



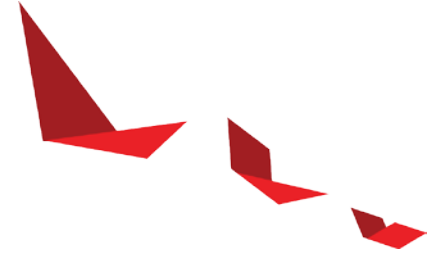
Vitis tool flow

Introduction to Vitis





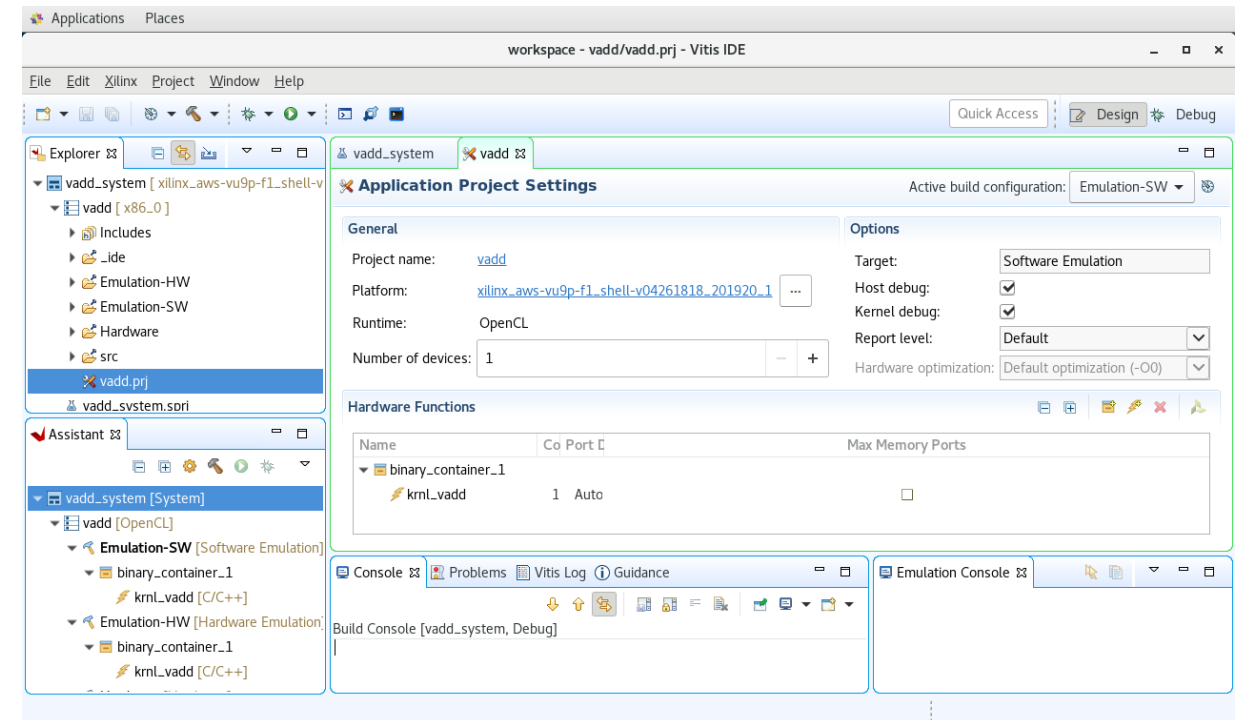
Goal



- ▶ Understand the Vitis tool flow
 - How a design is compiled to a software application and hardware kernel
 - How to boost application performance
 - How to debug and verify your design
 - What the Vitis Analyzer is for
 - What other help is available

