

LAB #A

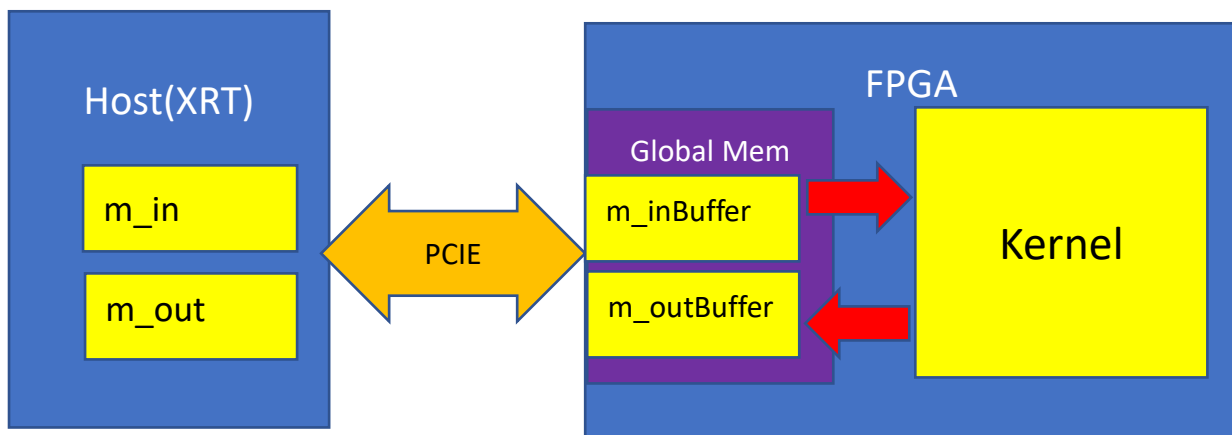
電機所 111061549 張耀明

這個主題是 Host_Code_Optimization，其中包含三個 lab，但跑的是同一個演算法。只是 kernel host code 定義不同的方式去呼叫這個演算法。當我們需要呼叫這個演算法很多次的時候，要怎麼呼叫才能有很好的效率。

系統介紹

系統描述：

為了瞭解這個 topic 在做甚麼，需要先了解整體系統的架構。
首先是 host-FPGA 的 block diagram



如上圖，在 host 端 allocate 了兩個 memory 分別是 m_in 和 m_out。

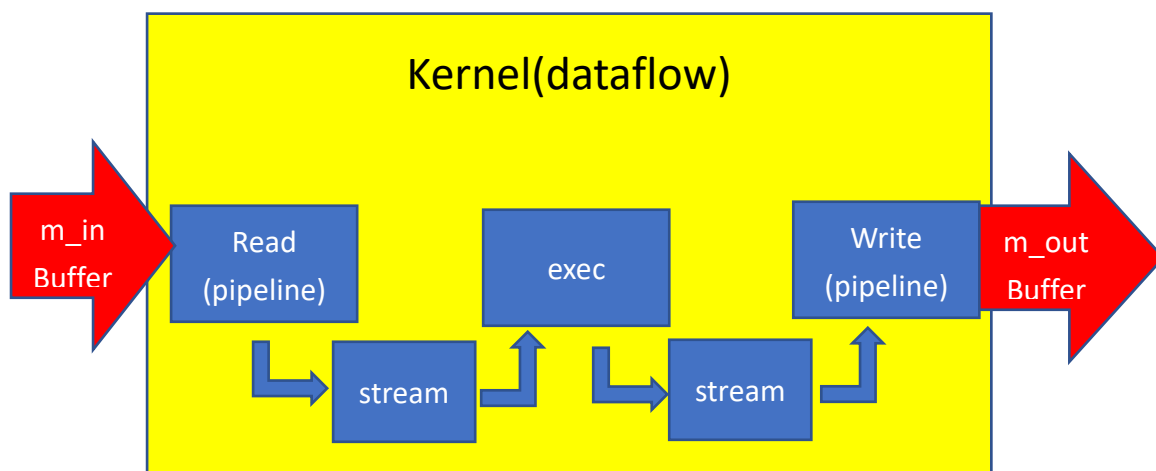
m_in 負責將值傳到 FPGA 上做運算，m_out 則是承接 FPGA 回來的回傳值。

