Optimizing the Performance of a Design

2022.1

Abstract

You will explore various optimization methods to improve the performance of a design in this lab.

This lab should take approximately 100-120 minutes depending on whether you use the prebuilt project or your own project.

CloudShare Users Only

You are provided three attempts to access a lab, and the time allotted to complete each lab is 2X the time expected to complete the lab. Once the timer starts, you cannot pause the timer. Also, each lab attempt will reset the previous attempt—that is, your work from a previous attempt is not saved.

Objectives

After completing this lab, you will be able to:

- Obtain the baseline performance of a design
- Optimize data transfer using the `clEnqueueMigrateMemObjects` API
- Run kernels in parallel
- Verify how the tools optimize a C++ kernel using automatic burst data transfer

Introduction

The application used in this lab processes vectors of different sizes and its flow can be represented by the figure below.

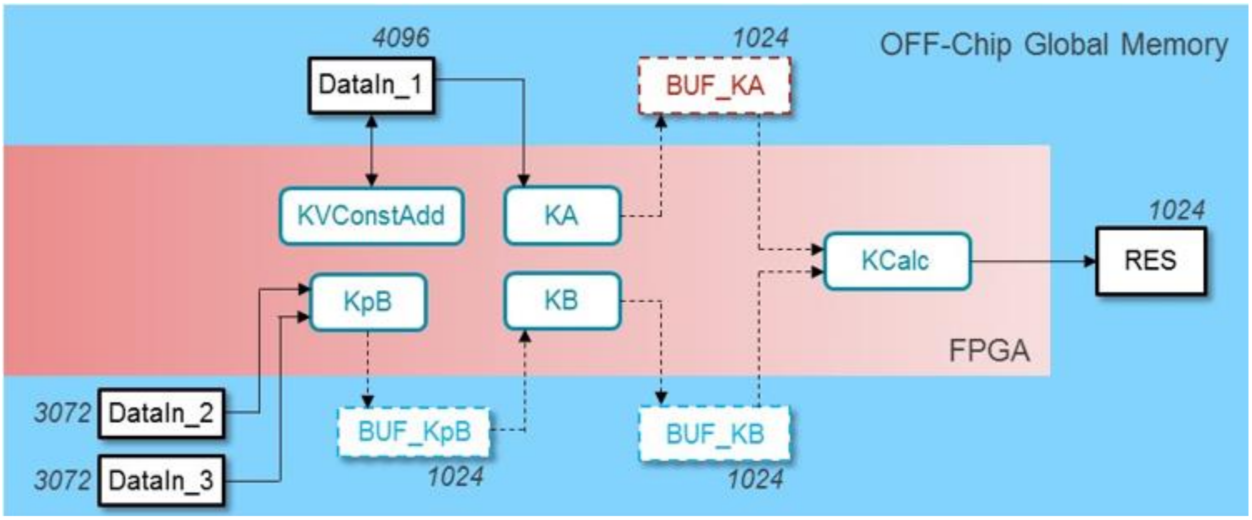


Figure 11-1: Design Overview

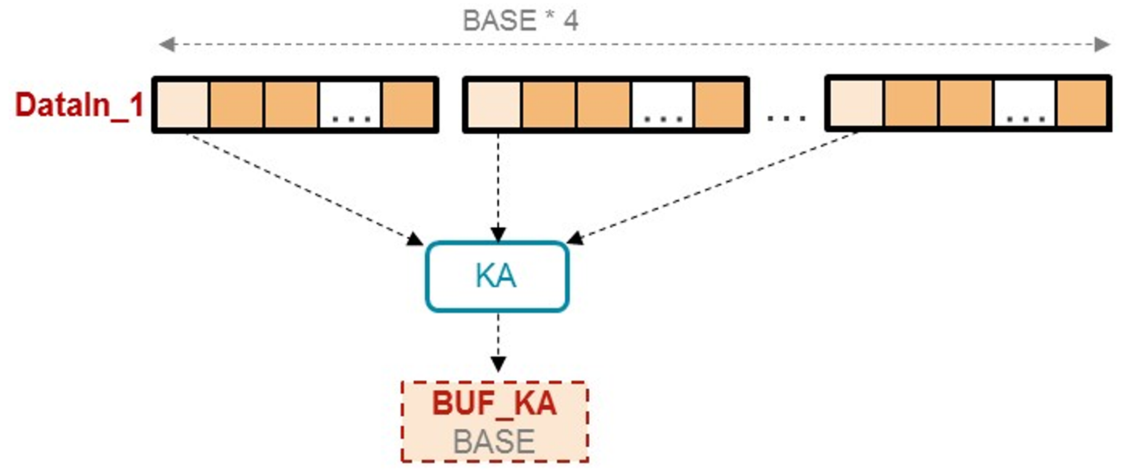
The application contains five simple kernels: **KVConstAdd**, **KpB**, **KA**, **KB**, and **KCalc**. The inputs to the application are the *DataIn_1*, *DataIn_2,* and *DataIn_3* input vectors, and the generated result is **RES**. The intermediate results (**BUF_KpB**, **BUF_KA**, and **BUF_KB**) generated by **KpB**, **KA**, and **KB** are stored in off-chip global memory.

The application defines the vector size using the macro **BASE**; i.e., BASE=1024 in the *kernel.h* file.

- **RES**, **BUF_KA**, and **BUF_KB** buffers all contain 1024 elements.
- **BUF_KpB** contains 3072 elements.
- The size of *DataIn_1* is defined as SIZE_DataIn_1=BASE*4.
- The size of *DataIn_2* and *DataIn_3* is defined as SIZE_DataIn_2=BASE*3 and SIZE_DataIn_3=BASE*3 .

KVConstAdd kernel (RTL): This is a pre-generated RTL kernel and implements a vector addition function with a constant **A[i] = A[i] + Const**. The constant value is set to 5 in the host program. At the beginning, *DataIn_1* is processed and updated by the **KVConstAdd** kernel. When **KVConstAdd** completes its work, **KA** reads *DataIn_1* and generates intermediate results, which are stored in the **BUF_KA** array (1024 integer values), located in the global memory.

KA kernel (C++ kernel): This is written in C and is located in the *K_KA.cpp* file. The kernel behavior is represented below. As you can see, it scans the *DataIn_1* vector and calculates a single **BUF_KA** cell using four *DataIn_1* cells.




```
extern "C" {
void KA(int *A, int *R)
{
      #pragma HLS INTERFACE m_axi port=A offset=slave bundle=gmem
      #pragma HLS INTERFACE m_axi port=R offset=slave bundle=gmem
      #pragma HLS INTERFACE s_axilite port=A bundle=control
      #pragma HLS INTERFACE s_axilite port=R bundle=control
      #pragma HLS INTERFACE s_axilite port=return bundle=control
      loop_st_1:
      for (int i=0; i<SIZE_BUF_KA; i++) {
            R[i] = A[2*i]*3 + A[2*i+1]*5 + A[2*i+2]*7 + A[2*i+3]*9;
      }
}
}
```

Figure 11-2: KA Kernel Behavior (C++)

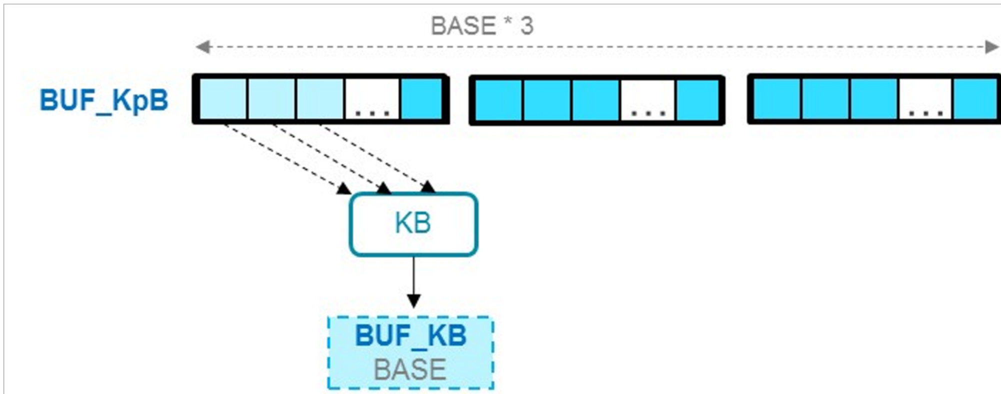
KpB kernel (C++): This is written in C++ and is located in the *K_KpB_1.cpp* file.The code below shows where the first loop in the code performs vector addition and the second loop performs a modulo operation to each result of the first loop.

```
extern "C" {
void KpB(int *A, int *B, int *R) {

        int TMP_RES[SIZE_BUF_KpB];

        #pragma HLS INTERFACE s_axilite port=A bundle=control
        #pragma HLS INTERFACE s_axilite port=B bundle=control
        #pragma HLS INTERFACE s_axilite port=R bundle=control
        #pragma HLS INTERFACE s_axilite port=return bundle=control

        #pragma HLS INTERFACE m_axi port=A  offset=slave bundle=gmem
        #pragma HLS INTERFACE m_axi port=B  offset=slave bundle=gmem
        #pragma HLS INTERFACE m_axi port=R  offset=slave bundle=gmem

        for(int i=0; i < SIZE_BUF_KpB; i+=1) {
                TMP_RES[i] = A[i] + B[i];
        }

        for(int i=0; i < SIZE_BUF_KpB; i+=1) {
                R[i] = TMP_RES[i] % 3;
        }
}
}
```

Figure 11-3: KpB Kernel Behavior (C++)

The **KpB** kernel (in parallel with the **KVConstAdd** kernel) reads *DataIn_2* and *DataIn_3* and stores the generated results in the **BUF_KpB** buffer located in the global memory. Then the **KB** kernel processes **BUF_KpB** and generates **BUF_KB** (also located in the global memory).

