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

ESE532, Fall 2021 — I/O and FPGA Milestone — Wednesday, November 10

Due: Friday, Nov. 19, 5:00PM

Group: All work is group work. Single turn-in for group.

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 (Tutorial 1) to report the input throughput to your encoder in the writeup.

3. Use the supplied measurement routines to report the maximum real-time throughput the current design can sustain in the writeup (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) Compile the encoder code, copy the binary to the Ultra96 and run it.

   (b) Download `vmlinuz.tar` from the Project handout. 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
   ```

   (c) 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
   ```

   (d) 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;
   ```

(e) 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.

2. **Using Multiple Compute Units**

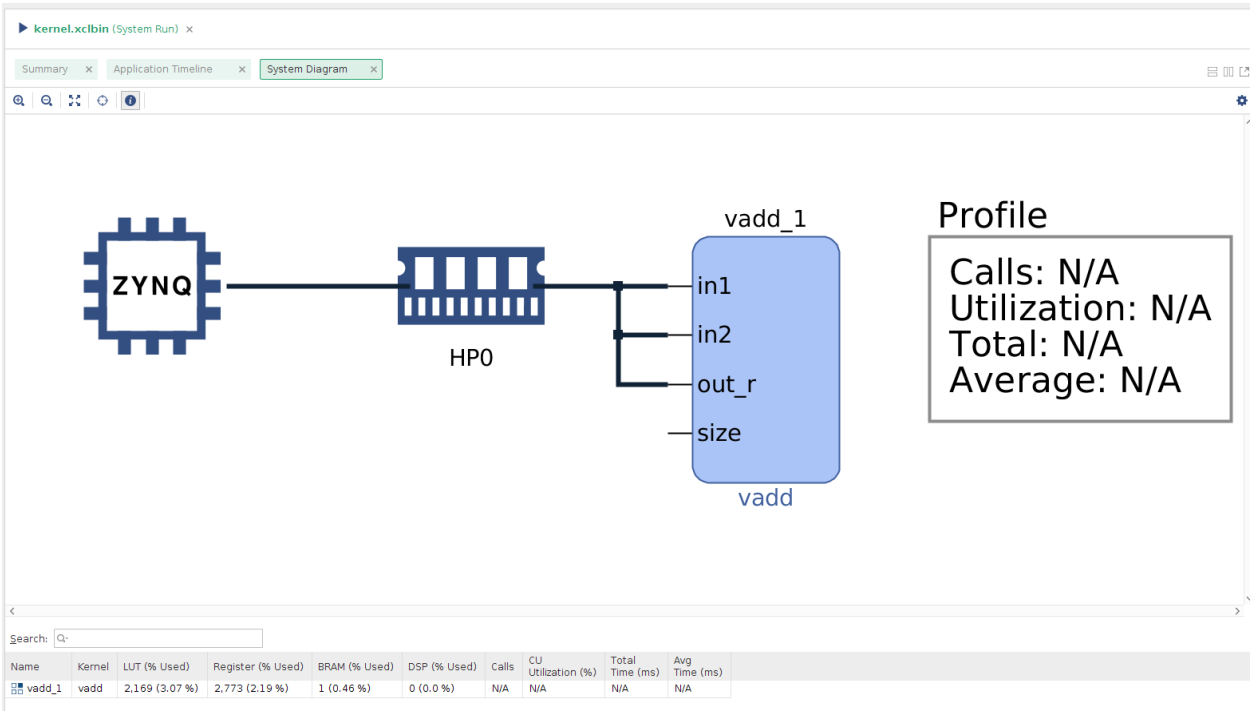
The code you will use for this section is in the `vitis_tutorials/mult_compute_units` directory. The directory structure looks like this:

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

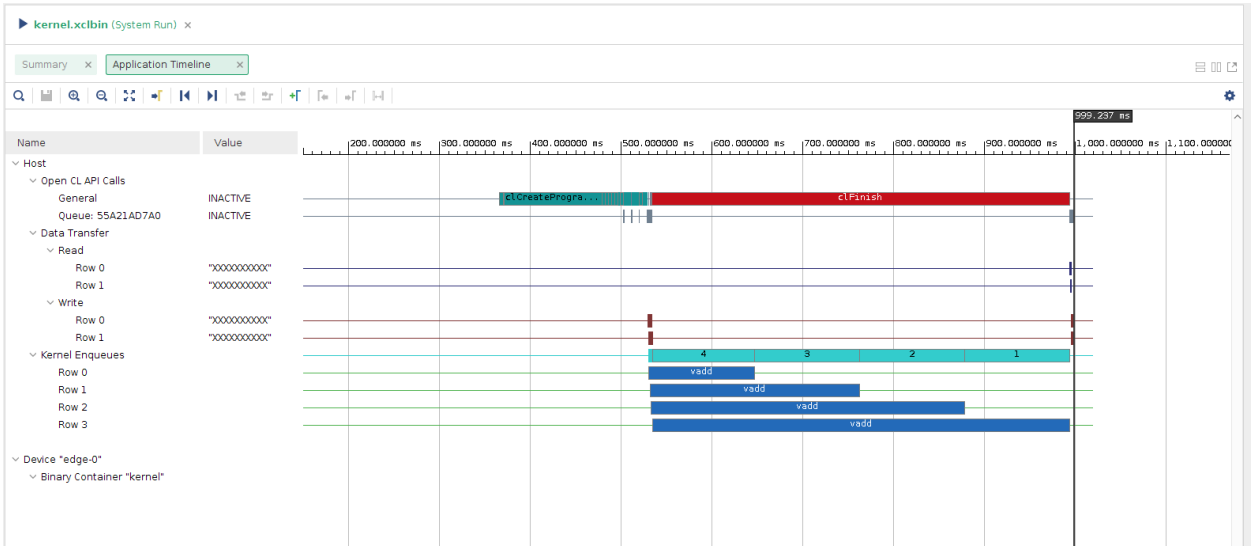
