AAHLS LabA report

R11943018 林子軒

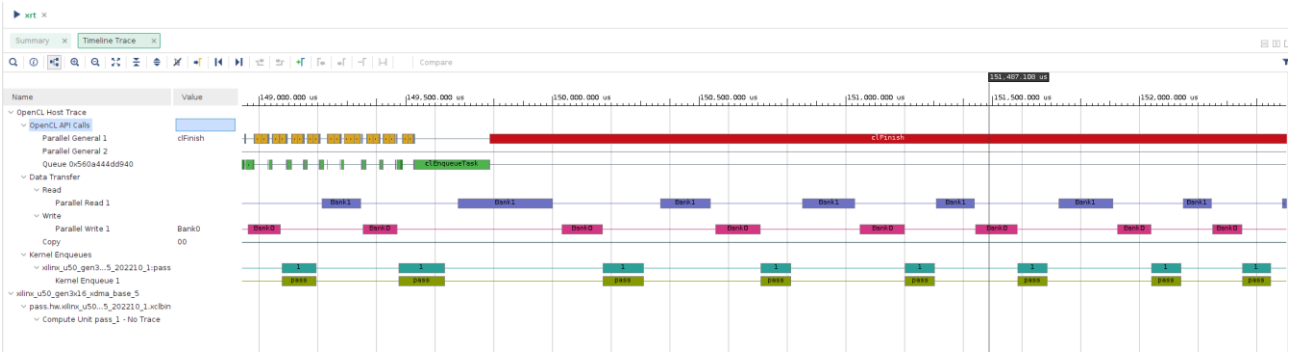
Github link: https://github.com/ezpzsyuan00/AAHLS_labA

Lab 1: Pipelined Kernel Execution Using Out-of-Order Event Queue

A. In ./src/pipeline_host.cpp, the oooQueue parameter is set to false.

```
// -- Common Parameters ----------------------------

unsigned int numBuffers                 = 10;
bool         oooQueue                   = false;
unsigned int processDelay               = 1;
unsigned int bufferSize                 = 8 << 11;
```

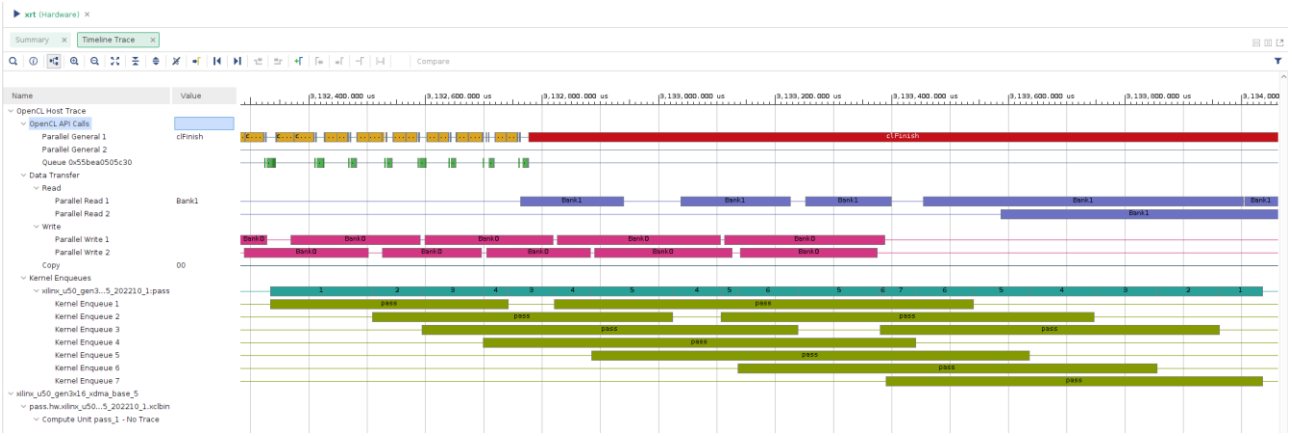


We can see that every task execution has a fixed order. The order o is Wirte, Execute then Read.