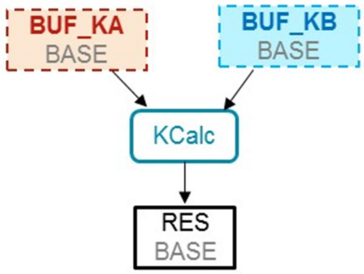
KB kernel (C++): This is written in C++ and is located in the *K_KB.cpp* file. Below is the source code and the representation of the KB kernel.




```
extern "C" {
void KB(int *A, int *R)
{
      #pragma HLS INTERFACE m_axi port=A offset=slave bundle=gmem
      #pragma HLS INTERFACE m_axi port=R offset=slave bundle=gmem
      #pragma HLS INTERFACE s_axilite port=A bundle=control
      #pragma HLS INTERFACE s_axilite port=R bundle=control
      #pragma HLS INTERFACE s_axilite port=return bundle=control

      int val;

      int TMP_BUF[SIZE_BUF_KpB];

      for (int i=0; i<SIZE_BUF_KpB; i++) {
            TMP_BUF[i] = A[i]+10;
      }

      for (int i=0; i<SIZE_BUF_KB; i++) {
            val  =  TMP_BUF[i]*3 + TMP_BUF[i+SIZE_RES]*5 + TMP_BUF[i+2*SIZE_RES]*7;
            R[i] = val;
      }

}
}
```

Figure 11-4: KB Kernel Behavior (C++)

KCalc kernel (C++): This is written in C++ and is located in the *K_KCalc.cpp* file. As soon as the data **BUF_KA** and **BUF_KB** are ready, the data will be processed by the **KCalc** kernel and the final results **RES** are generated. Below is the source code and the representation of the KCalc kernel.




```
extern "C" {
void KCalc(int *A, int *B, int *R)
{
      #pragma HLS INTERFACE m_axi port=A offset=slave bundle=gmem
      #pragma HLS INTERFACE m_axi port=B offset=slave bundle=gmem
      #pragma HLS INTERFACE m_axi port=R offset=slave bundle=gmem
      #pragma HLS INTERFACE s_axilite port=A bundle=control
      #pragma HLS INTERFACE s_axilite port=B bundle=control
      #pragma HLS INTERFACE s_axilite port=R bundle=control
      #pragma HLS INTERFACE s_axilite port=return bundle=control

      int val1, val2;
       int TMP_R[SIZE_RES];

      int TMP_A[SIZE_RES], TMP_B[SIZE_RES];
      for (int i=0; i<SIZE_RES; i++) {
            #pragma HLS PIPELINE
            TMP_A[i] = A[i]; TMP_B[i] = B[i];
      }


      for (int i=0; i<SIZE_RES; i++) {
            val1 = (TMP_A[i] - TMP_B[i]) * (TMP_A[i] + TMP_B[i]);

            if (val1 >= 0)
                val2 = val1 % 3;
            else
                  val2 = (val1 % 6) * val1;

            TMP_R[i] = val2;
      }

      for (int i=0; i<SIZE_RES; i++) {
            #pragma HLS PIPELINE
            R[i] = TMP_R[i];
      }
}
}
```

Figure 11-5: KCalc Kernel Behavior (C++)

The overall application structure is represented in the figure below.

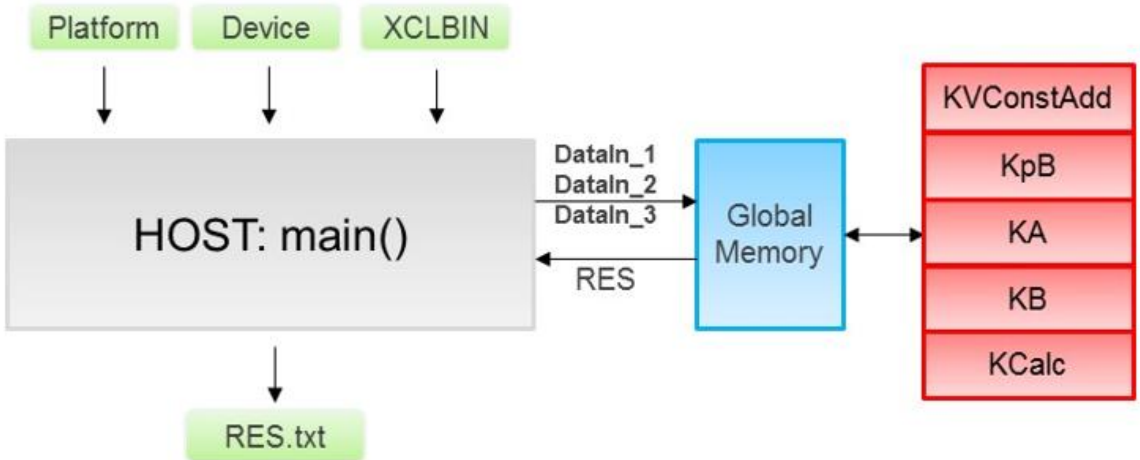


Figure 11-6: Overall Application Structure

The host has three input arguments:

- **Platform**: The name of the platform vendor.
  - In this lab, the vendor is Xilinx.
- **Device**: The target platform device.
  - In this lab, `xilinx_u50_gen3x16_xdma_201920_3` is used.
- **XCLBIN**: The name of the Xilinx binary container where all the kernels is precompiled.
  - In this lab, the binary container is the `kernels.<TARGET>.xclbin` file.

`<TARGET>` is **sw_emu** for the software emulation build configuration and **hw_emu** for the hardware emulation build configuration.

The application creates the **RES.txt** output file containing the results generated by the `K_Calc` kernel.

Understanding the Lab Environment

Customizable environment variables enable you to tailor your environment for specific machine configurations. The only environment variable (shown below) used in the customer training environment (CustEd_VM) points to the training directory where all the lab files are located.

This environment variable can be customized according to your specific location and can be set for Linux systems in the `/etc/profile` file.

The following is the environment variable used in the customer training VM:

Environment Variable Name	Description
`$TRAINING_PATH`	Points to the space allocated for students to work through their labs. This directory includes prebuilt images and starting points for the labs and demos. In the customer training VM, `$TRAINING_PATH` sets to the `/home/xilinx/training` directory.

Note: Environment variables are not supported by the Vitis IDE GUI. When using this tool, you must manually replace **`$TRAINING_PATH`** with the value of the variable, which in the customer training virtual machine, is **`/home/xilinx/training`**.

For Cloud development: Similarly, the customer training environment (CustEd_VM) sets the XRT tool install path to `/opt/xilinx/xrt`. Make sure that XRT (2022.1 version) is installed in your environment.

General Flow

Step 1: Reviewing the Projects → Step 2: Baselining the Design → Step 3: Using the Migrate Memory API → Step 4: Migrating Multiple Buffers → Step 5: Running Kernels in Parallel

Reviewing the Projects Step 1

You are provided a collection of projects representing different types of optimization. It's time to get familiar with these projects.

1-1. Configure the environment to support the Vitis IDE tool for accelerator development.

Developing accelerators, as opposed to traditional software development, requires additional libraries.

1-1-1. Either use an existing terminal window if available or press <**Ctrl** + **Alt** + **T**> to open a new terminal window.

1-1-2. Enter the following commands to source the Xilinx XRT and Vitis tools:

```
[host]$ source /opt/xilinx/xrt/setup.sh
[host]$ source /opt/Xilinx/Vitis/2022.1/settings64.sh
```

Note: The customer training environment (CustEd_VM) sets the Vitis tool install path to `/opt/Xilinx/Vitis`. If the tool is installed in a different location in your environment, use that install path.

For Cloud development: Similarly, the customer training environment (CustEd_VM) sets the XRT tool install path to `/opt/xilinx/xrt`. Make sure that XRT (`2022.1` version) is installed in your environment.

1-2. Briefly review the description of the lab projects.

1-2-1. Enter the following commands to change the path to the lab directory and view the different project directories:

```
[host]$ cd $TRAINING_PATH/accel_optimization/lab/makefiles
[host]$ ls
```

You will find the makefiles listed in the table below. These makefiles will create four different projects. The table also describes what you will be doing for each project in this lab.

Note: If your lab files are pointing to a different location, set the `ROOT_REPO` variable as appropriate in the makefile.

Project Name	Description
opt_1_baselining	Understand the performance of the application before starting any optimization effort (baselining).
opt_2_clenqueuem_api	Use the `clEnqueueMigrateMemObjects` API for data transfer.

Project Name	Description
opt_3_single_api	Modify the host code to minimize the number of `clEnqueueMigrateMemObjects` API calls.
opt_4_kernel_parallel	Modify the host code to run the kernels in parallel.

Baselining the Design Step 2

Gain insight into the design's behavior and performance by baselining the application. This will provide a reference from which to measure the effectiveness of the optimization techniques.

2-1. Review the source files in the *opt_1_baselining* project.

2-1-1. Enter the following command to change the path to the source directory:

```
[host]$ cd $TRAINING_PATH/accel_optimization/lab/src/
opt_1_baselining
```

2-1-2. Review the following files:

- `host_1.cpp`: Runs on the host and feeds the kernel
- `kernel.h`: Supports the kernel code modules
- `K_KA.cpp`: Contains the kernel code and buffer loader
- `K_KB.cpp`: Contains the kernel code and buffer loader
- `K_KCalc.cpp`: Contains the kernel code and does the math
- `K_KpB_1.cpp`: Contains the kernel code

These files make up the source code used for the first run through the compilation and emulation to form the baseline.

You can chose to use **`gedit <filename>`** to open the file with an editor, which provides search and scrolling capabilities, or **`more <filename>`**, which dumps the code to the screen page by page.

2-2. Review the Makefile.

2-2-1. Change the directory to review the Makefile and compile for software emulation:

```
[host]$ cd $TRAINING_PATH/accel_optimization/lab/makefiles/
opt_1_baselining
```

2-2-2. Enter the following command to open the Makefile and review it:

```
[host]$ gedit Makefile
```

Notice the default settings for the following variables:

- o PLATFORM_VENDOR = Xilinx
- o PLATFORM = xilinx_u50_gen3x16_xdma_5_202210_1
- o TARGET = sw_emu

These represent the default settings.

Review the other sections of the Makefile, such as building the host executable, kernel compilation/linking, and running the application.

Note: If your lab files are in a location other than the training folder, modify the `ROOT_REPO` variable as appropriate and save the file.

2-2-3. Press <**Ctrl** + **Q**> to close the editor.

2-3. Compile for software emulation.

2-3-1. Change the directory to where the compiled files will be generated:

[host]$ cd $TRAINING_PATH/accel_optimization/lab/makefiles/opt_1_baselining

2-3-2. Enter the following command to compile the design in software emulation:

[host]$ make build [TARGET=sw_emu]

Because by default `TARGET` is already set to `sw_emu`, the optional `TARGET=sw_emu` is redundant.

Note: Ignore any warnings.

A new directory `build` has been created and there you will find the target build configuration (`sw_emu`) directory, which contains all the build files.

The `host.exe` file represents the host executable and is required. You will also often find an `xclbin` file that contains the kernels.

The `emconfigutil` utility generates the `emconfig.json` file, which contains information about the target device. This file is used for the emulation flow.

The `make build` calls the host compilation, kernel compilation, and `emconfig` utilities.

```
# Build the design without running host application
build: $(BUILD_DIR)/$(HOST_EXE) $(BUILD_DIR)/$(XCLBIN) $(BUILD_DIR)/$(EMCONFIG_FILE)
```

Figure 11-7: Building the Host Executable, XCLBIN, and emconfig.json File

2-4. Review the application arguments and run the application for software emulation.

2-4-1. Enter the following command to open and review the makefile:

`[host]$ gedit Makefile`

```
# Build the design without running host application
build: $(BUILD_DIR)/$(HOST_EXE) $(BUILD_DIR)/$(XCLBIN) $(BUILD_DIR)/$(EMCONFIG_FILE)

# Build the design and then run host application
run: build
       pwd
       cp xrt.ini $(BUILD_DIR);
ifeq ($(TARGET), hw)
       cd $(BUILD_DIR) && unset XCL_EMULATION_MODE;    ./$(HOST_EXE) $(XCLBIN) ;
else
       cd $(BUILD_DIR) && XCL_EMULATION_MODE=$(TARGET) ./$(HOST_EXE) $(PLATFORM_VENDOR)
$(PLATFORM) $(XCLBIN)
endif
```

Figure 11-8: Reviewing the Run Command

Observe that the arguments are `PLATFORM_VENDOR`, `PLATFORM`, and `XCLBIN`.