The `host.cpp` code has the OpenCL host code. The top level HLS function is in `vadd.cpp`.

(a) Create an application project, compile and run the project.

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

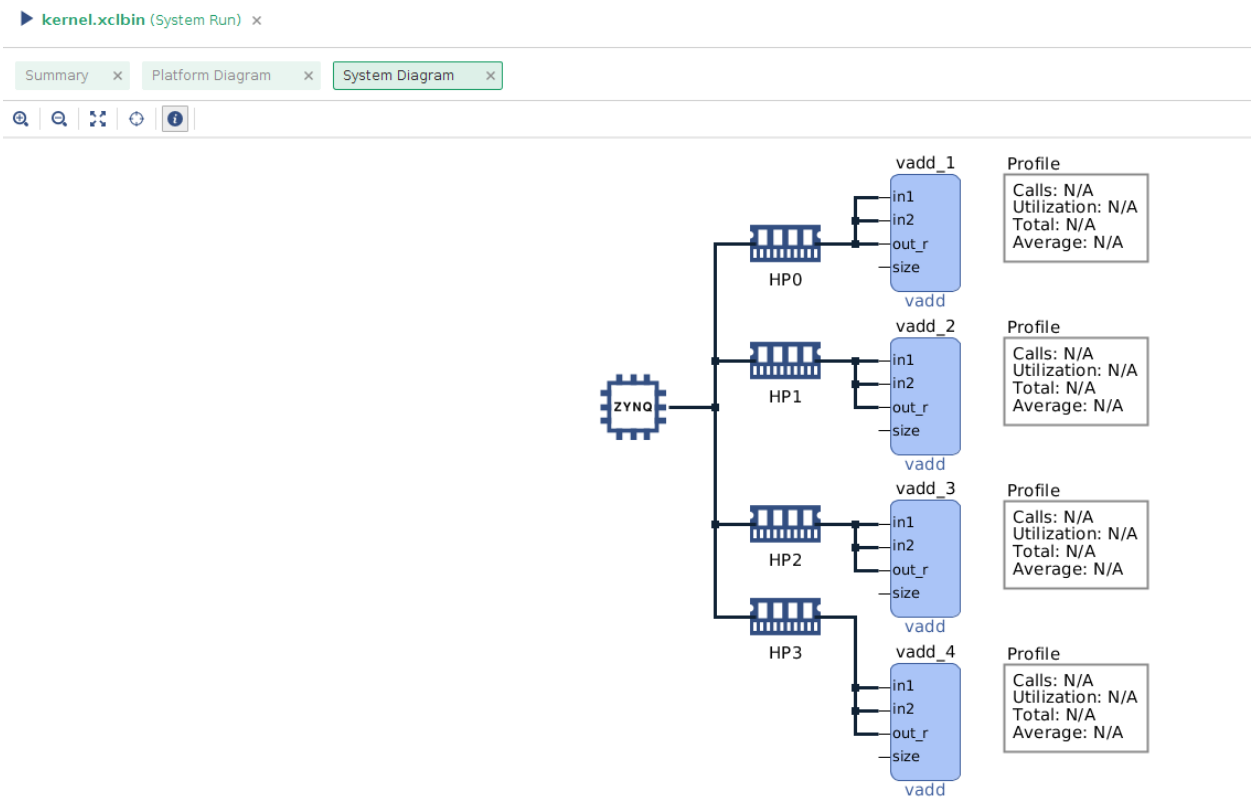


which shows that there is one vadd kernel. The application timeline looks like:

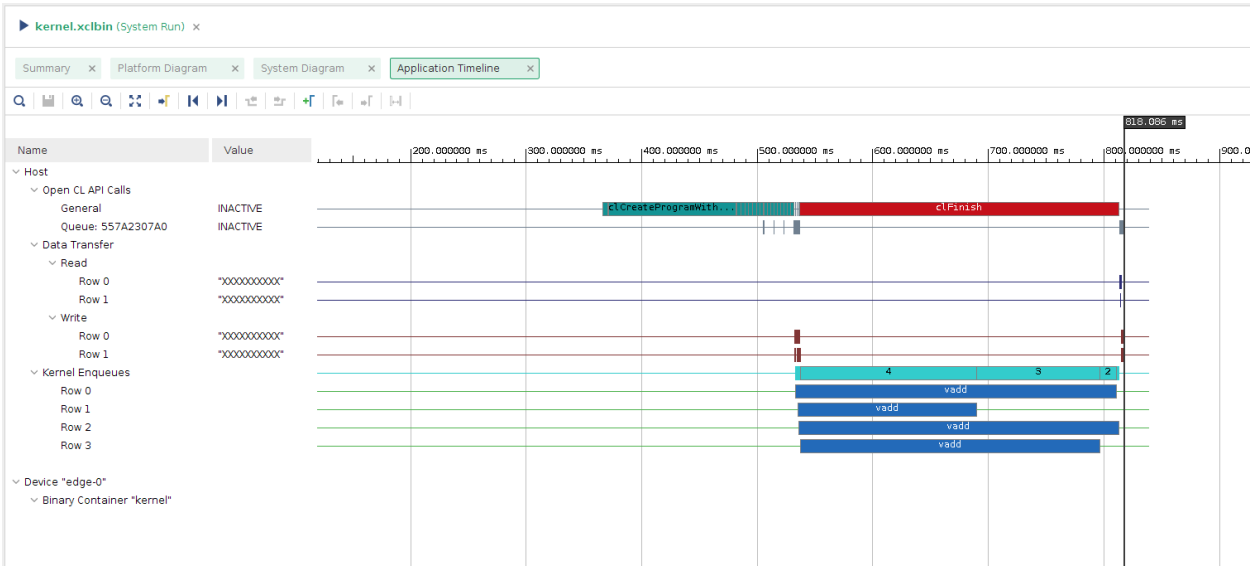


From the application trace, we can see that although the host scheduled all kernel executions concurrently, the second, third and fourth execution requests are delayed as there is only one compute unit on the FPGA.

