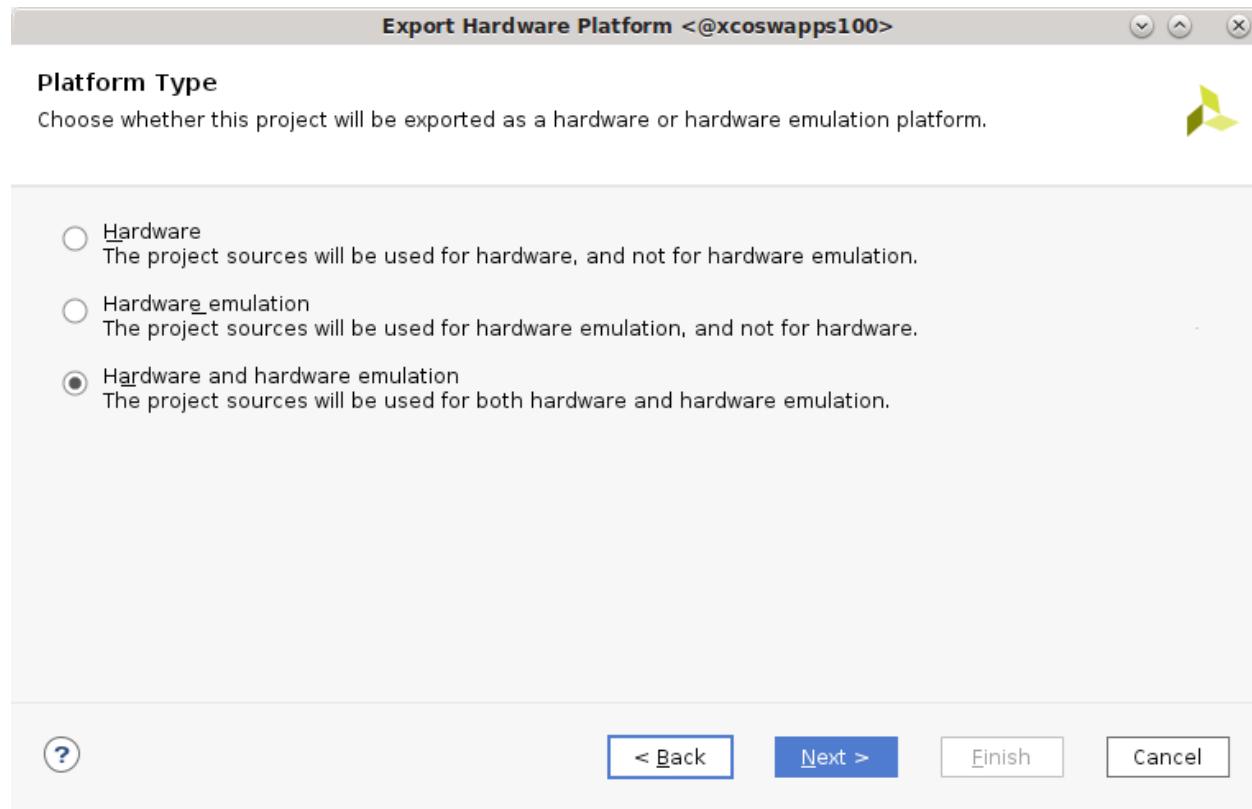
When the Vivado project type is set to extensible Vitis platform, the Export Platform wizard is available from the **File → Export → Export Platform** menu command.



---

**IMPORTANT!** The block design must have an HDL wrapper and have output targets generated to export the platform XSA. Use the Create HDL Wrapper from the right-click menu in the Sources window to create the wrapper, and Generate Block Design from the Flow Navigator in the Vivado IDE.

---

**Figure 108: Export Hardware Platform Wizard**

The Export Platform wizard contains five pages to help you export the extensible platform XSA:

- **Platform Type:** Specifies the XSA as supporting hardware emulation, and hardware targets.
- **Platform State:** Specifies the XSA as including platform implementation or only synthesis.
- **Platform Properties:** Defines platform properties and lets you specify Tcl scripts and XDC constraints for the Vitis compiler to use when building the system.
- **Output File:** Specifies the output file name and location.
- **Summary:** Reports the various settings that will be used during export.

In the Export Hardware Platform wizard, select the platform type. There are three types of platforms: platforms intended to run on hardware only, platforms intended for hardware emulation only, or platforms that can run in hardware emulation and on hardware. The differences between these options is that if some modules are not supported by emulation, you should create a separate emulation specific design, export it as "hardware emulation" platform and then use "Combine XSAs" option to combine a hardware XSA and a hardware emulation XSA into one XSA that is capable of performing both jobs.

For basic platforms, use the following steps:

1. Select **Hardware and Hardware Emulation**. Click **Next**.
2. Select **Pre-synthesis for Platform State**. Post-implementation is only needed when creating DFX platforms. Click **Next**.
3. Input **Platform Properties**. Click **Next**.
4. Input the XSA file name and the export target directory. Click **Next**.
5. Check summary and click **Finish**.

You can also perform this in the command line using the following command:

```
set_property pfm_name {vendor:board:name:version} [get_files <bd_file>]
write_hw_platform -hw -force <XSA file>
```

Commands can be used to export the XSA file for DFX platform only.

```
#emulation XSA
set_property platform.platform_state "pre_synth" [current_project]
write_hw_platform -hw_emu -force -file vck190_custom_dfx_hw_emu.xsa
#hardware and RP XSA
set_property platform.platform_state "impl" [current_project]
write_hw_platform -force -fixed -static -file vck190_custom_dfx_static.xsa
write_hw_platform -force -rp design_1_i/VitisRegion vck190_custom_dfx_rp.xsa
```

To create and combine a hardware XSA and a hardware emulation XSA, use the following commands:

```
write_hw_platform -hw <hw_platform>
write_hw_platform -hw_emu <hw_emu_platform>
combine_hw_platform -hw <hw_platform> -hw_emu <hw_emu_platform> -o
<combined_platform>
```

## Updating Software Components

### Prepare Common Image

Download the Xilinx common image from the [Xilinx download page](#), extract it, and place it in the project folder.

### Create DTB File

Use the "createdts" command in XSCT tool to generate DTB file. The zocl driver interface requires a device tree node to enable the interrupt connection. Add the -zocl option when using this command. The following code shows the usage of this command with its options.

```
createdts -hw <full path of XSA file> -zocl -platform-name mydevice -git-
branch xlnx_rel_v2021.1 -board zcu104-revc -compile
```

The `system.dtb` file is located in `<mydevice/psu_cortexaXX_0/device_tree_domain/bsp>` folder.

- `-name`: Platform name
- `-hw`: Hardware XSA file with path
- `-git-branch`: device tree branch
- `-board`: board name of the device. You can check the board name at `/device_tree/data/kernel_dtsi`.
- `-zocl`: enable the zocl driver support
- `-compile`: specify the option to compile the device tree

The following is an example of the zocl device node for your reference.

```
&amba {
    zyxclmm_drm {
        compatible = "xlnx,zocl";
        status = "okay";
        interrupt-parent = <&axi_intc_0>;
        interrupts = <0 4>, <1 4>, <2 4>, <3 4>,
                    <4 4>, <5 4>, <6 4>, <7 4>,
                    <8 4>, <9 4>, <10 4>, <11 4>,
                    <12 4>, <13 4>, <14 4>, <15 4>,
                    <16 4>, <17 4>, <18 4>, <19 4>,
                    <20 4>, <21 4>, <22 4>, <23 4>,
                    <24 4>, <25 4>, <26 4>, <27 4>,
                    <28 4>, <29 4>, <30 4>, <31 4>;
    };
};
```

For more information, refer to the XRT documentation: <https://xilinx.github.io/XRT/master/html/yocto.html>.

## Packaging a Vitis Acceleration Platform

With all requirements prepared for Vitis acceleration platforms, you can package them together and generate the final Vitis acceleration platform. You can do this using either the Vitis IDE or the Xilinx Software Command-Line Tool (XSCT).

- In the Vitis IDE, select **File → New → Platform Project** to create a Vitis platform.
- Using XSCT, you can use the `platform` command to create a platform and the `domain` command to add domains into a platform. For more information about XSCT, see *Vitis Unified Software Platform Documentation: Embedded Software Development (UG1400)*.

The platform is an encapsulation of multiple hardware and software components. This encapsulation makes it easier to hand off deliveries from hardware-oriented engineers to application developers.

The following files and information are packaged into the platform.

- **Hardware Specification:** This is an extensible XSA file.
- **Software Components:** These are added to the platform as a Linux domain that enables OpenCL runtime. Software components include the following:
  - Boot components
    - BIF file that describes the boot components and their properties for Bootgen to generate the `boot.bin` file.
    - A boot components directory that includes all the files described in the BIF file.
  - Image directory (optional): Contents in this directory will be copied into the FAT32 partition of the final SD card image.
  - Linux domain: The platform requires a Linux domain. The kernel, RootFS, and sysroot info can be added when creating a platform, or when creating an application.
  - Emulation support files (optional)

## Root Filesystem

FAT32 and Ext4 partition types are supported by Vitis. The root filesystem is optional in platform creation step because it can be assigned during Vitis application creation step.

An image directory needs to be set during platform creation. All contents in this directory will be packaged into final SD card image. If the target file system is FAT32, the files will be placed to SD card root directory; if the target file system is Ext4, the files will be placed to root directory of the first FAT32 partition.

## Boot Components

A BIF file must be provided so that the application build process can package the boot image.

The following is an example of a BIF file:

```
/* linux */
the_ROM_image:
{
    [fsbl_config] a53_x64
    [bootloader] <fsbl.elf>
    [pmufw_image] <pmufw.elf>
    [destination_device=p1] <bitstream>
    [destination_cpu=a53-0, exception_level=el-3, trustzone] <bl31.elf>
    [destination_cpu=a53-0, exception_level=el-2] <u-boot.elf>
}
```

A boot components directory, including all the files described in the BIF, should also be provided. In this example, the components directory provides `fsbl.elf`, `pmufw.elf`, `b131.elf`, and `u-boot.elf`. These boot components can be generated by PetaLinux or from common images. Xilinx recommends that you use a common image if there is no system customization.

In the Vitis application building and packaging state, `v++` looks for the files in the boot components directory and replaces the placeholders with real file names and paths. It then calls Bootgen to generate `BOOT.BIN`.

## Testing Your Platform

Before delivering the platform to the application developers, you should run some basic platform tests to make sure it works properly for acceleration applications.

Generally, you need to make sure the platform can pass these tests:

- **Boot test:** The Vivado project generated implementation result `BIT` file (from [Adding Hardware Interfaces](#)) and PetaLinux generated images (from [Updating Software Components](#)) should be able to successfully boot to the Linux console.
- **Platforminfo test:** The platform generated in [Packaging a Vitis Acceleration Platform](#) should have a proper `platforminfo` report for clock and memory information.
- **XRT basic test:** The XRT `xbutil examine` utility should be able to run on the target board and properly report platform information.
- **Vadd test:** Use Vitis to generate a vector addition sample application with the platform. The generated application and `xclbin` should print `test pass` on the board.

## Enabling Hardware Emulation for Extensible XSA

The following steps are used for custom platform developers.

1. Create a Vivado project with the necessary BD, RTL, test bench, and other sources.
  - a. Note that in 2020.1, only BD can be used in HW EMU, but starting 2020.2, other sources will also be allowed.
  - b. For Versal ACAPs only, test bench needs to include BD wrapper instead of including BD directly, because Vitis does performs jobs on this level to insert NoC into simulation.
  - c. For Versal ACAPs only, to enable AI Engine in the Vitis platform, the AI Engine block needs to be configured to have only one slave AXI4 Memory-Mapped port enabled and connected to NoC. Vitis based on AI Engine Graph software will make additional auto-connections during `v++` linking stage.
  - d. For DFX platforms, specify correct PFM properties in the dynamic region BD so that the Vitis tools can attach accelerators correctly.

2. Update the design HW Emulation packaging into XSA.
  - a. Before packaging the design into XSA, it is important that your design step through the simulator correctly.
  - b. For Versal ACAPs only, prepare the platform design to enable SystemC models. Update the CIPS and NoC IP setting to change SELECTED\_SIM\_MODEL property to TLM. This ensures that for CIPS IP, the design uses QEMU model on which SW can be run. Similarly, for Zynq®-7000 and Zynq® UltraScale+™ MPSoC devices, set the SELECTED\_SIM\_MODEL on the processing system IP instance. Following Tcl command can be used in the design. Also, set the parameter to enable SystemC simulation in Vivado:

```

foreach tlmCell [get_bd_cells * -hierarchical -filter {VLNV =~
"*:*:axi_noc:*" || VLNV =~ "*:*:versal_cips:*"}] {set_property
SELECTED_SIM_MODEL tlm $tlmCell }
set_param bd.generateHybridSystemC true

```

- c. Create a test bench in sim\_1 fileset and instantiate the <top> module of your design. For Versal ACAPs, Vivado requires that the user test bench should not instantiate the <top> module directly. Instead, it should instantiate <top>-sim\_wrapper module. A file called <top>-sim\_wrapper.v is generated when you call the launch\_simulation -scripts\_only command. The interface of this module is the same as your <top> module, but it instantiates additional simulations models related to an aggregated NoC module created from various logical NoC modules instantiated in the design. Use the following Vivado Tcl commands to generate the necessary NoC simulation file and use them in your simulation sources.

```

# Ensure that your top of synthesis module is also set as top for
simulation

set_property top <rtl_top> [get_filesets sim_1]

# Generate simulation top for your entire design which would include
# aggregated NOC in the form of xlnoc.bd

launch_simulation -scripts_only
update_compile_order -fileset sim_1

# Set the auto-generated <rtl_top>-sim_wrapper as the sim top

set_property top <rtl_top>-sim_wrapper [get_filesets sim_1]
update_compile_order -fileset sim_1

#Generate the final simulation script which will compile
# the <syn_top>-sim_wrapper and xlnoc.bd modules also
launch_simulation -scripts_only
launch_simulation -step compile
launch_simulation -step elaborate

```

- d. Compile the design, go through the above steps, and start simulation. Because the design is configured to use QEMU, the CIPS IP will not generate any transactions because there is no SW present when doing simulation in Vivado simulator. You will see the following ERROR message in the Vivado simulation, but it indicates that the basic design loads correctly in simulator.

```
#####
#      # Simulation does not work as Versal CIPS Emulation
# (SELECTED_SIM_MODEL=tlm) only works with Vitis
# tool(launch_emulator.py tool in Vitis)
#
#####
ERROR: [Simtcl 6-50] Simulation engine failed to start: The
Simulation shut down unexpectedly during initialization.
```

**Note:** To confirm that the design will have correct transactions, you can optionally perform a simulation of the design using the CIPS VIP first before changing it to use TLM (QEMU). First, you must keep the SELECTED\_SIM\_MODEL property to be RTL for NoC and CIPS IP. Also, create a different test bench which drives the CIPS VIP and also meet the requirement of the NoC Verilog model. Refer to the CIPS VIP and NoC IP documentation for additional details on how to set up test bench for Verilog-based simulation.

### 3. Package the HW Emulation only XSA.



**IMPORTANT!** *The source files for all elements of the Vivado project must be local to the project prior to exporting it as an XSA, or an error can be returned when using the platform in the Vitis tool.*

- a. Use **Vivado File → Export → Export Platform** to export a Hardware Emulation platform or use the following Tcl command:

```
set_property platform.platform_state "pre_synth" [current_project]
write_hw_platform -hw_emu -file platform_hw_emu.xsa
```

- b. This XSA can be used with pre-built Linux images or with PetaLinux to create a custom Linux image to create a full platform. Then, the remainder of the Vitis tools can be used to add a kernel to design with the XRT.

## Special Considerations for Embedded Platform Creation

### Divide Logic Functions to Platform and Kernel

While the designs on FPGA and SoC are getting more complex, it is common for multiple developers or teams to work on a design together. The Vitis software platform provides a clear boundary for application developers and platform developers. Platform developers could include board developers, BSP developers, system software developers, and so on.

In the view of a system architect, some logic functions might be in a gray area: they can be packaged with platforms, or they can work as an acceleration kernel. To help divide the system blocks, here are some general guidelines.

- When a component can be classified as either platform or acceleration kernel, set it as a kernel. The platform should be designed as thin as possible to help reduce the platform repackaging effort when this component needs an update. Also, if you iterate during the development phase, it can save time.
- Platforms should be the abstraction of hardware. When changing a hardware board, the application does not require changes, or very few changes if necessary, to target the new hardware.
- IPs in the platform are controlled by device tree and Linux drivers, acceleration kernels are controlled by XRT. If some IP needs standard Linux driver support, package them into the platform and describe them in the device tree.
- Acceleration kernels usually have standalone features and simple datapath and control path connections. If some components provide infrastructural features shared by multiple kernels or they have complicated connections with other modules, they usually can be packaged in the platform.

The following table shows the recommended platforms and kernels for logic types.

**Table 67: Recommended Platforms and Kernels**

Logic	Platform	Kernel
Hard Processors (PS of -7000 SoC and Zynq UltraScale+ MPSoC)	Only in Platform	
Soft Processors	Preferred in Platform	OK as an RTL kernel
I/O Interface IP with GPIO and SerDes (External pins, MIPI, Aurora, etc.)	Only in Platform	
I/O Interface Companion IP (DMA for PCIe®, Ethernet MAC for Ethernet PHY, etc.)	Standard and stable IP should be placed in the platform if they are not application specific.	Customized companion IP can work as acceleration kernels.
Traditional memory mapped IP which needs standard Linux driver (VPSS, etc.)	Only in Platform	
HLS AXI memory mapped IP	OK in Platform. Need to write control software or driver manually.	Preferred as Kernel. Controlled by XRT.
Acceleration memory mapped IP follows Vitis kernel register standard and open to XRT		Preferred as Kernel
Vitis Libraries		Only work as Kernel

## References

For more information on embedded platforms, see the following links:

- [Vitis Platform Creation tutorial](#)
- [Vitis Embedded Platform Source](#)

# Validating an Embedded Platform

As described in [Platform Creation Requirements](#), there are steps you should follow to validate the platform you have created. The following sections describe different actions you can take to examine and exercise the embedded platform for software, graph, and kernel development to ensure the platform meets your needs and expectations.

## Check Platform Metadata

During the platform creation process you will define the resources available in the platform. You can perform a quick examination of the platform definition, and the available resources using the `platforminfo` command. This utility can print platform metadata to let you confirm that the defined resources of the packaged platform meet your expectation.

## Use the Platform with Simple Test Case

By using the platform with a simple test case, you can ensure that the systems work as expected, without needing to debug the host application, graph, or kernel interactions. A simple test can exercise a specific feature set of the platform, without the complexities of design. Xilinx provides a number of examples and tutorials for use as test cases, some of which can be found at the following locations:

- [https://github.com/Xilinx/Vitis\\_Accel\\_Examples](https://github.com/Xilinx/Vitis_Accel_Examples)
- <https://github.com/Xilinx/Vitis-Tutorials>

A simple test can help to check whether the clock and reset, AXI4 interface settings, and interrupts are correct and functioning as expected.

## Run Baremetal Software Emulation

To step back from creating acceleration applications for platform verification, you can run emulation of baremetal host applications to test the hardware inside the platform. This lets you quickly test the embedded platform against the drivers required to access elements of the hardware through the baremetal application. This can help platform developers validate platform peripheral features and custom IP in the platform with simple applications. This validation requires you to convert the extensible platform to a fixed version for baremetal software development. This validation does not need Linux software components.

To create a fixed form of the extensible platform use the following steps.

- **Export a Fixed-XSA from the Extensible Hardware Platform:** As described in [Platform Types](#), there are two general types of platforms: extensible platforms and fixed platforms. Fixed platforms support embedded software development, and do not let you modify the PL kernels or the target platform defined by the device binary. However, the AI Engine flow does let you modify and recompile the graph application on a fixed platform.

A fixed-version of the hardware platform (.xsa) can be generated during the linking process of the Vitis compiler (v++) when building an Application Project as described in [Building the Device Binary](#).

In the Vitis IDE, the fixed-XSA can be generated by enabling the **Export hardware (XSA)** check box in the Hardware Link Project Settings view. For the command-line flow, the feature must be enabled by adding the following option to a configuration file used during the v++ --link command:

```
[advanced]
param=compiler.addOutputTypes="hw_export"
```

The Vitis compiler exports a fixed hardware specification containing elements of the target platform with any acceleration kernels and the AI Engine graph fixed into the hardware. The fixed-XSA provides a BSP for the system design that you can use for developing embedded software applications.

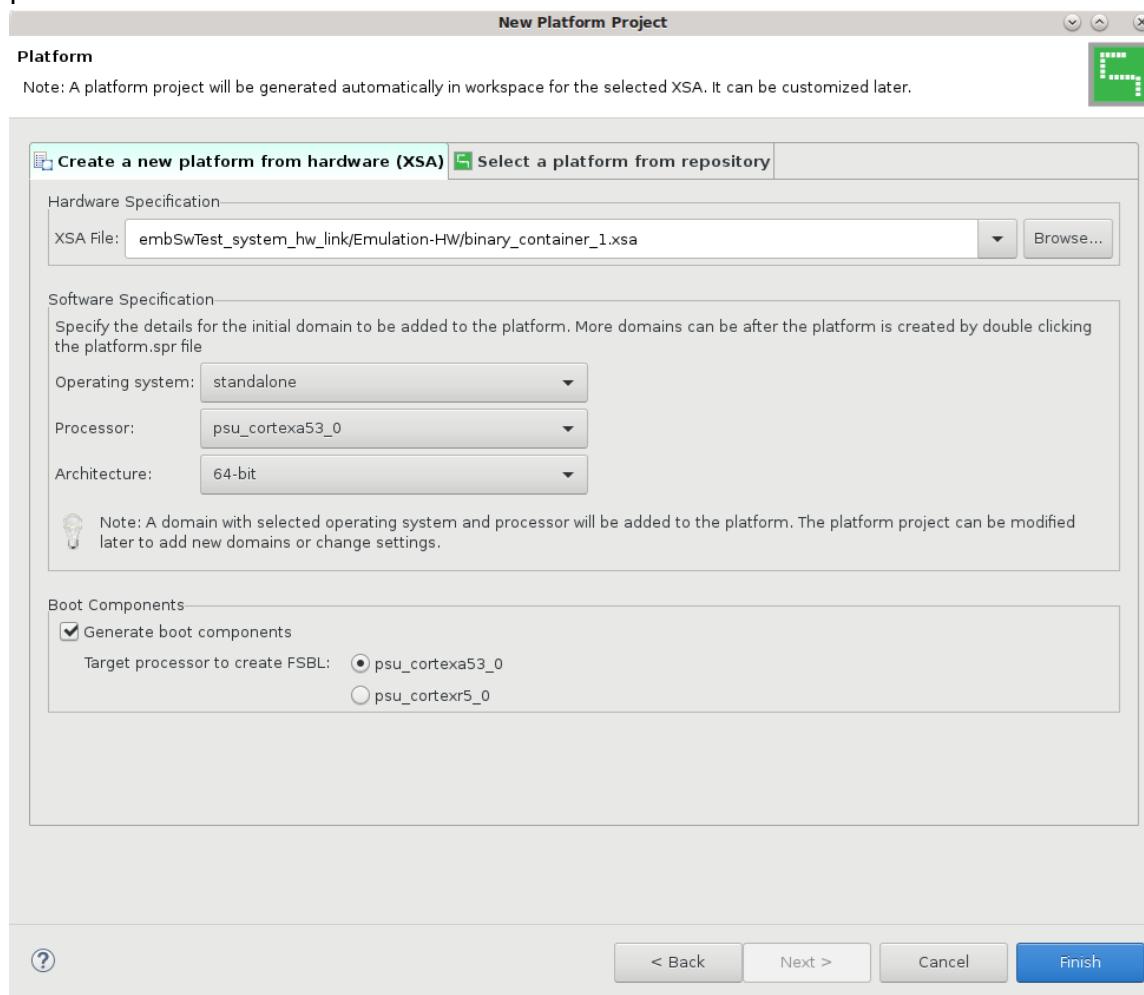


**IMPORTANT!** The fixed-XSA matches the specified build target of the Vitis compiler. So a hardware emulation capable fixed-XSA is generated from the `hw_emu` build target, and a hardware capable fixed-XSA is generated from the `hw` build target.

- **Generate a Baremetal Platform:**

1. The fixed-XSA can be used to create or generate a baremetal fixed-version of an embedded processor platform, that can be used to validate the platform. Open the Vitis IDE and select **File → New → Platform Project**. This opens the New Platform Project wizard.
2. Specify the Platform project name and click **Next**. This opens the Platform page as shown in the figure below.

3. In this dialog box you can define the platform by specifying the name of the XSA File, selecting the Operating system, Processor domain, and Generate boot components for the platform.



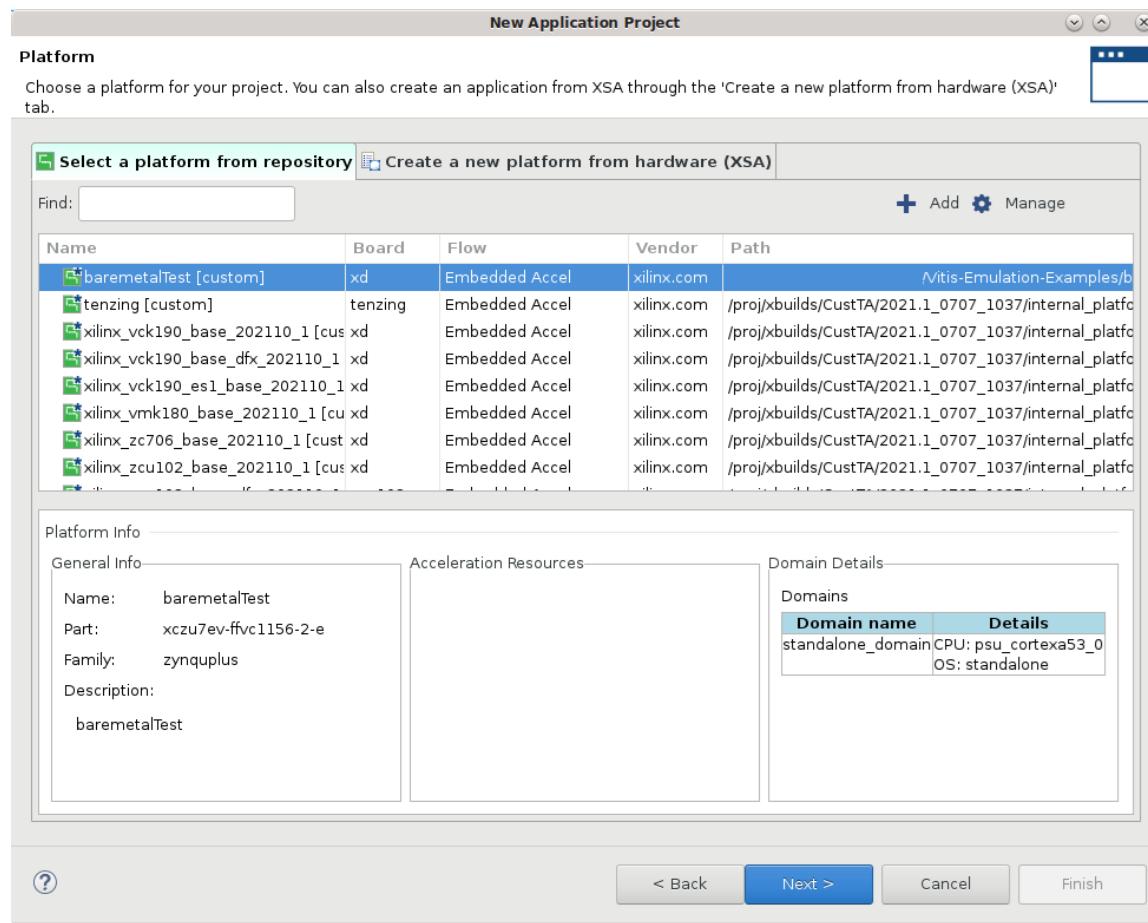
Select **standalone** for Operating system, specify the Processor, and Architecture to define the domain. Enable Generate boot components as shown above, and click **Finish** to generate the baremetal embedded platform.

When the platform is complete, you should see it added to your repository of available platforms. You will use it to create a baremetal application project as described next.

- **Validate the Baremetal Platform with an Application Project:** With the baremetal platform built, you can use a simple application running on the Arm processor to validate the platform. This approach lets you quickly exercise various features of the platform from compiled ELF files.

1. In the Vitis IDE, select **File → New → Application Project**. This opens the New Application Project wizard.

On the Platform page, select the baremetal platform you created in the prior step, and click **Next** to continue.

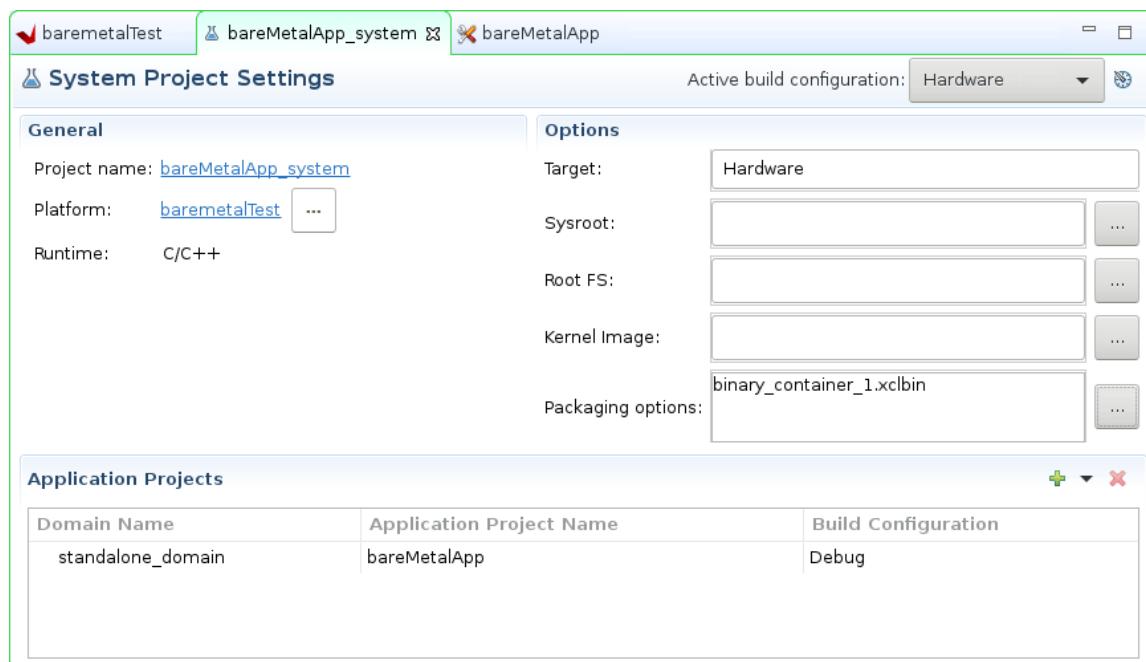


2. The New Application Project wizard presents the Application Project Details page next. Specify the Project name and the System project name, and click **Next** to continue.
3. The Domain page displays the domain information for the selected platform. Click **Next** to proceed.
4. The Templates page displays a list of Embedded software development templates for the selected platform. Select the **Hello World** template, or another simple application, or select the **Empty Application** to let you specify your own source files, and click **Finish** to create the application project.
5. With the baremetal application project in support of the baremetal platform, you are now ready to begin validation. You can use the baremetal platform with the same build target that the fixed-XSA was created from:
  - If the fixed-XSA was generated from a hardware emulation build (`v++ --link -t hw_emu`), it can be used for Emulation-HW builds in the baremetal application project.
  - If the fixed-XSA was generated from a hardware build (`v++ --link -t hw`), it can be used for Hardware builds in the baremetal application project.

In the System Project Settings window, specify the Active build configuration that is compatible with the fixed-XSA of the baremetal platform, as shown below.



**TIP:** You might also need to specify an `.xclbin` file for the baremetal application project in the Packaging options field as shown in the figure below. This is necessary for the hardware build, and you can copy the `.xclbin` file from the source project of the fixed-XSA into your baremetal Application Project.



6. For the Emulation-HW build configuration, you can run the application in the QEMU environment and see the application interacting with the fixed-platform to validate the system. To run the emulation build, select the **Launch HW Emulator** command from the Assistant view, or the main toolbar menu. QEMU is launched through the TCF agent and in the Emulation Console view you can observe the transcript of QEMU booting and the PS application running.



**TIP:** You can also select the **Debug As** command to launch the interactive debug environment in the Vitis IDE.

# Additional Information

This section contains the following chapters:

- [Methodology for Accelerating Data Center Applications with the Vitis Software Platform](#)
- [OpenCL Programming](#)
- [Migrating to a New Target Platform](#)
- [Streaming Data Transfers](#)
- [Output Structure of the Vitis Tools](#)
- [Using Vitis System Compilation Mode](#)

# Methodology for Accelerating Data Center Applications with the 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. To refine our analogy, a programmable device would be like a collection of empty lots waiting to be developed. This means that the device developer gets to build factories, assembly lines, and workstations, and then customizes them for the required task instead of using general purpose tools. And just like lot size, device real-estate is not infinite, which limits the number and size of the factories which can be built in the device. Properly architecting and configuring these factories is therefore a critical part of the device programming process.

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

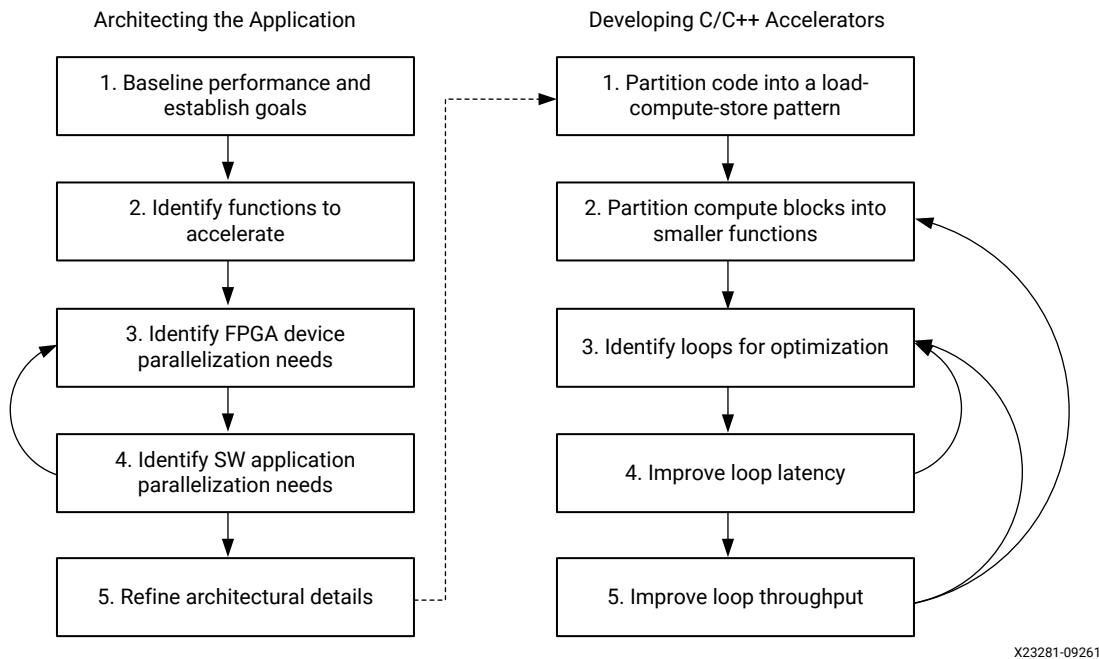
## Methodology Overview

The methodology is comprised of two major phases:

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

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

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

**Figure 109: 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 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:

- [Section III: Developing Applications](#)
- [Vitis Application Acceleration Development Flow Tutorials](#) on GitHub.

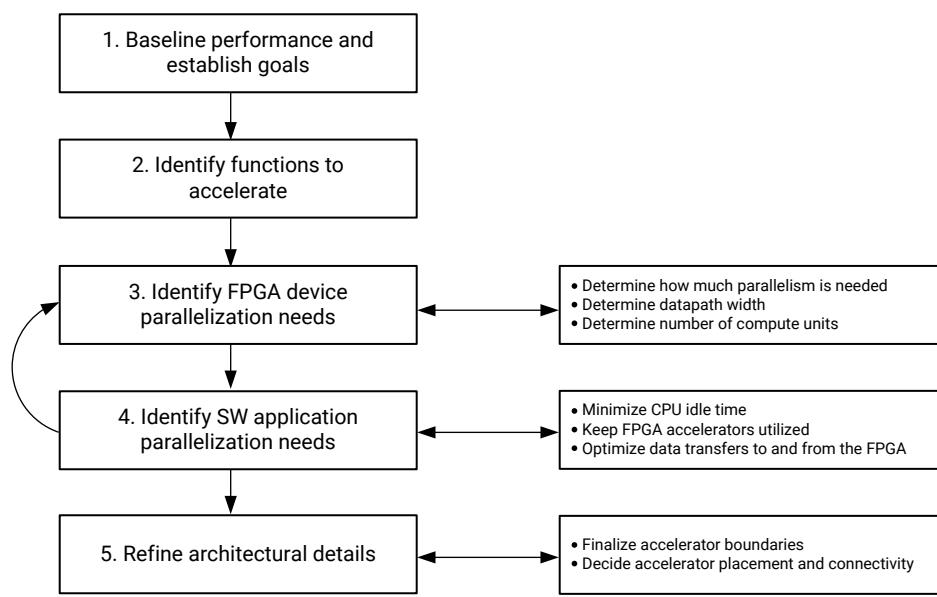
In addition to understanding the key aspects of the Vitis software platform, a good understanding of the following topics will help achieve optimal results with this methodology:

- 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 should be mapped to device kernels, how much parallelism is needed, and how it should be delivered.

Figure 110: Methodology for Architecting the Application



X23282-092619

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 should be generated for the entire application (end-to-end) as well as 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 111: 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 xbutil dma test.