2-4-2. Enter the following command to run the application in software emulation mode:

`[host]$ make run`

You should see that the application runs successfully, showing the following in the terminal:

Figure 11-9: Application Output - Software Emulation - opt_1_baselining - Example

As you can see, the test has been completed successfully. At the end, the host code generates additional profiling information (this is done using an OpenCL profiling API). You will use this profiling data to monitor the progress of the optimizations performed in this lab.

Note that the values generated in this part of the report become more meaningful at the hardware emulation step.

Building the hardware emulation in the VirtualBox environment takes approximately 10 minutes (based on your system configuration).

If you have time, you can follow step 2-5; otherwise, skip to step 2-6 to use a prebuilt project.

2-5. Compile for hardware emulation.

2-5-1. Ensure that you are in the `makefiles/opt_1_baselining` directory.

If not, enter the following command:

```
[host]$ cd $TRAINING_PATH/accel_optimization/lab/makefiles/
opt_1_baselining
```

2-5-2. Enter the following command to compile the design in hardware emulation:

```
[host]$ make build TARGET=hw_emu
```

Under the `build/opt_1_baselining` directory, you will see one more directory in the name of the target build configuration (`hw_emu`). You will find all the build files under the `hw_emu` directory.

Notice that two files should be generated:

- `host.exe` (host executable)
- `kernels.hw_emu.xclbin` (binary container)

The `emconfig.json` file has also been generated. The `emconfigutil` utility generates the `emconfig.json` file, which contains information about the target device. This file is used for the emulation flow.

The `make build` calls the host compilation, kernel compilation, and `emconfig` utility.

2-6. Run the application in hardware emulation.

2-6-1. Enter the following command to run the application in hardware emulation mode:

[Own project]: **`[host]$ make run TARGET=hw_emu`**

[Prebuilt project]: **`[host]$ make run TARGET=hw_emu PREBUILT=YES`**

Note: This may take 5-6 minutes to complete (using OpenCL profiling APIs to collect the customizable information).

If you receive the "Permission denied" message when you run the prebuilt project, change the path to the `support` directory and change the permission settings by using the `chmod` command as shown below:

```
[host]$ cd $TRAINING_PATH/accel_optimization/support/
prebuilt_opt_1
```

```
[host]$ chmod 777 host.exe
```

Change the path back to the `makefiles` directory and rerun the application for the prebuilt project:

```
[host]$ cd $TRAINING_PATH/accel_optimization/lab/makefiles/
opt_1_baselining
```

You should see that the application runs successfully.

The profiling information is displayed in the terminal as shown below. Note that the timing values may be different for you due to different PC configurations (processor speed, for example).

```
HOST-Info: ============================================================
HOST-Info: (Step 7) Custom Profiling
HOST-Info: ============================================================
HOST-Info: ------------------------------------------------------------------------------
HOST-Info: Name            | type          | start         | end           | Duration(ms)
HOST-Info: ------------------------------------------------------------------------------
HOST-Info: K_KVConstAdd    | kernel        |   60144286891 |   64147548604 |      4003.26
HOST-Info: K_KpB           | kernel        |   76169523005 |  111180151247 |      35010.6
HOST-Info: K_KA            | kernel        |   64147812972 |   76151487785 |      12003.7
HOST-Info: K_KB            | kernel        |  111180546220 |  129184563051 |        18004
HOST-Info: K_KCalc         | kernel        |  129185035884 |  147189434615 |      18004.4
HOST-Info: Transfer_1      | mem (H<->G)   |   60113245285 |   60144180756 |      30.9355
HOST-Info: Transfer_2      | mem (H<->G)   |   60113246698 |   60130403137 |      17.1564
HOST-Info: Transfer_3      | mem (H<->G)   |   60130441379 |   60130521323 |     0.079944
HOST-Info: Transfer_4      | mem (H<->G)   |   60113245033 |   60113245033 |            0
HOST-Info: Transfer_5      | mem (H<->G)   |   60113253807 |   60113253807 |            0
HOST-Info: Transfer_6      | mem (H<->G)   |   60113261227 |   60113261227 |            0
HOST-Info: Transfer_7      | mem (H<->G)   |   60113268686 |   60113268686 |            0
HOST-Info: Transfer_8      | mem (H<->G)   |  147190112156 |  147190846032 |     0.733876
HOST-Info: ------------------------------------------------------------------------------
HOST-Info:     Kernels Execution Time (ms) :  87045.1  (K_KCalc'end - K_KVConstAdd'begin)
HOST-Info: Application Execution Time (ms) :  87077.6  (Transfer_8'end - Transfer_4'begin)
HOST-Info: ------------------------------------------------------------------------------
```

Figure 11-10: Custom Profiling Information (Hardware Emulation) - opt_1_baselining

Let's understand the profiling information in the Console window. For that you need to open the Timeline Trace report.

2-7. Analyze the reports using the Vitis analyzer.

2-7-1. Enter the following command to see the reports generated for the hardware emulation:

[Own project]: `[host]$ make view_run_summary TARGET=hw_emu`

[Prebuilt project]: `[host]$ make view_run_summary TARGET=hw_emu PREBUILT=YES`

2-7-2. Expand the **xrt (Hardware Emulator)** entry in the left pane of the Vitis analyzer to see the various reports.

All references to the reports are relative to the expanded xrt (Hardware Emulation).

2-7-3. Click **Profile Summary** to view the Summary report.

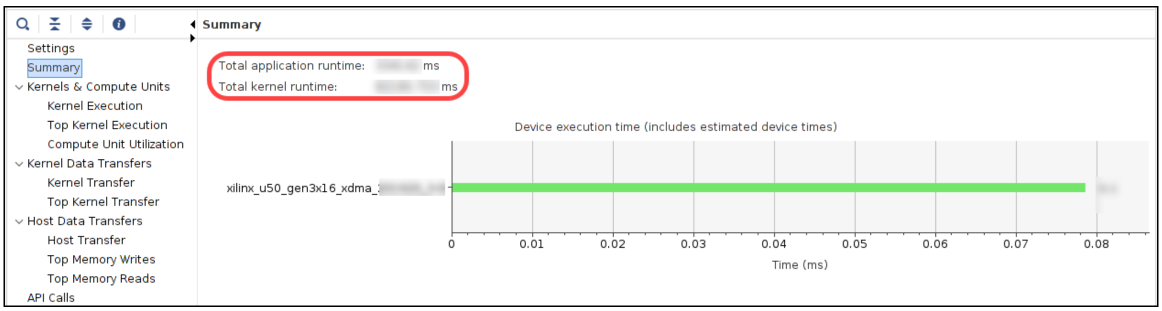


Figure 11-11: Application Runtime and Kernel Runtime (opt_1_baselining Project)

Observe the total application runtime and kernel runtime values. Compare with the profiling information from the print message.

Total application runtime shows the complete execution of the program, covering even the releasing of the objects and all other parts of the main code until the end. In OpenCL profiling, time is calculated based on the events after the memory transfer completed from the global memory to host memory. This is the reason you will see differences in the total application runtime.

2-7-4. Click **Timeline Trace** to view the Timeline Trace report.

2-7-5. Zoom in on the region shortly after clFinish begins (roughly between 20 and 30 seconds).

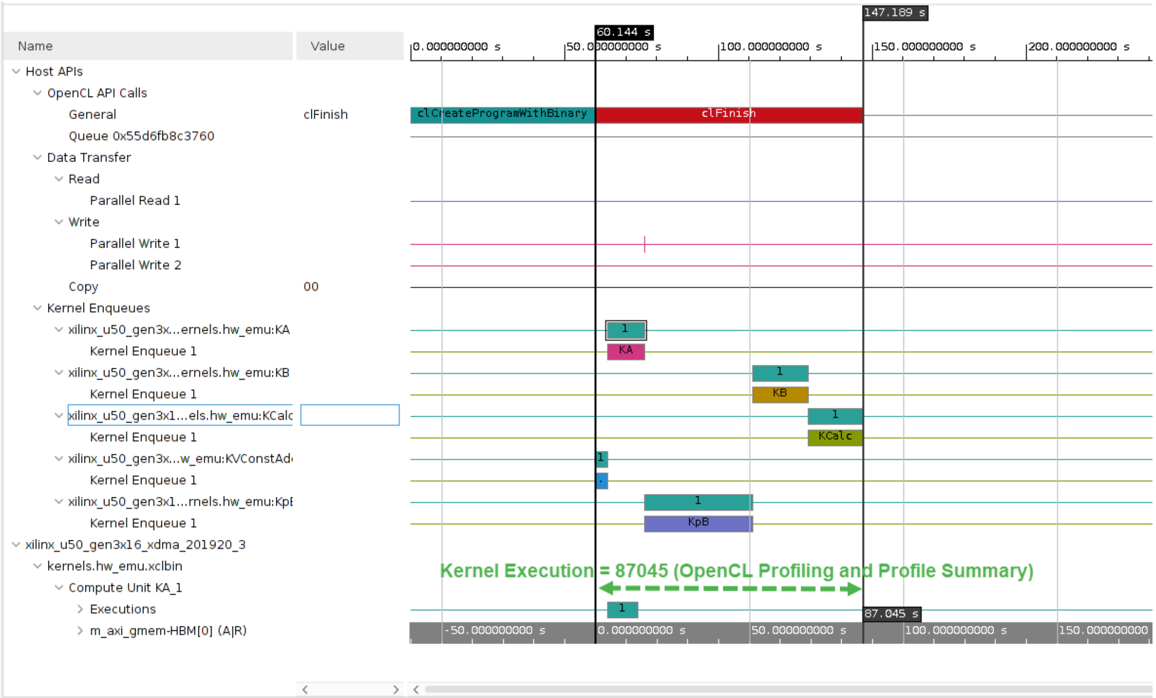


Figure 11-12: Timeline Trace Report (opt_1_baselining Project) - Profile Summary (Kernel)

2-7-6. Click the right edge of the block labeled KVConstAdd in the line Kernel Enqueues > xilinx_u50..._emu:KVConstAdd > Kernel Enqueue 1 to see the blue flow lines.

Hint: Clicking between the traces brings up a vertical timing bar with the position time showing in the flag.

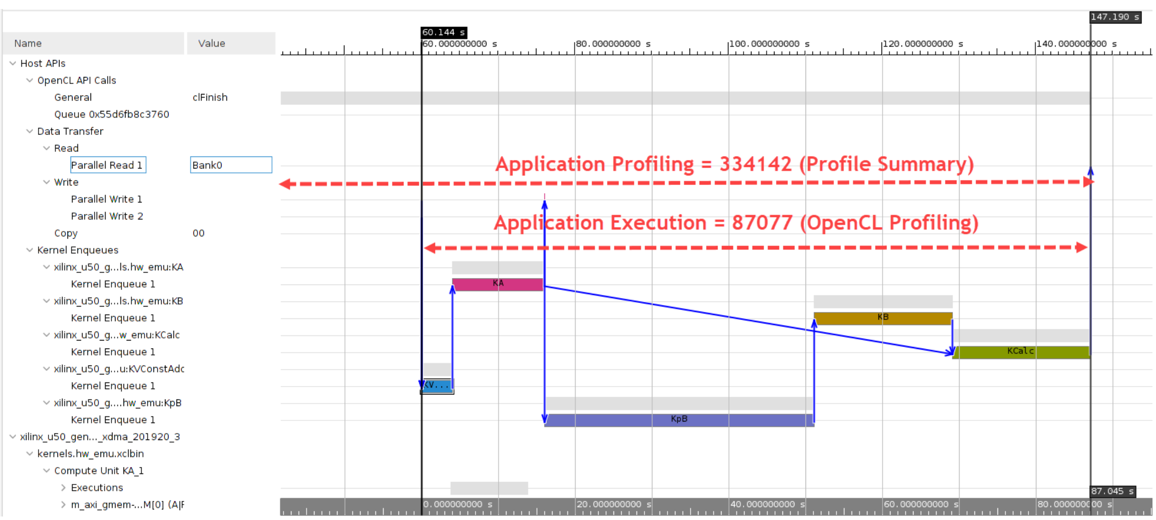


Figure 11-13: Timeline Trace Report (opt_1_baselining Project) - Profile Summary (Application)

Note: Your values may be different due to different PC configurations (processor speed, for example).

