Optimization Lab

Introduction

In this lab you will create a Vitis project and analyze the design to optimize the host code and kernel code to improve the performance of the design.

Objectives

After completing this lab, you will be able to:

- Analyze the design and read project reports
- Optimize the kernel code to improve throughput
- Optimize the host code to improve the data transfer rate
- Verify the functionality of the design in hardware

Create a Vitis Project

1. Start Vitis and select the default workspace (or continue with the workspace from the previous lab)

2. Create a new application project

   Use `Create Application Project` from Welcome page, or use `File > New > Application Project` to create a new application

3. In the *New Application Project's* page enter **optimization_lab** in the *Project name:* field and click **Next>**

4. Select your target platform and click **Next>**

   You should see `xilinx_aws-vu9p-f1_shell-v04261818_201920_1` as one of the platforms if you are continung with previous lab, otherwise add it from `~/aws-fpga/Vitis/aws_platform`

5. Select **Empty Application** template and click **Finish**

Import the source files into the project

1. In the *Explorer* view expand the *optimization_lab* folder if necessary, and right-click on the **src** folder and select **Import Sources…**

2. Browse to the source directory at **~/compute_acceleration/sources/optimization_lab** and click **OK**

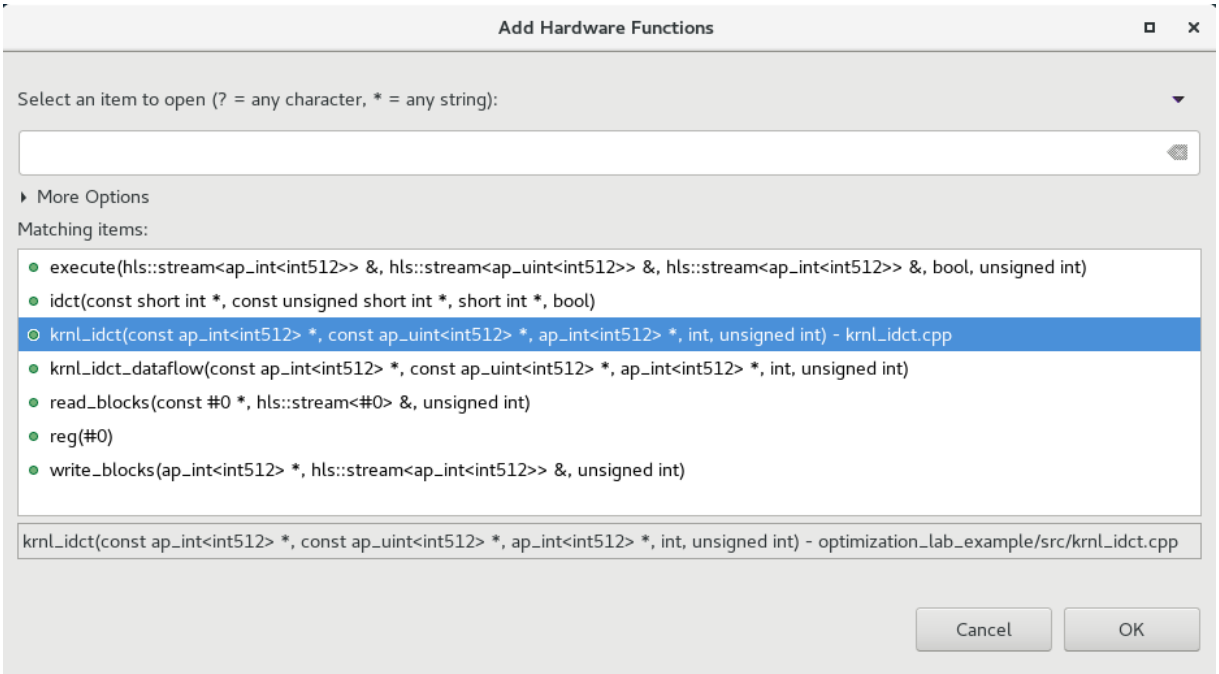
3. Select **idct.cpp** and **krnl_idct.cpp** files and click **Finish**

4. Expand the **src** folder in the *Explorer* view and note the two added files

Add a function as a hardware kernel

1. Click on the *Add Hardware Function* button icon ( ) in the **Hardware Functions** window to see functions available for implementation in hardware

2. Select *krnl_idct* function and click **OK**



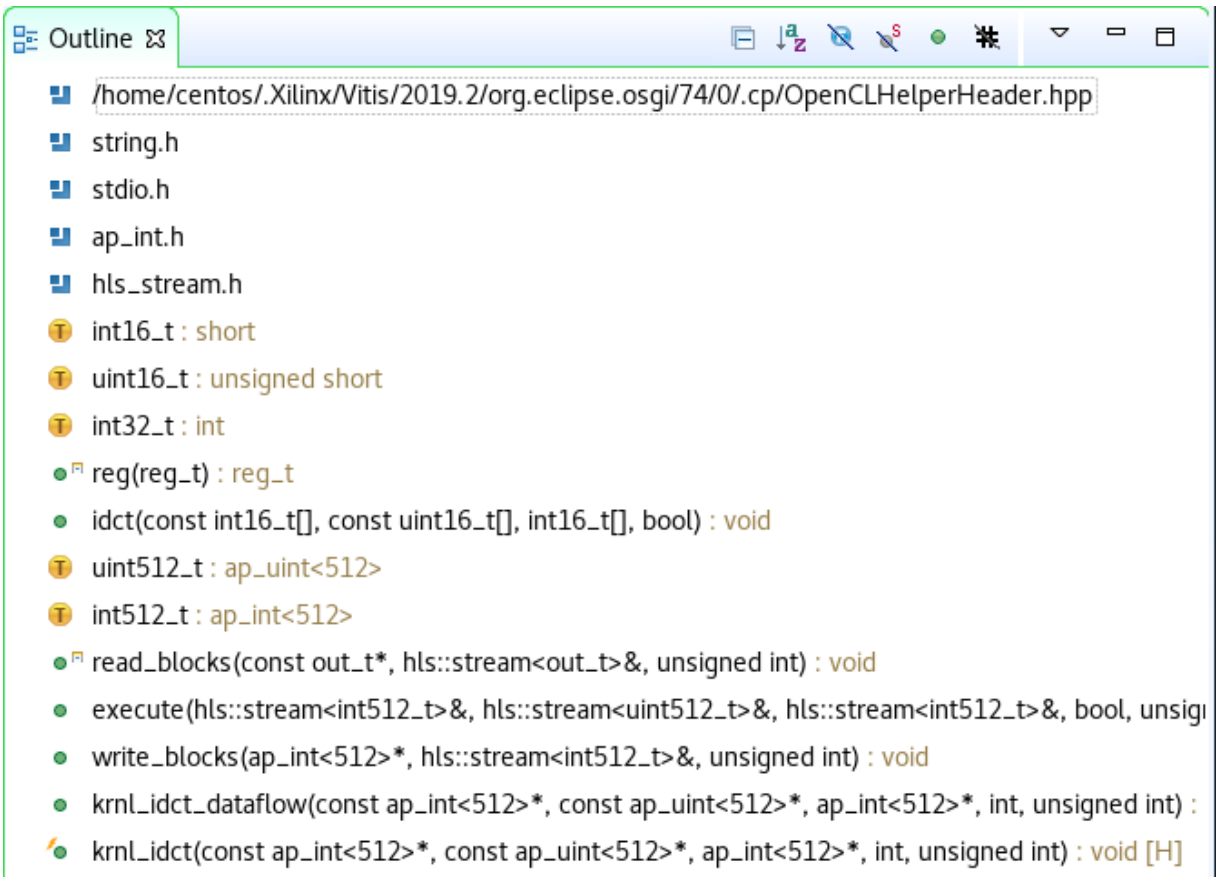
3. Notice a **binary_container_1** folder is created automatically under which the *krnl_idct* function is added

Analyze the source files

1. From the *Explorer* view open the **src > krnl_idct.cpp** file

2. The **Outline** panel should be visible. It displays an outline of the code of the source file that is currently in scope. If you can't see it, go to **Window > Show View** then select **General > Outline**

   The outline view can be used to navigate the source file. For example, function names are displayed in the outline view, and clicking on a function will jump to the line of code where the function is defined