Figure 112: 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 should be accelerated 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:

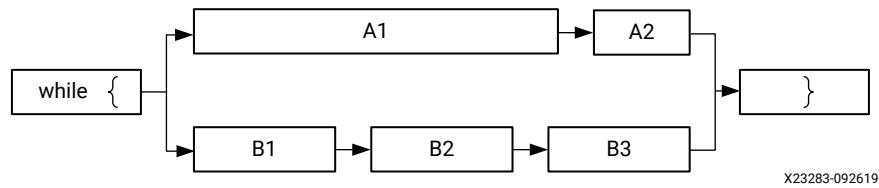
- **Performance bottlenecks:** Which functions are in application hot spots?
- **Acceleration potential:** Do these functions have the potential for acceleration?

## 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 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 113: 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, not just of individual functions.

## Identify Acceleration Potential

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

- What is the computational complexity of the function?

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

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

On Alveo Data Center accelerator cards, the PCIe bus imposes a throughput limit on data transfers. While it may not be the actual bottleneck of the application, it constitutes a possible upper bound and can therefore be used for early estimates. For example, considering a PCIe throughput of 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 have been identified and the overall acceleration goals have been established, the next step is to determine what level of parallelization is needed to meet the goals.

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

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

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

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

The developer 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 should then determine the best combination given their needs and the characteristics of their application.

## **Determine How Many Samples the Datapath Should be Processing in Parallel**

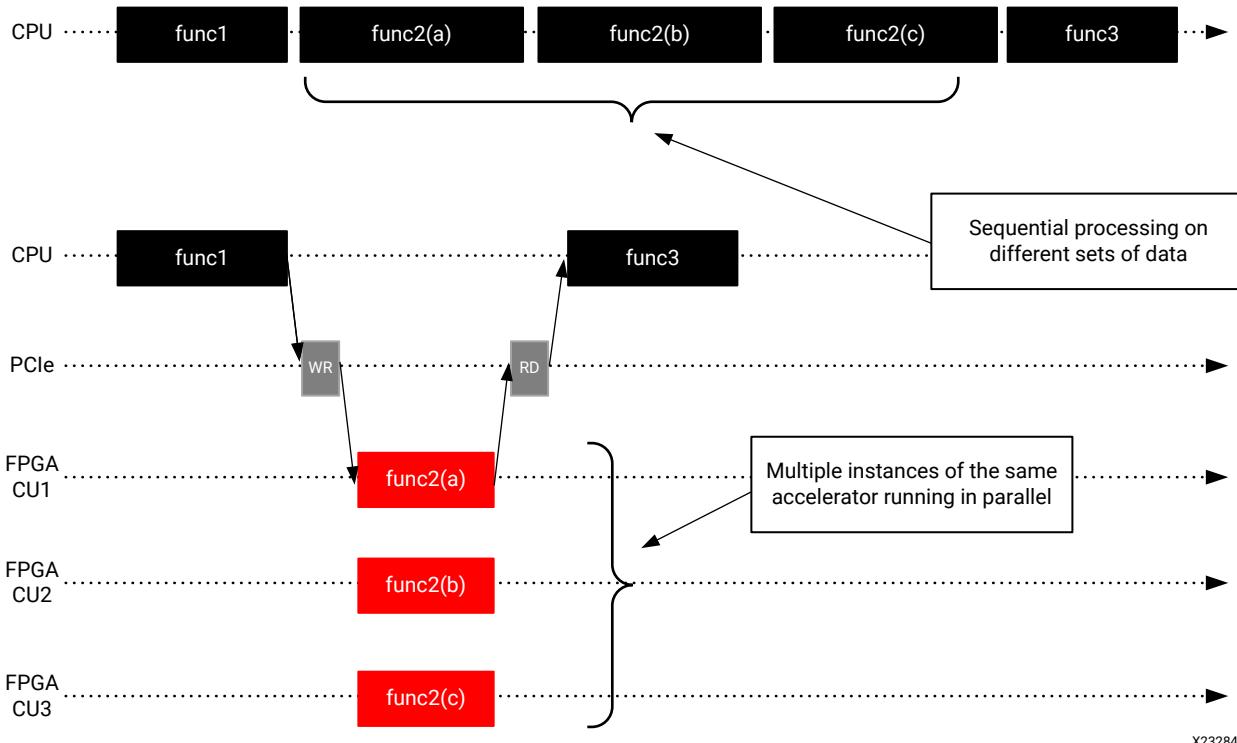
One possibility is to accelerate the computation by creating a wider datapath and processing more samples in parallel. Some algorithms lend themselves well to this approach, whereas others do not. It is important to understand the nature of the algorithm to determine if this approach will work and if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

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

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

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

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

**Figure 114: Improving Performance with Multiple Compute Units**

X23284-092619

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

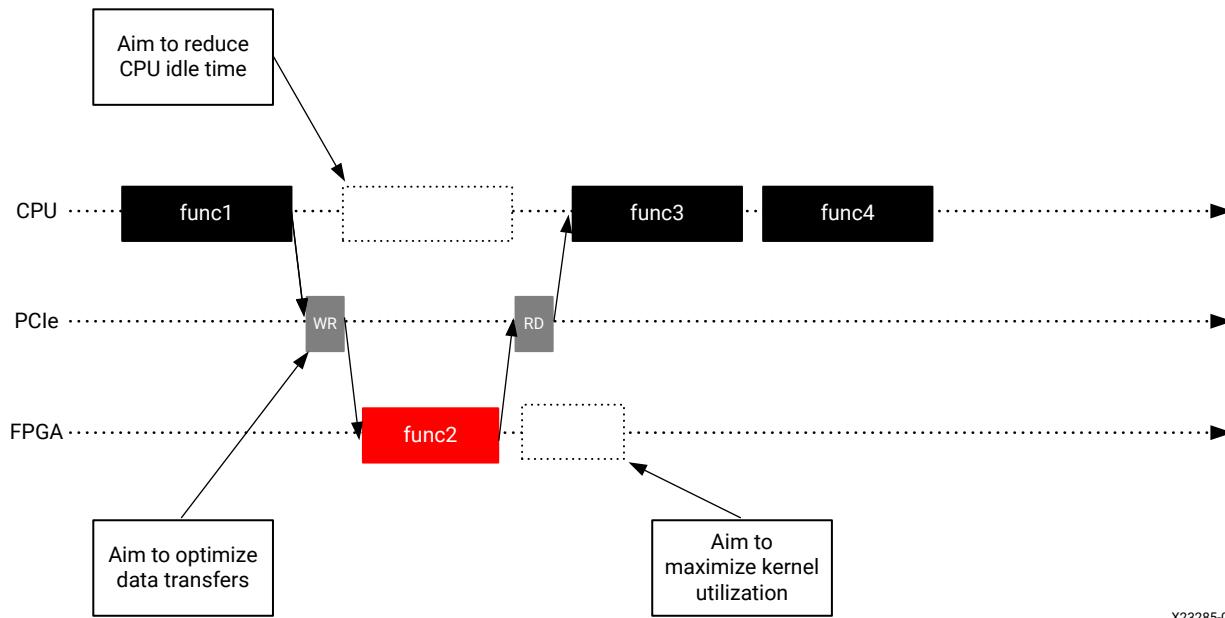
At this point, the developer should have a good understanding of the amount of parallelism necessary in the hardware to meet performance goals and through a combination of datapath width and kernel instances, how that parallelism will be achieved.

## Step 4: Identify Software Application Parallelization Needs

While the hardware device and its kernels are 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 115: 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.

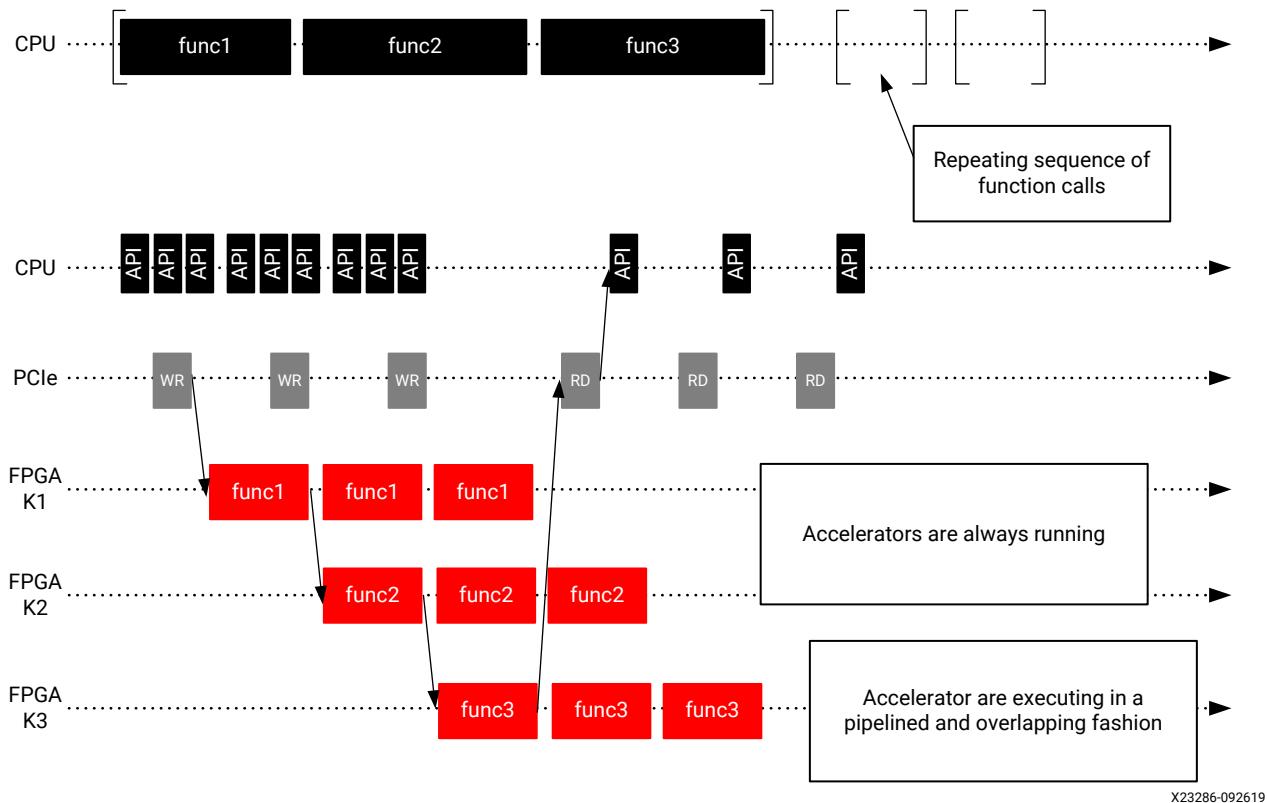
Instead, engineer the software application to avoid such idle cycles. Begin by identifying parts of the application that do not depend on the results of the kernel. Then structure the application so that these functions can be executed on the host in parallel to the kernel running in the device.

### ***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 will keep the kernels busy.

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

Figure 116: Pipelined Execution of Accelerators



X23286-092619

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

More information on software pipelining can be found in the [Vitis Application Acceleration Development Flow Tutorials](#).

## ***Optimize Data Transfers to and from the Device***

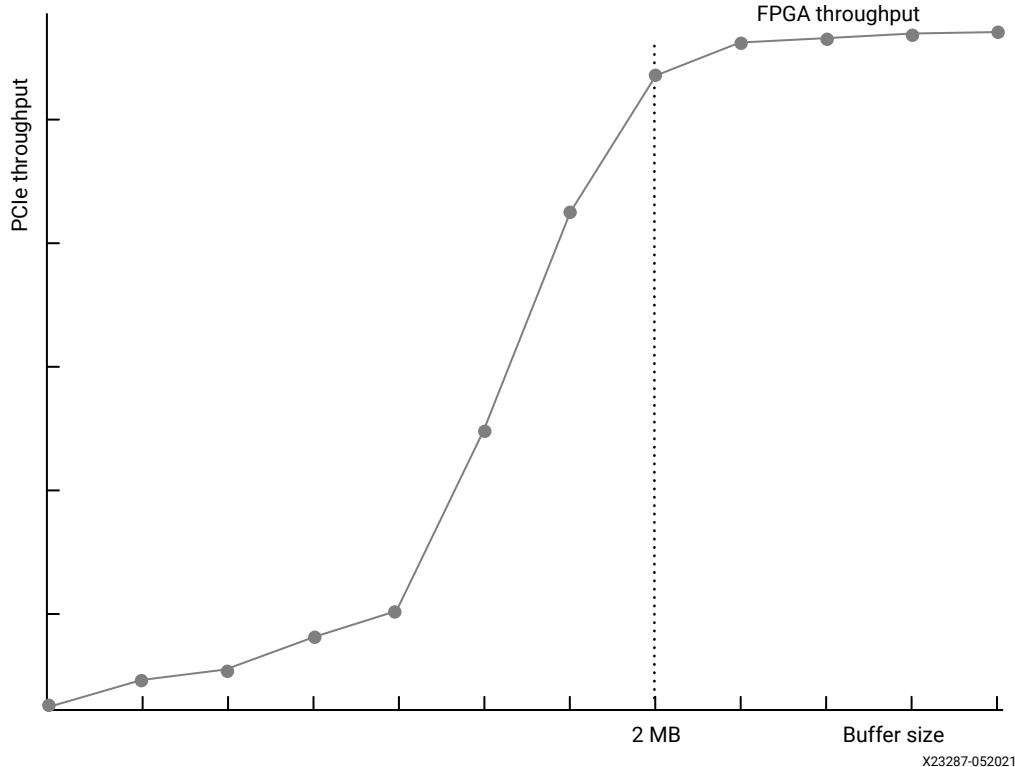
In an accelerated application, data must be transferred from the host to the device especially in the case of PCIe-based applications. This introduces latency, which can be very costly to the overall performance of the application.

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 `xbutil` utility can be useful for finding the optimal buffer sizes.

**Figure 117: Performance of PCIe Transfers as a Function of Buffer Size**

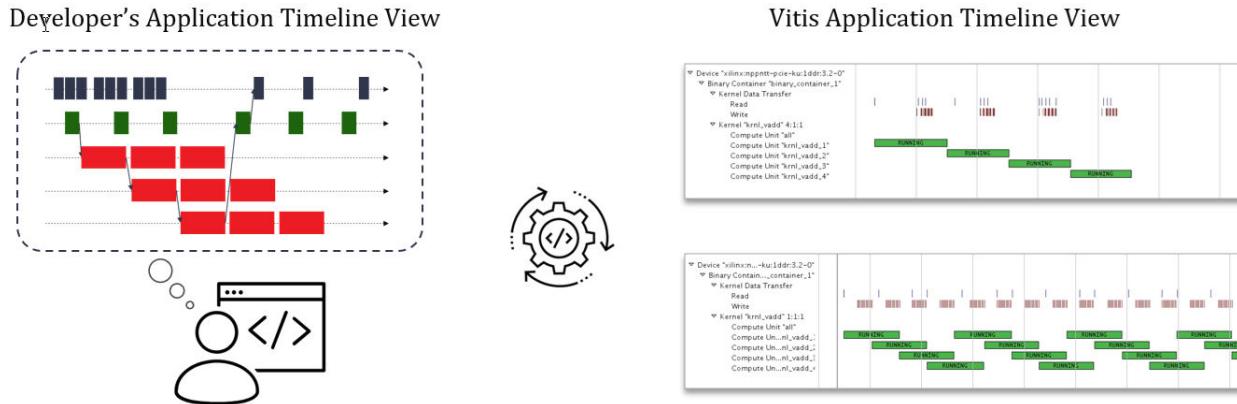
Xilinx recommends grouping multiple sets of data in a common buffer to achieve the highest possible throughput.

### **Conceptualize the Desired Application Timeline**

The developer should now have a good understanding of what functions need to be accelerated, what parallelism is needed to meet performance goals, and how 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 118: 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 have been finalized, the developer knows exactly how many kernels will be instantiated and therefore how many ports will need to be connected to global memory resources.

At this point, it is important to understand the features of the 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](#).

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 should have all the information necessary to start implementing the kernels, and ultimately, assembling the entire application.

---

# **Methodology for Developing C/C++ Kernels**

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

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

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

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

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

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

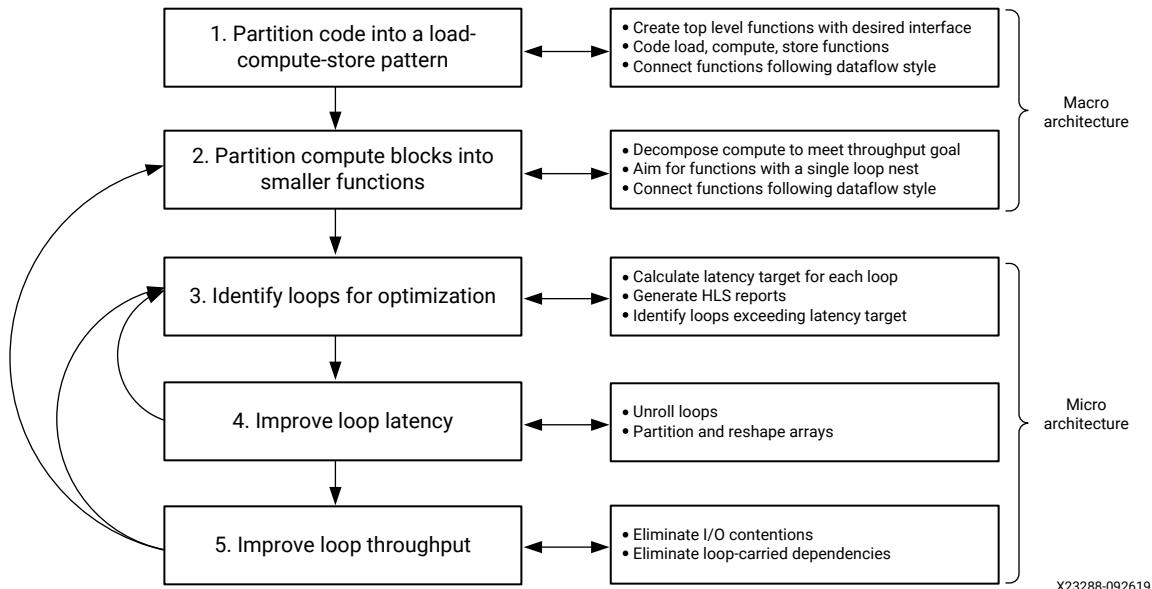
Before starting the kernel development process, it is essential to understand the difference between functionality, algorithm, and architecture; and how they pertain to the kernel development process.

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

You must understand that the Vitis compiler 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 as well as potential for computational parallelism, your choice of algorithm has a major impact on achievable performance, more so than the compiler's abilities or user specified pragmas. 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 119: Kernel Development Methodology**

## About the High-Level Synthesis Compiler

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

The HLS compiler is more restrictive than standard software compilers. For example, there are unsupported constructs including: system function calls, dynamic memory allocation and recursive functions. 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 will help 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. Xilinx® recommends verifying the code after each modification. This can be done using standard software verification methods or with the Vitis integrated design environment (IDE) software or hardware emulation flows. In either case, make sure your testing provides sufficient coverage and verification quality.

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

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

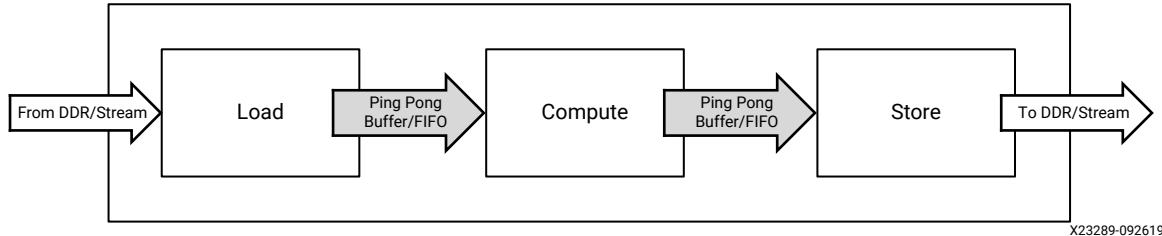
Knowing that kernel accesses to global memory are expensive and that bandwidth is limited, it is very important to carefully plan this aspect of the kernel.

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

This means creating a top-level function with:

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

Figure 120: 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 [Exploiting Task Level Parallelism: Dataflow Optimization](#) in the *Vitis HLS Flow*.

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 should be passed as scalars. Blocks of input and output data should be passed as pointers. Compiler pragmas should be used to finalize the interface definition.

## ***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.

Xilinx 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, then ports should 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 should always be avoided unless absolutely required. The load and store functions should be coded to always infer bursting transaction. This can be done using a `memcpy` operation as shown in the `vadd.cpp` file in the [Vitis Accel Examples](#), or by creating a tight `for` loop accessing all the required values sequentially, as explained in [Section III: Developing Applications](#).

## Minimize the Number of Data Transfers from Global Memory

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

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

## Code the Compute Functions

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

## Connect the Load, Compute, and Store Functions

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

Streaming is a type of data transfer in which data samples are sent in sequential order starting from the first sample. Streaming requires no address management and can be implemented with FIFOs. For more information about the `hls::stream` class, see the topic on Using HLS Streams in the Vitis HLS User Guide ([UG1399](#)).

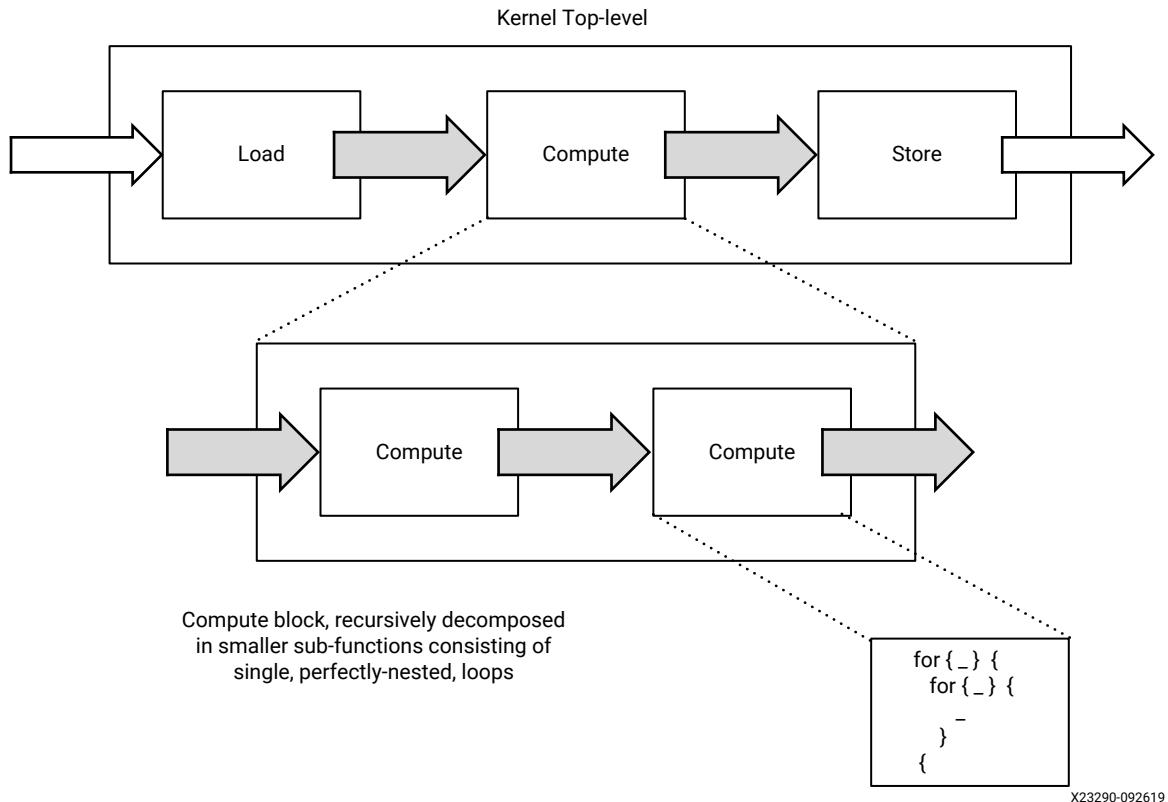
When connecting the functions, use the canonical form required by the HLS compiler. For more information, see the topic on Dataflow Optimization in the Vitis HLS User Guide ([UG1399](#)). This helps the compiler build a high-throughput set of tasks using the dataflow optimization. Key recommendations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

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

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

## Step 2: Partition the Compute Blocks into Smaller Functions

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

**Figure 121: Compute Block Sub-Functions**

## Decompose to Identify Throughput Goals

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

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

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

In the following steps of this methodology, the developer will get actual throughput numbers from running the 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 [Exploiting Task Level Parallelism: Dataflow Optimization](#) in the *Vitis HLS Flow*.

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

## **Connect Compute Functions Using the Dataflow ‘Canonical Form’**

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

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

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

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

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

## **Step 3: Identify Loops Requiring Optimization**

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

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

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

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

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

The latency of a loop can be calculated as follows:

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

Where:

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

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

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

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

Understanding this information is key to optimizing loops with latencies exceeding their target. By default, the 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.

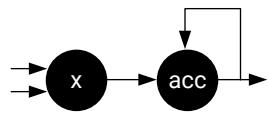
### Apply Loop Unrolling

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.

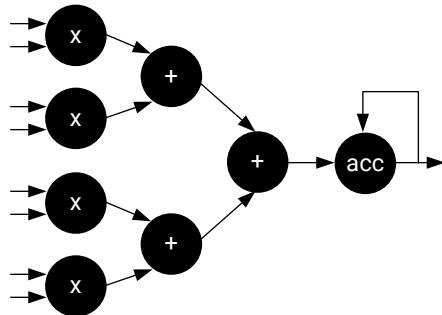
Figure 122: Loop Unrolling

```
for (int i = 0; i < N; i++)
{
    acc += A[i] + B[i];
}
```



1x datapath width  
N iterations  
1 sample per iteration

```
for (int i = 0; i < N; i++)
{
    #pragma HLS UNROLL factor=4
    acc += A[i] + B[i];
}
```



4x datapath width  
N/4 iterations  
4 samples per iteration

X23291-092619

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

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

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

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

## Apply Array Partitioning

Unrolling loops changes the I/O requirements and data access patterns of the function. If a loop makes array accesses, as is almost always the case, 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 to Improve Pipelining](#) 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 may not always be able to statically resolve potential dependencies and will in this case report false dependencies.

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

- Eliminating dependencies on scalars

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

## Advanced Techniques

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

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

---

# 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 [Methodology for Accelerating Data Center Applications with the 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 Xilinx FPGAs contain multiple partitions called super logic regions (SLRs). Keep the kernel in the same SLR as the global memory bank that it accesses.
- Use software and hardware emulation to validate your code frequently to make sure it is functionally correct.
- Frequently review the Vitis Guidance report as it provides clear and actionable feedback regarding deficiencies in your project.

# OpenCL Programming

## OpenCL Host Application

In the 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](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](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](#).

**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 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.

Xilinx does not provide the OpenCL ICD library, so the following should 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 Xilinx devices. The following code fragment shows a common method of identifying a Xilinx platform.

```
cl_platform_id platform_id;           // platform id
err = clGetPlatformIDs(16, platforms, &platform_count);

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

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

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



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

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

## Devices

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

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

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

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

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

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



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

## Sub-Devices

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

The following example shows:

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

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

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

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

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

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

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

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

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



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

## Context

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

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

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

## Command Queues

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

1. Single out-of-order command queue: Multiple kernel executions can be requested through the same command queue. XRT dispatches kernels as soon as possible, in any order, allowing concurrent kernel execution on the FPGA.
2. Multiple in-order command queue: Each kernel execution 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, Xilinx 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 in the FPGA

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

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

### Setting Up Kernels

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

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

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

### Setting Kernel Arguments

In the Vitis software platform, two types of arguments can be set for 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 enqueueing the kernel, you should set kernel arguments as early as possible. XRT will error out if you try to migrate a buffer before XRT knows where to put it on the device. Therefore, set the kernel arguments before performing any enqueue operation (for example, `clEnqueueMigrateMemObjects`) on any buffer.

---

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, you should 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, Xilinx 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 should 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 `clEnqueueMigrateMemObject`s. 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, nullptr, &err);
auto host_read_ptr =
clEnqueueMapBuffer(queue, dev_mem_write_ptr, true, CL_MAP_READ, 0, bytes, 0, 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
```

To work with an example using `clEnqueueMapBuffer`, refer to [Data Transfer \(C\)](#) in the [Vitis Examples](#) GitHub repository.

## Using Host Pointer Buffers



**IMPORTANT!** Using `CL_MEM_USE_HOST_PTR` is not recommended for embedded platforms. Embedded platforms require contiguous memory allocation and should 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.

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



**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. Hence you should align the host memory pointer with the 4K boundary to save the extra memory copy operation.

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

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

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

cl_mem dev_mem_ptr = clCreateBuffer(context,
                                     CL_MEM_READ_WRITE | 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 should only read the exact amount of data that the kernel has written.

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

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

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



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

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

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

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

// 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
```

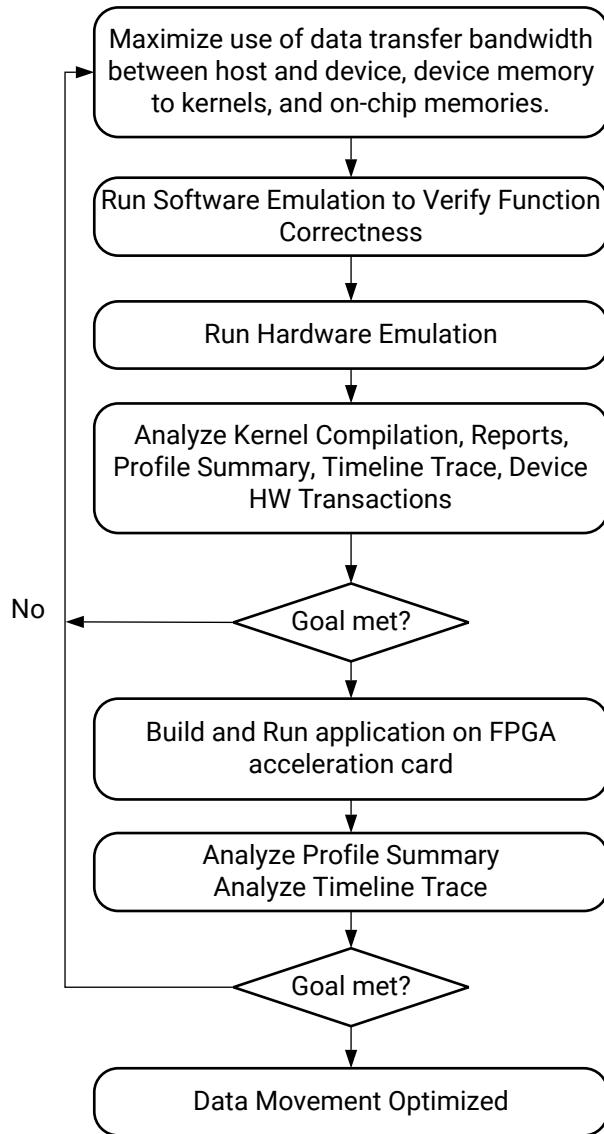
### Device Buffer Shared by Multiple Memory Ports or Multiple Kernels

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

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

## Optimizing Data Movement

Figure 123: Optimizing Data Movement Flow



X22239-102320

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 levels of memory 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. You should 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, as well as execution in hardware to achieve optimum performance.

### Overlapping Data Transfers with Kernel Computation

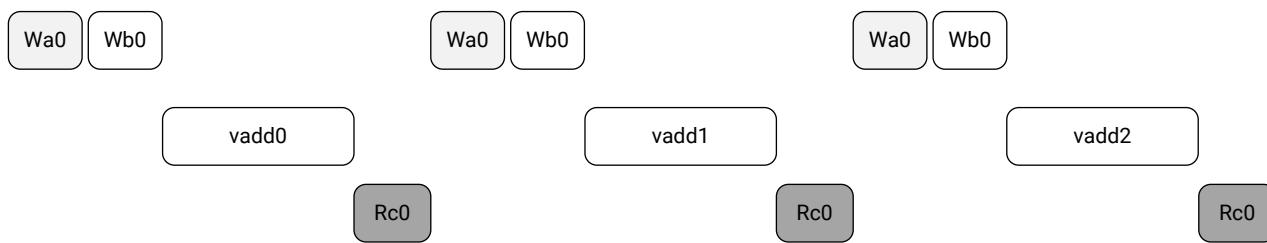
Applications, such as database analytics, have a much larger data set than can be stored in the available global device memory on the acceleration device. They require the complete data to be transferred and processed in blocks. Techniques that overlap the data transfers with the computation are critical to achieve high performance for these applications.

An example can be found in the `vadd` kernel from the [overlap](#) example in the [host](#) category of [Vitis Accelerated Examples](#) on GitHub. This examples demonstrates techniques to overlap Host (CPU) and FPGA computation in the application. In this 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 should look similar to the one shown below:

Figure 124: Host View of Tasks



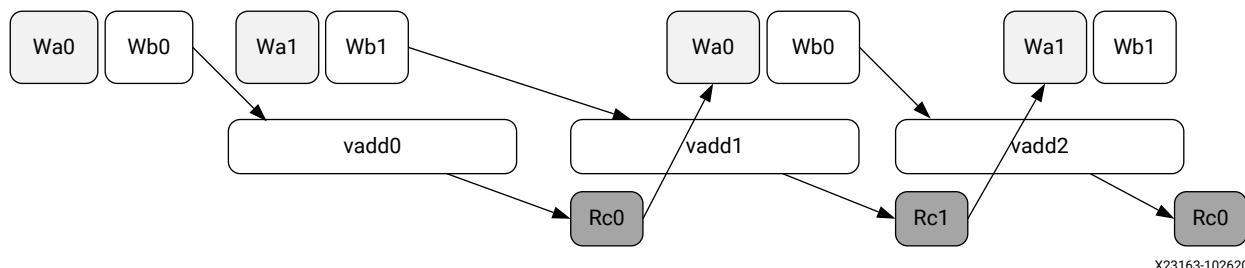
X24770-102720

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.

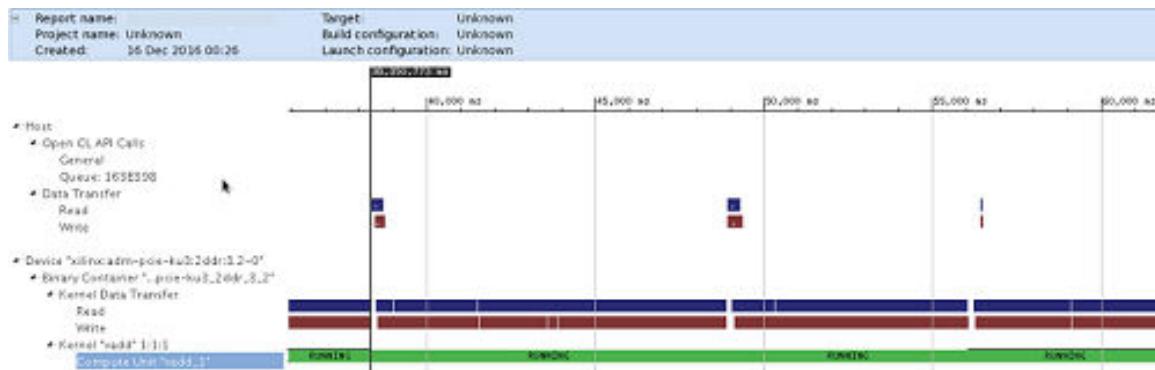
**Figure 125: Event Triggering Setup**



In the example, the host code (`host.cpp`) 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 126: 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 should be continuous in hardware. For large memory, it might take some time to wait for that space to be freed, when many buffers are allocated and deallocated. This can be resolved by allocating device buffer and reusing it between different enqueues of a kernel.

## 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 `c1EnqueueTask` as shown below.