B. Next step we change the oooQueue parameter to be set to true.

```
// -- Common Parameters ----------------------------

unsigned int numBuffers                 = 10;
bool         oooQueue                   = true;
unsigned int processDelay               = 1;
unsigned int bufferSize                 = 8 << 11;
```



We can see that the dependencies is only in their own group. This allows the read and write operations to overlap with the execution.

Lab 2: Kernel and Host Code Synchronization

A. The original code

```
// -- Execution ----------------------------

for(unsigned int i=0; i < numBuffers; i++) {
  tasks[i].run(api);
}
clFinish(api.getQueue());
```

The clfinish() function is blocked until all commands in the queue are executed, ensuring that all execution commands in the command queue are executed on the GPU before the host executes the next step.

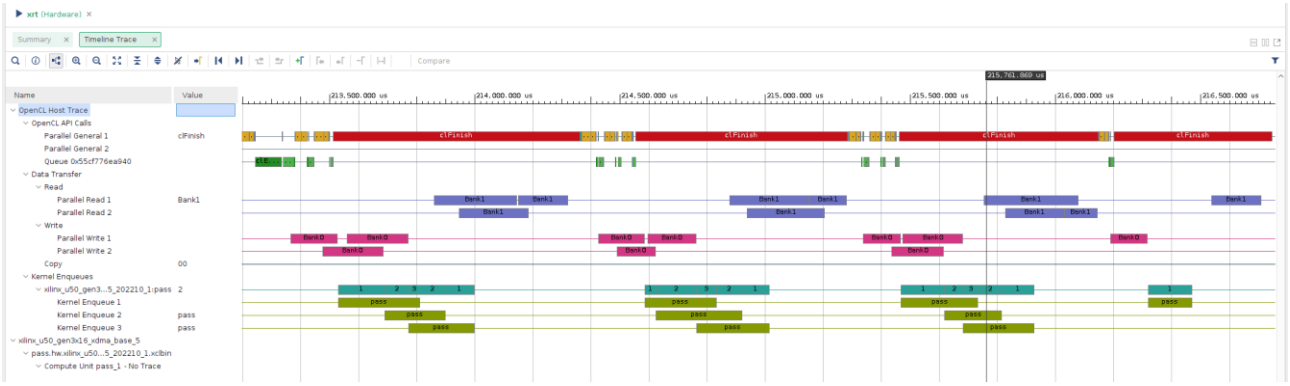
There could be issues if the numBuffer variable is increased to a large number, which would occur when processing a video stream. In this case, buffer allocation and memory usage can become problematic because the host memory is pre-allocated and shared with the FPGA. Finally, run out of memory.

B. To reduce the pressure of buffer, we modify the execution loop.

```
// -- Execution ----------------------------

int count = 0;
for(unsigned int i=0; i < numBuffers; i++) {
  count++;
  tasks[i].run(api);
  if(count == 3) {
  count = 0;
  clFinish(api.getQueue());
  }
}
clFinish(api.getQueue());
```

The call to clFinish() creates a synchronization point on the complete OpenCL command queue. That is to say all commands enqueued onto the given queue have to be completed before clFinish() returns control to the host program.