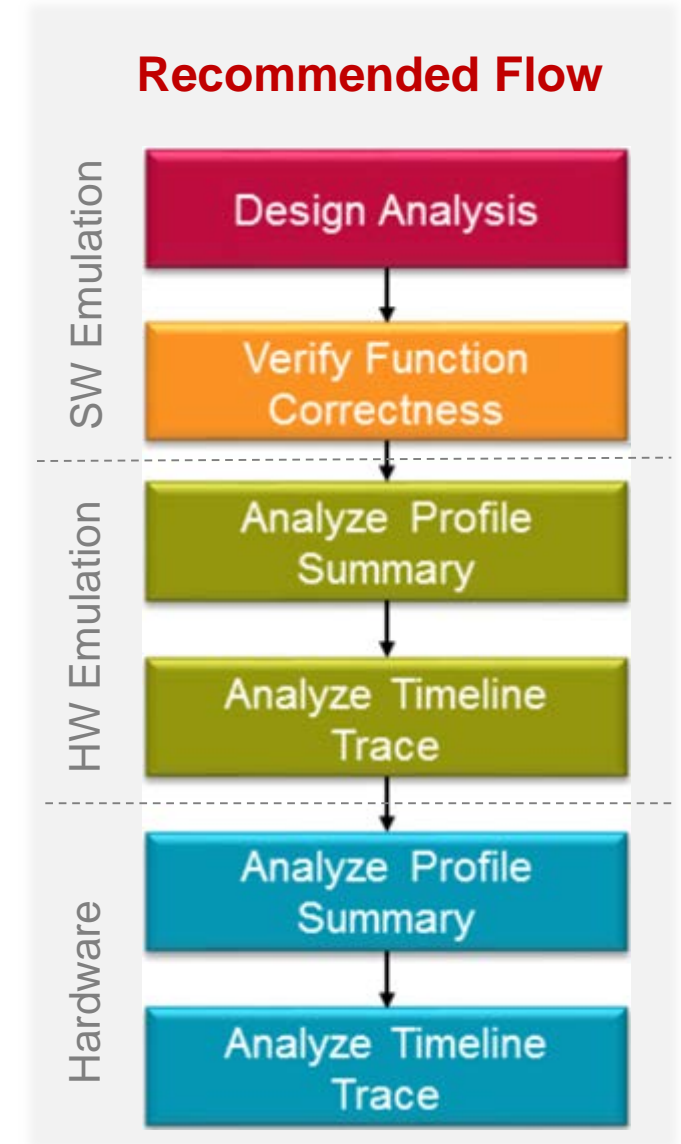
Vitis Development Environment

- ▶ Eclipse-based IDE
- ▶ Develop, profile and deploy Xilinx accelerated applications
- ▶ Concurrent programming of host application and hardware kernels
- ▶ Automatic build and execution flows
- ▶ Built-in debug, profiling and performance analysis tools



Vitis flow

- ▶ Three modes of operation
 - Software Emulation
 - Hardware Emulation
 - Run on Hardware
- ▶ Compilation time increases for each level
- ▶ Emulation is faster, but is less detailed
 - Good for verifying functional correctness
- ▶ Final performance evaluation needs to be done in real hardware



