University of Pennsylvania
Department of Electrical and System Engineering
System-on-a-Chip Architecture

ESE532, Fall 2020 | I/O and FPGA Milestone | Wednesday, November 11

Due: Friday, Nov. 20, 5:00PM

1. Move some part of your design onto the FPGA for acceleration.
   Writeup should identify what you moved onto the FPGA, how you validated it, and how you tuned it. Identify the current throughput achieved.

2. Use the supplied measurement routines (Refer to Tutorial 1) to report the input throughput to your encoder.

3. Use the supplied measurement routines to report the maximum real-time throughput the current design can sustain (Refer to Tutorial 2 and see how timers are used. Additionally, notice how you can use `getProfilingInfo` on `cl::Event` to get the kernel execution time).

4. Turn in a tar file with your FPGA accelerated code to the designated assignment component in canvas (one per group).

5. Turn in a tar or zip file with binaries to support execution of your code to the designated assignment component in canvas (one per group).

   (a) `encoder.xclbin` for FPGA kernel

   (b) `encoder` for OpenCL host code executable

   (c) `decoder` executable configured to work with your encoded file and that can be run on the Ultra96. (Most likely, this is just a compilation of the `Decoder.cpp` we supplied; however, if you chose a different maximum block size, you may need to change `CODE_LENGTH`; so give us back one with that change made.)
   Make sure to compile it with the `aarch64-linux-gnu-g++` compiler and test it on the Ultra96. While you could run the decoder on your host machine (which could be Linux/Mac OS/Windows), we will run your decoder on the Ultra96.

   - Your compression program (OpenCL host code) should take one argument:
     - the file name where the program should store the compressed data.

     Your program should assume that `encoder.xclbin` is in the same directory as the host executable.
   - Your compression program should start up ready to receive inputs.

We don't expect significant FPGA acceleration on this milestone, but we do want you to start exploring acceleration options.

Tutorials

1. **Measuring Ethernet Throughput in the Encoder**

   In P2, you measured the raw ethernet throughput using `iperf3` and got about 895 Mbits/s. Note that, by default `iperf3` sent TCP packets to the receiver in the Ultra96, whereas we are using UDP in the project, which is a faster protocol. We will now show you how to measure the input throughput in your encoder.

   (a) We have updated the server code we gave you in P2 with some instrumentation. Do the following to get the latest changes:

   ```
   cd ese532_code/
   git pull origin master
   ```

   (b) Compile the encoder code, copy the binary to the Ultra96 and run it.

   (c) Compile the client code and run the client with the supplied `vmlinuz.tar` file as follows:

   ```
   ./client -f vmlinuz.tar -i 10.10.7.1
   ```

   (d) You should see the following output in the Ultra96 terminal:

   ```
   root@ultra96v2-2020-1:~# ./encoder.elf
   setting up sever...
   server setup complete!
   write file with 69079040
   --------------- Key Throughputs ---------------
   Input Throughput to Encoder: 1079.33 Mb/s. (Latency: 0.512015s).
   root@ultra96v2-2020-1:~# diff vmlinuz.tar output_cpu.bin
   ```

   You should see the following output in the host terminal:

   ```
   filename is vmlinuz.tar
   ip is set to 10.10.7.1
   payload_size is 8192
   bytes_read 69079040
   ```

   (e) You can see that we are indeed getting about 1 Gb/s input throughput. You can look into `encoder.cpp` and see that we are using a timer to measure the total latency taken by the call: `server.get_packet(input[writer])` (ignoring the first call which waits for the first packet to arrive). Later in the code, we calculated the throughput as follows:

   ```
   float ethernet_latency = ethernet_timer.latency() / 1000.0;
   float input_throughput = (bytes_written * 8 / 1000000.0) / ethernet_latency;
   ```

```
std::cout << "Input Throughput to Encoder: "
          << input_throughput << " Mb/s."
          << " (Latency: " << ethernet_latency << "s)."
          << std::endl;
```

(f) Note that it is very important that you verify the output using `diff`. You can lose packets if your encoder cannot keep up with the input throughput, in which case you should use the `-s` option in the client to transfer at a lower speed.

(g) Note that we have increased the `PAYLOAD_SIZE` to 8192 bytes in the updated code.

2. **FPGA Acceleration Tutorial: Bloom Filter**

Using bloom filter as the application, this tutorial shows you:

- a significant speedup (8×) when computations are offloaded on the FPGA efficiently.
- how to write an HLS kernel for a CPU implementation (using `ap_uint`, `hls::stream`, and `pragmas`).
- how to achieve communication-compute overlap using sub-buffers.

(a) Clone the `ese532_code` repository using the following command:

```
git clone https://github.com/icgrp/ese532_code.git
```

If you already have it cloned, pull in the latest changes using:

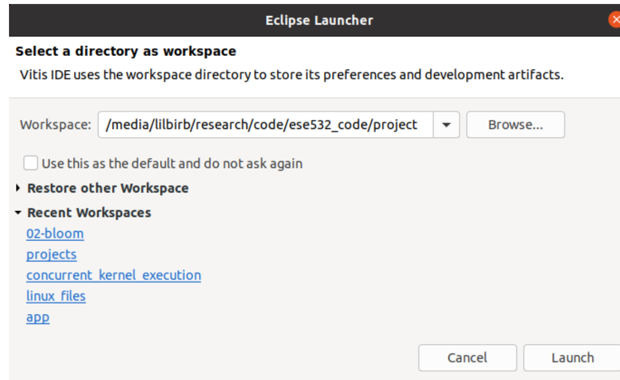
```
cd ese532_code/
git pull origin master
```

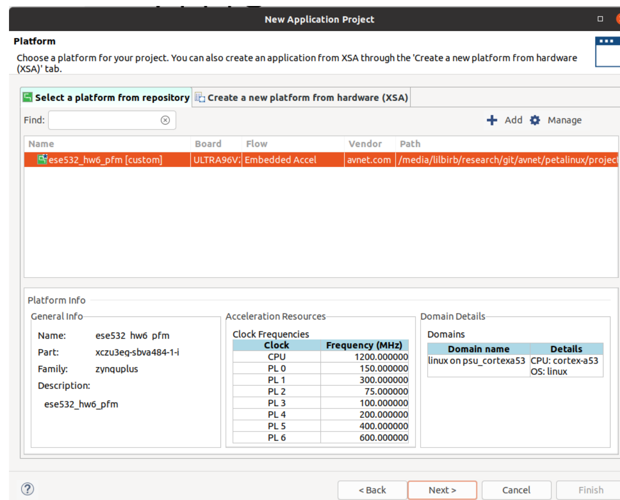
The code you will use for this section is in the `vitis_tutorials/bloom` directory. The directory structure looks like this:

```
bloom/
  cpu/
  fpga/
    MurmurHash2.c
    common.h
    compute_score_fpga_kernel.cpp
    compute_score_host.cpp
    hls_stream_utils.h
    main.cpp
    sizes.h
    xcl2.cpp
    xcl2.hpp
```

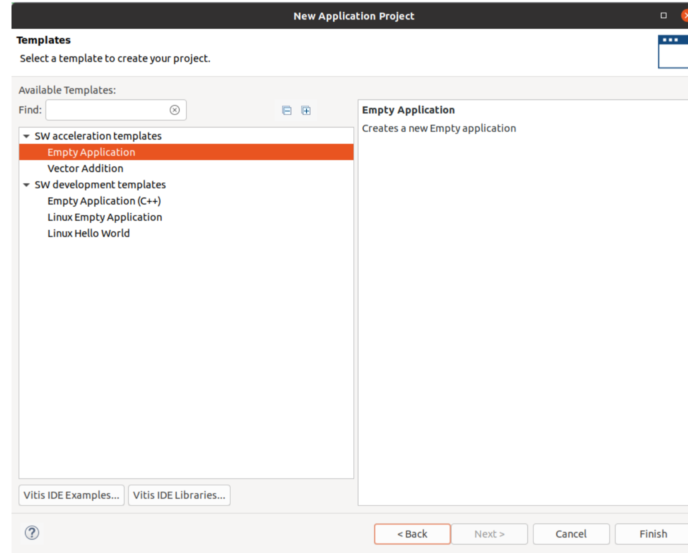
The `cpu` folder has a standalone CPU implementation of the bloom filter, which you can compile using the Vitis GUI flow from P2 and run it. The `main.cpp` code in the `fpga` folder has the OpenCL host code. The top level HLS function is in `compute_score_fpga_kernel.cpp`. We will now show how to use the Vitis GUI flow to compile OpenCL and HLS code (if you would like to use a `Makefile`, modify and use the one from Homework 6).

(b) Use the instructions from Milestone 2 and start `vitis`. Create or use an existing workspace, create a new application project and use the provided platform.