There are two important values:

- **Application Execution time (ms) = 87077** (from the terminal): This is the time measured between the start of data transfer from the host to global memory (before kernel execution) and the end of results transfer from the global memory to host memory. In the timeline trace view, this number is slightly bigger. This could be due to a non-precise manual selection of the start and end points on the timeline trace. Another reason is that the host code uses different start/end points comparing to the timeline trace.
- **Kernel Execution time (ms) = 87045** (from the terminal): This is the time measured between the start of the first kernel and the end of the last kernel. This information is very useful to observe when kernels are run in parallel.

2-7-7. Fill in the kernel and application execution times in the table below for the *opt_1_baselining* project.

Question 1

Fill in the tables below.

Project	**Profiling (ms) (From Custom OpenCL Profiling Values from Application Output in the Terminal)**	
	Kernel Execution Time (ms)	Application Execution Time (ms)
opt_1_baselining		
opt_3_single_api		
opt_4_kernel_parallel		

Kernel and Application Execution Times

Project	**Kernel Performance (ms) (From Profile Summary Report > Kernels & Compute Units)**				
	KVConstAdd	KpB	KA	KB	KCalc
opt_4_kernel_parallel					

Kernel Execution Time

Data movement is one of the critical aspects of the application. It is strongly suggested the `clEnqueueMigrateMemObjects` API be used to transfer data between host and global memory.

2-8. Analyze the Profile Summary report.

2-8-1. Click **Profile Summary** to view the Profile Summary report.

There are several sub-reports within the Profile Summary;

2-8-2. Expand **APIs Calls** to gain access to the OpenCL API Calls report.

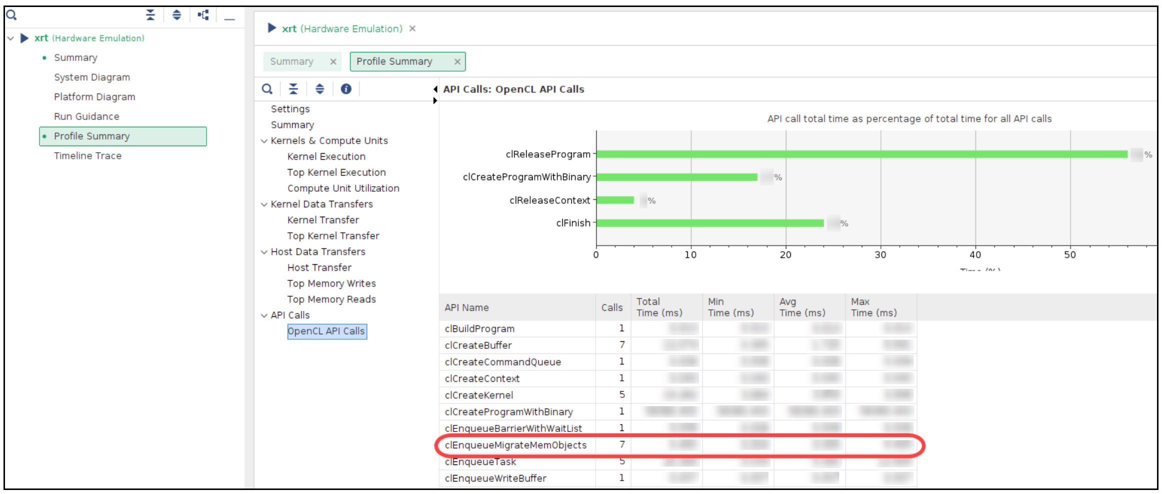


Figure 11-14: Profile Summary Report - OpenCL APIs (opt_1_baselining Project)

Notice that the `clEnqueueMigrateMemObjects` API is called several times (Calls column) for data transfer; however, the application uses a non-recommended `clEnqueueWriteBuffer` API to transfer data from host to global memory.

You will modify the host code in the next step to use `clEnqueueMigrateMemObjects` instead of `clEnqueueWriteBuffer`.

2-8-3. Select **File** > **Exit** to close the Vitis analyzer.

Using the clEnqueueMigrateMemObjects API Step 3

Your first attempt at improving the performance will be modifying the global memory buffer allocation. This change will enable you to use the `clEnqueueMigrateMemObjects` API for data transfer.

3-1. Review and update the host code in the *opt_2_clenqueuem_api* project.

3-1-1. Enter the following command to change the path to the source directory:

```
[host]$ cd $TRAINING_PATH/accel_optimization/lab/src/
opt_2_clenqueuem_api
```

3-1-2. Enter the following command to review the `host_2.cpp` file.

```
[host]$ gedit host_2.cpp
```

3-1-3. Search for the **`clEnqueueWriteBuffer`** API.

You will find the following command:

```
errCode = clEnqueueWriteBuffer(Command_Queue,
GlobMem_BUF_DataIn_3, 0, 0, SIZE_DataIn_3 * sizeof(int), DataIn_3,
0, NULL, &Mem_op_event[2]);
```

This API transfers the `GlobMem_BUF_DataIn_3` buffer to global memory.

3-1-4. Navigate to the following part of the code as shown below:

```
//#define USE_MIGRATEMEMOBJECTS_API
...
// ------------------------------------------------------------------
// Step 4.2: Create Buffers in Global Memory to store data
...
// Allocate Global Memory for GlobMem_BUF_DataIn_3
// .....................................................
#ifndef USE_MIGRATEMEMOBJECTS_API
GlobMem_BUF_DataIn_3 = clCreateBuffer(Context, CL_MEM_READ_ONLY, SIZE_DataIn_3 *
sizeof(int), NULL, &errCode);
#else
GlobMem_BUF_DataIn_3 = clCreateBuffer(Context, CL_MEM_READ_ONLY | CL_MEM_USE_HOST_PTR,
SIZE_DataIn_3 * sizeof(int), DataIn_3, &errCode);

#endif
...
```

Figure 11-15: Reviewing the host_2.cpp File

Allocate `GlobMem_BUF_DataIn_3` using a `CL_MEM_USE_HOST_PTR` flag so that you can use `clEnqueueMigrateMemObjects` instead of `clEnqueueWriteBuffer`.

This is given in the *else* part of the macro `USE_MIGRATEMEMOBJECTS_API`.

3-1-5. Navigate to the following part of the code as shown below, where you will replace `clEnqueueWriteBuffer` with `clEnqueueMigrateMemObjects`.

```
...
// -----------------------------------------------------
// Step 5.2: Copy Input data from Host to Global Memory
// -----------------------------------------------------
...
#ifndef USE_MIGRATEMEMOBJECTS_API
        errCode = clEnqueueWriteBuffer(Command_Queue, GlobMem_BUF_DataIn_3, 0, 0,
SIZE_DataIn_3 * sizeof(int), DataIn_3, 0, NULL, &Mem_op_event[2]);
#else
        errCode = clEnqueueMigrateMemObjects(Command_Queue, 1, &GlobMem_BUF_DataIn_3, 0, 0,
NULL, &Mem_op_event[2]);
#endif
...
```

Figure 11-16: Replacing clEnqueueWriteBuffer with clEnqueueMigrateMemObjects

3-1-6. Uncomment **#define USE_MIGRATEMEMOBJECTS_API** near the beginning of the host code to make the above changes.

Note: Do not uncomment *USE_MEMALIGN* macro.

3-1-7. Press <**Ctrl** + **S**> to save the file.

3-1-8. Press <**Ctrl** + **Q**> to close the editor.

3-2. Compile for software emulation.

3-2-1. Change the directory to where the compiled files will be generated:

`[host]$ cd $TRAINING_PATH/accel_optimization/lab/makefiles/opt_2_clenqueuem_api`

3-2-2. Enter the following command to compile the design in software emulation:

`[host]$ make build [TARGET=sw_emu]`

Because by default `TARGET` is already set to `sw_emu`, the optional `TARGET=sw_emu` is redundant.

Note: Ignore any warnings.

A new directory `build` has been created and there you will find the target build configuration (`sw_emu`) directory, which contains all the build files.

The `host.exe` file represents the host executable and is required. You will also often find an `xclbin` file that contains the kernels.

The `emconfigutil` utility generates the `emconfig.json` file, which contains information about the target device. This file is used for the emulation flow.

The `make build` calls the host compilation, kernel compilation, and `emconfig` utilities.

```
# Build the design without running host application
build: $(BUILD_DIR)/$(HOST_EXE) $(BUILD_DIR)/$(XCLBIN) $(BUILD_DIR)/$(EMCONFIG_FILE)
```

Figure 11-17: Building the Host Executable, XCLBIN, and emconfig.json File

3-3. Run the application in software emulation.

3-3-1. Enter the following command to run the application in software emulation mode:

`[host]$ make run`

When the application successfully completes, you will see the following warning messages in the Console tab:

```
HOST-Info: ============================================================
HOST-Info: (Step 5) Run Application
HOST-Info: ============================================================
HOST-Info: Setting Kernel arguments ...
XRT build version: [illegible]
Build hash: 7c93966ead2dec777b92bdc379893f22b5bd561e
Build date: [illegible]
Git branch: [illegible]
PID: 2872
UID: 1000
[[illegible]]
HOST: xilinx
EXE: /home/xilinx/training/optimization/lab/build/opt_2_clenqueuem_api/sw_emu/host.exe
[XRT] WARNING: unaligned host pointer '0x558c0a72f250' detected, this leads to extra memcpy
HOST_Info: Copy Input data to Global Memory ...
HOST-Info: Submitting Kernel K_KVConstAdd ...
```

Figure 11-18: Application Output - Unaligned Host Pointer (opt_2_clenquem_api Project)

3-4. Align the pointer size to 4096 bytes.

3-4-1. Enter the following command to open and edit the host code:

`[host]$ gedit ../../src/opt_2_clenqueuem_api/host_2.cpp`

3-4-2. Navigate to the following part of the code in `host_2.cpp` as shown below:

```
...
// ----------------------------------------------------------------
// Step 4.1: Generate data for DataIn_1 array
//           Generate data for DataIn_2 array
//           Generate data for DataIn_3 array
//           Allocate Memory to store the results: RES array
// ----------------------------------------------------------------
...
       cout << "HOST-Info: Generating data for DataIn_3 ... ";
#ifndef USE_MEMALIGN
       DataIn_3 = new int[SIZE_DataIn_3];
#else
       if (posix_memalign(&ptr,4096,SIZE_DataIn_3*sizeof(int))) {
              cout << endl << "HOST-Error: Out of Memory during memory allocation for
DataIn_2 array" << endl << endl;
              return EXIT_FAILURE;
       }
       DataIn_3 = reinterpret_cast<int*>(ptr);
#endif
       gen_int_values(DataIn_3,SIZE_DataIn_3, Values_Period);
       cout << "Generated " << SIZE_DataIn_3 << " values" << endl;
...
```

Figure 11-19: Reviewing the host_2.cpp File - memalign

The original part of the code **#ifndef USE_MEMALIGN** uses a new function to allocate host memory for **DataIn_3**. Now you will create a buffer using the **CL_MEM_USE_HOST_PTR** flag.