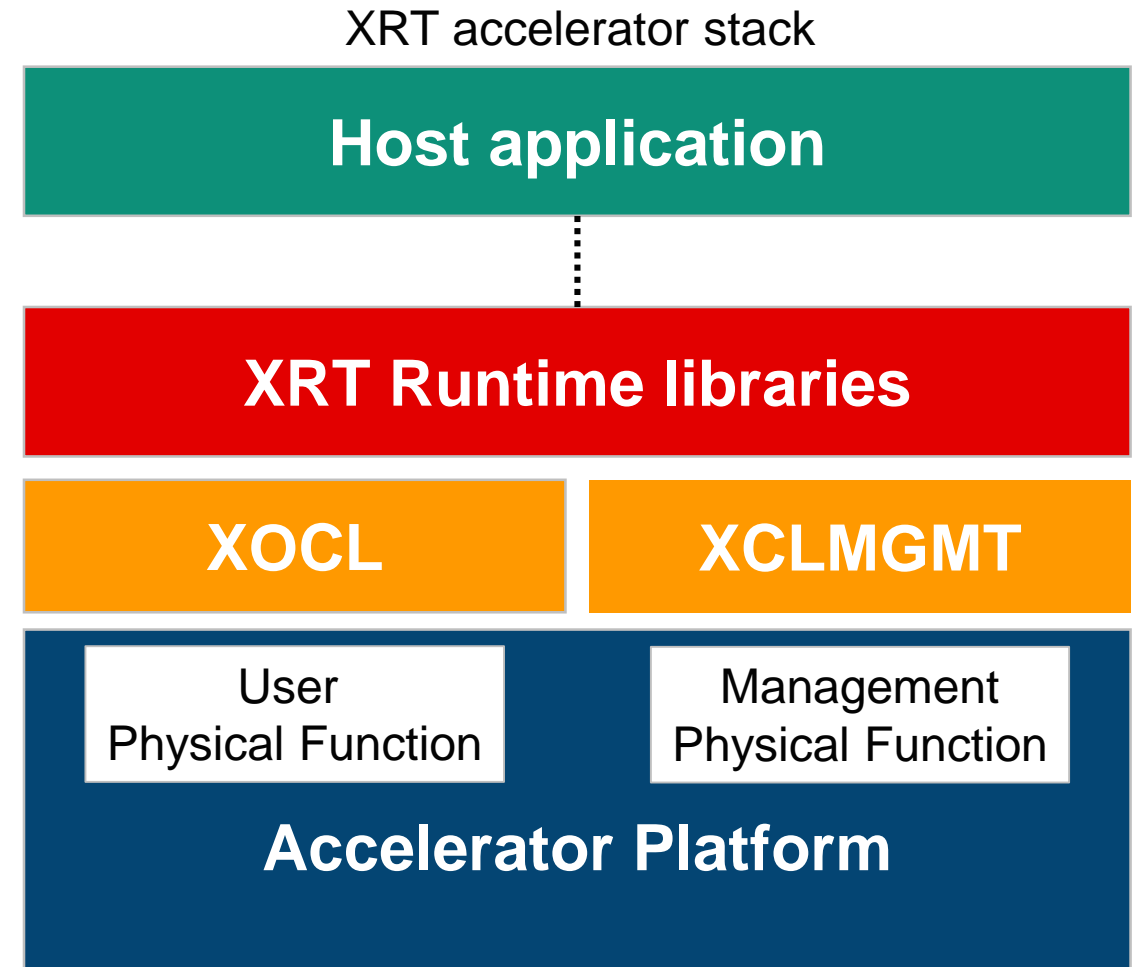


Testing and Execution Modes

Software Emulation	Hardware Emulation	Run on Hardware
Host application runs with a C/C++ or OpenCL model of the hardware Kernels	Host application runs with a simulated RTL model of the hardware Kernels	Host application runs with actual FPGA implementation of the hardware Kernels
Confirm functional correctness of the system	Test the host / kernel integration, get performance estimates	Confirm system runs correctly and with desired performance
Fastest turnaround time	Best debug capabilities	Accurate performance results

Host Framework and Execution Stack

- ▶ XRT runtime manages communication with the hardware or emulator
- ▶ User Interface
 - OpenCL popular in parallel computing
 - C/C++ API, Python also supported
- ▶ OpenCL
 - Host application submits work to FPGA kernels using standard OpenCL API
 - OpenCL runtime to manage kernel scheduling during execution



<https://xilinx.github.io/XRT>

Vitis environment

Vitis IDE

The screenshot displays the Vitis IDE interface with the following components:

- Explorer:** Located on the left, it shows a project tree for `test_gui_flow_system` and `test_gui_flow`. The `test_gui_flow.prj` file is selected and highlighted.
- Project Editor:** The central area displays the `Application Project Settings` for the active build configuration `Emulation-HW`. It includes sections for **General** (Project name, Platform, Runtime, Number of devices) and **Options** (Target, Host debug, Kernel debug, Report level, Hardware optimization). Below these is the **Hardware Functions** table.
- Assistant:** Located at the bottom left, it shows a list of system components for `test_gui_flow_system`, including `test_gui_flow` and its sub-components like `Emulation-SW`, `Emulation-HW`, and `Hardware`.
- Console:** Located at the bottom right, it displays the execution log. The log shows the process of reading input data, executing the kernel, and verifying the final results, concluding with `Test Successful` and `DONE`.