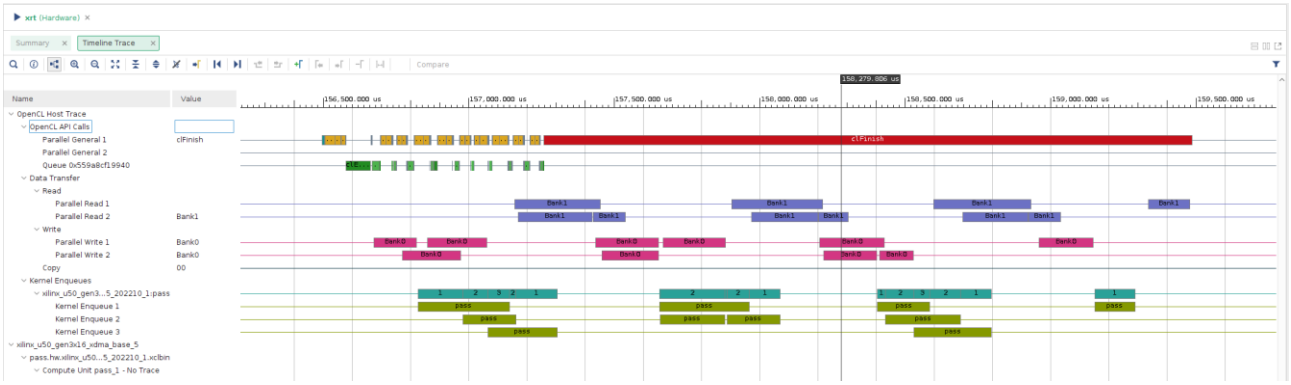


We can observe that Read/Execute/Write all form groups of three.

C. the synchronization is performed based on the completion of a previous execution of a call to the accelerator.

```
// -- Execution ----------------------------

for(unsigned int i=0; i < numBuffers; i++) {
  if(i < 3) {
    tasks[i].run(api);
  } else {
    tasks[i].run(api, tasks[i-3].getDoneEv());
  }
}
clFinish(api.getQueue());
```



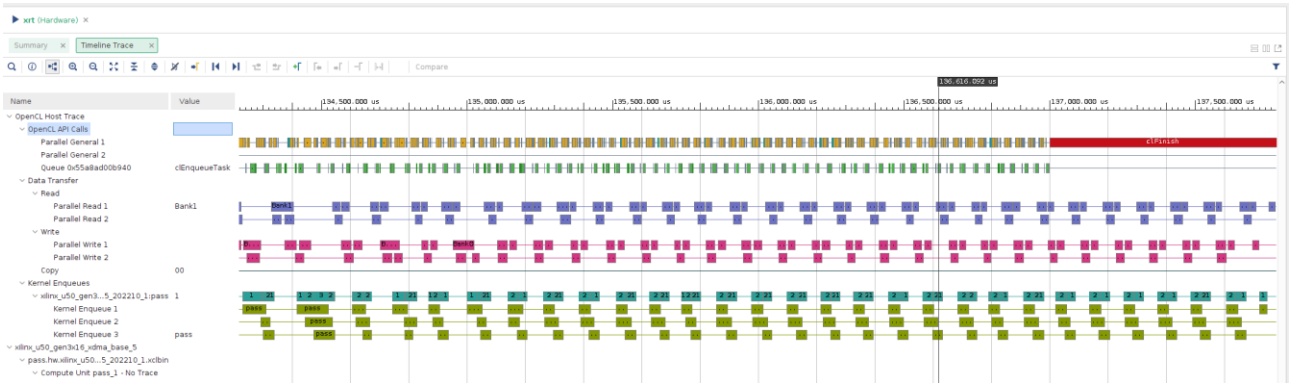
The biggest difference between B and C is the number of times clfinish() called. In addition, the gap between the previous Write and the next Read is vanished.

Lab 3: OpenCL API Buffer Size

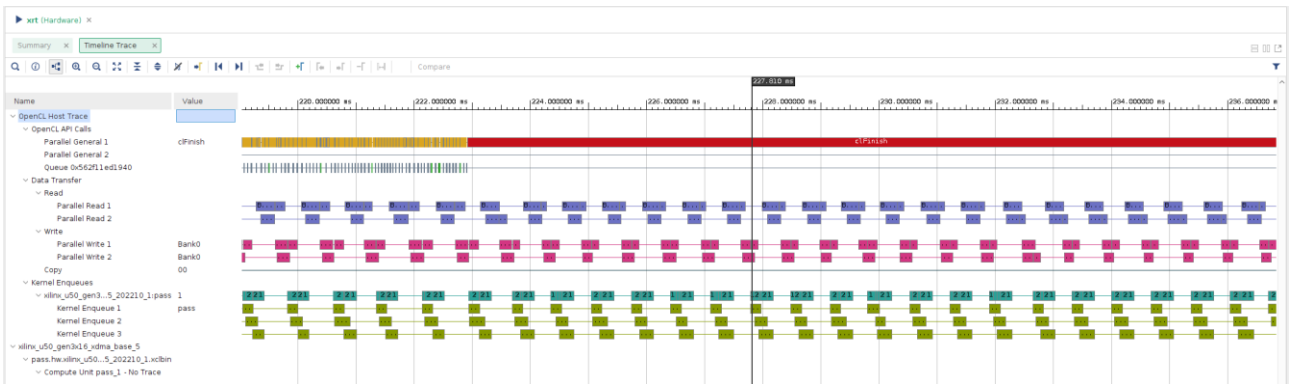
In the src/buf_host.cpp file the number of tasks to be processed has increased to 100.

