SDAccel Profiling and Optimization Guide

UG1207 (v2018.2) June 6, 2018

ATTENTION! SDAccel Development Environment 2018.2 XDF users: Click here to view the 2018.2.xdf version of this guide.

Revision History

The following table shows the revision history for this document.

Section	Revision Summary
06/06/2018 Version 2018.2	
	This document has gone through an extensive reorganization and rewrite. Some background material has been removed to focus on profiling and optimization.
Guidance	Discusses the use of the Guidance view for improving project performance.
Waveform View and Live Waveform Viewer	Discusses the use of the Waveform viewer with the SDAccel™ environment.
Utilizing Implementation Tools	Added a discussion of Vivado Backend Optimization.
Interface Optimization	Using Full AXI Data Width
Optimizing Computational Parallelism	Loop Parallelism and Task Parallelism
	Optimizing Computational Units
	Optimizing Memory Architecture
04/04/2018 Version 2018.1	
Entire document	Minor editorial edits for 2018.1
Command Line	Change to profile_kernel option.
Assigning DDR Bank for Kernel Code	RTL Kernel Wizard kernel naming convention.

Table of Contents

Introduction

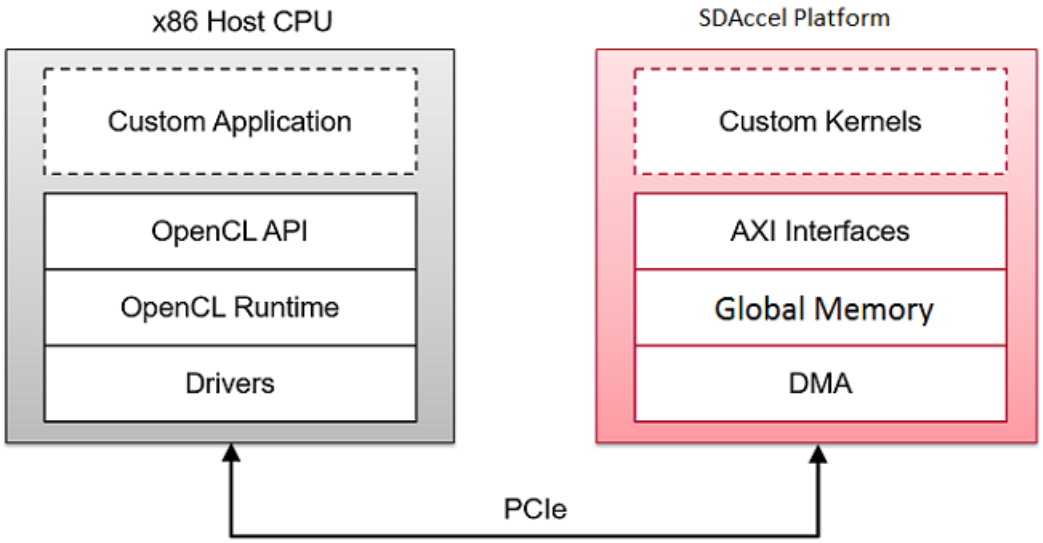
This book presents all SDx™ features related to performance analysis of the design. It is also logically structured to assist in the actual performance improvement effort. Dedicated sections are available for the main components of SDAccel™ performance bottlenecks, namely Accelerator, PCIe® bus transfer and Host code. Each of these sections is structured to guide the developer from recognizing bottlenecks all the way to solution approaches to increase overall system performance.

Note: Performance optimization assumes, as a starting point, a working design intended for performance improvement. If erroneous behavior is encountered, look for guidance in *SDAccel Environment Debugging Guide* (UG1281). Similarly, the general concepts regarding coding of host code or of accelerator kernels are not explained here, these are introduced in the *SDAccel Environment Programmers Guide* (UG1277).

Execution Model of an SDAccel Application

The SDAccel™ environment is designed to provide a simplified development experience for FPGA-based software acceleration platforms. The general structure of the SDAccel acceleration platform is shown in the following figure.

Figure 1: **Architecture of an SDAccel Application**



The custom application is running on the host x86 server and uses OpenCL™ API calls to interact with the FPGA accelerators. The SDAccel run time manages those interactions. The application is written in C/C++ using OpenCL APIs. The custom kernels are running within a Xilinx® FPGA through the SDAccel run time that manages interactions between the host application and the accelerator. Communication between the host x86 machine and the SDAccel accelerator board occurs across the PCIe® bus.

The SDAccel hardware platform contains global memory banks. The data transfer from the host machine to kernels and from kernels to the host happens through these global memory banks. The kernels running on the FPGA can have one or more memory interfaces. The connection from the memory banks to those memory interfaces are programmable and determined by linking options of the compiler.

The SDAccel execution model follows these steps:

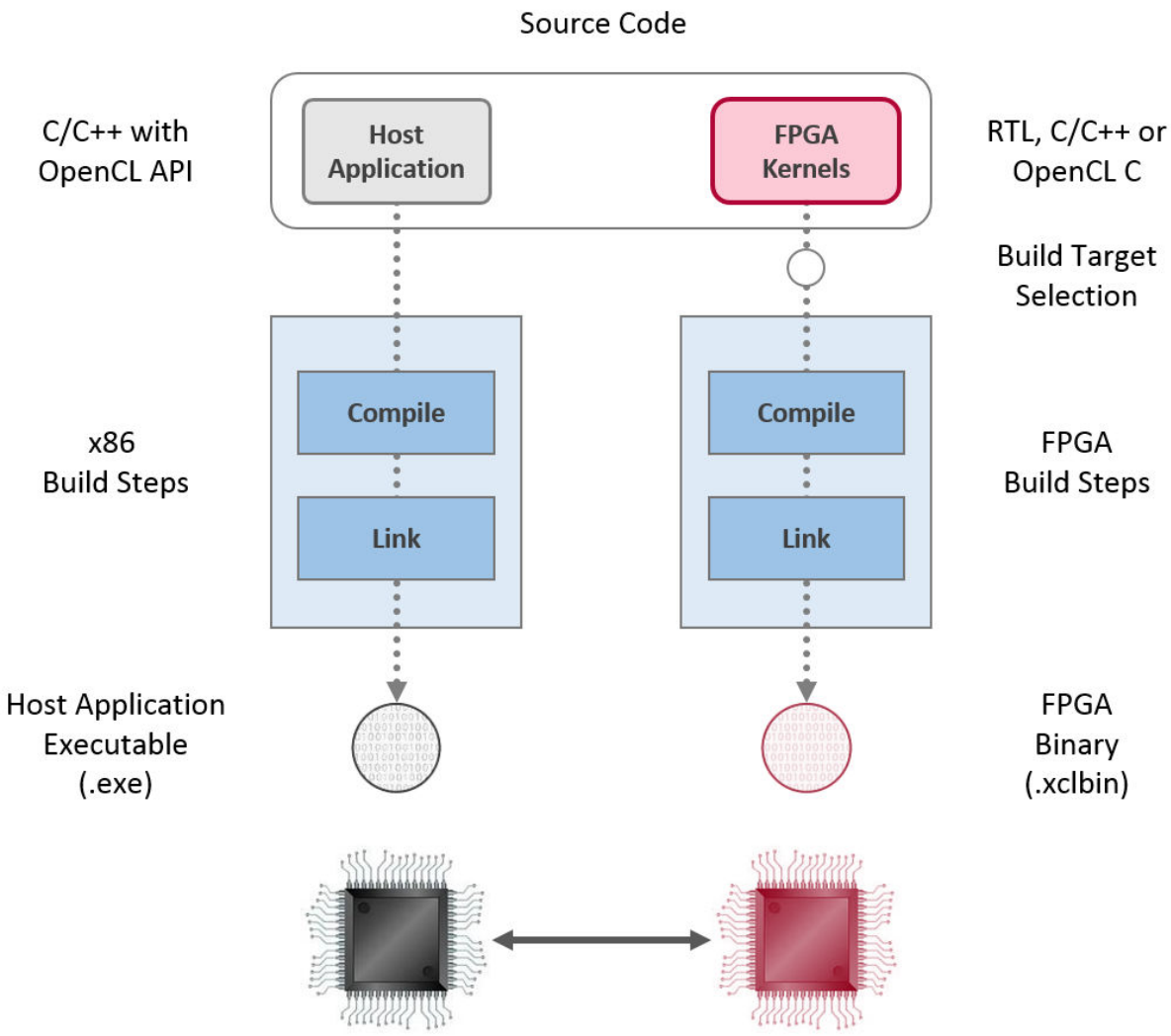
1. The host application writes the data needed by a kernel into the global memory of the SDAccel hardware platform, through the PCIe interface.
2. The host programs the kernel with its input parameters.
3. The host application triggers the execution of the kernel function on the FPGA.
4. The kernel performs the required computation, reading and writing data from global memory as necessary.
5. Kernels write data back to the memory banks, and notify the host that it has completed its task.
6. The host application reads data back from global memory into the host memory space, and continues processing as needed.

The FPGA can accommodate multiple kernel instances at one time; either different types of kernels or multiple instances of the same kernel. The SDAccel OpenCL run time transparently orchestrates the communication between the host application and the kernels in the FPGAs. The number of instances of a kernel is variable and determined by the host program and compilation options.

SDAccel Build Process

The SDAccel™ environment offers all of the features of a standard software development environment: an optimized compiler for host applications, cross-compilers for the FPGA, robust debugging environment to help you identify and resolve issues in the code, and performance profilers to let you identify the bottlenecks and optimize your code. Within this environment the SDAccel build process uses a standard compilation and linking process for both the software elements and the hardware elements of the project. As shown in the image below, the host application is built through one process using standard GCC, and the FPGA binary is built through a separate process using the Xilinx® XOCC compiler.

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



1. Host application build process using GCC:

- Each host application source file is compiled to an object file (`.o`).
- The object files (`.o`) are linked with the Xilinx SDAccel runtime shared library to create the executable (`.exe`).

2. FPGA build process using XOCC:
   - Each kernel is independently compiled to a Xilinx object (`.xo`) file.
   - ◦ C/C++ and OpenCL C kernels are compiled for implementation on an FPGA using the XOCC compiler. This step leverages the Vivado® HLS compiler. The same pragmas and attributes supported by Vivado HLS can be used in C/C++ and OpenCL C kernel source code to specify the desired kernel micro-architecture and control the result of the compilation process.
     ◦ RTL kernels are compiled using the `package_xo` utility. The RTL kernel wizard in the SDAccel environment can be used to simplify this process.
   - The kernel `.xo` files are linked with the hardware platform (`.dsa`) to create the FPGA binary (`.xclbin`). Important architectural aspects are determined during the link step. In particular, this is where connections from kernel ports to global memory banks are established and where the number of instances for each kernel is specified.
     ◦ When the build target is software or hardware emulation, as described below, `xocc` generates simulation models of the device contents.
     ◦ When the build target is the system, or actual hardware, `xocc` generates the FPGA binary for the device leveraging the Vivado® Design Suite to run synthesis and implementation.

***Note*:** The `xocc` compiler automatically uses the Vivado HLS and Vivado Design Suite tools to build the kernels to run on the FPGA platform. It uses these tools with pre-defined settings which have proven to provide good quality of results. Using the SDAccel environment and the `xocc` compiler does not require knowledge of these tools. However, hardware savvy developers can fully leverage these tools and use all their available features to implement kernels.

Build Targets

The SDAccel build process generates the host application executable (`.exe`) and the FPGA binary (`.xclbin`). The SDAccel build target defines the nature of FPGA binary generated by the build process.

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

- Software Emulation (`sw_emu`): Both the host application code and the kernel code are compiled to run on the x86 processor. This allows iterative algorithm refinement through fast build and run loops. This target is useful to identify syntax issues, perform source-level debugging of the kernel code running together with application and verify the behavior of the system.

- Hardware Emulation (`hw_emu`): The kernel code is compiled into a hardware model (RTL) which is run in a dedicated simulator. This build and run loop takes longer but provides a detailed, cycle-accurate, view of kernel activity. This target is useful for testing the functionality of the logic that will go in the FPGA and for getting initial performance estimates.
- System (`hw`): The kernel code is compiled into a hardware model (RTL) and is then implemented on the FPGA device, resulting in a binary that will run on the actual FPGA.

SDAccel Optimization Flow Overview

SDAccel™ Environment is a complete software development environment for creating, compiling, and optimizing C/C++/OpenCL™ applications to be accelerated on Xilinx® FPGAs. The figure below shows the recommended flow for optimizing an application in the SDAccel Environment.

Figure 3: **SDAccel Recommended Flow**

Baselining Functionalities and Performance

It is very important to understand the performance of your application before you start any optimization effort. This is achieved by baselining the application in terms of functionalities and performance.

The first step is to find out the bottlenecks of the current application running on your existing platform. The most effective way is to run the application with profiling tools like `valgrind`/`callgrind` and GNU `gprof`. The profiling data generated by these tools show the call graph with the number of calls to all functions and their execution time. The functions that consume the most execution time are good candidates to be offloaded and accelerated onto FPGAs.

Once the target functions are selected, convert them to OpenCL™ CL kernels or C/C++ kernels without any optimization. The application code calling these kernels will also need to be converted to use OpenCL APIs for data movement and task scheduling. Keep everything as simple as possible and minimize changes to the existing code in this step so you can quickly generate a working design on the FPGA and get the baselined performance and resource number.

Next, run CPU and hardware emulation to verify the function correctness and generate profiling data on the host code and the kernels. Analyze the kernel compilation reports, profile summary, timeline trace, and device hardware transactions to understand the baselined performance estimate such as timing, interval, and latency and resource utilization such as DSP, BRAM.

The last step in baselining is to build and run the application on an FPGA acceleration card. Analyze the reports from the system compilation and the profiling data from application execution to see the actual performance and resource utilization.

Save all the reports during the baselining so that you can be reference and compare during optimization exercise.

Optimizing Data Movement

In the OpenCL™ programming model, all data are transferred from the host memory to the global memory on the device first and then from the global memory to the kernel for computation. The computation results are written back from the kernel to the global memory and lastly from the global memory to the host memory. How data can be efficiently moved around in this programming model is a key factor for determining strategies for kernel computation optimization, so it is recommended to optimize the data movement in your application before taking on optimizing the computation.

During data movement optimization, it is important to isolate data transfer code from computation code because inefficiency in computation may cause stalls in data movement. Xilinx recommends that you modify the host code and kernels with data transfer code only for this optimization step. The goal is to maximize the system level data throughput by maximizing PCIe® bandwidth utilization and DDR bandwidth utilization. It usually takes multiple iterations of running CPU emulation, hardware emulation, as well as execution on FPGAs to achieve the goal.

Optimizing Kernel Computation

One of the key benefits of FPGA is that you can create custom logic for your specific application. The goal of kernel computation optimization is to create processing logic that can consume all the data as soon as they arrive at kernel interfaces. The key metric during this step is the initiation interval (II). This is generally achieved by expanding the processing code to match the data path with techniques such as function pipelining, loop unrolling, array partitioning, data flowing, etc. The SDAccel™ Environment produces various compilation reports and profiling data during hardware emulation and system run to assist your optimization effort. Refer to the Application Profiling section in the SDAccel Environment" chapter in *SDAccel Environment User Guide* (UG1023) for details on the compilation and profiling report.

SDAccel Profiling and Optimization Features

The SDAccel™ Environment generates various reports on the kernel resource and performance during compilation. It also collects profiling data during application execution in emulation mode and on the FPGA acceleration card. The reports and profiling data provide you with information on performance bottlenecks in the application and optimization techniques that can be used to improve performance. This chapter describes how to generate the reports and collect, display, and read the profiling results in the SDAccel Environment.

System Estimate

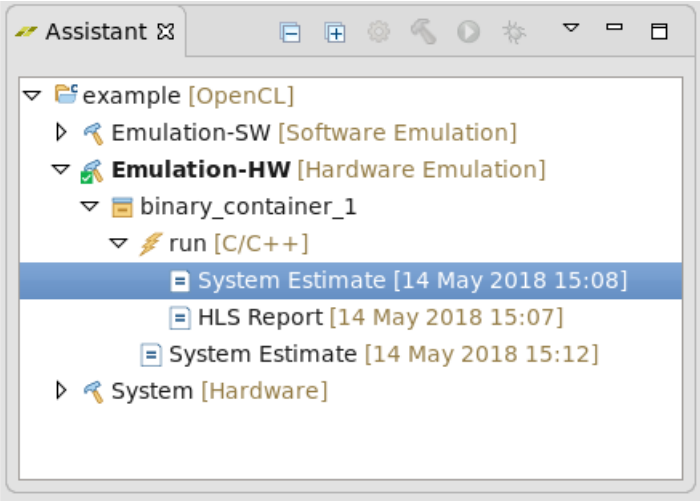
Generating FPGA programming files is the step in the SDAccel™ development environment with the longest execution time. It is also the step in which the execution time is most affected by the target hardware device and the number of compute units placed on the FPGA fabric. Therefore, it is essential for the application programmer to have a quicker way to understand the performance of the application before running it on the target device so they can spend more time iterating and optimizing their applications instead of waiting for the FPGA programming file to be generated.

The system estimate in the SDAccel development environment takes into account the target hardware device and each compute unit in the application. Although an exact performance metric can only be measured on the target device, the estimation report in the SDAccel environment provides an accurate representation of the expected behavior.

GUI Flow

This report is automatically generated during the hardware emulation flow. There is one report generated for each kernel and a top report for the complete binary container. It is easy to access the reports from the **Assistant** Window in the **Emulation-HW** folder.

The following image shows the **Assistant** window with a System Estimate report for the `binary_container_1`, and the kernel with the name `run`.



Command Line

The command below generates the system performance estimate report `system_estimate.xtxt` for all kernels in `kernel.cl`:

```
xocc -c -t hw_emu --platform xilinx:adm-pcie-7v3:1ddr:3.0 --report
estimate kernel.cl
```

The performance estimate report generated by the `xocc -report estimate` option provides information on every binary container in the application, as well as every compute unit in the design. The structure of the report is:

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

Data Interpretation

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

```
================================================================
Design Name:                _xocc_compile_kernel_bin.dir
Target Device:              xilinx:adm-pcie-ku3:2ddr-xpr:3.3
Target Clock:               200MHz
Total number of kernels: 1
================================================================

Kernel Summary
Kernel Name    Type  Target              OpenCL Library  Compute Units
-------------  ----  ------------------  --------------  -------------
smithwaterman  clc   fpga0:OCL_REGION_0  xcl_xocc        1
```

```
================================================================
OpenCL Binary:     xcl_xocc
Kernels mapped to: clc_region

Timing Information (MHz)
Compute Unit     Kernel Name    Module Name    Target Frequency
---------------  -------------  -------------  ----------------
smithwaterman_1  smithwaterman  smithwaterman  200

Estimated Frequency
-------------------
202.020203

Latency Information (clock cycles)
Compute Unit     Kernel Name    Module Name    Start Interval
---------------  -------------  -------------  --------------
smithwaterman_1  smithwaterman  smithwaterman  29468

Best Case  Avg Case  Worst Case
---------  --------  ----------
29467      29467     29467

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

Design and Target Device Summary

All design estimate reports begin with an application summary and information about the target hardware. Device information is provided by the following section of the report:

```
================================================================
Design Name:              _xocc_compile_kernel_bin.dir
Target Device:            xilinx:adm-pcie-ku3:2ddr-xpr:3.3
Target Clock:             200MHz
Total number of kernels: 1
================================================================
```

For the design summary, the only information you provide is the design name and the selection of the target device. The other information provided in this section is the target board and the clock frequency.

The target board is the name of the board that runs the application compiled by the SDAccel™ development environment. The clock frequency defines how fast the logic runs for compute units mapped to the FPGA fabric. Both of these parameters are fixed by the device developer. These parameters cannot be modified from within the SDAccel environment.

Kernel Summary

The kernel summary section lists all of the kernels defined for the current SDAccel™ solution. Following is an example kernel summary:

```
Kernel Summary
Kernel Name    Type  Target              OpenCL Library  Compute Units
-------------  ----  ------------------  --------------  -------------
smithwaterman  clc   fpga0:OCL_REGION_0  xcl_xocc        1
```

Along with the kernel name, the summary 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

The detail section for each binary container begins with the execution target of all compute units. It also provides timing information for every compute unit. As a general rule, an estimated frequency that is higher than that of the device target means that the compute unit will run in hardware. If the estimated frequency is below the target frequency, the kernel code for the compute unit needs to be further optimized for the compute unit to run correctly on the FPGA fabric. Following is an example of this information:

```
OpenCL Binary:     xcl_xocc
Kernels mapped to: clc_region

Timing Information (MHz)
Compute Unit     Kernel Name    Module Name    Target Frequency
---------------  -------------  -------------  ----------------
smithwaterman_1  smithwaterman  smithwaterman  200

Estimated Frequency
-------------------
202.020203
```

The importance of the timing information is the difference between the target and the estimated frequencies. Compute units are not placed in isolation into the FPGA fabric. Compute units 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 compute unit custom logic is generated one kernel at a time, an estimated frequency that is higher than the device target provides confidence to the developer using the SDAccel™ environment that there will not be any problems during the creation of the FPGA programming files.

Latency Information

The latency information presents the execution profile of each compute unit in the binary container. When analyzing this data, it is important to keep in mind that all values are measured from the compute unit 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 compute units targeted at the FPGA fabric. Following is an example of the latency report:

```
Latency Information (clock cycles)
Compute Unit     Kernel Name    Module Name    Start Interval  Best Case
---------------  -------------  -------------  --------------  ---------
smithwaterman_1  smithwaterman  smithwaterman  29468           29467

Avg Case  Worst Case
--------  ----------
29467     29467
```

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 compute unit for a given kernel.

The best, average, and worst case latency numbers refer to how much time it takes the compute unit 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:

- Do not have an explicit reqd_work_group_size(x,y,z)
- 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 SDAccel™ development environment to generate the custom logic for each compute unit in the design. The number of each fundamental resource needed to implement the custom logic in a compute unit determines how many compute units can be simultaneously loaded into the FPGA fabric. The following example shows the area information reported for a compute unit:

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

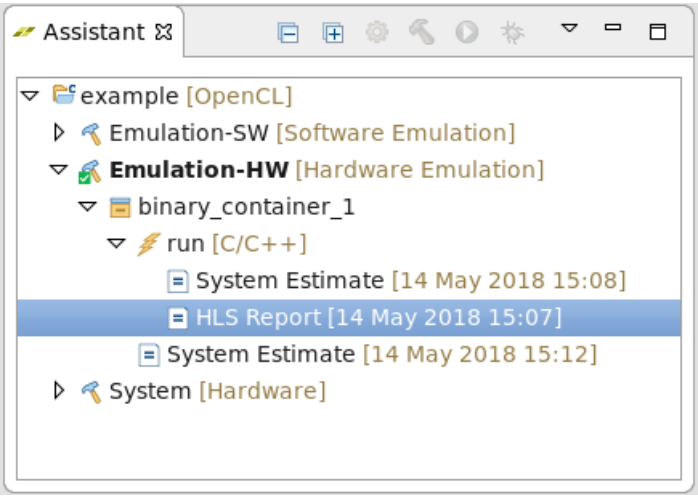
HLS Report

After compiling a kernel using the SDx™ GUI or the XOCC command line, the HLS (High-Level Synthesis) report is available to the user. The HLS report includes details about the performance and logic usage of the custom-generated hardware logic from user kernel code. These details provide advanced users many insights into the kernel compilation results to guide kernel optimization.

GUI Flow

After compiling a kernel using the SDx™ GUI, you can view the High Level Synthesis (HLS) in the **Assistant** window. The report is under the **Emulation-HW** or **System** build configuration, and has the *binary container* name, and the *kernel* name. This is illustrated in the following **Assistant** window:

Figure 4: **Assistant Window**



Command Line

The HLS Report is definitely designed to be viewed by the SDAccel™ GUI. However, for command line users, a textual representation of this report is also published. This report can be found inside the report directory situated under the kernel synthesis directory in the HLS solution directory.

Since the `xocc` command generates several additional levels of hierarchy above this synthesis directory it is best to simply locate the file by name:

```
find . -name <module>_csynth.rpt
```

where `<module>` is the name of module for which to look up the HLS synthesis report.

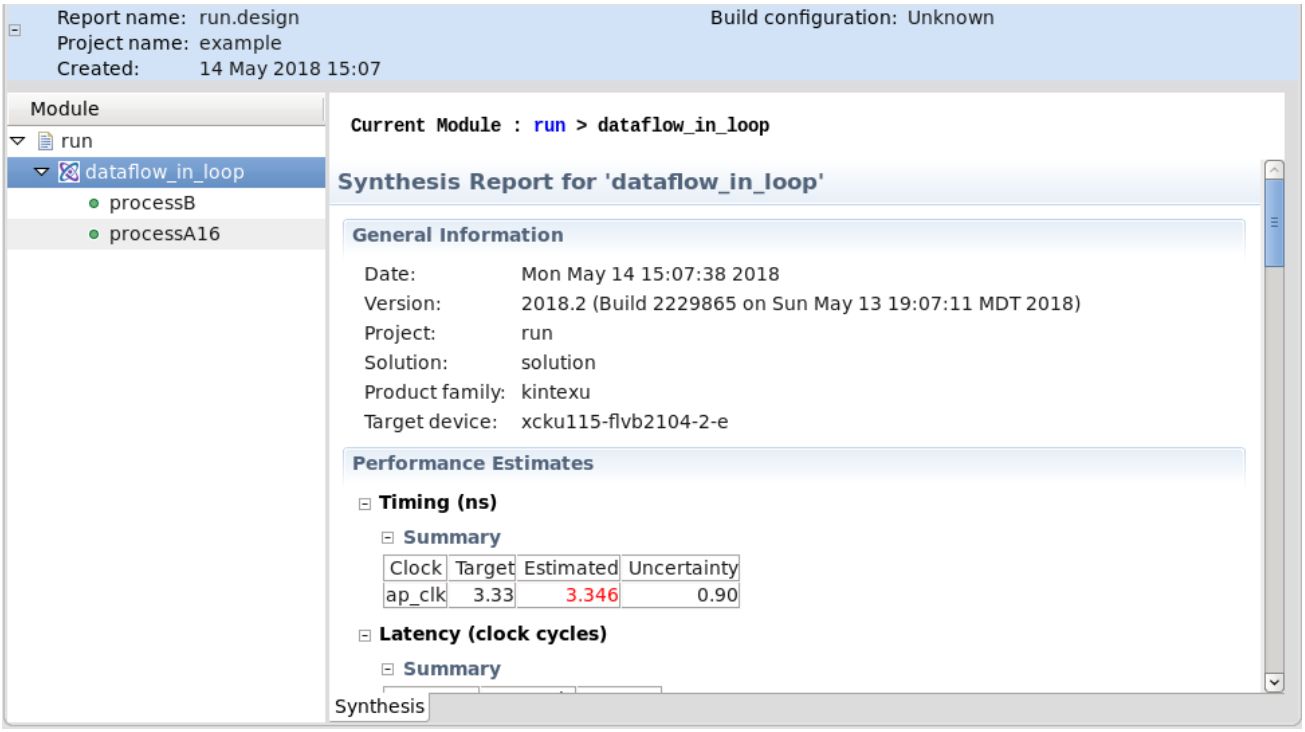
Note: The `find` command also supports the lookup using wildcards such that the following command will lookup all synthesis reports in any subdirectory:

```
find . -name "*_csynth.rpt"
```

Data Interpretation

The lefthand pane of the **HLS Report** shows the module hierarchy. Each module generated as part of the high level synthesis run is represented in this hierarchy. The SDAccel™ Environment allows you to select any of these modules and to present the synthesis details of the module in the right hand **Synthesis Report** tab.

Figure 5: **HLS Report Window**



The Synthesis Report is separated into several sections, namely **General Information**, **Performance Estimates** (timing & latency), **Utilization Estimates**, and **Interface Information**. If this information is part of a hierarchical block, it will sum up the information of the blocks contained in the hierarchy. Due to this fact, the hierarchy can also be navigated from within the report when it is clear which instance contributes what to the overall design.

CAUTION!: *Regarding the absolute counts of cycles/latency, these numbers are based on estimates identified during synthesis, especially with advanced transformations such as pipelining and dataflow, these numbers might not accurately reflect the final results. If you encounter question marks in the report, this might be due to variable bound loops and you are encouraged to set trip counts for such loops to have some relative estimates presented in this report.*

.

Profile Summary Report

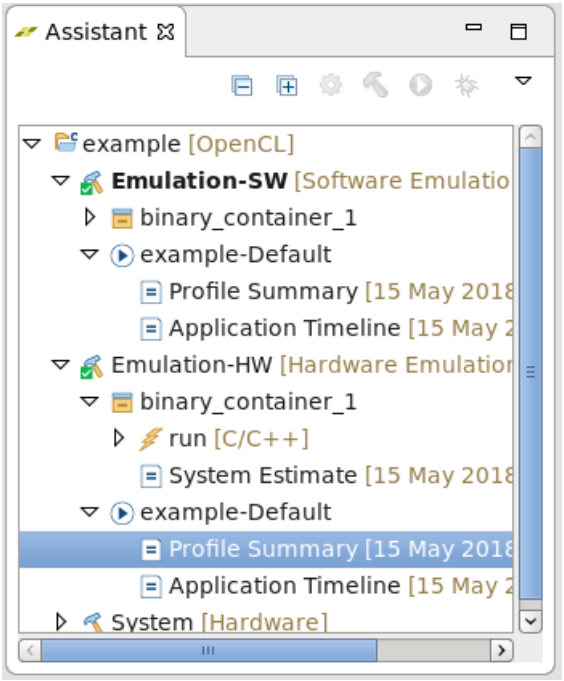
The SDAccel™ runtime automatically collects profiling data on host applications. After the application finishes execution, the profile summary is saved in HTML, `.csv`, and Google Protocol Buffer formats in the solution report directory or working directory. These reports can be reviewed in a web browser, spreadsheet viewer, or the integrated Profile Summary Viewer in SDAccel. The profile reports are generated in both SDAccel GUI and XOCC command line flows.

GUI Flow

When you compile and execute an application from SDAccel™ environment, the profile summary is automatically generated. To control the generation of profile information, simply edit the run configuration through the context menu of the build configuration and selecting **Run → Run Configurations**.

Once run, the **Assistant** enables easy access to the report from below the Run Configuration item. Once the a run configuration has executed, modifying the configuration can now be initiated directly through the context menu of the run configuration item in the **Assistant**.

Figure 6: **Profile Summary access in SDAccel GUI Flow**



Double-click the report to open it.

Command Line

Command Line users execute standalone applications outside the SDAccel™ environment. To generate the profile summary data, you can compile your design without any additional options. However, linking the bitstream file (xclbin) requires the `--profile_kernel` option.

The argument provided through the `--profile_kernel` option can be used to limit data collection, which might be required in large systems. The general syntax for the profile_kernel option with respect to the profile summary report is:

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

Three fields are required to determine the precise interface to which the performance monitor is applied to. However in case resource utilization isn't an issue, the keyword `all` enables the user to apply the monitoring to all existing kernels, compute units, and interfaces with a single option. Otherwise, the user can specify the `kernel_name`, `compute_unit_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) will include the collection of actual trace information.

Note: The `profile_kernel` option is additive and can be used multiple times on the link line.

Executing the program creates an `sdaccel_profile_summary.csv` file, if `profile = true` is specified in the `sdaccel.ini` file.

```
[Debug]
profile = true
```

The `.csv` file needs to be manually converted to Google Protocol Buffer format (`.xprf`) before the profiling result can be viewed in the integrated Profile Summary Viewer. The following is a command line example that generates an `.xprf` file from the `.csv` input file.

```
sdx_analyze profile sdaccel_profile_summary.csv
```

Displaying the Profile Summary

Use the following methods to display the SDAccel™ Profile Summary created from the command line.

Web Browser

Before the HTML profile summary can be displayed in a web browser the following command needs to be executed to create an HTML file representing the data.

```
sdx_analyze profile -i
sdaccel_profile_summary.csv -f html
```

This creates an HTML file that can be opened by the web browser of your choice. The following information shows the profiling result from a system run on the FPGA.

SDAccel Profile Summary

Profiled application: median_solution.exe

Target platform: Xilinx

Target devices: xilinx_xil-accel-rd-ku115_4ddr-xpr_4_0-0

Flow mode: System Run

OpenCL API Calls

API Name	Number Of Calls	Total Time (ms)	Minimum Time (ms)	Average Time (ms)	Maximum Time (ms)
clCreateProgramWithBinary	1	3472.65	3472.65	3472.65	3472.65
clCreateBuffer	2	3.23948	1.56216	1.61974	1.67733
clEnqueueReadBuffer	1	1.5815	1.5815	1.5815	1.5815
clWaitForEvents	2	1.57153	0.003686	0.785764	1.56784
clEnqueueWriteBuffer	1	0.820979	0.820979	0.820979	0.820979
clReleaseProgram	1	0.417685	0.417685	0.417685	0.417685
clEnqueueNDRangeKernel	1	0.071586	0.071586	0.071586	0.071586
clCreateKernel	2	0.042263	0.012971	0.0211315	0.029292
clCreateContext	1	0.023126	0.023126	0.023126	0.023126
clSetKernelArg	4	0.020083	0.001471	0.00502075	0.013706
clGetPlatformIDs	2	0.017263	0.001324	0.0086315	0.015939
clCreateCommandQueue	1	0.012844	0.012844	0.012844	0.012844
clGetDeviceIDs	1	0.010235	0.010235	0.010235	0.010235
clGetDeviceInfo	3	0.010174	0.00153	0.00339133	0.006278
clGetPlatformInfo	1	0.007695	0.007695	0.007695	0.007695
clReleaseKernel	1	0.005061	0.005061	0.005061	0.005061
clReleaseCommandQueue	1	0.00483	0.00483	0.00483	0.00483
clReleaseContext	1	0.000979	0.000979	0.000979	0.000979

Kernel Execution

Kernel	Number Of Enqueues	Total Time (ms)	Minimum Time (ms)	Average Time (ms)	Maximum Time (ms)
median	1	0.859783	0.859783	0.859783	0.859783

Compute Unit Utilization

Device	Compute Unit	Kernel	Global Work Size	Local Work Size	Number Of Calls	Total Time (ms)	Minimum Time (ms)	Average Time (ms)	Maximum Time (ms)	Clock Frequency (MHz)
xilinx_xil-accel-rd-ku115_4ddr-xpr_4_0-0	median_1	median	1:1:1	1:1:1	1	0.822565	0.822565	0.822565	0.822565	266

Data Transfer: Host and Global Memory

Context:Number of Devices	Transfer Type	Number Of Transfers	Transfer Rate (MB/s)	Average Bandwidth Utilization (%)	Average Size (KB)	Total Time (ms)	Average Time (ms)
context0:1	READ	1	706.107030	7.355282	1048.58	1.485010	1.485010
context0:1	WRITE	1	1743.932436	18.165963	1048.58	0.601271	0.601271

Data Transfer: Kernels and Global Memory

Device	Transfer Type	Number Of Transfers	Transfer Rate (MB/s)	Average Bandwidth Utilization (%)	Average Size (KB)	Average Time (ns)	Device Execution Time (ms)
xilinx_xil-accel-rd-ku115_4ddr-xpr_4_0-0	READ	1026	1221.96	10.6073	1.024	280.331	0.859783
xilinx_xil-accel-rd-ku115_4ddr-xpr_4_0-0	WRITE	1024	1219.58	10.5867	1.024	55	0.859783

Top Data Transfer: Kernels and Global Memory

Device	Kernel Name	Number of Transfers	Average Bytes per Transfer	Transfer Efficiency (%)	Total Data Transfer (MB)	Total Write (MB)	Total Read (MB)	Transfer Rate (MB/s)	Average Bandwidth Utilization (%)
xilinx_xil-accel-rd-ku115_4ddr-xpr_4_0-0	ALL	2050	1024	25	2.0992	1.04858	1.05062	2441.55	21.194

Profile Summary Viewer

Use the integrated Profile Summary Viewer to display profile summary generated by the command line flow.

Follow these steps to open the profile summary in the Profile Summary Viewer:

1. Convert the `.csv` data file into the protobuf format.

```
sdx_analyze profile -i
sdaccel_profile_summary.csv -f protobuf
```

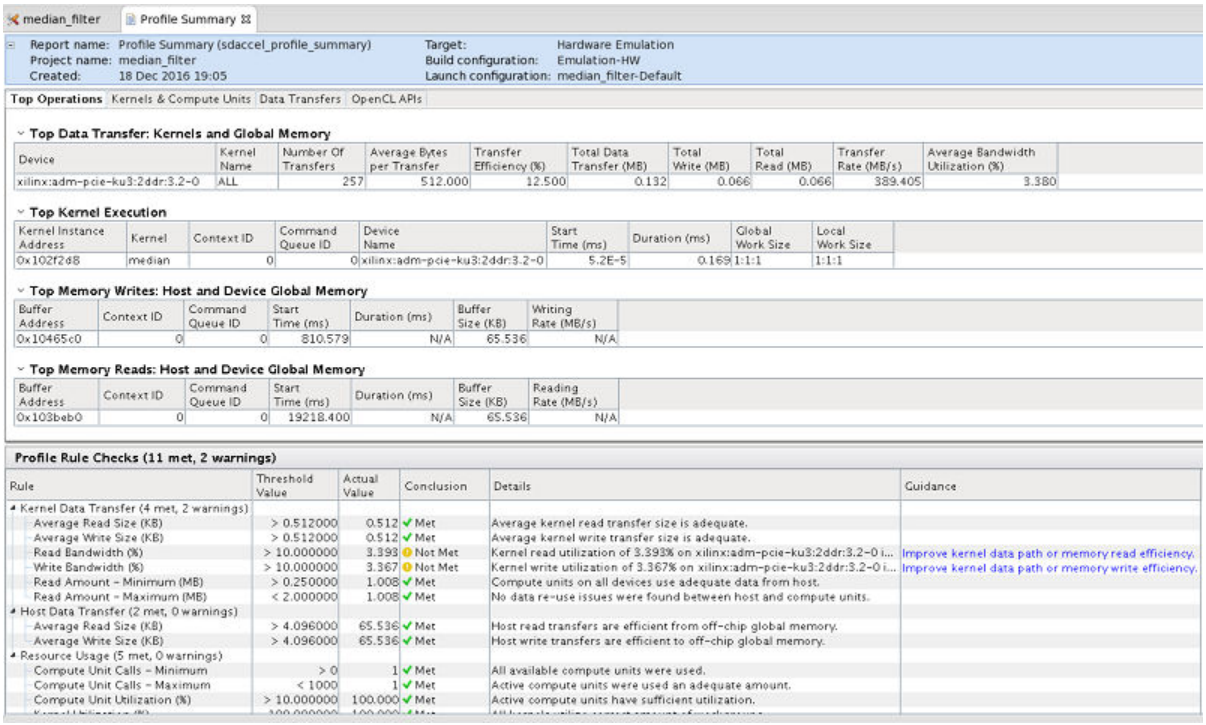
2. Start SDAccel GUI by running the `sdx` command:

```
$sdx
```

3. Choose the default workspace when prompted.
4. Select **File → Open File**, browse to and then open the `.xprf` file created by the `sdx_analyze` command run in step 1.

Below is a snapshot of the Profile Summary window that displays OpenCL™ API calls, kernel executions, data transfers, and profile rule checks (PRCs).

Figure 7: **Profile Summary Window**



Data Interpretation

The profile summary includes a number of useful statistics for your OpenCL™ application. This can provide you with a general idea of the functional bottlenecks in your application. The profile summary consists of the following tables:

- **Top Operations**
  - **Top Data Transfer: Kernels and Global Memory**: This table displays the profile data for top data transfers between FPGA and device memory.
    - **Device**: Name of device
    - **Compute Unit**: Name of compute unit
    - **Number of Transfers**: Sum of write and read AXI transactions monitored on device
    - **Average Bytes per Transfer**: (Total Read Bytes + Total Write Bytes) / (Total Read AXI Transactions + Total Write AXI Transactions)
    - **Transfer Efficiency (%)**: (Average Bytes per Transfer) / min(4K, (Memory Bit Width/8 * 256))

      AXI4 specification limits the max burst length to 256 and max burst size to 4K bytes.
    - **Total Data Transfer (MB)**: (Total Read Bytes + Total Write Bytes) / 1.0e6

    - **Total Write (MB)**: (Total Write Bytes) / 1.0e6
    - **Total Read (MB)**: (Total Read Bytes) / 1.0e6
    - **Transfer Rate (MB/s)**: (Total Data Transfer) / (Compute Unit Total Time)
  - **Top Kernel Execution**
    - **Kernel Instance Address**: Host address of kernel instance (in hex)
    - **Kernel**: Name of kernel
    - **Context ID**: Context ID on host
    - **Command Queue ID**: Command queue ID on host
    - **Device**: Name of device where kernel was executed (format: `<device>-<ID>`)
    - **Start Time (ms)**: Start time of execution (in ms)
    - **Duration (ms)**: Duration of execution (in ms)
    - **Global Work Size**: NDRange of kernel
    - **Local Work Size**: Work group size of kernel
  - **Top Memory Writes: Host and Devie Global Memory**
    - **Buffer Address**: Host address of buffer (in hex)
    - **Context ID**: Context ID on host
    - **Command Queue ID**: Command queue ID on host
    - **Start Time (ms)** : Start time of write transfer (in ms)
    - **Duration (ms)**: Duration of write transfer (in ms)
    - **Buffer Size (KB)**: Size of write transfer (in KB)
    - **Writing Rate (MB/s)**: Writing Rate = (Buffer Size) / (Duration)
  - **Top Memory Reads: Host and Device Global Memory**
    - **Buffer Address**: Host address of buffer (in hex)
    - **Context ID**: Context ID on host
    - **Command Queue ID**: Command queue ID on host
    - **Start Time (ms)**: Start time of read transfer (in ms)
    - **Duration (ms)**: Duration of read transfer (in ms)
    - **Buffer Size (KB)**: Size of read transfer (in KB)
    - **Reading Rate (MB/s)**: Reading Rate = (Buffer Size) / (Duration)
- **Kernels & Compute Units**

- **Kernel Execution (includes estimated device times)**: This table displays the profile data summary for all kernel functions scheduled and executed.
  - **Kernel**: Name of kernel
  - **Number of Enqueues**: Number of times kernel is enqueued
  - **Total Time (ms)** Sum of runtimes of all enqueues (measured from START to END in OpenCL execution model)
  - **Minimum Time (ms)** Minimum runtime of all enqueues
  - **Average Time (ms)** (Total Time) / (Number of Enqueues)
  - **Maximum Time (ms)** Maximum runtime of all enqueues
- **Compute Unit Utilization (includes estimated device times)**: This table displays the summary profile data for all compute units on the FPGA.
  - **Device**: Name of device (format: `<device>-<ID>`)
  - **Compute Unit**: Name of Compute Unit
  - **Kernel**: Kernel this Compute Unit is associated with
  - **Global Work Size**: NDRange of kernel (format is x:y:z)
  - **Local Work Size**: Local work group size (format is x:y:z)
  - **Number of Calls**: Number of times the Compute Unit is called
  - **Total Time (ms)**: Sum of runtimes of all calls
  - **Minimum Time (ms)**: Minimum runtime of all calls
  - **Average Time (ms)**: (Total Time) / (Number of Work Groups)
  - **Maximum Time (ms)**: Maximum runtime of all calls
  - **Clock Frequency (MHz)**: Clock frequency used for a given accelerator (in MHz)

- **Data Transfers**
  - **Data Transfer: Host and Global Memory**: This table displays the profile data for all read and write transfers between the host and device memory via PCI Express® link.
    - **Context:Number of Devices**: Context ID and number of devices in context
    - **Transfer Type**: READ or WRITE
    - **Number of Transfers**: Number of host data transfers (NOTE: may contain `printf` transfers)
    - **Transfer Rate (MB/s)** (Total Bytes Sent) / (Total Time in usec)

      where Total Time includes software overhead
    - **Average Bandwidth Utilization (%)**: (Transfer Rate) / (Max. Transfer Rate)

where Max. Transfer Rate = (256/8 bytes) * (300 MHz) = 9.6 GBps

- **Average Size (KB)**: (Total KB sent) / (number of transfers)
- **Total Time (ms)**: Sum of transfer times
- **Average Time (ms)**: (Total Time) / (number of transfers)

◦ **Data Transfer: Kernels and Global Memory**: This table displays the profile data for all read and write transfers between the FPGA and device memory.

- **Device**: Name of device
- **Compute Unit/Port Name**: <Name of Compute Unit>/<Name of Port>
- **Kernel Arguments**: List of arguments connected to this port
- **DDR Bank**: DDR bank number this port is connected to
- **Transfer Type**: READ or WRITE
- **Number of Transfers**: Number of AXI transactions monitored on device (NOTE: may contain `printf` transfers)
- **Transfer Rate (MB/s)**: (Total Bytes Sent) / (Compute Unit Total Time)
  - Compute Unit Total Time = Total execution time of compute unit
  - Total Bytes Sent = sum of bytes across all transactions
- **Average Bandwidth Utilization (%)**: (Transfer Rate) / (0.6 *Max. Transfer Rate)

  where Max. Transfer Rate = (512/8 bytes) * (300 MHz) = 19200 MBps
- **Average Size (KB)**: (Total KB sent) / (number of AXI transactions)
- **Average Latency (ns)**: (Total latency of all transaction) / (number of AXI transactions)

- `OpenCL API Calls`: This table displays the profile data for all OpenCL host API function calls executed in the host application.

◦ **API Name**: Name of API function (e.g., clCreateProgramWithBinary, clEnqueueNDRangeKernel)

◦ **Number of Calls**: Number of calls to this API

◦ **Total Time (ms)**: Sum of runtimes of all calls

◦ **Minimum Time (ms)**: Minimum runtime of all calls

◦ **Average Time (ms)**: (Total Time) / (Number of Calls)

◦ **Maximum Time (ms)**: Maximum runtime of all calls

Application Timeline

Application Timeline collects and displays host and device events on a common timeline to help you understand and visualize the overall health and performance of your systems. These events include:

- OpenCL™ API calls from the host code.
- Device trace data including AXI transaction start/stop, kernel start/stop, etc.

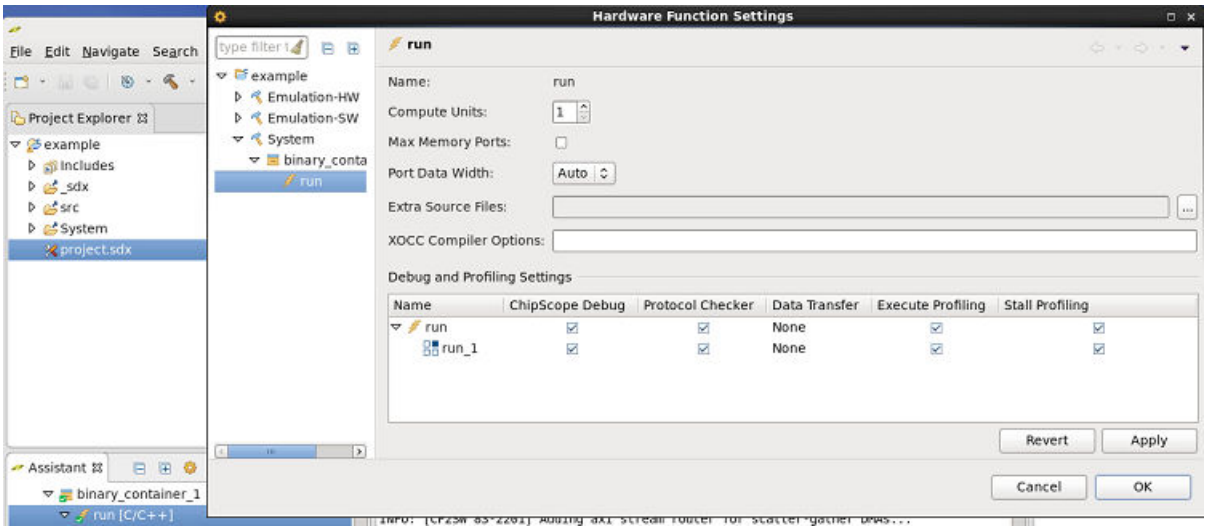
Timeline and device trace data are not collected by default because the runtime needs to periodically unload the trace data from the FPGA, which can add additional time to the overall application execution. However, the device data is collected with dedicated hardware inside the FPGA, so the data collection does not affect kernel functionality on the FPGA. The following sections describe setups required to enable time and device data collection.

Turning on device profiling is intrusive and can negatively affect overall performance. This feature should be used for system performance debugging only.

GUI Flow

Timeline and device trace data collection is part of run configuration for an SDAccel™ project created from the SDAccel integrated environment. Follow the steps below to enable it:

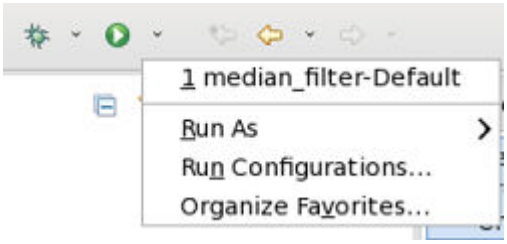
1. Instrumenting the code is required for System execution. This is done through the Hardware Function Settings dialog (**Assistant** → **Settings...**.



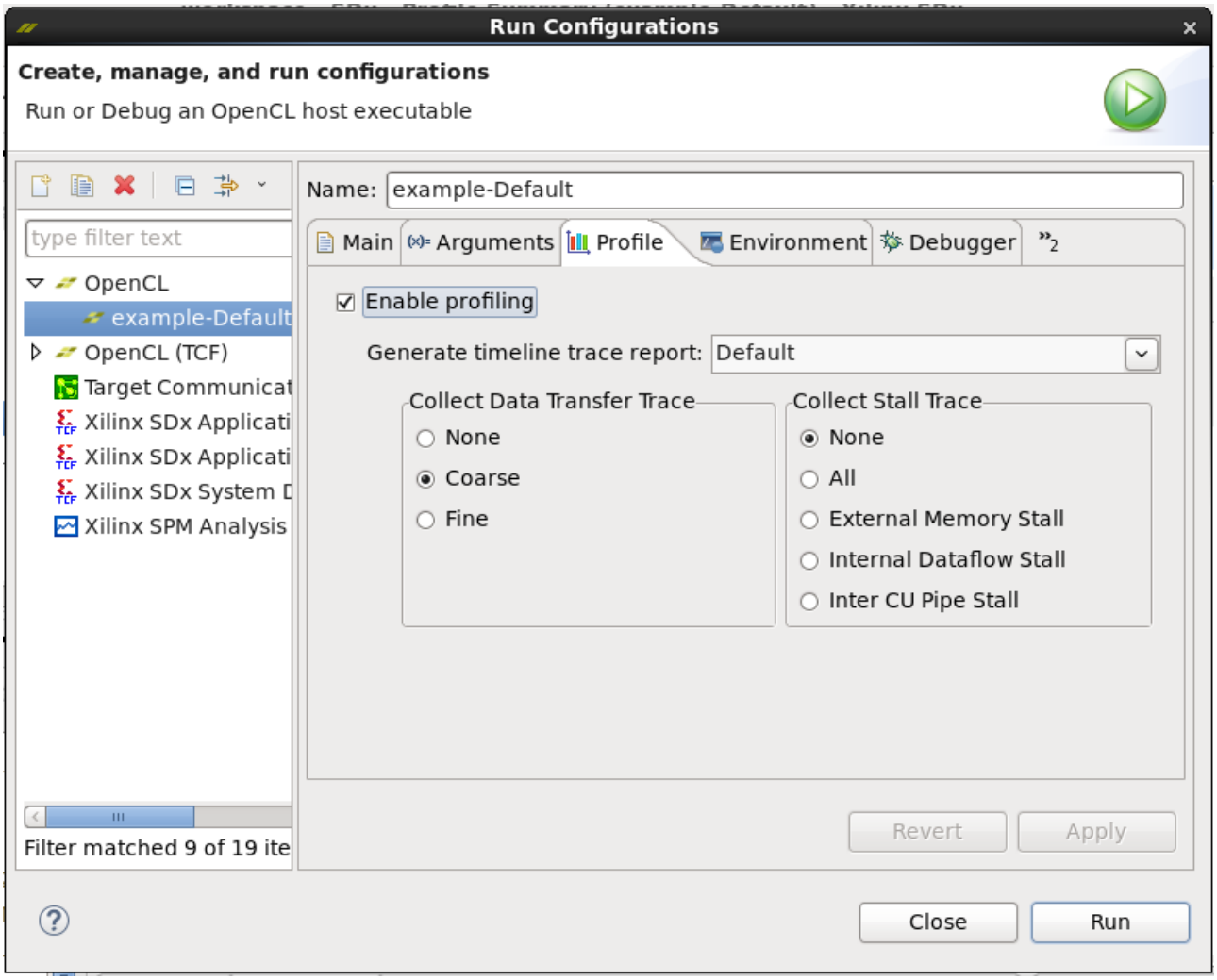
With respect to application timeline functionality, you can enable Data Transfer, Execute Profile, and Stall Profiling. These options are instrumenting all ports of each instance of any kernel. As these options insert additional hardware, instrumenting all ports might be too much. Towards that end, more control is available through command line options as detailed in the Command Line section. These options are ignored except in system emulation. During hardware emulation, this data is generated by default.

- `Data Transfer`: This option enables monitoring of data ports through SAM and SPM IPs.
- `Execute Profiling`: This option provides minimum port data collection during system run. It simply records the execution times of the kernel through the use of SAM IP. Execute profiling is enabled by default for data and stall profiling.
- `Stall Profiling`: This option includes the stall monitoring logic (using SAM IP) in the bitstream.

2. Specify what information is actually going to be reported during a run. Note, only information actually exposed from the hardware during system execution is reported. To configure reporting, click the down arrow next to the **Debug** or **Run** button and then select **Run Configurations** to open the **Run Configurations** window.



3. In the **Run Configurations** window, click the **Profile** tab and ensure the **Enable profiling** check box is selected. This enables basic profiling support. With respect to trace data, ensure that **Generate timeline trace report** actually gathers the information in the build config you are running. Default implies that no trace data capturing is supported in system execution, but enabled by default in hardware emulation.



In addition, you can select the amount of information to be gathered during runtime. You can select the granularity for trace data collection independently for Data Transfer, Trace, and Stall Trace.

- Data Transfer Trace options:
  - **coarse**: Show compute unit transfer activity from beginning of first transfer to end of last transfer (before compute unit transfer ends).
  - **fine**: Show all AXI-level burst data transfers.
  - **none**: Turn off reading and reporting of device-level trace during runtime.
- Stall Trace Options:
  - **none**: Turn off any stall trace information gathering.
  - **all**: Record all stall trace information.

- **internal data flow stall**: Intra-kernel streams (e.g., writing to full FIFO between data flow blocks).
- **internal memory stall**: Memory stalls to DDR (e.g., AXI-MM read from DDR).
- **inter compute unite pipe stall**: Inter-kernel pipe (e.g., writing to full OpenCL pipe between kernels).

If you have multiple run configurations for the same project, you must change the profile settings for each run configuration.

4. After running configurations, double-click **Application Timeline** in the **Assistant** window to open the **Application Timeline** window.

Command Line

Follow these instructions to enable timeline and device trace data collection in the Command Line flow:

1. This step is responsible for the FPGA bitstream instrumentation with SDx™ Accel Monitors (SAM) and SDx Performance Monitors (SPMs). The instrumentation is performed through the `--profile_kernel`, which has three distinct instrumentation options (`data`, `stall`, `exec`).

***Note*:** The `--profile_kernel` option is ignored except for system compilation and linking. During hardware emulation, this data is generated by default.

In general, the `--profile_kernel` option has three fields that are required to determine the precise interface to which the monitors is applied to. However, if resource utilization is not an issue, the keyword `all` enables the user to apply the monitoring to all existing kernels, compute units, and interfaces with a single option. Otherwise, the user can specify the `kernel_name`, `compute_unit_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*:** The `--profile_kernel` option is additive and can be used multiple times on the link line.

- `data`: This option enables monitoring of data ports through SAM and SPM IPs. This option needs to be set only during linking.

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

- `stall`: This option needs to be applied during compilation:

```
-c --profile_kernel <[stall]:<[kernel_name|all]:[compute_unit_name|
all]:[counters|all]>
```

and during linking:

```
-l --profile_kernel <[stall]:<[kernel_name|all]:[compute_unit_name|
all]:[counters|all]>
```

This option includes the stall monitoring logic (using SAM IP) in the bitstream. However, it does require that stall ports are present on the kernel interface. To facilitate this, the option is required for compilation of the C/C++/OpenCL™ kernel modules.

- `exec`: This option provides minimum port data collection during system run. It simply records the execution times of the kernel through the use of SAM IP. This feature is by default enabled on any port that uses the data or stall data collection. This option needs to be provided only during linking.

```
-l --profile_kernel <[exec]:<[kernel_name|all]:[compute_unit_name|
all]>:[counters|all]
```

2. After the kernels are instrumented, data gathering needs to be enabled during runtime execution. Do this through the use of the `sdaccel.ini` file that is in the same directory as the host executable. The following `sdaccel.ini` file will enable maximum information gathering during runtime:

```
[Debug]
profile=true
timeline_trace=true
data_transfer_trace=coarse
stall_trace=all
```

- `profile=<true|false>`: When this option is specified as true, basic profile monitoring is enabled. Without any additional options, this implies that the host runtime logging profile summary is enabled. However, without this option enabled, no monitoring is performed at all.
- `timeline_trace=<true|false>`: This option will enable timeline trace information gathering of the data. Without adding profile IP into the FPGA (data), it will only show host information. At a minimum, to get more compute unit start and end execution times in the timeline trace the cu needs to be linked with `--profile_kernel exec`.
- `data_transfer_trace=<coarse|fine|off>`: This option enables device-level AXI data transfers trace:
  - `coarse`: Show compute unit transfer activity from beginning of first transfer to end of last transfer (before compute unit transfer ends).
  - `fine`: Show all AXI-level burst data transfers.
  - `off`: Turn off reading and reporting of device-level trace during runtime.
- `stall_trace=<dataflow|memory|pipe|all|off>`: Specify what type(s) of stalls to capture and report in timeline trace. The default is `all`.
  - `off`: Turn off any stall trace information gathering.

- `all`: Record all stall trace information.
- `dataflow`: Intra-kernel streams (e.g., writing to full FIFO between dataflow blocks).
- `memory`: External memory stalls (e.g., AXI-MM read from DDR).
- `pipe`: Inter-kernel pipe (e.g., writing to full OpenCL pipe between kernels).

3. In command line mode, CSV files are generated to capture the trace data. These CSV reports need to be converted to the Application Timeline format using the `sdx_analyze` utility before they can be opened and displayed in the SDAccel GUI.

```
sdx_analyze trace sdaccel_timeline_trace.csv
```

   This creates the `sdaccel_timeline_trace.wdb` file by default, which can be opened from the GUI.

4. To visualize the timeline report host and device events during application execution:

   a. Start the SDx IDE by running the command:

```
$sdx
```

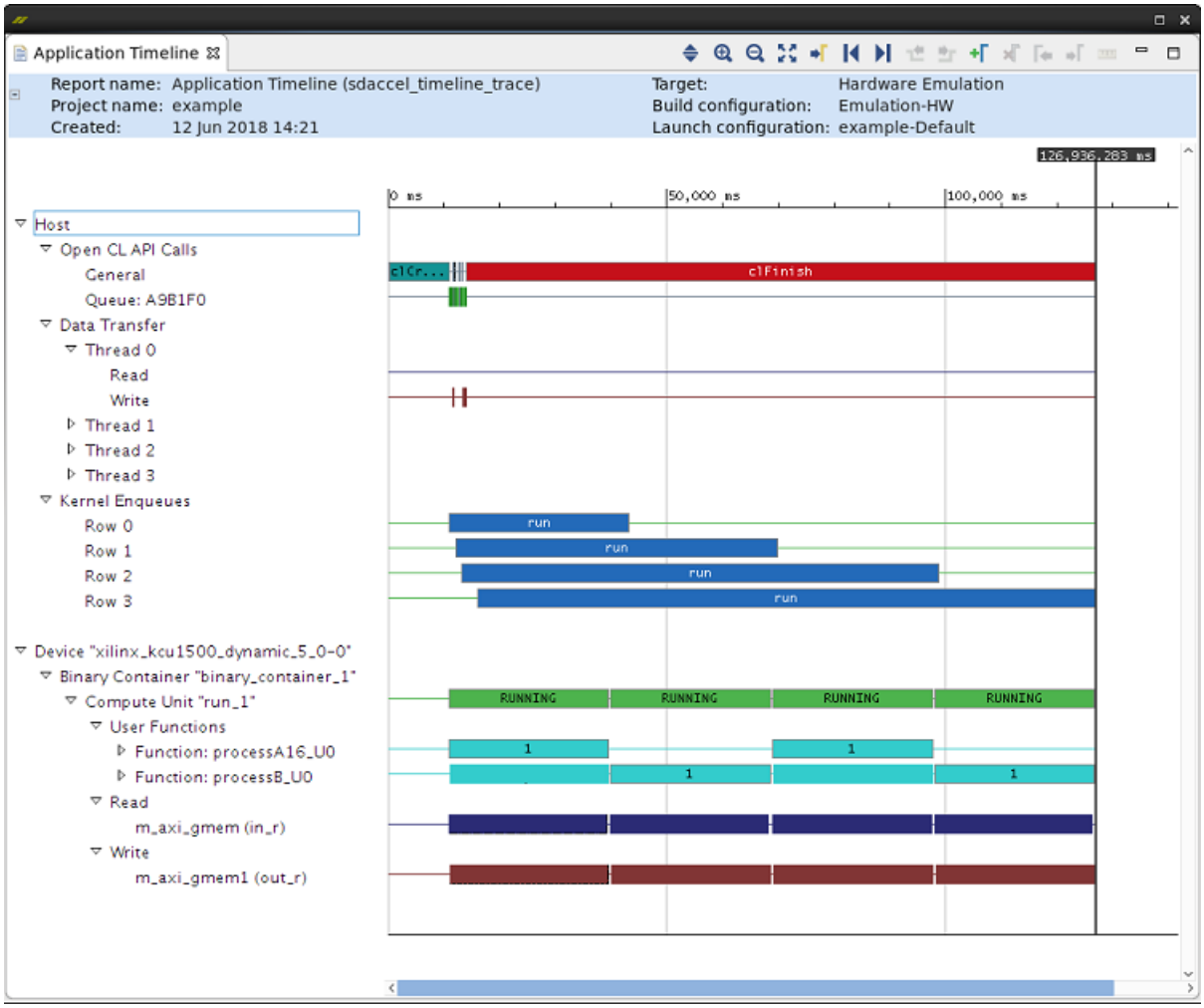
   b. Choose a workspace when prompted.

   c. Select **File → Open File**, browse to the `.wdb` file generated during hardware emulation or system run, and open it.

Data Interpretation

Below is a snapshot of the Application Timeline window that displays host and device events on a common Timeline. This information helps you to understand details of application execution and identify potential areas for improvements.

Figure 8: **Application Timeline Window**



Application timeline trace 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 compute-units on the FPGA.

Under the host different activities are categorized as OpenCL™ API calls, Data Transfer, and the Kernels.

The complete tree 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(), clCreateCommandQueue etc are traced here

- **Queue**: OpenCL API calls that are associated with a specific command queue are traced here. This includes commands such as clEnqueueMigrateMemObjects, clEnqueueNDRangeKernel etc. If the user application creates multiple command queues, then this section show as many queues and activities under it.

  - **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**, and the transfers from device to host appear under **Read**.

    - **Thread 0**
      - **Read**
      - **Write**
    - **Thread 1**
    - **Thread 2**
    - **Thread 3**

  - **Kernel Enqueues**: The active kernel executions are shown here. The kernels here should not be confused with the users kernels/compute-unit on the device. By kernels here we mean the NDRangeKernels and the Tasks created by APIs clEnqueueNDRangeKernels() and clEnqueueTask() and 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 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 real parallel execution on the device as the process might not be ready to actually execute right away.

    - **Row 0**
    - **Row 1**
    - **Row 2**
    - **Row 3**

- **Device "name"**

  - **Binary Container "name"**

    - **Accelerator "name"**: This is the name of the compute unit (aka. Accelerator) on the FPGA.

- **User Functions**: In the case of the HLS kernels, the user 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 hw emulation when waveform is enabled.
  - **Function: "name a"**
  - **Function: "name b"**
- **Read**: A compute unit reads from the DDR over AXI-MM ports. The trace of data a read by a compute unit is shown here. The activity is shown as transaction and the tool-tip for each transaction shows more details of the AXI transaction. These names are generated when `--profile_kernel data` is used.
  - **m_axi_<bundle name>(port)**
- **Write**: A compute unit writes to the DDR over AXI-MM ports. The trace of data written by a compute unit is shown here. The activity is shown as transactions and the tool-tip for each transaction shows more details of the AXI transaction. This is generated when `--profile_kernel data` is used.
  - **m_axi_<bundle name>(port)**

Waveform View and Live Waveform Viewer

The SDx™ Development Environment can generate a Waveform view and spawn a Live Waveform viewer when running hardware emulation. It displays in-depth details on the emulation results at system level, compute unit (CU) level, and at function level. The details include data transfers between the kernel and global memory, data flow via inter-kernel pipes as well as data flow via intra-kernel pipes. They provide many insights into the performance bottleneck from the system level down to individual function call to help developers optimize their applications.

Waveform View and Live Waveform are not enabled by default because they require the runtime to generate a simulation waveform during hardware emulation, which consumes more time and disk space. The following sections describe setups required to enable data collection.

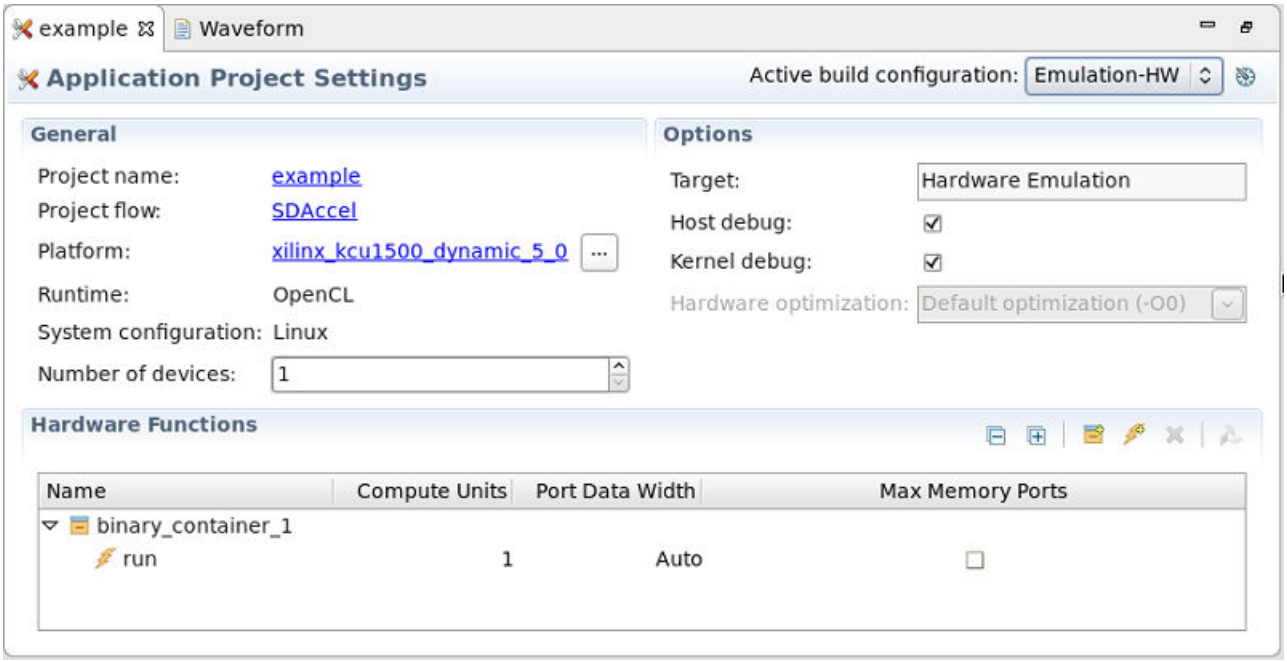
Note: The Waveform view allows you to look directly at the device transactions from within the SDx™ Development Environment. In contrast, the Live Waveform capability actually spawns the simulation waveform viewer that visualizes the hardware transactions in addition to potentially user selected internal signals.

GUI Flow

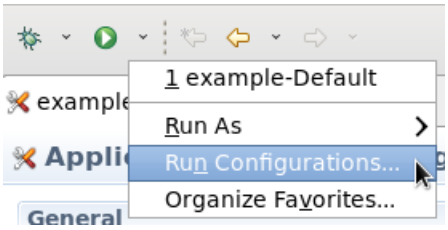
Follow the steps below to enable waveform data collection and to open the viewer:

1. Open the **Application Project Settings** window and check the **Kernel debug** check box.

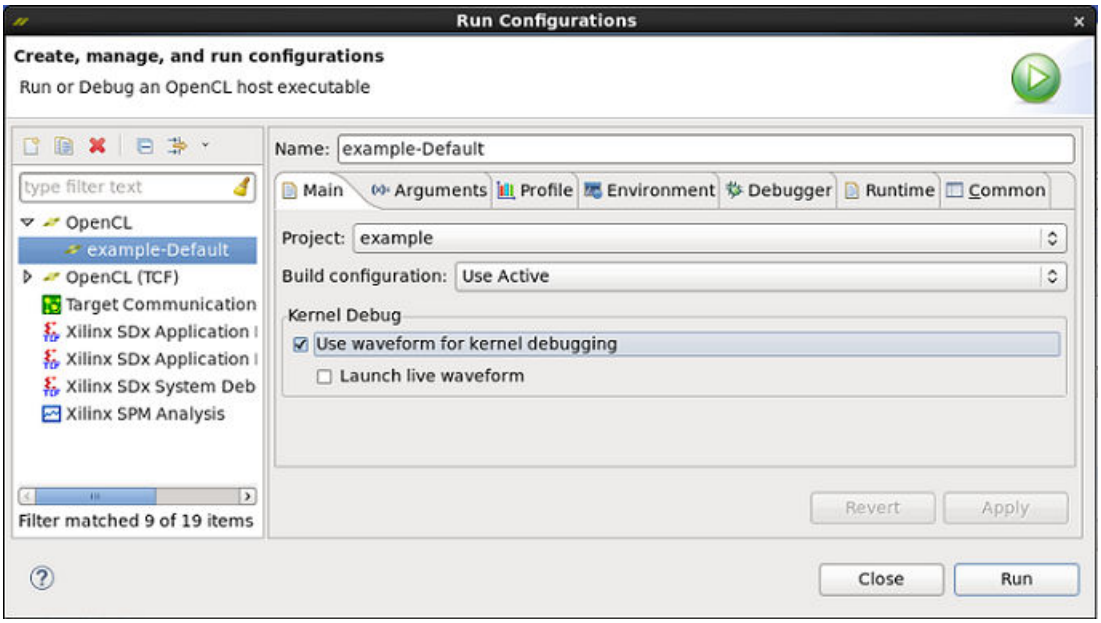
.



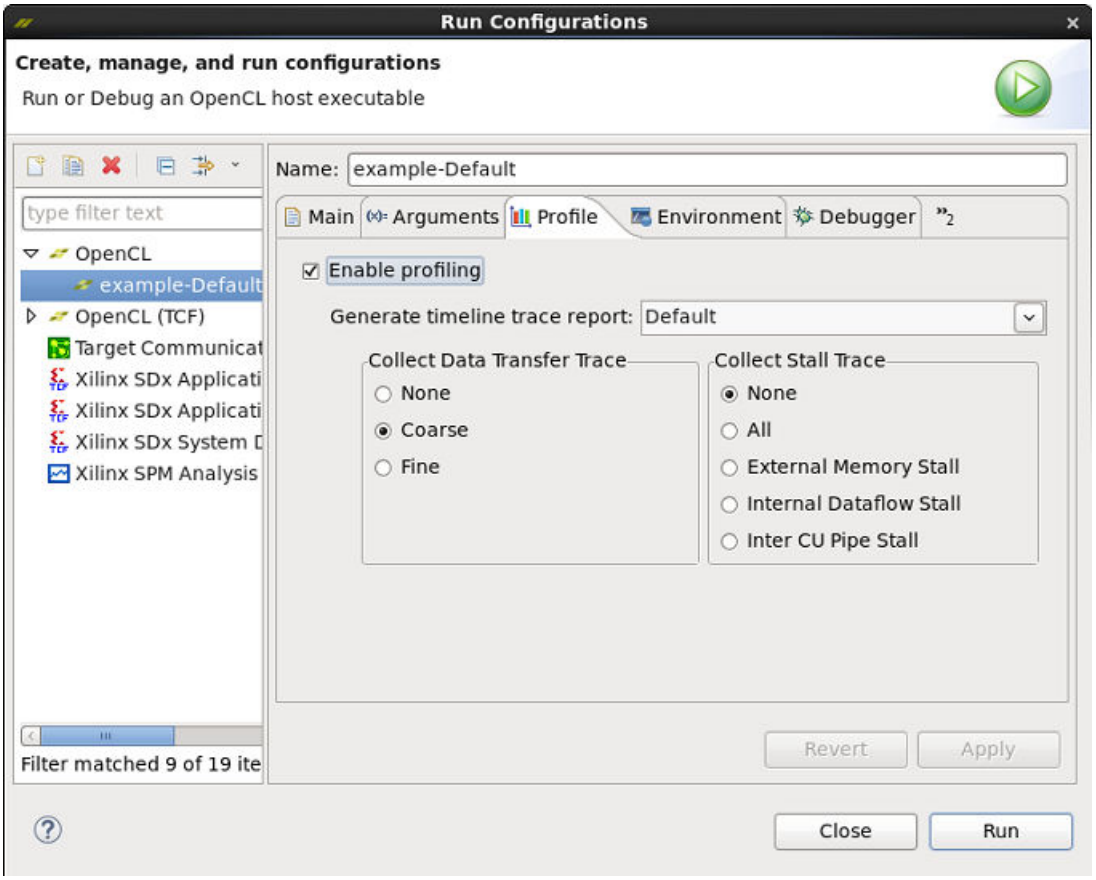
2. Click the down arrow next to the **Run** button and select **Run Configurations** to open the **Run Configurations** window.



3. On the **Run Configurations** window, click the **Main** tab and check the **Use waveform for kernel debugging** check box. Optionally, you can check **Launch live waveform** to bring up the **Simulation** window to view the Live Waveform while the hardware emulation is running.



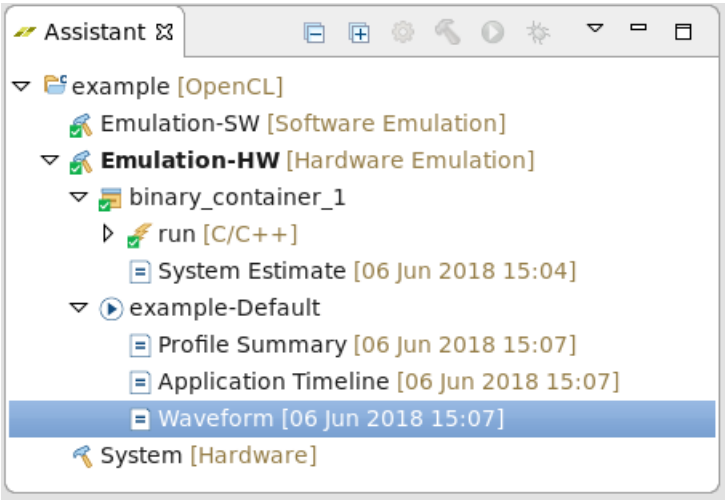
4. In the **Run Configurations** window, click the **Profile** tab and ensure the **Enable profiling** check box is selected. This enables basic profiling support.



If you have multiple run configurations for the same project, you must change the profile settings for each run configuration.

5. If you have not selected the Live Waveform viewer to be spawned automatically, open the Waveform view from the SDx™ Development Environment.

   In the SDx Development Environment, double-click **Waveform** in the **Assistant** window to open the **Waveform** view window.



Command Line

Follow these instructions to enable waveform data collection from the Command Line during hardware emulation and open the viewer:

1. Turn on debug code generation during kernel compilation.