```
err = c1EnqueueTask(commands, kernel, 0, NULL, NULL);
```

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



**IMPORTANT!** Though using `c1EnqueueNDRangeKernel` is supported (only for OpenCL kernel), Xilinx recommends using `c1EnqueueTask`.

However, sometimes using a single `c1EnqueueTask` 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 may 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 Enqueuing

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, you should 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 may be free to begin processing a new set of data. In essence, like a factory assembly line, the kernel can accept new data while the original data moves down the line.

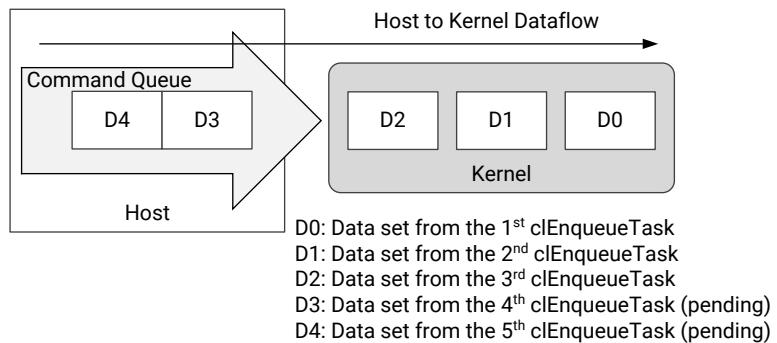
To understand this approach, assume a kernel has only one CU on the FPGA, and the host application enqueues the kernel multiple times with different sets of data. As shown in [Using Host Pointer Buffers](#), the host application can migrate data to the device global memory ahead of the kernel execution, 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 127: Host to Kernel Dataflow



X22774-042519

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

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



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

## 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

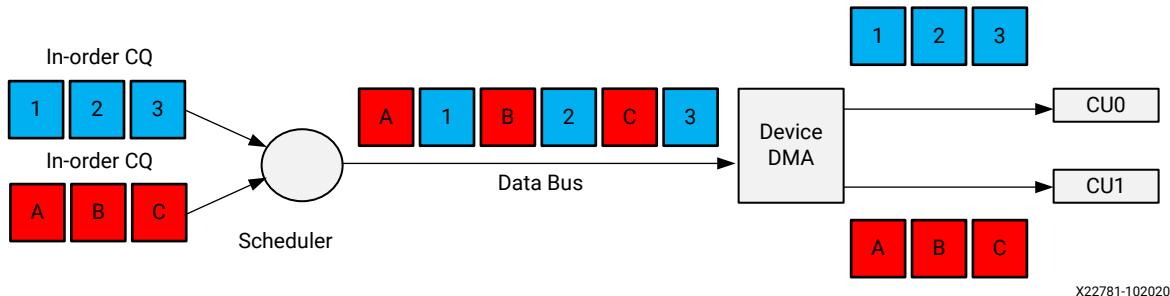
Scheduling kernel operations is key to overall system performance. This becomes even more important when implementing multiple compute units (of the same kernel or of different kernels). This section examines the different command queues responsible for scheduling the kernels.

### Multiple In-Order Command Queues

 **RECOMMENDED:** For improved performance, Xilinx 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 128: 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,
```

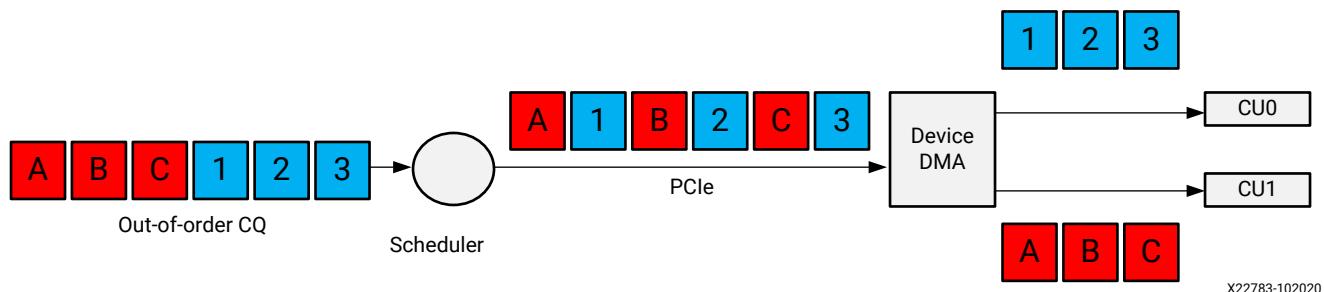
```

        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 129: Example with Single Out-of-Order Command Queue*



X22783-102020

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:

```

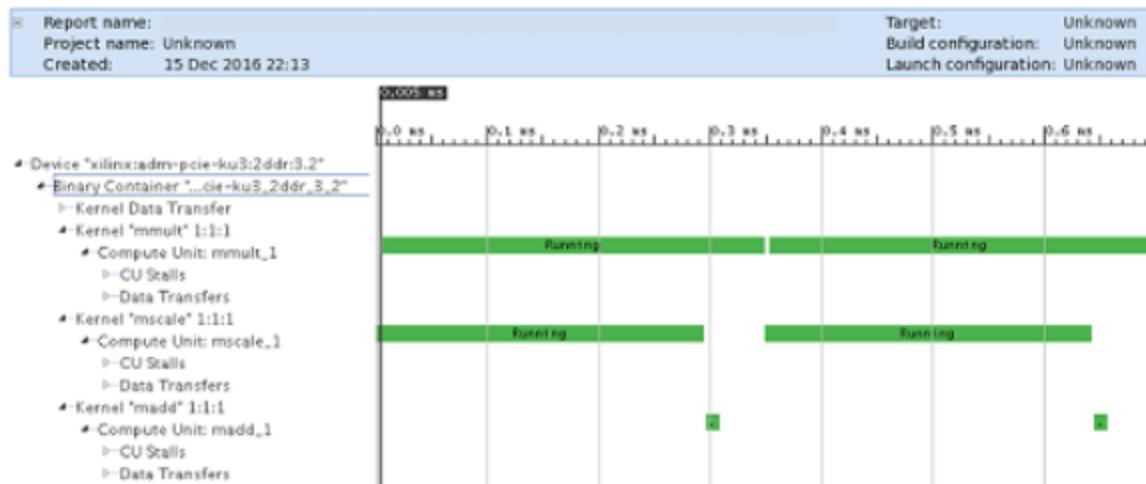
OCL_CHECK(
    err,
    cl::CommandQueue ooo_queue(context,
        device,
        CL_QUEUE_PROFILING_ENABLE | 
```

```

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_mmultiplication,
        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 130: 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 how the commands have been used in the example above:

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

## 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 may need to manually assign the buffer location (or special property) in the host code. For this purpose, the Xilinx 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 Xilinx 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 should 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);
```

# Streaming Data Transfers

While transferring data from the host typically requires memory mapped interfaces (`m_axi`) to access global memory, or directly access host memory on some platforms, the Vitis™ core development kit also supports streaming data transfer between kernels. This lets you create kernels that access data from the host system, and then stream it directly to other kernels.

Consider the situation where one kernel is performing some part of the computation, and a second or third kernel completes the operation after receiving the data from the first kernel. With kernel-to-kernel streaming support, data can move directly from one kernel to another without having to transmit back through the global memory. This results in a significant performance improvement. Finally, the data can be passed back to the host application through global memory. An example of this can be found in the [Mixed Kernels Design Tutorial with AXI Stream and Vitis](#) on GitHub.

---

## Streaming Kernel Coding Guidelines

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

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

```
// Producer kernel - provides output as a data stream
// For simplicity the example only shows the streaming output.

void kernel1 (.... , hls::stream<ap_axiu<32, 0, 0, 0>>& stream_out) {

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

// Consumer kernel - reads data stream as input
// For simplicity the example only shows the streaming input.

void kernel2 (hls::stream<ap_axiu<32, 0, 0, 0>>& stream_in, .... ) {

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

```
    int a = v.data; // Extract the data
    // Do further processing
}
```

Because the `hls::stream` data type is defined, the Vitis HLS tool infers `axis` interfaces. The following INTERFACE pragmas are shown as an example, but are not required in the code.

```
#pragma HLS INTERFACE axis port=stream_out
#pragma HLS INTERFACE axis port=stream_in
```



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

## Free-Running Kernels

While the Vitis core development kit provides support for streaming interfaces on PL kernels, it also supports a special kind of data-driven kernel called a free-running kernel. Free-running kernels have no control signal ports, no mechanism for interaction with a software application, and cannot be started or stopped. Free-running kernels have the following characteristics:

- When the device is programmed by the binary container (`xclbin`), free-running kernels start running automatically and do not need to be started from a software application.
- Has no memory input or output port, and therefore interacts with other kernels only through input or output streams
- The kernel operates on the stream data as soon as the data is received and the kernel stalls when data is not available.

The main advantage of a free-running kernel is that it does not follow the C-semantics where all the functions should be executed an equal number of times. This modeling style is more like RTL designs as shown in the example below. The compiler models the kernel to restart automatically after the previous function call finishes. This functionality is similar to a `while(1)` loop in software code, without having to specify the loop in the kernel code.

A free-running kernel only contains `hls::stream` inputs and outputs. The recommended coding guidelines include:

- Use `hls::stream<ap_axiu<D, 0, 0, 0>>` for the kernel interfaces.
- The `hls::stream` data type for the function parameter causes Vitis HLS to infer an AXI4-Stream port (`axis`) for the interface.
- The kernel only supports streaming interfaces (`axis`). It should not use `m_axi` or `s_axilite` interfaces, as shown in the example below.

- The free-running kernel must also specify the following block control protocol using the INTERFACE pragma.

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



**TIP:** This will create a kernel without control signals or an AXI4-Lite interface. This modeling technique is called a free-running kernel, as it is free of any control handshake and will start automatically and run continuously. For kernels requiring some interaction with software applications, consider using auto-restarting kernels as described in Vitis High-Level Synthesis User Guide ([UG1399](#)).

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

```
void increment(hls::stream<ap_axiu<32, 0, 0, 0>>& input,
    hls::stream<ap_axiu<32, 0, 0, 0>>& output){
#pragma HLS interface ap_ctrl_none port = return
    ap_axiu<32, 0, 0, 0> v = input.read();
    v.data = v.data + 1;
    output.write(v);
    if (v.last){
        break;
    }
}
```



**IMPORTANT!** Software emulation is not supported for free-running kernels.

# Migrating to a New Target Platform

This migration content is intended for users who need to migrate their accelerated Vitis™ technology application from one target platform to another. For example, moving an application from an Alveo™ U200 Data Center accelerator card, to an Alveo U280 card.

The following sections are included:

- [Design Migration](#): An overview of the Design Migration Process including the physical aspects of FPGA devices.
- [Migrating Releases](#): Any changes to the host code and design constraints if a new release is used.
- [Modifying Kernel Placement](#): Controlling kernel placements and DDR interface connections.
- [Address Timing](#): Timing issues in the new target platform which might require additional options to achieve performance.

---

## Design Migration

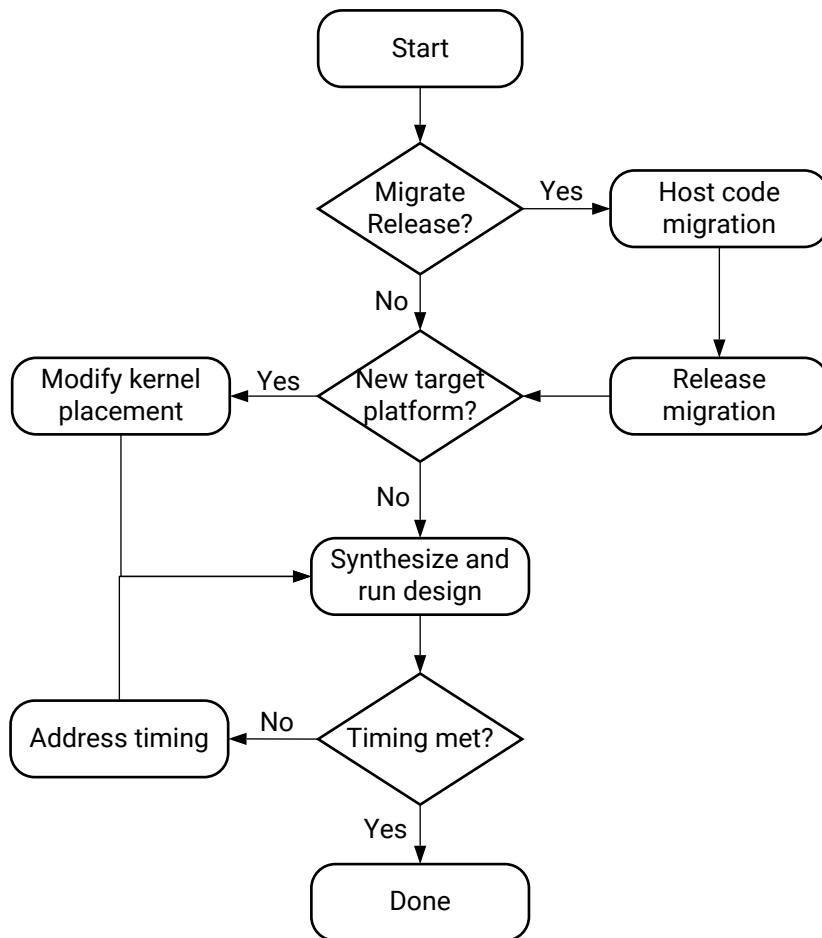
When migrating an application implemented in one target platform to another, it is important to understand the differences between the target platforms and the impact those differences have on the design.

Key considerations:

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

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

Figure 131: Target Platform Migration Flowchart



X21401-092519



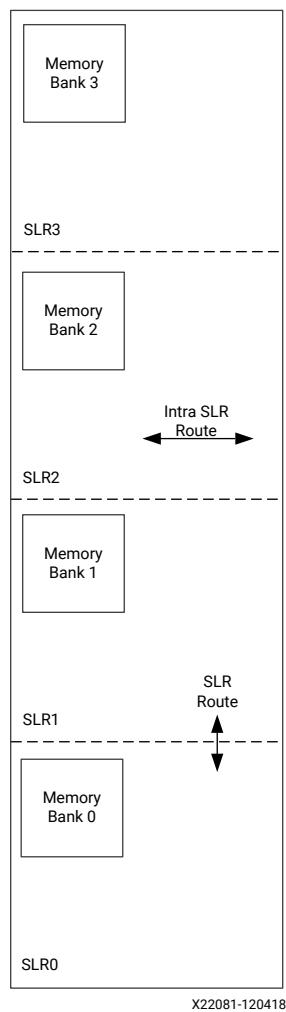
**IMPORTANT!** Before starting to migrate a design, it is important to understand the architecture of an FPGA and the target platform.

## Understanding an FPGA Architecture

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

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

Figure 132: Physical View of Xilinx FPGA with Four SLR Regions



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

## Stacked Silicon Interconnect Devices

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

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

One or more kernels can be implemented within an SLR. A single kernel 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.

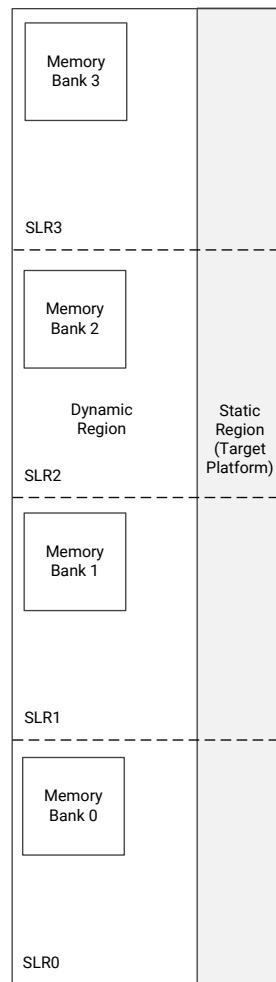
# Understanding Target Platforms

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 133: Target Platform on an FPGA with Four SLR Regions**



X22082-092519

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

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

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

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

Details on how much logic can be implemented in the dynamic region of each target platform is provided in the [Vitis Software Platform Release Notes](#). This topic is also addressed in [Modifying Kernel Placement](#).

---

## Migrating Releases

Before migrating to a new target platform, you should also determine if you will need to target the new platform to a different release of the Vitis technology. If you intend to target a new release, Xilinx highly recommends to first target the existing platform using the new software release to confirm there are no changes required, and then migrate to a new target platform.

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

- Host Code Migration
- Release Migration



**IMPORTANT!** Before migrating to a new release, Xilinx recommends that you review the [Vitis Software Platform Release Notes](#).

---

## 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.

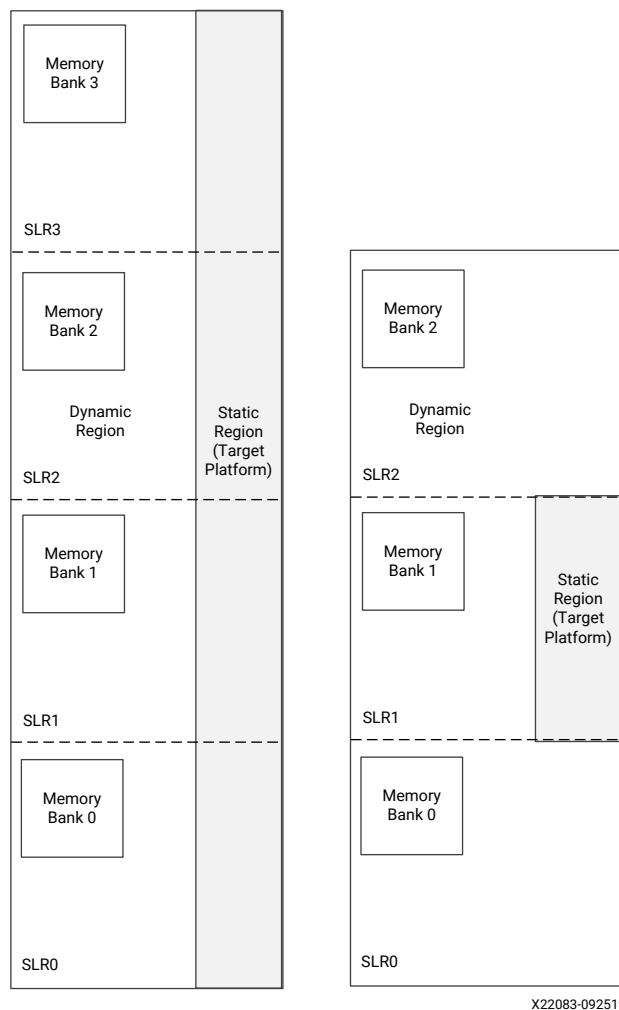
Address these issues using the same techniques you would use during the development of any kernel. At this stage, ensure the throughput performance of the target platform using the new release meets your requirements. If there are changes to the final timing (the maximum clock frequency), you can address 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 134: Comparison of Target Platforms of the Hardware Platform**

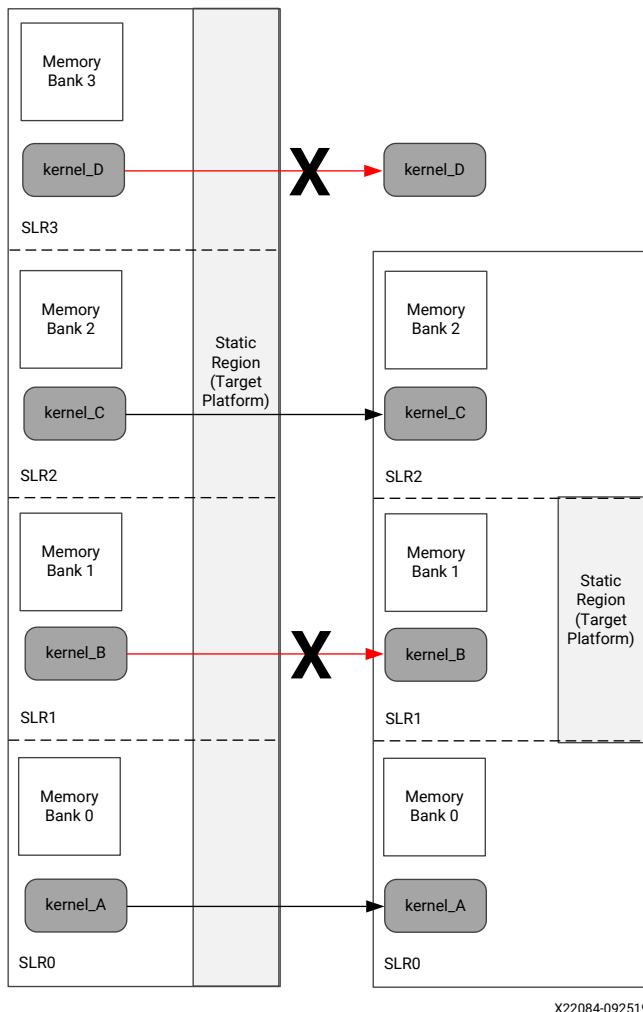
This section explains how to modify the placement of the kernels.

## 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 135: Migrating Platforms – Kernel Placement



X22084-092519

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

- Understand the resources available in each SLR of the new target platform, as documented in the [Vitis Software Platform Release Notes](#).
- Understand the resources required by each kernel in the design.
- Use the `v++ --config` option to specify which SLR each kernel is placed in, and which DDR bank each kernel connects to. For more details, refer to [Assigning Compute Units to SLRs 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](#). 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 68: 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>]
<b>Global memory resources available in dynamic region</b>			
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)
<b>Approximate available fabric resources in dynamic region</b>			
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 136: System Estimate Report

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

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

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

## Assigning Kernels to SLRs

Each kernel in a design can be assigned to an SLR region using the `connectivity.slr` option in a configuration file specified for the `v++ --config` command line option. Refer to [Assigning Compute Units to SLRs on Alveo Accelerator Cards](#) for more information.

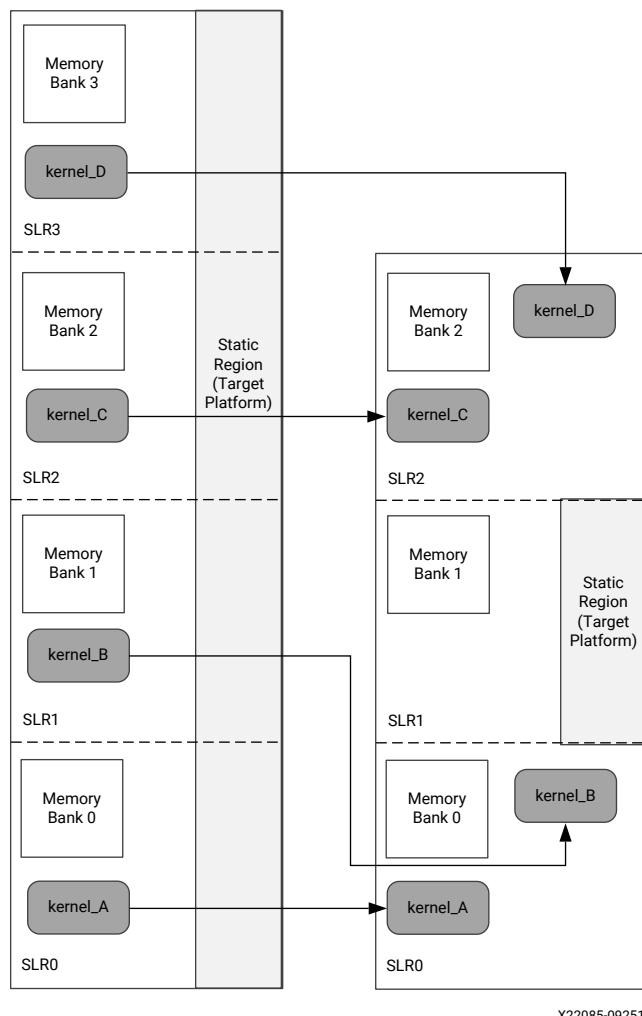
When placing kernels, Xilinx recommends assigning the specific DDR memory bank that the kernel will connect to using the `connectivity.sp config` option as described in [Mapping Kernel Ports to Memory](#).

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 137: Mapping of Kernels Across SLRs



X22085-092519

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

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

- Kernel\_A is connected to Memory Bank 0.
- Kernel\_B is connected to Memory Bank 1.
- Kernel\_C is connected to Memory Bank 2.
- Kernel\_D is connected to Memory Bank 1.

The configuration file to perform these connections would be as follows, and passed through the `v++ --config` command:

```
[connectivity]
nk=kernel:4:kernel_A.kernel_B.kernel_C.kernel_D

slr=kernel_A:SLR0
slr=kernel_B:SLR0
slr=kernel_C:SLR2
slr=kernel_D:SLR2

sp=kernel_A.arg1:DDR[0]
sp=kernel_B.arg1:DDR[1]
sp=kernel_C.arg1:DDR[2]
sp=kernel_D.arg1:DDR[1]
```



**IMPORTANT!** When using the `connectivity.sp` option to assign kernel ports to memory banks, you must map all interfaces/ports of the kernel. Refer to [Mapping Kernel Ports to 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 Virtex UltraScale device. Any custom Tcl constraints for timing closure will need to be evaluated and might need to be modified for the new platform.

Custom constraints can be passed to the Vivado® tools using the `[advanced]` directives of the `v++` configuration file specified by the `--config` option. Refer to [Managing Vivado Synthesis and Implementation Results](#) more information.

## Timing Closure Considerations

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

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

The utilization guidelines provide a threshold above which the compilation of the design can take longer, or performance can be lower than initially estimated. For larger designs which usually require using more than one SLR, specify the kernel/DDR association with the `v++ --config` option, as described in [Mapping Kernel Ports to Memory](#), while verifying that any floorplan constraint ensures the following:

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

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

For quickly reviewing all aspects listed above, use the fail-fast reports generated throughout the Vitis application acceleration development flow using the `-R` option as described below (refer to [Controlling Report Generation](#) for more information):

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

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

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

For more information about timing closure and the fail-fast report, see the *UltraFast Design Methodology Guide for FPGAs and SOCs* ([UG949](#)).

# Output Structure of the Vitis Tools

## Output Directories of the v++ Command

The directory structure generated by the command-line flow has been organized to let you easily find and access files from the project. By navigating the various `compile`, `link`, `logs`, and `reports` directories, you can easily find generated files. Similarly, each kernel also has a directory structure created.

You can optionally change the directory structure using the following `v++` options:

```
--temp_dir <dir_name>
--log_dir <dir_name>
--report_dir <dir_name>
```

When using `v++` on the command line, by default it creates a directory structure during compile and link. The `.xo` and `xclbin` files are always generated in the current working directory. All the intermediate files are created under the directory specified by the `--temp_dir` option, which defaults to `_x` when `--temp_dir` is not specified. The `link`, `logs`, and `reports` directories default to inside of the `temp_dir`, and contain the respective information on the builds.

The example directory provided below results from the following command lines:

```
## Kernel Compilation command:
v++ -c -t sw_emu --platform xilinx_u200_gen3x16_xdma_2_202110_1 --
config ./src/u200.cfg \
-k vadd -I./src ./src/vadd.cpp -o hw_emu/vadd.xo

## Device Binary Linking Command:
v++ -l -t sw_emu --platform xilinx_u200_gen3x16_xdma_2_202110_1 --
config ./src/u200.cfg \
hw_emu/vadd.xo -o hw_emu/vadd.xclbin
```

The `u200.cfg` file defines the following options:

```
debug=1
save_temps=1

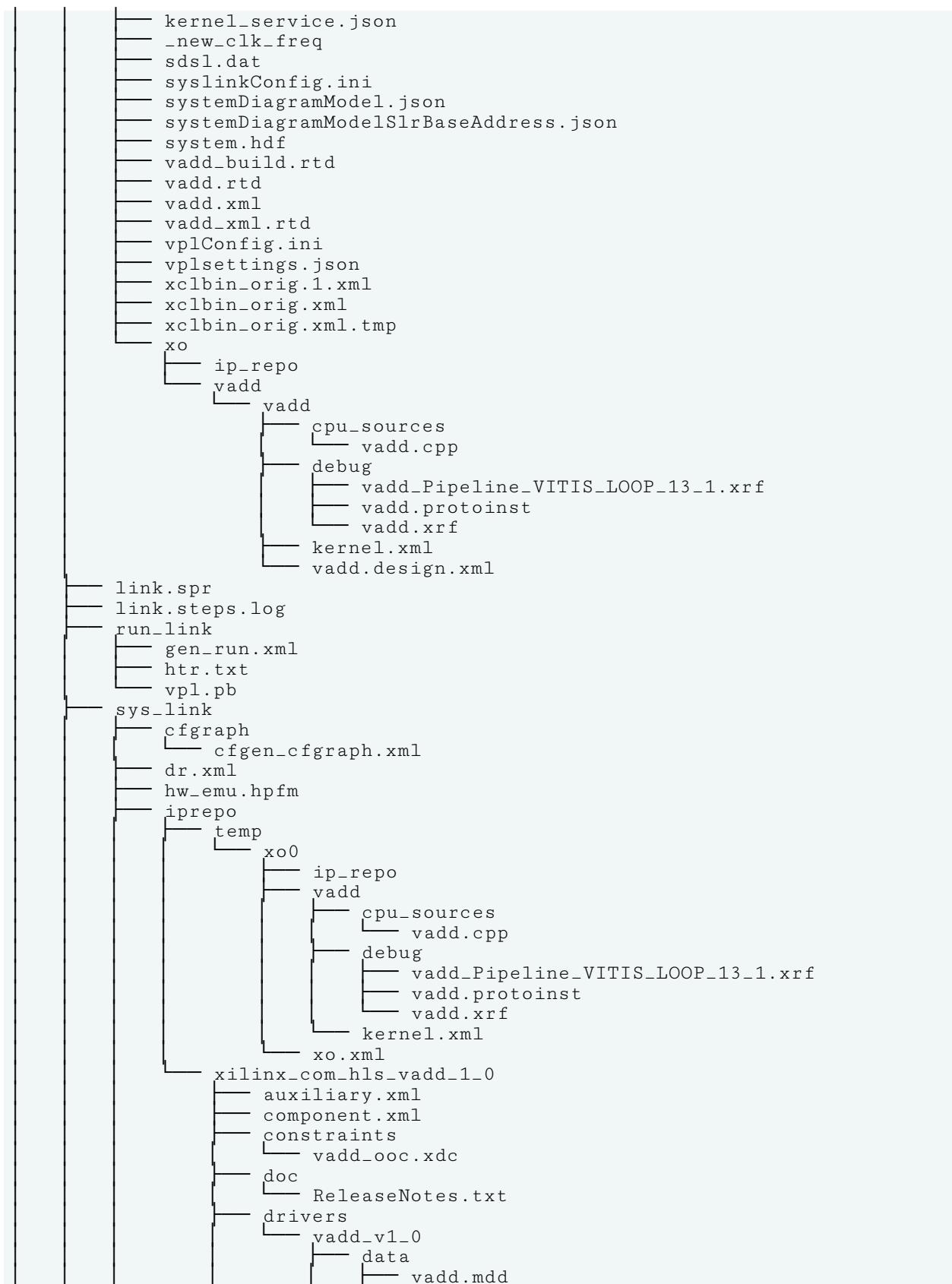
[connectivity]
nk=vadd:1:vadd_1
sp=vadd_1.in1:DDR[1]
```

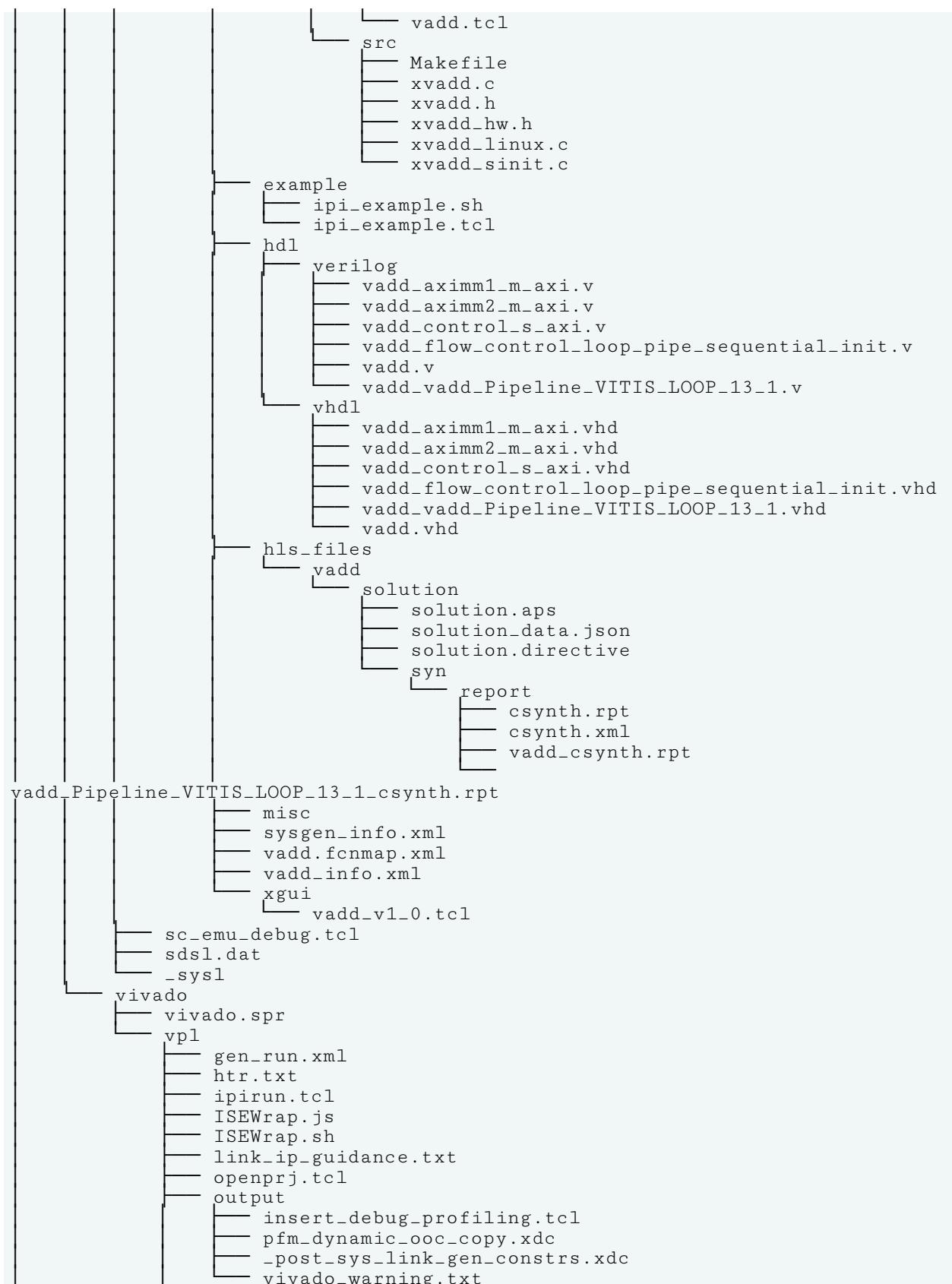
```
sp=vadd_1.in2:DDR[2]
sp=vadd_1.out:DDR[1]

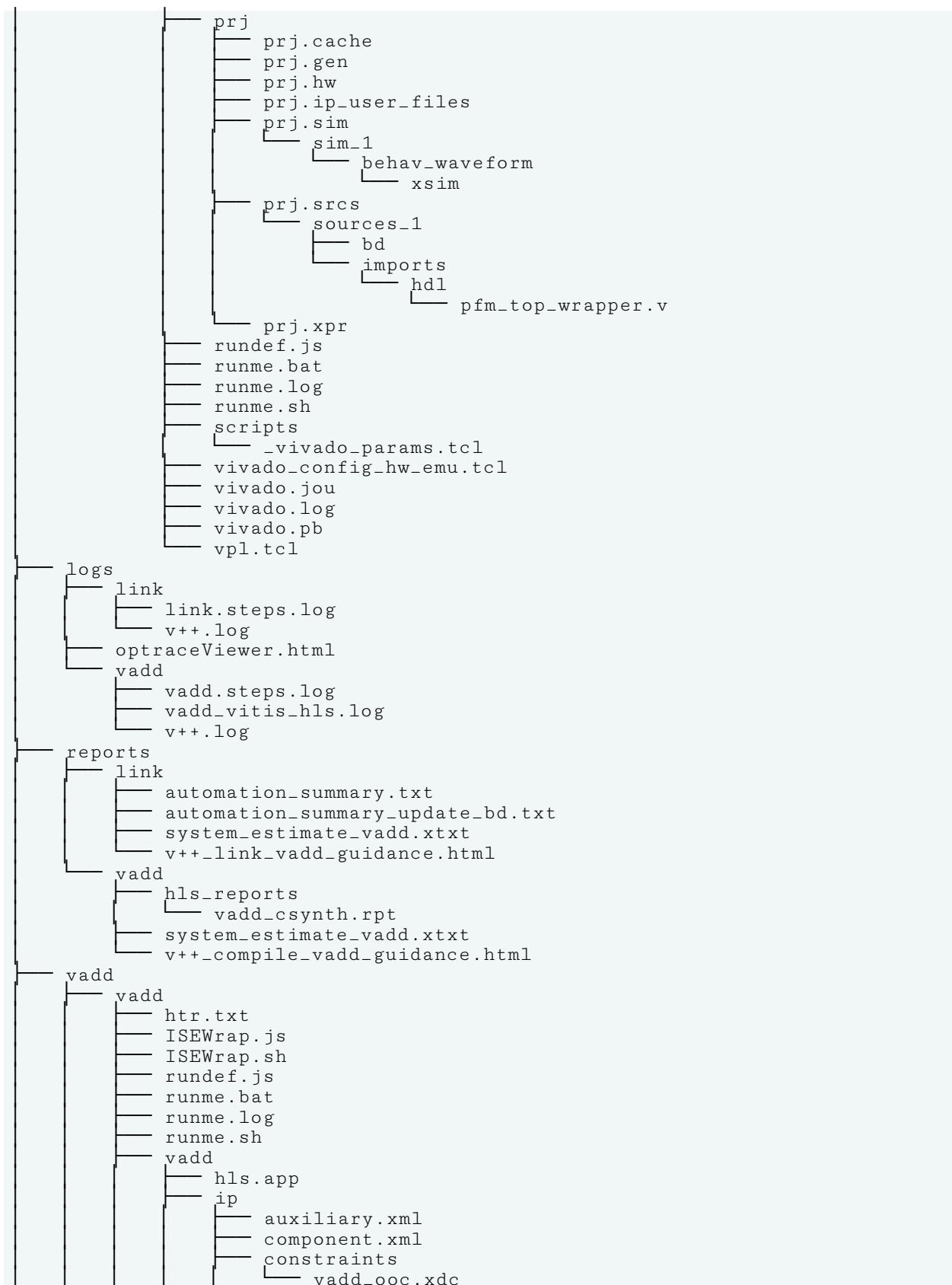
[profile]
data=all:all:all
```