3. In the *Outline* viewer, click **idct** to look up the function

The `idct()` function is the core algorithm in the kernel. It is a computationally intensive function that can be highly parallelized on the FPGA, providing significant acceleration over a CPU-based implementation

1. Review the code

   - **krnl_idct** : Top-level function for the hardware kernel. Interface properties for the kernel are specified in this function
   - **krnl_idct_dataflow** : Called by the **krnl_idct** function and encapsulates the main functions of the kernel
   - **read_blocks** : Reads data from global memory data sent by the host application and streams to the *execute* function
   - **execute** : For each 8x8 block received, calls the **idct** function to perform the actual IDCT computation
   - **write_blocks** : Receives results from the **execute** function and writes them back to global memory for the host application

2. Open the **idct.cpp** file. Again, use the *Outline* viewer to quickly look up and inspect the important functions of the host application:

   - **main** : Initializes the test vectors, sets-up OpenCL resources, runs the reference model, runs the hardware kernel, releases the OpenCL resources, and compares the results of the reference IDCT model with the hardware implementation
   - **runFPGA** : Takes in a vector of inputs and for each 8x8 block calls the hardware accelerated IDCT using the **write** , **run** , **read** , and **finish** helper functions. These function use OpenCL API calls to communicate with the FPGA
   - **runCPU** : Takes in a vector of inputs and, for each 8x8 block, calls **idctSoft** , a reference implementation of the IDCT
   - **idctSoft** : Software implementation of the IDCT algorithm, used to check the results from the FPGA
   - **oclDct** : This class is used to encapsulate the OpenCL runtime calls to interact with the kernel in the FPGA
   - **aligned_allocator** , **smalloc** , **load_file_to_memory** : These are small helper functions used during test vector generation and OpenCL setup

3. Look at the code around line number 500 of the **idct.cpp** file by pressing Ctrl+l (small L) and entering 496.
   This section of code is where the OpenCL environment is setup in the host application. It is typical of most Vitis application

and will look very familiar to developers with prior OpenCL experience. This body of code can often be reused as-is from project to project
To setup the OpenCL environment, the following API calls are made:

- **clGetPlatformIDs** : This function queries the system to identify any available OpenCL platforms. It is called twice as it first extracts the number of platforms before extracting the actual supported platforms
- **clGetPlatformInfo** : Gets specific information about the OpenCL platform, such as vendor name and platform name
- **clGetDeviceIDs** : Obtains list of devices available on a platform
- **clCreateContext** : Creates an OpenCL context, which manages the runtime objects
- **clGetDeviceInfo** : Gets information about an OpenCL device like the device name
- **clCreateProgramWithBinary** : Creates a program object for a context, and loads specified binary data into the program object. The actual program is obtained before this call through `load_file_to_memory()` function
- **clCreateKernel** : Creates a kernel object
- **clCreateCommandQueue** : Creates a command-queue on a specific device

Note: all objects accessed through a **clCreate…** function call should be released before terminating the program by calling a corresponding **clRelease…** This avoids memory leakage and clears the locks on the device

Configure the System Port options

Configure the System Port in the Vitis GUI

In the *idct.cpp* file, locate lines 286-297. Note that two memory buffers, *mInBuffer* and *mOutBuffer* are being used. The memory buffers will be located in external DRAM. The kernel will have one or more ports connected to the memory bank(s). By default, the compiler will connect all ports to BANK0 or DDR[0]. Memory interfaces can be configured from the Vitis GUI, or via a "System Port" switch (--sp) that is passed to the XOCC Kernel Linker.

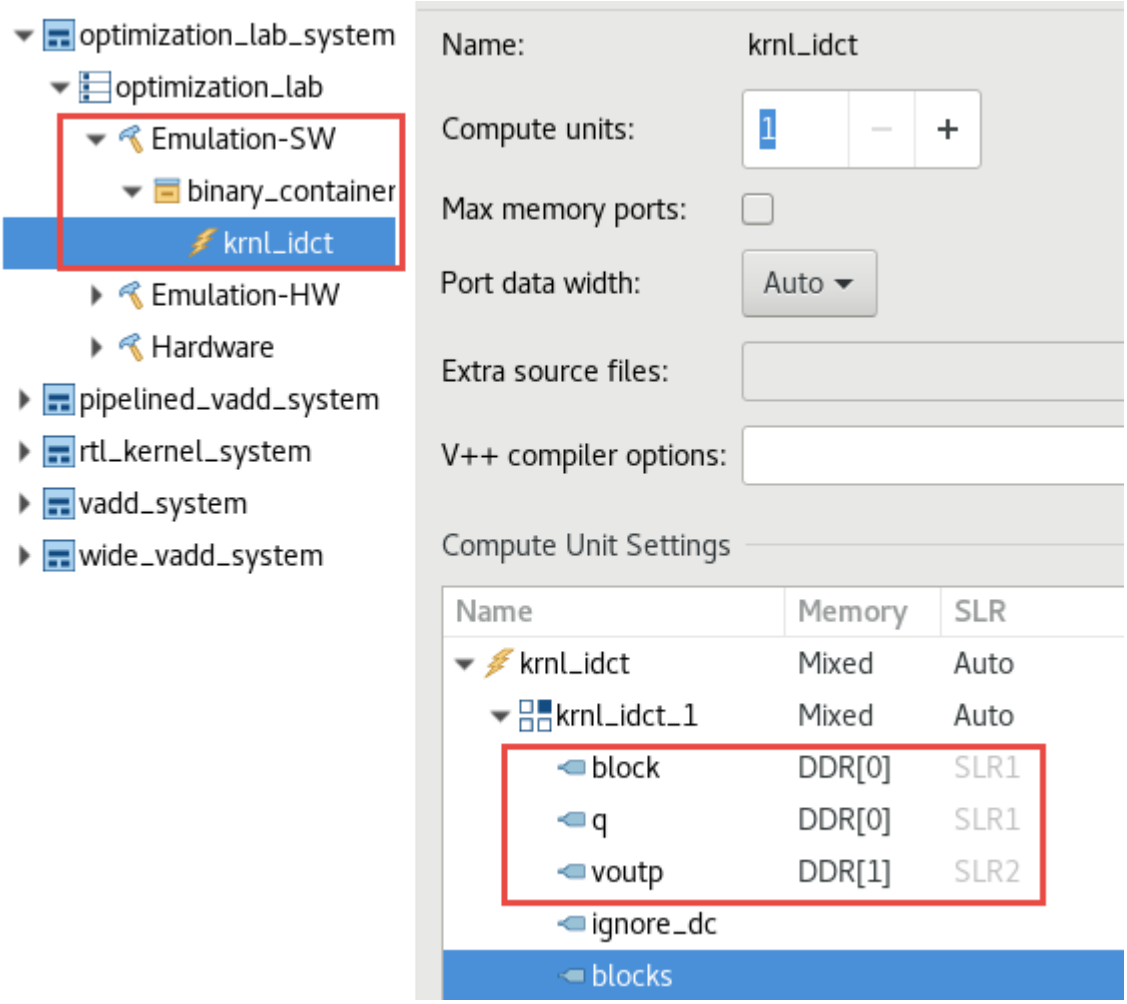
1. In the *Assistant* view, right click on Emulation-SW and click **Settings**

2. In the *Hardware Function Settings* expand *optimization_lab* > *Emulation-SW* > *binary_container_1* and select **krnl_idct**

3. Under *Compute Unit Settings* expand *krnl_idct* and *krnl_idct_1*

4. From the dropdown block under *Memory* select the following:

    - **block**: DDR[0]
    - **q**: DDR[0]
    - **voutp**: DDR[1]



5. Click **Apply and Close**

Configure the System Port command line switch

In Vitis, you can set memory interfaces by passing a config file with the `--config` compiler switch to the Vitis kernel linker. To this end, the file must contain the `[connectivity]` indicator

```
[connectivity]
sp=<kernel_instance_name>.<interface_name>:<bank_name>
```

- **kernel_instance_name** is the instance name of the kernel
- **interface_name** is the name of the memory interface
- **bank_name** is the name of the bank where the memory is going to be mapped

The interface names can be found in the *Explorer* view **Emulation-SW (or Emulation-HW) > binary_container_1.xclbin.info** log file.