這兩個分別對應到 FPGA 上的 m_inBuffer 和 m_outBuffer，並由 PCIE 去傳遞資料。

那個 Kernel 的系統架構如下：

如下圖，kernel 先從 global memory 把 data **read** 進來，然後 **exec**(把 data 加上一個常數)，接著 **write** 回 global memory，而 memory 到 kernel 的傳輸是使用 m_axi 來做。值得注意的是，read write 都是利用 pipeline 的方式優化，但整個 Kernel 的流程是使用 **Dataflow**。所以 Kernel 的三個 block 之間皆是以 stream 的方式做傳送。

至於為何要提到 Dataflow，是因為後面有一些有趣的狀況發生。



系統參數：

首先在本報告中，我會頻繁提到” task”。一個 task 代表一個資料從 host 端傳到 fpga 做完運算再傳回 host 這整個流程。

這個系統有以下的參數

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

numBuffers：要執行多少次 task

oooQueue: out of order queue 的啟用，這意味著 out of order execution

processDelay，實際 data 在 kernel 運算的時間(不會改變)

bufferSize，每個 buffer 的大小(不會改變)

專案架構：

專案由這些檔案構成：

Host 端：

```
srcCommon/      ->opencl code 來定義 hostcode 流程  
    AlignedAllocator.h //宣告在 host 端的記憶體  
    ApiHandle.h //基本設定：context、program、device_id、exec kernelcommand_queue  
    task.h //定義資料從 host 端傳到 fpga 做完運算再傳回 host 這整個流程  
src/            ->host code optimization 透過不同的設定來優化 hostcode  
    buf_host.cpp //lab3  
    pipeline_host.cpp //lab1  
    sync_host.cpp //lab2
```

Kernel 端：

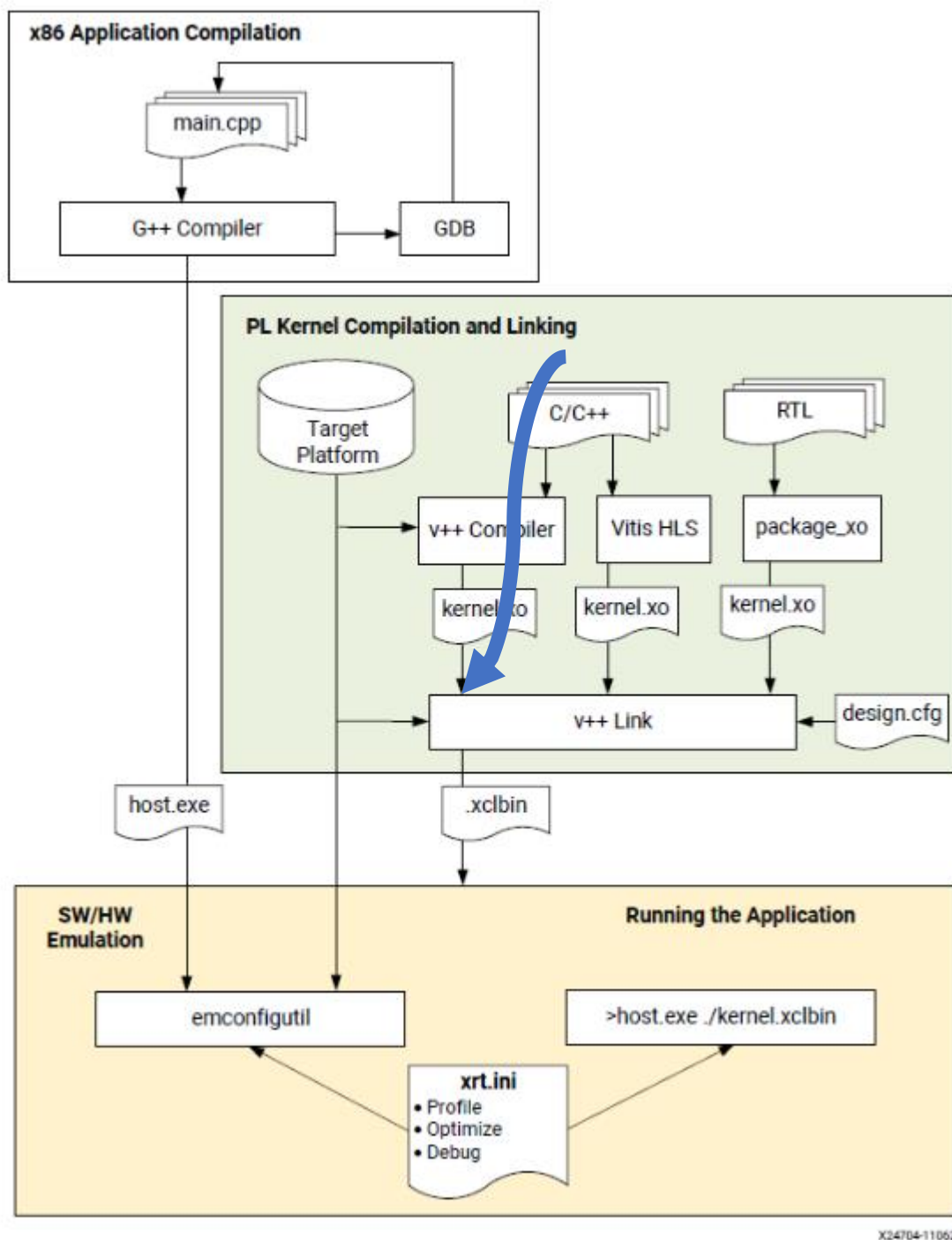
```
src/  
    pass.cpp 只是把輸入加上 processdelay 而已。它幹嘛不是本報告的重點。
```