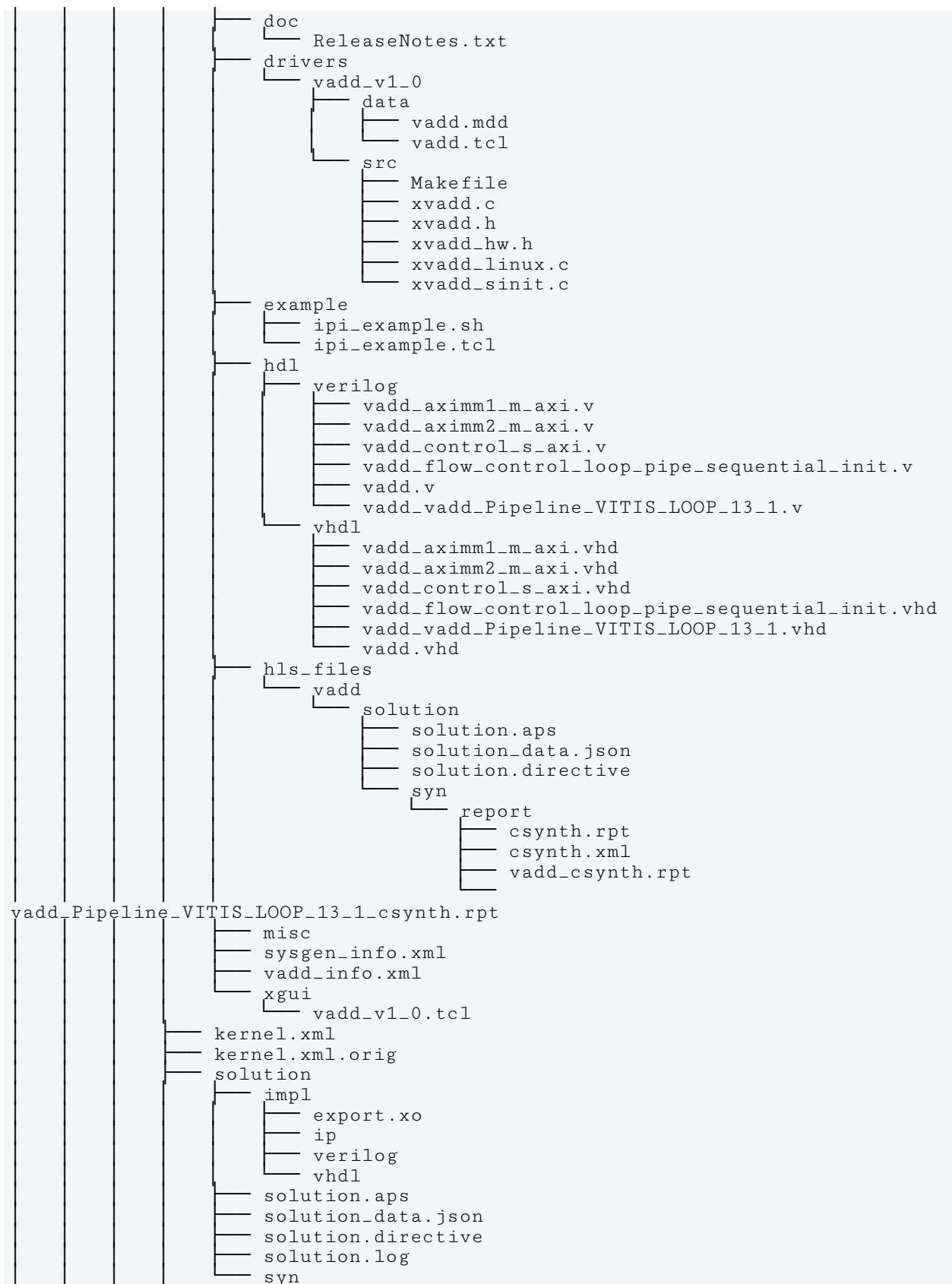
An example of the `temp_dir` output of the `v++ link` command follows:

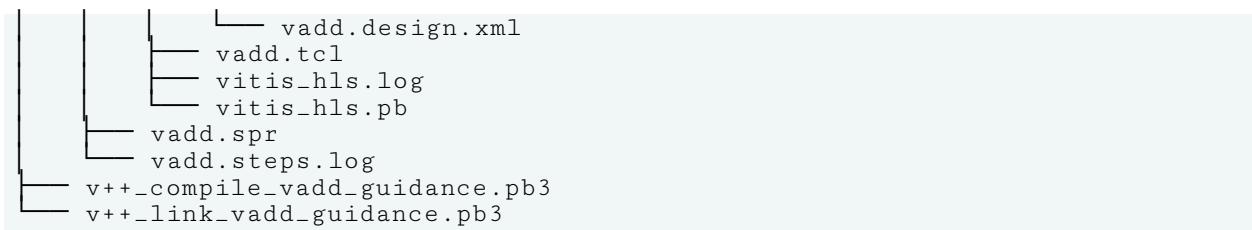
```
link
  activetask.json
  int
    address_map.xml
    appendSection.rtd
    automation_summary.txt
    automation_summary_update_bd.txt
    behav_waveform
      xsim
        compile.log
        compile.sh
        dr_behav.protoinst
        dr_behav.wcfg
        elaborate.log
        elaborate.sh
        glbl.v
        launch_hw_emu.sh
        libdapi.so
        pfm_top_wrapper.tcl
        pfm_top_wrapper_vhdl.prj
        pfm_top_wrapper_vlog.prj
        pfm_top_wrapper_xsc.prj
        post_sim_tool_scripts.tcl
        preprocess_profile.tcl
        pre_sim_tool_scripts.tcl
        prj.smi
        profile.tcl
        protoinst_files
        run.sh
        sc_xtlm_bd_d216_interconnect_DDR4_MEMORY0_0.mem
        sc_xtlm_bd_d216_interconnect_M0_AXI_MEMORY0_0.mem
        sc_xtlm_pfm_dynamic_smartconn_data_0_0.mem
        sc_xtlm_pfm_top_smartconnect_0_0.mem
        simulate.log
        simulate.sh
        vitis_params.tcl
        waveform_debug_enable.txt
        xelab.pb
        xsc.log
        xsc.pb
        xsim.dir
        xsim.ini
        xsim.ini.bak
        xvhd़.log
        xvhd़.pb
        xvlog.log
        xvlog.pb
      behav.xse
      cf2sw_full.rtd
      cf2sw.rtd
      debug_ip_layout.rtd
      dr.bd.tcl
      kernel_info.dat
      _kernel_inst_paths.dat
```











## Output Directories from the Vitis IDE

Unlike the command-line flow, which is defined largely by the user through command or Makefile, the Vitis IDE defines the structure of the projects and output directories in a system design project. In the Vitis IDE, an Application project for acceleration or heterogeneous design can have four different projects associated with it. The projects include:

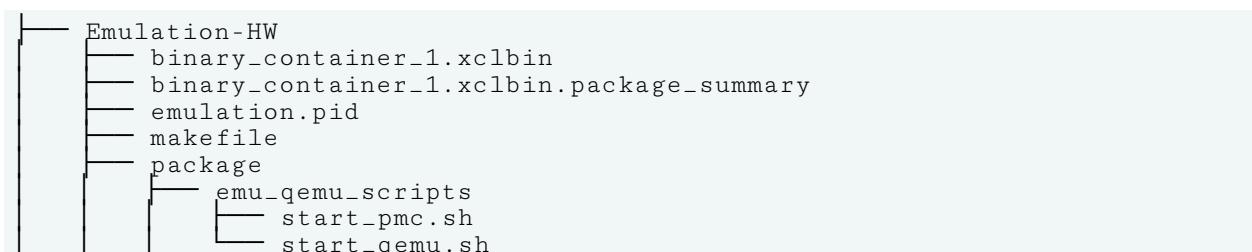
- Top-level System project: This is the project used to build the integrated system design, and is where you will find the consolidated contents of the packaged system design which is the result of the `v++ --package` process
- Software Application project: This contains the source and compiled results of the software application which runs on an x86 or Arm processor
- PL Kernels project: This project contains the source files for one or more PL kernels used by the system, and the results of the `v++ --compile` command
- Hardware Linking project: This project contains the linked system design which is the results of the `v++ --link` command



**TIP:** Each of the projects below starts with the `Emulation-HW` folder which is where the Vitis IDE places the build content for the hardware emulation build. The files presented here are for that build target. Some of the files at the deeper levels are not displayed, but can be found by navigating into the hierarchy of completed builds.

### Top-Level System Project

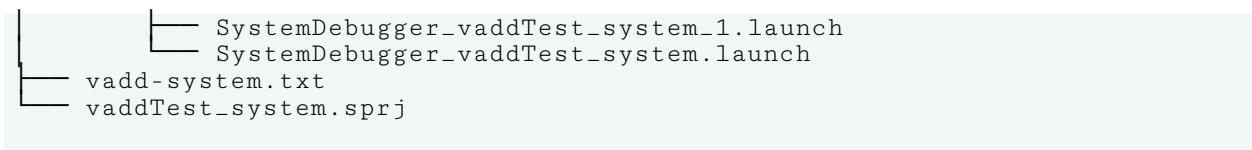
The top-level system project contains all of the other projects as sub-projects. This is where the definition and construction of the system design are provided. This includes the output of the package process, which generates the final fixed platform and device binary, and packages it into an SD card or QSPI image.



```

├── launch_hw_emu.sh
├── pmu_args.txt
├── qemu_args.txt
├── qemu_output.log
└── qemu_resize_img.sh
├── sd_card
│   ├── boot.scr
│   ├── emconfig.json
│   ├── Image
│   ├── vaddTest
│   └── vadd.xclbin
└── sd_card.img
└── sim
    └── behav_waveform
        └── xsim
            ├── xsim.ini
            ├── xsim.ini.bak
            ├── xsim.jou
            ├── xvhdl.log
            ├── xvhdl.pb
            ├── xvlog.log
            └── xvlog.pb
├── package.build
└── logs
    └── package
        └── package.steps.log
            └── v++.log
├── package
│   ├── behav.xse
│   ├── extractedSystemDiagram.json
│   ├── packagedSystemDiagram.json
│   ├── package.spr
│   └── package.steps.log
└── sim
    └── behav_waveform
        └── xsim
            ├── xsim.ini
            ├── xsim.ini.bak
            ├── xvhdl.log
            ├── xvhdl.pb
            ├── xvlog.log
            └── xvlog.pb
└── reports
    └── package
        └── v++_package_vadd_guidance.html
├── v++_package_binary_container_1_guidance.json
├── v++_package_binary_container_1_guidance.pb
├── v++_package_vadd_guidance.json
└── v++_package_vadd_guidance.pb
└── package.cfg
└── vaddTest_system_Emulation-HW.build.ui.log
└── vadd.xclbin
└── vadd.xclbin.package_summary
└── v++_binary_container_1.log
└── v++_vadd.log
└── xcd.log
└── xrc.log
└── ide
    └── launch

```



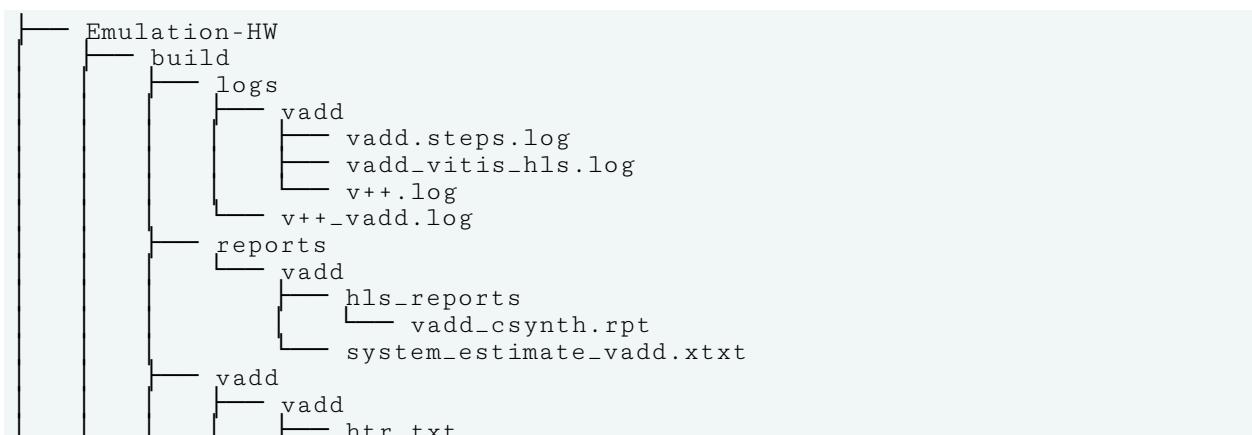
## Software Application Project

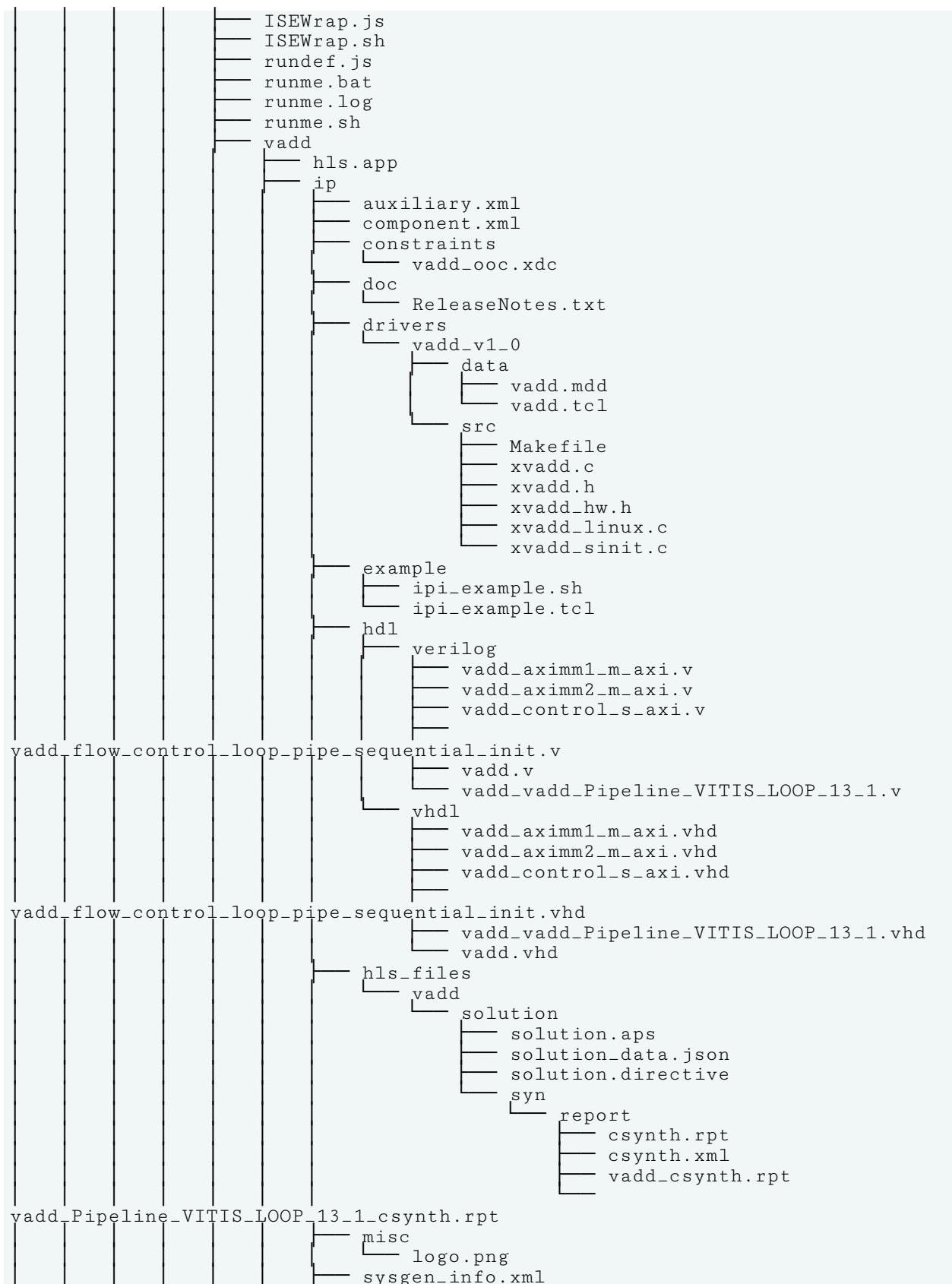
The software application project contains the source code for the software application, and the resulting .o object files, and .exe or .elf executable files.

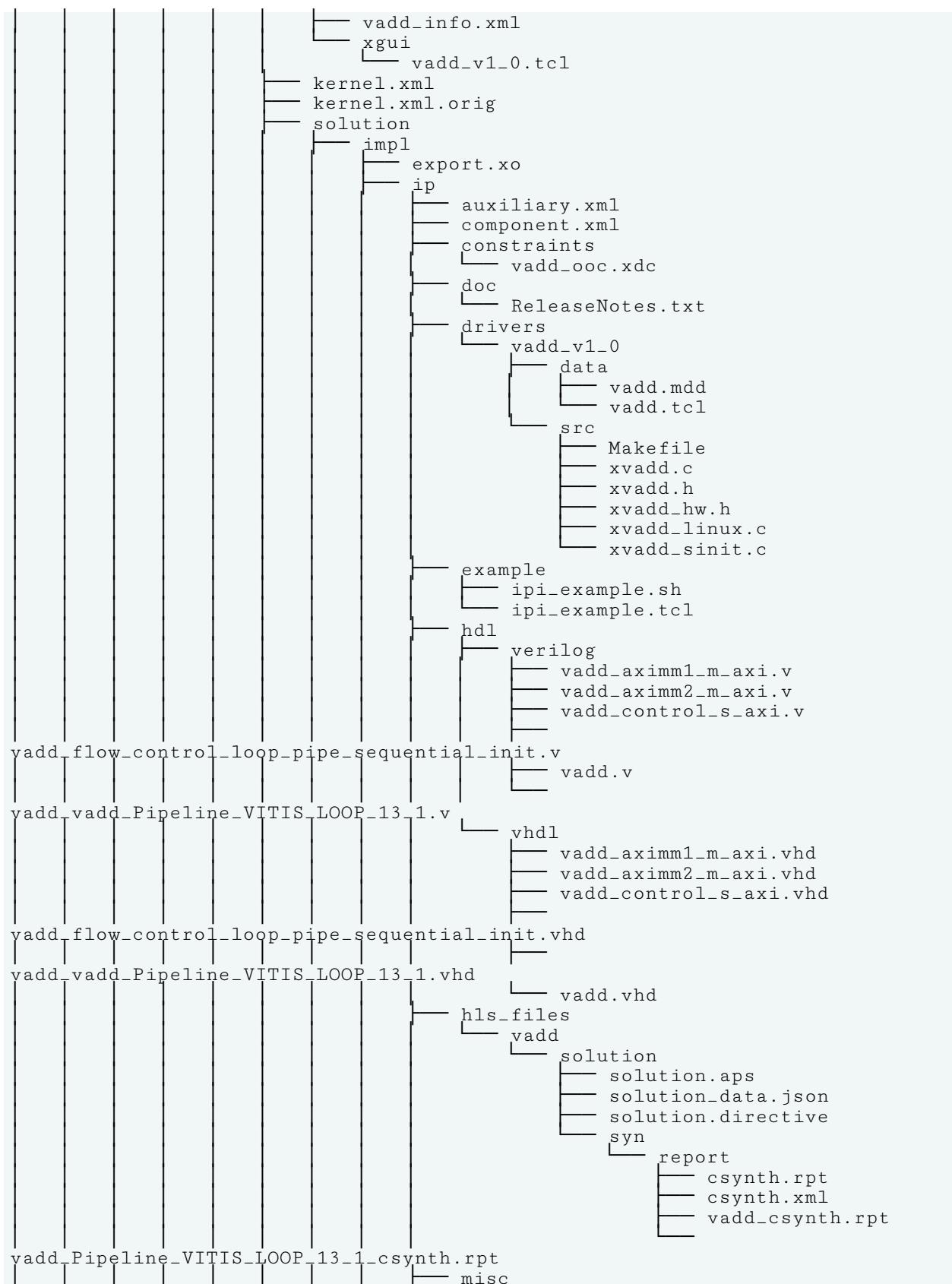


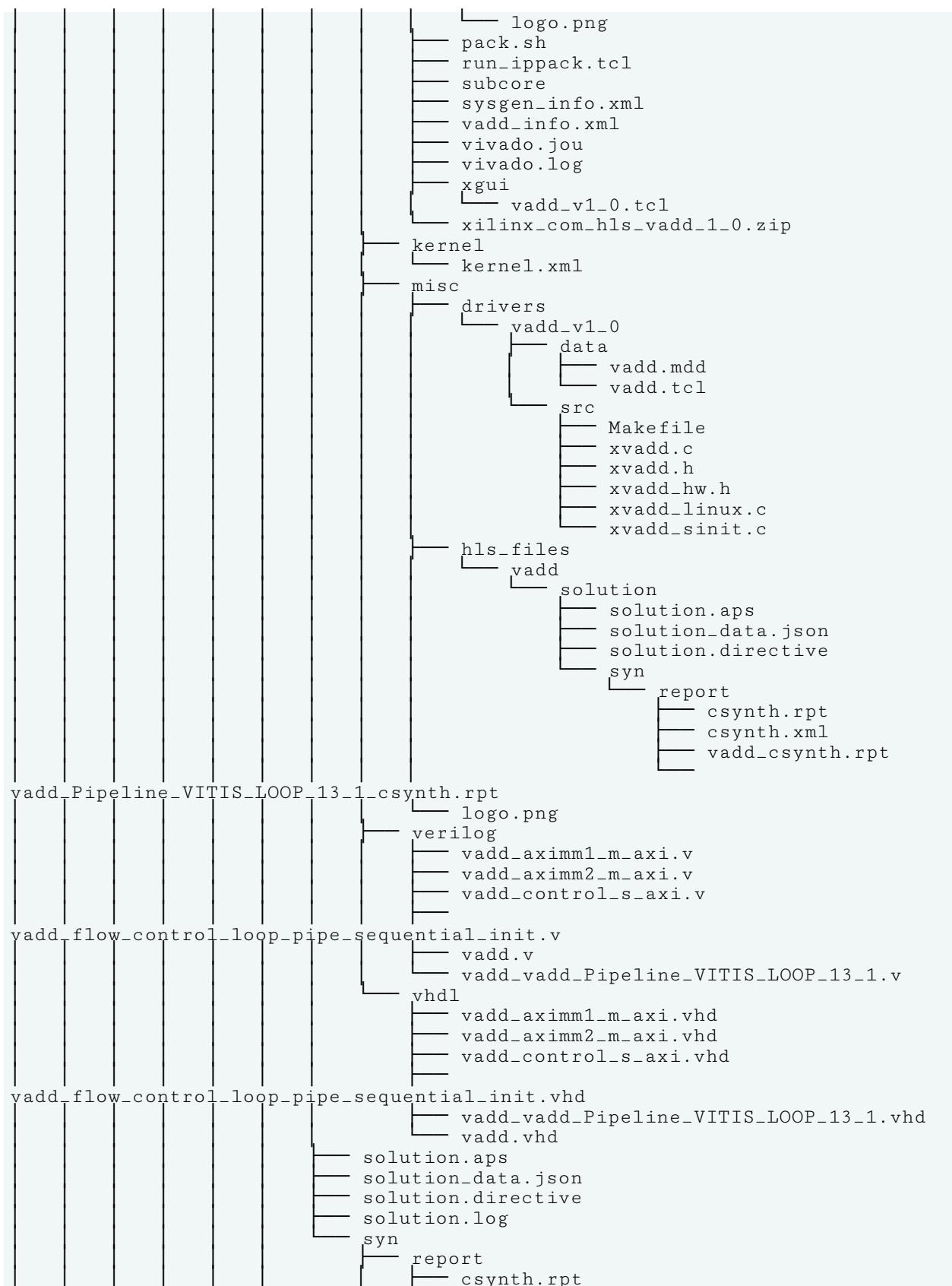
## PL Kernels Project

The PL kernels project contains the source code for C++ kernels and the compiled Xilinx object files (.xo), as well as RTL kernels (.xo), and libadff.a files containing AI Engine graph applications. The directory also holds the resulting logs and projects required by Vitis HLS to build the PL kernel objects, such as the .compile\_summary produced during compilation.







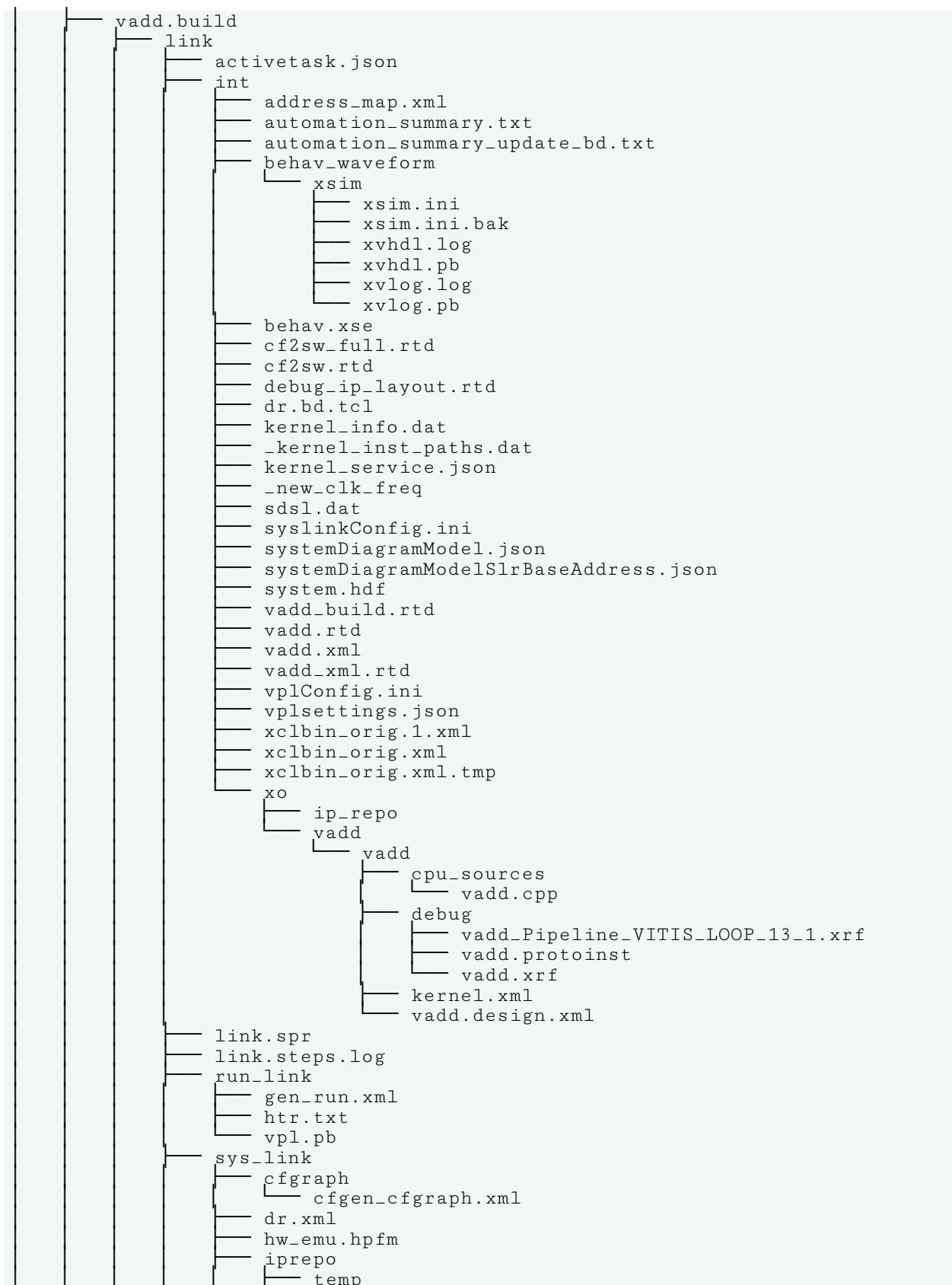


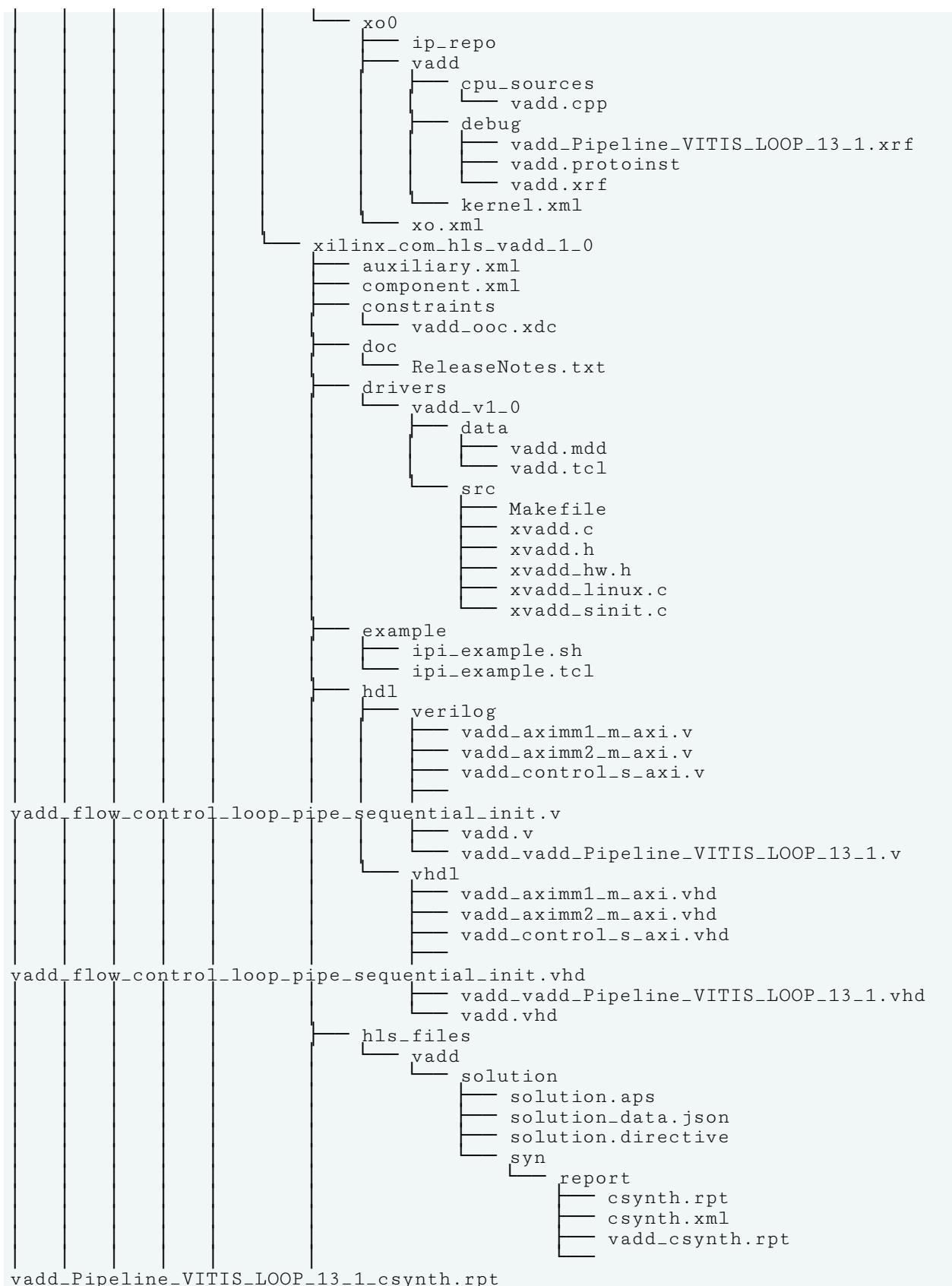


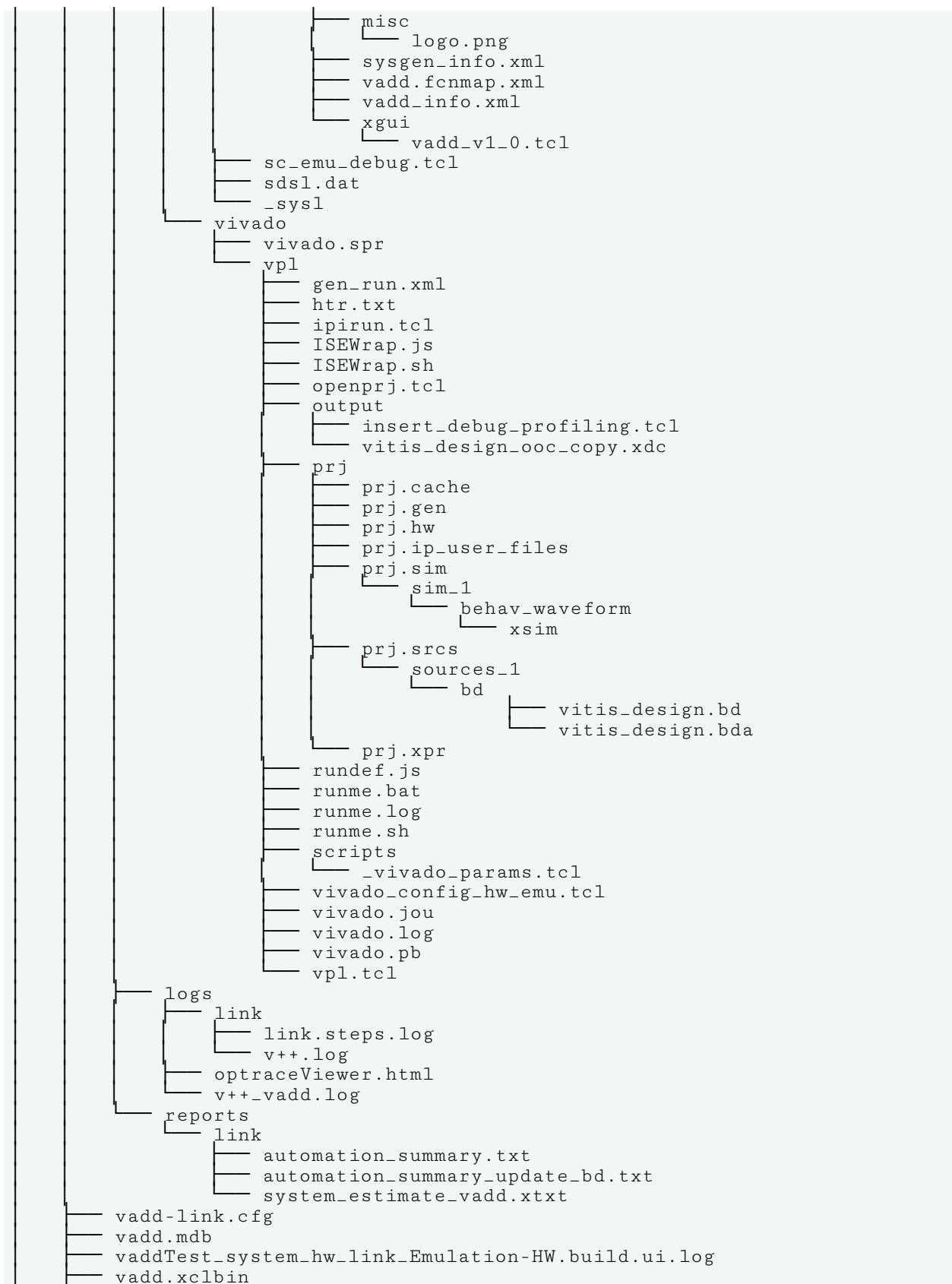
## Hardware Link Project

The hardware link project contains the linked fixed hardware platform (.xsa) and the device binary (.xclbin) used to program the Xilinx device. The directory also holds the resulting logs and projects produced during the linking process, such as the .link\_summary that can be viewed in the Vitis analyzer.









```
vadd.xclbin.info  
vadd.xclbin.link_summary  
vadd.xclbin.sh  
xcd.log  
vadd-hw-link.txt  
vaddTest_system_hw_link.prj
```

# Using Vitis System Compilation Mode

The Vitis™ System Compilation mode (shortened as VSC) lets you create a unified single-source C++ model that captures the system composition, containing both the hardware accelerator code and the host application code (based on C-Threads). For use only in Data Center acceleration projects, VSC creates a system structure that captures design intent specified in the C++ model and generates the necessary host code linking to Xilinx Runtime (XRT) to execute the design. The model incorporates a run time software layer that has many automation features for executing the accelerator hardware and managing data transfers. VSC uses Vitis compilation and linking tools to generate an FPGA bitstream and design-dependent object code, or software drivers, which can be linked with the rest of the application to create an executable.

VSC supports most Alveo Data Center accelerator cards with a list available at [Supported Platforms and Startup Examples](#).

This section contains the following chapters:

- [Introduction to Vitis System Compilation Mode](#)
- [Quick Start Example](#)
- [Creating a VSC Makefile](#)
- [Building Hardware](#)
- [Application Software Interface](#)
- [Debugging and Validation](#)
- [Supported Platforms and Startup Examples](#)

# Introduction to Vitis System Compilation Mode

A typical hardware acceleration flow consists of identifying compute-intensive portions of applications or algorithms and implementing them as custom-designed hardware. These custom-designed hardware elements are typically called accelerators in domains such as heterogeneous computing; in Vitis™ nomenclature, they are also called kernels. Generally, these functions or kernels take significant execution time when run in pure software on x86 or Arm®-based host machines. With the Vitis acceleration flow, once a function or set of functions is chosen for acceleration, Vitis HLS or an RTL flow based on the Vivado Design Suite can be used to design or generate custom hardware. The host application can be written using XRT native APIs, or OpenCL™ C or C++ that links to the Xilinx Runtime (XRT).

The process of designing an accelerated application consists of choosing the compute-intensive functions for acceleration and designing custom hardware to accelerate these functions. Additionally, you must write software that runs on a host machine that orchestrates data movements to and from the accelerator. An important and tedious step in the overall application design consists of optimizing the data movements, especially over PCIe® or a network interface. These data movements consist of transfers between the host and kernel through global memories or network ports, and can be optimized based on the nature of the transfers. The accelerated hardware can also be composed of multiple functions, either as repeated functionality to process data in parallel or as a connected pipeline. Such compositions can consist of the same or different accelerated function instances, or it can be a mix of software and hardware functions.