```
xocc -g -t hw_emu ...
```

2. Create an `sdaccel.ini` file in the same directory as the host executable with the contents below:

```
[Debug]
profile=true
timeline_trace=true
```

   This enables maximum observability. The options in detail are:

   - **profile=<true|false>**: Setting this option to true, enables profile monitoring. Without any additional options, this implies that the host runtime logging profile summary is enabled. However, without this option enabled, no monitoring is performed at all.
   - **timeline_trace=<true|false>**: This option enables timeline trace informatioin gathering of the data.

3. Execute hardware emulation. The hardware transaction data is collected in the file `<hardware_platform>-<device_id>-<xclbin_name>.wdb`.
4. To see the live waveform and additional simulation waveforms, add the following to the emulation section in the `sdaccel.ini`:

   ```
   [Emulation]
   launch_waveform=gui
   ```

   A Live Waveform viewer is spawned during the execution of the hardware emulation, which allows you to examine the waveforms in detail.
5. If no Live Waveform viewer was requested, follow the steps below to open the Waveform view:
   a. Start the SDx™ IDE by running the following command:

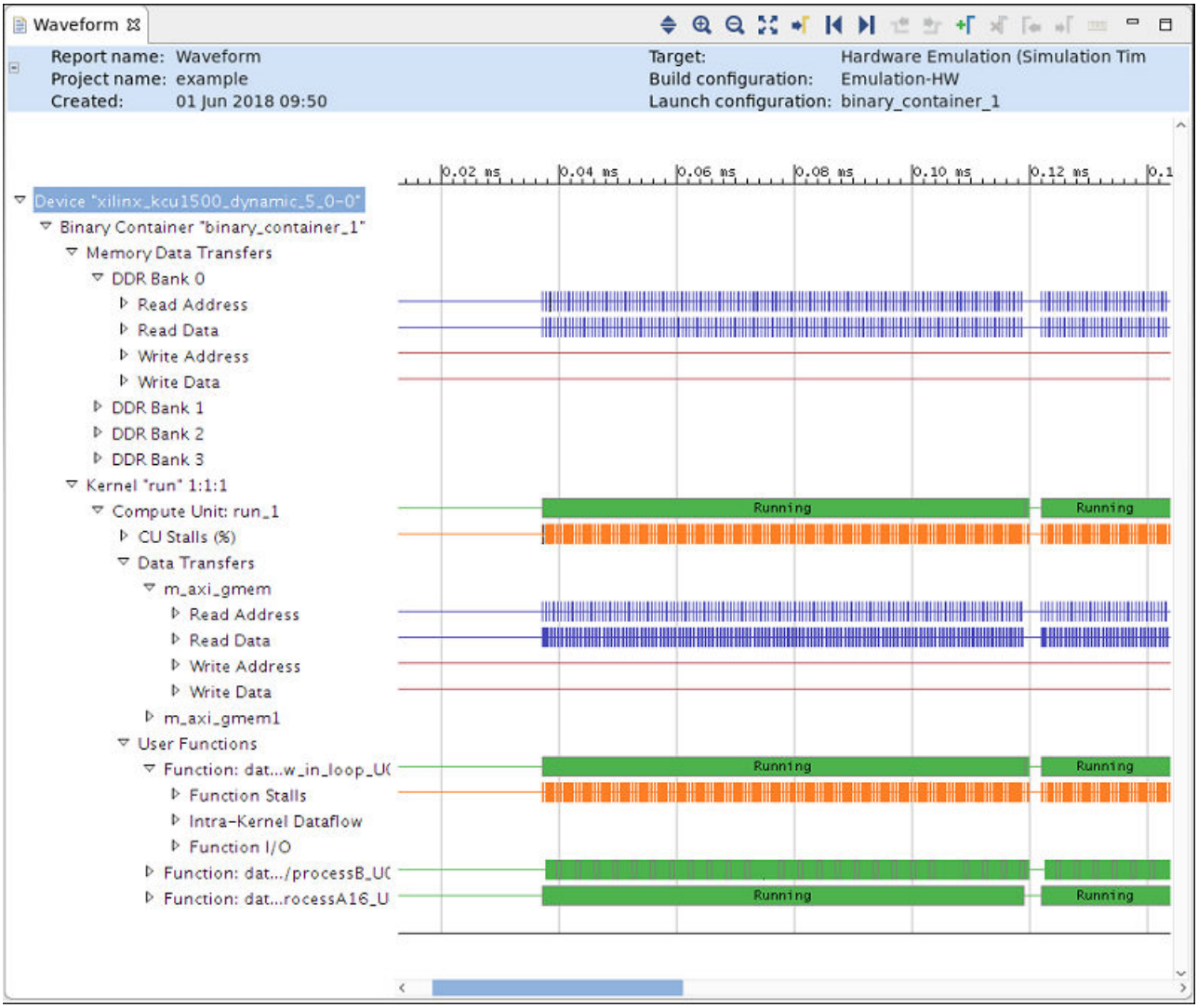
      ```
      $sdx
      ```

   b. Choose a workspace when prompted.
   c. Select **File** → **Open File**, browse to the `.wdb` file generated during hardware emulation.

Data Interpretation Waveform View

Below is a snapshot of the Waveform view:

Figure 9: **Waveform View**



The Waveform view is organized hierarchically for easy navigation. Note, this viewer is based on the actual waveforms generated during hardware emulation (Kernel Trace). This allows this viewer to descend all the way down to the individual signals responsible for the abstracted data. However, as it is post processing the data, no additional signals can be added and some of the runtime analysis such as DATAFLOW transactions cannot be visualized. Below are the hierarchy tree and descriptions:

- **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.
      - **DDR Bank 0**

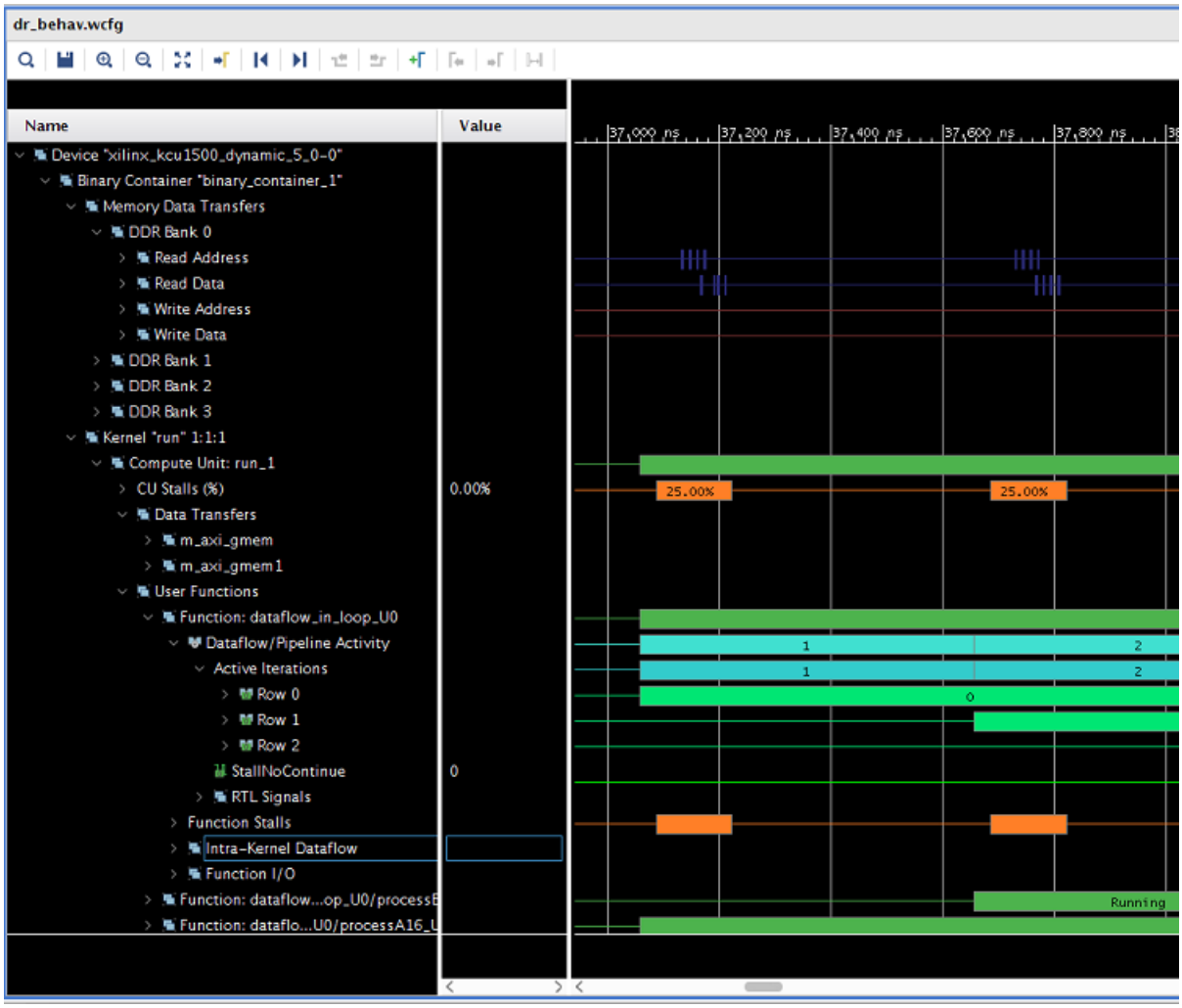
        - **Read Address**
        - **Read Data**
        - **Write Address**
        - **Write Data**
    - **DDR Bank 1**
    - **DDR Bank 2**
    - **DDR Bank 3**
- **Kernel "name" 1:1:1**: For each kernel and for each compute unit of that kernel, this section breaks down the activities originating from the compute unit.
    - **Compute Unit: "name"**: Compute unit name.
        - **CU Stalls (%)**: Stall signals are provided by HLS to inform the user when a portion of their circuit is stalling because of external memory accesses, internal streams (i.e., dataflow), or external streams (i.e., 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 would be 9%.
        - **Data Transfers**: This shows the read/write data transfer accesses originating from each Master AXI port of the compute unit to the DDR.
            - **m_axi_<bundle name>**
                - **Read Address**
                - **Read Data**
                - **Write Address**
                - **Write Data**
            - **m_axi_<bundle name>**
        - **User Functions**: This information is available for the HLS kernels and shows the user functions.
            - **Function: "name"**
                - **Function Stalls**: Shows the different type stalls experienced by the process. It contains External Memory and Internal-Kernel Pipe stalls. The number of rows is dynamically incremented to accommodate the visualization of any concurrent execution.
                - **Intra-Kernel Dataflow**: FIFO activity internal to the kernel.

  - **Function I/O**: Actual interface signals.
- **Function: “name”**
- **Function: “name”**

Data Interpretation Live Waveform

Below is a snapshot of the Live Waveform view while running hardware emulation.



The Live Waveform view is organized hierarchically for easy navigation. Below are the hierarchy tree and descriptions.

***Note*:** As the Live Waveform is presented only as part of the actual hardware simulation run (xsim), you can annotate extra signals and internals of the RTL to the same view. Also, all grouped and combined groups can be expanded all the way to the actual contributing signals.

- **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.
    - **DDR Bank 0**
      - **Read Address**
      - **Read Data**
      - **Write Address**
      - **Write Data**
    - **DDR Bank 1**
    - **DDR Bank 2**
    - **DDR Bank 3**
  - **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 HLS to inform you when a portion of the circuit is stalling because of external memory accesses, internal streams (i.e., dataflow), or external streams (i.e., 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 would be 9%.
    - **Data Transfers**: This shows the read/write data transfer accesses originating from each Master AXI port of the compute unit to the DDR.
      - **m_axi_<bundle name>**
        - **Read Address**
        - **Read Data**
        - **Write Address**
        - **Write Data**
      - **m_axi_<bundle name>**
    - **User Functions**: This information is available for the HLS kernels and shows the user functions.
      - **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.
    - **Row 0**
    - **Row 1**
    - **Row 2**
  - **StallNoContinue**: This is a stall signal that tells if there were any output stalls experienced by the dataflow processes (function is done, but it has not received a continue from the adjacent dataflow process).
  - **RTL Signals**: These are the underlying RTL control signals that were used to interpret the above transaction view of the dataflow process.
- **Function Stalls**: Shows the different types of stalls experienced by the process.
  - **External Memory**: Stalls experienced while accessing the DDR memory.
  - **Internal-Kernel Pipe**: If the compute units communicated between each other through pipes, then this will show the related stalls.
- **Intra-Kernel Dataflow**: FIFO activity internal to the kernel.
- **Function I/O**: Actual interface signals.

- **Function: “name”**
- **Function: “name”**

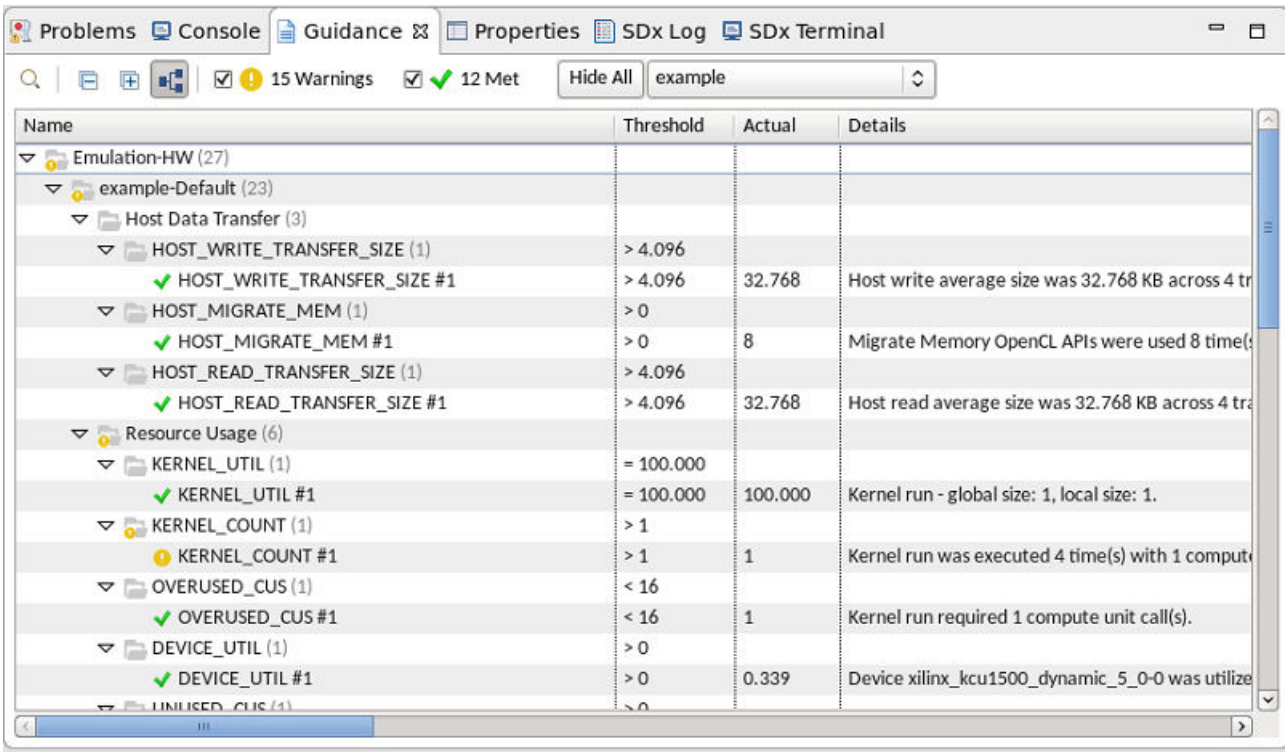
Guidance

The Guidance view is designed to provide feedback to users throughout the SDAccel™ execution flow. It presents in a single location all issues encountered from building the actual design all the way through runtime analysis.

It is crucial to understand that the Guidance Viewer is intended to help you to identify potential issues in the design. These issues might be source code related or due to missed tool optimizations. Also, the rules are generic rules based on experiences on a vast set of reference designs. Nevertheless, these rules might not be applicable for a specific design. Therefore, it is up to you to understand the specific guidance rules and take appropriate action based on your specific algorithm and requirements.

GUI Flow

The Guidance view is automatically populated and displayed in the lower central tab view. After running hardware emulation the Guidance view might look like the following:



In addition, to simply visualizing the guidance information, the GUI flow allows you to search/filter the Guidance view to locate specific guidance rule entries. It is also possible to collapse or expand the tree view or even suppress the hierarchical tree representation and visualize a condensed representation of the guidance rules. Finally, it is possible to select what is shown in the Guidance view. You can enable/disable the visualization of warnings as well as met rules, and 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, use the command **Window** → **Preferences**, select the category **Xilinx SDx** → **Guidance**, and uncheck **Group guidance rule checks by project**.

Command Line

The Guidance data is best analyzed through the GUI, which consolidates all guidance information for the flow. Nevertheless, the tool automatically generates HTML files containing the guidance information. As guidance information is generated throughout the tool flow, several guidance files are generated. The simplest way to locate the guidance reports is to search for the `guidance.html` files.