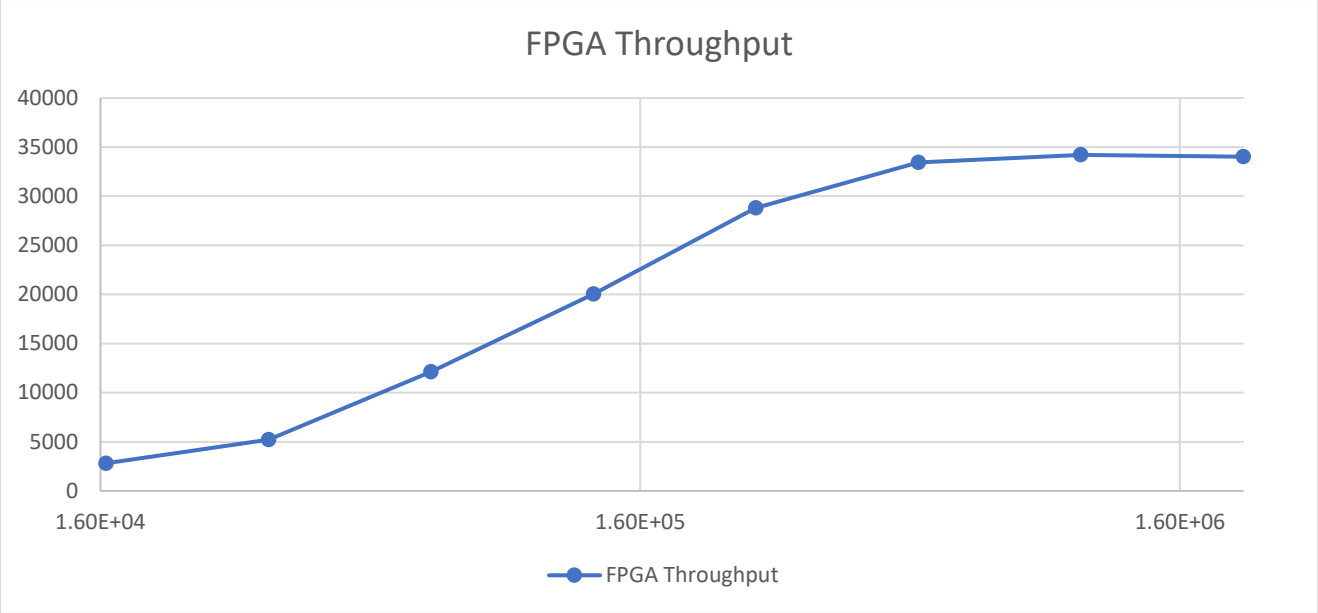
Size = 8

Size = 11



Size = 14





This image shows that the buffer size (x-axis, bytes per transfer) clearly impacts performance (y-axis, FPGA Throughput in MB/s), and starts to level out around 2 MB.