1. In this case the configuration file would be

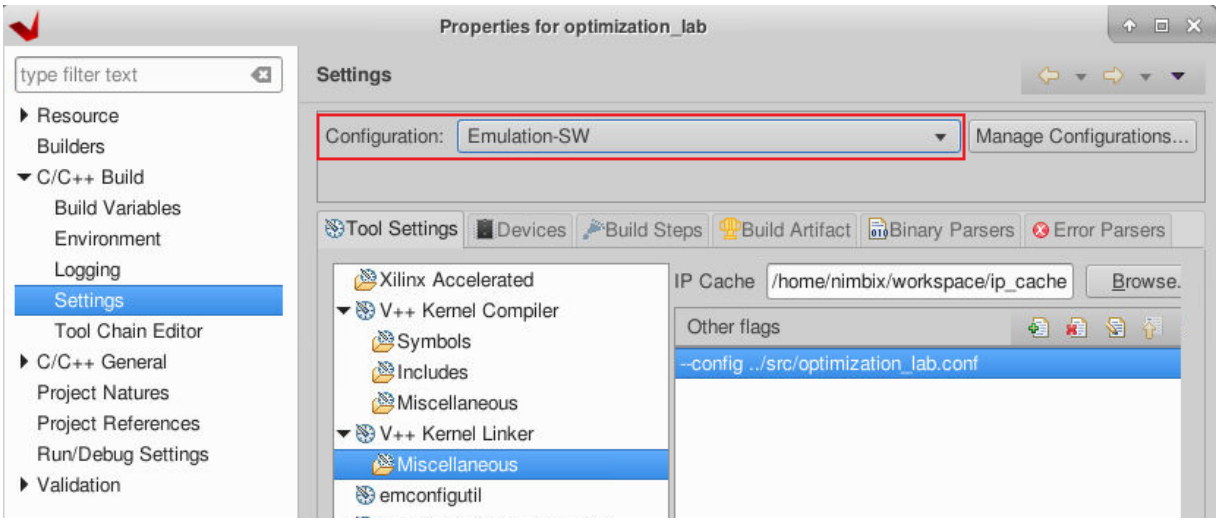
```
[connectivity]
sp=krnl_idct_1.block:DDR[0]
sp=krnl_idct_1.q:DDR[0]
sp=krnl_idct_1.voutp:DDR[1]
```

2. In the *Explore* view, right-click the project **optimization_lab** and select the **C/C++ Settings**

3. Select **C/C++ Build** > **Settings** in the left pane

4. Select the **Miscellaneous** under **Vitis Kernel Linker**. Make sure you add this switch for every solution.

5. You can also pass the configuration file using the **V++ compiler options:** of the Hardware function settings windows, but again this must be done for every solution



Build and run software emulation (Emulation-SW)

1. Make sure the **optimization_lab.prj** under *optimization_lab* in the *Explorer* view is selected

2. Select **Emulation-SW** as the *Active Build Configuration*

3. Build the project by clicking () button

4. In the Explorer pane, right-click the project **optimization_lab** and select **Run As > Run Configurations…**

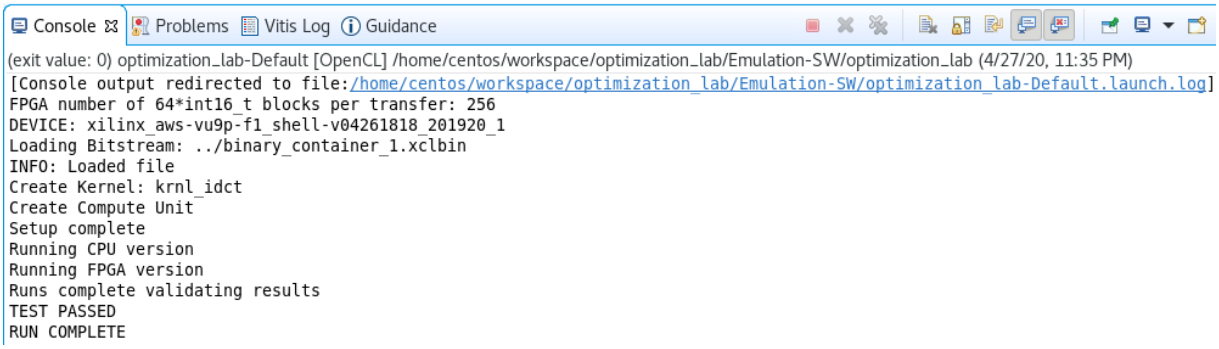
5. Select the **Arguments** tab

6. Click on the **Automatically add binary container(s) to arguments** check box.

   This will add **../binary_container_1.xclbin** automatically

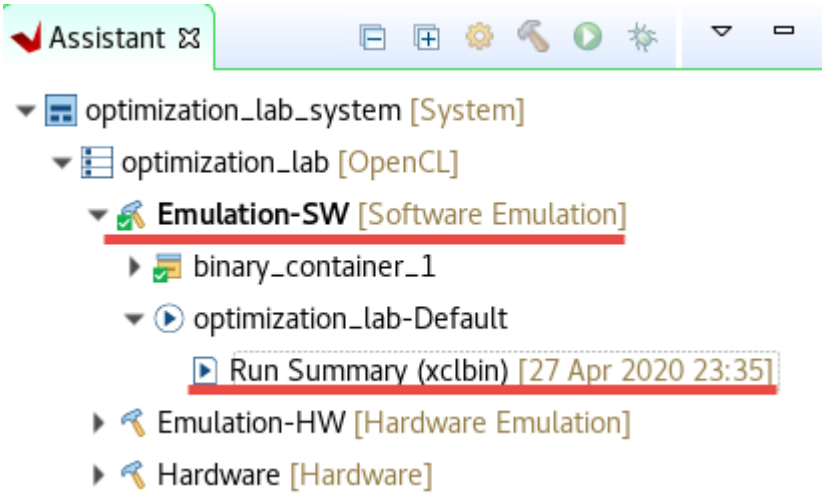
7. Click **Apply** and then click **Run**

   The application will be run and the output will be displayed in the Console tab



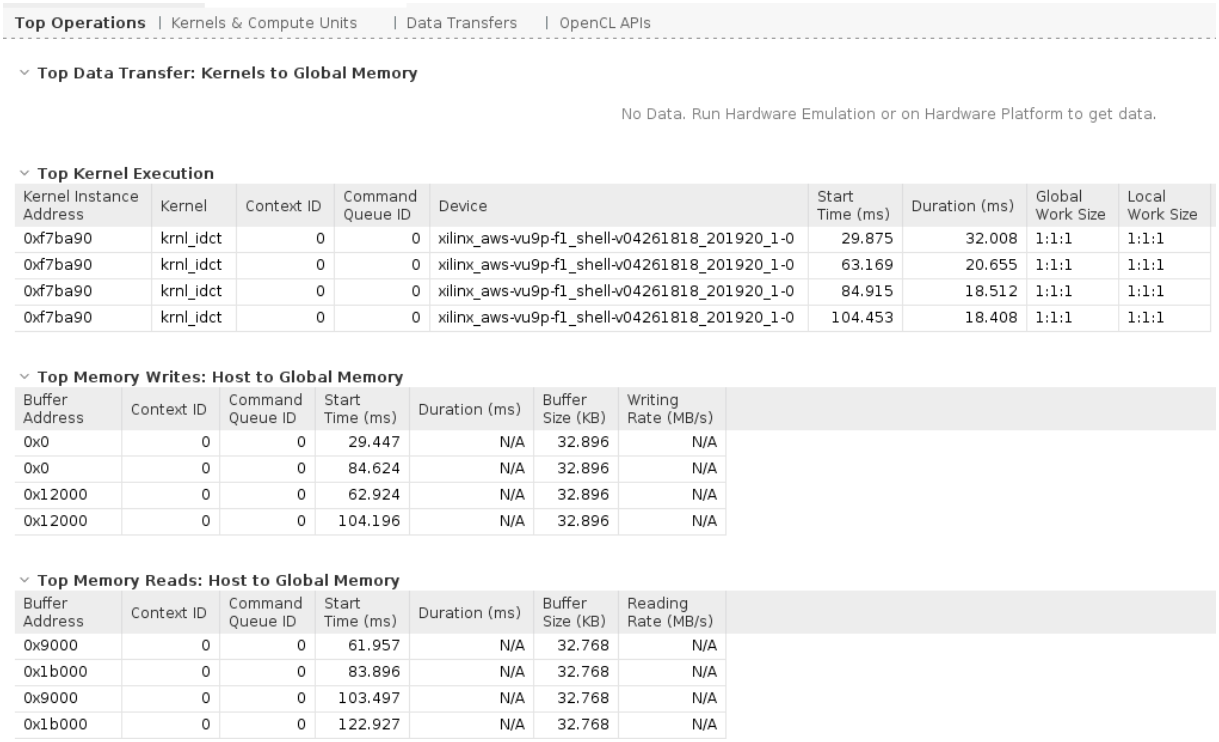
Review the software emulation reports

1. In the *Assistant* view, expand **optimization_lab** > **Emulation-SW** > **optimization_lab-Default** and double-click on **Run Summary (xclbin)**



The *Vitis Analyzer* application will open, showing two panels on left: Profile Summary and Application Timeline

2. Click on the **Profile Summary** and review it



Top Operations | Kernels & Compute Units | Data Transfers | OpenCL APIs

Top Data Transfer: Kernels to Global Memory

No Data. Run Hardware Emulation or on Hardware Platform to get data.