(c) Increase the number of compute units to 4 and assign separate ports by going to the window mentioned in the Bloom filter tutorial on P2. 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.

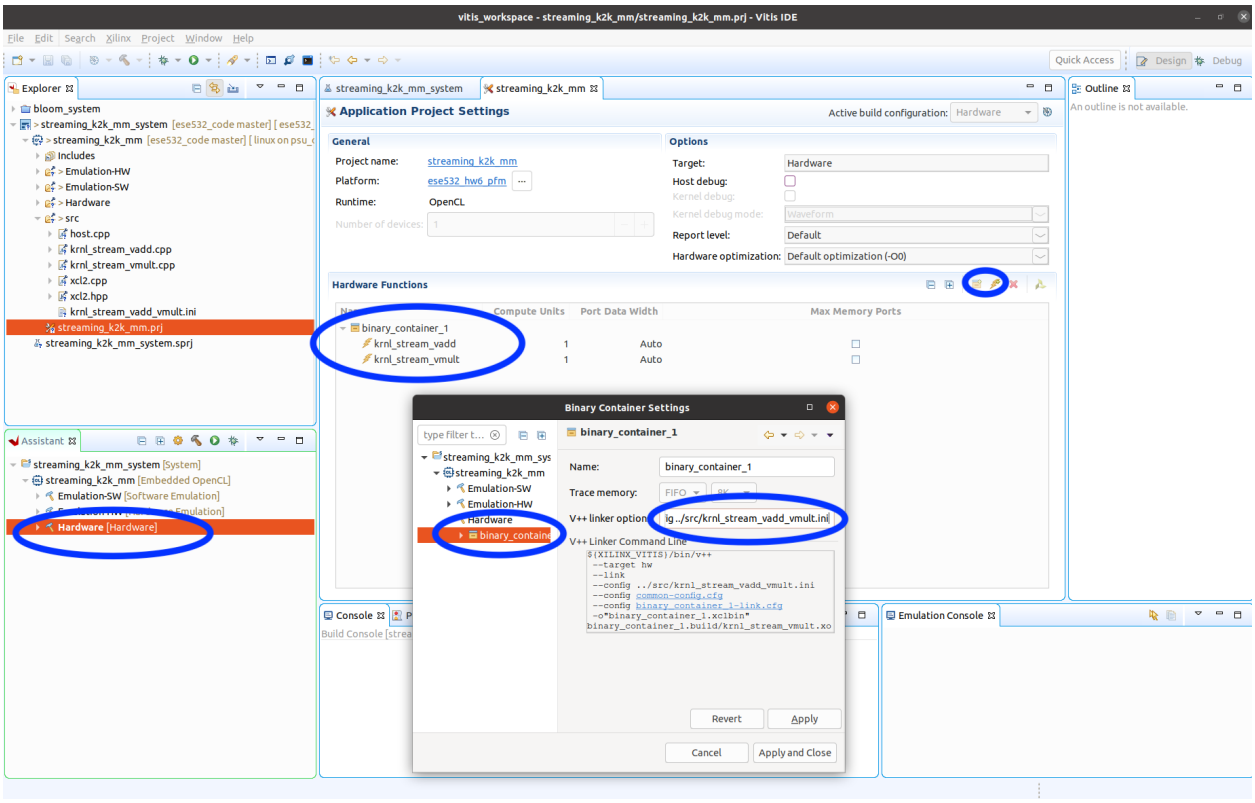
3. **Streaming Kernel to Kernel Memory Mapped**

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

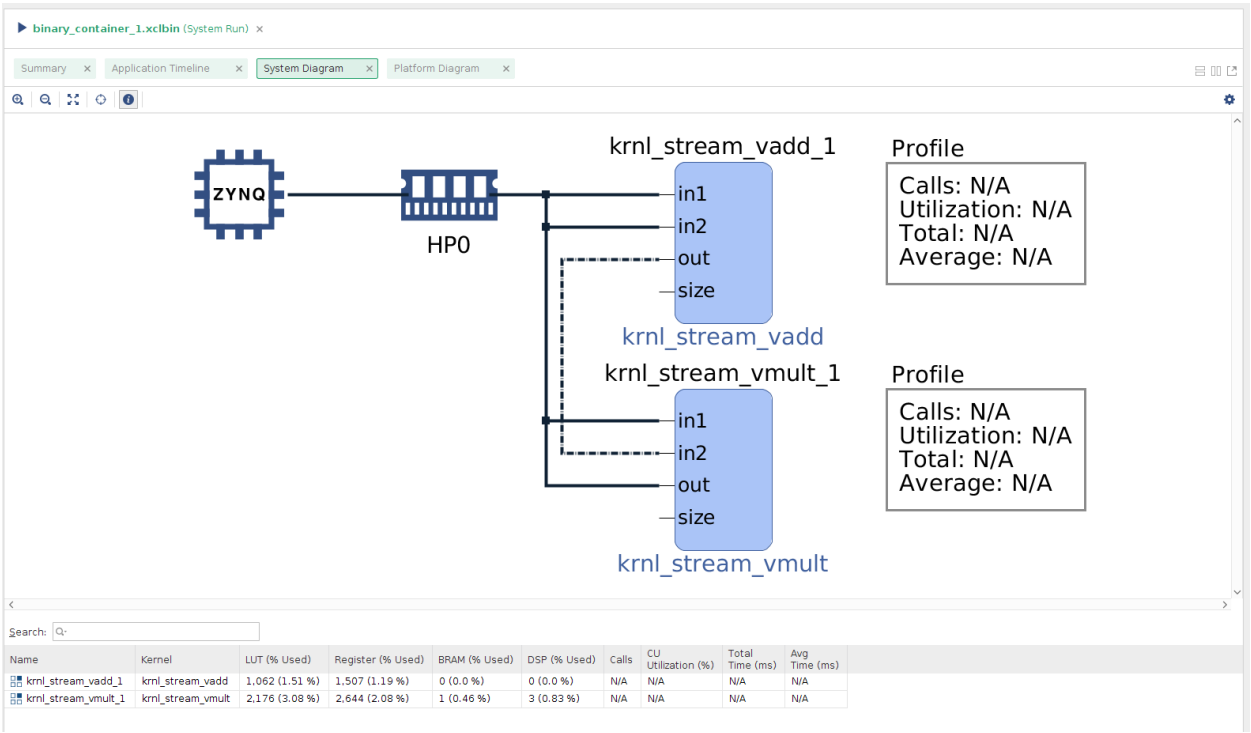
```
streaming_k2k_mm/
  host.cpp