This means that the OpenCL API will use memory referenced by **DataIn_3** for the memory object. Therefore, in order to have efficient data transfer it should be aligned to 4096 bytes.

Therefore, you should use the **posix_memalign** command to allocate memory aligned to 4096 bytes, which is given in the *else* part of the **#ifndef USE_MEMALIGN** macro.

3-4-3. Uncomment **#define USE_MEMALIGN** near the beginning of the host code.

3-4-4. Press <**Ctrl** + **S**> to save the file.

3-4-5. Press <**Ctrl** + **Q**> to close the editor.

3-5. Build the project for software emulation to verify the correctness of the design.

3-5-1. Enter the following command to compile the design in software emulation:

```
[host]$ make build
```

Note: Ignore any warnings.

Important: If the build does not report that the `emconfig.json` file was created, then delete the **`$TRAINING_PATH/accel_optimization/lab/build/opt_2_clenqueuem_api`** directory and rerun this instruction.

3-6. Run the application in software emulation.

3-6-1. Enter the following command to run the application in software emulation:

```
[host]$ make run
```

The application has been successfully completed and you should see the messages in the Console tab. Notice that there are no warnings on unaligned memory.

3-7. Analyze the Profile Summary report using the Vitis analyzer.

3-7-1. Enter the following to see the reports generated for the software emulation:

```
[host]$ make view_run_summary
```

3-7-2. Expand the **xrt (Software Emulation)** entry in the left pane of the Vitis analyzer to see the various reports.

All references to the reports are relative to the expanded xrt (Software Emulation).

3-7-3. Click **Profile Summary** to view the Profile Summary report.

3-7-4. Click **APIs Calls**.

API Name	Calls	Total Time (ms)	Min Time (ms)	Avg Time (ms)	Max Time (ms)
clBuildProgram	1	[illegible]	[illegible]	[illegible]	[illegible]
clCreateBuffer	7	[illegible]	[illegible]	[illegible]	[illegible]
clCreateCommandQueue	1	[illegible]	[illegible]	[illegible]	[illegible]
clCreateContext	1	[illegible]	[illegible]	[illegible]	[illegible]
clCreateKernel	5	[illegible]	[illegible]	[illegible]	[illegible]
clCreateProgramWithBinary	1	[illegible]	[illegible]	[illegible]	[illegible]
clEnqueueBarrierWithWaitList	1	[illegible]	[illegible]	[illegible]	[illegible]
clEnqueueMigrateMemObjects	8	[illegible]	[illegible]	[illegible]	[illegible]
clEnqueueTask	5	[illegible]	[illegible]	[illegible]	[illegible]
clFinish	1	[illegible]	[illegible]	[illegible]	[illegible]

Figure 11-20: Profile Summary Report - OpenCL APIs (opt_2_clenqueuem_api Project)

Notice that eight `clEnqueueMigrateMemObjects` API calls to transfer data have been used. However, a single API can be used to transfer a set of buffers.

3-7-5. Select **File** > **Exit** to close the Vitis analyzer.

Using a Single API to Migrate Multiple Memory Buffers Step 4

In this next step, you will modify the host code to minimize the number of `clEnqueueMigrateMemObjects` API calls.

4-1. Review and update the host code in the *opt_3_single_api* project.

4-1-1. Enter the following command to change the path to the source directory:

```
[host]$ cd $TRAINING_PATH/accel_optimization/lab/src/
opt_3_single_api
```

4-1-2. Enter the following command to review the `host_3.cpp` file.

```
[host]$ gedit host_3.cpp
```

4-1-3. Navigate to the following part of the code as shown below:

```
//#define USE_SINGLE_API
...
// -------------------------------------------------------
// Step 5.2: Copy Input data from Host to Global Memory
// -------------------------------------------------------
...

#ifndef USE_SINGLE_API
        errCode = clEnqueueMigrateMemObjects(Command_Queue, 1, &GlobMem_BUF_DataIn_1, 0, 0,
NULL, &Mem_op_event[0]);
...
        errCode = clEnqueueMigrateMemObjects(Command_Queue, 1, &GlobMem_BUF_DataIn_2, 0, 0,
NULL, &Mem_op_event[1]);
...
        errCode = clEnqueueMigrateMemObjects(Command_Queue, 1, &GlobMem_BUF_DataIn_3, 0, 0,
NULL, &Mem_op_event[2]);
...
// ---------------------------------------------------------

        errCode = clEnqueueMigrateMemObjects(Command_Queue, 1, &GlobMem_BUF_KpB,
CL_MIGRATE_MEM_OBJECT_CONTENT_UNDEFINED, 0, NULL, &Mem_op_event[3]);
...
        errCode = clEnqueueMigrateMemObjects(Command_Queue, 1, &GlobMem_BUF_KA,
CL_MIGRATE_MEM_OBJECT_CONTENT_UNDEFINED, 0, NULL, &Mem_op_event[4]);
...
        errCode = clEnqueueMigrateMemObjects(Command_Queue, 1, &GlobMem_BUF_KB,
CL_MIGRATE_MEM_OBJECT_CONTENT_UNDEFINED, 0, NULL, &Mem_op_event[5]);
...
        errCode = clEnqueueMigrateMemObjects(Command_Queue, 1, &GlobMem_BUF_RES,
CL_MIGRATE_MEM_OBJECT_CONTENT_UNDEFINED, 0, NULL, &Mem_op_event[6]);
...
#else
        cl_mem Mem_Pointers_1[3], Mem_Pointers_2[4];

        Mem_Pointers_1[0] = GlobMem_BUF_DataIn_1;
        Mem_Pointers_1[1] = GlobMem_BUF_DataIn_2;
        Mem_Pointers_1[2] = GlobMem_BUF_DataIn_3;

        errCode = clEnqueueMigrateMemObjects(Command_Queue, 3, Mem_Pointers_1, 0, 0, NULL,
&Mem_op_event[0]);
...
        Mem_Pointers_2[0] = GlobMem_BUF_KpB;
        Mem_Pointers_2[1] = GlobMem_BUF_KA;
        Mem_Pointers_2[2] = GlobMem_BUF_KB;
        Mem_Pointers_2[3] = GlobMem_BUF_RES;

        errCode = clEnqueueMigrateMemObjects(Command_Queue, 4, Mem_Pointers_2,
CL_MIGRATE_MEM_OBJECT_CONTENT_UNDEFINED, 0, NULL, &Mem_op_event[1]);
...
#endif
```

Figure 11-21: Reviewing the host_3.cpp File - Single API to Transfer Multiple Data

The original part of the code **#ifndef USE_SINGLE_API** contains two groups of `clEnqueueMigrateMemObjects` APIs. The first three APIs (first group) in the **#ifndef USE_SINGLE_API** part of the code transfer memory objects with data (DataIn_1, DataIn_2, DataIn_3), while the next three API (second group) make a transfer with a **CL_MIGRATE_MEM_OBJECT_CONTENT_UNDEFINED** flag. This means that the buffers are allocated in the global memory but no real data transfer is done.

Therefore, each group of APIs can be replaced by a single API call by specifying the list of buffers to transfer, which is given in the *#else* part of the **#ifndef USE_SINGLE_API** macro.

Since the number of APIs is now reduced, the number of events used to synchronize also needs to be reduced. The modified part of the code is given in the *#else* part of the **#ifndef USE_SINGLE_API** macro shown below.

```
//#define USE_SINGLE_API
...
// ============================================================================
// Step 5: Set Kernel Arguments and Run the Application
//         o) Set Kernel Arguments...
...
#ifndef USE_SINGLE_API
        int Nb_Of_Mem_Events = 8,
        Nb_Of_Exe_Events = 5;
#else
        int Nb_Of_Mem_Events = 3,
        Nb_Of_Exe_Events = 5;
#endif
...

// ---------------------------------------------------------
// Step 5.4: Submit Copy Results from Global Memory to Host
// ---------------------------------------------------------
...
#ifndef USE_SINGLE_API
        errCode = clEnqueueMigrateMemObjects(Command_Queue, 1, &GlobMem_BUF_RES,
CL_MIGRATE_MEM_OBJECT_HOST, 1, &K_exe_event[4], &Mem_op_event[7]);
#else
        errCode = clEnqueueMigrateMemObjects(Command_Queue, 1, &GlobMem_BUF_RES,
CL_MIGRATE_MEM_OBJECT_HOST, 1, &K_exe_event[4], &Mem_op_event[2]);
#endif
...
```

Figure 11-22: Reviewing the host_3.cpp File - Reducing the Number of Events

4-1-4. Uncomment **#define USE_SINGLE_API** near the beginning of the host code to make the above changes.

4-1-5. Press <**Ctrl** + **S**> to save the file.

4-1-6. Press <**Ctrl** + **Q**> to close the editor.

4-2. Compile for software emulation.

4-2-1. Change the directory to where the compiled files will be generated:

```
[host]$ cd $TRAINING_PATH/accel_optimization/lab/makefiles/
opt_3_single_api
```

4-2-2. Enter the following command to compile the design in software emulation:

```
[host]$ make build [TARGET=sw_emu]
```

Because by default `TARGET` is already set to `sw_emu`, the optional `TARGET=sw_emu` is redundant.

Note: Ignore any warnings.

A new directory `build` has been created and there you will find the target build configuration (`sw_emu`) directory, which contains all the build files.

The `host.exe` file represents the host executable and is required. You will also often find an `xclbin` file that contains the kernels.

The `emconfigutil` utility generates the `emconfig.json` file, which contains information about the target device. This file is used for the emulation flow.

The `make build` calls the host compilation, kernel compilation, and `emconfig` utilities.

```
# Build the design without running host application
build: $(BUILD_DIR)/$(HOST_EXE) $(BUILD_DIR)/$(XCLBIN) $(BUILD_DIR)/$(EMCONFIG_FILE)
```

Figure 11-23: Building the Host Executable, XCLBIN, and emconfig.json File

Note: Since there is no change to the kernel code, the *opt_1_baselining* project build location is being pointed to using the variable name `$(XCLBIN_DIR)`.

4-3. Run the application in software emulation.

4-3-1. Enter the following command to run the application in software emulation:

```
[host]$ make run
```

The application completes, and the output messages appear in the terminal.

4-4. Analyze the Profile Summary report using the Vitis analyzer.

4-4-1. Enter the following to see the reports generated for the software emulation:

```
[host]$ make view_run_summary
```

4-4-2. Expand the **xrt (Software Emulation)** entry in the left pane of the Vitis analyzer to see the various reports.

All references to the reports are relative to the expanded xrt (Software Emulation).

4-4-3. Click **Profile Summary** to view the Profile Summary report.