Top Kernel Execution

Kernel Instance Address	Kernel	Context ID	Command Queue ID	Device	Start Time (ms)	Duration (ms)	Global Work Size	Local Work Size
0xf7ba90	krnl_idct	0	0	xilinx_aws-vu9p-f1_shell-v04261818_201920_1-0	29.875	32.008	1:1:1	1:1:1
0xf7ba90	krnl_idct	0	0	xilinx_aws-vu9p-f1_shell-v04261818_201920_1-0	63.169	20.655	1:1:1	1:1:1
0xf7ba90	krnl_idct	0	0	xilinx_aws-vu9p-f1_shell-v04261818_201920_1-0	84.915	18.512	1:1:1	1:1:1
0xf7ba90	krnl_idct	0	0	xilinx_aws-vu9p-f1_shell-v04261818_201920_1-0	104.453	18.408	1:1:1	1:1:1

Top Memory Writes: Host to Global Memory

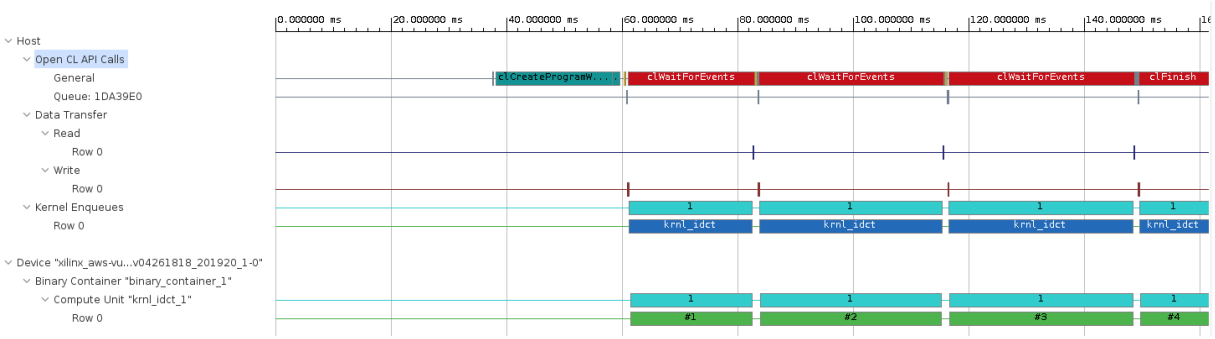
Buffer Address	Context ID	Command Queue ID	Start Time (ms)	Duration (ms)	Buffer Size (KB)	Writing Rate (MB/s)
0x0	0	0	29.447	N/A	32.896	N/A
0x0	0	0	84.624	N/A	32.896	N/A
0x12000	0	0	62.924	N/A	32.896	N/A
0x12000	0	0	104.196	N/A	32.896	N/A

Top Memory Reads: Host to Global Memory

Buffer Address	Context ID	Command Queue ID	Start Time (ms)	Duration (ms)	Buffer Size (KB)	Reading Rate (MB/s)
0x9000	0	0	61.957	N/A	32.768	N/A
0x1b000	0	0	83.896	N/A	32.768	N/A
0x9000	0	0	103.497	N/A	32.768	N/A
0x1b000	0	0	122.927	N/A	32.768	N/A

3. Click on each of tabs and review them

4. Click the **Application Timeline** report and review it



The **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: when they happen and how long each of them takes.

5. Close the Vitis Analyzer by clicking **File > Exit** and then clicking **OK**

Perform HW Emulation

1. Click on the drop-down button of *Active build configuration* and select **Emulation-HW**

2. Assign the System Ports as you did in the Emulation-SW mode

   - Right-click on *Assistant view* > *Emulation-HW* and click **Settings**

   - Expand *optimization_lab* > *Emulation-HW* > *binary_container_1* and select **krnl_idct**

   - Under *Compute Unit Settings* expand *krnl_idct* and krnl_idct_1

   - Select the following:

     - **block**: DDR[0]
     - **q**: DDR[0]
     - **voutp**: DDR[1]

3. Click **Apply and Close**

4. Build the project ()

   Wait for the build to complete which will generate binary_container_1.xclbin file in the Emulation-HW directory

   > There is a bug in waveform viewer not showing in Vivado simulation. To workaround that a tcl script is provided in ~/compute_acceleration/sources/optimization_lab folder

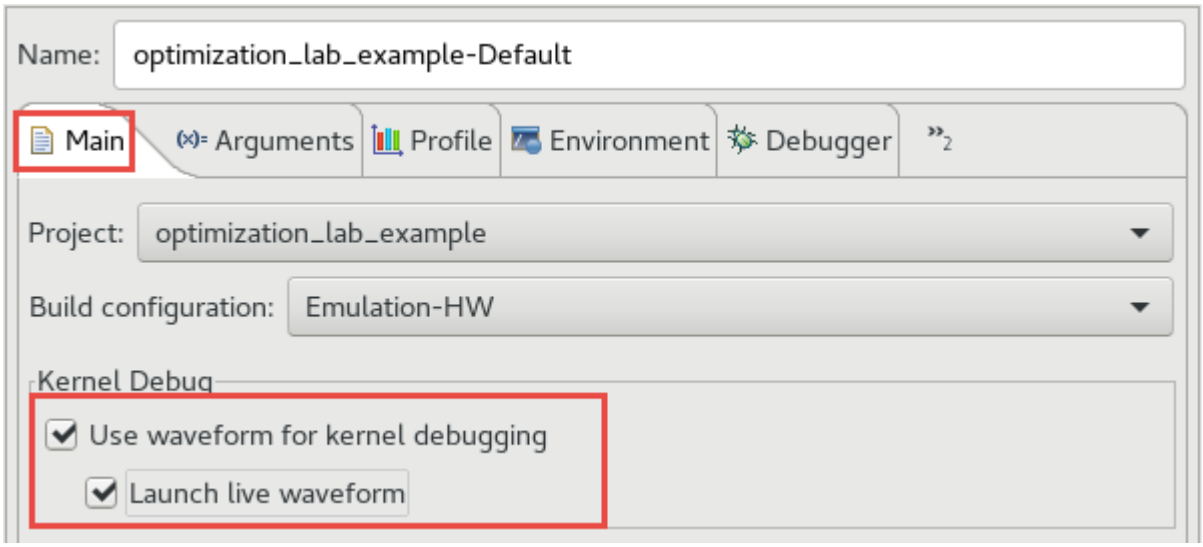
5. In a terminal window, execute the following commands to create another xclbin file which will be used to view the waveform

```
cd ~/workspace/optimization_lab/Emulation-HW
~/compute_acceleration/sources/optimization_lab/waveform_patch.tcl binary_container_1.xclbin binary_container_2.xcl
```

   This will create binary_container_2.xclbin which will be used in run_configuration settings

6. Select **Run > Run Configurations…** to open the configurations window

7. In the *Main* tab, click to select **Use waveform for kernel debugging** and **Launch live waveform**



8. Click on the **Arguments** tab, uncheck **Automatically add binary container(s) to arguments**, and enter *../binary_container_2.xclbin* in the box as a file to be used

9. Click **Apply** and then **Run** to run the application

   The Console tab shows that the test was completed successfully along with the data transfer rate