```
find . -name "*guidance.html" -print
```

This command lists all guidance files generated, which can be opened with any web-browser.

Data Interpretation

The Guidance view places each entry in a separate row. Each row may contain the name of the guidance rule, threshold value and actual value, followed by 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 are 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.

Below is a list of all fields and their purpose as included in the HTML guidance reports.

Id

Each guidance rule is assigned a unique **Id**. Use this id to uniquely identify a specific message from the guidance report.

Name

The **Name** column displays a mnemonic name uniquely identifying the guidance rule. These names are designed to assist in memorizing specific guidance rules in the view.

Severity

The **Severity** column allows the easy identification of the importance of a guidance rule.

Full Name

The **Full Name** provides a less cryptic name compared to the mnemonic name in the Name column.

Categories

Most messages are grouped within different categories. This allows the GUI to display groups of messages within logical categories under common tree nodes in the Guidance view.

Threshold

The **Threshold** column displays an expected threshold value, which determines whether or not a rule is met. The threshold values are determined from many applications that follow good design and coding practices.

Actual

The **Actual** column displays the values actually encountered on the specific design. This value is compared against the expected value to see if the rule is met.

Details

The **Details** column provides a brief but specific message describing the specifics of the current rule.

Resolution

The **Resolution** column provides a pointer to common ways the model source code or tool transformations can be modified in order to meet the current rule. Clicking the link brings up a pop-up window or the documentation with tips and code snippets that you can apply to the specific issue.

Utilizing Implementation Tools

Exploring Kernel Optimizations Using Vivado HLS

All kernel optimizations, using OpenCL™ or C/C++, can be performed from within the SDAccel™ Environment. The primary performance optimizations, such as those discussed in this chapter (pipelining function and loops, applying dataflow to enable greater concurrency between functions and loops, unrolling loops, etc.), are performed by the Xilinx® FPGA design tool Vivado® HLS.

Vivado HLS is called automatically by the SDAccel Environment, however you also have the option of launching Vivado HLS directly from within the SDAccel Environment. Using Vivado HLS in standalone mode enables the following enhancements to the optimization methodology:

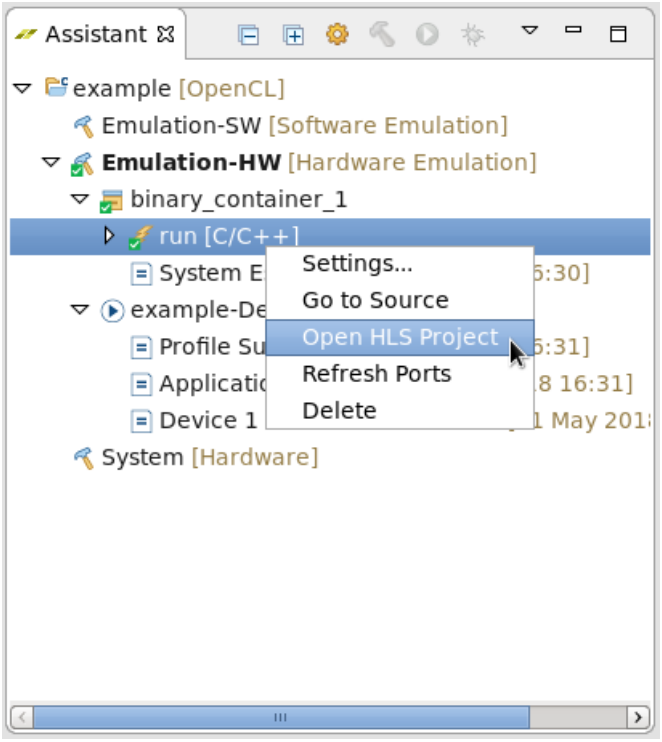
- Focusing solely on the kernel optimization, there is no requirement to execute emulation.

- The ability to create multiple solutions, compare their results, and explore the solution space to find the most optimum design.
- The ability to use the interactive Analysis Perspective to analyze the design performance.

IMPORTANT!: *Only the kernel source code is incorporated back into the SDAccel Environment. After exploring the optimization space, ensure that all optimizations are applied to the kernel source code as OpenCL attributes or C/C++ pragmas.*

To open Vivado HLS in standalone mode, from the **Assistant** tab, right-click the hardware function object and select **Open HLS Project** (see image below).

Figure 10: **Open HLS Project**



Vivado Backend Optimization

SDx™ provides a smooth flow from an OpenCL™/C/C++ model all the way to an FPGA accelerated implementation. In most cases, this flow completely abstracts away the underlying fact that the programmable region in the FPGA is configured to implement the kernel functionality. This fully isolates you from typical hardware constraints such as routing delays and kernel placement. However, in some cases these concerns will have to be looked at especially when large designs are to be implemented. Towards this end, SDx allows you to fully control the Vivado® Design Suite backend tool.

The SDAccel™ Environment calls Vivado to automatically run RTL Kernel synthesis and implementation. You also have the option of launching Vivado directly from within the SDAccel Environment. When invoking Vivado in standalone mode in the SDAccel Environment, you can open the Vivado Synthesis project or the Vivado Implementation project.

The Vivado project can be opened in the SDAccel Environment after the build with build configuration set to **System** has completed. To open Vivado IDE in standalone mode, from the **Xilinx** drop-down menu, select **Vivado Integration** and **Open Vivado Project**. Choose between the Vivado projects for synthesis and implementation and click **OK**.

Using Vivado in standalone mode enables the exploration of various Vivado synthesis and implementation options for further optimizing the kernel for performance and area. Familiarity with the Vivado tool suite is recommended to make the most use of these parameters.

***Note*:** The optimization switches applied in the standalone Vivado project are not automatically incorporated back into the SDAccel Environment. After exploring the optimization space, ensure that all optimization parameters are passed to the SDAccel Environment using the `--xp` option for `xocc`. For example:

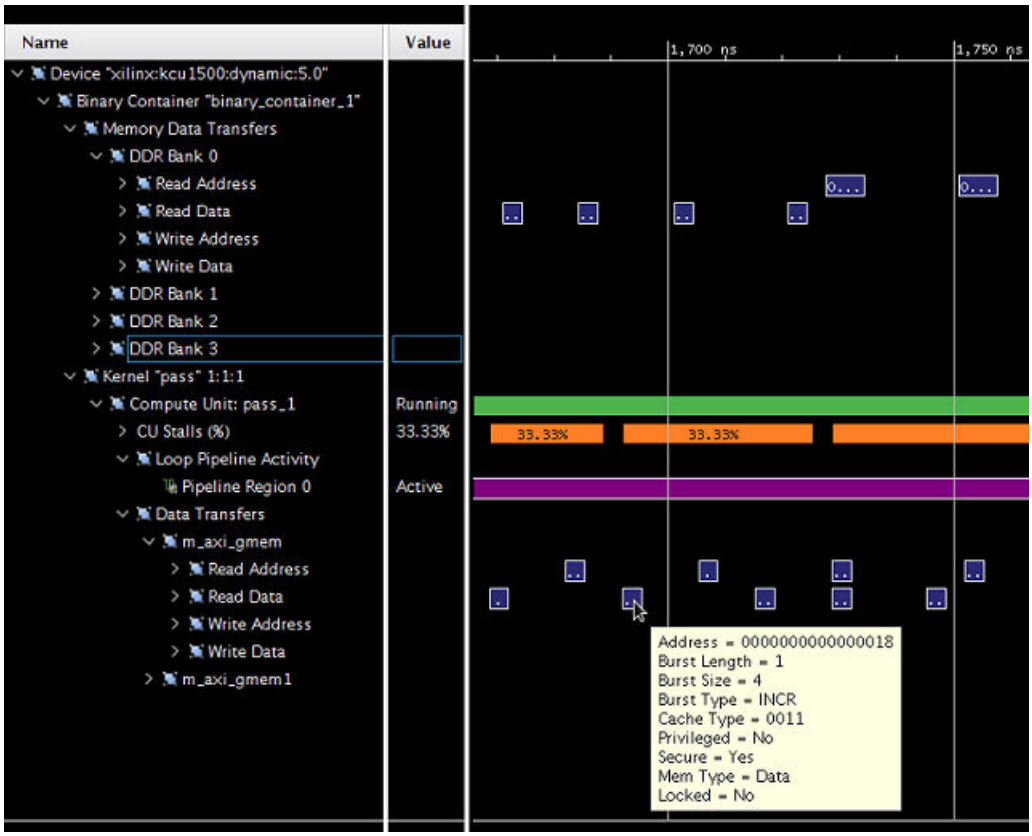
```
 --xp "vivado_prop:run.impl_1.{STEPS.PLACE_DESIGN.ARGS.TCL.POST}={<File
and path>}"
```

Interface Optimization

This section focuses on the interface optimization. It looks at the attributes of the implemented interface and suggests corrective actions to improve overall performance.

Interface Attributes (Detailed Kernel Trace)

The detailed kernel trace provides easy access to the AXI transactions and their properties. The AXI transactions are presented for the DDR side (Memory Data Transfers) 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.



Most interesting with respect to performance are the fields:

- **Burst Length:** Which describes how many packages are sent within one transaction
- **Burst Size:** Which 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. It should be noted, that SDx™ 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.

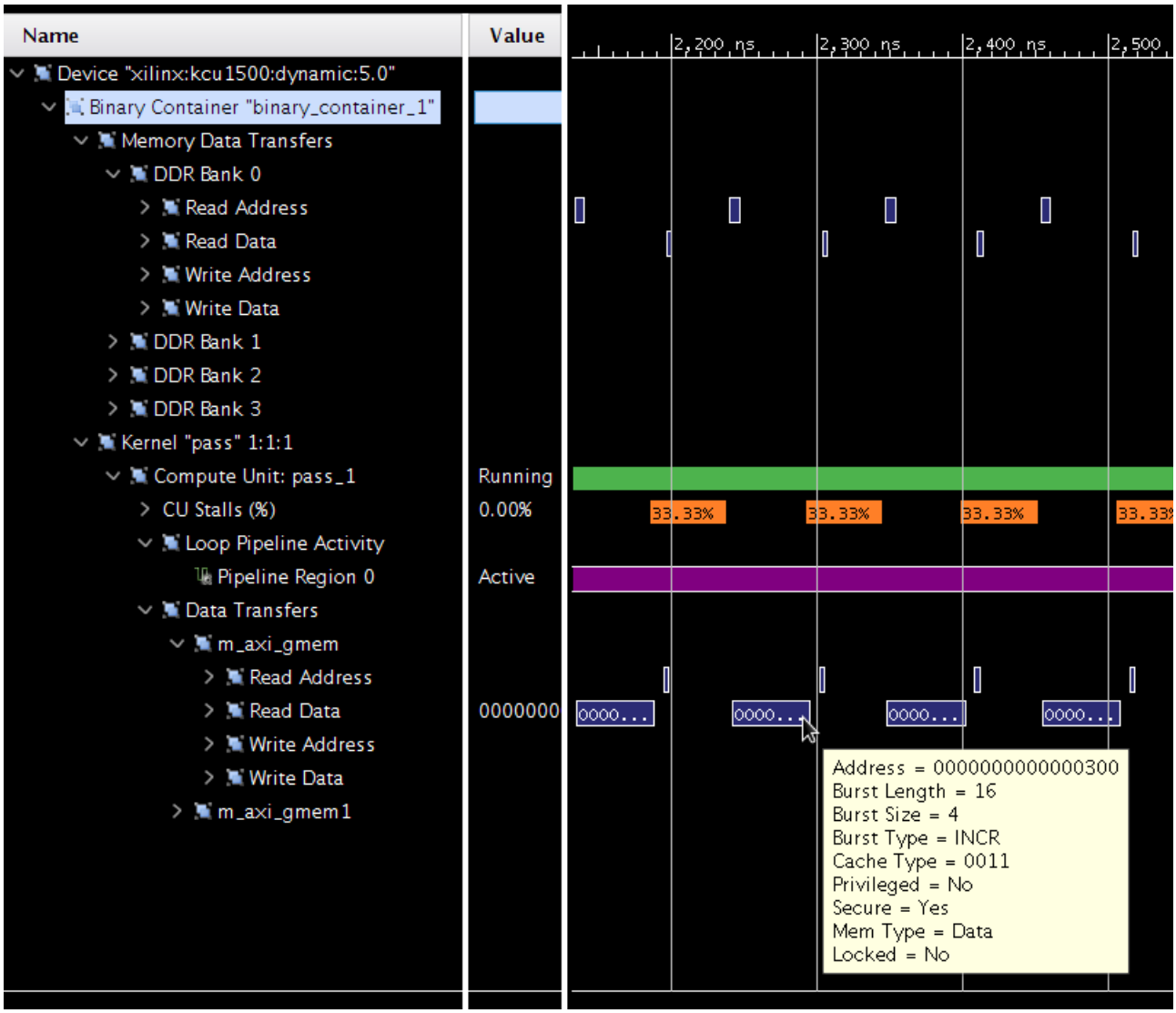
Small Burst Lengths as well as Burst Sizes considerably less than 512 Bits are therefore good opportunities to optimize interface performance. The following sections show improved implementations:

- Using Burst Data Transfers
- Utilize Full User Data Width of Memory Bandwidth

Using Burst Data Transfers

Transferring data in bursts hides the memory access latency as well as improves bandwidth utilization and efficiency of the memory controller. It is recommended to infer burst transfers from successive requests of data from consecutive address locations. Refer to "Inferring Burst Transfer from/to Global Memory" in *SDAccel Environment Programmers Guide* (UG1277) for more details.

If burst data transfers occur, the detailed kernel trace will reflect the higher burst rate as a larger burst length numbers



In this figure, it is also possible to observe that the Memory Data transfers following the AXI interconnect are actually implemented rather differently (shorter transaction time). If we would hover over these transactions, we would actually be able to see that the AXI interconnect has packed the 16x4 Byte transaction into a single package transaction of 1x64 Bytes. This effectively utilizes the full AXI bandwidth which is even more favorable. The next section illustrates focuses on this optimization technique in more detail.

Burst inference is heavily dependent on coding style and access pattern. Refer to the coding style guide to avoid potential modeling pitfalls. However, there is a general rule which when followed can ease burst detection and improve performance. This is done by isolating Data Transfer and Computation as shown in the following code snippet:

```
void kernel(T in[1024], T out[1024]) {
    T tmpIn[1024];
    T tmpOu[1024]t;
    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 actual computation is implemented by the function `process` working on the internal variables `tmpIn` and `tmpOut` and the function `write` takes the produced output and writes to the AXI output.

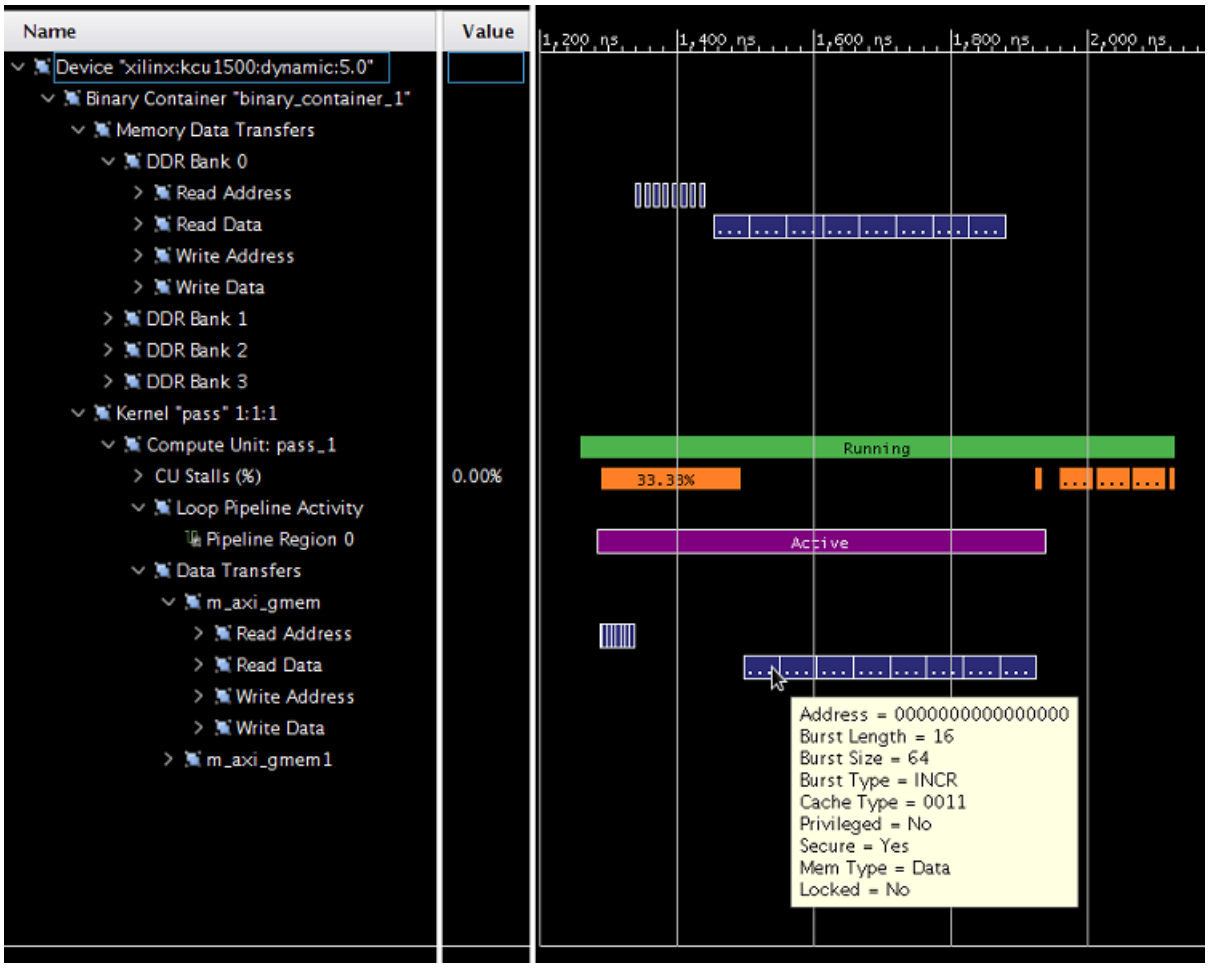
The isolation of the read and write function from the computation results in:

- Simple control structures (loops) in the read/write function which makes burst detection simpler.
- The isolation of the computational function away from the AXI interfaces, simplifies potential kernel optimization. See the Kernel Optimization chapter for more information.
- The internal variables are mapped to on-chip memory which allow faster access compared to AXI transactions. Acceleration platforms supported in SDAccel™ environment can have as much as 10MB on-chip memories that can be used as pipes, local 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 SDAccel™ compiler based on the data types of the kernel arguments. To maximize the data throughput, Xilinx recommends that you choose data types that map to the full data width on the memory controller. The memory controller in all supported acceleration cards supports 512-bit user interface, which can be mapped to OpenCL™ vector data types such as `int16` or C/C++ arbitrary precision data type `ap_int<512>`.

As shown on the following image, it is possible to observe burst AXI transactions (Burst Length 16) and a 512 bit package size (Burst Size 64 Bytes).



This example shows good interface configuration as it maximizes AXI data width as well as it shows 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. Xilinx recommends to use simple structs for kernel arguments that can be always packed to 32-bit boundary. Refer to the *Custom Data Type Example* in *kernel_to_gmem* category at Xilinx On-boarding Example GitHub for the recommended way to use structs.

OpenCL Attributes

OpenCL provides attributes to support a more automatic approach to incrementing AXI data width utilization. The change of the interface data types as stated above is supported in OpenCL as well but will require the same code changes as C/C++ to the algorithm to accommodate the larger input vector.

To eliminate manual code modifications, the following OpenCL attributes are interpreted to perform data path widening and vectorization of the algorithm. A detailed description can be found in the *SDx Pragma Reference Guide* (UG1253).

- `vec_type_hint`
- `reqd_work_group_size`
- `xcl_zero_global_work_offset`

Let us examine the combined functionality on the following case:

```
__attribute__((reqd_work_group_size(64, 1, 1)))
__attribute__((vec_type_hint(int)))
__attribute__((xcl_zero_global_work_offset))
__kernel void vector_add(__global int* c, __global const int* a, __global
const int* b) {
    size_t idx = get_global_id(0);
    c[idx] = a[idx] + b[idx];
}
```

Here, the hard coded interface is a 32-bit wide data path `(int *c, int* a, int *b)`, which drastically limits the memory throughput if implemented directly. However, the automatic widening and transformation is applied based on the values of the three attributes.