Name	Compute Units	Port Data Width	Max Memory Ports
binary_container_1			
K_VADD	1	Auto	

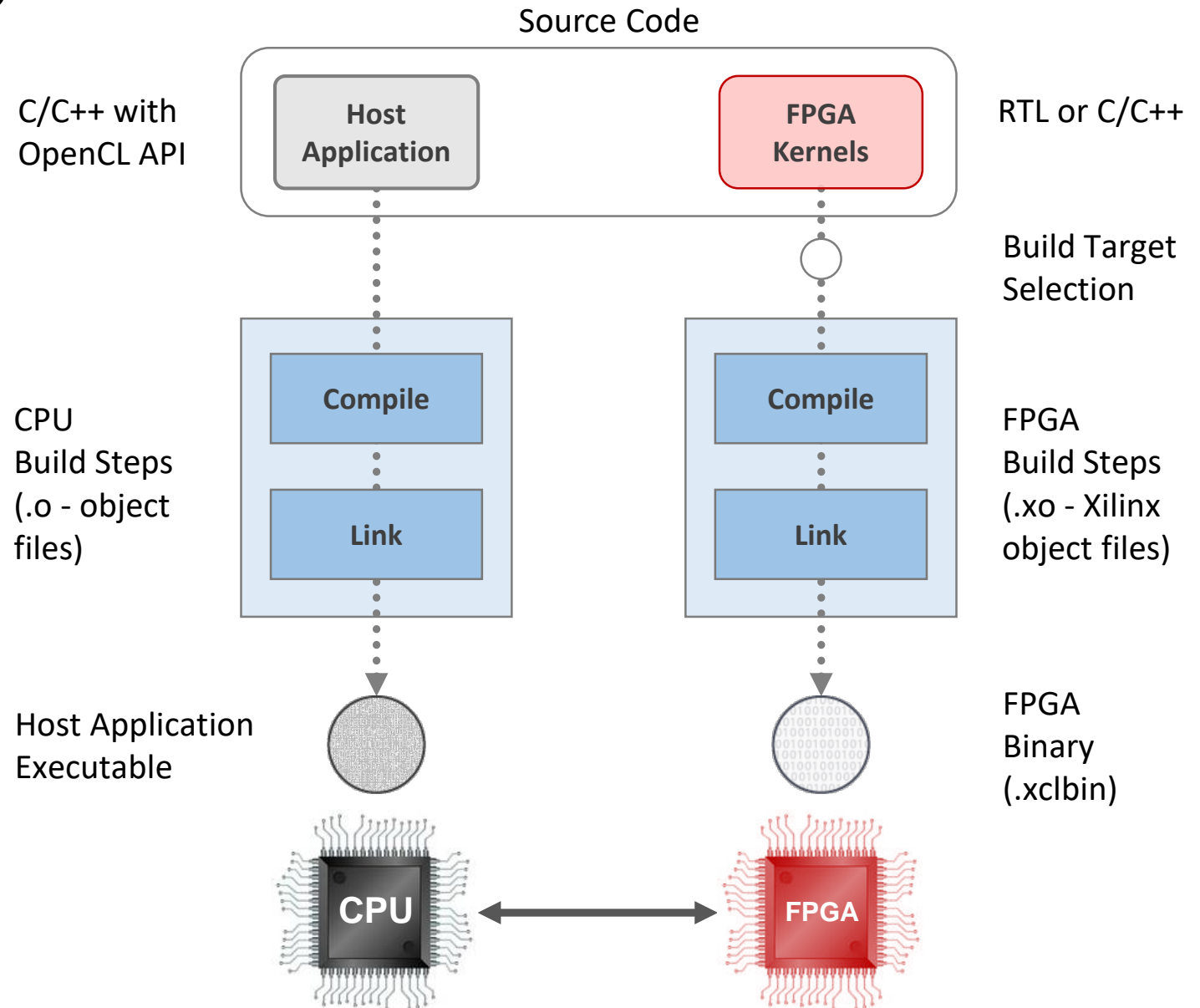
```
<terminated> Emulation-SW, test_gui_flow-Default, test_gui_flow (06/11/19, 2:23 PM)
HOST-Info: Reading input data from the ../src/data_2.txt file ... Read 1024 values