Thus, the perfect system architecture is developed over an iterative design process. Essentially there are three design aspects being developed together: the kernel code, the supporting application or host code, and the hardware/software system interface and composition. While improving the overall design performance, changing one design aspect will invariably require changing another aspect. Vitis™ System Compilation mode (shortened as VSC) provides a single source unified C++ model that can make the design exploration process fast and easy. VSC is designed to be a framework to:

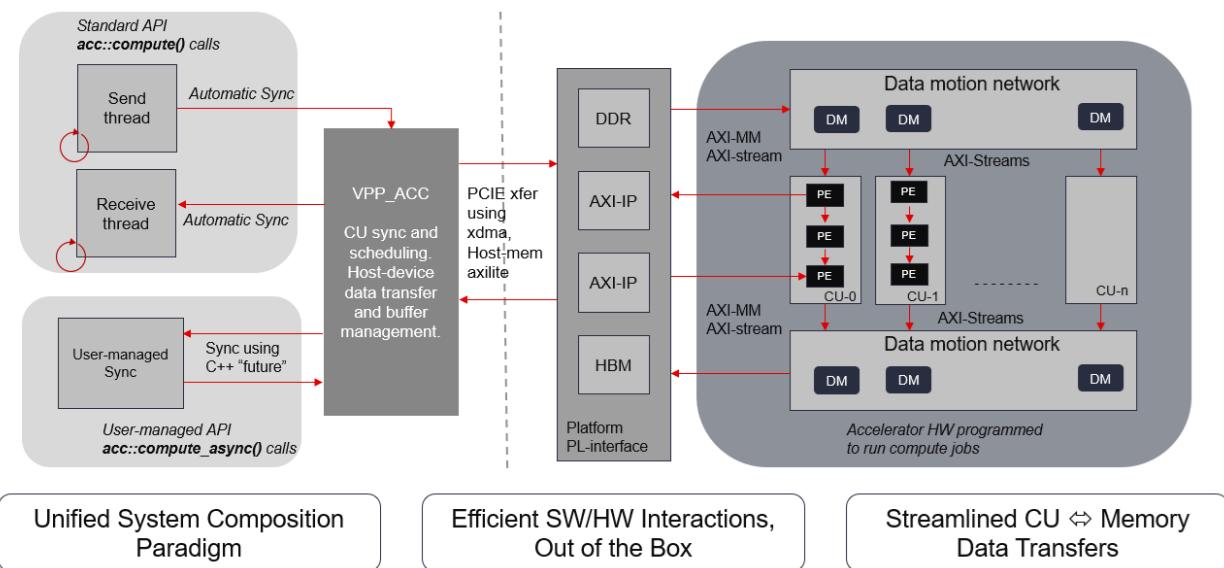
- Ease host side development by simplifying host code that controls the accelerated hardware
- Facilitate experimentation with compute unit replication to leverage data parallelism in the application
- Enable task-chaining composition on software and hardware side with multiple accelerated and software functions
- Serve the purpose of abstracting low-level hardware APIs, making it look like software programming
- Easily model concurrent or multi-threaded software code, and making it seamlessly integrate with end-user application code

## Terminology

- **PL or Programmable Logic:** The part of the FPGA that is made of fundamental programmable building blocks such as flip-flops registers, look-up-tables (LUTs), digital signal processing units (DSPs), RAM-based memory block or clocking circuitry. The PL region can contain one or more accelerators as defined below.
- **Accelerator (ACC):** Designates everything that is custom-generated by the VSC mode and sits inside the programmable logic of the FPGA or ACAP device. The accelerator contains one or more replicated compute units (CUs) in the hardware. In some situations, the term accelerator might also be loosely used to describe the whole Programmable Logic design also including the platform and/or .xo kernels generated by the Vitis compile flow. The VSC accelerator is specified in a user-defined C++ class derived from a predefined `VPP_ACC` class. The interface specification contains connections into Vitis platform ports to access the peripheral resources such as global memories or ethernet ports. The accelerator also contains data movers, which are IP designed to efficiently move data from global memory to the compute units and back.
- **Compute Unit (CU):** Designates the composition of one or more processing elements (PEs) as defined below, that connect to global memory and streams to move data to other PEs. The interface is described using the `compute()` method in the accelerator class. This named API acts as the software entry-point function to the accelerator and therefore specifies the hardware-software arguments. The CU makes the system composition self-describing from the source code and not via the use of text config files.
- **Processing Element (PE):** Designates the core building block of a compute functionality that performs specific actions on data. This is a function-call within the scope of the `compute()` method in the accelerator class. The processing element's functionality can be written in C++ and each PE is compiled by individually by Vitis HLS.
- **Compose/Composition:** Refers to a method of assembling PEs to form a structural network for a CU. The body of the `compute()` method semantically refers to a composition of PEs in hardware, which is unlike the procedural semantics of a C-function body. VSC will enable validation of such a specification. The PEs collectively compose a CU, and one or more replicated CUs exist in an accelerator.

## Hardware and Software Organization

A good system design model makes it very easy to use hardware acceleration for specific functions in an existing application with minimal changes to instantiate compute hardware and run it efficiently. In the Vitis HLS based acceleration flow, the efficiency of the compute hardware will still depend on modeling/coding style and pragmas. In the case of RTL flow, it depends on the chosen architecture. The invocation of accelerated function or CU and interaction with the host should be automated as much as possible, this includes pipelining data through the hardware, using and composing multiple CU's etc.



VSC provides a way to compile your accelerator design and the application software interface from a unified C++ model. The right side of the figure shows the hardware design is a system using the AXI4 framework which is plugged into the dynamic region of a standard Vitis platform. This user-defined composition might consist of replicable compute units (CU), where each CU can be a data-pipelined network of processing elements (PE), and each PE can work on the data:

- in the device memory, typically a DDR, local to the accelerator card having the FPGA
- in a smartSSD connected to the FPGA over PCIe
- arriving to the PE input through one or more AXI4-Stream.

The CUs must connect to platform ports which are typically memory-mapped AXI4 (M\_AXI) for data transfers to/from a host CPU through a DDR, or AXI4-Lite for low bandwidth scalar word transfers. The CUs may operate on independent data sets to achieve macro parallelism inherent to the application, achieving compelling acceleration. VSC provides the ability to use a data-mover (DM) for each M\_AXI. The DM is an RTL IP that efficiently implements DDR transfers by automating well-defined protocols such as AXI-bursting. The CUs may also transfer data to another user-defined accelerator's CUs through the device memory.

VSC provides an application layer interface as shown on the left-side of the above figure. This is a C++ API interface consisting primarily of two threads for each hardware accelerator, or a cluster of CUs. The send-thread controls forwarding data and launching jobs on the accelerator, while the receive-thread allows gathering results from the accelerator. The send-thread uses a named C-function called `compute()` which acts as the software interface to launch the corresponding call-job on the accelerator. The run time layer will automate the several details in scheduling such

jobs onto CU group and managing efficient data transfers of the `compute()` arguments. These independent threads allow the software to asynchronously interact with the hardware execution, thereby efficiently modeling the application-specific computation and data transfers. The VSC software interface also provides several controls for user-driven synchronization with the hardware.

VSC provides a unified system composition paradigm in C++, provides a runtime layer that allows a hardware composition with streamlined data transfer between CUs and device memory, and efficiently implements hardware-software interactions out of the box.

Because the compilation of hardware is a very time-consuming process, it is important that changes to the hardware code should not trigger recompilation of the hardware. This is avoided by using a specific coding style from the user, and VSC will allow the creation of reusable user-space libraries. Those libraries also act as software stack (of C++ APIs) on top of the hardware accelerator system specified by the user. Such a library may even be used as a dynamic run time shared library to be integrated with a third-party software application.

## Design Capture and Modeling

The previous section described the organization, software interfaces, and hardware compiled by VSC from a unified C++ model. The following is a quick summary of some key features of the source C++ model that you can use to easily capture specific design intent:

1. Explicitly model data transfer from host to device and back using two C-threads, to send/receive data to/from each accelerator and its group of CUs
2. Enable performance exploration through guidance parameters, changes such as the use of multiple memory banks, change number of CUs, and concurrent (ping-pong style) data transfers between host and device with minimal or no changes to host or kernel code
3. Replication of CUs to be run in parallel to exploit coarse grain parallelism using only a single numeric guidance parameter
4. Automatic job scheduling on compute cluster (multiple CUs) using round-robin or free-polling
5. Automatic data transfer and accelerator job scheduling on replicated CUs within an accelerator
6. Automatic pipelining of each accelerator job and other optimizations over PCIe®:
  - a. to send/receive data using multiple buffer objects on the host side
  - b. data mover plugin for every CU, based on the data movement pattern guidance
    - i. to copy data between global (DDR/HBM) and on-chip (RAM) memory resources
    - ii. to pre-fetch the next transaction from global memory before the current one finishes on a CU
  - c. Clustered transactions (a sequence of n data sets transferred as one data block) to be processed in one shot, for amortizing PCIe latency

- d. Improve throughput by automatic concurrent (ping-pong style) data transfers using multiple host and device-memory buffers for each CU
  - e. Allow variable payload size using peak memory allocation (allocate for largest data size).
  - f. Dynamic output buffer sizes (allocated at runtime) can be supported, when:
    - i. max buffer size is known at compile-time, and
    - ii. dynamic size is determined by the application code
7. System-level composition using a mix of software (host side) and hardware (compute units):
- a. Hardware-only composition with direct connection (AXI4-Stream) inference. You can create a PE pipeline or a network within each CU, and easily replicate such units.
  - b. Allow free-running PEs with streaming interfaces within a synchronous pipeline (in a CU)
  - c. Mixed hardware and software composition for creating a data processing pipeline. Software tasks can be processing data in-between hardware tasks, with different accelerators compiled into the same xclbin
8. The entire system design is captured in C++ which can be validated in by software compilation of the C++ source and execution

---

## Quick Start Example

The convolutional filter example is based on a video filtering use case. An image filter is applied to all the channels of a video frame. The host generates a stream of images for each color channels which are transferred to the device for processing. The RGB video has three parallel channels providing coarse-grain parallelism which allows to process three color channels independently using a separate compute unit per channel. The problem is to design an FPGA based video filter where each video channel can be processed in parallel by a separate CU. The convolutional filter applied to each channel is the same so it is possible that you can build a single compute unit and use three instances of the CU for acceleration. The following sections describe in detail how the host side and CU are modeled and implemented using the Vitis System Compilation (VSC) mode.

## Modeling the Convolution Filter

The focus of this section is high-level discussion of modeling the convolutional filter using VSC, and not on the lower level details of the implementation itself. At the top level, the CU is modeled as having a memory interface using three 8-bit pointers (i.e. `char *`) to the color values of the input data and output data, as well as filter coefficients. Some other constant parameters are also passed to the CU like the bias, image height and image width.

The header file `conv_acc_filter.hpp` declares one `compute()` wrapper function used for the host code and software interaction and one or more processing elements (PEs). In this example there is only one PE named `krnl_conv`.

In VSC all accelerators should be written as a class derived from the `VPP_ACC` base class. In the user code, every class that inherits from the `VPP_ACC` class should intend to be an accelerator that compiles to hardware. Specifically the child class should provide a function named `compute()` that is the software entry point to the compiled hardware accelerator, as shown in the example below. Every derived class should have a unique name to represent a unique compute unit function.

```
#pragma once
#include "common.hpp"
#include "vpp_acc.hpp"

class conv_acc : public VPP_ACC<conv_acc, /*NCU*/3>
{
    ACCESS_PATTERN(src, SEQUENTIAL);
    ACCESS_PATTERN(coeffs, SEQUENTIAL);
    ACCESS_PATTERN(dst, SEQUENTIAL);
    // Data copy macros : specifies that size of data to be copied for
    // kernel call in
    // case the kernel can process variable size data.
    DATA_COPY(src, src[width*height]);
    DATA_COPY(coeffs, coeffs[FILTER_V_SIZE*FILTER_H_SIZE]);
    DATA_COPY(dst, dst[width*height]);

    // Kernel DDR connections
    SYS_PORT(coeffs, DDR[0]);
    SYS_PORT(src, DDR[0]);
    SYS_PORT(dst, DDR[0]);

public:
    static void compute(
        char             *coeffs,
        float            factor,
        short            bias,
        unsigned short   width,
        unsigned short   height,
        unsigned char    *src,
        unsigned char    *dst
    );
    static void krnl_conv(
        char             *coeffs,
        float            factor,
        short            bias,
        unsigned short   width,
        unsigned short   height,
        unsigned char    *src,
        unsigned char    *dst
    );
};
```

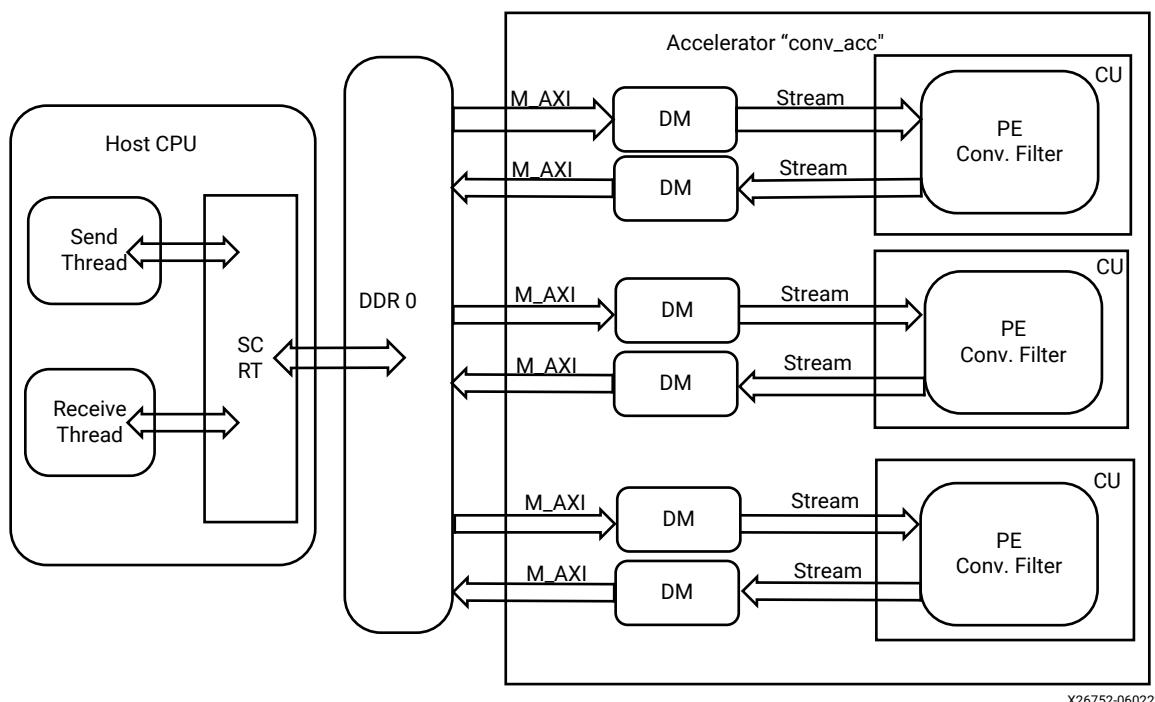
This header file defines a number of different things including the top-level interface of the CU. It declares two static functions `compute()` and `krnl_conv()`. The `compute()` function acts as software entry point on the host side, which is called from the `send_while` thread sending data to the CU (here the `conv_acc` class). The `krnl_conv()` function implements all the functionality of the CU.

The header file also defines some other important items that any kernel model for VSC is required to specify:

## Hardware System Architecture

The system architecture and accelerator architecture can be fully specified using an accelerator class derived on the `VPP_ACC` class. In the example code, the convolution filter system architecture, compute composition, number of CUs, and compute connectivity is fully defined by the `conv_acc` class. The following system architecture is generated.

Figure 138: Hardware Architecture



The `conv_acc` class specifies that the `krnl_conv()` function be accelerated by wrapping it inside the `compute()` function. Because a video has three different color channels which can be processed independently using the same functionality, the accelerator class also specifies three CUs to be used. These CUs are replicated and plugged into the memory system using data movers as specified by the data `ACCESS_PATTERN` macro. On the host side the figure shows two separate threads for the receive and send threads which interact with the hardware using lower level runtime drivers from VSC.

The CU functionality must be defined via the `compute()` function in a separate `.cpp` file. In this example there is only one processing element (PE), so `compute()` simply wraps it. The following code snippet shows a part of the `.cpp` file with the sub-functions details not provided here. Refer to [Supported Platforms and Startup Examples](#) for more complete examples.

```
// the compute() function wraps the processing function in this example
void conv_acc::compute(
    char          *coeffs,
    float         factor,
    short         bias,
    unsigned short width,
    unsigned short height,
    unsigned char  *src,
    unsigned char  *dst)
{
    krnl_conv(coeffs, factor, bias, width, height, src, dst);
}

// ... the processing function implements the CU functionality
void conv_acc::krnl_conv(
    char          *coeffs,
    float         factor,
    short         bias,
    unsigned short width,
    unsigned short height,
    unsigned char  *src,
    unsigned char  *dst) {
#pragma HLS DATAFLOW

    hls::stream<window,3> window_stream; // Set a stream depth to 3

    // Read incoming pixels and form valid HxV windows
    Window2D(width, height, src, window_stream);

    // Process incoming stream of pixels, and stream pixels out
    Filter2D(width, height, factor, bias, coeffs, window_stream, dst);
}
```

## Host Side Data Generation

The convolutional filter consists of three independent CUs which will each process one color channel from the video image stream. The example does not use an actual video stream to keep project simple and stay focused on VSC. The example host code simply generates a random image with three color channels which is passed to a software function which is using the VSC specific code. This results in C-threads for sending and receiving data from host to device plus the `compute()` call.

The function passes all the filter parameters and data pointers to source and destination images. The code snippet below gives the definition of this function. The essential steps include:

- Create buffer pool handles (`srcBufPool...`) to enable sending and receiving data between the host and the device. Attributes indicate if the buffers are inputs or outputs.

- Call `conv_acc::send_while()` using a lambda function. The lambda function allocates buffers on the device, copies host data to the device buffers, then calls the `compute()` function (which ultimately runs the hardware accelerated function). `send_while()` keeps calling the lambda function as long as it return `true`.
- Call `conv_acc::receive_all_in_order()` also using a lambda function to receive the processed buffers.
- Use a `join()` call to wait and synchronize everything.



**TIP:** Refer to [VPP\\_ACC Class API](#) for an explanation of the various functions.

```
#include "conv_filter_acc_wrapper.hpp"

int conv_filter_execute_fpga(
    const char          coeffs[FILTER_V_SIZE][FILTER_H_SIZE],
    float               factor,
    short               bias,
    unsigned short      width,
    unsigned short      height,
    unsigned int         numImages,
    YUVImage           srcImage,
    YUVImage           dstImage
)
{
    auto srcBufPool = conv_acc::create_bufpool(vpp::input);
    auto dstBufPool = conv_acc::create_bufpool(vpp::output);
    auto coeffsBufPool = conv_acc::create_bufpool(vpp::input);

    int run = 0;
    int dataSizePerChannel = width * height ;
    // sending input
    conv_acc::send_while([]()>bool {
        conv_acc::set_handle(run);
        unsigned char * srcBuf = (unsigned char
*)conv_acc::alloc_buf(srcBufPool, 3*dataSizePerChannel);
        unsigned char * dstBuf = (unsigned char
*)conv_acc::alloc_buf(dstBufPool, 3*dataSizePerChannel);
        char * coeffsBuf = (char
*)conv_acc::alloc_buf(coeffsBufPool, FILTER_V_SIZE*FILTER_H_SIZE);

        // initialize all input data before parallel computes
        unsigned char * srcChannel[3] = {srcImage.yChannel,
srcImage.uChannel, srcImage.vChannel};
        for (int ch = 0; ch < 3; ch++){
            std::memcpy(srcBuf+ch*dataSizePerChannel, srcChannel[ch],
dataSizePerChannel);
        }
        std::memcpy(coeffsBuf, coeffs, 256);
        // execute conv_acc<NCU> parallel computes
        for (int ch = 0; ch < 3; ch++){
            conv_acc::compute(coeffsBuf,
                            factor,
                            bias,
                            width,
                            height,
                            srcBuf + ch*dataSizePerChannel,
                            dstBuf + ch*dataSizePerChannel);
        }
    return (++run < numImages);
}
```

```
};

// receive lambda function for receive thread
conv_acc::receive_all_in_order([-]() {
    int run = conv_acc::get_handle();
    unsigned char * dstBuf = (unsigned char
*)conv_acc::get_buf(dstBufPool);
    unsigned char * dstChannel[3] = {dstImage.yChannel,
dstImage.uChannel, dstImage.vChannel};
    for (int ch = 0; ch < 3; ch++){
        std::memcpy(dstChannel[ch], dstBuf+ch*dataSizePerChannel,
dataSizePerChannel);
    }
});
// wait for both loops to finish
conv_acc::join();

return 0;
}
```

## Creating a VSC Makefile

### Environment Setup

Setup your environment by sourcing both the Vitis setup script and the XRT setup script as shown below:

```
$ source <Vitis_install_path>/Vitis/<version>/settings64.sh
$ source <XRT_install_path>/xrt/setup.sh
```

You should also set up the `PLATFORM_PATH` environment variable to point to the platforms that you have downloaded for your accelerator cards. `PLATFORM_PATH` is used by the Makefile template described below. If you do not set it up, you will need to provide it as a command-line argument to the `make` command.

```
$ export PLATFORM_PATH=<path_to>/platforms
```



**TIP:** The `PLATFORM_PATH` environment variable is the same as the `PLATFORM_REPO_PATHS`, and points to directories containing platform files (`.xpfm`).

## Using the Template Makefile

A template Makefile is provided in the Vitis installation folder that can be included by your Makefile to greatly ease the development of custom Makefiles. The following describes variables required to use the template Makefile by your own Makefile:

```
# Accelerator source files (ie. Vitis HLS code, or vpp_acc code)
# must be specified here; they will be processed by v++ --compile
ACC_SRCS := src/conv_filter_acc.cpp

# Host source files are defined here:
HOST_SRCS := src/host.cpp src/function_cpu.cpp

# Include tool provided Makefile template:
include ${XILINX_VITIS}/system_compiler/examples/vpp_sc.mk
```

The build targets of the template Makefile include: `all`, `run`, `build`, `clean`, and `ultraclean`. A valid `make` command to build the design for hardware emulation (`hw_emu`) could be:

```
$ make DEVICE=<platform> TARGET=hw_emu build
```

A `make` command to run the target as well could be:

```
$ make DEVICE=<platform> TARGET=hw_emu run
```



**IMPORTANT!** VSC uses `g++ -dumpversion` to get the version number of `g++`, and stops the build if the version is older than 7.1.0. However, on some OSes this command returns only the major version, for example version 7.5.0 is seen as version 7, and returns the following error:

```
vpp_sc.mk: Using /usr/bin/g++
vpp_sc.mk: g++ (Ubuntu 7.5.0-3ubuntu1~18.04) 7.5.0 <Installation path
of Vitis 2022.2>/system_compiler/examples/vpp_sc.mk:29:
*** gcc version lower then 7.1.0 is not supported. Stop.
```

To resolve this error use the `make` command with `GCC_VERSION=75000`. In this case the version is returned as 75000 in the template Makefile, and the build will continue.

## Writing a Custom Makefile

Your custom Makefile should provide similar functions as the template Makefile. The basic steps to compile in VSC are similar to other Vitis tool flows, as shown below:

```
v++ --compile my_acc.cpp -o my_acc.o
v++ --link my_acc.o -o hw.xclbin # also outputs hw.o
g++ -c main.cpp -o main.o
g++ -o host.exe main.o my_acc.o hw.o -lvpp_acc -lxrt_core -lpthread
```

Some things to notice in these commands:

1. The `v++` compile command does not use the `-k` option to specify a kernel name. This is because the source code defines a class derived from the `VPP_ACC` class that enables the Vitis tools to compile the VSC system.
2. The `v++ --compile` compile command produces `.o` files instead of `.xo` files. These files are used by Vitis compiler and `g++` to build the hardware and host systems.
3. The `v++ --link` command will use the `.o` files as inputs and generate both an `.xclbin` for the hardware accelerator, and a `hw.o` which contains compiled objects for the VSC hardware-software interfaces required by the host application and used by `g++`.
4. The commands will also need to have specific options for the respective tools and modes, for example `--platform` and `--target`.

Alternatively, you can create a shared (dynamically loadable) library to let the accelerator be easily integrated into a third-party application. For example, `libmy_acc.so` as given below:

```
v++ -c my_acc.cpp -o my_acc.o
v++ -l my_acc.o kernel.xo -o hw.xclbin # also produces hw.o
g++ -c -fPIC app.cpp -o app.o
## Command to create the shared library:
g++ -shared -fPIC -o libmy_acc.so app.o my_acc.o \
    -Wl,--whole-archive ${XILINX_VITIS}/system_compiler/lib/x86/
libvpp_acc.a \
    -Wl,--no-whole-archive
## The link command using the shared library
g++ -o host.exe main.o -lmy_acc -lxrt_hw -lpthread
```

## OS and Tool-Chain Compatibility

The VSC tools should work with any Linux distribution which is compatible with RHEL/CentOS 8 toolchain (including `gcc-8.3` and `binutils-2.30`).

RHEL/CentOS 7 is also supported, with the following requirements for building and running:

1. Use of `devtoolset-8`. For example:
  - It can be installed with `sudo yum install devtoolset-8`
  - It can be used in a shell as `scl enable devtoolset-8 bash` for Vitis compilation
2. The host code needs to be compiled with this flag: `-D_GLIBCXX_USE_CXX11_ABI=0` explicitly when compiling with `g++`.



**TIP:** When using the VSC template Makefile this is automatically handled, as described above.

3. Final linking needs to use  `${XILINX_VITIS}/system_compiler/lib/centos7/ libvpp_acc.a`.



**IMPORTANT!** Because of the need to disable the use of `CXX11 ABI` while building on RHEL/CentOS 7, the executable cannot be run on a RHEL/CentOS 8 platform, and vice versa.

# Building Hardware

This chapter will describe the key aspects of building the accelerator hardware using VSC. The first section describes the user-defined C++ accelerator class specification, and the following sections describe creating the hardware interface with user-guidance macros, the data transfer types, and the different styles of system composition for the accelerator.

## Accelerator HW Interface

### *User-Defined Accelerator Class*

Vitis enables VSC for hardware and host-code compilation when the source code contains a class derived from the `VPP_ACC` class. The class derived from `VPP_ACC` will be compiled as an accelerator.

A VSC accelerator interface is defined in the single-source C++ model as a class definition that is required in a C++ header file. Multiple accelerator class definitions can be found in one compilation unit (as a single `v++ --compile` command), all of which will be implemented in the same PL hardware. You might also provide multiple accelerators across multiple compilation units. However each compilation unit must define the complete kernel (or HLS) code for that unit.

Every user-defined accelerator class:

- Must be derived from the Vitis pre-defined `VPP_ACC` class
- Must have a static method named `compute()`
- The arguments and functionality of `compute()` can be user-defined
- The class template has two arguments:
  - The first argument must be the same as the user-defined class name
  - The second argument is the number of compute units to be replicated in the hardware

An example accelerator class definition (`xmmult`) is provided below:

```
#include "vpp_acc.hpp"
class xmmult : VPP_ACC<xmmult, /*NCU=*/4>
{
public:
    // Platform port connections
    SYS_PORT(A, DDR[0]);
    SYS_PORT(B, DDR[1]);
    SYS_PORT(C, DDR[2]);
    SYS_PORT_PFM(u50, A, (HBM[0]:HBM[4]:HBM[8]:HBM[12]));
    SYS_PORT_PFM(u50, B, (HBM[1]:HBM[5]:HBM[9]:HBM[13]));
    SYS_PORT_PFM(u50, C, (HBM[2]:HBM[6]:HBM[10]:HBM[14]));
    // Data interfaces
```

```
ACCESS_PATTERN(A, SEQUENTIAL);
ACCESS_PATTERN(B, RANDOM);
DATA_COPY(A, A[SZ]); // move to local memory
DATA_COPY(B, B[SZ]);
ZERO_COPY(C); // direct AXI-mm

// define the SW entry point of the accelerator
static void compute(data_t* A, data_t* B, data_t* C);
// define the HW top-level of the accelerator (HLS top)
static void mmult(data_t* A, data_t* B, data_t* C);
};
```

This class interface model captures all hardware system-related considerations in a single unified source and allows easy integration with the application layer through the `compute()` function as described in [The compute\(\) API](#). The `compute()` function is the host application's entry point to the hardware accelerator.

The class definition also contains guidance macros that refer to `compute()` arguments. By providing these, you will guide VSC to make specific hardware choices during implementation. Refer to [Guidance Macros](#) for more information.

The example shown above describes the accelerator class named `xmmult`, which will be compiled by VSC to have four CUs in hardware (`/*NCU= */ 4`). The accelerator has a PE (or kernel) code defined in the `mmult()` function, which is called within `compute()` that takes three arguments A, B, and C. For each of these arguments, two types of guidance macros are specified: memory port connections, and data access.

1. Platform port connections using `SYS_PORT()` and `SYS_PORT_PFM()` macros.
  - a. These are typically global memory I/O connections or other AXI4 interface connections available in the hardware platform used during Vitis compilation. In this example, the first three `SYS_PORT()` macros connect the three A, B, and C arguments of `compute()` to different DDR banks (0, 1 and 2).



**TIP:** The `SYS_PORT()` guidance macro connectivity for each argument applies to all the CU instances (NCU=4) in the `xmmult` accelerator, because it only specifies one memory bank.

1. Platform port connections using `SYS_PORT()` and `SYS_PORT_PFM()` macros.
  - i. It only applies when the target platform name contains u50, as in the U50 Alveo card.
  - ii. For each of the four CUs, each argument connects to a different HBM banks. This is because the syntax used for the `SYS_PORT_PFM()` for each argument specifies connectivity four times:

```
SYS_PORT_PFM(u50, A, (HBM[0]:HBM[4]:HBM[8]:HBM[12]));
```

2. Data access is specified using `ACCESS_PATTERN()`, `DATA_COPY()`, and `ZERO_COPY()` macros:

- a. The ACCESS\_PATTERN() macros directs VSC to infer a sequential or random access for data transfer. In the example, argument A is defined as sequentially accessed and therefore can be implemented with a stream connection. Argument B is accessed randomly and therefore requires a local (on-chip) memory buffer to support randomized data access from the PE mmult().
- b. For arguments A and B, the DATA\_COPY() macros direct VSC to infer a local memory (RAM) next to the accelerator.
- c. The ZERO\_COPY() macro directs VSC to create a memory-mapped AXI (M\_AXI) connection for argument C.

## Guidance Macros

The guidance macros that VSC supports allow the function arguments (both PE and compute) in the accelerator class to use to various types of hardware interfaces. The following defines the different types of guidance macros:

- **SYS\_CLOCK\_ID(<PE-name>, <clock-ID>);:**
  - <PE-name> specifies a processing element function instantiated in the body of the compute() function.
  - <clock-ID> is an integer value referring to any clock supported by the platform and is available for connection in the user-logic partition. When this macro is specified, the kernel implementation will use the corresponding clock net for the kernel. The available clock IDs for a platform can be found using the `platforminfo` command.



**TIP:** When two PEs are connected through AXI-streams, clock-ID may be different if those PEs are marked free-running. In this case, VSC will automatically insert clock-domain crossing (CDC) connection for data transfer between the PEs. When a free-running PE is connected to a PE that it not, they both need to have the same clock-ID.

- **SYS\_PORT(<port>, <global\_memory>);:**

Specify the platform interface to use for a given argument of the `compute()` function. The global memory is typically a memory bank that will be used to data transfer to/from the FPGA.

- <port> refers to the name of a specific `compute()` argument.
- <global\_memory> can be specified in one of the following forms.
  - <bank-ID>: A single bank ID that applies to all CU instances. For example DDR, DDR[1], or HBM[5]. The bank names for a platform can be found by using the `platforminfo` command.



**IMPORTANT!** Observe the following limitations:

- Host memory (HOST[0]) is only supported for the X3 hybrid platforms (xilinx\_x3522p\*)
- Specifying memory bank ranges (for example HBM[0:3]) is not supported

- Specifying PLRAM is not supported

- (`<CU1-bank-ID>:<CU2-bank-ID>:...:<NCU-bank-ID>`): Within parenthesis, a list of bank-IDs for each CU separated by colons. The bank-IDs are specified for each CU in numerical order, but does not include the CU name:  
`(HBM[0] : HBM[4] : HBM[8] : HBM[12])`. The number of entries must match the specified number of CUs in the class (`/*NCU= */4`).

- **SYS\_PORT\_PFM(<substr>, <port>, <global\_memory>);:**

This can be used to configure accelerator port connections for specific platforms, but defined through a single class header. For example in the code given below, port-A will be connected to HBM[0] for a u50 platform, and to DDR[0] for all other platforms.

```
SYS_PORT(A, DDR[0]);  
SYS_PORT_PFM(u50, A, HBM[0]);
```

- The `<substr>` refers to a sub-string of the platform name. For example, using `u50` would employ the `SYS_PORT_PFM` connections only when the platform name contains the specified string.
- The `<port>` and `<global_memory>` arguments work as described above for `SYS_PORT` macros.



**IMPORTANT!** When multiple `SYS_PORT` and `SYS_PORT_PFM` macros are provided for the same `<port>`, VSC will apply the last suitable `SYS_PORT` or `SYS_PORT_PFM` guidance macro that is read.

- **ACCESS\_PATTERN(<port>, <pattern>);:**

Enables VSC to infer a data mover between the hardware accelerator interface and the global memory in the device.

- `<port>` refers to the name of a specific `compute()` argument.
- `<pattern>` defines one of two different memory access patterns:
  - **SEQUENTIAL:** data transfers occur through AXI4-Stream connections to the acceleration interface. The CU (or kernel) code must strictly follow a sequential access pattern on the corresponding argument, otherwise it will lead to incorrect hardware behavior. For example, pointer indices should be sequentially incremented as with the coding style `pointer[i++]` or `*pointer++`.
  - **RANDOM:** data transfers into an on-chip memory acting as a cache to the accelerator. Therefore the CU code does not need to follow a sequential access pattern.



**IMPORTANT!** On-chip memory resources are limited (for example, typically 32 Kbits per BRAM which could be accessed as 1024 words of 32 bits). If a large payload size per compute job uses too many on-chip RAMs, it can lead to timing closure issues in the Vivado tools. It might be better to use a `ZERO_COPY` guidance macro connecting the accelerator directly to global memory as described below.

- **DATA\_COPY(<port>, <port>[<Num>]);:**

Infers a data-mover IP between the global memory and the accelerator interface. At the runtime of each `compute()` call this data-mover IP will copy the data for the specific `compute()` argument from (or to) a source memory specified by `SYS_PORT` or `SYS_PORT_PFM` guidance macro, or from (or to) local on-chip memory.

- `<port>` refers to the name of a specific `compute()` argument.
- `<port>[<Num>]` specifies the number of array elements that the (array or pointer) argument refers to. `Num` can be an expression of C-constants and/or scalar arguments of `compute()`. This allows the accelerator to have a dynamic payload size determined at run time, and enables automatic bursting on AXI4 connections, data width conversion, and padding for the user-defined argument data-types.



**IMPORTANT!** When using `DATA_COPY` along with `RANDOM` access pattern, the corresponding argument in the prototype of the `compute` API must be declared as an array with fixed size. For example: `compute(int A[10], ...)`.

- **ZERO\_COPY(<port>);:**

Directs VSC to not infer a data-mover IP. Instead, let the accelerator use a AXI4 interface directly connected to the specified global memory for the specified argument of the `compute()` function.

- `<port>` refers to the name of a specific `compute()` argument.

- **ASSIGN\_SLR(<PE>, <SLR-IDS>);:**

VSC will request Vivado to place the related logic of the named PE into the specified SLR(s). However, this is only a request and the final determination made during placement.

- `<PE>`: Specifies the name of the processing element.



**TIP:** If the `compute()` function is specified, the SLR assignment is applied to all PEs inside the `compute()` function.

- `<SLR-IDS>`: Specifies the SLRs to use to place the PE. This can be specified in one of the following forms.
  - `<SLR-ID>`: Applies the specified SLR-ID to all CU instances of this PE.
  - `(<CU1-SLR-ID> : . . . : <NCU-SLR-ID>)`: Within parenthesis, a list of SLR-IDs separated by colons. These SLR-IDs are assigned to the CU instances. The number of entries must match the specified number of CUs in the class (`/*NCU= */4`).

- **FREE\_RUNNING(<PE>);:**

Enables the named PE function to be marked as free-running or an always executing kernel in hardware. Refer to [Accelerator System Composition](#) for more information.

## The `compute()` API

The `compute()` method is a special user-defined method in the user-defined accelerator class definition which is used to represent the compute unit. The arguments of `compute()` provide the software interface of the CU to the host-side application, and the body of `compute()` specifies the hardware composition of the CU. The accelerator class must have one or more additional methods defined, each of which may be individually called only within the body of `compute()`. Each such function called is a processing element (PE) in the hardware. The body of `compute()` can be used to create a structural composition of PEs.

The `compute()` body has the following features:

- Is a structural network of processing elements:
  - Using AXI4 standard protocols in the hardware.
  - Using a start/stop synchronization per `compute()` job
  - Can represent a synchronized hardware pipeline
- Only calls to PE functions and local variable declarations, and no other C-language constructs, are allowed in the `compute()` body
- Each PE is a processing element running in parallel in hardware:
  - The code of this function is implemented as an FSM and datapath using Vitis HLS
  - PE functions can include Vitis HLS pragmas (`#pragma HLS`)
- PEs can be connected to global memory or other platform AXI4 ports
- PEs can be connected to each other through AXI4-Stream interfaces
- A PE can be free-running:
  - Unaware of transaction start/stop with no `ap_ctrl` signals
  - Always executing (data-driven) without a reset/start state
  - Only AXI4-Stream interfaces are allowed

An example `compute()` function definition is given below:

```
typedef vpp::stream<float, STREAM_DEPTH> InternalStream;
void pipelined_cu::compute(float* A, float* B, float* E, int M) {
    static InternalStream STR_X("str_X");
    static InternalStream STR_Y("str_Y");
    mmult(A, B, STR_X, M);
    incr_10(STR_X, STR_Y, M);
    incr_20(STR_Y, E, M);
}
```

- This example is a hardware pipeline of three PEs connected by two internal AXI4-Stream interfaces: `STR_X` and `STR_Y`.

- The `mmult` PE reads matrices A and B from global memory using their associated pointers and writes to the stream `STR_X`.
- The `incr_10` PE reads from stream `STR_X` and writes to stream `STR_Y`.
- The `incr_20` PE read from stream `STR_Y` and writes to global memory E.
- The scalar argument M is the dimension of the matrices and is provided to all the PE in the CU.

This model can be used to define various types of accelerator system compositions.

The `compute()` interface is intended to succinctly capture a hardware system while providing a simple software application interface. The following section describes the native C++ data types allowed at the `compute()` interface. The entire accelerator class definition with `compute()` and other PEs, along with an application code can be functionally validated as described in [Debugging and Validation](#).

## Interface Data Types

All the basic C++ data types are supported for `compute()` arguments: `bool`, `char`, `int`, `short`, `long`, `float`, `double`. These C++ type modifiers are supported as well: `signed`, `unsigned`, `short` and `long`.