- The `__attribute__((vec_type_hint(int)))` declares that `int` is the main type used for computation and memory transfer (32 bit). This knowledge is used to calculate the vectorization/widening factor based on the target band width of the AXI interface (512 bits). In this example the factor would be 16 = 512 bits / 32 bit. This implies that in theory, 16 values could be processed if vectorization can be applied.
- The `__attribute__((reqd_work_group_size(X, Y, Z)))` where `X`, `Y`, and `Z` are positive constants, defines the total number of work items. `X*Y*Z` is the maximum number of work items therefore defining the maximum possible vectorization factor which would saturate the memory bandwidth. In this example, the total number of work items is `64*1*1=64`.

The actual vectorization factor to be applied will be the greatest common divider of the vectorization factor defined by the actual coded type or the vec_type_hint, and the maximum possible vectorization factor defined through `reqd_work_group_size`.

The quotient of maximum possible vectorization factor divided by the actual vectorization factor provides the remaining loop count of the OpenCL description. As this loop is pipelined, it can be advantageous to have several remaining loop iterations to take advantage of a pipelined implementation. This is especially true if the vectorized OpenCL code has long latency.

There is one optional parameter that is highly recommended to be specified for performance optimization on OpenCL interfaces.

- The `__attribute__((xcl_zero_global_work_offset))` instructs the compiler that no global offset parameter is used at runtime, and all accesses are aligned. This gives the compiler valuable information with regard to alignment of the work groups, which in turn usually propagate to the alignment of the memory accesses (less hardware).

It should be noted, that the application of these transformations changes the actual design to be synthesized. Partially unrolled loops require reshaping of local arrays in which data is stored. This usually behaves nicely, but can interact poorly in rare situations.

For example:

- For partitioned arrays, when the partition factor is not divisible by the unrolling/vectorization factor.
    - The resulting access requires a lot of multiplexers and will create a difficult problem for the scheduler (might severely increase memory usage and compilation time), Xilinx recommends that you use partitioning factors that are powers of two (as the vectorization factor is always a power of two).
- If the loop being vectorized has an unrelated resource constraint, the scheduler complains about II not being met.
    - This is not necessarily correlated with a loss of performance (usually it is still performing better) since the II is computed on the unrolled loop (which has therefore a multiplied throughput for each iteration).
    - The scheduler informs you of the possible resources constraints and resolving those will further improve the performance.
    - Note that a common occurrence is that a local array does not get automatically reshaped (usually because it is accessed in a later section of the code in non-vectorizable way).

Reducing Kernel to Kernel Communication Latency with OpenCL Pipes

The OpenCL™ 2.0 specification introduces a new memory object called a pipe. A pipe stores data organized as a FIFO. Pipe objects can only be accessed using built-in functions that read from and write to a pipe. Pipe objects are not accessible from the host. Pipes can be used to stream data from one kernel to another inside the FPGA without having to use the external memory, which greatly improves the overall system latency.

In the SDAccel development environment, pipes must be statically defined outside of all kernel functions. Dynamic pipe allocation using the `OpenCL 2.x clCreatePipe` API is not currently supported. The depth of a pipe must be specified by using the `xcl_reqd_pipe_depth` attribute in the pipe declaration. The valid depth values are 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768.

```
pipe int p0 __attribute__((xcl_reqd_pipe_depth(32)));
```

A given pipe, can have one and only one producer and consumer in different kernels.

Pipes can be accessed using standard OpenCL `read_pipe()` and `write_pipe()` built-in functions in non-blocking mode or using Xilinx® extended `read_pipe_block()` and `write_pipe_block()` functions in blocking mode. The status of pipes can be queried using OpenCL `get_pipe_num_packets()` and `get_pipe_max_packets()` built-in functions. See the OpenCL C Specification, Version 2.0 from Khronos Group for more details on these built-in functions.

The following are the function signatures for the currently supported pipe functions, where `gentype` indicates the built-in OpenCL C scalar integer or floating-point data types.

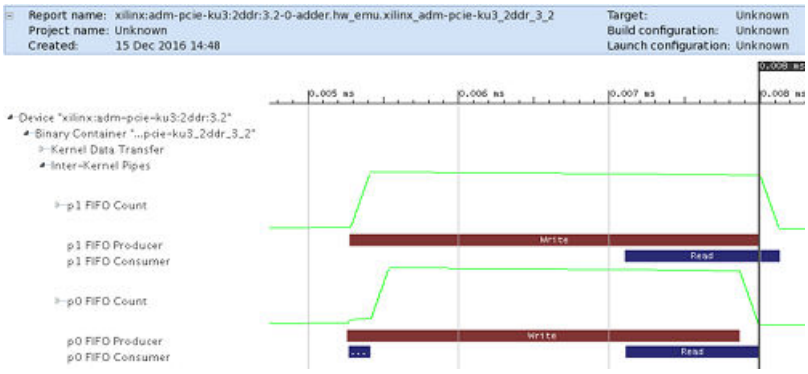
```
int read_pipe_block (pipe gentype p, gentype *ptr)
int write_pipe_block (pipe gentype p, const gentype *ptr)
```

The following is the "Blocking Pipes Example" from Xilinx On-boarding Example GitHub that use pipes to pass data from one processing stage to the next using `blocking read_pipe_block()` and `write_pipe_block()` functions:

```
pipe int p0 __attribute__((xcl_reqd_pipe_depth(32)));
pipe int p1 __attribute__((xcl_reqd_pipe_depth(32)));
// Input Stage Kernel : Read Data from Global Memory and write into Pipe P0
kernel __attribute__ ((reqd_work_group_size(1, 1, 1)))
void input_stage(__global int *input, int size)
{
    __attribute__((xcl_pipeline_loop))
    mem_rd: for (int i = 0 ; i < size ; i++)
    {
        //blocking Write command to pipe P0
        write_pipe_block(p0, &input[i]);
    }
```

```
}
// Adder Stage Kernel: Read Input data from Pipe P0 and write the result
// into Pipe P1
kernel __attribute__ ((reqd_work_group_size(1, 1, 1)))
void adder_stage(int inc, int size)
{
    __attribute__((xcl_pipeline_loop))
    execute: for(int i = 0 ; i < size ;  i++)
    {
        int input_data, output_data;
        //blocking read command to Pipe P0
        read_pipe_block(p0, &input_data);
        output_data = input_data + inc;
        //blocking write command to Pipe P1
        write_pipe_block(p1, &output_data);
    }
}
// Output Stage Kernel: Read result from Pipe P1 and write the result to
Global
// Memory
kernel __attribute__ ((reqd_work_group_size(1, 1, 1)))
void output_stage(__global int *output, int size)
{
    __attribute__((xcl_pipeline_loop))
    mem_wr: for (int i = 0 ; i < size ; i++)
    {
        //blocking read command to Pipe P1
        read_pipe_block(p1, &output[i]);
    }
}
```

The device traceline view shows the detailed activities and stalls on the OpenCL pipes after hardware emulation is run. This info can be used to choose the correct FIFO sizes to achieve the optimal application area and performance.



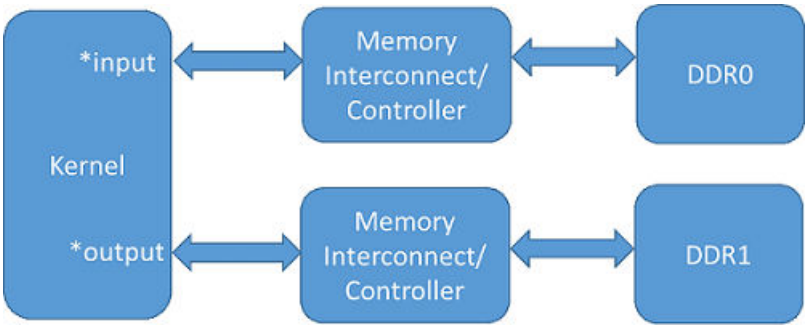
Using Multiple DDR Banks

Acceleration cards supported in SDAccel™ Environment provide 1, 2 or 4 DDR banks and up to 80GB/s raw DDR bandwidth. For kernels moving large amount of data between the FPGA and the DDR, Xilinx recommends that you direct the SDAccel compiler and runtime library to use multiple DDR banks.

To take advantage of multiple DDR banks, you need to assign CL memory buffers to different banks in the host code as well as configure XCL binary file to match the bank assignment in `xocc` command line.

The block diagram shows the *Global Memory Two Banks Example* in "kernel_to_gmem" category on Xilinx On-boarding Example GitHub that connects the input pointer to DDR bank 0 and output pointer to DDR bank 1.

Figure 11: **Global Memory Two Banks Example**



Assigning DDR Bank in Host Code

Bank assignment in host code is supported by Xilinx® vendor extension. The following code snippet shows the header file required as well as assigning input and output buffers to DDR bank 0 and bank 1 respectively:

```
#include <CL/cl_ext.h>
…
int main(int argc, char** argv)
{
…
    cl_mem_ext_ptr_t inExt, outExt;  // Declaring two extensions for both
buffers
    inExt.flags  = XCL_MEM_DDR_BANK0; // Specify Bank0 Memory for input
memory
    outExt.flags = XCL_MEM_DDR_BANK1; // Specify Bank1 Memory for output
Memory
    inExt.obj = 0   ; outExt.obj = 0; // Setting Obj and Param to Zero
    inExt.param = 0 ; outExt.param = 0;

    int err;
    //Allocate Buffer in Bank0 of Global Memory for Input Image using
Xilinx Extension
    cl_mem buffer_inImage = clCreateBuffer(world.context, CL_MEM_READ_ONLY
| CL_MEM_EXT_PTR_XILINX,
            image_size_bytes, &inExt, &err);
    if (err != CL_SUCCESS){
        std::cout << "Error: Failed to allocate device Memory" <<
std::endl;
        return EXIT_FAILURE;
    }
    //Allocate Buffer in Bank1 of Global Memory for Input Image using
Xilinx Extension
```

```
    cl_mem buffer_outImage = clCreateBuffer(world.context,
CL_MEM_WRITE_ONLY | CL_MEM_EXT_PTR_XILINX,
            image_size_bytes, &outExt, NULL);
    if (err != CL_SUCCESS){
        std::cout << "Error: Failed to allocate device Memory" <<
std::endl;
        return EXIT_FAILURE;
    }
…
}
```

`cl_mem_ext_ptr_t` is a `struct` as defined below:

```
typedef struct{
    unsigned flags;
    void *obj;
    void *param;
  } cl_mem_ext_ptr_t;
```

- Valid values for `flags` are XCL_MEM_DDR_BANK0, XCL_MEM_DDR_BANK1, XCL_MEM_DDR_BANK2, and XCL_MEM_DDR_BANK3.
- `obj` is the pointer to the associated host memory allocated for the CL memory buffer only if `CL_MEM_USE_HOST_PTR` flag is passed to `clCreateBuffer` API, otherwise set it to NULL.
- `param` is reserved for future use. Always assign it to 0 or NULL.

Assigning DDR Bank for Kernel Code

A kernel needs to have multiple AXI4 MM interfaces before it can be connected to multiple DDR banks.

Note: The `map_connect` option is deprecated and has been replaced with the `--sp` option.

- For OpenCL™ kernel, `--max_memory_ports` option is required to generate one AXI MM interface for each global pointer on the kernel argument. The AXI MM interface name is based on the order of the global pointers on the argument list.

  ```
  __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) {
    ...
  }
  ```

  This code is lifted from the example *Global Memory Two Banks (CL)* in the *Kernel To Global Memory Examples* category from the Xilinx On-boarding Example GitHub. In this example, the first global pointer `input` is assigned an AXI MM name `M_AXI_GMEM0` and the second global pointer `output` is assigned a name `M_AXI_GMEM1`. The AXI MM interface names are then needed in the `--sp` option.

- For C/C++ kernel, multiple AXI4 interfaces are generated by specifying different "bundle" names in the interface pragma for different global pointers. Below is a code snippet from the Global Memory Two Banks C Example that assigns the `input` pointer to the bundle `gmem0` and the `output` pointer to the bundle `gmem1`. Note that 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`.

```
#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 `--sp` option.

Once multiple AXI4 interfaces are created, they are connected to DDR banks using the `--sp` option. The `--sp` option value is in the format of `<kernel_instance_name>.<AXI_IF_name>:<DDR_bank_name>`. Valid DDR bank names for the `--sp` option are bank0, bank1, bank2, and bank3 for DDR banks 0, 1, 2, 3 respectively.

Below is the command line example that connects the input pointer (`M_AXI_GMEM0`) to DDR bank 0 and the output pointer (`M_AXI_GMEM1`) to DDR bank 1:

```
xocc --max_memory_ports apply_watermark --sp
apply_watermark_1.m_axi_gmem0:bank0 --sp
apply_watermark_1.m_axi_gmem1:bank1
```

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

Figure 12: **Device Hardware Transaction View Transactions on DDR Bank**

Kernel Optimization

One of the key advantages of an FPGA is its flexibility and capacity to create customized designs specifically for your algorithm. This enables various implementation choices to trade off algorithm throughput vs. power consumption. The downside of creating custom logic is that the design needs to go through the traditional FPGA design flow.

The following guidelines help manage the design complexity and achieve the desired design goals.

Optimizing Computational Parallelism

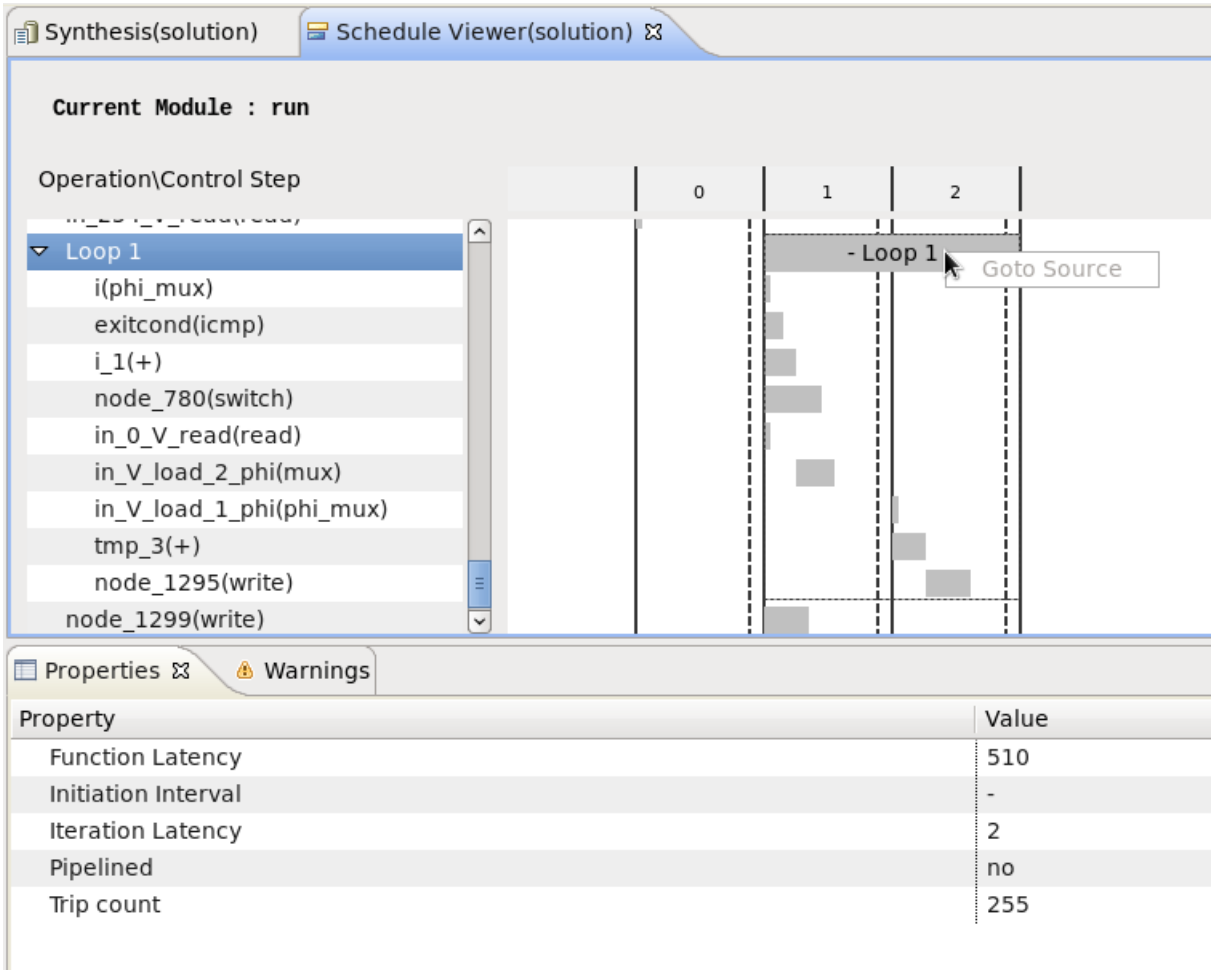
By default, C/C++ does not model computational parallelism, as it always executes any algorithm sequentially. On the other hand, OpenCL™ does model computational parallelism with respect to work groups, but it does not use any additional parallelism within the algorithm description. However, especially when it comes to fully configurable computational engines like FPGAs, it allows more freedom to exploit this computational parallelism.

Loop Parallelism

Loops are the basic C/C++/OpenCL™ method of representing repetitive algorithmic code. The following example illustrates various implementation aspects of a loop structure:

```
for(int i = 0; i<255; i++) {
  out[i] = in[i]+in[i+1];
}
out[255] = in[255];
```

This code iterates over an array of values and adds consecutive values, except if the last value. If this loop is implemented as is, each loop iteration requires two cycles for implementation, which results in a total of 510 cycles for implementation. This can be analyzed in detail through the Schedule Viewer in the HLS Project :



Or in terms of total numbers and latency through the synthesis results:

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	5.00	3.123	0.62

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
511	511	511	511	none

Detail

Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	47
FIFO	-	-	-	-
Instance	-	-	0	1362
Memory	-	-	-	-
Multiplexer	-	-	-	1145
Register	-	-	52	-
Total	0	0	52	2554
Available	4320	5520	1326720	663360
Available SLR	2160	2760	663360	331680
Utilization (%)	0	0	~0	~0
Utilization SLR (%)	0	0	~0	~0

The key numbers here are the latency numbers and total LUT usage. For example, depending on the configuration, you could get latency of 511 and total LUT usage of 47. As you will see, these values can widely vary based on the implementation choices. While this implementation will require very little area, it results in a rather lengthy latency.

Unrolling Loops

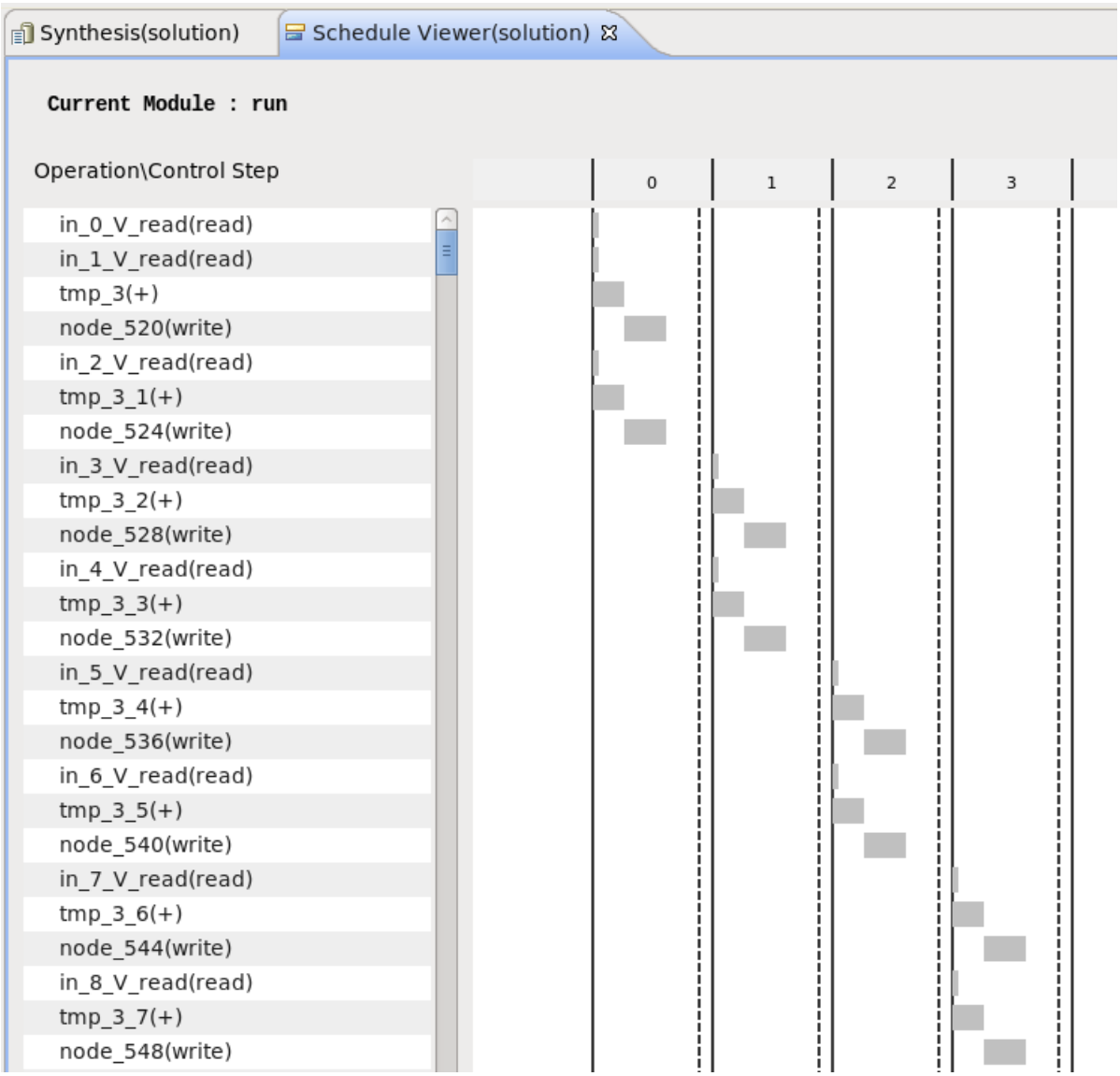
Unrolling a loop enables the full parallelism of the model to be exploited. To do this, you can simply mark a loop to be unrolled and the tool will create the implementation with the most parallelism possible. To mark a loop to be unrolled, an OpenCL loop can be marked with the Unroll attribute:

```
__attribute__((opencl_unroll_hint))
```

or a C/C++ loop can utilize the unroll pragma:

```
#pragma HLS UNROLL
```

When applied to our specific example, the Schedule View in the HLS Project will be:



With an estimated performance of:

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	5.00	3.123	0.62

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
127	127	127	127	none

Detail

Instance

Loop

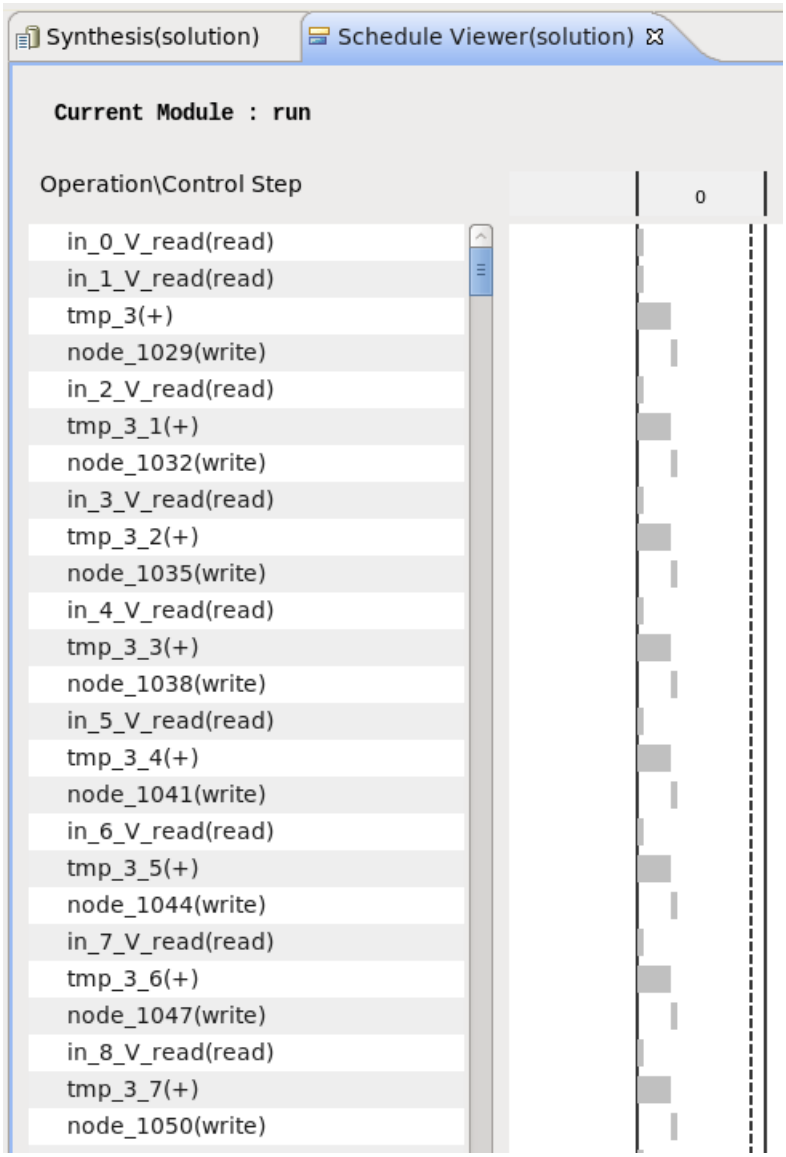
Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	4845
FIFO	-	-	-	-
Instance	-	-	-	-
Memory	-	-	-	-
Multiplexer	-	-	-	2905
Register	-	-	129	-
Total	0	0	129	7750
Available	4320	5520	1326720	663360
Available SLR	2160	2760	663360	331680
Utilization (%)	0	0	~0	1
Utilization SLR (%)	0	0	~0	2

As you can see, the total latency was considerably improved to now be 127 cycles and as expected the computational hardware was increased to 4845 LUTs, to perform the same computation in parallel.

However, if you analyze the for-loop, you might ask why this algorithm cannot be implemented in a single cycle, as each addition is completely independent of the previous loop iteration. The reason is the memory interface to be utilized for the variable `out`. SDx™ uses dual port memory by default for an array. However, this implies that at most two values can be written to the memory per cycle. Thus to see a fully parallel implementation, you must specify that the variable `out` should be kept in registers. The results of this transformation can be observed in the following Schedule View:



The associated estimates are:

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	5.00	1.352	0.62

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
0	0	0	0	none

Detail

Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	4845
FIFO	-	-	-	-
Instance	-	-	-	-
Memory	-	-	-	-
Multiplexer	-	-	-	-
Register	-	-	-	-
Total	0	0	0	4845
Available	4320	5520	1326720	663360
Available SLR	2160	2760	663360	331680
Utilization (%)	0	0	0	~0
Utilization SLR (%)	0	0	0	1

As you can see, this code can be implemented as a combinatorial function requiring only a fraction of the cycle to complete.

Pipelining Loops

Pipelining loops allows you to overlap iterations of a loop in time. Allowing iterations to operate concurrently is often a good compromise, as resources can be shared between iterations (less utilization), while requiring less execution time compared to loops that are not unrolled.

Pipelining is enabled in C/C++ via the following pragma:

```
#pragma HLS PIPELINE
```

While OpenCL uses the following attribute:

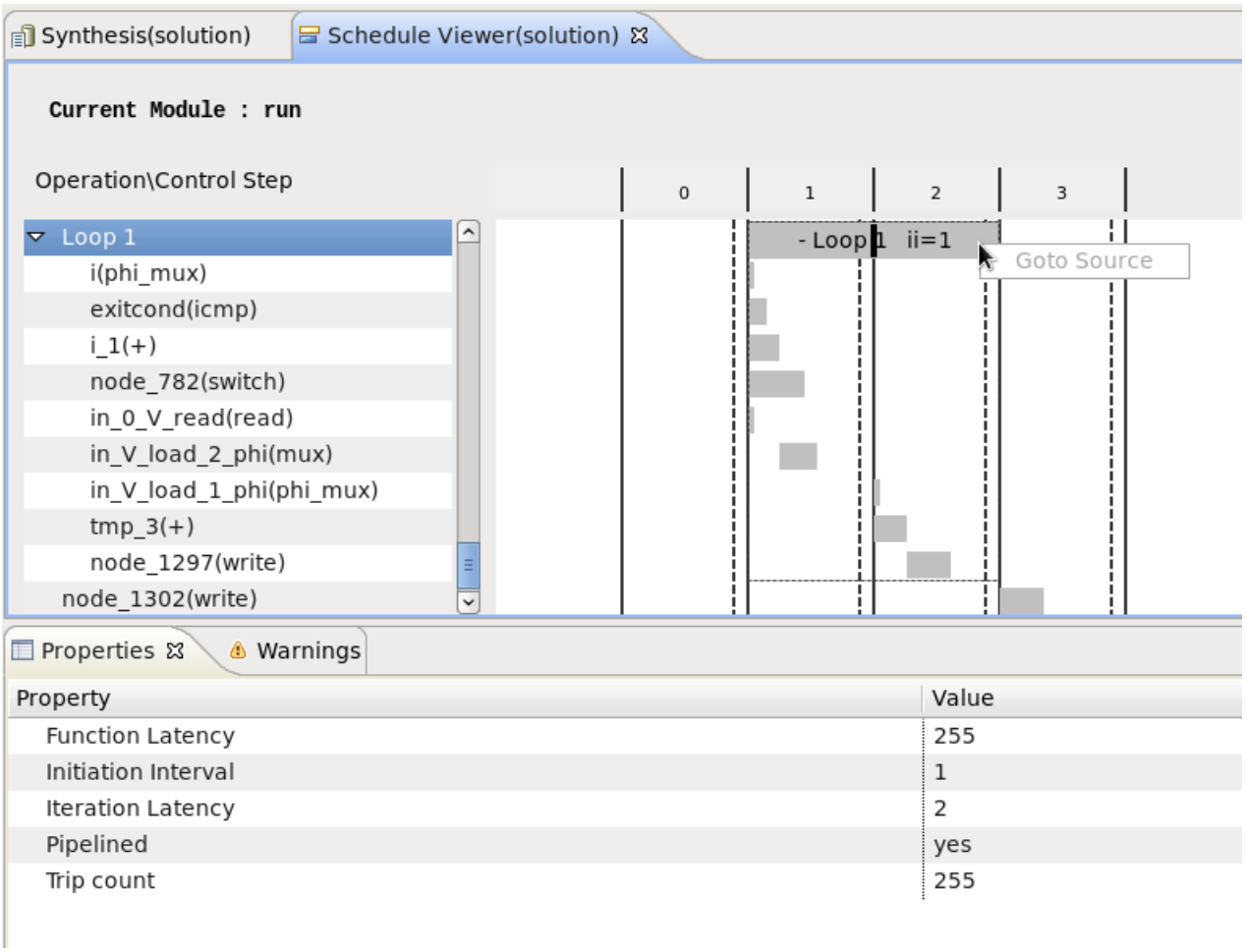
```
__attribute__((xcl_pipeline_loop))
```

Note that OpenCL has an additional way of specifying loop pipelining. This has to do with the fact that work item loops are not explicitly stated and pipelining these loops requires the attribute:

```
__attribute__((xcl_pipeline_workitems))
```

More details to any of these specifications are provided in the *SDx Pragma Reference Guide* (UG1253) and the *SDAccel Environment Programmers Guide* (UG1277).

In this example, the Schedule View in the HLS Project produces:



With the overall estimates being:

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	5.00	3.123	0.62

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
257	257	257	257	none

Detail

Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	53
FIFO	-	-	-	-
Instance	-	-	0	1362
Memory	-	-	-	-
Multiplexer	-	-	-	1173
Register	-	-	47	-
Total	0	0	47	2588
Available	4320	5520	1326720	663360
Available SLR	2160	2760	663360	331680
Utilization (%)	0	0	~0	~0
Utilization SLR (%)	0	0	~0	~0

Since each iteration of a loop consumes only two cycles of latency, there can only be a single iteration overlap. This enables the total latency to be cut into half compared to the original, resulting in 257 cycles of total latency. However, this reduction in latency was achieved by a marginal increment in resources when compared to unrolling.

In most cases loop pipelining by itself can improve overall performance. However, the effectiveness of the pipelining will depend on the structure of the loop. Some common limitations are:

- Resources with limited availability such as memory ports or process channels can limit the overlap of the iterations (Initiation Interval).
- Similarly, loop-carried dependencies such as those created by variables / conditions computed in one iteration affecting the next, might increase the initial interval of the pipeline.

These are reported by the tool during high-level synthesis and can be observed and examined in the Schedule Viewer. For best possible performance, the code might have to be modified to eliminate these limiting factors or the tool needs to be instructed to eliminate some dependency by restructuring the memory implementation of an array or breaking the dependencies all together.

Task Parallelism

Task parallelism allows you to take advantage of data flow parallelism. In contrast to loop parallelism, when task parallelism is deployed, full execution units (tasks) are allowed to operate in parallel taking advantage of extra buffering introduced between the tasks.

Look at the following example:

```
void run (ap_uint<16> in[1024],
      ap_uint<16> out[1024]
      ) {
  ap_uint<16> tmp[128];
  for(int i = 0; i<128; i++) {
    processA(&(in[i*128]), tmp);
    processB(tmp, &(out[i*128]));
  }
}
```

When this code is executed, the function `processA` and `processB` are executed sequentially 128 times in a row. Given the combined latency for `processA` and `processB` in the loop is 278, the total latency can be estimated as:

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	3.33	2.433	0.90

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
35585	35585	35585	35585	none

Detail

Instance

Loop

The extra cycle is due to loop setup and can be observed in the Schedule View.

Now, applying Task Parallelism is performed by adding in C/C++ models the pragma:

```
#pragma HLS DATAFLOW
```

And in OpenCL™ models the attribute:

```
__attribute__ ((xcl_dataflow))
```

Refer to *SDx Pragma Reference Guide* (UG1253) and *SDAccel Environment Programmers Guide* (UG1277) for more details regarding the specifics and limitations of these modifiers. As illustrated by the estimates in the HLS Report applying the transformation will considerably improve the overall performance effectively using a double buffer scheme between the tasks:

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	3.33	3.346	0.90

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
17931	17931	17931	17931	none

Detail

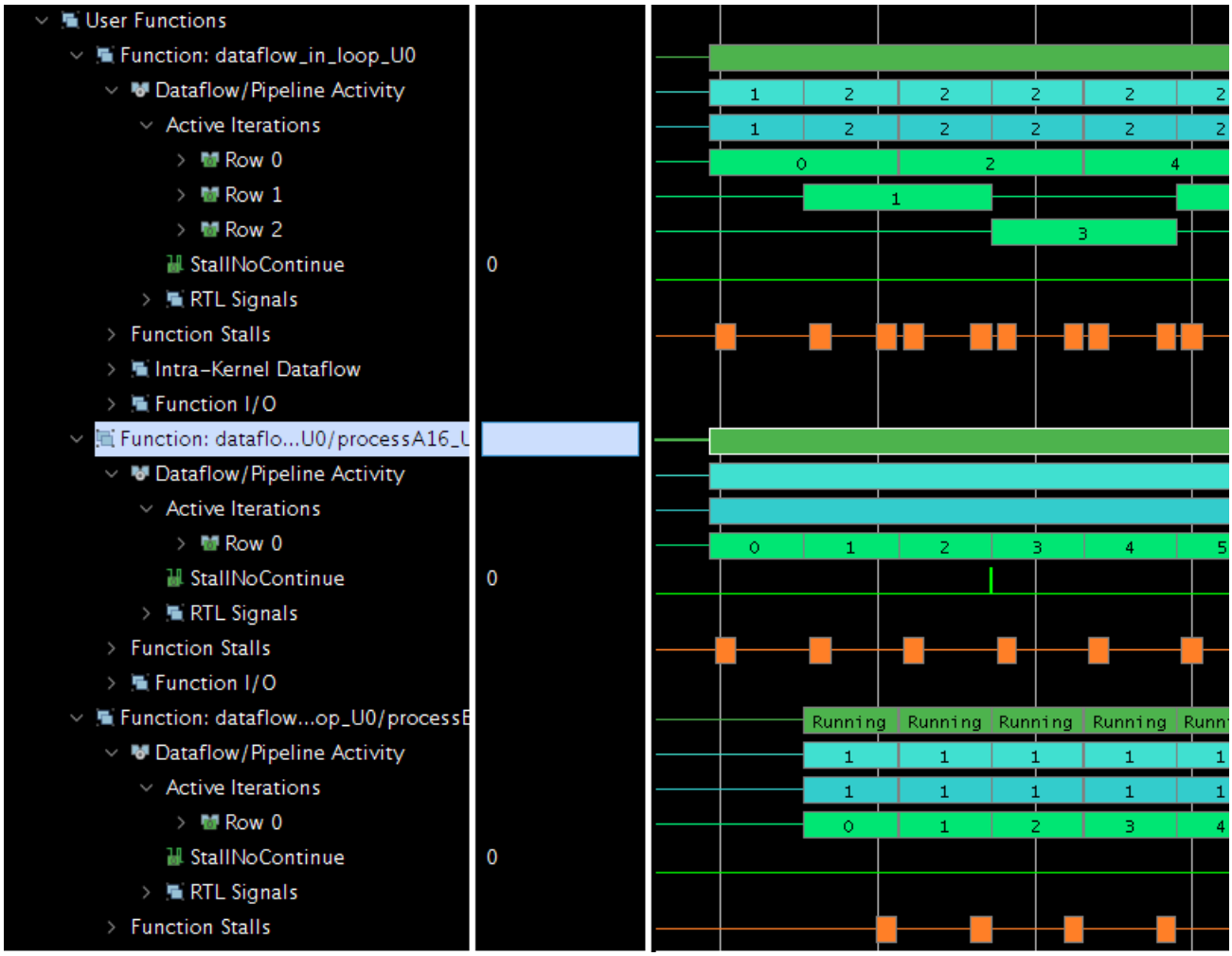
Instance

Loop

The overall latency of the design has almost halved in this case due to concurrent execution of the different tasks of the different iterations. Given the 139 cycles per processing function and the full overlap of the 128 iterations, this allows the total latency to be:

```
(1x only processA + 127x both processes + 1x only processB) * 139 cycles =
17931 cycles
```

Using task parallelism is a very powerful way improve performance when it comes to implementation. However, the effectiveness of applying the DATAFLOW pragma to specific and arbitrary piece of code might vary vastly. The coding guidelines for applying DATAFLOW effectively are provided in *SDx Pragma Reference Guide* (UG1253) and *SDAccel Environment Programmers Guide* (UG1277). However, it is often necessary to actually look at the execution pattern of the individual tasks to understand the final implementation of the DATAFLOW pragma. Towards that end, the SDAccel environment provides the Detailed Kernel Trace, which nicely illustrates concurrent execution.



In this Detailed Kernel Trace, the tool displays the start of the dataflowed loop. It illustrates how processA is starting up right away with the beginning of the loop, while processB waits until the completion of the processA before it can start up its first iteration. However, while processB completes the first iteration of the loop, processA begins operating on the second iteration and so forth.

A more abstract representation of this information is presented in the Application Timeline (Host & Device) and Device Hardware Transaction View (device only during hardware emulation).

Optimizing Computational Units

Datawidth

One, if not the most important, aspect for performance is the datawidth required for the implementation. The tool propagates port widths throughout the algorithm. In some cases, especially when starting out with an algorithmic description, the C/C++/OpenCL™ code might only utilize large data types such as integers even at the ports of the design. However, as the algorithm gets mapped to a fully configurable implementation, smaller data types such as 10 or 12 bit might often suffice. Towards that end it is beneficial to check the size of basic operations in the HLS Synthesis report during optimization. In general, when SDx™ maps an algorithm onto the FPGA, much processing is required to comprehend the C/C++/OpenCL structure and extract operational dependencies. Therefore, to perform this mapping SDx generally partitions the source code into operational units which are than mapped onto the FPGA. Several aspects influence the number and size of these operational units (ops) as seen by the tool.

In the following table, the basic operations and their bitwidth are reported.

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	102
FIFO	-	-	-	-
Instance	-	-	-	-
Memory	0	-	24	12
Multiplexer	-	-	-	80
Register	-	-	51	-
Total	0	0	75	194
Available	4320	5520	1326720	663360
Available SLR	2160	2760	663360	331680
Utilization (%)	0	0	~0	~0
Utilization SLR (%)	0	0	~0	~0

Detail

Instance

DSP48

Memory

FIFO

Expression

Variable Name	Operation	DSP48E	FF	LUT	Bitwidth P0	Bitwidth P1
i_1_fu_124_p2	+	0	0	11	3	1
i_2_fu_148_p2	+	0	0	15	7	1
i_3_fu_179_p2	+	0	0	15	7	1
sum_i9_fu_194_p2	+	0	0	15	8	8
sum_i_fu_158_p2	+	0	0	15	8	8
exitcond_fu_118_p2	icmp	0	0	9	3	4
exitcond_i6_fu_173_p2	icmp	0	0	11	7	8
exitcond_i_fu_142_p2	icmp	0	0	11	7	8
Total	8	0	0	102	50	39

Multiplexer

Register

Simply look for typical Bitwidths of 16, 32, and 64 bits as commonly used in algorithmic descriptions and verify if the associated operation from the C/C++/OpenCL source actually requires to be this large. This can considerably improve the implementation of the algorithm, as smaller operations require less computation time.

Fixed Point Arithmetic

Some applications use floating point computation only because they are optimized for other hardware architecture. As explained in "Deep Learning with INT8 Optimization on Xilinx Devices," using fixed point arithmetic for applications like deep learning can save the power efficiency and area significantly while keeping the same level of accuracy. It is recommended to explore fixed point arithmetic for your application before committing to using floating point operations.

Macro Operations

It is sometimes advantageous to think about larger computational units. The tool will operate on the source code independently of the remaining source code, effectively mapping the algorithm without consideration of surrounding operations onto the FPGA. When applied, SDx keeps operational boundaries, effectively creating macro operations for specific code. This utilizes the following principles:

- Operational locality to the mapping process.
- Reduction in complexity for the heuristics.

This might create vastly different results when applied. In C/C++ macro operations are created with the help of

```
#pragma HLS inline off
```

While in OpenCL the same kind of macro operation can be generated by **not** specifying the attribute:

```
__attribute__((always_inline))
```

when defining a function.

Utilizing Optimized Libraries

The OpenCL Specification provides a wealth of math built-in functions. All math built-in functions with the `native_` prefix are mapped to one or more native device instructions and will typically have better performance compared to the corresponding functions (without the `native_` prefix). The accuracy and in some cases the input ranges of these functions is implementation-defined. In SDAccel™ environment these `native_` built-in functions use the equivalent functions in Vivado® HLS Math library, which are already optimized for Xilinx® FPGAs in terms of area and performance. Xilinx recommends that you use `native_` built-in functions or HLS Math library if the accuracy meets the application requirement.

Optimizing Memory Architecture

Memory architecture is a key aspect of implementation. Due to the limited access bandwidth, it can heavily impact the overall performance. Look at the following example:

```
void run (ap_uint<16> in[256][4],
          ap_uint<16> out[256]
         ) {
  ...
```

```
  ap_uint<16> inMem[256][4];
  ap_uint<16> outMem[256];