HOST-Info: Executing Kernel ...
HOST-Info: The Output Result file: RES.txt

Host-Info: =====
Host-Info: Verifying final results (only failed tests are printed)
Host-Info: =====
Host-Info: Test Successful

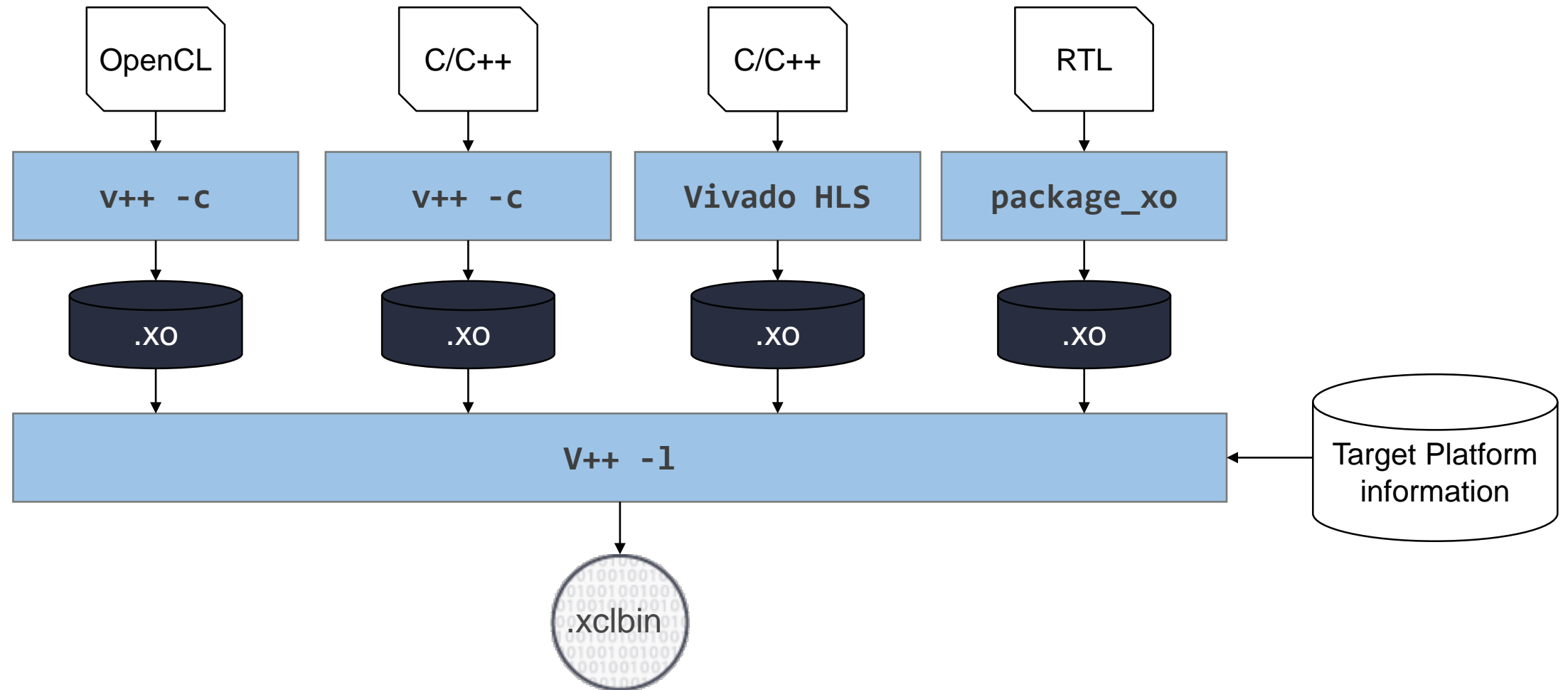
HOST-Info: DONE
```