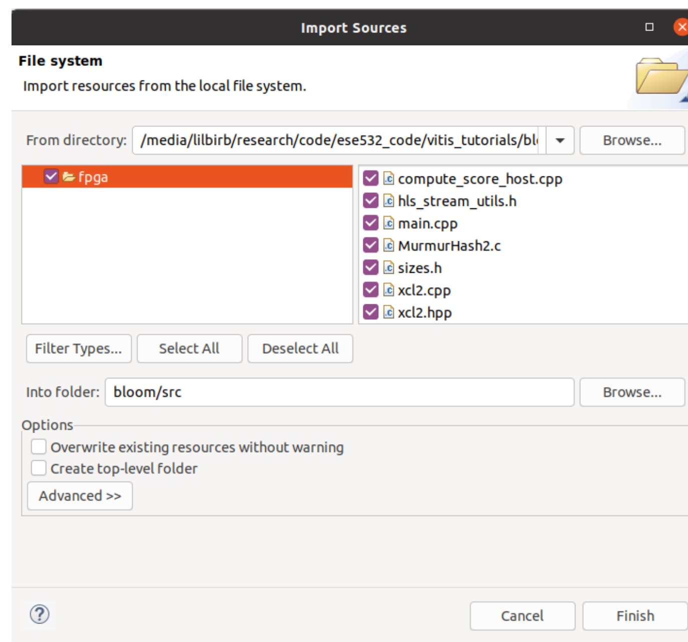




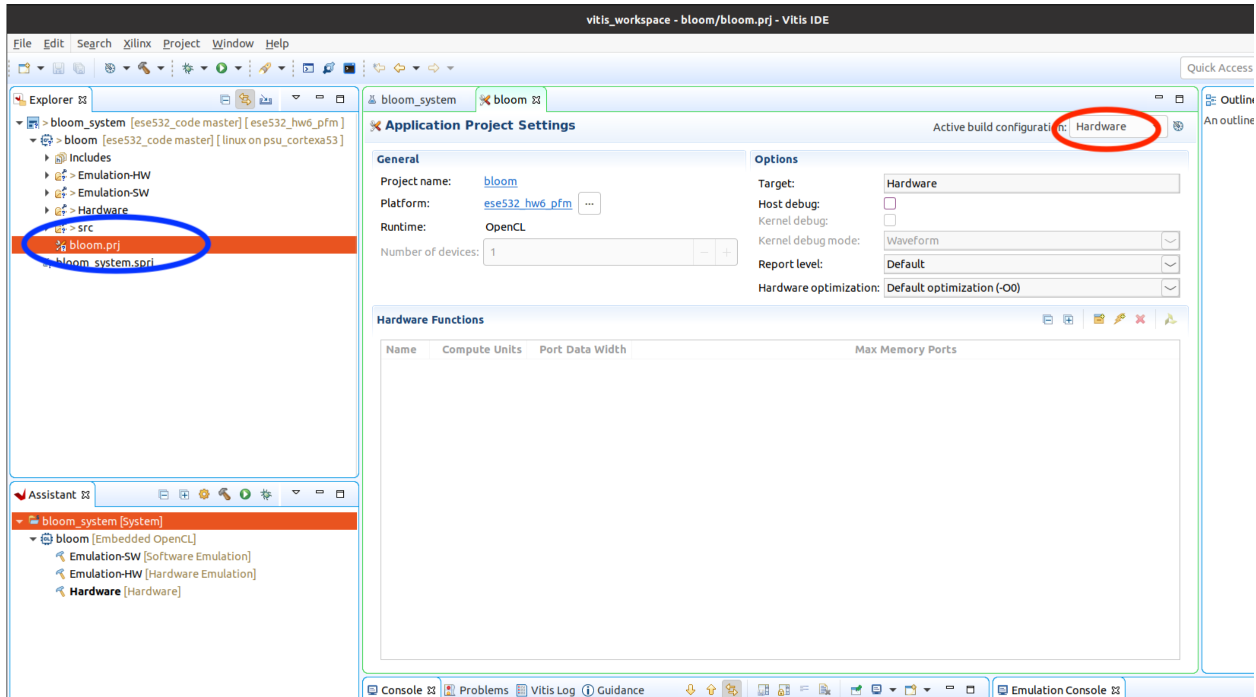
(c) Choose `Empty Application` from the `SW acceleration templates` as follows:



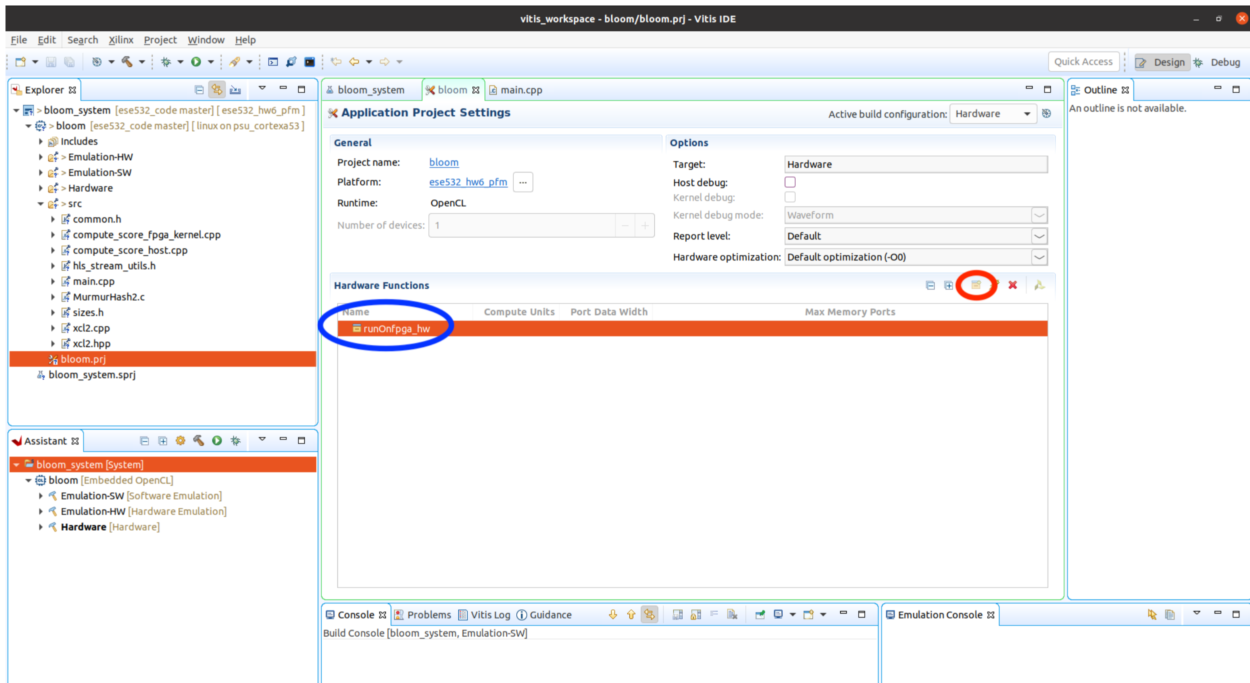
(d) Right-click on the `src` folder and click on import sources. Import the source files for this project as follows:



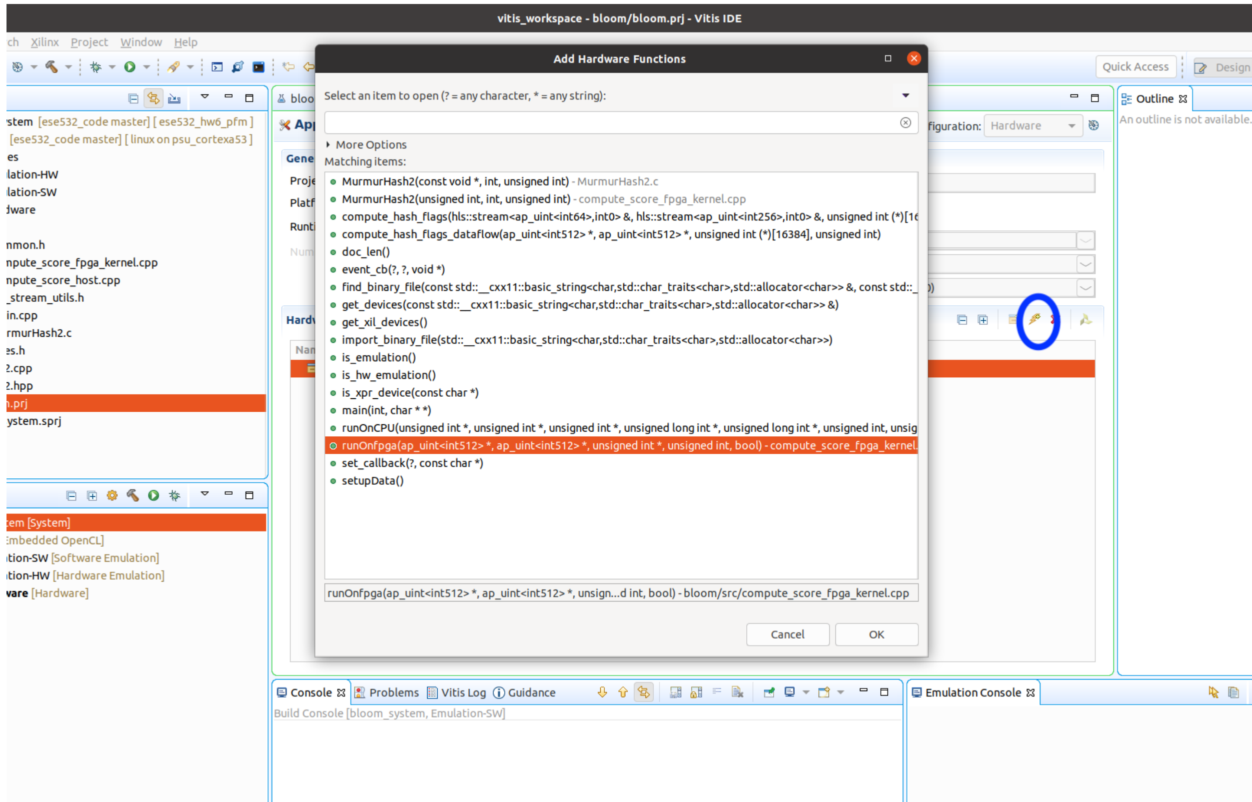
(e) Click on `bloom.prj` from the Explorer and change the `Active build configuration` to `Hardware` as shown below:



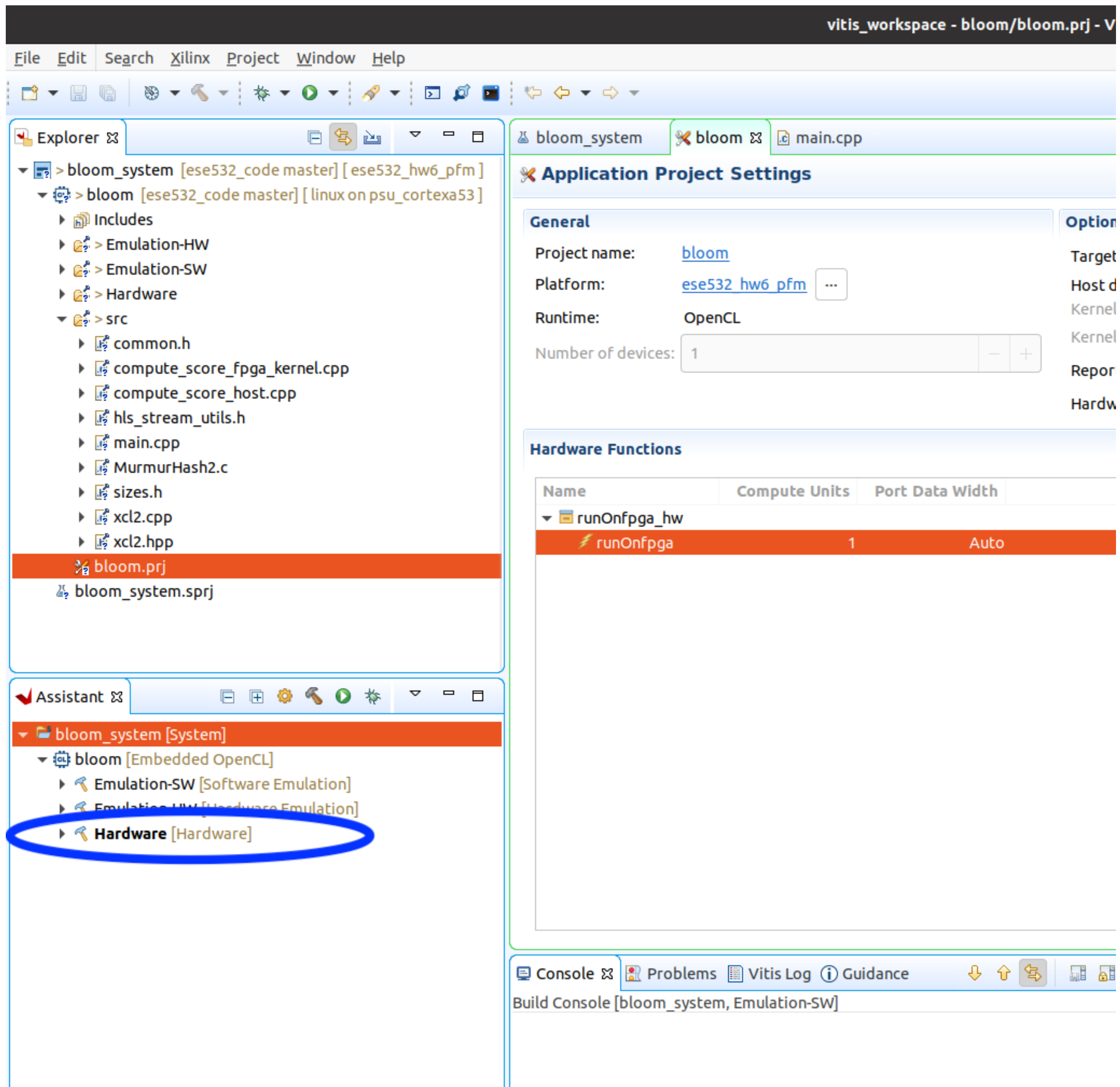
(f) Add an `xclbin` container by clicking on the button circled in red below. Rename the container to `runOnfpga_hw` by clicking on the name `binary_container_1`. This is the `xclbin` name that the host code uses.



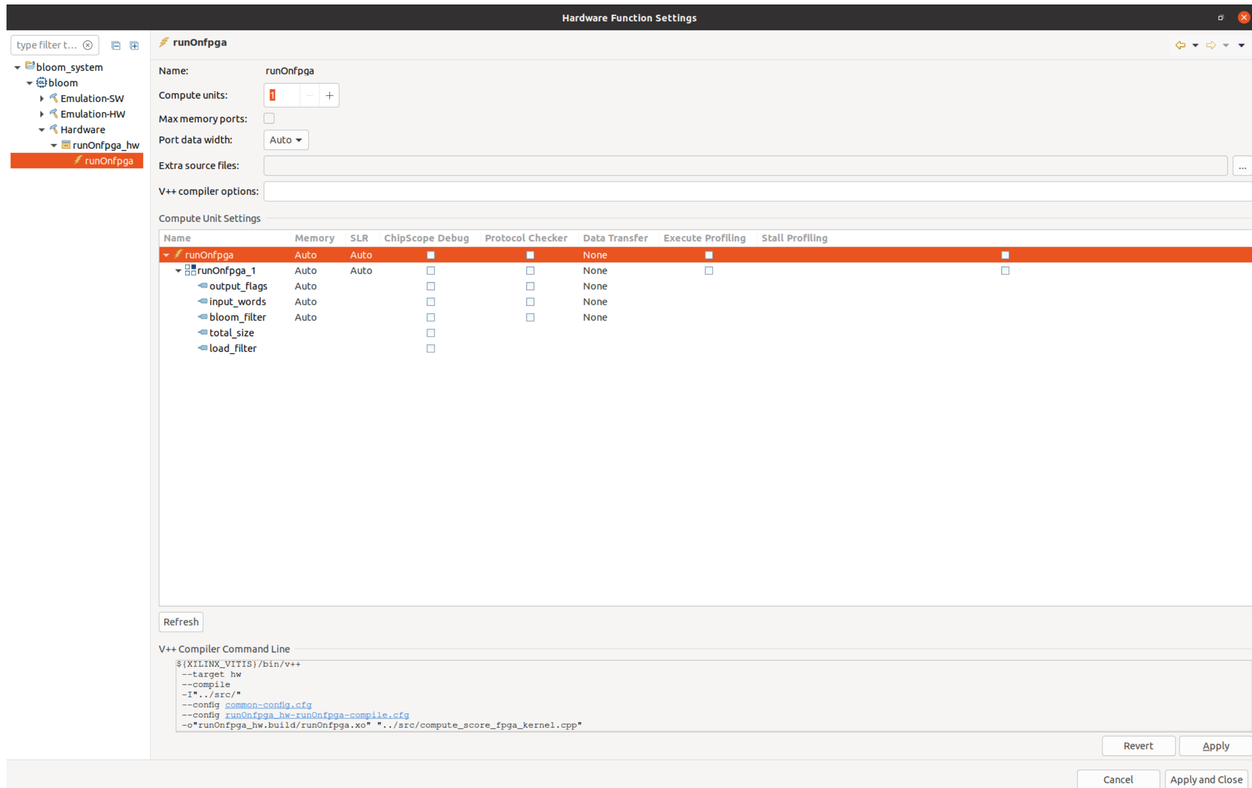
(g) Now click on the button circled in blue below. Choose the `runOnfpga` function as the hardware function.



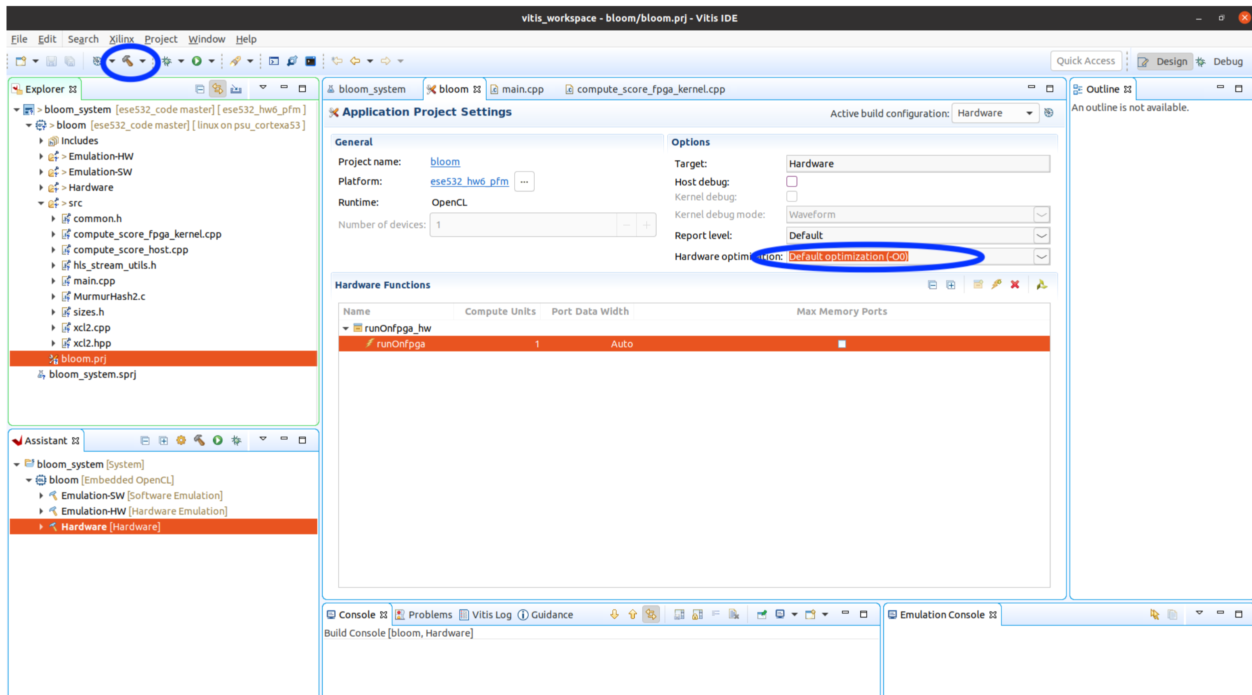
(h) From the `Assistant` view, double click on `Hardware` as follows:



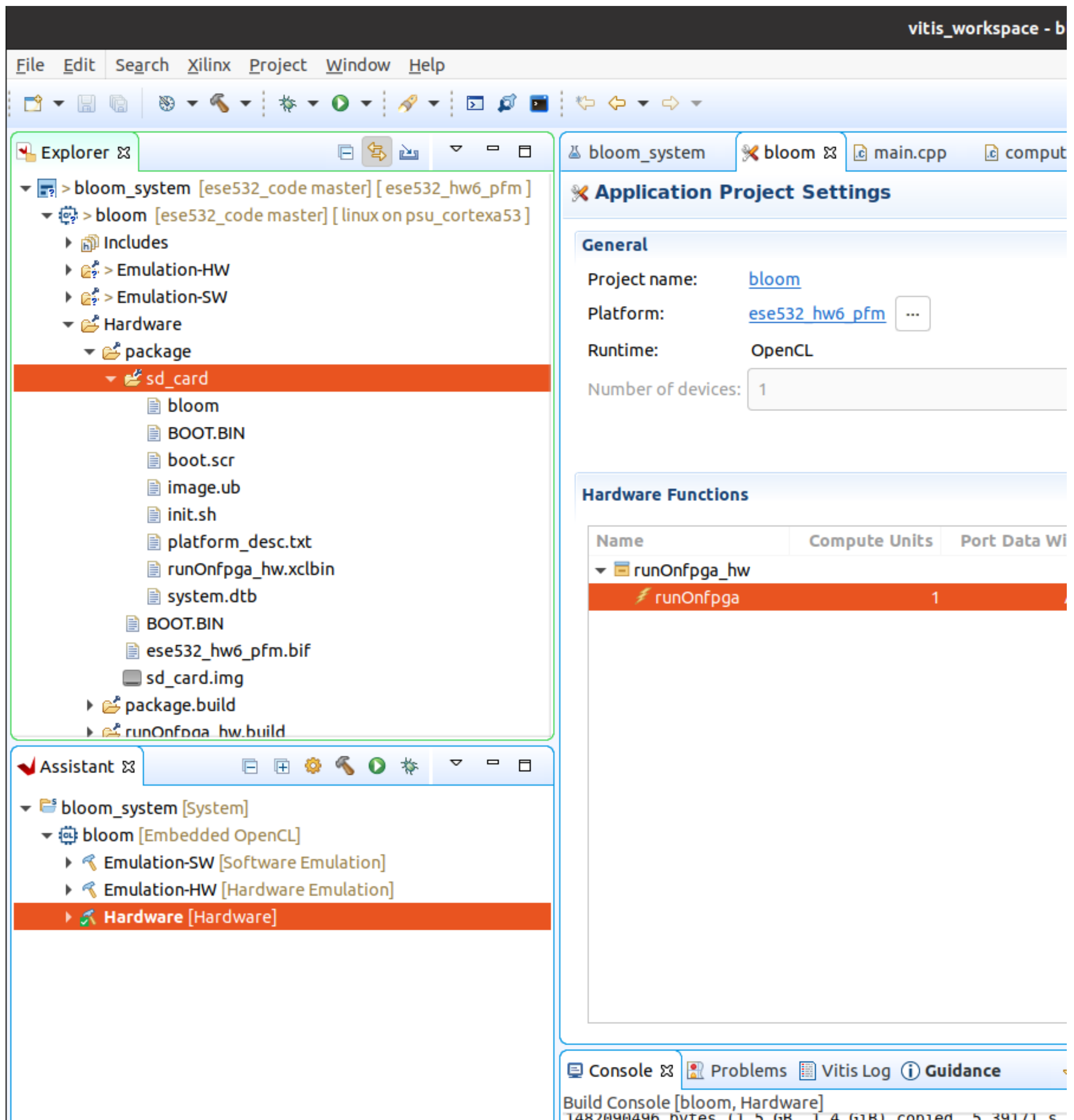
(i) Click on Hardware→runOnfpga_hw→runOnfpga. This brings the screen where you can specify the number of compute units, compiler options for the kernel, assign different ports to inputs etc. Keep the defaults for now:



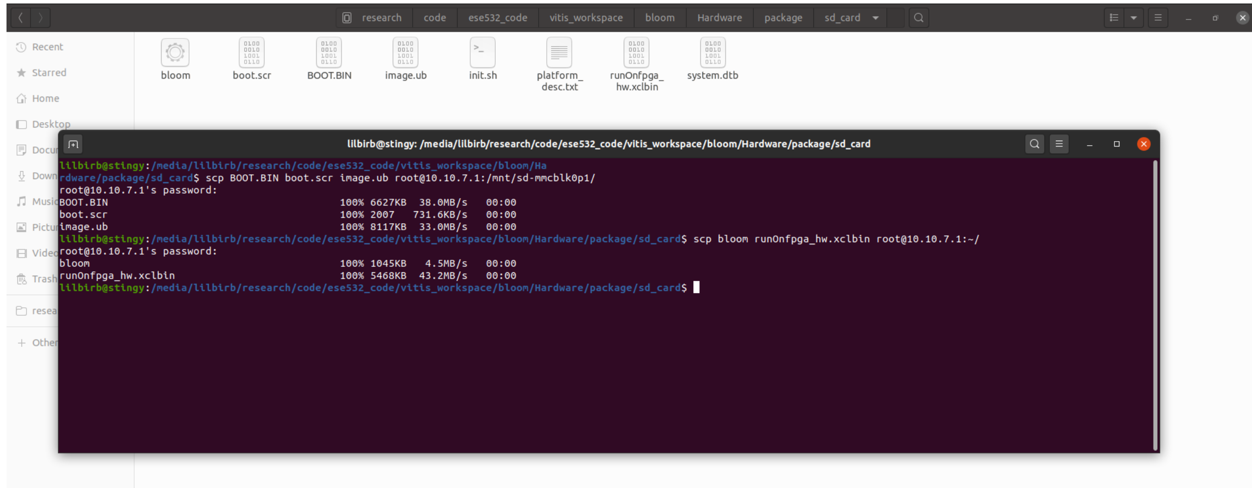
(j) Click on `bloom.prj`. Check out the `Hardware optimization` option where you can change the optimization level for the hardware function. Additionally, recall from P2 that you can change the optimization level of the host code from the C/C++ build settings. Now click on the build button on the menu bar to start compilation:



(k) Once the compilation completes, open the `Hardware` folder from the `Explorer`. The binaries are in the `package/sd_card` folder.



(l) Copy the binaries and the `xrt.ini` to the Ultra96 as follows and then reboot the Ultra96.



(m) Run the code using the following commands in the Ultra96:

```
ifconfig eth0 10.10.7.1 netmask 255.0.0.0
export XILINX_XRT=/usr
./bloom 40000 64
```

You should see the following output in the terminal:

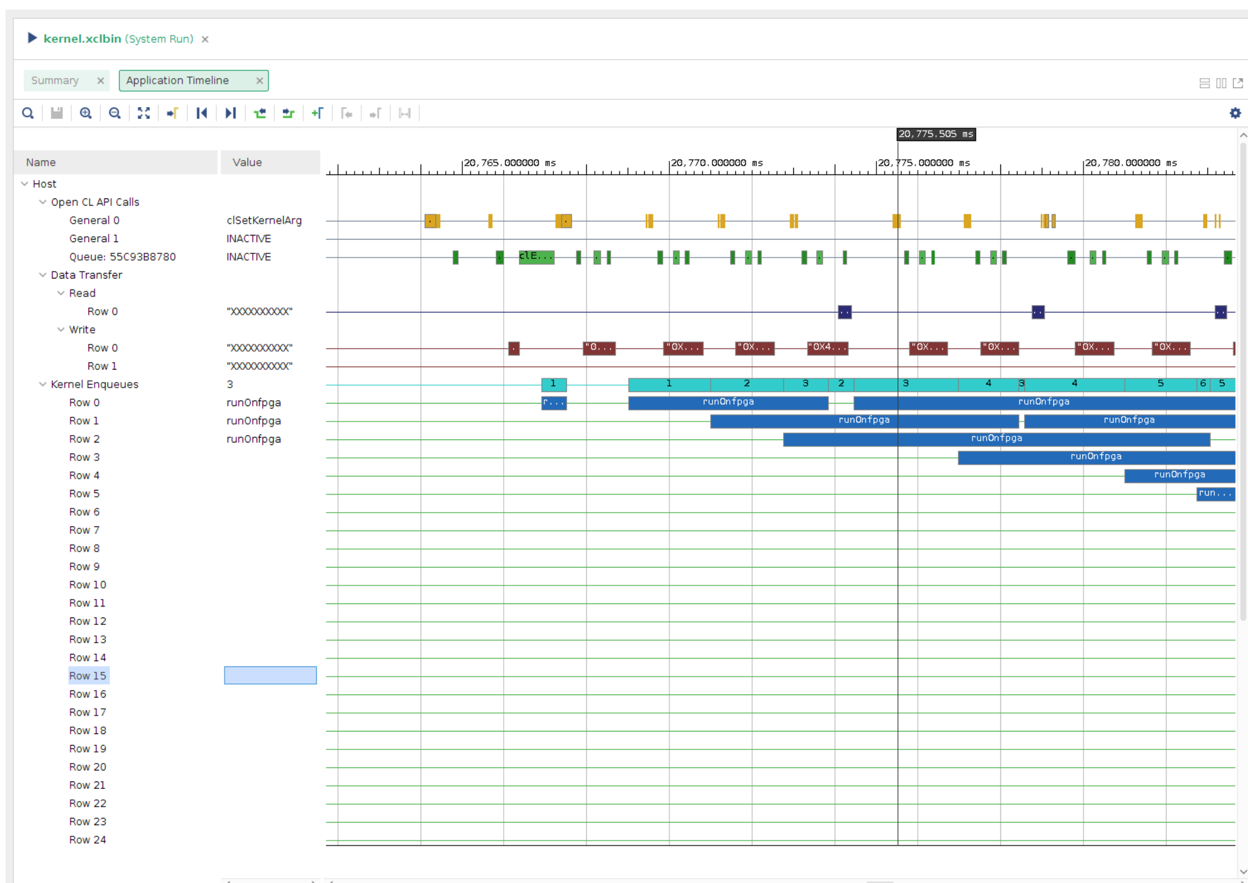
```
root@ultra96v2-2020-1:~# ./bloom 40000 64
Initializing data
Creating documents - total size : 559.858 MBytes (139964416 words)
Creating profile weights
[ 1018.547572] [drm] Pid 769 opened device
[ 1018.551450] [drm] Pid 769 closed device
[ 1018.558627] [drm] Pid 769 opened device
Loading runOnfpga_hw.xclbin
[ 1018.617733] [drm] zocl_xclbin_read_axlf The XCLBIN already loaded
[ 1018.617765] [drm] zocl_xclbin_read_axlf 3c650f2f-9cc2-408a-8c92-0ec3
[ 1018.633496] [drm] bitstream 3c650f2f-9cc2-408a-8c92-0ec3bc335ce3 loc
[ 1018.641197] [drm] Reconfiguration not supported
[ 1018.652995] [drm] bitstream 3c650f2f-9cc2-408a-8c92-0ec3bc335ce3 unl
 Processing 559.858 MBytes of data
 Splitting data in 64 sub-buffers of 8.748 MBytes for FPGA processing
--------------------------------------------------------------------
[ooqueu]: Completed buffer migrate
[ooqueu]: Completed buffer migrate
[ooqueu]: Completed buffer migrate
[ooqueu]: Completed buffer migrate
```

```
[ooqueu]: Completed buffer migrate
[ooqueu]: Completed buffer migrate
[ooqueu]: Completed buffer migrate
[ooqueu]: Completed buffer migrate
[ooqueu]: Completed buffer migrate
[ooqueu]: Completed buffer migrate
[ooqueu]: Completed buffer migrate
[ooqueu]: Completed buffer migrate
[ooqueu]: Completed buffer migrate
[ooqueu]: Completed buffer migrate
[ooqueu]: Completed buffer migrate
[ooqueu]: Completed buffer migrate
[ooqueu]: Completed buffer migrate
[ooqueu]: Completed buffer migrate
[ooqueu]: Completed buffer migrate
[ooqueu]: Completed buffer migrate
[ooqueu]: Completed buffer migrate
[ooqueu]: Completed buffer migrate
[ooqueu]: Completed buffer migrate
[ooqueu]: Completed buffer migrate
[ooqueu]: Completed buffer migrate
[ooqueu]: Completed buffer migrate
waiting...
[ooqueu]: Completed buffer migrate
[ooqueu]: Completed buffer migrate
[ooqueu]: Completed buffer migrate
[ooqueu]: Completed buffer migrate
[ooqueu]: Completed buffer migrate
[ooqueu]: Completed buffer migrate
[ooqueu]: Completed buffer migrate
[ooqueu]: Completed buffer migrate
[ooqueu]: Completed buffer migrate
waiting...
[ooqueu]: Completed buffer migrate
[ooqueu]: Completed buffer migrate
[ooqueu]: Completed buffer migrate
[ooqueu]: Completed buffer migrate
[ooqueu]: Completed buffer migrate
[ooqueu]: Completed buffer migrate
[ooqueu]: Completed buffer migrate
[ooqueu]: Completed buffer migrate
[ooqueu]: Completed buffer migrate
waiting...
.
.
```