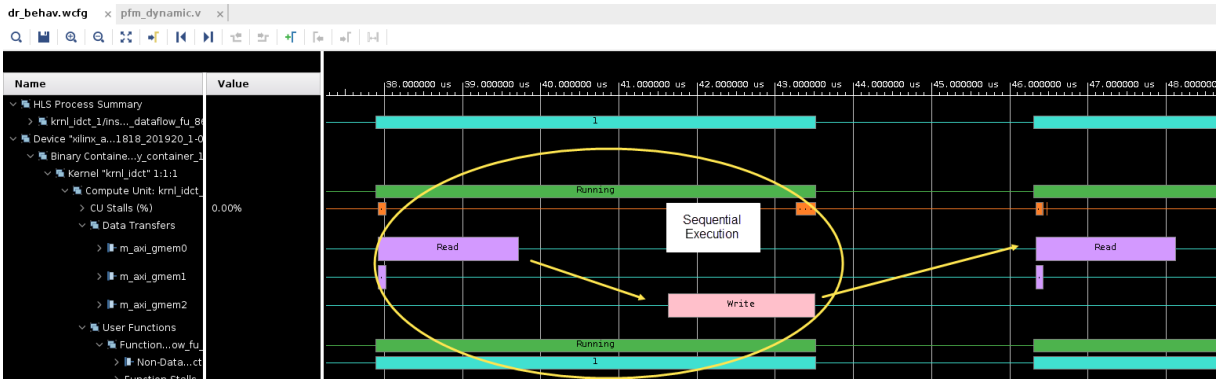
```
[Console output redirected to file:/home/centos/workspace/optimization_lab/Emulation-HW/optimization_lab-Default.launch.log]
FPGA number of 64*int16_t blocks per transfer: 256
DEVICE: xilinx_aws-vu9p-f1_shell-v04261818_201920_1
Loading Bitstream: ../binary_container_2.xclbin
INFO: Loaded file
INFO: [HW-EM 01] Hardware emulation runs simulation underneath. Using a large data set will result in long simulation times.
Create Kernel: krnl_idct
Create Compute Unit
Setup complete
Running CPU version
Running FPGA version
Runs complete validating results
TEST PASSED
RUN COMPLETE
INFO::[ Vitis-EM 22 ] [Time elapsed: 0 minute(s) 49 seconds, Emulation time: 0.0771904 ms]
Data transfer between kernel(s) and global memory(s)
krnl_idct_1:m_axi_gmem0-DDR[0]          RD = 128.000 KB             WR = 0.000 KB
krnl_idct_1:m_axi_gmem1-DDR[0]          RD = 0.500 KB               WR = 0.000 KB
krnl_idct_1:m_axi_gmem2-DDR[1]          RD = 0.000 KB               WR = 128.000 KB

INFO: [HW-EM 06-0] Waiting for the simulator process to exit
INFO: [HW-EM 06-1] All the simulator processes exited successfully
```

   Notice that Vivado was started and the simulation waveform window is updated.

10. Click on the Zoom Fit () button and scroll down the waveform window to see activities taking place in the kernel

    Notice that the execution is sequential



Close Vivado when you are ready by selecting **File > Exit** and clicking **OK**. We will not examine the transactions in detail.

Understand the HLS Report, profile summary, and Application Timeline

1. In the *Assistant* view, expand *optimization_lab* > *Emulation-HW* > *optimization_lab-Default* and double-click on **Run Summary (xclbin)**

2. The *Vitis Analyzer* window will update and now it includes *xclbin (Hardware Emulation)*

3. Click the *xclbin (Hardware Emulation)* > *Profile Summary* report and review it

Top Operations | Kernels & Compute Units | Data Transfers | OpenCL APIs | Kernel Internals

Top Data Transfer: Kernels to Global Memory

Device	Compute Unit	Number Of Transfers	Average Bytes per Transfer	Transfer Efficiency (%)	Total Data Transfer (MB)	Total Write (MB)	Total Read (MB)	Total Transfer Rate (MB/s)
xilinx_aws-vu9p-f1_shell-v04261818_201920_1-0	krnl_idct_1	260	1010.220	24.663	0.263	0.131	0.132	17506.500

Top Kernel Execution

Kernel Instance Address	Kernel	Context ID	Command Queue ID	Device	Start Time (ms)	Duration (ms)	Global Work Size	Local Work Size
0x7172f0	krnl_idct	0	0	xilinx_aws-vu9p-f1_shell-v04261818_201920_1-0	0.052	0.008	1:1:1	1:1:1
0x7172f0	krnl_idct	0	0	xilinx_aws-vu9p-f1_shell-v04261818_201920_1-0	0.036	0.008	1:1:1	1:1:1
0x7172f0	krnl_idct	0	0	xilinx_aws-vu9p-f1_shell-v04261818_201920_1-0	0.045	0.007	1:1:1	1:1:1
0x7172f0	krnl_idct	0	0	xilinx_aws-vu9p-f1_shell-v04261818_201920_1-0	0.061	0.007	1:1:1	1:1:1

Top Memory Writes: Host to Global Memory

Buffer Address	Context ID	Command Queue ID	Start Time (ms)	Duration (ms)	Buffer Size (KB)	Writing Rate (MB/s)
0x4000000000	0	0	59542.200	N/A	32.896	N/A
0x4000001000	0	0	72557.900	N/A	32.896	N/A
0x4000009000	0	0	63547.400	N/A	32.896	N/A
0x4000012000	0	0	68553.000	N/A	32.896	N/A

Top Memory Reads: Host to Global Memory

Buffer Address	Context ID	Command Queue ID	Start Time (ms)	Duration (ms)	Buffer Size (KB)	Reading Rate (MB/s)
0x400018000	0	0	76561.600	N/A	32.768	N/A
0x400000000	0	0	63545.000	N/A	32.768	N/A
0x400010000	0	0	72555.800	N/A	32.768	N/A
0x400008000	0	0	68550.500	N/A	32.768	N/A

4. Click on the **Kernels & Compute Units** tab of the Profile Summary report

5. Review the Kernel **Total Time (ms)**

   This number will serve as a baseline (reference point) to compare against after optimization. This baseline may be different depending on the target platform

Top Operations | **Kernels & Compute Units** | Data Transfers | OpenCL APIs | Kernel Internals

Kernel Execution (includes estimated device times)

Kernel	Number Of Enqueues	Total Time (ms)	Minimum Time (ms)	Average Time (ms)	Maximum Time (ms)
krnl_idct	4	0.031	0.007	0.008	0.008

Compute Unit Utilization (includes estimated device times)

Device	Compute Unit	Kernel	Global Work Size	Local Work Size	Number Of Calls	Dataflow Execution	Max Parallel Executions	Dataflow Acceleration	Total Time (ms)	Minimum Time (ms)	Average Time (ms)	Maximum Time (ms)	Clock Freq (MHz)	CU Utilization (%
xilinx_aws-vu9p-f1_shell-v04261818_201920_1-0	krnl_idct_1	krnl_idct	1:1:1	1:1:1	4	No	1	1.000000x	0.023	0.006	0.006	0.006	300	73.120

6. In the *Assistant* view, expand *optimization_lab* > *Emulation-HW* > *binary_container_1* > *krnl_idct* and double-click on **Compile Summary (krnl_idct)**

7. The *Vitis Analyzer* window will update and now it includes *krnl_idct (Hardware Emulation)* within *BUILD*

8. Click on **HLS Synthesis** and review it

- krnl_idct
  - krnl_idct_dataflow
    - execute
      - reg_int_s
    - read_blocks_ap_int_512_s
    - write_blocks
    - read_blocks_ap_uint_512_s

Synthesis Report for 'krnl_idct_dataflow'

Vivado HLS Report for 'krnl_idct_dataflow'

Date: Tue Apr 28 21:29:10 2020
Version: 2019.2 (Build 2698951 on Thu Oct 24 19:15:34 MDT 2019)
Project: krnl_idct
Solution: solution
Product family: virtexuplus
Target device: xcvu9p-flgb2104-2-i