In addition, arbitrary precision data types are also supported, `ap_int<N>` and `ap_uint<N>`, as described in [Arbitrary Precision \(AP\) Data Types](#) in the Vitis HLS tool. The `ap_int` and `ap_uint` are pre-defined data types in Vitis HLS, where N is a integer with a max limit of 1024. This allows finer bit-width control, especially to handle data packing or to match a global memory data bus width.

The `compute()` interface arguments can be classified as Scalar types or Buffer types of arguments as described below.

### Scalar Type

A scalar argument is a basic data-type that represents a single word passed to the `compute()` call. VSC will infer a scalar argument type when any of the basic data types described earlier are used as a pass-by-value argument of `compute()`. For example `size` and `value` are scalars here: `compute(int size, float value);`



**TIP:** The transfer of a scalar word is done using the AXI4-Lite hardware interface.

A user-defined C-struct can be used as a pass-by-value scalar argument, with the following rules:

- Struct cannot have a field with C++ bit specifiers, for example: `int field_x:4;`
- Struct cannot have a pointer field, for example: `int* ptr_field;`

- The total byte-size of the struct must be a strict power of two. If not, the user is required to declare a dummy field to fill in the remaining bytes (e.g. `char pad[remainder]`).

Scalar arguments are always passed by value, and are typically inputs to the accelerator. A standard C++ reference argument is not allowed. For example:

```
compute(int size, float& result_value) // reference argument is not allowed
```

However, a C++ pointer type can be used in place of a reference when a PE function argument is still a reference. For example:

```
my_acc::compute(int inp, data_t *out_p) { // reference not allowed
    my_PE(inp, *out_p); // deref the ptr for my_PE out_ref
    my_PE_two(inp ... );
}
my_acc::my_PE(int in, data_t &out_ref) { // reference is allowed
...
// application code
my_acc::send_while ... {
    out_p = my_acc::alloc_buf(bp, 1); // required
    my_acc::compute(inp, out_p);
```

VSC will infer a scalar for `out_p`, when this output argument when all these conditions apply:

- It must be pointing to just one element.



**IMPORTANT!** *The application code must allocate memory of size 1 using `alloc_buf()` as described in [VPP\\_ACC Class API](#).*

- The kernel code writes exactly one value to it, for example: `*out_p = result;`
- No VSC guidance macros are specified on this argument.

A scalar argument of `compute()` can be passed to multiple PEs. In hardware a single AXI4-Lite port will drive individual scalar registers of every PE function that takes this scalar argument.

## Buffer Type

A buffer is a pointer or array argument to the `compute()` function, which can hold one or more elements. The contents of the buffer is transferred to/from the device based on the corresponding platform interface (SYS\_PORT connection). The base (element) type of the pointer or array argument can be any basic data types described in the earlier section. In addition, a user-defined C-struct data type can be also be used as a base element type.

**Note:** A buffer type argument requires global memory space both on the device and in the CPU host. Therefore, when using a pointer type the application code is required to call memory allocation (`alloc_buf` described in [VPP\\_ACC Class API](#)).

When using a C-struct as a base type for an pointer or an array, the following rules apply:

- Struct cannot have field with C++ bit specifiers, for example: `int field_x:4;`
- Struct cannot have a pointer field, for example: `int* ptr_field;`



**IMPORTANT!** The C++ data modifiers (like `const`, `register`, `volatile`) are not allowed at the `compute()` interface.

The following example shows coding styles, both allowed and not-allowed, for buffer type arguments:

```
// accelerator interface
my_acc::compute(int A, int *B, int C[10]) {
...
// application code
int S, *BB, *CC;
my_acc::send_while ... { ...
    BB = my_acc::alloc_buf( ... ); // required
    CC = my_acc::alloc_buf( ... ); // required
    compute (S, AA, BB);
```

In the example above, the `compute()` arguments A and B are treated the same way in the application code so that it is required to allocate memory for these buffer arguments.



**TIP:** A buffer argument of `compute()` can only be passed to a single PE function within.

In the example, A is a scalar input argument. The caller just passes an integer argument to the `compute()` call, the value is transferred to the device using AXI4-Lite. Whereas, B and C are buffer arguments. They can be either input, output, bi-directional or remote. The caller has to allocate memory for this buffer using `alloc_buf()`.

## Platform Specific Hardware Config

As mentioned in [Guidance Macros](#), `SYS_PORT` connections can be customized per platform using the `SYS_PORT_PFM` macro. But if a new platform needs to be added, and thus the header file gets modified, then `make` will want to rebuild the hardware for already existing and compiled platforms which is not desirable. A good way to prevent this is to create separate configuration header files for each platform, and use the `-I` CFLAG to include the correct one for a given platform. For example:

```
ifneq (,$(findstring u50,$(DEVICE)))
    EXTRA_CFLAGS := -I include/u50
else
    EXTRA_CFLAGS := -I include/u200
endif
```

The accelerator header file includes a common file (example `config.hpp`), which is customized and provided in separate directories to make it easier to have platform-specific configuration, like number of compute units and port mapping as shown in this example:

Table 69: Additional Makefile Variables

In the ACC class specification	include/u50/config.hpp	include/u200/config.hpp
<pre>#include "config.hpp" // Accelerator class class pipelined_cu : public VPP_ACC&lt;pipelined_cu, NCU&gt; {     ...     SYS_PORT(A, PORT_MAP_A);     SYS_PORT(B, PORT_MAP_B);     SYS_PORT(E, PORT_MAP_E);</pre>	<pre>#define NCU 2 // global memory connections to Accelerator ports #define PORT_MAP_A (HBM[0]:HBM[2]) #define PORT_MAP_B (HBM[0]:HBM[2]) #define PORT_MAP_E (HBM[0]:HBM[2])</pre>	<pre>#define NCU 3 // global memory connections to Accelerator ports #define PORT_MAP_A (DDR[0]:DDR[1]:DDR[2]) #define PORT_MAP_B (DDR[0]:DDR[1]:DDR[2]) #define PORT_MAP_E (DDR[0]:DDR[1]:DDR[2])</pre>

## Accelerator System Composition

As described in [the compute\(\) API](#) section, the body of the `compute()` function represents a structural composition of PEs, which is unlike the procedural C semantics but can be validated in the Vitis tools by running software emulation. The `compute()` scope can only hold two types of C++ statements:

1. Function calls to represent hardware PEs. The structural semantics of the `compute()` body allows various hardware composition styles that are described in the following sections.
2. Static `vpp::stream` object declarations to represent data streams in-between PEs. This is described in more detail in the following section.

### Stream connections using `vpp::stream`

Each argument of a PE function has to be connected to either an argument of the top-level `compute()` function, or to a statically declared stream. A `vpp::stream` inherits from the Vitis HLS `hls::stream` which is described in [HLS Stream Library](#).

The `vpp::stream` object provides a few additional features:

- The `vpp::stream` constructor takes an additional argument, `post_check`, the default value is true. Most stream variables will be expected to be empty at the end of each `compute` call, but others might be designed to carry over data across multiple `compute()` calls. In software emulation, by default `vpp::stream` will be checked to be empty at the end of each `compute()` call, and will assert if it is not. The optional `post_check` argument lets you turn off this assertion.
- The `DEPTH` parameter allows the user to specify a FIFO depth to be implemented in hardware, as well as used during software and hardware emulation.



**TIP:** The default value of `DEPTH` is set to 1024 and that is typically an over-utilization of resources in hardware compilation. You are recommended to specify the depth for streams used in the scope of `compute()` to conserve resources. You can run software emulation with a small depth of 2 to test functionality and help determine the required depth.

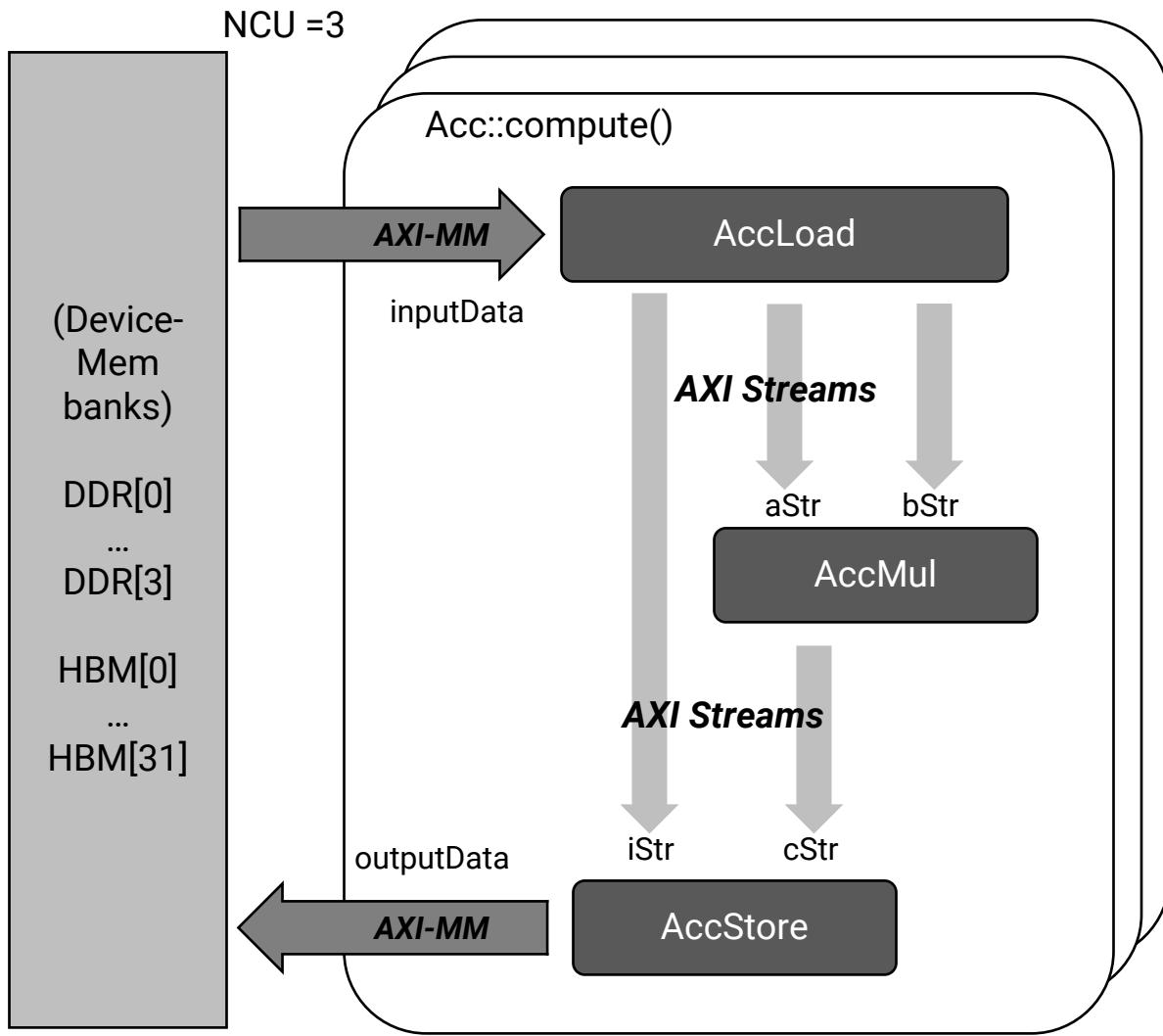
- A `vpp::stream` variable can be passed down to a PE-function argument which can be of type `hls::stream`.
- The `vpp::stream` variable must be declared `static` to ensure proper functional validation in Vitis software emulation. In hardware, a stream/FIFO is not emptied implicitly and its content is persistent across different runs. It behaves in the same way as a `static` variable.

The following sections describe popular styles of composing a pipelined system of PEs using basic C++ coding within the `compute()` body scope. The PEs in the pipeline can use direct AXI4-Stream connections or AXI4 (M\_AXI) global memory access to move data across PEs.

### ***Single-path Synchronous Pipeline***

The following figure shows an example of a single-path pipeline which has three PEs, namely `AccLoad`, `AccMul`, and `AccStore`. The PEs `AccLoad` and `AccStore` access data stored in the global memory through M\_AXI channels. Note that the accelerator class header ties the `inputData` and `outputData` ports to DDR[0]. In this case, the `ZERO_COPY` code was used for these PEs to directly access the global memory.

Figure 139: Single-path Pipeline



X27503-120522

The following is the code example for the figure.

```

typedef vpp::stream<DT> STREAM;
class Acc : public VPP_ACC<Acc, NCU>
{
    ZERO_COPY(inputData);
    ZERO_COPY(outputData);
    SYS_PORT(inputData, DDR[0]);
    SYS_PORT(outputData, DDR[0]);
public:
    static void compute(DT* inputData, DT* outputData);

    static void AccLoad(DT* inputData, STREAM& aStr,
                        STREAM& bStr, STREAM& iStr);
    static void AccMul(STREAM& aStr, STREAM& bStr,
                      STREAM& cStr);
    static void AccStore(STREAM& iStr, STREAM& cStr,
                        DT* outputData);
}

```

```
}

void Acc::compute(DT* inputData, DT* outputData)
{
    static STREAM aStr, bStr, cStr, iStr;

    AccLoad (inputData, aStr, bStr, iStr);
    AccMul (aStr, bStr, cStr);
    AccStore(iStr, cStr, outputData);
}
void Acc::AccMul(STREAM& aStr, STREAM& bStr, STREAM& cStr)
{
    for (int i = 0 ; i < N_WORDS ; i++) {
        int res = aStr.read() * bStr().read();
        cStr.write(res);
    }
}
```

The `compute()` function body represents a hardware pipeline. There are three function calls corresponding to the PEs, and there are four local stream variables declared:

1. `AccLoad` takes `inputData` and writes to three streams
2. `AccMul` processes a fixed number of words in input streams `aStr` and `bStr` and writes the results to `cStr`
3. The `AccStore` function will further process the incoming data in `iStr` and `cStr` to write results on `outputData` connected to DDR[0] port.

The PEs in this system will execute in a synchronous fashion such that data flows through in a pipelined fashion. Every call of `compute()` will load `inputData` and trigger all PEs for a new transaction. Every call to `compute()` requires every PE to complete execution (start and stop) exactly once. This example is a pipeline with 3-stages, or PEs chained in a single-path. Thus, with a simple C++ coding style the user can create a hardware pipeline.

Note that the `VPP_ACC` class allows replication of such pipeline using the `NCU` parameter. If `NCU` is more than 1, then the hardware contains as many replicated pipelines. The calls to `compute()` are automatically loaded in the next available pipeline slot. Thus, the application layer remains simple and easy to maintain, and automates running data through multiple pipelines in the hardware.

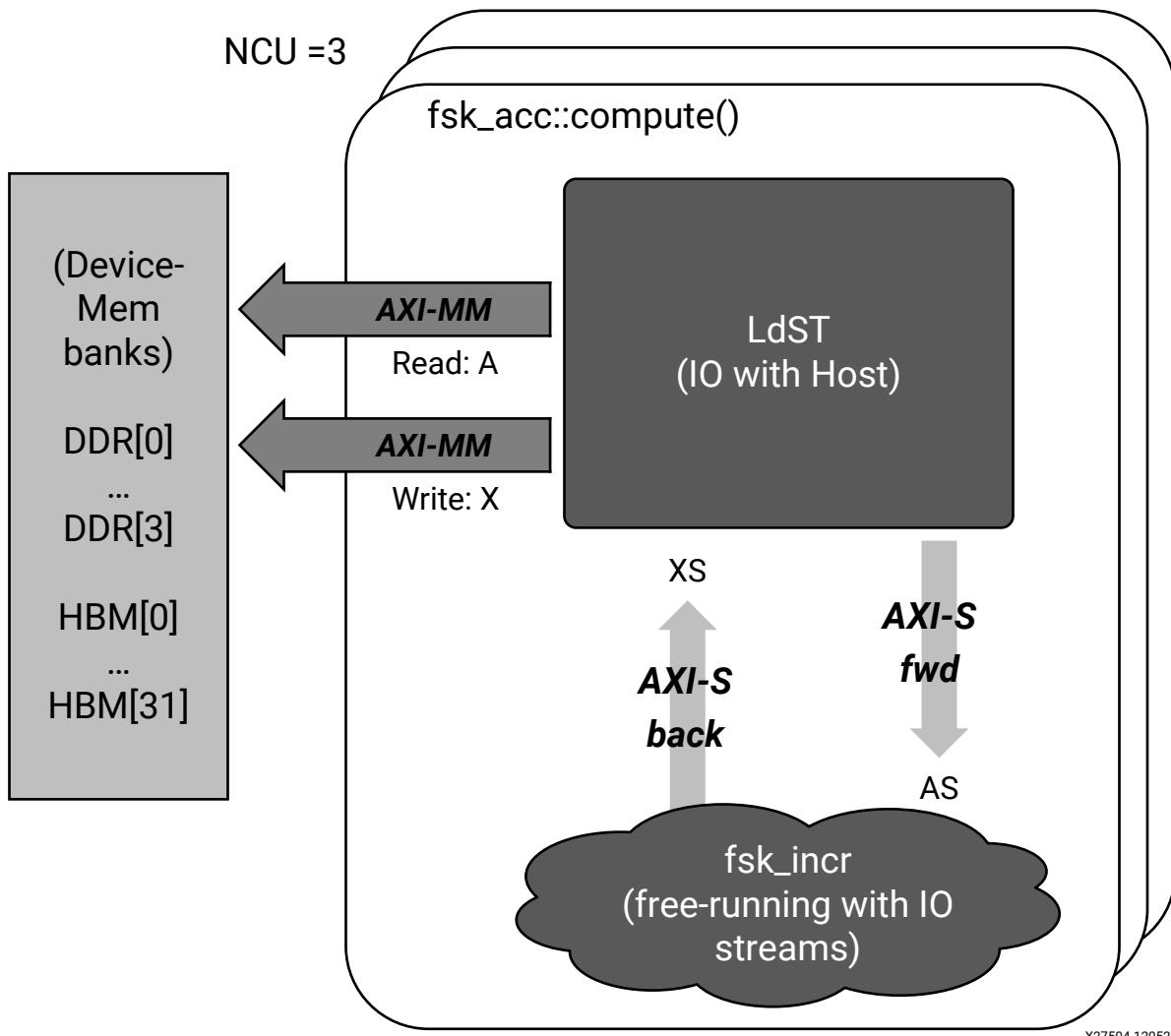
## Free Running PE

The PEs in a pipeline operate synchronously on transactions passing through. For every `compute()` call a PE will start and stop exactly once. However, when a PE is marked as `FREE_RUNNING` as described in [Guidance Macros](#), it has the following hardware semantics:

- The PE does not start, stop, or reset per transaction or `compute()` call. It is an HLS kernel with the `ap_none` control interface as described in [Block-Level Control Protocols](#)
- The interface must have only AXI4-Stream arguments, or scalar inputs
- Operation is data-driven, with the PE acting only on the input stream words, and unaware of the payload size of the transaction

- The PE begins execution immediately after the hardware bitstream is programmed into the device

Figure 140: Free-Running



X27504-120522

The figure above shows a diagram of the free running PE. This accelerator contains two PEs, a LdStr PE that has global memory access, and the fsk\_incr which is a free-running PE. In the `compute()` scope these PEs are connected by two AXI4-Stream interfaces: AS that moves words from LdStr to fsk\_incr, and XS which is the feedback path.

The code for this example is provided below.

```
class fsk_acc : public VPP_ACC<fsk_acc, NCU>
{
    ZERO_COPY(A);
    ZERO_COPY(X);
    SYS_PORT(A, DDR[0]);
    SYS_PORT(X, DDR[0]);
    FREE_RUNNING (fsk_incr);
```

```
public:
    static void compute(DT* A, DT* X, int sz);
    static void loadstore(DT* A, DT* X, hls::stream<DT>& AS,
                          hls::stream<DT>& XS);
    static void fsk_incr(hls::stream<DT>& AS, hls::stream<DT>& XS);
};
Void fsk_acc::compute(DT* A, DT* X, int sz)
{
    static vpp::stream<DT> AS, XS;
    ldst(A, X, sz, AS, XS);
    fsk_incr(AS, XS);
}
void fsk_acc::ldst(DT* A, DT* X, int sz, hls::stream<DT>& AS,
                   hls::stream<DT>& XS)
{
    for (int i = 0; i < sz; i++) {
        AS.write(A[i]);
    }
    for (int i = 0; i < sz; i++) {
        XS.read(X[i]);
    }
}
void fsk_acc::fsk_incr(hls::stream<DT>& AS, hls::stream<DT>& XS)
{
    DT val;
    AS.read(val);
    XS.write(val + 1);
}
```

The `LdSt` PE operates on `sz` words reading and writing to the global memory ports `A` and `X` respectively. Whereas, the free-running PE `fsk_incr` is agnostic to `sz`, and reacts only to the words on the incoming `AS` stream.

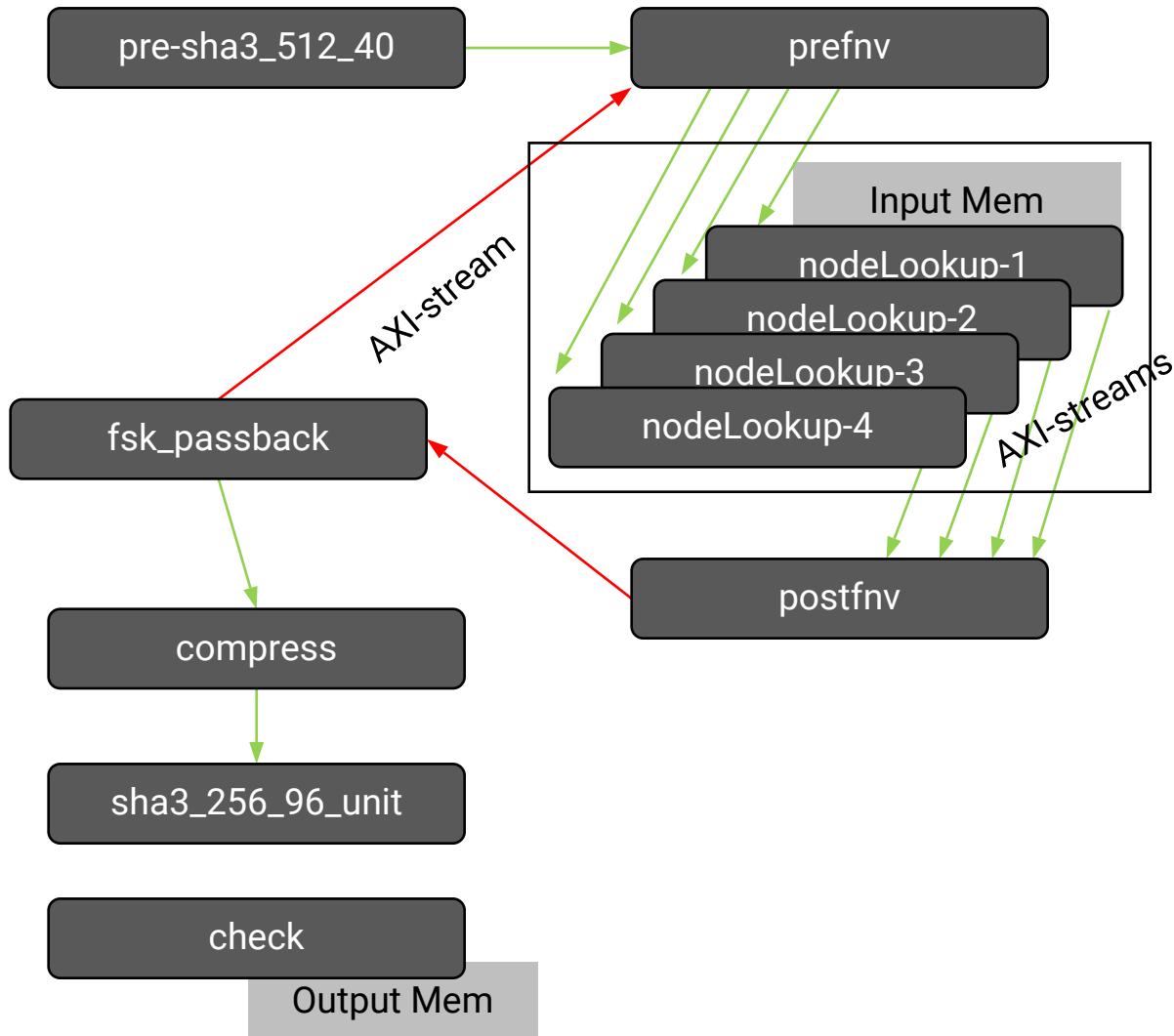
The free-running semantics described earlier greatly simplifies the implementation of a free-running PE, often simplifying the FPGA utilization and routing resources required. It enables the design of a streaming pipeline design where the intermediate PEs can be free-running, thereby operating only on the input AXI4-Stream.

With any pipeline composition, when hardware replication is enabled (NCU is more than 1), the hardware contains as many replicated pipelines, and `compute()` jobs run on available pipeline slot. Thus, the application layer remains simple and automates running data through multiple pipelines in the hardware.

## ***Streaming Network of PEs***

Using stream variables between functions in the `compute()` scope, you can design an arbitrary network of PEs streaming data across the PEs. The body of the `compute()` method semantically describes a structural composition of PEs. It is unlike the procedural semantics in the C-language, and VSC allows software-emulation based validation of the `compute()` body semantics. An example using such a network is a design developed for Etherium hashing - a popular algorithm used in cryptocurrency mining. The VSC code for this design is available in the [Ethash](#) example on GitHub.

Figure 141: Streaming Network



X27505-120522

The picture shows the system architecture of this design. It is a pipelined network of PEs connected by AXI4-Stream. There are four PEs, nodeLookUp-1 to 4 that read from global memory, and each of these also read from an input stream produced by the PE prefnv. The resulting AXI4-Stream from these four PEs are consumed by the PE postfnv.

Notice that there is an AXI4-Stream feedback loop from postfnv in fsk\_passback and back to prefnv. This loop is expected to converge after iterating several times over data flowing through the AXI4-Stream. The entire system of PEs will deterministically start and stop execution for each `compute()` call.

Such streaming architectures are typically efficient in utilizing FPGA resources, and particularly lower in routing resources compared to using AXI4 M\_AXI interfaces. Therefore, such architectures have the potential to achieve higher clock frequency and better accelerator performance.

This VSC model is written entirely in C++ and it captures the network with function calls in the `compute()` scope. The C++ model can be functionally validated in VSC using software emulation, without needing to compile any hardware. This enables early validation of the original design intent in the Vitis tools.



**TIP:** A less efficient way to compose a system is by creating multiple accelerators (different derivative classes from `VPP_ACC`), composing a pipeline in the application layer as described in [Multi-Accelerator Pipeline Composition](#).

## Data Transfer Interface Considerations

In a system design, it is important to correctly specify the mode of data transfer between the accelerator and the host. The following sections provide more details on the design aspects.

### Global Memory I/O

VSC allows two modes of transfers between the global memory and the accelerator interface as described in [Guidance Macros](#):

1. `DATA_COPY` - drop-in a data mover, an efficiently designed RTL IP, that will automate certain features like bursting and data width manipulation on M\_AXI interface to global memory. On the accelerator side, this interface supports both sequential access of data as in the Vitis HLS `ap_fifo` interface, and random access of data as in the Vitis HLS `ap_memory` interface.
2. `ZERO_COPY` - connects the M\_AXI interface from the global memory platform port to the accelerator

The following considerations are recommended for definition of the `compute()` function interface:

1. If the data size (all compute arguments) is too big to fit into the device DDR, split up the large `alloc_buf` into smaller chunks of computation over multiple send iterations.
2. It is generally recommended to use `DATA_COPY` and `SEQUENTIAL` access pattern, especially when the accelerator code does sequential input data access.
3. If the accelerator code does random access of the data, and the code cannot be modified to access the data sequentially, then use the random access pattern if the data fits into device RAM (BRAM or URAM). Otherwise use `ZERO_COPY`.

### Sequential Access Pattern

For a PE port `data` that is accessed sequentially by the kernel code, use  
`ACCESS_PATTERN(data, SEQUENTIAL);`

VSC requires the size of the data at runtime. This must be done through the `DATA_COPY` macro. For example, `DATA_COPY(data, data[numData]);`

Both `data` and `numData` will have to be passed as arguments to the kernel, even if the kernel for example would not use `numData`, it still has to be provided as an argument. In general, `numData` can be any expression as long as it can be evaluated at runtime in terms of the kernel function arguments. The following example is allowed in the accelerator class declaration, though perhaps not very practical:

```
DATA_COPY(data, data[m * log(n) + 5]);  
...  
static void PE_func(int n, int m, float* data);
```

For both input and output data, the exact size of the `DATA_COPY` is important and has to exactly match the amount of data the kernel is reading. If the size does not match the design there can be functional issues like:

- A hang at runtime, if the kernel reads more data than provided by the application code, or
- the kernel will read garbage data, if the previous kernel call did not read all the data from a previous compute call

To prevent and debug this, you can use `ZERO_COPY` to make sure that the kernel code works properly.

## Random Access Pattern

If the kernel has to access a `compute()` function argument, `data`, in a random fashion, use `ACCESS_PATTERN(data, RANDOM)`:



---

**TIP:** The random access pattern will require a local FIFO buffer which has size limitations imposed by BRAMs, which are typically in 32 Kb blocks. The on-chip memory will require as many BRAMs as the size of the user-defined argument.

---

Therefore, you must ensure that the data would fit in the on-chip FPGA memory. The kernel code must declare the data as a static array, for example:

```
ACCESS_PATTERN(data, RANDOM);  
...  
static void PE_func(int n, int m, float data[64]);
```

If the data size is accessed randomly and too big for on chip FPGA memory, you should use `ZERO_COPY(data)`.



---

**IMPORTANT!** Do not use an `ACCESS_PATTERN` macro together with `ZERO_COPY`.

---

The unit amount of data transferred between host and the global memory is not necessarily the same as the DATA\_COPY size. It is actually determined by the size argument of VPP\_ACC::alloc\_buf() call. This size can be bigger than the data size needed for each kernel compute, for example when sending data for multiple compute() calls in one-shot. Thus, clustering PCIe data transfers say for N calls to compute() is easily done as follows:

```
send_while ... { ...
    clustered_buffer = acc::alloc_buf( N * size );
    for (i = 0; i < N; i++) { ...
        acc::compute(&clustered_buf[ i * size ] ...
    }
```

1. Allocate the appropriate data buffer size for the N compute calls
2. Call compute() N times where each call indexes into the clustered buffer

### Compute Payload Data Type

The compute() data type also determines the data layout on the global memory and therefore will affect accelerator performance. To allow the kernel to access the data as fast as possible it is important to choose the appropriate data type for the compute() arguments. For example, if the kernel is processing integers and is required to process one integer every clock cycle (the HLS II = 1), then the interface can use an array of integers, such as:

```
static void compute ( int* A );
```

Consider another example with the following two coding styles that add four integers in every compute call.

<pre>// --- acc interface DATA_COPY(data, data[numData*4]);  // --- application code int data[numData*4]; ...  static void acc::compute(int* data, ...);  // --- 4-cycle kernel code void PE_func(int* data, int numData, int *out) {     for (int i=0; i &lt; numData; i++) {         int o = i * 4;         out = data[o+0]+data[o+1]+data[o+2]+data[o+3];     } }</pre>	<pre>// --- acc interface DATA_COPY(data, data[numData]);  // --- application code struct data_t { int i[4]; }; data_t *data; ...  static void acc::compute(data_t* data, ...);  // --- 1-cycle kernel code with packed data // type void PE_func(data_t* di, int numData, int *out) {     for (int i=0; i &lt; numData; i++) {         data_t data = di[i];         out = data.i[0]+data.i[1]+data.i[2]+data.i[3];     } }</pre>
--	---

The kernel adds up every four integers from the input array. The straightforward implementation on the left would need 4 clock cycles for each result, assuming one cycle per memory access. However, it would be better to pack all 4 integers into a single global memory access, as shown on the right. Therefore, in this case it is recommended to:

- Use a C-struct to pack all the data and pass to the kernel

- Ensure the correct data copy size is provided, using `DATA_COPY(data, data[numData]);`

Some other key points to note:

- Using `int* data[4]` will not pack the integers and will result in the same hardware as just `int* data;`
- Typically packing more than 64 bytes (for a 512-bit M\_AXI bus width) will not improve performance, but should also not degrade performance

---

## Application Software Interface

### Host Code Organization

VSC provides a simple template for application code development and managing software calls to the accelerator hardware. Regardless of the hardware composition this template provides a unified style of creating a C++ software API.

```
auto arg1BP = myACC::create_bufpool( .... ); (1)
...
myACC::send_while([= .... ]() // (2)
{
    int* arg1 = myAcc::alloc_buf<int>(arg1BP, .... ); // (4)
    ...
    myACC::compute( arg1, .... ); // (5)
    ...
    return ( while_cond ); // (3)
}
myACC::receive_all_in_order([= .... ]() // (6)
{
    ...
}
myACC::join(); // (7)
```

The structure of this template shown in this pseudo-code above has the following parts. Refer to [VPP\\_ACC Class API](#) for details of these elements.

- `create_bufpool()` - Creates a buffer pool for each pointer argument passed to the accelerator and provides the specification of the argument data (for example: input, output, and remote).
- `send_while()` - A thread to control the overall scheduling of jobs on the accelerator, providing data for each job by using a lambda function.

- `return ( while_cond );` - The body of the lambda function executes in a loop and must return a Boolean value. The `return` statement in the `send_while` body allows the user to continue running the loop as long as the value returned is true, and to stop the loop and exit the `send_while` thread when the value returned is false. A `return` statement could be `(+ +sent_value<MAX_SEND)` if `sent_value` was set to 0 before declaring and using the lambda function.
- `alloc_buf()` - Allocation of a memory buffer object from the buffer pool for the current loop iteration.
- `compute()` - The software call to schedule one job execution of the `compute()` function on the accelerator hardware.
- `receive_all_in_order()` - A thread that waits on the results from the scheduled jobs. This is another user-defined lambda function and executes in a loop as long as the `send_while()` thread runs.
- `join()` - Waits on the completion of the send and receive threads.

The following sections provide additional details for creating the application code.

## VPP\_ACC Class API

You can define a hardware accelerator derived from the `VPP_ACC` base class, and build the hardware and software interface with VSC. This section describes the software API provided by the `VPP_ACC` class.

### Controlling the Accelerator

The `VPP_ACC` class API provides methods for scheduling jobs on the accelerator hardware, processing results, and other software controls at run time.

- `send_while():`

This API executes the `SendBody` repeatedly until it returns a boolean false; the pseudo-code of the `send_while` is similar to `do{ bool ret=f() } while(ret==true);`

```
void send_while(SendBody f, VPP_CC& cc = *s_default_cc);
```

Argument	Description
SendBody f	<p>SendBody is a user-defined C++ lambda function. The lambda function captures variables from the enclosing scope. Notably all used buffer pool variables need to be captured. They should be captured by value. Using <code>[=]</code> will automatically capture those by value as well any other variable you might use inside the lambda function. Any variable which gets modified inside the lambda function needs to be passed by reference. For example <code>[=, &amp;m]</code>. Passing variables by reference unnecessarily can result in degraded host code performance, but on the other hand, passing a large class object by value might lead to an unnecessary deep copy. The latter however is unlikely the needed for the send (or receive) functions.</p> <p><b>TIP:</b> Any variable which needs to be passed by reference can be captured explicitly with <code>[=, &amp;var]</code>.</p>
VPP_CC& cc	Optional argument used for grouping CUs. For example it can be used to specify which multi-card cluster of CUs to use as described in <a href="#">CU Cluster and Multi-Card Support</a> .

- **compute():**

As described in [The compute\(\) API](#), the `compute()` method is a special user-defined method in the derived `VPP_ACC` accelerator class definition which is used to represent the CU and contain the processing elements (PEs).

```
void compute(Args ...);
```

- a call to the hardware accelerator that schedules one job
- one or multiple `compute()` calls can be made inside the `SendBody` function
- each `compute()` call is non-blocking and will return immediately, but will block when the task pipeline is full.
- in the background, a `compute()` call will make sure that all its inputs get transferred to the device and then executed on any available CU
- once all `compute()` calls of an iteration have finished, the output buffers are transferred back to the host and a `receive_all` iteration will be started for that iteration
- The following conditions must be followed by the application code, and are asserted during software emulation
  - once a `compute()` call has been made, input buffers and file buffers cannot be modified anymore, and no more calls to `alloc_buf` or `file_buf` can be made in that iteration.
  - output buffers cannot be read or written until data is received in the corresponding `receive` iteration

- **receive\_all\_xxx():**

Executes a C++ lambda function repeatedly, either in order or ASAP, whenever a compute request completes to receive data results from the hardware accelerator. Exits when `send_while()` has exited and all iterations have been received.

```
void receive_all_in_order(RecvBody f, VPP_CC& cc = *s_default_cc);
```

```
void receive_all_asap(RecvBody f, VPP_CC& cc = *s_default_cc);
```

Argument	Description
RecvBody f	RecvBody is a user-defined C++ lambda function. Refer to the explanation of lamda functions in <a href="#">send_while()</a> .
VPP_CC& cc	Optional argument used for grouping CUs. For example it can be used to specify which multi-card cluster of CUs to use as described in <a href="#">CU Cluster and Multi-Card Support</a> .

- **receive\_one\_xxx():**

As described in [Multi-Accelerator Pipeline Composition](#), this is used to receive one iteration of this accelerator inside the `send_while()` loop of another accelerator.

```
void receive_one_in_order(RecvBody f, VPP_CC& cc = *s_default_cc);
```

```
void receive_one_asap(RecvBody f, VPP_CC& cc = *s_default_cc);
```

Argument	Description
RecvBody f	RecvBody is a user-defined C++ lambda function. Refer to the explanation of lamda functions in <a href="#">send_while()</a> .
VPP_CC& cc	Optional argument used for grouping CUs. For example it can be used to specify which multi-card cluster of CUs to use as described in <a href="#">CU Cluster and Multi-Card Support</a> .

- **join():**

Wait for send and receive loops to finish.

```
void join(VPP_CC& cc = *s_default_cc);
```

Argument	Description
VPP_CC& cc	Optional argument used for grouping CUs. For example it can be used to specify which multi-card cluster of CUs to use as described in <a href="#">CU Cluster and Multi-Card Support</a> .