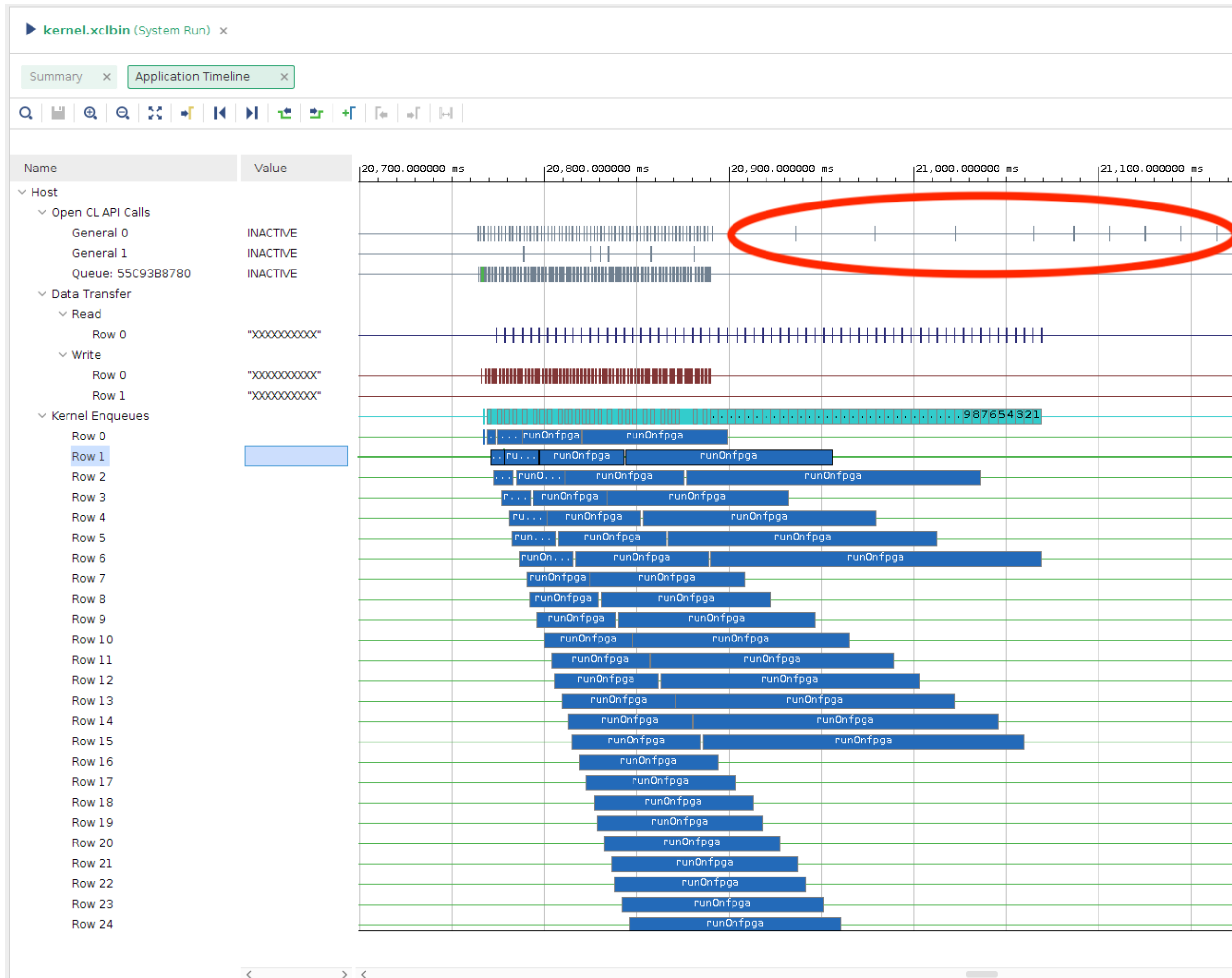
```
Executed FPGA accelerated version   |  1414.5707 ms   ( FPGA 247.878 ms
Executed Software-Only version      |  11779.4325 ms
--------------------------------------------------------------------
Verification: PASS
```

(n) The verbose outputs in the terminal is caused by the call `set_callback(flagDone, "oooqueue");` in the host code. This is a helper function in `xcl2.hpp` that prints out the state of an OpenCL event. You can use it to debug OpenCL calls. In addition, you can also use the `OCL_CHECK` macro from `xcl2.hpp` to see if an OpenCL call succeeded.

(o) Copy the generated run summary and csv files to your host computer and open vitis analyzer. You can see overlap of kernel execution with data transfer and OpenCL API calls.



(p) If you scroll forward in the timeline, you can see overlap between computation in the cpu and the fpga as shown below. From the output in 2m, these calls



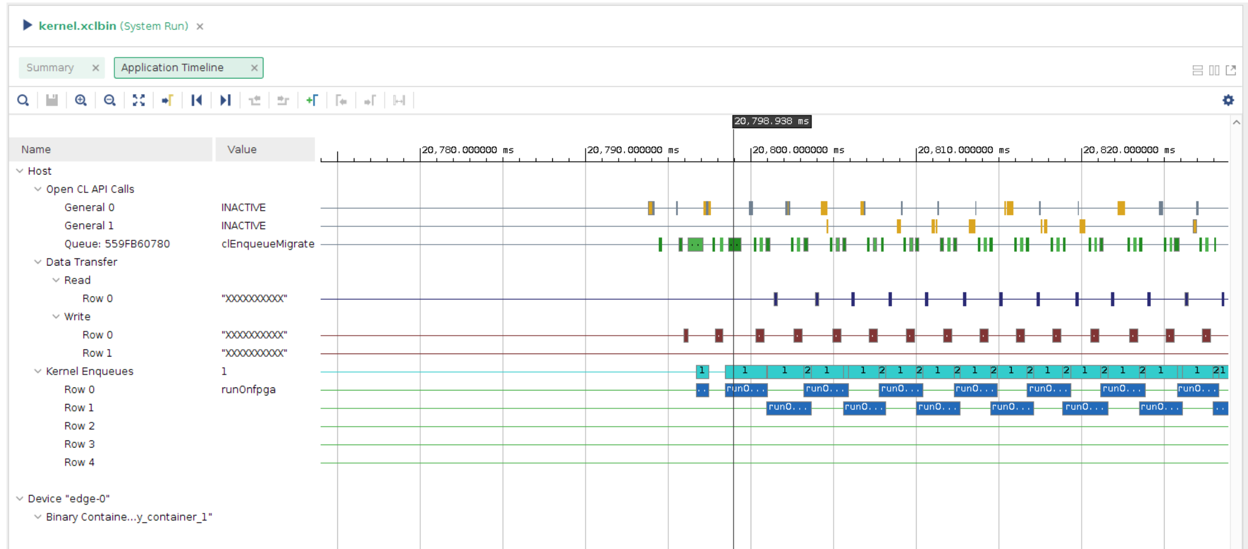
correspond to the `waiting...` print outs. You can check in the host code, how we wait for a `cl::Event` to finish based on a condition, and when the event notifies that it's finished, we start executing the cpu code, so that it overlaps with the fpga execution:

```
needed += size;
if (needed > available) {
  clWaitForEvents(1, (const cl_event *) &flagWait[iter]);

  std::cout << "waiting..." << std::endl;
  available += subbuf_doc_info[iter].size / sizeof(uint);
  iter++;
}
```

(q) Now run with a different ITER value and look at the updated trace:

```
./bloom 40000 128
```



You can see that since the kernel execution time gets smaller as you increase the iteration number, the next kernel execution starts almost immediately.