Performance Estimates

Timing

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	4.00 ns	2.920 ns	1.08 ns

Latency

Summary

min	max	min	max	min	max	Type
6192	6192	24.768 us	24.768 us	6192	6192	none

Detail

Instance

Instance	Module	min	max	min	max	min	max	Type
grp_execute_fu_106	execute	2075	2075	8.300 us	8.300 us	2075	2075	none
grp_read_blocks_ap_int_512_s_fu_115	read_blocks_ap_int_512_s	2057	2057	8.228 us	8.228 us	2057	2057	none
grp_write_blocks_fu_126	write_blocks	2055	2055	8.220 us	8.220 us	2055	2055	none
grp_read_blocks_ap_uint_512_s_fu_136	read_blocks_ap_uint_512_s	11	114	44.000 ns	44.000 ns	11	11	none

9. In the **Performance Estimates** section, expand the **Latency (clock cycles)** > **Summary** and note the following numbers:

   - Latency (min/max): ~6,000

   The numbers may vary slightly depending on the target hardware you selected.
   The numbers will serve as a baseline for comparison against optimized versions of the kernel

10. In the HLS report, expand **Latency (clock cycles)** > **Detail** > **Instance**

    - Note that the 3 sub-functions read, execute and write have roughly the same latency and that their sum total is equivalent to the total Interval reported in the Summary table
    - This indicates that the three sub-functions are executing sequentially, hinting to an optimization opportunity

11. Close all the reports by selecting **File > Exit**

Analyze the kernel code and apply the DATAFLOW directive

1. Open the **src > krnl_idct.cpp** file

2. Using the **Outline** viewer, navigate to the **krnl_idct_dataflow** function

Observe that the three functions are communicating using **hls::streams** objects. These objects model a FIFO-based communication scheme. This is the recommended coding style which should be used whenever possible to exhibit streaming behavior and allow **DATAFLOW** optimization

3. Enable the DATAFLOW optimization by uncommenting the **#pragma HLS DATAFLOW** present in the krnl_idct_dataflow function (line 319)

   The DATAFLOW optimization allows each of the subsequent functions to execute as independent processes. This results in overlapping and pipelined execution of the read, execute and write functions instead of sequential execution. The FIFO channels between the different processes do not need to buffer the complete dataset anymore but can directly stream the data to the next block.

4. Save the file

Build the project in Hardware Emulation configuration and analyze the HLS report

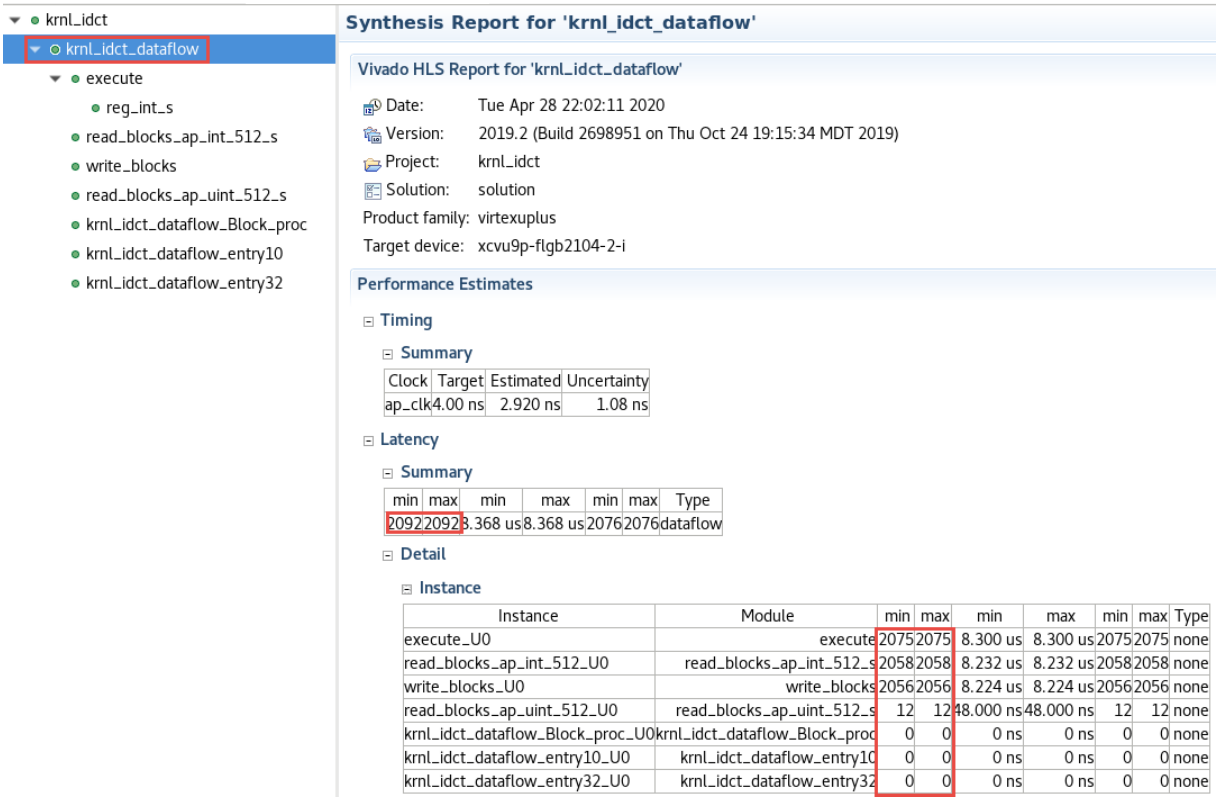
1. Make sure the active configuration is **Emulation-HW**

2. Click on the Build button () to build the project

3. In the *Assistant* view, expand *optimization_lab* > *Emulation-HW* > *binary_container_1* > *krnl_idct* and double-click on **Compile Summary (krnl_idct)**

4. The *Vitis Analyzer* window will update *krnl_idct (Hardware Emulation)* with in *BUILD*

5. Click on **HLS Synthesis** and review it



6. In the **Performance Estimates** section, expand the *Latency (clock cycles)* > *Summary* and note the following numbers:

   - Latency (min/max): ~2,000

Run the Hardware Emulation

1. In a terminal window, execute the following commands to create another xclbin file which will be used to view the waveform

```
cd ~/workspace/optimization_lab/Emulation-HW
~/compute_acceleration/sources/optimization_lab/waveform_patch.tcl binary_container_1.xclbin binary_container_3.xcl
```

   This will create binary_container_3.xclbin which will be used in run_configuration settings

2. Change the run configuration to use *binary_container_3.xclbin* and then run the hardware emulation. Wait for the run to finish with RUN COMPLETE message

   Notice the effect of the dataflow optimization in the Vivado simulation waveform view. Execution of reading, writing, pipelining and kernel is not sequential.

3. In the *Assistant* view, expand *optimization_lab* > *Emulation-HW* > *optimization_lab-Default* and double-click the **Run Summary (xclbin)**

   The *Vitis Analyzer* will update. Click on **xclbin (Hardware Emulation) & Profile Summary** within **RUN**

4. Select the **Kernels & Compute Units** tab.
   Compare the **Kernel Total Time (ms)** with the results from the un-optimized run (numbers may vary slightly to the results displayed below)

Kernel Execution (includes estimated device times)

Kernel	Number Of Enqueues	Total Time (ms)	Minimum Time (ms)	Average Time (ms)	Maximum Time (ms)
krnl_idct	4	0.020	0.005	0.005	0.005

Compute Unit Utilization (includes estimated device times)

Device	Compute Unit	Kernel	Global Work Size	Local Work Size	Number Of Calls	Dataflow Execution	Max Parallel Executions	Dataflow Acceleration	Total Time (ms)	Minimum Time (ms)	Average Time (ms)	Maximum Time (ms)	Clock Freq (MHz)	CU Utilization (%
xilinx_aws-vu9p-f1_shell-v04261818_201920_1-0	krnl_idct_1	krnl_idct	1:1:1	1:1:1	4	No	1	1.000000x	0.009	0.002	0.002	0.002	300	44.220