When setting size >= 16

```
 Total number of buffers: 100
              BufferSize: 65536
        Bits per Element: 512
     Bytes per Transfer: 4194304
            processDelay: 1
     Out of Order Queue: true

Running FPGA
XRT build version: 2.13.466
Build hash: f5505e402c2ca1ffe45eb6d3a9399b23a0dc8776
Build date: 2022-04-14 17:43:11
Git branch: 2022.1
PID: 2454447
UID: 1065
[Sun Oct 16 18:28:24 2022 GMT]
HOST: HLS04
EXE: /mnt/HLSNAS/04.HckaBV/Vitis-Tutorials/Hardware_Acceleration/Design_Tutorials/07-host-code-opt/reference-files/buf/pass
[XRT] ERROR: std::bad_alloc
[XRT] ERROR: std::bad_alloc
[XRT] ERROR: std::bad_alloc
[XRT] ERROR: std::bad_alloc
[XRT] ERROR: std::bad_alloc
[XRT] ERROR: event is nullptr
[XRT] ERROR: std::bad_alloc
[XRT] ERROR: std::bad_alloc
[XRT] ERROR: std::bad_alloc
[XRT] ERROR: std::bad_alloc
[XRT] ERROR: std::bad_alloc
[XRT] ERROR: event is nullptr
[XRT] ERROR: std::bad_alloc
[XRT] ERROR: std::bad_alloc
[XRT] ERROR: std::bad_alloc
[XRT] ERROR: std::bad_alloc
[XRT] ERROR: std::bad_alloc
/bin/sh: line 1: 2454447 Bus error               (core dumped) ./pass .././xclbin/pass.hw.xilinx_u50_gen3x16_xdma_5_202210_1.xclbin 16
make: *** [Makefile:52: run] Error 135
```

Explain what problem you encountered and how you solved it

1. The boards we can access are xilinx_u50_gen3x16_201920_3 and xilinx_u50_gen3x16_5_202110_1.

   No matter which board we choose, we both need to handle this error.

   Run the following makefile command to compile the kernel to the specified accelerator card.

   ```
   make TARGET=hw DEVICE=xilinx_u200_gen3x16_xdma_2_202110_1 xclbin
   ```

   We can see the sp by “platforminfo -p xilinx_u50_gen3x16_5_202210_1” and know that the form of sp tag is HBM.

   ```
   ERROR: [CFGEN 83-2287] --sp tag applied with an invalid sp tag: DDR[0]
   ERROR: [CFGEN 83-2287] --sp tag applied with an invalid sp tag: DDR[1]
   ERROR: [CFGEN 83-2297] Please consult platforminfo <platform.xpfm path> for sptag information
   ```

   Therefore, I change DDR to HBM in design.cfg

   ```
   [connectivity]
   nk=pass:1
   sp=pass_1.m_axi_p0:HBM[0]
   sp=pass_1.m_axi_p1:HBM[1]
   ```

2. The most annoying part is this command in the tutorial is wrong. I can't find where the output file is at first so I can't view the timeline. Spending a lot of time to looking for “.run_summary” in a bunch of folders.

   After the run completes, open the Application Timeline using the Vitis analyzer, then select the Application Timeline located in left side panel.

   ```
   vitis_analyzer runPipeline/pass.hw.xilinx_u200_gen3x16_xdma_2_202110_1.xclbin.run_summary
   ```

   ```
   04.HckaBV@HLS04:~/Vitis-Tutorials/Hardware_Acceleration/Design_Tutorials/07-host-code-opt/reference-files$ make TARGET=hw DEVICE=xilinx_u200_gen3x16_xdma_2_202110_1 bufRunSweep
   cp auxFiles/xrt.ini buf
   cp auxFiles/run.py buf
   cd buf; ./run.py
   /bin/sh: ./run.py: Permission denied
   ```

3. In ./auxFiles/run.py, There's a big problem in run.stdout.readline. It won't produce anything and lead to infinite loop. In the end, I can't get the FPGA throughput automatically like the tutorial. I have to draw a line chart by manually adjust the values of size and repeat the experiment many times!!

   ```
   for i in range(8, 20):
       fields = {}
       buffersize = 1 << i
       print (" Running with argument %s transfers %s bytes" % (i, buffersize*512/8))
       run = subprocess.Popen(['./pass', '../xclbin/pass.hw.'+sys.argv[1]+'.xclbin', str(i)], stdout = subprocess.PIPE)
       for line in iter(run.stdout.readline, ''):
           extract ( fields , line.rstrip())
       s=""
       s += '%s, %s' % ( fields["Bytes per Transfer"], fields["FPGA Throughput"])
       out.write(s)
       out.write("\n")
   ```