4-4-4. Click **API Calls**.

API Name	Calls	Total Time (ms)	Min Time (ms)	Avg Time (ms)	Max Time (ms)
clBuildProgram	1	[illegible]	[illegible]	[illegible]	[illegible]
clCreateBuffer	7	[illegible]	[illegible]	[illegible]	[illegible]
clCreateCommandQueue	1	[illegible]	[illegible]	[illegible]	[illegible]
clCreateContext	1	[illegible]	[illegible]	[illegible]	[illegible]
clCreateKernel	5	[illegible]	[illegible]	[illegible]	[illegible]
clCreateProgramWithBinary	1	[illegible]	[illegible]	[illegible]	[illegible]
clEnqueueBarrierWithWaitList	1	[illegible]	[illegible]	[illegible]	[illegible]
clEnqueueMigrateMemObjects	3	[illegible]	[illegible]	[illegible]	[illegible]
clEnqueueTask	5	[illegible]	[illegible]	[illegible]	[illegible]
clFinish	1	[illegible]	[illegible]	[illegible]	[illegible]

Figure 11-24: Profile Summary Report - OpenCL APIs (opt_3_single_api Project)

Notice that three `clEnqueueMigrateMemObjects` API calls to transfer data have been used.

4-4-5. Select **File** > **Exit** to close the Vitis analyzer.

4-5. Compile for hardware emulation.

Note: If you have already built the hardware emulation for the *opt_1_baselining* project, use the [Own project] task below; otherwise, follow the [Prebuilt project] task.

4-5-1. Enter the following command to compile the design in hardware emulation:

[Own project]: `[host]$ make build TARGET=hw_emu`

[Prebuilt project]: `[host]$ make build TARGET=hw_emu PREBUILT=YES`

Note: This should only take a few seconds to complete as there is no need to build the kernel and the already built kernels in the *opt_1_baselining* project are being pointed to.

4-6. Run the application in hardware emulation.

4-6-1. Enter the following command to run the application in hardware emulation:

[Own project]: **`[host]$ make run TARGET=hw_emu`**

[Prebuilt project]: **`[host]$ make run TARGET=hw_emu PREBUILT=YES`**

Note: This may take 5-6 minutes to complete (using OpenCL profiling APIs to collect the customizable information).

If you receive the "Permission denied" message when you run the prebuilt project, change the path to the `support` directory and change the permission settings by using the `chmod` command as shown below:

```
[host]$ cd $TRAINING_PATH/accel_optimization/support/
prebuilt_opt_3
```

```
[host]$ chmod 777 host.exe
```

Change the path back to the `makefiles` directory and rerun the application for the prebuilt project:

```
[host]$ cd $TRAINING_PATH/accel_optimization/lab/makefiles/
opt_3_single_api
```

You should see that the application runs successfully.

The profiling information is displayed in the terminal as shown below. Note that the timing values may be different for you due to different PC configurations (processor speed, for example).

```
HOST-Info: ============================================================
HOST-Info: (Step 7) Custom Profiling
HOST-Info: ============================================================
HOST-Info: ------------------------------------------------------------------------------------
HOST-Info: Name           | type           | start         | end           | Duration(ms)
HOST-Info: ------------------------------------------------------------------------------------
HOST-Info: K_KVConstAdd   | kernel         |   54491975880 |   56499990026 |      2008.01
HOST-Info: K_KpB          | kernel         |   66507934138 |   98537878666 |      32029.9
HOST-Info: K_KA           | kernel         |   56500483034 |   66506950219 |      10006.5
HOST-Info: K_KB           | kernel         |   98538828342 |  113552837777 |        15014
HOST-Info: K_KCalc        | kernel         |  113553291227 |  129563045261 |      16009.8
HOST-Info: Transfer_1     | mem (H<->G)    |   54450603357 |   54491950360 |       41.347
HOST-Info: Transfer_2     | mem (H<->G)    |   54450597764 |   54450597764 |            0
HOST-Info: Transfer_3     | mem (H<->G)    |  129564471123 |  129565125103 |      0.65398
HOST-Info: ------------------------------------------------------------------------------------
HOST-Info:     Kernels Execution Time (ms) :  75071.1  (K_KCalc'end - K_KVConstAdd'begin)
HOST-Info: Application Execution Time (ms) :  75114.5  (Transfer_3'end - Transfer_2'begin)
HOST-Info: ------------------------------------------------------------------------------------
```

Figure 11-25: Application Output - OpenCL Profiling Results (opt_3_single_api Project)

4-6-2. Fill in the Kernel and Application Execution Times table for the *opt_3_single_api* project.

4-7. Analyze the Profile Summary report using the Vitis analyzer.

4-7-1. Enter the following to see the reports generated for the software emulation:

[Own project]: `[host]$ make view_run_summary TARGET=hw_emu`

[Prebuilt project]: `[host]$ make view_run_summary TARGET=hw_emu PREBUILT=YES`

4-7-2. Expand the **xrt (Software Emulation)** entry in the left pane of the Vitis analyzer to see the various reports.

All references to the reports are relative to the expanded xrt (Software Emulation).

4-7-3. Click **Timeline Trace** to view the Profile Summary report.

Figure 11-26: Application Timeline Trace - Kernels Running Sequentially

Let's look more closely at the application behavior itself. As you can see, the KVConstAdd, KA, KpB, and KB kernels are executed sequentially, while according to the algorithm, they can run in parallel.

4-7-4. Select **File** > **Exit** to close the Vitis analyzer after you complete your review.

4-8. Review the host code again to verify that out-of-order execution is enabled.

4-8-1. Enter the following command to review the `host_3.cpp` file.

`[host]$ gedit ../../src/opt_3_single_api/host_3.cpp`

4-8-2. Navigate to the following part of the code as shown below:

```
// -------------------------------------------------------------------------------------
// Step 2.4: Create Command Queue
// -------------------------------------------------------------------------------------
...
Command_Queue = clCreateCommandQueue(Context, Target_Device_ID,
CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE | CL_QUEUE_PROFILING_ENABLE, &errCode);
...
```

Figure 11-27: Reviewing the host_3.cpp File - Out-of-Order Execution Flag

As you can see, the command queue was created using out-of-order mode (set by the **CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE** flag). This means that the runtime should have a possibility to schedule (**KVConstAdd** + **KA**) and (**KpB** + **KB**) to run in parallel. The reason could be to synchronize the kernels properly to run in parallel.

Let's verify the events on how the kernels are synchronized.

4-8-3. Navigate to the following part of the code as shown below:

```
// ----------------------------------------
// Step 5.3: Submit Kernels for Execution
// ----------------------------------------
...
 errCode = clEnqueueTask(Command_Queue, K_KVConstAdd, 0, NULL, &K_exe_event[0]);

...

errCode = clEnqueueTask(Command_Queue, K_KA, 1, &K_exe_event[0], &K_exe_event[2]);

...

errCode = clEnqueueTask(Command_Queue, K_KpB, 1, &K_exe_event[2], &K_exe_event[1]);

...

errCode = clEnqueueTask(Command_Queue, K_KB, 1, &K_exe_event[1], &K_exe_event[3]);

...

errCode = clEnqueueTask(Command_Queue, K_KCalc, 2, &K_exe_event[2], &K_exe_event[4]);

...
```

Figure 11-28: Reviewing the host_3.cpp File - Verify the Events

As you can see, the kernel submission is synchronized using the K_eve_event array. If you draw the dependency graph between these events, then you will see the following figure:

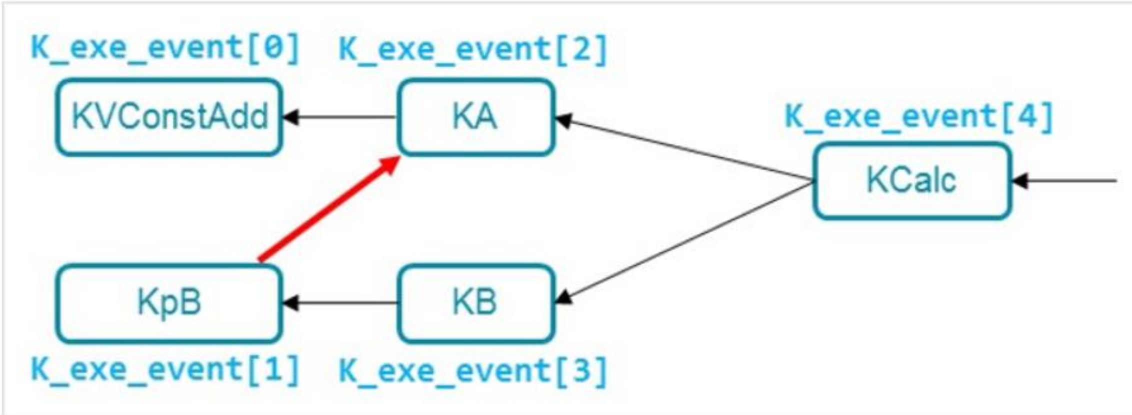


Figure 11-29: Dependency Graph

For example, this graph shows that **KCalc** can be launched if both **KA** and **KB** are completed – this is correct. However, the original host code has an extra dependency between launching **KpB** and completion **KA**. This prevents run time to launch (**KVConstAdd** + **KA**) and (**KpB** + **KB**) in parallel.

You will need to modify the host code in order to obtain the following dependency graph:

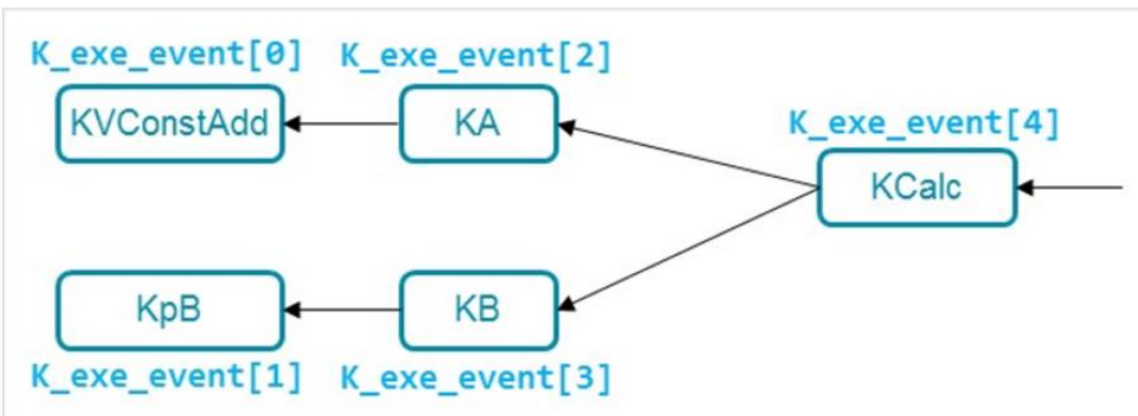


Figure 11-30: Dependency Graph - To Run in Parallel

4-8-4. Press <**Ctrl** + **Q**> to close the editor without saving any changes.

Running the Kernels in Parallel Step 5

In this step, you will modify the host code to run the kernels in parallel.

5-1. Review and update the host code in the *opt_4_kernel_parallel* project.