Building the Host Code

- ▶ Vitis builds host code and kernel code
- ▶ Host code
 - Written in C/C++, using OpenCL
 - Build using g++
 - Include XRT libraries
- ▶ Hardware kernel code
 - Written in C/C++/OpenCL or RTL
 - Built using the Vitis compiler v++



Building the hardware Kernel – FPGA Binary



Debugging

Debugger	Source	Software Emulation	Hardware Emulation	System Run
GNU debugging (GDB) tool	Host code	✓	✓	✓
	Kernel code	✓	Limited support	✗

▶ Waveform-based kernel debugging

- Hardware designers use this approach

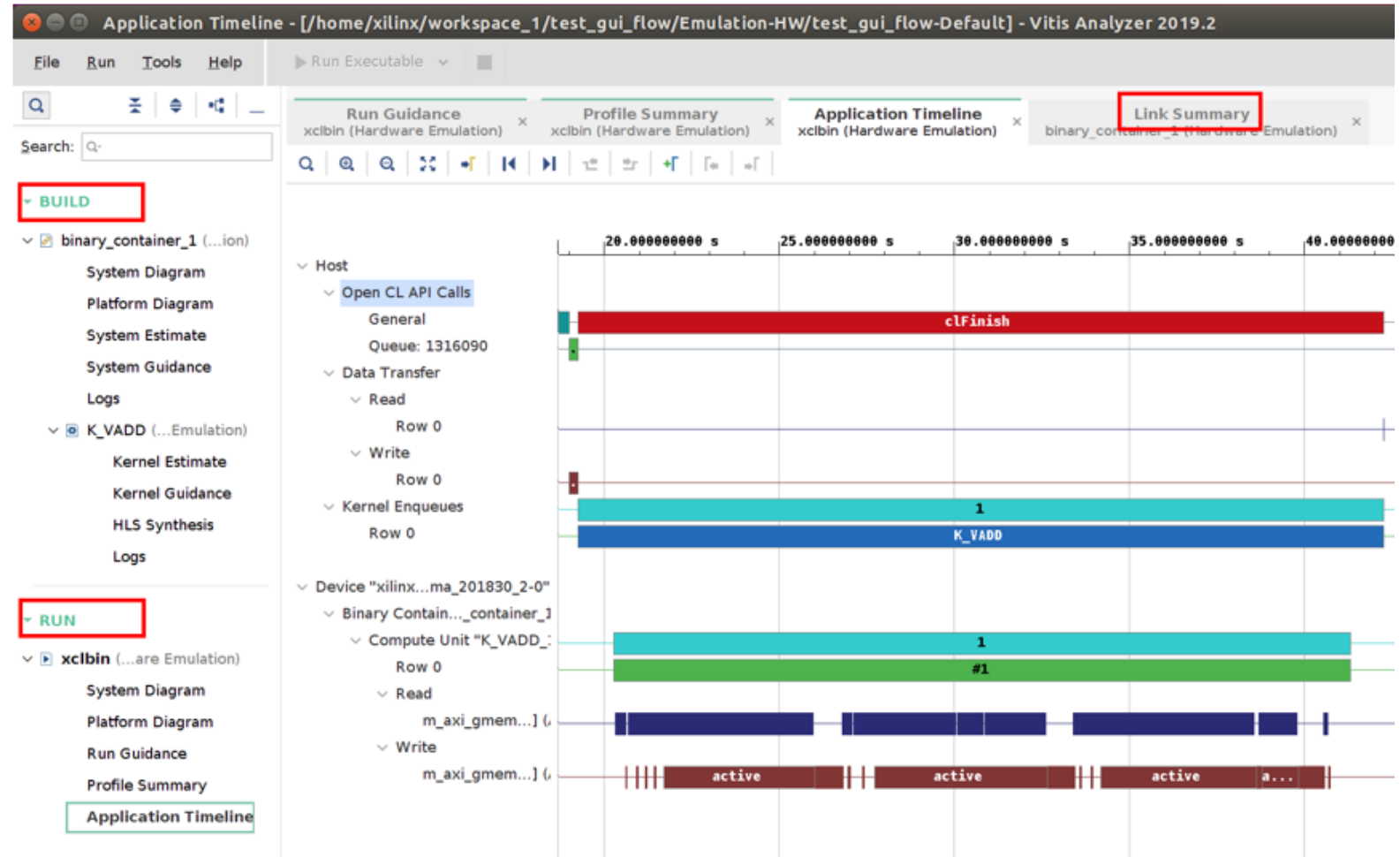
▶ `printf()` or `cout` can be used

- C/C++-based kernels
 - Only supported during software emulation
 - Excluded from Vivado HLS synthesis
- OpenCL kernels
 - Supports `printf()` for all build configurations: SW Emulation, HW Emulation, and system run

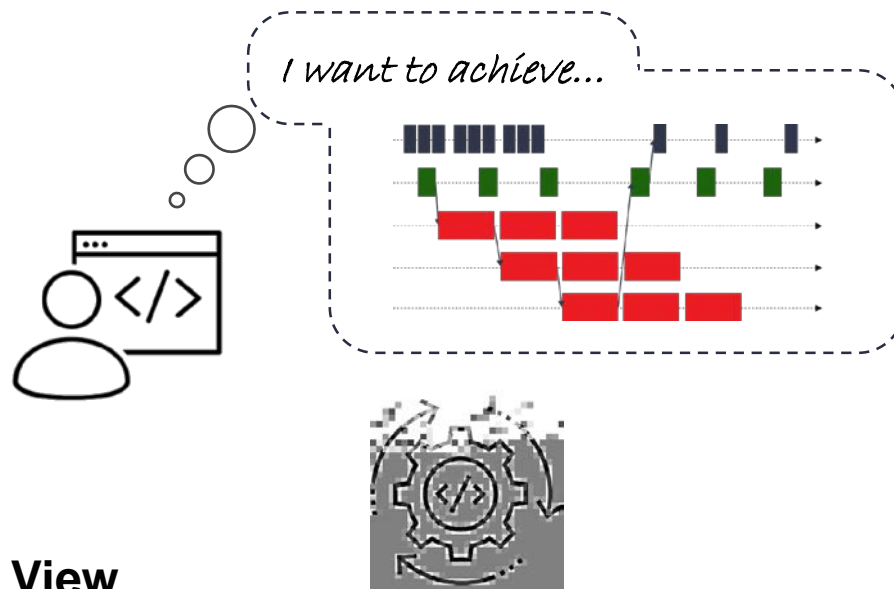
```
#ifndef __SYNTHESIS__
    printf("Checkpoint 1 reached");
#endif
```