  krnl_stream_vadd.cpp
  krnl_stream_vadd_vmult.ini
  krnl_stream_vmult.cpp
  xcl2.cpp
  xcl2.hpp
```

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.

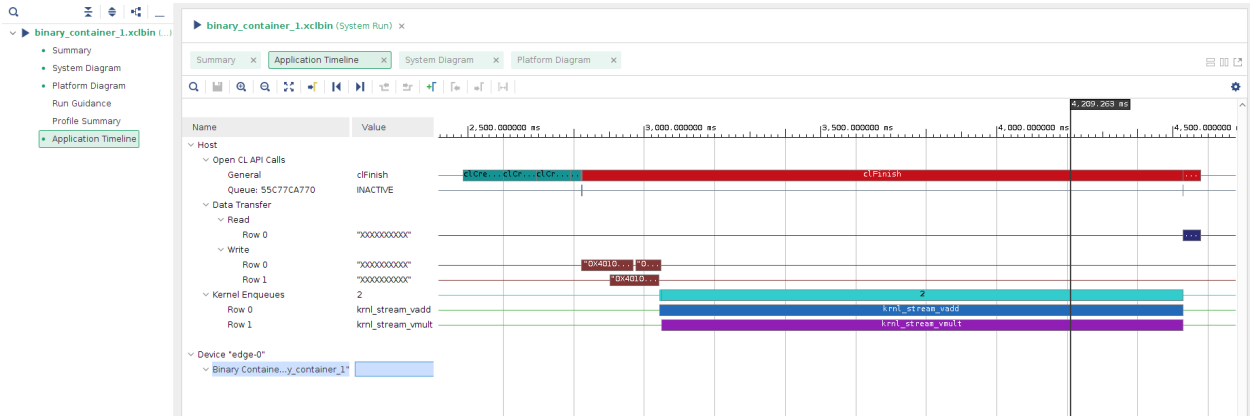
(a) Create an application project. Add the two kernels as hardware functions, add the V++ linker option:
`--config ../src/krnl_stream_vadd_vmult.ini`



(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.