  ... Preprocess input to local memory

  for( int j=0; j<256; j++) {
    #pragma HLS PIPELINE OFF
    ap_uint<16> sum = 0;
    for( int i = 0; i<4; i++) {

      sum += inMem[j][i];
    }
    outMem[j] = sum;
  }

  ... Postprocess write local memory to output
}
```

This code adds the four values associated with the inner dimension of the two dimensional input array. If implemented without any additional modifications, it results in the following estimates:

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	3.33	2.433	0.90

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
5908	5908	5908	5908	none

Detail

Instance

Loop

	Latency			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- Loop 1	1034	1034	12	1	1	1024	yes
- Loop 2	4608	4608	18	-	-	256	no
+ Loop 2.1	16	16	4	-	-	4	no
- Loop 3	257	257	3	1	1	256	yes

The overall latency of 4608 (Loop 2) is due to 256 iterations of 18 cycles (16 cycles spent in the inner loop, plus the reset of sum, plus the output being written). This is can be observed in the Scheduler View in the HLS Project. The estimates become considerably better when unrolling the inner loop.

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	3.33	2.433	0.90

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
2580	2580	2580	2580	none

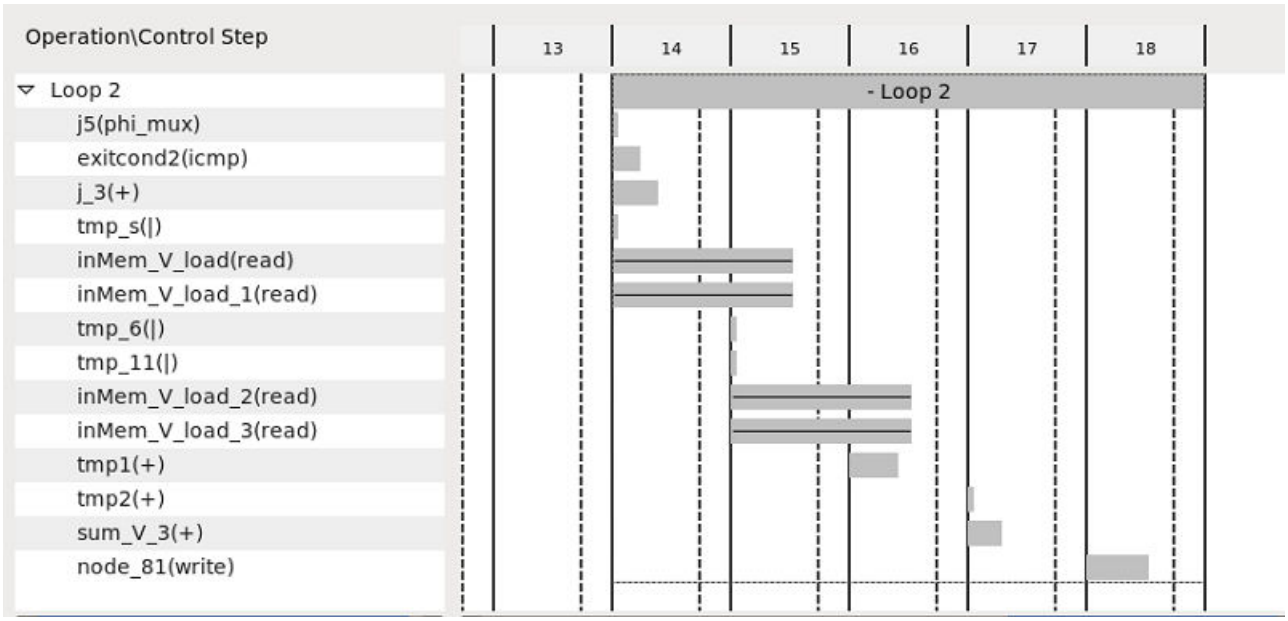
Detail

Instance

Loop

	Latency			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- Loop 1	1034	1034	12	1	1	1024	yes
- Loop 2	1280	1280	5	-	-	256	no
- Loop 3	257	257	3	1	1	256	yes

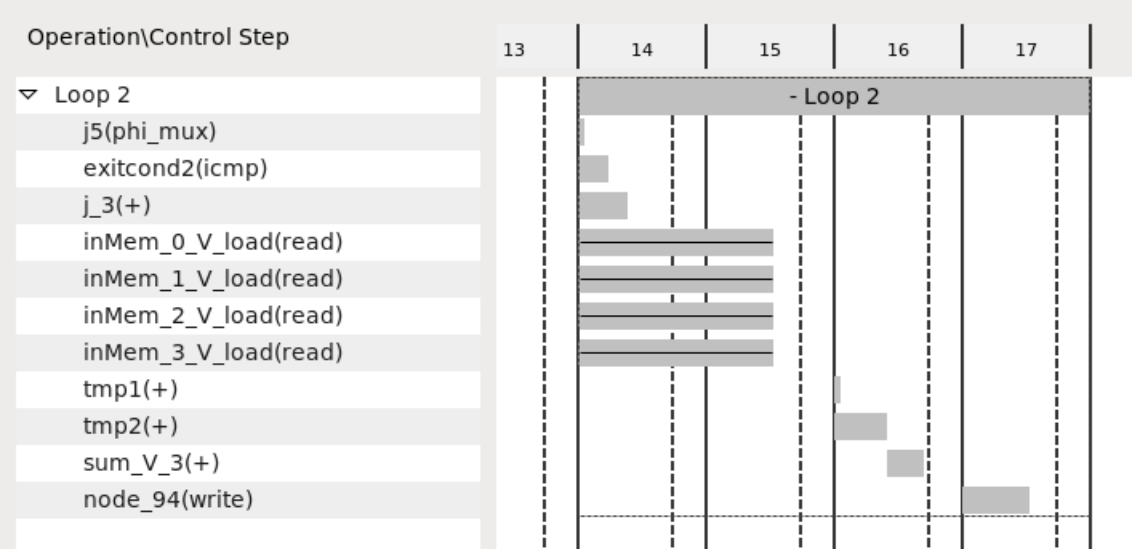
However, this improvement is largely due to the fact that this process uses both ports of a dual port memory. This can be seen from the Schedule Viewer in the HLS Project:



As you can see, two read operations are performed per cycle to access all the values from the memory to calculate the sum. This is often an undesired result as this completely blocks the access to the memory. To further improve the results, the memory can be split into four smaller memories along the second dimension:

```
#pragma HLS ARRAY_PARTITION variable=inMem complete dim=2
```

This results in four array reads, all executed on different memories using a single port:



Using a total of 256 * 4 cycles = 1024 cycles for loop 2.

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	3.33	2.433	0.90

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
2324	2324	2324	2324	none

Detail

Instance

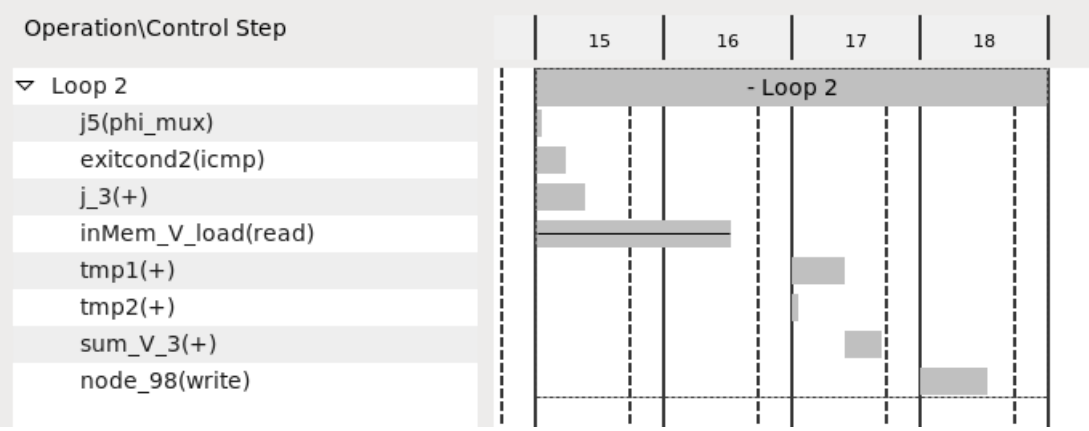
Loop

	Latency			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- Loop 1	1034	1034	12	1	1	1024	yes
- Loop 2	1024	1024	4	-	-	256	no
- Loop 3	257	257	3	1	1	256	yes

Alternatively, the memory can be reshaped into to a single memory with four words in parallel. This is performed through the pragma:

```
#pragma HLS array_reshape variable=inMem complete dim=2
```

This results in the same latency as when the array partitioning, but with a single memory using a single port:



Although, either solution creates comparable results with respect to overall latency and utilization, reshaping the array results in cleaner interfaces and less routing congestion making this the preferred solution. Note that this completes array optimization, in a real design the latency could further improved by exploiting loop parallelism (see the Loop Parallelism section).

```
void run (ap_uint<16> in[256][4],
      ap_uint<16> out[256]
      ) {
  ...

  ap_uint<16> inMem[256][4];
  ap_uint<16> outMem[256];
  #pragma HLS array_reshape variable=inMem complete dim=2

  ... Preprocess input to local memory

  for( int j=0; j<256; j++) {
    #pragma HLS PIPELINE OFF
    ap_uint<16> sum = 0;
    for( int i = 0; i<4; i++) {
      #pragma HLS UNROLL
      sum += inMem[j][i];
    }
    outMem[j] = sum;
  }

  ... Postprocess write local memory to output

}
```

Host Optimization

This section focuses on Host Code optimization. The host code uses the OpenCL™ API to schedule the individual compute unit executions and data transfers from and to the FPGA board. As a result, you need to be thinking about concurrent execution through the OpenCL queue(s). This section discusses in detail common pitfalls and how to recognize and address them.

Reducing Overhead of Kernel Enqueing

The OpenCL™ execution model supports data parallel and task parallel programming models. Kernels are usually enqueued by the OpenCL Runtime multiple times and then scheduled to be executed on the device. You must send the command to start the kernel in one of two ways:

- Implicitly, using clEnqueueNDRange API for the data parallel case
- Explicitly, using clEnqueueTask for the task parallel case

The dispatching process is executed on the host processor and the actual commands and kernel arguments need to be sent to the FPGA via PCIe® link. In the current OpenCL Runtime Library of SDAccel™ Environment, the overhead of dispatching the command and arguments to the FPGA is between 30us and 60us, depending the number of arguments on the kernel. You can reduce the impact of this overhead by minimizing the number of times the kernel needs to be executed.

For the data parallel case, Xilinx recommends that you carefully choose the global and local work sizes for your host code and kernel so that the global work size is a small multiple of the local work size. Ideally, the global work size is the same as the local work size as shown in the code snippet below:

```
size_t global = 1;
size_t local = 1;
clEnqueueNDRangeKernel(world.command_queue, kernel, 1, nullptr,
                       &global, &local, 2, write_events.data(),
                       &kernel_events[0]));
```

For the task parallel case, Xilinx recommends that you minimize the calls to clEnqueueTask. Ideally, you should finish all the work load in a single call to clEnqueueTask.

Overlapping Data Transfers with Kernel Computation

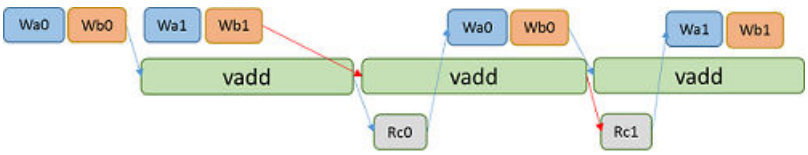
Applications like database analytics have much larger data set than the available memory on the acceleration device. They require the complete data to be transferred and processed in blocks. Techniques that overlap the data transfers with the computation are critical to achieve high performance for these applications.

Below is the vector add kernel from the OpenCL™ Overlap Data Transfers with Kernel Computation Example in the host category from Xilinx On-boarding Example GitHub.

```
kernel __attribute__((reqd_work_group_size(1, 1, 1)))
void vadd(global int* c,
          global const int* a,
          global const int* b,
          const int offset,
          const int elements)
{
    int end = offset + elements;
    vadd_loop: for (int x=offset; x<end; ++x) {
        c[x] = a[x] + b[x];
    }
}
```

There are four tasks to perform in the host application for this example: write buffer a (Wa), write buffer b (Wb), execute vadd kernel, and read buffer c (Rc). The asynchronous nature of OpenCL data transfer and kernel execution APIs allows overlap of data transfers and kernel execution as illustrated in the figure below. In this example, double buffering is used for all buffers so that the compute unit can process one set of buffers while the host can operate on the other set of buffers. The OpenCL event object provides an easy way to set up complex operation dependencies and synchronize host threads and device operations. The arrows in the figure below show how event triggering can be set up to achieve optimal performance.

Figure 13: **Event Triggering Set Up**



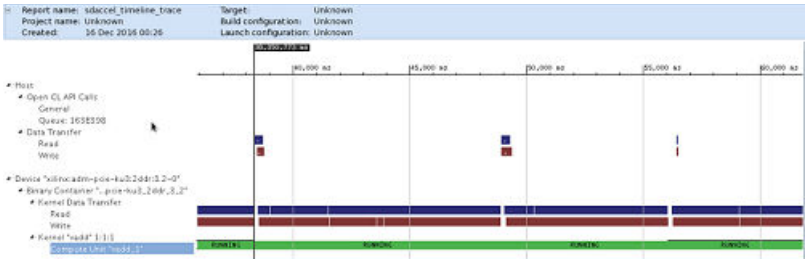
The host code snippet below enqueues the four tasks in a loop. It also sets up event synchronization between different tasks to ensure that data dependencies are met for each task. The double buffering is set up by passing different memory objects values to `clEnqueueMigrateMemObjects` API. The event synchronization is achieved by having each API to wait for other event as well as trigger its own event when the API completes.

```
  for (size_t iteration_idx = 0; iteration_idx < num_iterations;
iteration_idx++) {
    int flag = iteration_idx % 2;

    if (iteration_idx >= 2) {
        clWaitForEvents(1, &map_events[flag]);
        OCL_CHECK(clReleaseMemObject(buffer_a[flag]));
        OCL_CHECK(clReleaseMemObject(buffer_b[flag]));
        OCL_CHECK(clReleaseMemObject(buffer_c[flag]));
        OCL_CHECK(clReleaseEvent(read_events[flag]));
        OCL_CHECK(clReleaseEvent(kernel_events[flag]));
    }

    buffer_a[flag] = clCreateBuffer(world.context,
            CL_MEM_READ_ONLY | CL_MEM_USE_HOST_PTR,
           bytes_per_iteration, &A[iteration_idx *
elements_per_iteration], NULL);
    buffer_b[flag] = clCreateBuffer(world.context,
            CL_MEM_READ_ONLY | CL_MEM_USE_HOST_PTR,
           bytes_per_iteration, &B[iteration_idx *
elements_per_iteration], NULL);
    buffer_c[flag] = clCreateBuffer(world.context,
            CL_MEM_WRITE_ONLY | CL_MEM_USE_HOST_PTR,
           bytes_per_iteration, &device_result[iteration_idx *
elements_per_iteration], NULL);
    array<cl_event, 2> write_events;
    printf("Enqueueing Migrate Mem Object (Host to Device) calls\n");
    // These calls are asynchronous with respect to the main thread
because we
    // are passing the CL_FALSE as the third parameter. Because we are
passing
    // the events from the previous kernel call into the wait list, it
will wait
    // for the previous operations to complete before continuing
    OCL_CHECK(clEnqueueMigrateMemObjects(
        world.command_queue, 1, &buffer_a[iteration_idx % 2],
        0 /* flags, 0 means from host */,
        0, NULL,
        &write_events[0]));
    set_callback(write_events[0], "ooo_queue");

    OCL_CHECK(clEnqueueMigrateMemObjects(
        world.command_queue, 1, &buffer_b[iteration_idx % 2],
        0 /* flags, 0 means from host */,
        0, NULL,
        &write_events[1]));
    set_callback(write_events[1], "ooo_queue");

    xcl_set_kernel_arg(kernel, 0, sizeof(cl_mem), &buffer_c[iteration_idx
% 2]);
    xcl_set_kernel_arg(kernel, 1, sizeof(cl_mem), &buffer_a[iteration_idx
% 2]);
    xcl_set_kernel_arg(kernel, 2, sizeof(cl_mem), &buffer_b[iteration_idx
% 2]);
    xcl_set_kernel_arg(kernel, 3, sizeof(int), &elements_per_iteration);
```

```
    printf("Enqueueing NDRange kernel.\n");
    // This event needs to wait for the write buffer operations to complete
    // before executing. We are sending the write_events into its wait
list to
    // ensure that the order of operations is correct.
    OCL_CHECK(clEnqueueNDRangeKernel(world.command_queue, kernel, 1,
nullptr,
                                     &global, &local, 2 ,
write_events.data(),
                                     &kernel_events[flag]));
    set_callback(kernel_events[flag], "ooo_queue");

    printf("Enqueueing Migrate Mem Object (Device to Host) calls\n");
    // This operation only needs to wait for the kernel call. This call
will
    // potentially overlap the next kernel call as well as the next read
    // operations
    OCL_CHECK( clEnqueueMigrateMemObjects(world.command_queue, 1,
&buffer_c[iteration_idx % 2],
                CL_MIGRATE_MEM_OBJECT_HOST, 1, &kernel_events[flag],
&read_events[flag]));

    set_callback(read_events[flag], "ooo_queue");
    clEnqueueMapBuffer(world.command_queue, buffer_c[flag], CL_FALSE,
CL_MAP_READ, 0,
            bytes_per_iteration, 1, &read_events[flag], &map_events[flag],
0);
    set_callback(map_events[flag], "ooo_queue");

    OCL_CHECK(clReleaseEvent(write_events[0]));
    OCL_CHECK(clReleaseEvent(write_events[1]));
  }
```

The Application Timeline view below clearly shows that the data transfer time is completely hidden, while the compute unit vadd_1 is running constantly.

Figure 14: **Data Transfer Time Hidden in Application Timeline View**



Using Multiple Compute Units

Depending on available resources on the target device, multiple compute units of the same kernel or different kernels can be created and run in parallel to improve the system processing time and throughput.

An application can use multiple compute units in the target device by creating multiple in-order command queues or a single out-of-order command queue.

Multiple In-Order Command Queues

The following figure shows an example with two in-order command queues, CQ0 and CQ1. The scheduler dispatches commands from each queue in order, but commands from CQ0 and CQ1 can be pulled out by the scheduler in any order. You must manage synchronization between CQ0 and CQ1 if required.

Figure 15: **Example with Two In-Order Command Queues**

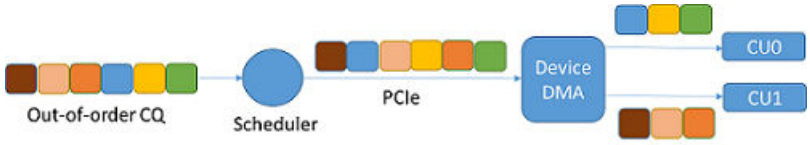
Below is the code snippet from the *Concurrent Kernel Execution Example* in host category from Xilinx On-boarding Example GitHub that sets up multiple in-order command queues and enqueues commands into each queue:

```
cl_command_queue ordered_queue1 = clCreateCommandQueue(
      world.context, world.device_id, CL_QUEUE_PROFILING_ENABLE, &err)

cl_command_queue ordered_queue2 = clCreateCommandQueue(
world.context, world.device_id, CL_QUEUE_PROFILING_ENABLE, &err);

clEnqueueNDRangeKernel(ordered_queue1, kernel_mscale, 1, offset,
                       global, local, 0, nullptr,
                       &kernel_events[0]));

clEnqueueNDRangeKernel(ordered_queue1, kernel_madd, 1, offset,
                       global, local, 0, nullptr,
                       &kernel_events[1]);

clEnqueueNDRangeKernel(ordered_queue2, kernel_mmult, 1, offset,
                       global, local, 0, nullptr,
                       &kernel_events[2]);
```

Single Out-of-Order Command Queue

The figure below shows an example with a single out-of-order command queue CQ. The scheduler can dispatch commands from CQ in any order. You must set up event dependencies and synchronizations explicitly if required.

Figure 16: **Example with Single Out-of-Order Command Queue**



Below is the code snippet from the *Concurrent Kernel Execution Example* from Xilinx On-boarding Example GitHub that sets up a single out-of-order command queue and enqueues commands:

```
cl_command_queue ooo_queue = clCreateCommandQueue(
      world.context, world.device_id,
      CL_QUEUE_PROFILING_ENABLE | CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE,
&err);

clEnqueueNDRangeKernel(ooo_queue, kernel_mscale, 1, offset, global,
                       local, 0, nullptr, &ooo_events[0]);

clEnqueueNDRangeKernel(ooo_queue, kernel_madd, 1, offset, global,
                      local, 1,
                      &ooo_events[0], // Event from previous call
                      &ooo_events[1]);

clEnqueueNDRangeKernel(ooo_queue, kernel_mmult, 1, offset, global,
                       local, 0,
                       nullptr, // Does not depend on previous call
                       &ooo_events[2])
```

The Application Timeline view below 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 17: **Application Timeline View Showing mult_1 Running with mscale_1 and madd_1**

Using clEnqueueMigrateMemObjects to Transfer Data

OpenCL™ 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. This makes it difficult for the host application to overlap the placements of the memory objects onto the device with the computation carried out by kernels.

OpenCL 1.2 introduced a new API, `clEnqueueMigrateMemObjects`, with which memory migration can be explicitly performed ahead of the dependent commands. This allows the application to preemptively change the association of a memory object, through regular command queue scheduling, in order 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 transfer latencies. Once the event associated by `clEnqueueMigrateMemObjects` has been marked CL_COMPLETE, the memory objects specified in mem_objects have been successfully migrated to the device associated with command_queue.

The `clEnqueueMigrateMemObjects` API can also be used to direct the initial placement of a memory object after creation, possibly avoiding the initial overhead of instantiating the object on the first enqueued command to use it.

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 for transferring data for more than one memory object.

Below is the code snippet showing the usage of `clEnqueueMigrateMemObjects` from *Vector Multiplication for XPR Device* example in the host category from Xilinx On-boarding Example GitHub.

```
int err = clEnqueueMigrateMemObjects(
                        world.command_queue,
                        1,
                        &d_mul_c,
                        CL_MIGRATE_MEM_OBJECT_HOST,
                        0,
                        NULL,
                        NULL);
```

On-Boarding Examples

To help users quickly get started with the SDAccel™ Environment, Xilinx On-boarding Example GitHub 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 that are illustrated by individual examples in both OpenCL™ C and C/C++ when applicable. All examples include Makefile for running software emulation, hardware emulation, and running on hardware and a `README.md` file that explains the example in details.

Appendix B

Additional Resources and Legal Notices

Xilinx Resources

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

Solution Centers

See the Xilinx Solution Centers for support on devices, software tools, and intellectual property at all stages of the design cycle. Topics include design assistance, advisories, and troubleshooting tips

Xilinx Resources

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

Documentation Navigator and Design Hubs

Xilinx® Documentation Navigator provides access to Xilinx documents, videos, and support resources, which you can filter and search to find information. To open the Xilinx Documentation Navigator (DocNav):

- From the SDAccel™ or SDSoC™ 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 the Xilinx Documentation Navigator, click the **Design Hubs View** tab.
- On the Xilinx website, see the Design Hubs page.

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

References

1. *SDx Environments Release Notes, Installation, and Licensing Guide* (UG1238)
2. *SDAccel Environment User Guide* (UG1023)
3. *SDAccel Environment Profiling and Optimization Guide* (UG1207)
4. *SDAccel Environment Getting Started Tutorial* (UG1021)
5. SDAccel Development Environment web page
6. Vivado® Design Suite Documentation
7. *Vivado Design Suite User Guide: Designing IP Subsystems using IP Integrator* (UG994)
8. *Vivado Design Suite: Creating and Packaging Custom IP* (UG1118)
9. *Vivado Design Suite User Guide: Partial Reconfiguration* (UG909)
10. *Vivado Design Suite User Guide: High-Level Synthesis* (UG902)
11. *UltraFast Design Methodology Guide for the Vivado Design Suite* (UG949)
12. *Vivado Design Suite Properties Reference Guide* (UG912)
13. Khronos Group web page: Documentation for the OpenCL standard
14. Xilinx Virtex UltraScale+ FPGA VCU1525 Acceleration Development Kit
15. Xilinx Kintex UltraScale FPGA KCU1500 Acceleration Development Kit

Please Read: Important Legal Notices

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