Profiling: Vitis Analyzer

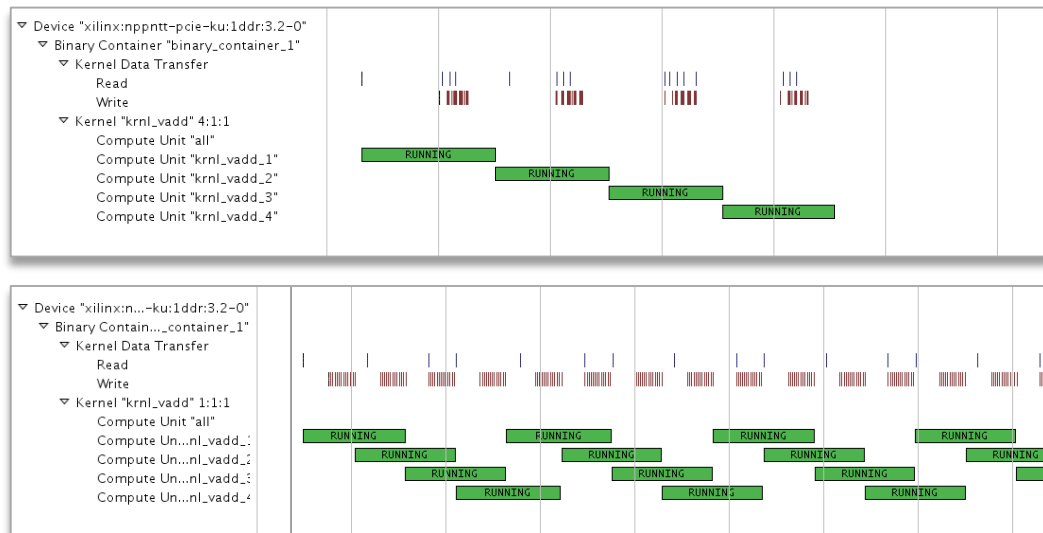
- ▶ Vitis analyzer
 - View and analyze the reports
- ▶ Useful reports
 - Compile Summary
 - Link Summary
 - Run Summary



Vitis Platform Supports and Guides Optimization Process



Application Timeline View



Design Guidance

Name	Threshold	Actual	Details	Resolution
Host Data Transfer (4)				
HOST_MIGRATE_MEM (1)	> 0			
✓ HOST_MIGRATE_MEM	> 0	3	Migrate Memory OpenCL APIs were used 3 time(s).	
HOST_READ_TRANSFER_SIZE (1)	> 4.096			
✓ HOST_READ_TRANSFER_SIZE	> 4.096	4.096	Host read average size was 4.096 KB across 1 transfers.	
HOST_WRITE_TRANSFER_SIZE (1)	> 4.096			
✓ HOST_WRITE_TRANSFER_SIZE	> 4.096	4.096	Host write average size was 4.096 KB across 2 transfers.	
P2P_HOST_TRANSFERS (1)	= 0			
✓ P2P_HOST_TRANSFERS	= 0	0	Host performed 0 transfers from peer to peer buffer.	
Kernel Data Transfer (11)				
KERNEL_PORT_DATA_WIDTH (1)	= 512			
⚠ KERNEL_PORT_DATA_WIDTH	= 512	32	Port K_VADD_1/m_axi_gmem has a data width of 32. Utilize the entire memory data width. Click here .	
KERNEL_READ_TRANSFER_AMOUNT_MAX (1)	< 2.000			
✓ KERNEL_READ_TRANSFER_AMOUNT_MAX	< 2.000	1.000	Total kernel read of 0.008192 MB on xilinx_u200_xdma_201830_2-0 was 1.000 MB.	
KERNEL_READ_TRANSFER_AMOUNT_MIN (1)	> 0.250			
✓ KERNEL_READ_TRANSFER_AMOUNT_MIN	> 0.250	1.000	Total kernel read of 0.008192 MB on xilinx_u200_xdma_201830_2-0 was 1.000 MB.	
KERNEL_READ_TRANSFER_SIZE (1)	> 0.512			

Resolution
The specific port, that is not scalar based, is not utilizing the optimum data width (512-bit). Kernel arguments are implemented through memory mapped AXI ports. For detailed explanation and recommendation: [KERNEL_PORT_DATA_WIDTH](#)



Thank You