5-1-1. Enter the following command to change the path to the source directory:

[host]$ cd $TRAINING_PATH/accel_optimization/lab/src/opt_4_kernel_parallel

5-1-2. Enter the following command to review the `host_4.cpp` file.

[host]$ gedit host_4.cpp

5-1-3. Navigate to the following part of the code as shown below:

```
//define RUN_PARALLEL
...
// ----------------------------------------
// Step 5.3: Submit Kernels for Execution
// ----------------------------------------
...
#ifndef RUN_PARALLEL
       errCode = clEnqueueTask(Command_Queue, K_KpB, 1, &K_exe_event[2], &K_exe_event[1]);
#else
       errCode = clEnqueueTask(Command_Queue, K_KpB, 0, NULL, &K_exe_event[1]);
#endif
...
```

Figure 11-31: Reviewing the host_4.cpp File - Events

In order to remove the extra dependency, you will need to use the `clEnqueueTask` *#else* part of **#ifndef RUN_PARALLEL**.

5-1-4. Uncomment **#define RUN_PARALLEL** near the beginning of the host code.

5-1-5. Press <**Ctrl** + **S**> to save the file.

5-1-6. Press <**Ctrl** + **Q**> to close the editor.

5-2. Compile for software emulation.

5-2-1. Change the directory to where the compiled files will be generated:

[host]$ cd $TRAINING_PATH/accel_optimization/lab/makefiles/opt_4_kernel_parallel

5-2-2. Enter the following command to compile the design in software emulation:

[host]$ make build [TARGET=sw_emu]

Because by default `TARGET` is already set to `sw_emu`, the optional `TARGET=sw_emu` is redundant.

Note: Ignore any warnings.

A new directory `build` has been created and there you will find the target build configuration (`sw_emu`) directory, which contains all the build files.

The `host.exe` file represents the host executable and is required. You will also often find an `xclbin` file that contains the kernels.

The `emconfigutil` utility generates the `emconfig.json` file, which contains information about the target device. This file is used for the emulation flow.

The `make build` calls the host compilation, kernel compilation, and `emconfig` utilities.

```
# Build the design without running host application
build: $(BUILD_DIR)/$(HOST_EXE) $(BUILD_DIR)/$(XCLBIN) $(BUILD_DIR)/$(EMCONFIG_FILE)
```

Figure 11-32: Building the Host Executable, XCLBIN, and emconfig.json File

Note: Since there is no change to the kernel code, the *opt_1_baselining* project build location is being pointed to using the variable name `$(XCLBIN_DIR)`.

5-3. Run the application in software emulation.

5-3-1. Enter the following command to run the application in software emulation:

`[host]$ make run`

The application has been successfully completed and you should see the messages in the Terminal.

This process may take a few seconds to complete.

5-4. Compile for hardware emulation.

Note: If you have already built the hardware emulation for the *opt_1_baselining* project, use the [Own project] task below; otherwise, follow the [Prebuilt project] task.

5-4-1. Enter the following command to compile the design in hardware emulation:

[Own project]: **`[host]$ make build TARGET=hw_emu`**

[Prebuilt project]: **`[host]$ make build TARGET=hw_emu PREBUILT=YES`**

Note: This may take a few seconds to complete as there is no need to build the kernel as the already built kernels in the *opt_1_baselining* project are being pointed to.

5-5. Run the application in hardware emulation.

5-5-1. Enter the following command to run the application in hardware emulation:

[Own project]: **`[host]$ make run TARGET=hw_emu`**

[Prebuilt project]: **`[host]$ make run TARGET=hw_emu PREBUILT=YES`**

Note: This may take approximately 2 minutes to complete (using OpenCL profiling APIs to collect the customizable information).

If you receive the "Permission denied" message when you run the prebuilt project, change the path to the `support` directory and change the permission settings by using the `chmod` command as shown below:

```
[host]$ cd $TRAINING_PATH/accel_optimization/support/
prebuilt_opt_4
```

```
[host]$ chmod 777 host.exe
```

Change the path back to the `makefiles` directory and rerun the application for the prebuilt project:

```
[host]$ cd $TRAINING_PATH/accel_optimization/lab/makefiles/
opt_4_kernel_parallel
```

You should see that the application runs successfully.

The profiling information is displayed in the terminal as shown below. Note that the timing values may be different for you due to different PC configurations (processor speed, for example).

```
HOST-Info: ============================================================
HOST-Info: (Step 7) Custom Profiling
HOST-Info: ============================================================
HOST-Info: ------------------------------------------------------------------------------
HOST-Info: Name           | type          | start        | end          | Duration(ms)
HOST-Info: ------------------------------------------------------------------------------
HOST-Info: K_KVConstAdd   | kernel        |  90834090587 |  92834962031 |     2000.87
HOST-Info: K_KpB          | kernel        |  90834379844 | 126783881993 |     35949.5
HOST-Info: K_KA           | kernel        | 126783787817 | 138786556968 |     12002.8
HOST-Info: K_KB           | kernel        | 126784214278 | 145046403909 |     18262.2
HOST-Info: K_KCalc        | kernel        | 145046967485 | 161050138369 |     16003.2
HOST-Info: Transfer_1     | mem (H<->G)   |  90788510378 |  90834067927 |     45.5575
HOST-Info: Transfer_2     | mem (H<->G)   |  90788492284 |  90788492284 |           0
HOST-Info: Transfer_3     | mem (H<->G)   | 161050469761 | 161051027109 |    0.557348
HOST-Info: ------------------------------------------------------------------------------
HOST-Info:     Kernels Execution Time (ms) :  70216  (K_KCalc'end - K_KVConstAdd'begin)
HOST-Info: Application Execution Time (ms) :  70262.5  (Transfer_3'end - Transfer_2'begin)
HOST-Info: ------------------------------------------------------------------------------
```

Figure 11-33: Profiling Results (opt_4_kernel_parallel Project)

5-5-2. Fill in the Kernel and Application Execution Times table for the *opt_4_kernel_parallel* project.

5-6. Analyze the Profile Summary report using the Vitis analyzer.

5-6-1. Enter the following to see the reports generated for the hardware emulation:

[Own project]: `[host]$ make view_run_summary TARGET=hw_emu`

[Prebuilt project]: `[host]$ make view_run_summary TARGET=hw_emu PREBUILT=YES`

5-6-2. Expand the **xrt (Software Emulation)** entry in the left pane of the Vitis analyzer to see the various reports.

All references to the reports are relative to the expanded xrt (Software Emulation).

5-6-3. Click **Timeline Trace** to view the Timeline Trace report.

Note: Kernels KA and KB now run in parallel.

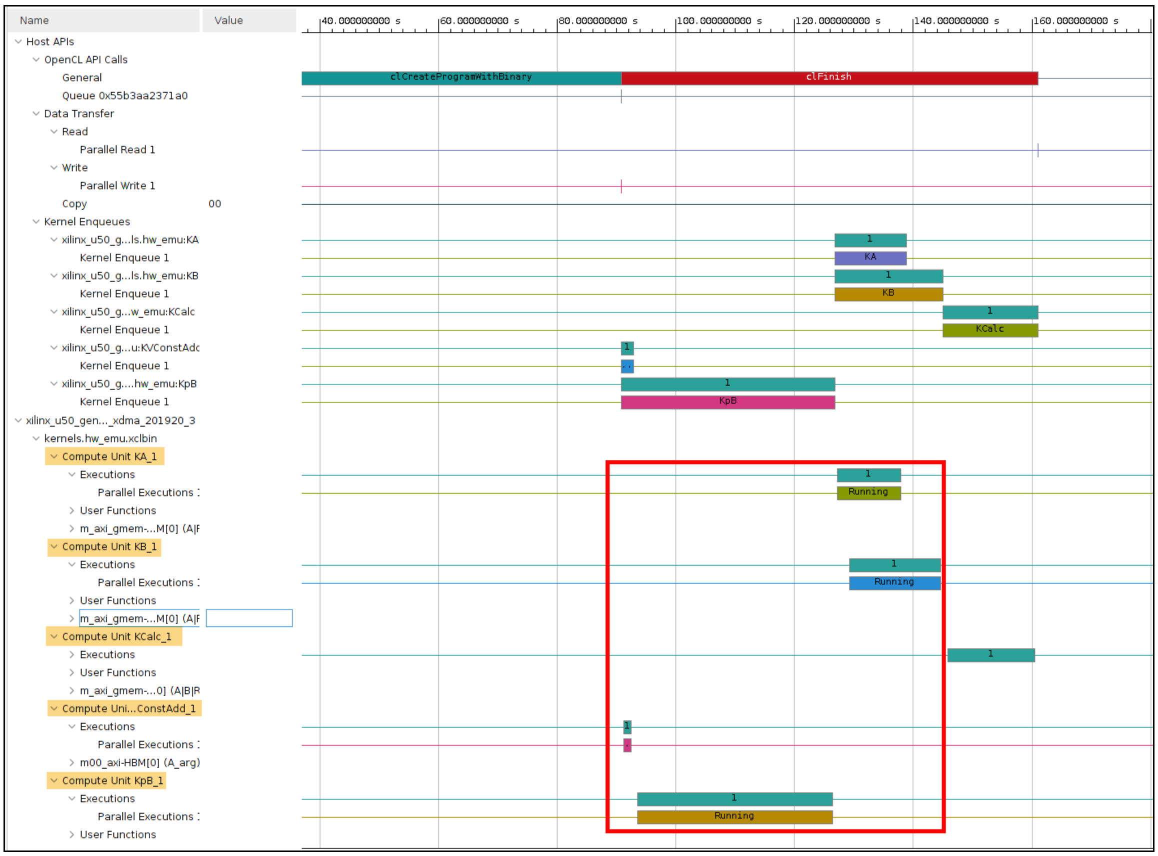


Figure 11-34: Application Timeline - Kernels Running Parallel

5-7. Analyze the Profile Summary report.

5-7-1. Click **Profile Summary** to view the Profile Summary report.

5-7-2. Select the **Kernels & Compute Units** section.

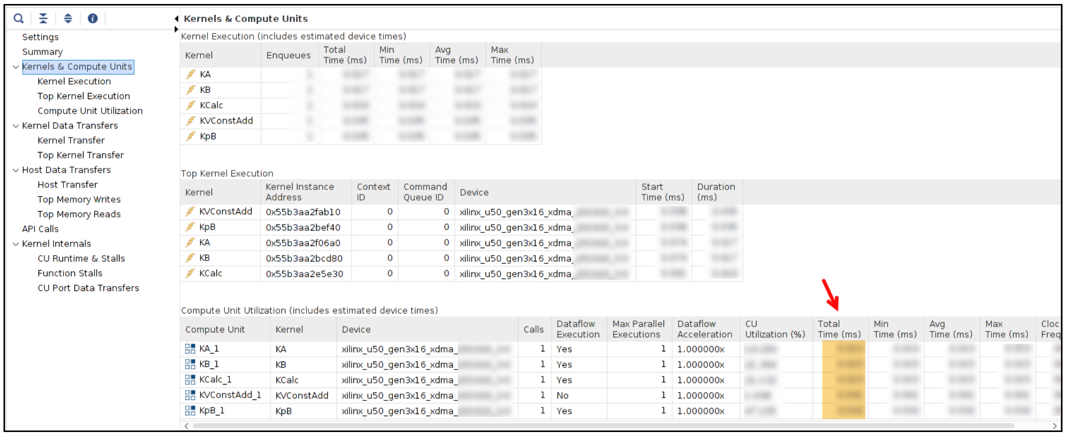


Figure 11-35: Profile Summary Report (Kernels & Compute Units)

5-7-3. Fill in the Kernel Execution Time table for the *opt_4_kernel_parallel* project.