4. **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 3. 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:

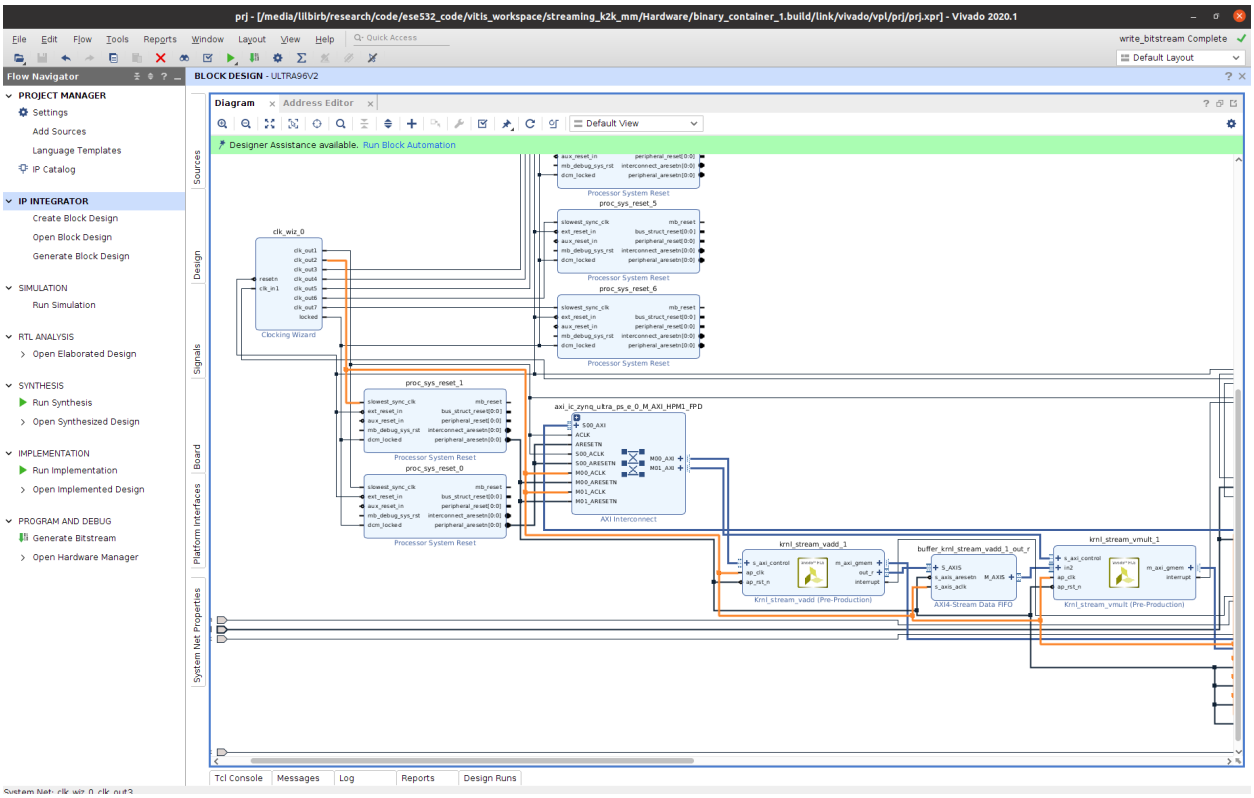
```
[connectivity]
stream_connect=krnl_stream_vadd_1.out:krnl_stream_vmult_1.in2:64

[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.