實驗與分析

專案建置：

在進入實際操作之前，需要了解一下這個專案的建置方法。

這個專案主要是走如下圖的箭頭所示的建置路徑，並且透過 **Makefile** 來描述他的編譯方式，然後利用 V++ Compiler 編譯，生成 Kernel.xo，透過 V++ link 跑出.xclbin 檔。可能之前已經利用 Vitis HLS 的 tool 設計好了，我們僅需要修改一些參數跟下不同的指令就可以完成這個 LAB。



但是這篇 code 並非設計給現有的 U50 平台，而是設計給 U200 平台來做使用。所以在做這個 LAB 之前，需要修改 Makefile 檔案和對應的一些 cfg 檔案來確保整篇 project 可以跑在 U50 平台上。

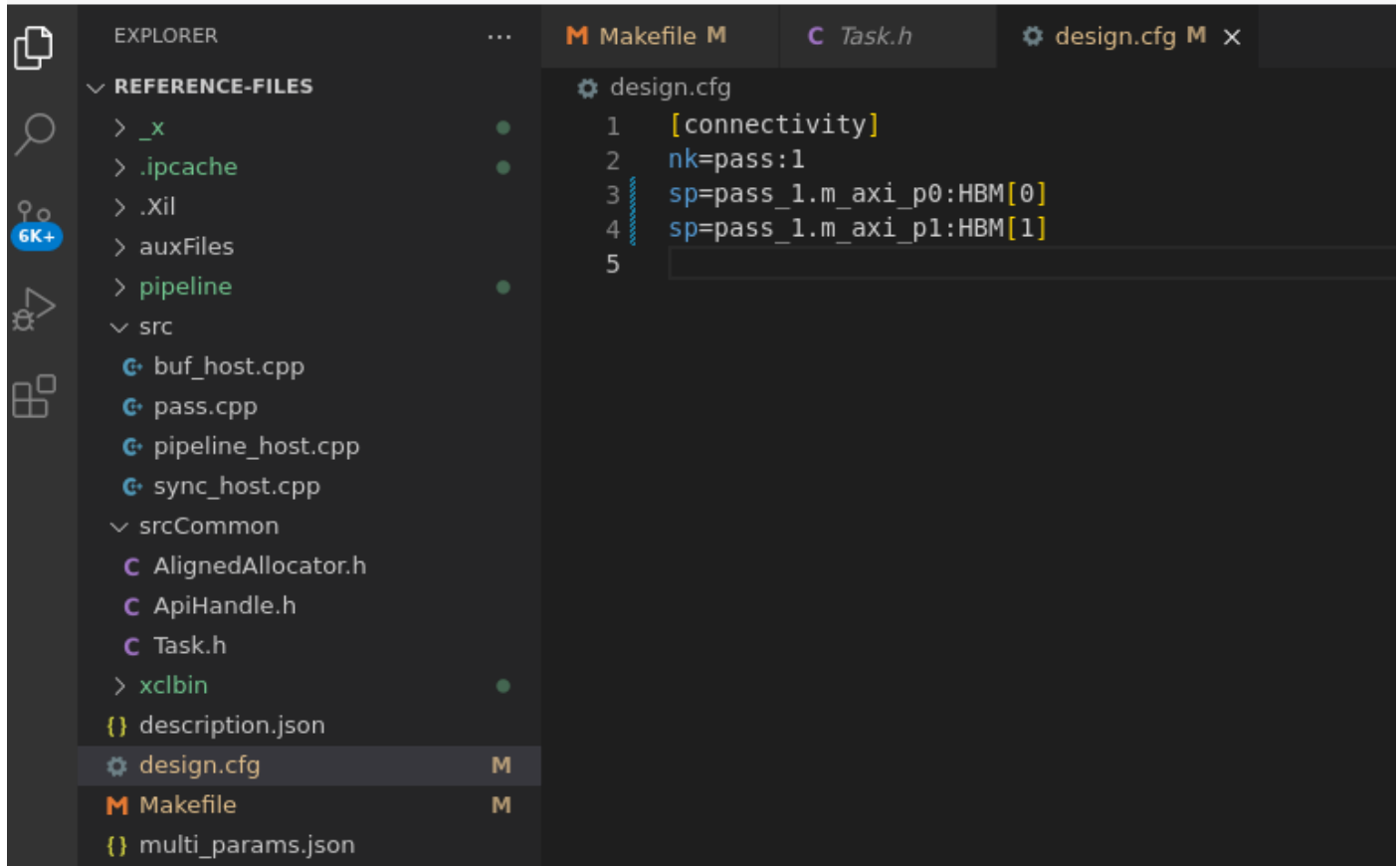
前置設定操作

首先先借用 u50 的板子，接著從 github 上把 code 給 clone 下來：

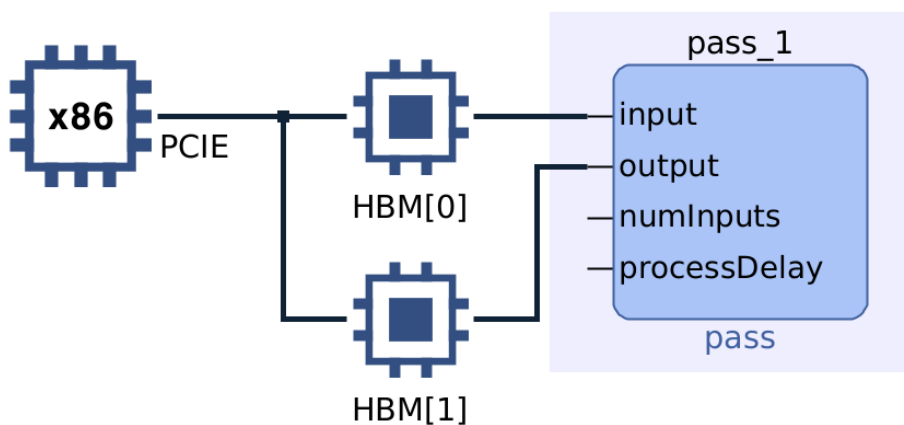
```
>git clone https://github.com/Xilinx/Vitis-Tutorials
```

然後 cd 到/Vitis-Tutorials/Hardware_Acceleration/Design_Tutorials/07-host-code-opt/reference-files\$底下，輸入 code . 打開 vscode