- **set\_ncu():**

Set the number of CUs the driver should use. Use this method before starting the send/receive loop to establish the number of CUs the `compute()` function should use.

```
void VPP_ACC::set_ncu(int ncu);
```

Argument	Description
int ncu	The number of CUs specified (ncu) should be (1 <= ncu <= NCU) where NCU is the template parameter of VPP_ACC< . . . , NCU>, as described in <a href="#">User-Defined Accelerator Class</a> .

- **get\_ncu():**

Returns the number of CU currently used by the driver, as previously set by VPP\_ACC::set\_ncu, or if not modified, the NCU template parameter of VPP\_ACC< . . . , NCU>.

```
int VPP_ACC::get_ncu();
```

- **get\_NCU:**

Returns the number of CUs implemented in HW (i.e. the value of the NCU template parameter provided when building hardware, and specified in the base class VPP\_ACC< . . . , NCU>.

```
int VPP_ACC::get_NCU();
```

## Setup I/O for Computation

The API methods described here are used to setup input and output buffers to the hardware accelerator.

- **create\_bufpool():**

Creates and returns an opaque class object to be used in other methods that require a buffer handle, such as alloc\_buf(). Use before starting the send/receive loops.

```
VPP_BP VPP_ACC::create_bufpool(vpp::Mode m, vpp::FileXfer = vpp::none);
```

Argument	Description
vpp::Mode m	<p>Can specify any of the following values to denote the data transfer type for each compute() argument:</p> <ul style="list-style-type: none"> <li>• vpp::input: data transfers into the accelerator</li> <li>• vpp::output: data transfers out of the accelerator</li> <li>• vpp::bidirectional: data transfer into and out of the accelerator</li> <li>• vpp::remote: data is resident only on the device memory connected to the accelerator, and not send or received by the host code</li> </ul>

Argument	Description
vpp::FileXfer = vpp::none	<p>Can specify any of the following values to indicate the location of a file for data transfer:</p> <ul style="list-style-type: none"> <li>• <code>vpp::p2p</code>: file is transferred over the P2P bridge to the accelerator. This works only on platforms that support the P2P feature, for example the U2 card with a connected smartSSD.</li> <li>• <code>vpp::h2c</code>: file is transfer from a host CPU (connected file server) to the card over PCIe. This is standard for most Alveo cards connected to a host CPU over PCIe.</li> <li>• <code>vpp::none</code>: uses regular buffer objects, not supporting a file transfer. This is the default value.</li> </ul>

- **alloc\_buf():**

Returns a pointer to the buffer object. Use inside the send thread's lambda function.

```
void* VPP_ACC::alloc_buf(VPP_BP bp, int byte_sz);

T* VPP_ACC::alloc_buf<T>(VPP_BP bp, int numT);
```



**IMPORTANT!** *The buffer gets allocated from the given buffer pool. The lifetime of the buffer is until the end of the matching receive iteration. At that point the buffer will automatically be returned to the buffer pool.*

Argument	Description
VPP_BP bp	A buffer pool object returned by <code>create_bufpool()</code>
int byte_sz	Specifies the number of bytes for the requested buffer
int numT	Specifies the number of elements for the requested <T> array buffer

- **file\_buf():**

This method will map a given file, or part of the file to a buffer from the specified buffer pool object. Use inside the send thread's lambda function. The `file_buf()` method can be called multiple times to map multiple files (or file segments) to different locations in a single buffer.

The method returns a pointer to the buffer object which is just a host handle. The host cannot be used to read or write it.

```
void* VPP_ACC::file_buf(VPP_BP bp, int fd, int byte_sz, off_t
fd_byte_offset=0, off_t buf_byte_offset=0);

T* VPP_ACC::file_buf<T>(VPP_BP bp, int fd, int numT, off_t fd_T_index=0,
off_t buf_T_index);
```

Argument	Description
VPP_BP bp	A buffer pool object returned by <code>create_bufpool()</code> .
int fd	The file descriptor to read from or write to (or 0 when using <code>custom_sync_outputs()</code> as described below). In P2P mode the file must have been opened with <code>O_DIRECT</code> flag.
int byte_sz	Specifies the number of bytes for the requested buffer. In P2P mode, this must align to the file system block size (4 kB).
fd_offset	Offset in the file to read from/write to.
buf_offset	Offset in the buffer to write to/read from.
int numT	Specifies the number of elements for the requested <T> array buffer. In P2P mode, this must align to the file system block size (4 kB).
fd_T_index	The array index in the file to start reading from/writing to.
buf_t_index	The buffer index to start writing to/reading from.

Additional notes:

- The statement `T* buf = file_buf<T>(bp, fd, num, fd_idx, buf_idx);` is the same as `T* buf = (T*)file_buf(bp, fd, num*sizeof(T), fd_idx*sizeof(T), buf_idx*sizeof(T));`
- The actual size of the buffer will be adjusted as required. As a result the buffer returned by the last call needs to be used in the `compute()` call(s). Once used in a compute call, no more mappings can be added in that iteration.
- See `file_filter_sc` under **Startup Example** in [Supported Platforms and Startup Examples](#)
- **get\_buf():**
  - Use inside the receive loop associated with a matching send loop. This returns the buffer object which was allocated in the matching send iteration.

```
void* VPP_ACC::get_buf(VPP_BP bp);

T* VPP_ACC::get_buf<T>(VPP_BP bp);
```

Argument	Description
VPP_BP bp	A buffer pool object returned by <code>create_bufpool()</code>

- **transfer\_buf():**

This method is to be used in multi-accelerator composition, as described in [Multi-Accelerator Pipeline Composition](#), to transfer ownership of a buffer from one accelerator using a `receive_one_xxx()` method inside the `send_while` of another accelerator, to that other accelerator. This extends the lifetime of the buffer till the end of the receive iteration matching the current send iteration.

---

 **TIP:** This is especially useful for `vpp::remote_buffers`, because then the buffer will remain on the device and no copying or syncing will be needed.

---

```
void* VPP_ACC::transfer_buf(VPP_BP bp);
```

```
T* VPP_ACC::transfer_buf<T>(VPP_BP bp);
```

Argument	Description
VPP_BP bp	A buffer pool object returned by <code>create_bufpool()</code>

- **custom\_sync\_outputs():**

This method can be called in the body of a `send_while` loop, before the call to the `compute()` function. This lets you provide a custom sync function to sync output buffers back to the host application. It is useful when only some (and not all) output buffers data need to be transferred back from the hardware accelerator.



**IMPORTANT!** This will disable any automatic syncing of all output buffers.

---

```
void custom_sync_outputs(std::function<void()> sync_outputs_fn)
```

Argument	Description
sync_outputs_fn	Specifies the custom sync function that will be called automatically for each iteration of the <code>send_while</code> loop when the compute tasks of the iteration have finished. When the <code>sync_outputs_fn</code> returns AND all requested <code>sync_output()</code> calls are complete, a receive will be triggered for the <code>send_while</code> loop iteration.

- **sync\_output():**

This method is to be called inside the `sync_output_fn` passed to the `custom_sync_outputs()` method. It will perform the requested sync in the background, returning a future which the caller can check for completion of the transfer.

```
std::future<void> sync_output(void* buf, size_t byte_sz, off_t byte_offset = 0);
```

```
std::future<void> sync_output<T>(T* buf, size_t numT, off_t Tindex = 0);
```

Argument	Description
buf	Buffer pointer obtained as a capture from the <code>sendBody()</code> scope.
byte_sz	Specifies the number of bytes for the requested buffer.
byte_offset	Offset in the buffer to write to/read from.

Argument	Description
numT	Specifies the number of elements for the requested <T> array buffer. In P2P mode, this must align to the file system block size (4 kB).
Tindex	The buffer index to start writing to/reading from.

- **sync\_output\_to\_file():**

This method is to be called inside the `sync_output_fn` passed to the `custom_sync_outputs()` method. It will perform the requested sync in the background, returning a future which the caller can check for completion of the transfer.

```
std::future<void> sync_output_to_file(void* buf, int fd, size_t byte_sz,
off_t fd_byte_offset = 0, off_t buf_byte_offset = 0);
```

```
std::future<void> sync_output_to_file<T>(T* buf, int fd, size_t numT,
off_t fd_T_index = 0, off_t buf_T_index = 0);
```

Argument	Description
buf	Buffer pointer obtained as a capture from the <code>sendBody()</code> scope.
fd	The file descriptor to write to.
byte_sz	Specifies the number of bytes for the requested buffer.
fd_offset	Offset in the file to read from/write to.
buf_offset	Offset in the buffer to write to/read from.
numT	Specifies the number of elements for the requested <T> array buffer. In P2P mode, this must align to the file system block size (4 kB).
fd_T_index	The array index in the file to start reading from/writing to.
buf_t_index	The buffer index to start writing to/reading from.

- **set\_handle():**

Use inside the `sendBody` to identify any objects you want associated with the current iteration of the `send_while` loop.

```
void VPP_ACC::set_handle(intptr_t hndl);
```

```
void VPP_ACC::set_handle<T>(T hndl);
```

Argument	Description
hndl	Anything you might want to associate with the current iteration of the <code>send_while</code> loop.  <b>TIP:</b> With the templated form any class which has a simple assignment/copy operator can be used as a handle.

- **get\_handle():**

Used inside the `RecvBody`, this method returns the handle of an object that was set (`set_handle()`) in the matching send iteration of the `send_while` loop.

```
intptr_t VPP_ACC::get_handle();
```

```
T VPP_ACC::get_handle<T>();
```

## Special Data Transfer Models

This section describes certain specialized data transfers to and from the accelerator, such as parts of device (result) buffers, and different styles of file transfers allowing a single accelerator computation to simultaneously process multiple files.

### Customized Transfer of I/O Sub-Buffers

Sometimes you do not want to sync the whole output buffer back to the host, and the sizes of what you want to sync back are also not known up front in the host code. A good example is compression. Especially if the compression algorithm is compressing multiple chunks into the same output buffer and you want to sync back only the exact compressed chunks from the output buffer. As shown in the following example, this can be done by registering a function which will control exactly what will be synced back once the data is available.

```
xfilter::custom_sync_outputs( [=]()
{
    auto fut = xfilter::sync_output<int>(outSz, chunks, 0);
    fut.get();
    for (int chunk = 0; chunk < chunks; ++chunk) {
        xfilter::sync_output<int>(out, outSz[chunk], chunk * chunkSz);
    }
});
```

The `custom_sync_outputs()` method registers a callback function which will determine which buffer/sub-buffers will be synced back to the host. This method would have to be called inside the `send_while` body, before the first `compute()` call.



---

**IMPORTANT!** Calling `custom_sync_outputs()` disables any automatic transfer-back of output buffers.

---

Inside the callback function the user-defined code has full control of what is synced back and if there needs to be synchronization. The example above syncs back an `outSz` buffer. This buffer contains the sizes of the compressed chunks. The `sync_output` API returns a `std::future`. Calling `fut.get()` on that future will make the code wait for that sync to finish. Then the code transfers all chunks, with their exact sizes, back to the host. These calls will also return a future, but there is no need to synchronize for those syncs anymore. The System Compilation runtime layer will make sure that all those syncs have finished before starting the corresponding `receive_all` iteration.

## File Transfer Modes

System Compilation mode also supports easy reading and writing to files as shown in the `file_filter_sc` in [Supported Platforms and Startup Examples](#). This can be enabled in the `create_bufpool()` method using either of the following modes as discussed in [VPP\\_ACC Class API](#):

- `vpp::p2p` mode : the file is transferred over the peer-to-peer PCIe bridge to the accelerator. This works only on platforms that support the P2P feature, for example the U2 card with a connected smartSSD.
- `vpp::h2c` mode : file is transfer from a host CPU (connected file server) to the card over PCIe. This is standard for most Alveo cards connected to a host CPU over PCIe.

This simple file-transfer switch (`vpp::p2p` and `vpp::h2c`) makes accelerator design portable across platforms. It it is a simple matter to test a design for any platform using software emulation on a typical host CPU (connected to a file server) hosting the data files. However, eventually, the design must be compiled for a specific platform supporting P2P, such as the U2 card connected to a smartSSD, thereby allowing direct porting without changing the design sources.



**TIP:** *Running hardware emulation or running on hardware will not work on an Alveo platform that does not support P2P.*

The following example code demonstrates this:

```
auto inBP = my_acc::create_bufpool(vpp::input, P2P ? vpp::p2p : vpp::h2c);
my_acc::send_while([=]() {
    if (P2P) o_flags |= O_DIRECT;
    int fd = open(fnm, o_flags, s_flags);
    DT* in = (DT*)xfilter::file_buf(inBP, fd, fsz);
    my_acc::compute(in, ...);
    ...
});
```

Based on the value of the `P2P` flag, this code will either do peer-to-peer (P2P) transfer of a host mapped NVMe device file, or it will do host file to device transfer (H2C). In the P2P case, the file will be loaded into the device memory directly from the NVMe device, without any data transfer through the host. In case of H2C, the System Compilation runtime layer will automatically transfer the file from the host to the device buffer (or vice versa for outputs).

To enable P2P the file has to be opened with `O_DIRECT` flag specified as shown in the example above. When in P2P mode, the host pointer, as returned by the call to `VPP_ACC::file_buf`, is just a handle for the `compute` call argument, and cannot be read from or written to.

## Multi-File Buffers

As described in [VPP\\_ACC Class API](#), you can make multiple calls to the `file_buf` method before calling the `compute()` method, to map multiple files, or multiple file segments into a single device buffer. The code example below shows small portions of multiple files being processed simultaneously by the accelerator in one `compute()` call.

```
void* VPP_ACC::file_buf(VPP_BP bp, int fd, size_t sz, off_t fos = 0, off_t
bos = 0)

xfilter::send_while([=, &total_out_size]()
{
    static int iter = 0;
    int* in;
    // collect all "chunks" input files into one "in" buffer
    for (int chunk = 0; chunk < chunks; ++chunk) {
        std::stringstream nm;
        nm << DATA << iter << '-' << chunk << ".orig";
        int ifd = open(nm.str().c_str(), rd_o_flags);
        assert(ifd > 2);
        in = xfilter::file_buf<int>(inBP, ifd, chunkSz, 0, chunk * chunkSz);
    }
    // prepare output buffer to be able to hold all chunks
    int* out = xfilter::file_buf<int>(outBP, 0, chunks * chunkSz, 0);
    // output buffer to provide the actual filtered size of each chunk
    int* outSz = xfilter::alloc_buf<int>(outSzBP, chunks);
    ...
    xfilter::compute(chunks, chunkSz, in, out, outSz);
    ...
});
```

The code creates an input (`in`) buffer that holds these file segments, and an output (`out`) buffer that holds the processed output data. In every `send_while` iteration the `file_buf()` provides the `chunkSz` and read offset (`chunk*chunkSz`) for each file associated with the `in` buffer. In the subsequent call to `compute()` all those files segments will be written to the input or read from the output device buffers.

**Note:** The assignment `in = xfilter::file_buf<...>` is repeated in the for-loop so that the last returned pointer is assigned to `in`. This is important to follow while writing the application code.

## Custom Transfer of Output files

You can also use custom sync for file buffers, in which an output buffer can be synced to a file descriptor in `custom_sync_outputs()`. As explained in [VPP\\_ACC Class API](#), you can do this by calling the `sync_output_to_file()` method as shown in the following example.

```
VPP_ACC::sync_output_to_file(void* buf, int fd, size_t byte_sz,
                             off_t fd_byte_offset = 0,
                             off_t buf_byte_offset = 0);
```

In this case files added by a call to `VPP_ACC::file_buf(bufPool, fd, sz)` will not get synced automatically. So a call to the `file_buf` API is not needed to actually add a file, but it is needed just to return a host pointer. Adding a dummy file like this is the best approach:

```
VPP_ACC::file_buf(bufPool, 0, 0);
```

Here's an example code snippet:

```
my_acc::send_while([=]() {
    DT* out = (DT*)my_acc::file_buf(outBP, 0, 0);
    ...
    my_acc::custom_sync_outputs([=](){
        ...
        auto fut = my_acc::sync_output_to_file(out, fd, sz, fd_offset,
buf_offset);
        ...
    });
    ...
}
```

## Accelerator Execution Models

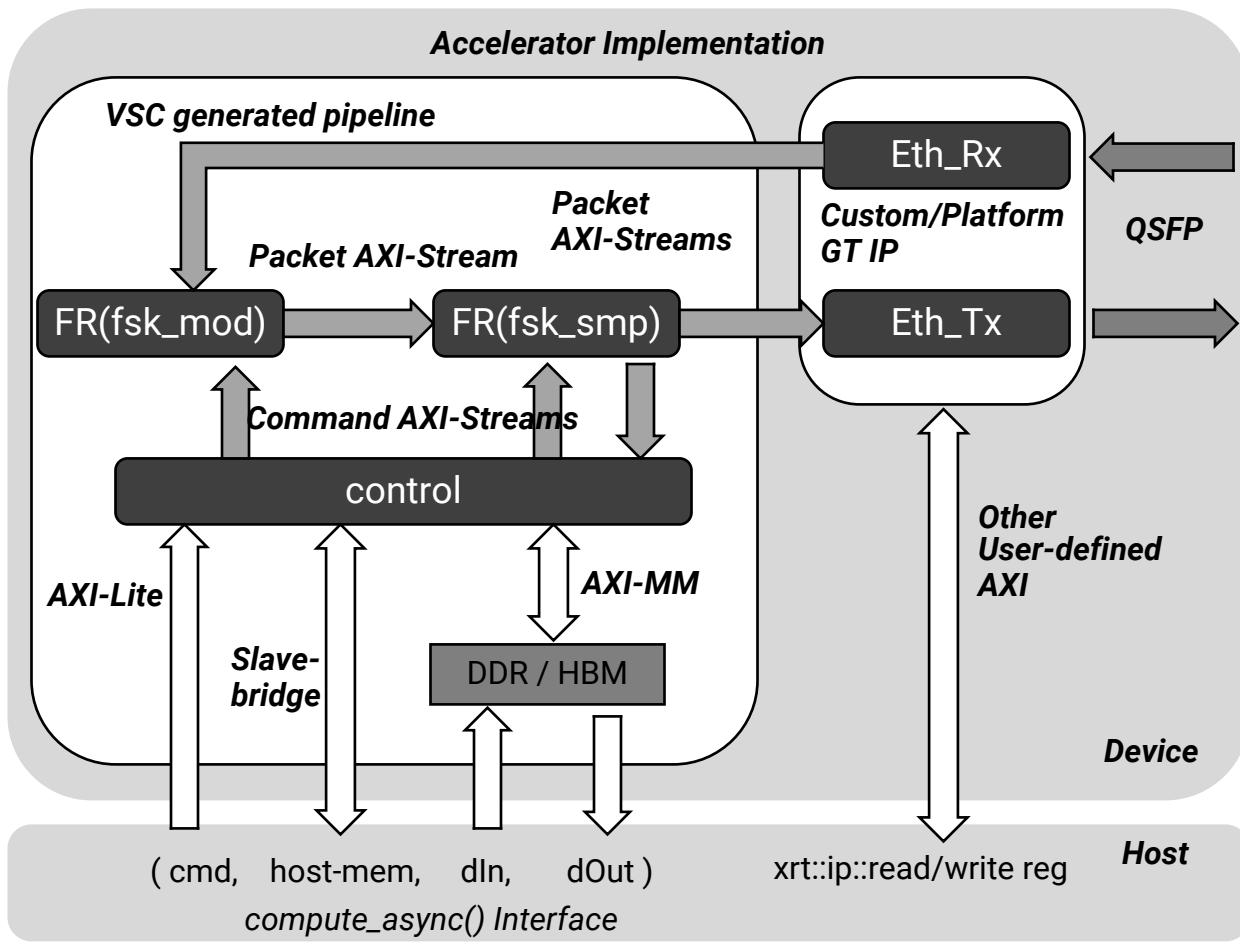
The following sections describe various styles of accelerator execution models. Each one is uniquely defines a system architecture that may be used for an accelerator application domain.

### ***Asynchronous Host Control of Accelerator***

The VSC mode allows compilation of accelerators with CUs that contain user-defined hardware pipelines, as described in [Building Hardware](#). Such a pipeline is composed of PEs that connect to each other through AXI4-Stream and can also connect to platform ports that are AXI4 connections, such as global memory or IO interfaces such as an ethernet QSFP port. The platform will provide IP that translates such interfaces into AXI4-Stream ports which can be connected to PEs in the user-defined pipeline.

Using VSC such hardware pipelines can be easily configured to dynamically change processing behavior at runtime from an application running in the host CPU. The following describes how such an accelerator can be created. An example system composition is shown in the picture given below.

Figure 142: Accelerator Implementation



The **Eth\_Rx** and **Eth\_Tx** modules are typically platform IP that translate AXI4-Stream words into ethernet packets. These can also be custom IP with user-defined AXI4 interfaces.

The rest of the accelerator pipeline, shown in the white box, is created with VSC using AXI4-Stream connections. The PEs in the pipeline are user-defined functionality, such as packet processing like an internet protocol packet filter. In this example there is a pipeline created with two tasks, which are the PEs called `mod` and `smp`. Additionally, the system is composed of another control PE that has AXI4-Stream connections to these pipeline PEs. Example accelerator code is provided below, with the `.hpp` file on the left and the `.cpp` on the right.

```
// -- file: ETH.hpp --
#include "vpp_acc.hpp"
class ETH : public VPP_ACC<ETH,1>
{
public:
    ZERO_COPY(dIn);
    ZERO_COPY(dOut);

    SYS_PORT(dIn, MEM_BANK0);
    SYS_PORT(dOut, MEM_BANK1);

    FREE_RUNNING(fsk_mod);
    FREE_RUNNING(fsk_smp);
    FREE_RUNNING(eth_rx);
    FREE_RUNNING(eth_tx);

    static void compute(int cmd, Pkt* dIn,
Pkt* dOut);

    static void control(...);
    static void fsk_mod(...);
    static void fsk_smp(...);
    static void eth_tx(...);
    static void eth_rx(...);
};
```

```
// -- file: ETH.cpp --
void ETH::compute(int cmd, Pkt* dIn, Pkt*
dOut)
{
    static IntStream dropS("drop");
    static IntStream addS("add");
    static IntStream reqS("req");
    static PktStream smpS("smp");
    static BitStream getS("get");
    static IntStream sntS("snt");
    static PktStream Ax("Ax", /*post_check=*/false);
    static PktStream Bx("Bx", /*post_check=*/false);
    static PktStream Cx("Cx", /*post_check=*/false);

    control(cmd, dIn, dOut,
            dropS, addS, reqS, smpS, getS,
            sntS);
    eth_rx(Ax);
    fsk_mod(Ax, Bx, dropS, addS);
    fsk_smp(Bx, Cx, reqS, smpS);
    eth_tx(Cx, getS, sntS);
}
```

In this example, five PEs are defined including the `eth_tx` and `eth_rx` which mock the platform IP behavior in receiving and transmitting words in the AXI4-Stream. The `compute()` scope implements the accelerator pipeline using AXI4-Stream connections between these PEs. The `control` PE can send command words on these streams and the task PEs (`fsk_mod` and `fsk_smp`) monitor these command AXI4-Stream and react by changing behavior. The `fsk_smp` PE reacts by sampling a requested number of packets back to the `control` PE. The `fsk_mod` PE reacts by adding a value to the packet data or by dropping packets that are being passed from Eth\_Rx into Eth\_Tx.

The pipeline PEs, `fsk_mod` and `fsk_smp`, are `FREE_RUNNING` as described in [Guidance Macros](#) because they are never-ending PEs driven to operation by the words in their input streams.

The `control` PE talks to the host CPU through two `SYS_PORT` connections for interface argument data pointers for input (`dIn`) and output (`dOut`), as well as the scalar command argument (`cmd`). The `control` PE is not free-running and reacts to `compute()` calls from the host CPU. This system composition is entirely user-defined including the nature of the commands and corresponding PE functionality.

The host code snapshot is shown here and the entire example is available on [GitHub](#).

```
// -- file: host.cpp --
#include "vpp_acc_core.hpp" // required
#include "ETH.hpp"
int config_sample(int sz)
{
    printf("main: sample %d\n", sz);

    Pkt* sample = (Pkt*)ETH::alloc_buf(sz * sizeof(Pkt), vpp::output);
    Pkt* config = (Pkt*)ETH::alloc_buf(sizeof(Pkt), vpp::input);
    config[0].dt = sz;

    auto fut = ETH::compute_async(cmd_sample, config, sample);
```

```
fut.get();
print_sample(sample, sz);
int pkt_nr = sample[0].nr;
ETH::free_buf(config);
ETH::free_buf(sample);
return pkt_nr;
}
```

The job commands issued by the VSC host code, specifically using the `compute_async()` API, enables the control PE to translate the command and in-turn pass configuration words to the pipeline PEs through the command streams. This snapshot shows a user-defined API that issues a packet sampling command. A `sample` buffer of a required `sz` is allocated at run time, and the `compute_async()` call will trigger the control PE to capture `sz` number of packets and return the words back to the host. The `fut` returned by `compute()` is blocking in the host code until the results are available. However, the `compute_async()` as name denotes is an asynchronous call that triggers the accelerator. Once the sample words are returned and processed by the host and the corresponding buffers can be freed.

Because host control in this case is not a continuous pipeline of compute jobs, but just an occasional, non-timing critical job, the `send_while/receive_all` thread will not manage this. Instead, the synchronization is application managed using the `compute_async()` API defined in `vpp_acc_core.hpp`.



**IMPORTANT!** This `vpp_acc_core.hpp` header file needs to be included only in the host code file, and before any `vpp_acc.hpp` is included.

With VSC such hardware pipelines can be composed and be controlled asynchronously from a host CPU. One of the applications for such an accelerator is a packet processing accelerator on a NIC card. For example the X3 Hybrid platforms provides ethernet transmission and reception IPs which convert ethernet packets arriving at the QSFP ports into AXI4-Stream, through a MAC interface. Furthermore, a NIC interface allows the accelerator to provide data to a connected host CPU over PCIe, or a Host-Memory access may be used for direct host CPU memory access over PCIe. Using VSC, the accelerator packet processing pipeline can be composed on the PL and can be controlled by a CPU asynchronously over PCIe.

## CU Cluster and Multi-Card Support

If a host machine has only one accelerator card installed, VSC will try to use that card. There will be a fatal error if the card does not match the platform that the design was compiled for. However, on a host machine which has multiple cards installed, VSC will by default pick the first card that exactly matches the platform the design was compiled for. The host code can override the default in two ways:

- Setting environment variable `XILINX_SC_CARD` to the desired `<cardIndex>`.
- From the host code, call the following API before making any other calls.

```
VPP_ACC call: my_acc::add_card(<cardIndex>)
```



**TIP:** With multiple cards installed, if there is any mismatch between the names of installed platforms and the platform the design was compiled for VSC will not identify a default card. In such a case you must specify the card index as shown above. For example, platform `xilinx_u2_gen3x4_xdma_gc_base_2` will not match a design compiled for `xilinx_u2_gen3x4_xdma_gc_2_202110_1`, even though the platforms are compatible.

As shown in the `sysc_multi_card` example in [Supported Platforms and Startup Examples](#), if the host has identical accelerator cards installed you can use multiple cards to run your VSC accelerator. This is supported in a mode where all CUs of any given card are running as a separate compute cluster as explained below.

The separate compute cluster mode is useful for performance improvement in scenarios like the U2 card with a local smartSSD. This example code show below creates CU-clusters and assigns a card to each of them. Then, the user code can perform data selection based on the index-i to ensure that the subsequent `compute()` job will automatically use the card-i because the selected data-i resides on the same SSD.

```
VPP_CC* cuCluster = new VPP_CC[ncards];
for (int i = 0; i < ncards; ++i) {
    my_acc::add_card(cuCluster[i], i);
}
for (int i = 0; i < ncards; ++i) {
    my_acc::send_while(
        [=]() -> bool {
            ... // data-i selection
        }
        , cuCluster[i]);
    my_acc::receive_all_in_order(
        [=]() {
            ...
        }
        , cuCluster[i]);
}
for (int i = 0; i < ncards; ++i) {
    my_acc::join(cuCluster[i]);
}
```

## Multi-Accelerator Pipeline Composition

In VSC you can also create multiple accelerators with different functionality in a single .xclbin and runtime environment. With such composition you can create a pipeline, where two or more VSC accelerators operating of different data sets in a pipelined fashion, as shown in the `sysc_compose` example in [Supported Platforms and Startup Examples](#). There are two possible use models described in the following sections.

## ACC1-CPU-ACC2 Pipeline

This model defines a pipeline of tasks where the first accelerator computes an intermediate result which is then processed by a host application, and then further processed by a second accelerator. The output buffer of the first accelerator needs to be modified (modify while copying) and then passed to the second accelerator. Example code is provided below.

```
auto inBP      = my_acc1::create_bufpool(vpp::input);
auto ttmpoBP   = my_acc1::create_bufpool(vpp::output);
auto tmpiBP    = my_acc2::create_bufpool(vpp::input);
auto outBP     = my_acc2::create_bufpool(vpp::output);
my_acc1::send_while(
    [=] () -> bool {
        int* in = my_acc1::alloc_buf<int>(inBP, inSz);
        int* tmp = my_acc1::alloc_buf<int>(ttmpoBP, tmpSz);
        my_acc1::compute(in, tmp, ...);
        ...
        return ...;
    });
my_acc2::send_while(
    [=] () -> bool {
        int* tmp2 = my_acc2::alloc_buf<int>(tmpiBP, tmpSz);
        bool cond = my_acc1::receive_one_in_order( // or receive_one_asap
            [=] () {
                int* tmp1 = my_acc1::get_buf<int>(ttmpoBP);
                ...; // tmp1 -> copy and modify -> tmp2
            });
        if (!cond) return false;
        int* out = my_acc2::alloc_buf<int>(outBP, outSz);
        my_acc2::compute(tmp2, out, ...);
        ...
        return true;
    });
my_acc2::receive_all_in_order(
    [=] () {
        int* out = xfilter1::get_buf<int>(outBP);
        ...
    });
my_acc1::join();
my_acc2::join();
```

This code uses the dedicated `receive_one_in_order` API within the scope of the `send_while` loop of the second accelerator, `my_acc2`. The `receive_one_in_order` or `receive_one_asap` APIs requires a user-defined lambda function body, as discussed in [VPP\\_ACC Class API](#).

In this case, the `my_acc1::receive_one_in_order` (or `receive_one_asap`) will wait for the next job in-order (or asap) to finish, and then execute the lambda function body. Because it is called from the `send_while` of the second accelerator, `my_acc2` computes in lock-step with the results generated from the first accelerator `my_acc1`. This API returns a boolean value which will be true when the `send_while` loop has exited and there are no more jobs to be received.

## ACC1-ACC2 Pipeline

This model defines a pipeline of tasks where an output buffer of the first accelerator can be used directly (without any host CPU synchronization) by the second accelerator. As described in [VPP\\_ACC Class API](#), the `transfer_buf()` API takes a buffer pool object and returns the buffer corresponding to the correct iteration. Using this API allows seamless transfer of the result buffer from the first to the second accelerator without needing to copy data. VSC runtime automatically manages the re-use of such transferred buffers across accelerators and job iterators.

The following is a code example:

```
auto inBP      = my_acc1::create_bufpool(vpp::input);
auto tmpBP     = my_acc1::create_bufpool(vpp::remote);
auto outBP     = my_acc2::create_bufpool(vpp::output);
my_acc1::send_while(
    [=] () -> bool {
        int* in = my_acc1::alloc_buf<int>(inBP, inSz);
        int* tmp = my_acc1::alloc_buf<int>(tmpBP, tmpSz);
        my_acc1::compute(in, tmp, ...);
        ...
        return ...;
    });
my_acc2::send_while(
    [=] () -> bool {
        int* tmp;
        bool cond = my_acc1::receive_one_in_order( // or receive_one_asap
            [=, &tmp] () {
                tmp = my_acc1::transfer_buf<int>(tmpBP);
            });
        if (!cond) return false;
        int* out = my_acc2::alloc_buf<int>(outBP, outSz);
        my_acc2::compute(tmp, out, ...);
        ...
        return true;
    });
my_acc2::receive_all_in_order(
    [=] () {
        int* out = xfilter1::get_buf<int>(outBP);
        ...
    });
my_acc1::join();
my_acc2::join();
```

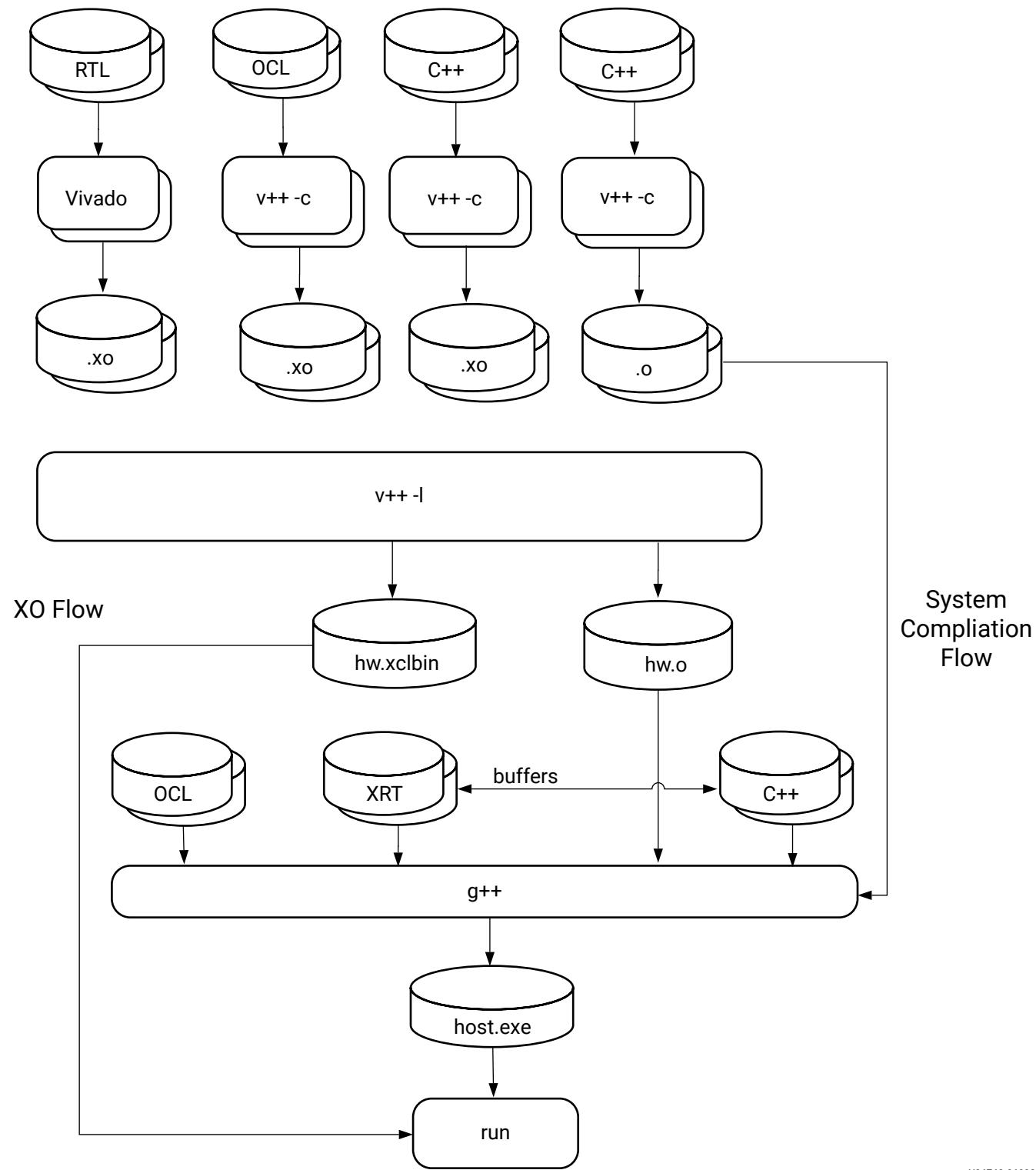
## Interoperability with XO and RTL Kernels



**IMPORTANT!** There is no software emulation support for the XO design flow described here.

Vitis System Compilation mode can also be used with standard Vitis kernels (`.xo`) and the application acceleration design flow. Therefore, you can use VSC with Vitis HLS C++ kernels, as described in *Vitis High-Level Synthesis User Guide* ([UG1399](#)), and RTL kernels, as described in *Vitis Unified Software Platform Documentation: Application Acceleration Development* ([UG1393](#)).

Figure 143: XO Flow



X26762-060322

Given a VSC accelerator code, `sc_acc.cpp`, and a Vitis kernel, `kernel.xo`, the following would be the command lines required to build the project:

```
v++ -c sc_acc.cpp -o sc_acc.o
v++ -l sc_acc.o kernel.xo -o hw.xclbin # also produces hw.o

g++ -c host.cpp -o host.o
g++ -l -o host.exe host.o sc_acc.o hw.o -lvpp_acc -lxrt_hw -lpthread
```

In the scenario as it has been described, the `kernel.xo` is already compiled. So, the VSC accelerator code (`sc_acc.cpp`) needs to be compiled as shown in the first `v++` command above. The output of this is the object file `sc_acc.o`. The `v++` command recognizes the VSC accelerator code, and compiles it properly into the object file.

The `v++ --link` command is then run to link the `.xo` and `.o` files into the `.xclbin`. Notice that this produces a `hw.o` file in addition to the `hw.xclbin` file. The `hw.o` file is required for compilation of the host application by the `g++` command as is shown in the next steps.

The `g++` command compiles the host code, and then links the various object files (`host.o`, `sc_acc.o`, and `hw.o`) to produce the `host.exe` executable. Notice the addition of the `vpp_acc` and the `xrt_hw` libraries to the `g++` command line as they are required for the linking process.

In a standard Vitis flow, using native-XRT or OpenCL code, the user host code has to explicitly load the `.xclbin`. In a mixed VSC-XO flow the user host code can still load the `.xclbin` explicitly, but is not required. Instead, the device-ID can be obtained by calling API `vpp::sc::get_xrt_device()`.

Buffers objects can be easily passed between the native XRT and VSC APIs. The two possible buffer transfer scenarios are described in the following sections.