Analyze the host code

1. Open the **src > idct.cpp** file

2. Using the *Outline* viewer, navigate to the **runFPGA** function

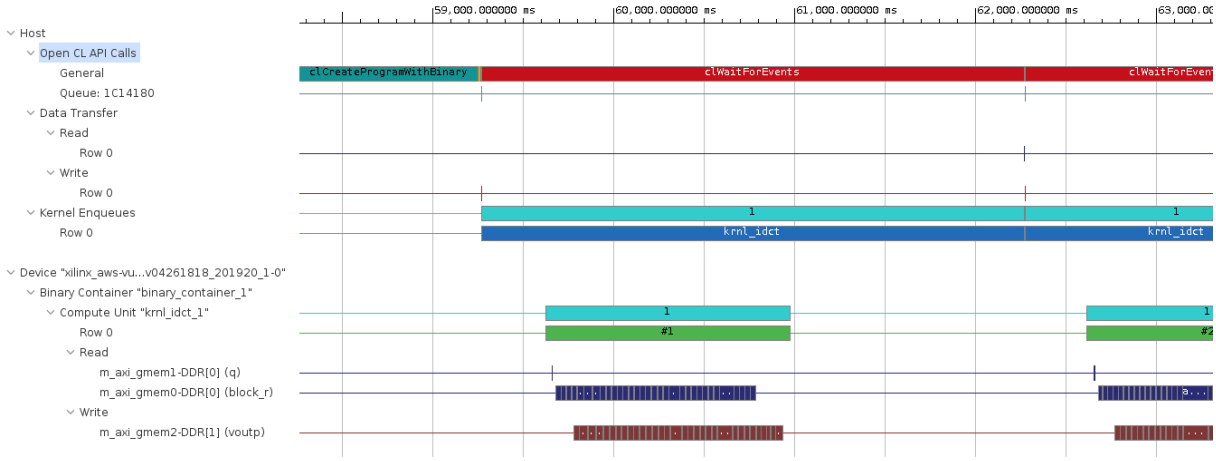
   For each block of 8x8 values, the **runFPGA** function writes data to the FPGA, runs the kernel, and reads results back. Communication with the FPGA is handled by the OpenCL API calls made within the `cu.write()`, `cu.run()` and `cu.read()` function calls

   - `clEnqueueMigrateMemObjects()` schedules the transfer of data to or from the FPGA
   - `clEnqueueTask()` schedules the executing of the kernel

   These OpenCL functions use events to signal their completion and synchronize execution

3. Open the **Application Timeline** of the *Emulation-HW* run in **Vitis Analyzer**.

   The green segments at the bottom indicate when the IDCT kernel is running



Notice that there are gaps between each of the green segments indicating that the operations are not overlapping

4. Zoom in by performing a left mouse drag across one of these gaps to get a more detailed view

   - The two green segments correspond to two consecutive invocations of the IDCT kernel
   - The gap between the two segments is indicative of the kernel idle time between these two invocations
   - The **Data Transfer** section of the timeline shows that **Read** and **Write** operations are happening when the kernel is idle
   - The Read operation is to retrieve the results from the execution which just finished and the Write operation is to send inputs for the next execution
   - This represents a sequential execution flow of each iteration

5. Close the **Application Timeline**

6. In the **idct.cpp** file, go to the `oclDct::write()` function (line ~260)

   Notice on line ~274, the function synchronizes on the **outEvVec** event through a call to `clWaitForEvents()`

```
clWaitForEvents(1, &outEvVec[mCount]);
```

   - This event is generated by the completion of the `clEnqueueMigrateMemObjects()` call in the `oclDct::read()` function (line ~346)
   - Effectively the next execution of the `oclDct::write()` function is gated by the completion of the previous `oclDct::read()` function, resulting in the sequential behavior observed in the **Application Timeline**

7. Use the **Outline** viewer to locate the definition of the **NUM_SCHED** macro in the **idct.cpp** file

   - This macro defines the depth of the event queue
   - The value of 1 explains the observed behavior: new tasks (write, run, read) are only enqueued when the previous has completed effectively synchronizing each loop iteration
   - By increasing the value of the **NUM_SCHED** macro, we increase the depth of the event queue and enable more blocks to be enqueued for processing, which may result in the write, run and read tasks to overlap and allow the kernel to execute continuously or at least more frequently
   - This technique is called software pipelining

8. Modify line 152 to increase the value of **NUM_SCHED** to 6 as follows

```
#define NUM_SCHED 6
```

9. Save the file

Run Hardware Emulation

1. Build the application by clicking on the ( ) button

2. Change the run configuration by unchecking the **Use waveform for kernel debugging** option, click **Apply**, and then click **Close**

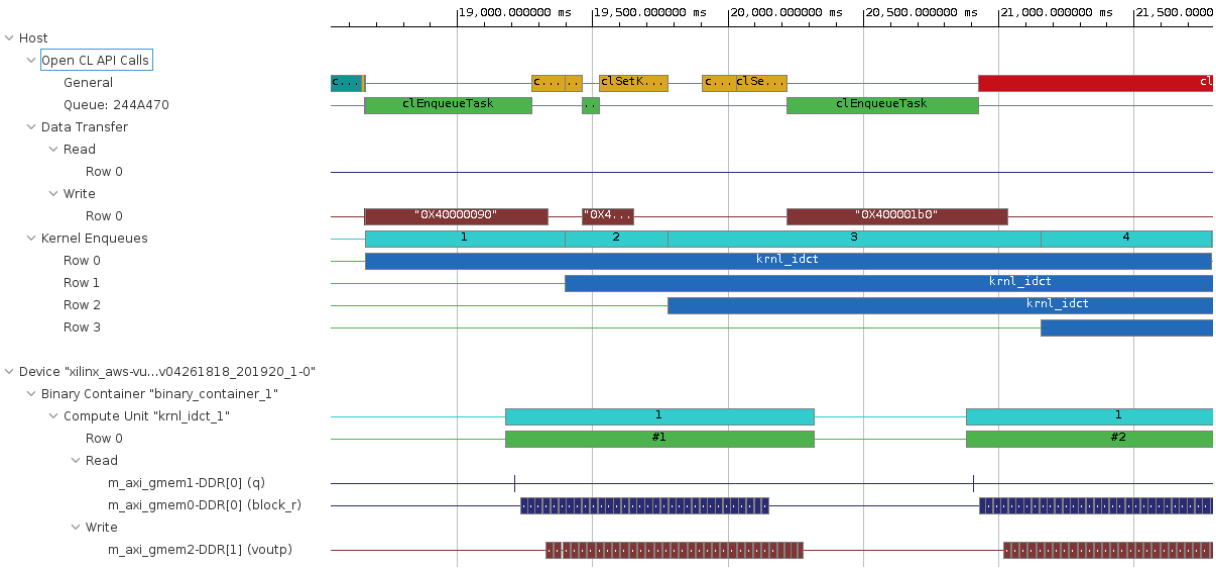
3. Run the application by clicking the Run button ( )
   Since only the idct.cpp file was changed, the incremental makefile rebuilds only the host code before running emulation

   This should be much faster as no recompiling the kernel to hardware is needed

4. In the *Assistant* view, expand *optimization_lab* > *Emulation-HW* > *optimization_lab-Default* and click on **Run Summary (xclbin)**

5. On *Vitis Analyzer* window click on **xclbin (Hardware Emulation) > Application Timeline** report within **RUN**

   Observe how **software pipelining** enables overlapping of data transfers and kernel execution.



   Note: system tasks might slow down communication between the application and the hardware simulation, impacting on the performance results. The effect of software pipelining is considerably higher when running on the actual hardware.

Run the Application in hardware

As before, building the FPGA hardware takes some time, and a precompiled solution is provided.

1. Set *Active build configuration:* to **Hardware**

2. Change to the solution directory by executing the following command

```
cd ~/compute_acceleration/solutions/optimization_lab
```

3. Copy the *binary_container_1.xclbin*, *binary_container_1.awsxclbin*, and *optimization_lab* files into `~/workspace/optimization_lab/Hardware` folder. Make sure *optimization_lab* has executable permissions. Use the following commands:

```
cp ~/compute_acceleration/solutions/optimization_lab/* ~/workspace/optimization_lab/Hardware/.
chmod +x ~/workspace/optimization_lab/Hardware/optimization_lab
```

Setup the run configuration so you can run the application and then analyze results from GUI

4. Right-click on Hardware in *Assistant* view, select `Run > Run Configurations...`

   Change Generate timeline trace report option from *Default* to *Yes* using the drop-down button in the Main tab.