5-8. Analyze the waveform.

5-8-1. Click **Waveform** to view the waveform.

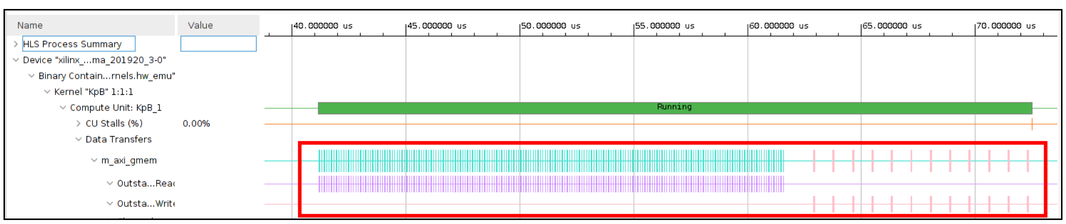


Figure 11-36: Reviewing the Waveform

KpB spends a lot of time transferring data from global memory. Zoom in to have a more detailed view on Read Data (so you can see a single read data transfer). Point your mouse to a single transfer to see its details.

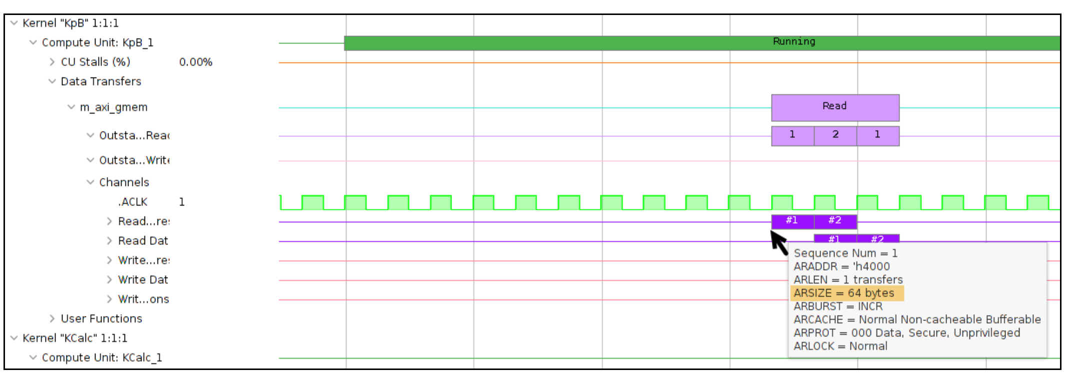


Figure 11-37: Reviewing the Read Data for More Information

The Vitis compiler does data burst transfers that are ARSIZE (64 bytes) long. Some scenarios may require that the code be written in a such a way as to enable the burst data transfer.

5-8-2. Select **File** > **Exit** to close the Vitis analyzer.

5-9. Analyze the Vitis HLS reports for the kernels KA and KB.

5-9-1. Enter the following to open the Vitis HLS tool report for the KA kernel:

`[host]$ gedit ../../../support/prebuilt_opt_4/reports/KA/ KA_csynth.rpt`

```
================================================================
== Vitis HLS Report for 'KA'
================================================================
* Date:           [illegible]

* Version:        [illegible] (Build [illegible])
* Project:        KA
* Solution:       solution (Vitis Kernel Flow Target)
* Product family: virtexuplus
* Target device:  xcu50-fsvh2104-2-e


================================================================
== Performance Estimates
================================================================
+ Timing:
    * Summary:
    +--------+---------+----------+------------+
    |  Clock |  Target | Estimated| Uncertainty|
    +--------+---------+----------+------------+
    |ap_clk  | [illegible] ns| [illegible] ns| [illegible] ns|
    +--------+---------+----------+------------+

+ Latency:
    * Summary:
    +---------+---------+----------+----------+------+------+---------+
    |  Latency (cycles) |  Latency (absolute) |   Interval  | Pipeline|
    |   min   |   max   |    min   |    max   |  min |  max |   Type  |
    +---------+---------+----------+----------+------+------+---------+
    |  [illegible]|  [illegible]| [illegible] us| [illegible] us| [illegible]| [illegible]|       no|
    +---------+---------+----------+----------+------+------+---------+

    + Detail:
        * Instance:
        +---------------------------------+----------------------+---------+---------+----------+----------+------+------+---------+
        |                                 |                      |  Latency (cycles) |  Latency (absolute) |   Interval  | Pipeline|
        |             Instance            |        Module        |   min   |   max   |    min   |    max   |  min |  max |   Type  |
        +---------------------------------+----------------------+---------+---------+----------+----------+------+------+---------+
        |grp_KA_Pipeline_loop_st_1_fu_76  |KA_Pipeline_loop_st_1 |  [illegible]|  [illegible]| [illegible] us| [illegible] us| [illegible]| [illegible]|       no|
        +---------------------------------+----------------------+---------+---------+----------+----------+------+------+---------+
```

Figure 11-38: HLS Report for the KA Kernel

Observe the Interval value under the Instance section.

5-9-2. Press <**Ctrl** + **Q**> to close the editor.

5-9-3. Enter the following command to review the v++ log report for the KA kernel:

`[host]$ gedit ../../../support/prebuilt_opt_4/reports/KA/v++.log`

Observe that the Initiation Interval (II) achieved is 2.

5-9-4. Press <**Ctrl** + **Q**> to close the editor.

5-9-5. Similarly, enter the following command to review the HLS report for the KB kernel:

`[host]$ gedit ../../../support/prebuilt_opt_4/reports/KB/KB_csynth.rpt`

```
================================================================
== Vitis HLS Report for 'KB'
================================================================
* Date:           [illegible]

* Version:        [illegible] (Build [illegible])
* Project:        KB
* Solution:       solution (Vitis Kernel Flow Target)
* Product family: virtexuplus
* Target device:  xcu50-fsvh2104-2-e


================================================================
== Performance Estimates
================================================================
+ Timing:
    * Summary:
    +--------+---------+----------+------------+
    |  Clock |  Target | Estimated| Uncertainty|
    +--------+---------+----------+------------+
    |ap_clk  | [illegible] ns| [illegible] ns| [illegible] ns|
    +--------+---------+----------+------------+

+ Latency:
    * Summary:
    +---------+---------+-----------+-----------+------+------+---------+
    |  Latency (cycles) |  Latency (absolute)   |   Interval  | Pipeline|
    |   min   |   max   |    min    |    max    |  min |  max |   Type  |
    +---------+---------+-----------+-----------+------+------+---------+
    | [illegible]| [illegible]| [illegible] us| [illegible] us| [illegible]| [illegible]|       no|
    +---------+---------+-----------+-----------+------+------+---------+

    + Detail:
        * Instance:
        +---------------------------------------+------------------------------+---------+---------+-----------+-----------+------+------+---------+
        |                                       |                              |  Latency (cycles) |  Latency (absolute)   |   Interval  | Pipeline|
        |                Instance               |            Module            |   min   |   max   |    min    |    max    |  min |  max |   Type  |
        +---------------------------------------+------------------------------+---------+---------+-----------+-----------+------+------+---------+
        |grp_KB_Pipeline_VITIS_LOOP_44_1_fu_95  |KB_Pipeline_VITIS_LOOP_44_1   | [illegible]| [illegible]| [illegible] us| [illegible] us| [illegible]| [illegible]|       no|
        |grp_KB_Pipeline_VITIS_LOOP_48_2_fu_103 |KB_Pipeline_VITIS_LOOP_48_2   | [illegible]| [illegible]| [illegible] us| [illegible] us| [illegible]| [illegible]|       no|
        +---------------------------------------+------------------------------+---------+---------+-----------+-----------+------+------+---------+
```

Figure 11-39: HLS Report for the KB Kernel

Observe the the Interval value under the Instance section.

5-9-6. Press <**Ctrl** + **Q**> to close the editor.

5-9-7. Enter the following command to review the v++ log report for the KB kernel:

`[host]$ gedit ../../../support/prebuilt_opt_4/reports/KB/v++.log`

Observe that the Initiation Interval (II) achieved is 1 for the loops VITIS_loop_44_1 and VITIS_LOOP_48_2.

In addition, notice that the kernels KA and KB need to access more than two data at a time from the memory. The Vitis HLS tool automatically applies the array partitioning and optimizes the performance of the kernels.

In some scenarios, you may have to apply the HLS pragma for the array partition as shown below:

```
#pragma HLS array_partition variable=<name> <type>  factor=<int> dim=<int>
```

- `variable=<name>`: A required argument that specifies the array variable to be partitioned.
- `<type>`: Optionally specifies the partition type. The default type is complete. The following types are supported:
  - `cyclic`: Cyclic partitioning creates smaller arrays by interleaving elements from the original array. The array is partitioned cyclically by putting one element into each new array before coming back to the first array to repeat the cycle until the array is fully partitioned block and complete.

  - `block`: Block partitioning creates smaller arrays from consecutive blocks of the original array. This effectively splits the array into N equal blocks, where N is the integer defined by the `factor=` argument.
  - `complete`: Complete partitioning decomposes the array into individual elements. For a one-dimensional array, this corresponds to resolving a memory into individual registers. This is the default `<type>`.
- `factor=<int>`: Specifies the number of smaller arrays that are to be created.

5-9-8. Press <**Ctrl** + **Q**> to close the editor.

5-9-9. Enter **`exit`** to close the terminal window.

VM users: Run the "Clean up the VirtualBox file system" instruction bellow to free the spaced used by this lab. Maximizing the available space helps other labs run. CloudShare uses can skip this step.

5-10. Clean up the VirtualBox file system.

5-10-1. Enter the following command to delete the contents of the workspace:

```
[host]$ rm -rf $TRAINING_PATH/accel_optimization
```

This will recursively delete all of the files in the `$TRAINING_PATH/accel_optimization` directory.

Summary

You just applied several optimization techniques, such as:

- Using the `clEnqueueMigrateMemObjects` API for data transfer
- Running kernels in parallel
- Optimizing a C++ kernel using burst data transfer and the dataflow attribute
- Optimizing C++ based kernels using burst data transfer
- Array partitioning

Understanding when and how to apply these techniques can help you design systems that meet your specifications.

Answers

1. Fill in the tables below.

   Note that the timing values may be different for you due to different PC configurations (processor speed, for example).

Project	**Profiling (ms) (From Custom OpenCL Profiling Values from Application Output)**	
	Kernel Execution Time (ms)	Application Execution Time (ms)
opt_1_baselining	87045	87077
opt_3_single_api	75071	75114
opt_4_kernel_parallel	70216	70262

Kernel and Application Execution Times

Project	**Kernel Performance (ms) (From Profile Summary Report > Kernels & Compute Units)**				
	KVConstAdd	KpB	KA	KB	KCalc
opt_1_baselining	-	-	-	-	-
opt_3_single_api	-	-	-	-	-
opt_4_kernel_parallel	0.001	0.032	0.010	0.015	0.015

Kernel Performance