## ***Passing Buffer from XO Kernel to VPP\_ACC***

The following API can be used to pass an `xrt::bo` over to a VSC accelerator:

```
std::shared_ptr<...> vpp::sc::set_xrt_bo(xrt::device, xrt::bo, void* buf,
int memBank);
```

This `buf` can then be used in the VSC accelerator `compute()` calls as long as the returned shared pointer keeps references.



**TIP:** Such buffers will not be auto-synced from/to the device, unlike buffers obtained by `vpp_acc::alloc_buf`.

Following is an example use case:

```
xrt::memory_group memA = 0;
auto Abo = xrt::bo(device, bytes, memA);
auto Abuf = Abo.map();
auto Aref = vpp::sc::set_xrt_bo(device, Abo, Abuf, memA);
auto Bbp = my_acc::create_bufpool(vpp::input);
my_cc::send_while([=]()->bool {
    auto Bbuf = my_acc::alloc_buf(Bbp, bytes);
    my_acc::compute(Abuf, Bbuf, ...);
    ...
});
```

## ***Passing a Buffer from VPP\_ACC to XO Kernel***

To go from the VSC accelerator buffer to an `xrt::bo` use the following code:

```
xrt::bo vpp::sc::get_xrt_bo(void* buf);
```



**TIP:** Be aware though that the lifetime of such a buffer will be fully controlled by the VSC runtime layer. The lifetime of a VSC buffer starts from the time it gets allocated in an iteration of the `send_while` loop, using `vpp_acc::alloc_buf`, until the end of the corresponding `receive_all` iteration.

For example:

```
Acc::send_while([=]()->bool {
    auto* Abuf = Acc::alloc_buf(Abp, size);
    xrt::bo Abo = vpp::sc::get_xrt_bo(Abuf);
    auto run = xo_kernel(Abo, ...);
    run.wait();
    ...
});
```

---

# **Debugging and Validation**

This section describes various types of debugging and validation features available for use with the VSC mode, which includes software emulation, hardware emulation, and profiling the actual execution on the physical accelerator card. All of these methods use the standard Vitis interfaces as described in [Section V: Simulating the Application with the Emulation Flow](#) and [Section VI: Profiling and Debugging the Application](#).

## **Software Emulation**

VSC uses a single source C++ model which can be functionally validated fast. The software emulation target (-t sw\_emu) must be specified during v++ compile step. For this target, VSC will compile the accelerator and application source files using C-compilation and will not run a hardware compilation. The Vitis HLS tool will not be run to create RTL, and the object files are linked using g++.

The following is an example code of an accelerator with two PEs, `ldSt` and `fsk`. The `ldSt` PE writes `sz` words in the AXI4-Stream `L`, and the `fsk` PE simply copies those words back in the feedback stream `S`. The `fsk` function body reads one word from `L` and writes one word to `S`. The `fsk` PE is marked free-running and therefore is agnostic to `sz`.

```
void compute(...) {
    hls::stream<T> L, S;
    ldSt(..., L, S); // S is feedback
    fsk(L, S);      // fsk is free-running
}
void ldSt(..., hls::stream<T>& L,
          hls::stream<T>& S) {
    for (int i=0; i<sz; i++) {
        L << input[i];
    }
    // Error-1: non-empty stream (i < sz-1)
    // Error-2: deadlock stream (i < sz+1)
    for (int i=0; (i < sz); i++) {
        S >> output[i];
    }
}
void fsk(hls::stream<T>& L,
          hls::stream<T>& S) {
    T word;
    L >> word;
    S << word;
}
```

Because the C++ source is entirely built with C-compilation the process is very fast. Additionally, VSC checks the model against certain hardware behavior semantics of the VSC C++ model:

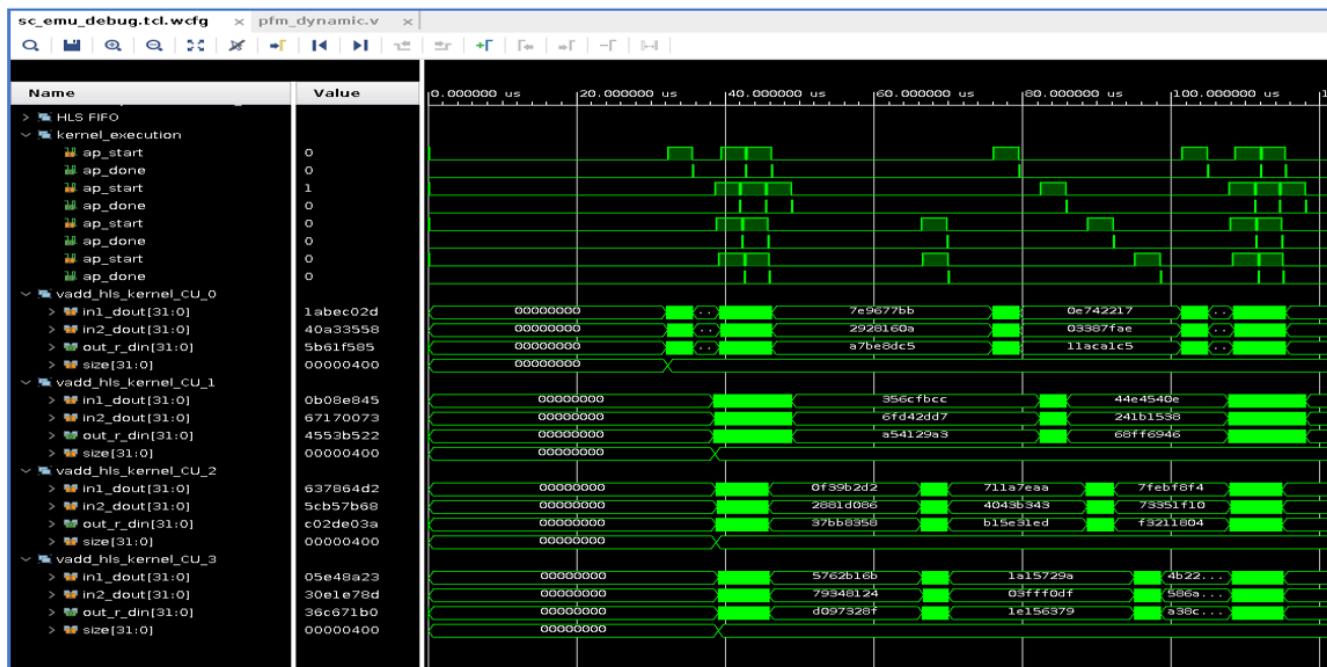
- Automatic runtime assertion for host-device data movement, in the application layer
  - Input data buffer might not be written after it is synced to the device
  - Output data buffer might not be read before the result is synced from the device
- The CUs and PE are executed in parallel, to model the hardware semantics. Therefore,
  - feedback connections, that are non-procedural C++ code, can be functionally validated
  - free-running PEs can be functionally validated along with regular PEs
- Certain erroneous hardware behavior can be detected
- Error-1: If the second loop in `ldSt` was using "`i < sz-1`" then it will lead to non-empty AXI4-Stream '`S`', which will be asserted
- Error-2: If the second loop in `ldSt` was using "`i < sz+1`" then it will lead to a deadlock as `ldStr` is expecting one more word than what is written to stream '`S`'. This scenario will be captured as a hang in this early C-based validation.

## Hardware Emulation

VSC mode when compiled for the hardware emulation target (-t hw\_emu) will generate RTL from the accelerator sources and run RTL simulation along with the application layer code. To view simulation waveforms the user can enable the following switch in the `xrt.ini` file:

```
[Emulation]
debug_mode=gui
```

Figure 144: Simulation Waveform View



The picture above shows the waveforms viewer in the Vivado XSim interface. By default, VSC will create grouped waveform objects corresponding to the `compute()` interface. The kernel code must have been compiled with -g flag, otherwise the grouping will error out in Vivado XSim. In this design, there are four instances (NCU=4) of the accelerator enumerated as grouped objects `vadd*_CU_0` through `vadd*_CU_3`. Each of these groups further contains the signals that correspond to `compute()` arguments, `in1`, `in2`, `out`, and `size`. The group `kernel_execution` is also auto-created and contains the `ap_start` and `ap_done` signals for each CU instance.



**TIP:** In hardware emulation, the timing of the start signals is not timing accurate with respect to realistic hardware behavior. This is because interactions of the host with the emulation model (data transfer latency over PCIe or with DDR/HBM memories), and application will not reflect real-time hardware execution behavior. However, the timing from the start to a stop of a `compute()` call is cycle-accurate.

## Profiling Execution on the Card

VSC Profiling allows getting real-time statistics of the application execution along with the accelerator card. There are three parts to the profiling statistics:

1. profile summary: summary tables with runtime statistics from an application execution
2. application traces: A graph of timeline traces as observed from the application layer
3. hardware traces: A graph of timelines traces of the hardware interfaces within the accelerator

Profiling can be enabled with these settings in the `xrt.ini` file:

```
[Debug]
sc_profile=true
xrt_trace=true
```

When the application is run on the card, at the end of the execution, the file `xrt.run_summary` will be created for viewing in the Vitis Analyzer tool as described in *Vitis Unified Software Platform Documentation: Application Acceleration Development* ([UG1393](#)).

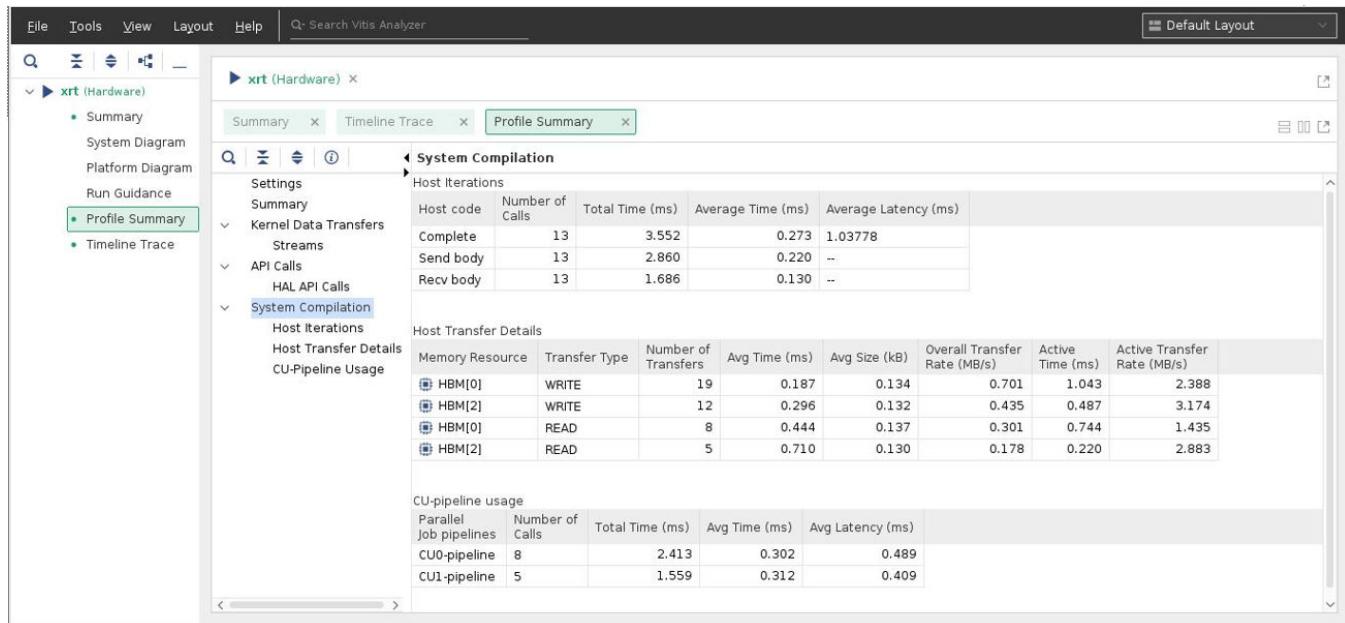
## Profile Summary

The `xrt.run_summary` file can be loaded into Vitis Analyzer and an example of the results is shown in the picture. The application run summary is presented under the System Compilation section of the Profile Summary tab. It contains the following three tables,

1. Host Iterations: This table shows a summary of the send and the receive threads iterations, and the number of `compute()` calls, issued by the application layer and useful runtime statistics. In this example, 13 compute jobs were run with a total runtime of 3.552 milliseconds.
2. Host Transfer Details: This table captures the number of data transfers between the application code (running on a host) and the device. In this example, data is transferred using the HBM banks 0 and 2. Several average runtime statistics on the data transfers are shown.
3. CU-pipeline usage: This table captures the activity in the compute unit pipelines. Each row represents a CU in the hardware and average run time statics are shown, In this example, the 13 compute jobs were distributed with 8 on CU0 and 5 on CU1.

**Note:** These columns have tooltips in Vitis Analyzer. If the mouse hovers over any column header in the table, a pop-up will show the explanation of the column. For example, hovering over "Average Time (ms)" will show "= Total time (in milliseconds) / Number of calls."

**Figure 145: Profile Summary**

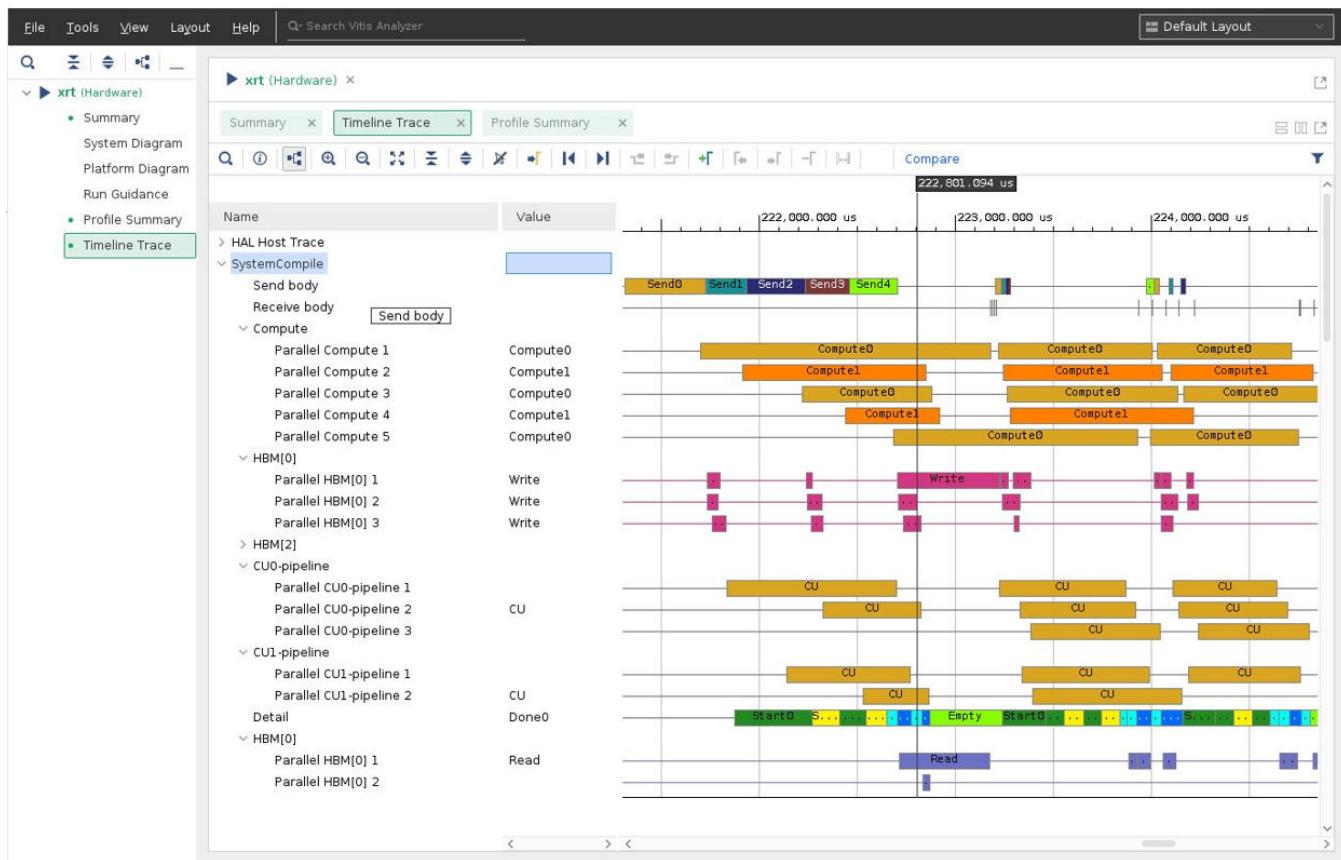


## Timeline Trace

Profiling also shows a timeline trace for various phases of run in the application layer, as shown in the picture. This is presented under the "SystemCompile" tab in Vitis Analyzer. These timeline trace rows are explained below.

- Send body: each box captures the start and end of a send iteration, including the execution of any user-written code in the send lambda function.
- Receive body: each box captures the start and end of a receive iteration, including any user-written code. In this example, the receive iteration is very fast (tiny in the picture) compared to that of send.
- Compute: each box captures the start of a compute call until the application receives a result (a get() on the C++ future).
- Input transfers: these are input data transfers for each compute job. In this example, they are writes to the HBM memory bank.
- CU-pipeline: the application layer maintains multiple software job pipelines, typically more than the number of CUs in the hardware, for efficient execution. Each row is a separate software pipeline, and it captures every job execution from the start of submission in this pipeline until the completion of the job by the hardware.
- Detail: this row captures some key events in the execution model, primarily the issuance of the start signal and receiving the done for each compute call.
- Output transfers: these are output data transfers for each compute job. In this example, they are data reads from the HBM memory bank.

*Figure 146: Timeline Trace*



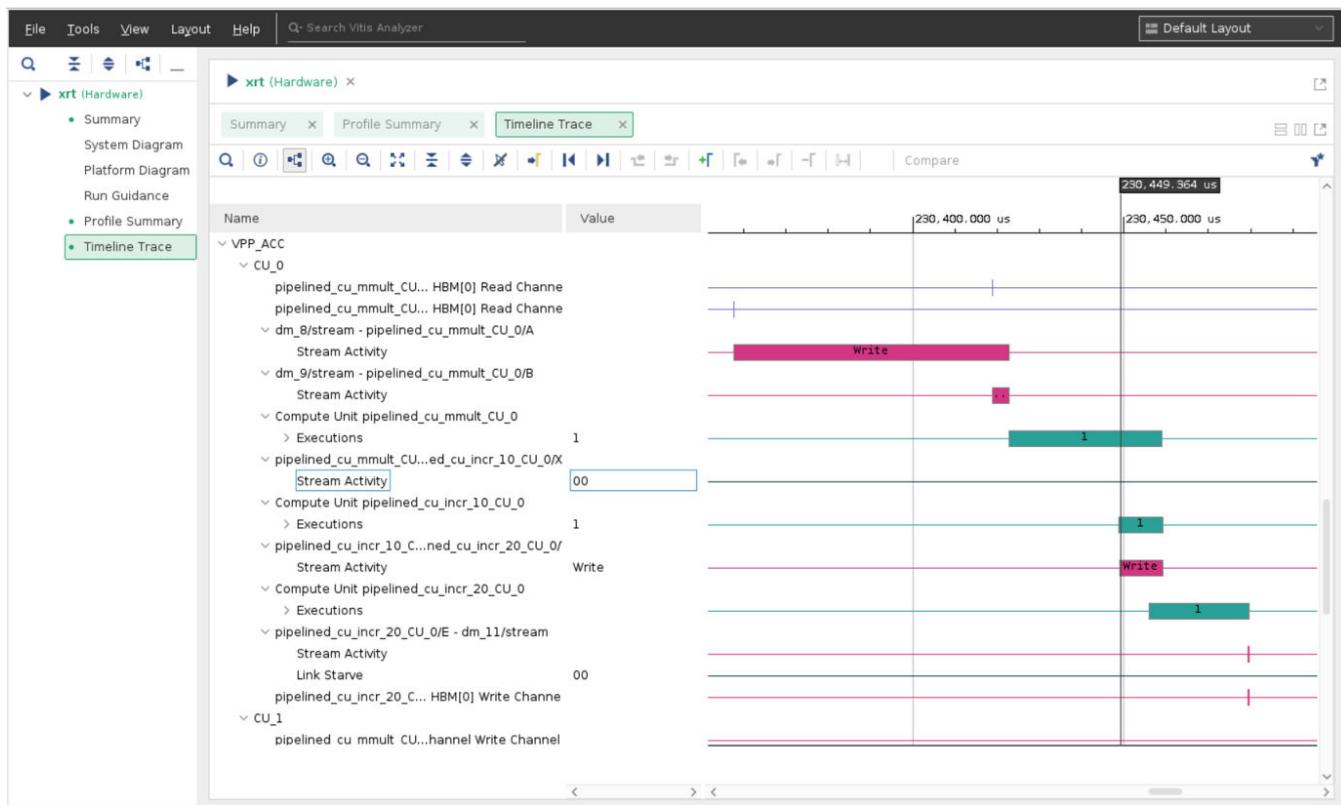
## Hardware Event Tracing

You can also profile hardware events within the accelerator system composition. Particularly, hardware events on the AXI4 ports of the platform connections and PE interfaces can be captured and presented as a timeline trace.

The picture below is for an example of an accelerator with two CUs, where each CU is a pipeline of three PEs in a chain: `mmult` → `incr_10` → `incr_20`. As seen in the picture, the rows are topologically organized to show the sequence of events following the order of the pipeline.

The picture show the execution of a single compute call. The first two rows shows two input argument being read from a HBM bank. The DATA\_COPY and SEQUENTIAL access macros allow a data movers (`dm_8` and `dm_9`) to stream this data to the PE `mmult` which then executes. The `mmult` then writes to a stream connected to the subsequent PE `incr_10`, which in turn writes to another stream to the PE `incr_20`. This PE finally triggers the output data mover streams which eventually write results back to the HBM bank.

**Figure 147: HW Event Trace**



Profiling in VSC mode can be enabled with these settings,

1. The accelerator class requires these macros,
  - a. `PROFILE_KERNEL("PE function names")`: To enable tracing the start and stop of every PE execution
  - b. `PROFILE_PORT("PE argument names")`: To enable profiling of AXI ports for any of the PE arguments
  - c. The keyword `all` can be used for profiling all PEs or all ports.

**Note:** Using `all` on accelerator with many AXI ports (every PE argument and platform port connection) can cause Vivado routing issues, preventing the design from closing timing. Then, it is recommended to created hardware traces on specific ports as required.
- d. Modifying these macros will trigger a full Vitis compile and linking including Vivado place and route.
2. Hardware traces can be enabled with this setting in the `xrt.ini` file:

```
[Debug]
device_trace=[fine|coarse]
```

## PROFILE\_KERNEL Examples

1. To enable profiling for all kernels:

```
PROFILE_KERNEL( "all" );
```

2. To enable profiling for all instances of *hls\_kernel* function:

```
PROFILE_KERNEL( "hls_kernel" );
```

3. To enable profiling for compute unit 0 and 2 of *hls\_df\_kernel* function:

```
PROFILE_KERNEL( "hls_df_kernel[0] hls_df_kernel[2]" );
```

## PROFILE\_PORT Examples

1. To enable profiling for all ports of all kernels:

```
PROFILE_PORT( "all" );
```

2. To enable profiling for all ports of *hls\_kernel* function on compute unit 0 and 2:

```
PROFILE_PORT( "hls_kernel[0]/all hls_kernel[2]/all" );
```

3. To enable profiling for all ports of all instances of *hls\_df\_kernel*:

```
PROFILE_PORT( "hls_df_kernel/all" );
```

4. To enable profiling on port A in *hls\_kernel* for compute unit 0 and port C in *hls\_kernel* for compute unit 2:

```
PROFILE_PORT( "hls_kernel[0]/A hls_kernel[2]/C" );
```

5. To enable profiling on port A for all instances of *hls\_kernel* and port C for all instances of *hls\_kernel*:

```
PROFILE_PORT( "hls_kernel/A hls_kernel/C" );
```

## Hardware Event Tracing Issues and Solutions

Hardware events are captured in the device FIFO or global memory buffers and sent back (offloaded) to the CPU to generate the timeline traces. Hardware trace events can be dropped in different scenarios as explained in the table.

*Table 70: Hardware Trace Event Scenarios*

Issue	Solution
Once the FIFO or DDR/HBM buffer gets full, all subsequent hardware trace events will be dropped.	Look for a runtime warning and rerun with a larger buffer size, using this setting in <code>xrt.ini</code> <code>trace_buffer_size=&lt;size&gt;</code> , where size must fit within the global memory limits.

Table 70: Hardware Trace Event Scenarios (cont'd)

Issue	Solution
Using the same global memory for compute data transfers can cause congestion leading to dropped hardware trace events.	Use a resource for offloading that is different from what the <code>compute()</code> IO uses. The macro <code>PROFILE_OFFLOAD</code> can be used to specify the profiling offload method in the accelerator class. <code>PROFILE_OFFLOAD("FIFO"   "DDR[0-3]"   "HBM[0-31]");</code> <b>TIP:</b> This cannot be used when targeting compilation for <code>hw_emu</code> , or it will cause a profiling logic insertion error. It will also be ignored for the <code>sw_emu</code> target.
The application testing created too many hardware events.	Lower the number of hardware events generated with any of these setting in the <code>xrt.ini</code> file, <ol style="list-style-type: none"> <li>1. <code>device_trace=coarse</code>: generates the fewest hardware events</li> <li>2. <code>device_trace=fine</code>: generates more hardware events and greater detail</li> <li>3. <code>stall_trace=true</code>: generates the most hardware events and can quickly fill up any size buffer. Do not use it.</li> </ol>

# Supported Platforms and Startup Examples

## Platforms

While VSC mode should work on most of the Vitis supported platforms. Here is the list of platforms in production that VSC mode is tested on,

- `xilinx_u200_gen3x16_xdma_2_202110_1`
- `xilinx_u55c_gen3x16_xdma_3_202210_1`
- `xilinx_u50_gen3x4_xdma_2_202010_1`
- `xilinx_u50_gen3x16_xdma_5_202210_1`
- `xilinx_u25_gen3x8_xdma_1_202010_1`
- `xilinx_u250_gen3x16_xdma_3_1_202020_1`
- `xilinx_u2_gen3x4_xdma_flat_gc_2_202110_1`
- `xilinx_u2_gen3x4_xdma_gc_2_202110_1`

The following are not supported in this release

- All NoDMA Platforms such as `xilinx_u50_gen3x16_nodma_1_202110_1`
- Embedded platforms such as those using ZyncMPSoC

With the following specific platforms, hardware emulation is not supported and running emulation will lead to an error.

- `xilinx_u25_gen3x8_xdma_1_202010_1`
- `xilinx_u250_gen3x16_xdma_2_1_202010_1`

However, validation with software emulation and profiling on the physical card should work as expected. The newer versions of these platforms will support emulation with VSC.

## Start-Up Examples

The following examples cover some of the popular design features supported by VSC. They are published on GitHub and the table provides the links to each of those.

**Table 71: Popular Design Features Supported by VSC**

Example	Name	Feature Coverage
Convolution Filter	<a href="#">quick_start_sc</a>	This example is described in detail the quick-start section. It covers some basic VSC features such as multiple CUs.
Hardware Emulation	<a href="#">debug_profile_sc</a>	This example covers validation features: 1. hardware emulation. 2. Profiling the accelerator execution on the card.
CU-to-GMIO transfer types	<a href="#">gmio_transfers_sc</a>	Global memory IO transfers.
File transfers to CU	<a href="#">file_filter_sc</a>	This examples covers multiple features of specialized IO transfers: 1. P2P and H2C file transfer using ping-pong buffers. Refer to <a href="#">Special Data Transfer Models</a> for more details. 2. Create and use multi-file buffer objects that can work with multi-CU computation. Refer to <a href="#">Special Data Transfer Models</a> for more details. 3. User-defined custom synchronization of input and output buffers to ACC.
Free-running PEs with AXI4-Streams	<a href="#">streaming_sc</a>	ACC containing free-running processing elements with feedback AXI4-Stream connections. Showcase stream depth. Software emulation.
Multiple ACCs with CPU	<a href="#">mult_acc_compose_sc</a>	Allow multiple accelerators (VPP_ACC) in one xclbin, and compose them into a pipeline, with or without CPU processing in-between the PEs.
Multi-card accelerators	<a href="#">mult_card_sc</a>	Application code controlling a multi-card multiple accelerator design.

## Vitis C-Library Examples

The following Vitis C-library APIs are some design examples using the VSC mode.

**Table 72: Design Examples Using the VSC Mode**

API	Platform	API Level
regexEngine	U50	L3
U.2 Gunzip + CSV	U.2	L3
GeoSpatial-Join	U200	L3
GeoSpatial-KNN	U50	L2
Ehash	U55N	L2
CRC32C	U.2/U50/U200	L3

# Additional Resources and Legal Notices

## Xilinx Resources

For support resources such as Answers, Documentation, Downloads, and Forums, see [Xilinx Support](#).

## Documentation Navigator and Design Hubs

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

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

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

- In DocNav, click the **Design Hubs View** tab.
- On the Xilinx website, see the [Design Hubs](#) page.

**Note:** For more information on DocNav, see the [Documentation Navigator](#) page on the Xilinx website.

# Revision History

## Getting Started with Vitis Revision History

The following table shows the revision history for [Section I: Getting Started with Vitis](#).

Section	Revision Summary
<b>12/7/2022 Version 2022.2</b>	
No Changes.	
<b>10/21/2022 Version 2022.2</b>	
<a href="#">Installation Requirements</a>	Updated OS support information.
<b>10/19/2022 Version 2022.2</b>	
<a href="#">Section II: Introduction to Vitis Flows</a>	Added details on elements of Vitis software platform.
<a href="#">Introduction to Data Center Acceleration for Software Programmers</a>	Revised the Terminology section.
<a href="#">Installation</a>	Revised the section on Installing Embedded Platforms.

## Introduction to Vitis Flows Revision History

The following table shows the revision history for [Section II: Introduction to Vitis Flows](#).

Section	Revision Summary
<b>12/7/2022 Version 2022.2</b>	
No changes.	
<b>10/21/2022 Version 2022.2</b>	
No changes.	
<b>10/19/2022 Version 2022.2</b>	
<a href="#">Introduction to Vitis Tools for Embedded System Designers</a>	Updated the section on Vitis PL Kernel Development Flow.
<a href="#">Introduction to Data Center Acceleration for Software Programmers</a>	Updated the section on Vitis Application Development Flow and Best Practices for Host Programming.
<a href="#">Tutorials and Examples</a>	Updated the section.
<a href="#">Versal AI Engine Graph Development</a>	Updated the section.
<a href="#">Introduction to Data Center Acceleration for RTL Designers</a>	Updated the section.

## Developing Applications Revision History

The following table shows the revision history for [Section III: Developing Applications](#).

Section	Revision Summary
<b>12/7/2022 Version 2022.2</b>	
<a href="#">Best Practices for Data Center Acceleration with Vitis</a>	Moved to <a href="#">Section XI: Additional Information</a> section, <a href="#">Methodology for Accelerating Data Center Applications with the Vitis Software Platform</a>

Section	Revision Summary
<b>10/21/2022 Version 2022.2</b>	
No changes.	
<b>10/19/2022 Version 2022.2</b>	
Design Topology	Updated the section.
Writing the Software Application	Updated the section on Enabling Auto-Restart of Kernels and Developing PL Kernels using C++.
Packaging RTL Kernels	Updated the section.
C++ Kernel Development	Moved to the <i>Vitis High-Level Synthesis User Guide</i> ( <a href="#">UG1399</a> )

## Building and Running the Application Revision History

The following table shows the revision history for [Section IV: Building and Running the Application](#).

Section	Revision Summary
<b>12/7/2022 Version 2022.2</b>	
Building and Running the Application.	Renamed to <a href="#">Running the Application</a>
Additional Memory Mapping Techniques for Alveo Accelerator Cards	Cleaned up the topic and deleted duplicate content.
Associating an ELF file with MicroBlaze	Added topic.
<b>10/21/2022 Version 2022.2</b>	
No changes.	
<b>10/19/2022 Version 2022.2</b>	
Building the Device Binary	Updated the section.
Packaging for Data Center Platforms	Updated the section.
Build Targets	Updated the section.
Packaging the System	Updated the section on Packaging for Embedded Platforms.

## Simulating the Application with the Emulation Flow Revision History

The following table shows the revision history for [Section V: Simulating the Application with the Emulation Flow](#).

Section	Revision Summary
<b>12/7/2022 Version 2022.2</b>	
No changes.	
<b>10/21/2022 Version 2022.2</b>	
No changes.	
<b>10/19/2022 Version 2022.2</b>	
Working with I/O Traffic Generators	Updated the section.
Working with Functional Model of the HLS Kernel	Updated the section.
Running Emulation on an Embedded Processor Platform	Updated the section.
Data Center vs. Embedded Platforms	Updated the section.

Section	Revision Summary
<a href="#">Working with Simulators in Hardware Emulation</a>	Updated the section.

## Profiling, Optimizing, and Debugging the Application Revision History

The following table shows the revision history for [Section VI: Profiling and Debugging the Application](#).

Section	Revision Summary
	<b>12/7/2022 Version 2022.2</b>
No changes.	
	<b>10/21/2022 Version 2022.2</b>
No changes.	
	<b>10/19/2022 Version 2022.2</b>
<a href="#">Generating and Opening the Waveform Reports</a>	Updated Running Emulation link.
<a href="#">Detailed Kernel Trace</a>	Updated the sections on Using Burst Data Transfers and Using Full AXI Data Width.

## Vitis Commands and Utilities Revision History

The following table shows the revision history for [Section VII: Vitis Commands and Utilities](#).

Section	Revision Summary
	<b>12/7/2022 Version 2022.2</b>
No changes.	
	<b>10/21/2022 Version 2022.2</b>
No changes.	
	<b>10/19/2022 Version 2022.2</b>
<a href="#">Section VII: Vitis Commands and Utilities</a>	Updated the v++ Command section.

## Using the Vitis Analyzer Revision History

The following table shows the revision history for [Section VIII: Using the Vitis Analyzer](#).

Section	Revision Summary
	<b>12/7/2022 Version 2022.2</b>
No changes.	
	<b>10/21/2022 Version 2022.2</b>
No changes.	
	<b>10/19/2022 Version 2022.2</b>
<a href="#">Working with Summary Reports</a>	Updated the section.

## Using the Vitis IDE Revision History

The following table shows the revision history for [Section IX: Using the Vitis IDE](#).

Section	Revision Summary
<b>12/7/2022 Version 2022.2</b>	
No changes.	
<b>10/21/2022 Version 2022.2</b>	
No changes.	
<b>10/19/2022 Version 2022.2</b>	
G++ Host Compiler and Linker Settings	Updated the section.

## Using Vitis Embedded Platforms Revision History

The following table shows the revision history for [Section X: Using Vitis Embedded Platforms](#).

Section	Revision Summary
<b>12/7/2022 Version 2022.2</b>	
No changes.	
<b>10/21/2022 Version 2022.2</b>	
No changes.	
<b>10/19/2022 Version 2022.2</b>	
No changes.	

## Additional Information

The following table shows the revision history for [Section XI: Additional Information](#).

Section	Revision Summary
<b>12/7/2022 Version 2022.2</b>	
Best Practices for Data Center Acceleration with Vitis	Moved to <a href="#">Section XI: Additional Information</a> section, <a href="#">Methodology for Accelerating Data Center Applications with the Vitis Software Platform</a>
<b>10/21/2022 Version 2022.2</b>	
No changes.	
<b>10/19/2022 Version 2022.2</b>	
Methodology for Developing C/C++ Kernels	Updated the section on Verification Considerations steps.
Methodology for Accelerating Data Center Applications with the Vitis Software Platform	Updated the section.
Using Vitis System Compilation Mode	Updated the section.

# Please Read: Important Legal Notices

The information disclosed to you hereunder (the "Materials") is provided solely for the selection and use of Xilinx products. To the maximum extent permitted by applicable law: (1) Materials are made available "AS IS" and with all faults, Xilinx hereby DISCLAIMS ALL WARRANTIES AND CONDITIONS, EXPRESS, IMPLIED, OR STATUTORY, INCLUDING BUT NOT LIMITED TO WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, OR FITNESS FOR ANY PARTICULAR PURPOSE; and (2) Xilinx shall not be liable (whether in contract or tort, including negligence, or under any other theory of liability) for any loss or damage of any kind or nature related to, arising under, or in connection with, the Materials (including your use of the Materials), including for any direct, indirect, special, incidental, or consequential loss or damage (including loss of data, profits, goodwill, or any type of loss or damage suffered as a result of any action brought by a third party) even if such damage or loss was reasonably foreseeable or Xilinx had been advised of the possibility of the same. Xilinx assumes no obligation to correct any errors contained in the Materials or to notify you of updates to the Materials or to product specifications. You may not reproduce, modify, distribute, or publicly display the Materials without prior written consent. Certain products are subject to the terms and conditions of Xilinx's limited warranty, please refer to Xilinx's Terms of Sale which can be viewed at <https://www.xilinx.com/legal.htm#tos>; IP cores may be subject to warranty and support terms contained in a license issued to you by Xilinx. Xilinx products are not designed or intended to be fail-safe or for use in any application requiring fail-safe performance; you assume sole risk and liability for use of Xilinx products in such critical applications, please refer to Xilinx's Terms of Sale which can be viewed at <https://www.xilinx.com/legal.htm#tos>.

## AUTOMOTIVE APPLICATIONS DISCLAIMER

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

## Copyright

© Copyright 2019-2022 Advanced Micro Devices, Inc. Xilinx, the Xilinx logo, Alveo, Artix, Kintex, Kria, Spartan, Versal, Vitis, Virtex, Vivado, Zynq, and other designated brands included herein are trademarks of Xilinx in the United States and other countries. AMBA, AMBA Designer, Arm, ARM1176JZ-S, CoreSight, Cortex, PrimeCell, Mali, and MPCore are trademarks of Arm Limited in the EU and other countries. OpenCL and the OpenCL logo are trademarks of Apple Inc. used by permission by Khronos. PCI, PCIe, and PCI Express are trademarks of PCI-SIG and used under license. All other trademarks are the property of their respective owners.