5. Click **Arguments** tab, uncheck the *Automatically add binary container(s) to arguments* option, and then enter `../binary_container_1.awsxclbin`

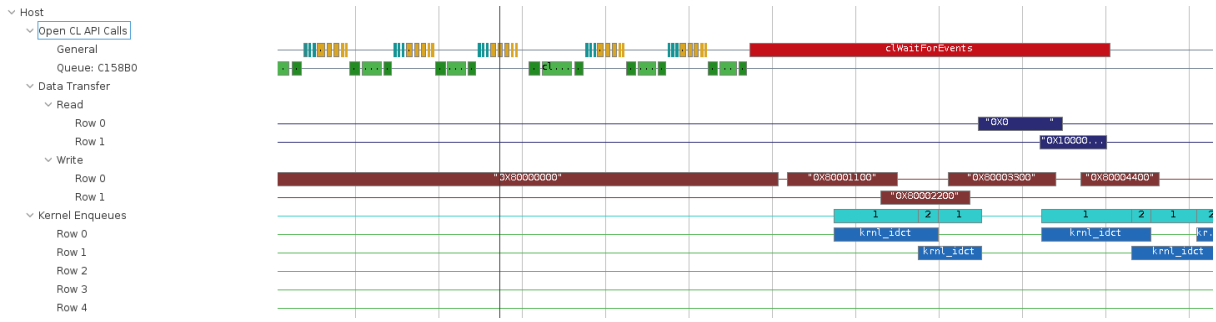
6. Execute the application by clicking on **Apply** and then **Run**. The FPGA bitstream will be downloaded and the host application will be executed showing output similar to:

```
[Console output redirected to file:/home/centos/workspace/optimization_lab/Hardware/optimization_lab-Default.launch.log]
FPGA number of 64*int16_t blocks per transfer: 512
DEVICE: xilinx_aws-vu9p-f1_dynamic_5_0
Loading Bitstream: ../binary_container_1.awsxclbin
INFO: Loaded file
Create Kernel: krnl_idct
Create Compute Unit
Setup complete
Running CPU version
Running FPGA version
Runs complete validating results
TEST PASSED
CPU Time:        0.792981 s
CPU Throughput:  645.665 MB/s
FPGA Time:       5.09267 s
FPGA Throughput: 100.537 MB/s
FPGA PCIe Throughput: 201.073 MB/s
```

Analyze hardware application timeline and profile summary

1. In the *Assistant* view, double click `Hardware > optimization_lab-Default > Run Summary (xclbin)` to open Vitis Analyzer

   Vitis Analyzer shows **Run Guidance**, **Profile Summary** and **Application Timeline** panels on the left-hand side. Click **Application Timeline**. Zoom in between 185,800,000 and 186,800,000 microsecond area (note for your output the range may differ depending on what else was executed on the instance) and observe the activities in various parts of the system. Note that the kernel proceses data in one shot



2. Click on the *Profile Summary* entry in the left panel, and observe multi-tab (four tabs) output

   - Top Operations

Top Operations | Kernels & Compute Units | Data Transfers | OpenCL APIs

Top Data Transfer: Kernels to Global Memory

No Data. Please use 'v++ -l --profile_kernel' to monitor and report kernel data tra

Top Kernel Execution

Kernel Instance Address	Kernel	Context ID	Command Queue ID	Device	Start Time (ms)	Duration (ms)	Global Work Size	Local Work Size
0xc15650	krnl_idct	0	0	xilinx_aws-vu9p-f1_dynamic_5_0-0	9269.420	2.287	1:1:1	1:1:1
0xc15650	krnl_idct	0	0	xilinx_aws-vu9p-f1_dynamic_5_0-0	8981.510	2.027	1:1:1	1:1:1
0xc15650	krnl_idct	0	0	xilinx_aws-vu9p-f1_dynamic_5_0-0	9269.940	1.859	1:1:1	1:1:1
0xc15650	krnl_idct	0	0	xilinx_aws-vu9p-f1_dynamic_5_0-0	6686.810	1.474	1:1:1	1:1:1
0xc15650	krnl_idct	0	0	xilinx_aws-vu9p-f1_dynamic_5_0-0	4676.150	1.426	1:1:1	1:1:1
0xc15650	krnl_idct	0	0	xilinx_aws-vu9p-f1_dynamic_5_0-0	9270.470	1.421	1:1:1	1:1:1
0xc15650	krnl_idct	0	0	xilinx_aws-vu9p-f1_dynamic_5_0-0	9270.870	1.321	1:1:1	1:1:1
0xc15650	krnl_idct	0	0	xilinx_aws-vu9p-f1_dynamic_5_0-0	6817.170	1.264	1:1:1	1:1:1
0xc15650	krnl_idct	0	0	xilinx_aws-vu9p-f1_dynamic_5_0-0	6690.880	1.230	1:1:1	1:1:1
0xc15650	krnl_idct	0	0	xilinx_aws-vu9p-f1_dynamic_5_0-0	4625.540	1.217	1:1:1	1:1:1

Top Memory Writes: Host to Global Memory

Buffer Address	Context ID	Command Queue ID	Start Time (ms)	Duration (ms)	Buffer Size (KB)	Writing Rate (MB/s)
0x800055000	0	0	8969.610	10.674	65.664	6.152
0x800000000	0	0	4586.830	8.989	65.664	7.305
0x800033000	0	0	9498.450	5.789	65.664	11.343
0x800011000	0	0	8426.670	5.580	65.664	11.767
0x800022000	0	0	8427.050	5.275	65.664	12.449
0x800022000	0	0	6008.070	5.204	65.664	12.619
0x800011000	0	0	4959.540	4.691	65.664	13.997
0x800022000	0	0	8385.590	4.677	65.664	14.041
0x800011000	0	0	8171.610	4.652	65.664	14.115
0x800000000	0	0	7785.750	4.537	65.664	14.474

Top Memory Reads: Host to Global Memory

Buffer Address	Context ID	Command Queue ID	Start Time (ms)	Duration (ms)	Buffer Size (KB)	Reading Rate (MB/s)
0x40000	0	0	8969.430	10.833	65.536	6.050
0x10000	0	0	6007.940	5.319	65.536	12.322
0x0	0	0	4959.430	4.779	65.536	13.713
0x10000	0	0	8385.560	4.706	65.536	13.927
0x0	0	0	8171.600	4.670	65.536	14.034
0x50000	0	0	7785.670	4.581	65.536	14.305
0x30000	0	0	7275.810	4.437	65.536	14.771
0x10000	0	0	9607.920	4.351	65.536	15.063
0x50000	0	0	4671.110	3.936	65.536	16.652
0x30000	0	0	9476.720	2.754	65.536	23.798

- Kernels & Compute Units

Top Operations | **Kernels & Compute Units** | Data Transfers | OpenCL APIs

Kernel Execution

Kernel	Number Of Enqueues	Total Time (ms)	Minimum Time (ms)	Average Time (ms)	Maximum Time (ms)
krnl_idct	8192	1149.630	0.075	0.140	2.287

- Data Transfers

Top Operations | Kernels & Compute Units | **Data Transfers** | OpenCL APIs

Data Transfer: Host to Global Memory

Context:Number of Devices	Transfer Type	Number Of Buffer Transfers	Transfer Rate (MB/s)	Average Bandwidth Utilization (%)	Average Buffer Size (KB)	Total Time (ms)	Average Time (ms)
context0:1	READ	8192	375.907	3.916	65.536	1428.201	0.174
context0:1	WRITE	8192	349.744	3.643	65.664	1538.036	0.188

3. When finished, close the analyzer by clicking `File > Exit` and clicking **OK**

Conclusion

In this lab, you used Vitis to create a project and add a kernel (hardware) function. You performed software and hardware emulation, analyzed the design and the various reports generated by the tools. You then optimized the kernel code using the DATAFLOW pragma, and host code by increasing the number of read, write, and run tasks to improve throughput and data transfer rates. You then validated the functionality in hardware.

Start the next lab: RTL-Kernel Wizard Lab

Appendix Build Full Hardware

Set the build configuration to Hardware and build the system (Note that since the building of the project takes over two hours skip this step in the workshop environment).

- Click on the drop-down button of *Active build configuration:* and select **Hardware**
- Set the Vitis Kernel Linker flag as before.
- Either select **Project > Build Project** or click on the () button.
  This will build the project under the **Hardware** directory. The built project will include **optimization_lab** file along with **binary_container_1.xclbin** file. This step takes about two hours

AWS-F1

Once the full system is built, you should create an AWS F1 AFI to run this example in AWS-F1.