(r) Running a sweep on the number of iterations, we see that ITER=32 is the most performant for this design:

```
./bloom 40000 8
Executed FPGA accelerated version  |  1413.3252 ms   ( FPGA 305.796 ms )
Executed Software-Only version     |  11780.3703 ms
--------------------------------------------------------------------
Verification: PASS

./bloom 40000 16
Executed FPGA accelerated version  |  1396.5072 ms   ( FPGA 298.034 ms )
Executed Software-Only version     |  11770.1858 ms
--------------------------------------------------------------------
Verification: PASS

./bloom 40000 32
Executed FPGA accelerated version  |  1391.2062 ms   ( FPGA 284.336 ms )
Executed Software-Only version     |  11768.1306 ms
--------------------------------------------------------------------
Verification: PASS

./bloom 40000 64
Executed FPGA accelerated version  |  1414.5707 ms   ( FPGA 247.878 ms )
```

```
Executed Software-Only version     |  11779.4325 ms
--------------------------------------------------------------------
Verification: PASS

./bloom 40000 128
Executed FPGA accelerated version  |  1458.1155 ms   ( FPGA 179.195 ms )
Executed Software-Only version     |  11782.2403 ms
--------------------------------------------------------------------
Verification: PASS

./bloom 40000 256
Executed FPGA accelerated version  |  1533.5603 ms   ( FPGA 10.701 ms )
Executed Software-Only version     |  11798.4502 ms
--------------------------------------------------------------------
Verification: PASS
```

(s) This concludes a top-down walk-through of this tutorial. To learn more about this design, read the following in-order:

   i. Overview of the Original Application
   ii. Architect a Device-Accelerated Application
   iii. Implementing the Kernel
   iv. Data Movement Between the Host and Kernel

   Note that the tutorial is written for data center cards. Some of the parameter choices, such as port data width, DDR memory etc. should be reconsidered for the Ultra96 to get optimal performance (refer to this paper: Unexpected Diversity: Quantitative Memory Analysis for Zynq UltraScale+ Systems).

3. **Using Multiple Compute Units**

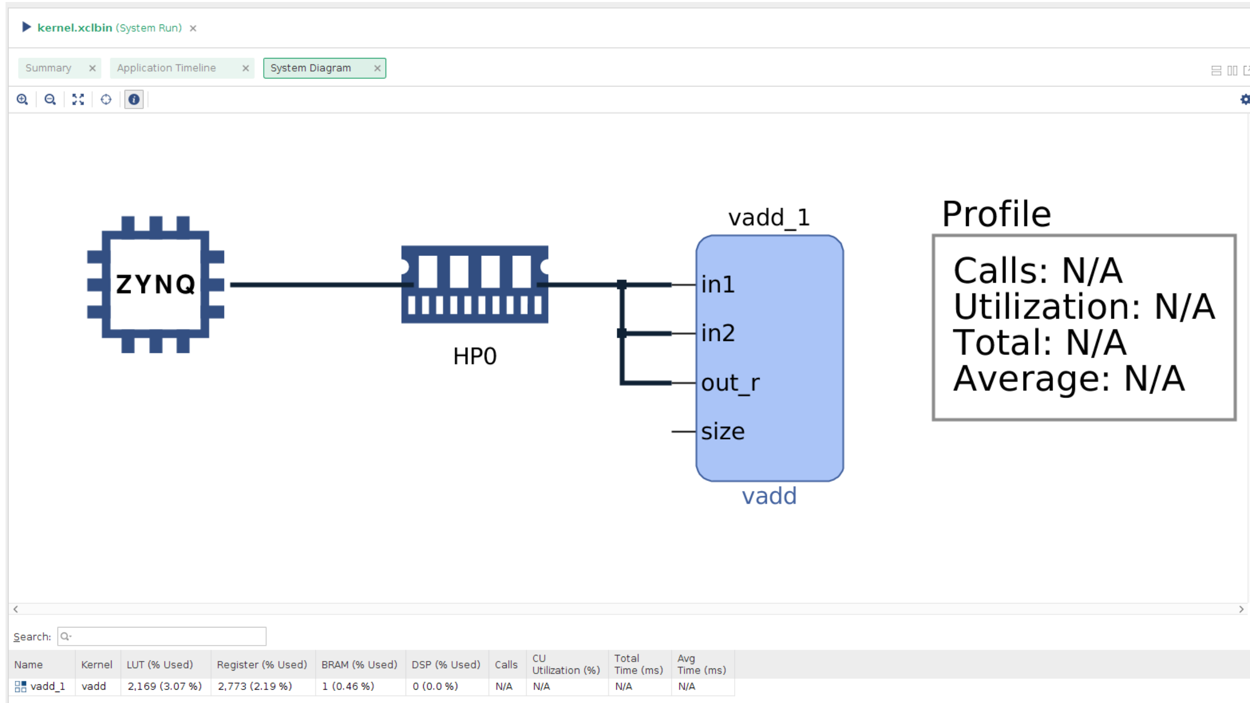
   The code you will use for this section is in the `vitis_tutorials/mult_compute_units` directory. The directory structure looks like this:

```
mult_compute_units/
  host.cpp
  vadd.cpp
  xcl2.cpp
  xcl2.hpp
```

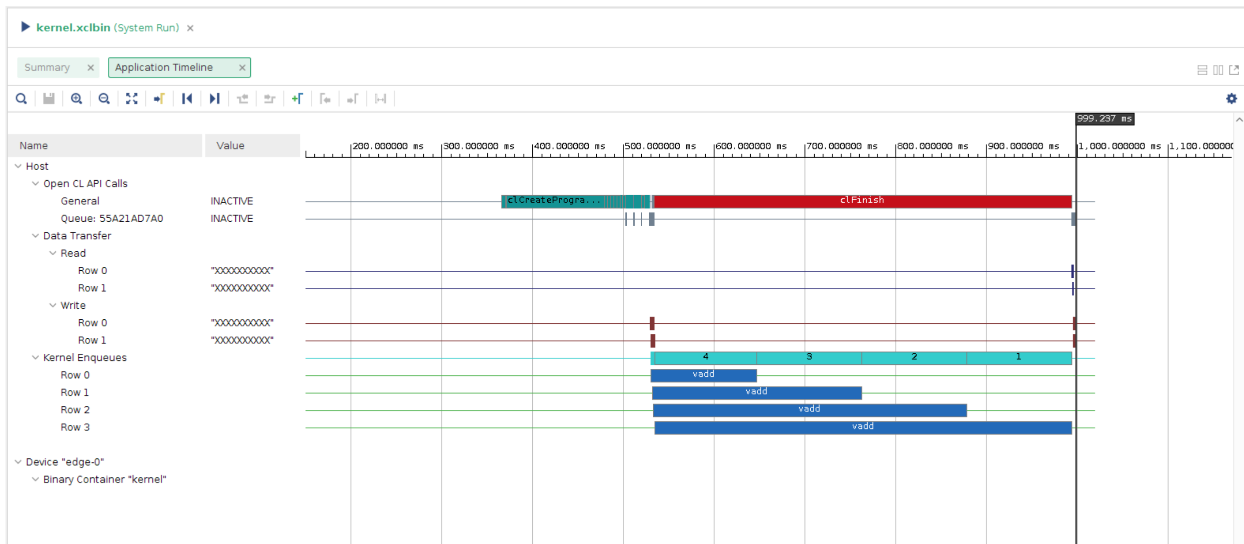
   The `host.cpp` code has the OpenCL host code. The top level HLS function is in `vadd.cpp`.

   (a) Create an application project as described in Tutorial 2, compile and run the project.

   (b) The system diagram in vitis analyzer looks like:

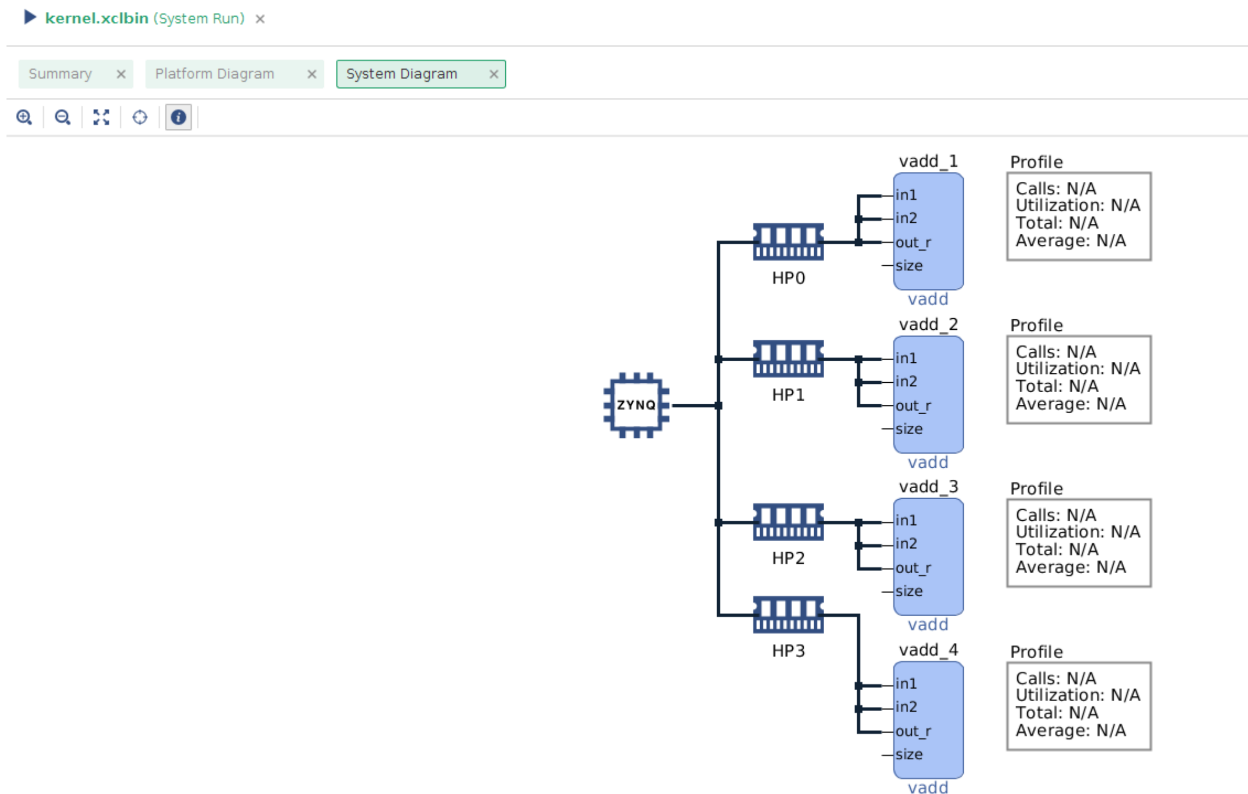


which shows that there is one vadd kernel. The application timeline looks like:

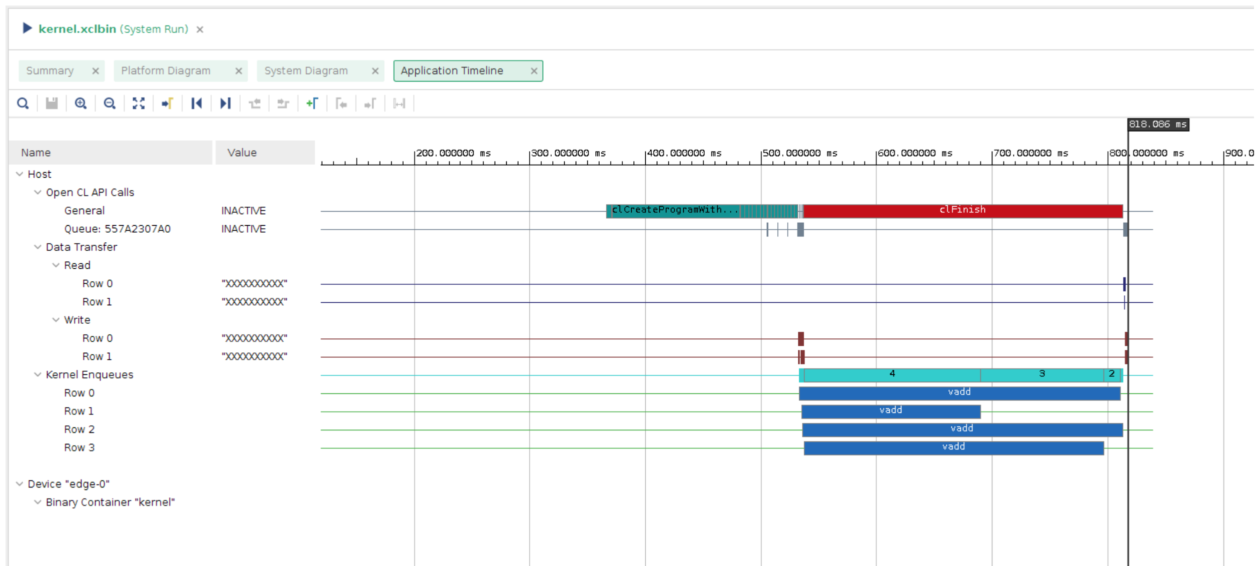


From the application trace, we can see that although the host scheduled all kernel executions concurrently, the second, third and fourth execution requests are delayed as there is only one compute unit on the FPGA.

(c) Increase the number of compute units to 4 and assign separate ports by going to the window mentioned in 2i. Compile and run the updated configuration. The vitis analyzer system diagram would look like:



The application timeline looks like:



You can now see that the application takes advantage of the four compute units, and that the kernel executions overlaps and executes in parallel.

(d) Look into the host code and learn how the multiple compute units are utilized:

```
for (int i = 0; i < num_cu; i++) {
  int narg = 0;

  // Setting kernel arguments
  OCL_CHECK(err, err = krnls[i].setArg(narg++, buffer_in1[i]));
  OCL_CHECK(err, err = krnls[i].setArg(narg++, buffer_in2[i]));
  OCL_CHECK(err, err = krnls[i].setArg(narg++, buffer_output[i]));
  OCL_CHECK(err, err = krnls[i].setArg(narg++, chunk_size));

  // Copy input data to device global memory
  OCL_CHECK(err, err = q.enqueueMigrateMemObjects( { buffer_in1[i],
      buffer_in2[i] }, 0 /* 0 means from host*/));

  // Launch the kernel
  OCL_CHECK(err, err = q.enqueueTask(krnls[i]));
}
```

You can see from the code that by creating an array of kernels and enqueueing them in a loop, you can utilize the multiple compute units.

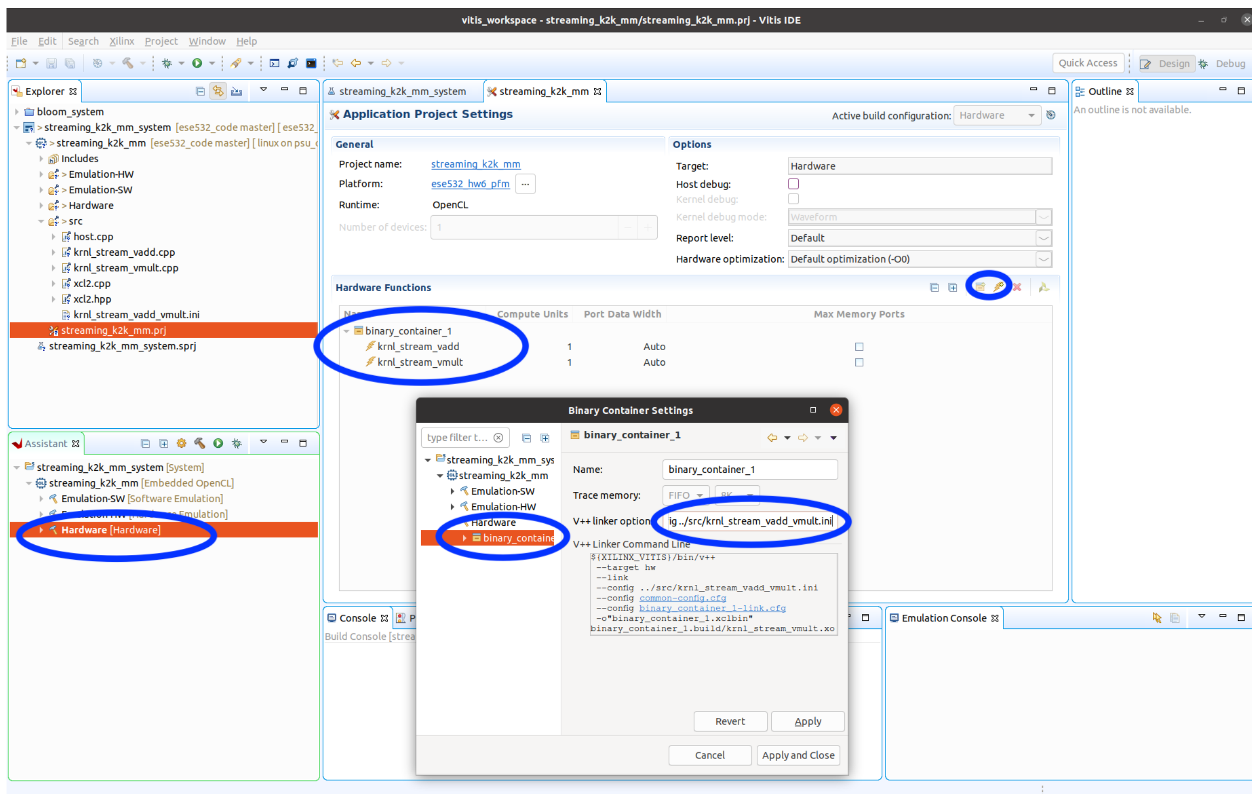
4. **Streaming Kernel to Kernel Memory Mapped**

The code you will use for this section is in the `vitis_tutorials/streaming_k2k_mm` directory. The directory structure looks like this:

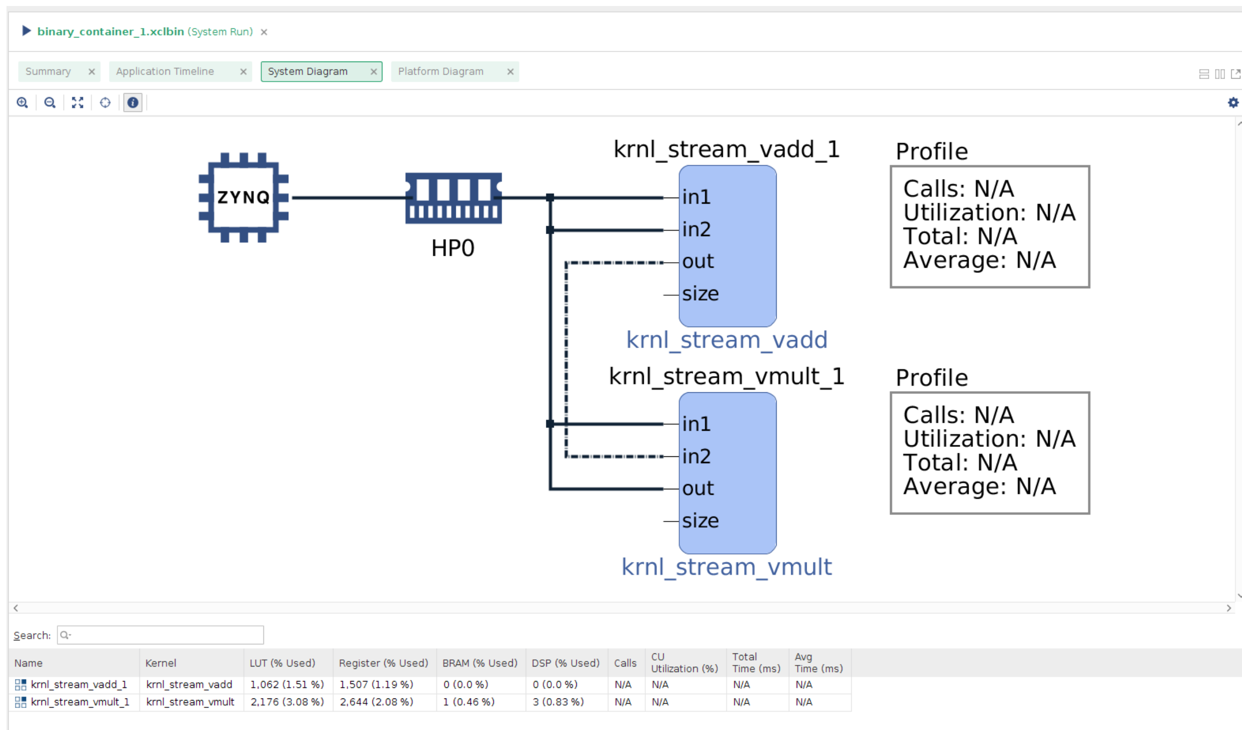
```
streaming_k2k_mm/
  host.cpp
  krnl_stream_vadd.cpp
  krnl_stream_vadd_vmult.ini
  krnl_stream_vmult.cpp
  xcl2.cpp
  xcl2.hpp
```

The `host.cpp` code has the OpenCL host code. There are two disjoint HLS kernels: `krnl_stream_vadd.cpp` and `krnl_stream_vmult.cpp`. `krnl_stream_vadd_vmult.ini` specifies how the two kernels are connected with each other. Read about the tutorial from here and then continue.

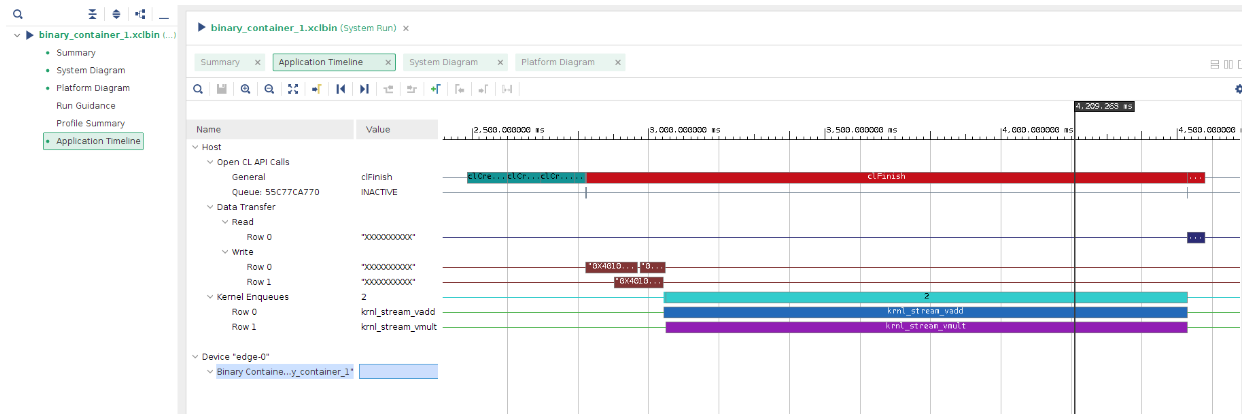
(a) Create an application project as described in Tutorial 2. Add the two kernels as hardware functions, add the V++ linker option:
`--config ../src/krnl_stream_vadd_vmult.ini`



(b) Compile and run the project. The system diagram in vitis analyzer looks like:



which shows that the two kernels are reading from the DRAM and are also connected via a stream connection. The application timeline looks like:



From the application trace, we can see that the two kernels are running concurrently.

5. **Using Faster Clocks**

(a) In Homework 6, we saw the our platform provides multiple clocks:

```
=========================
Basic Platform Information
=========================
Platform:           ese532_hw6_pfm
File:               /media/lilbirb/research/git/avnet/petalinux/projects/ese532_hw6_pfm/export/ese532_hw6_pfm/ese532_hw6_pfm.xpfm
Description:
ese532_hw6_pfm


=====================================
Hardware Platform (Shell) Information
=====================================
Vendor:                           avnet.com
Board:                            ULTRA96V2
Name:                             ULTRA96V2
Version:                          1.0
Generated Version:                2020.1
Software Emulation:               1
Hardware Emulation:               0
FPGA Family:                      zynquplus
FPGA Device:                      xczu3eg
Board Vendor:                     avnet.com
Board Name:                       avnet.com:ultra96v2:1.1
Board Part:                       xczu3eg-sbva484-1-i
Maximum Number of Compute Units:  60

=================
Clock Information
=================
  Default Clock Index: 0
  Clock Index:         0
    Frequency:         150.000000
  Clock Index:         1
    Frequency:         300.000000
  Clock Index:         2
    Frequency:         75.000000
  Clock Index:         3
    Frequency:         100.000000
  Clock Index:         4
    Frequency:         200.000000
  Clock Index:         5
    Frequency:         400.000000
  Clock Index:         6
    Frequency:         600.000000

=====================
Resource Availability
=====================
  =====
  Total
  =====
    LUTs:  57915
    FFs:   126868
    BRAMs: 212
    DSPs:  360
```

(b) We can assign faster clocks to our kernels in Tutorial 4. You can specify them in a configuration file and pass it in the V++ Linker Options. Looking at the `krnl_stream_vadd_vmult.ini`, you can see that we have assigned Clock Index 1 (300 Mhz) to the kernels:

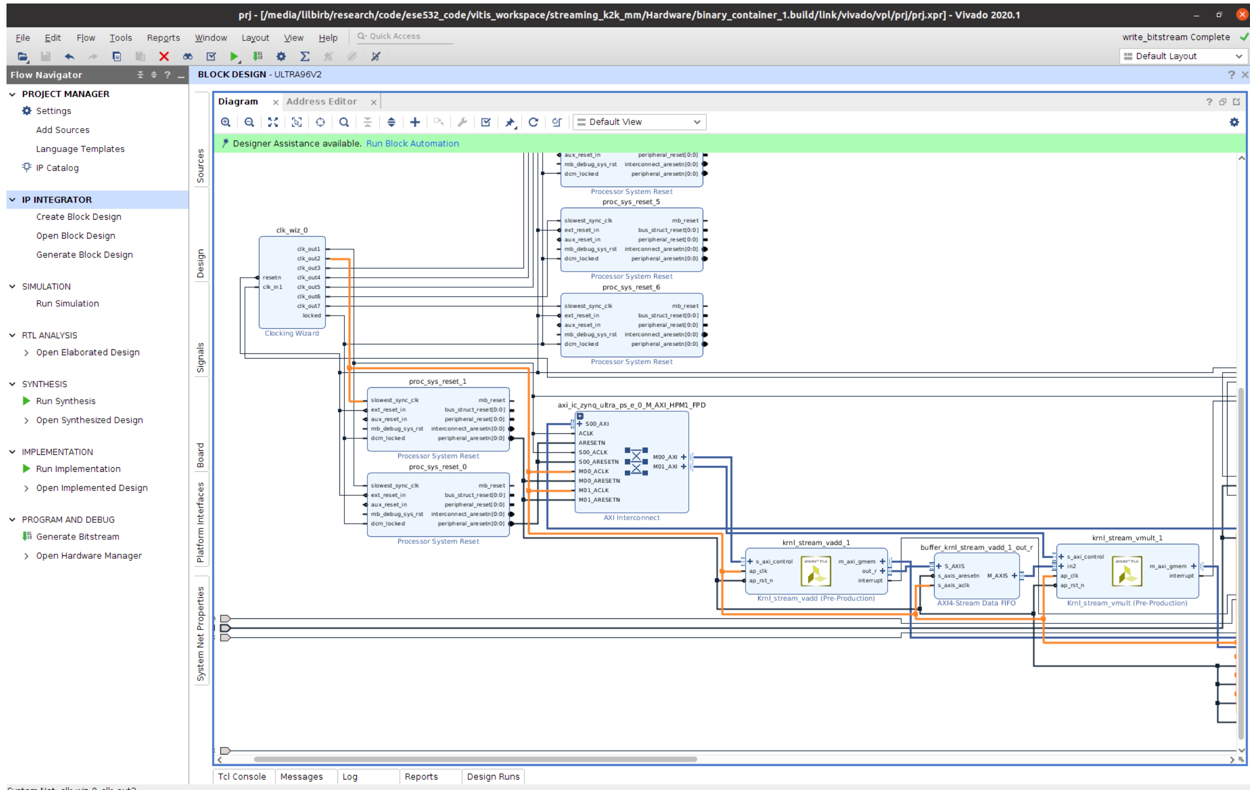
```
[connectivity]
stream_connect=krnl_stream_vadd_1.out:krnl_stream_vmult_1.in2:64

[clock]
id=1:krnl_stream_vadd_1
id=1:krnl_stream_vmult_1
```

where the format of the specification is `id=<clock index>:<compute unit name>`. You should start with a slower clock in your project so that you can meet timing easily. After you have made HLS and host code optimizations, you can try

increasing the clock frequency until your design fails to meet timing.

(c) You can check if the clocks were correctly assigned by opening the vivado project as instructed in Homework 6:



You can see from the vivado block diagram that clock index 1 is assigned. Moreover, you can also see that an AXI Stream FIFO is connecting the two kernels.