因為 u50 沒有 DDR 記憶體，要將 DDR 改成 HBM。

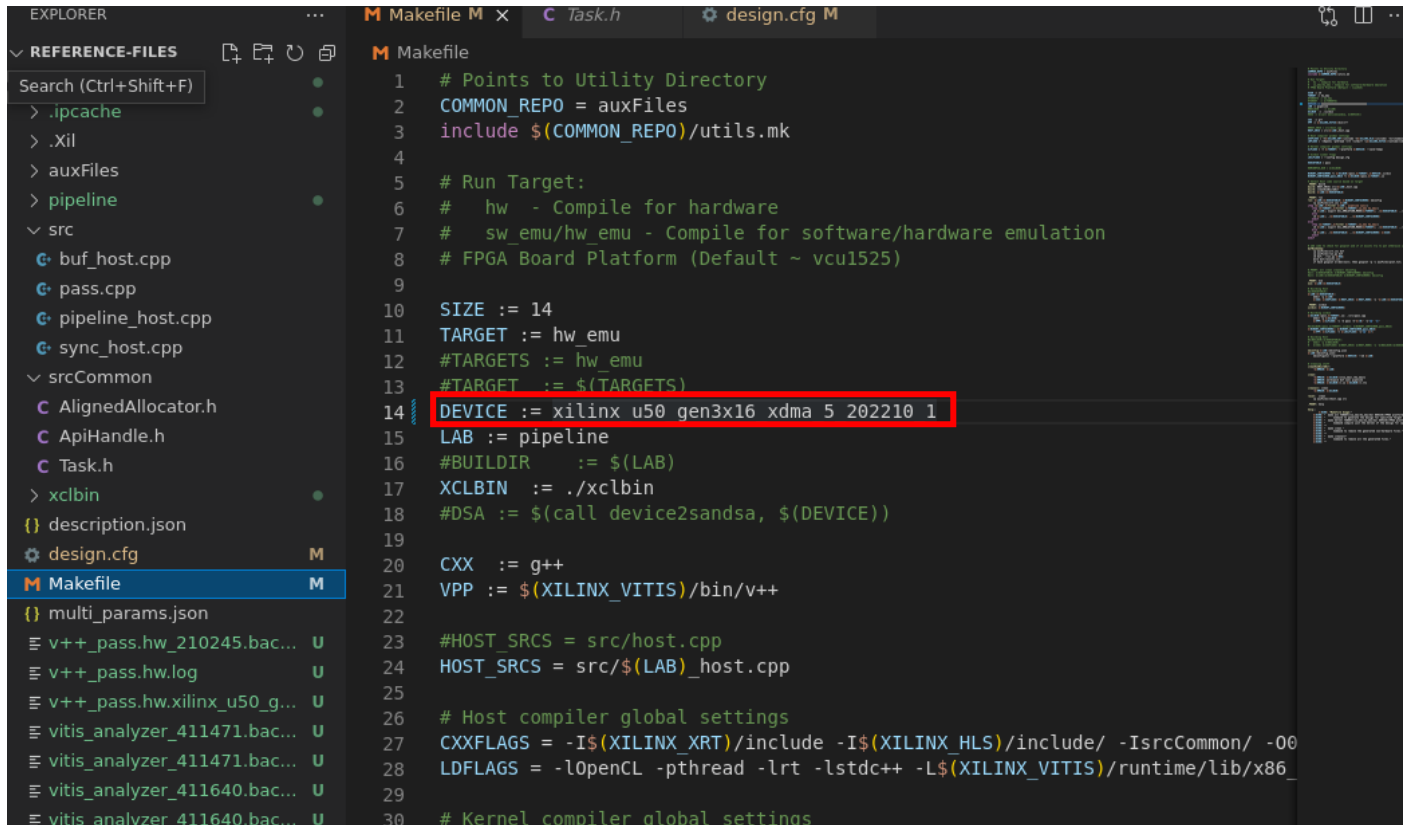


在這邊是設定 FPGA 的 global memory 的 m_inbuffer 跟 m_outBuffer 所要使用的 Memory。如下圖所示。



然後去 makefile 改 device。

因為原本這邊是 u200 這個板子，所以要改成 u50。因為這篇 code 原本是為了 u200 寫的。



```
1 # Points to Utility Directory
2 COMMON_REPO = auxFiles
3 include $(COMMON_REPO)/utils.mk
4
5 # Run Target:
6 # hw - Compile for hardware
7 # sw_emu/hw_emu - Compile for software/hardware emulation
8 # FPGA Board Platform (Default ~ vcu1525)
9
10 SIZE := 14
11 TARGET := hw_emu
12 #TARGETS := hw_emu
13 #TARGET := $(TARGETS)
14 DEVICE := xilinx u50 gen3x16 xdma 5 202210 1
15 LAB := pipeline
16 #BUILDIR := $(LAB)
17 XCLBIN := ./xclbin
18 #DSA := $(call device2sandsa, $(DEVICE))
19
20 CXX := g++
21 VPP := $(XILINX_VITIS)/bin/v++
22
23 #HOST_SRCS = src/host.cpp
24 HOST_SRCS = src/$(LAB)_host.cpp
25
26 # Host compiler global settings
27 CXXFLAGS = -I$(XILINX_XRT)/include -I$(XILINX_HLS)/include/ -IsrcCommon/ -O0
28 LDFLAGS = -lOpenCL -pthread -lrt -lstdc++ -L$(XILINX_VITIS)/runtime/lib/x86_
29
30 # Kernel compiler global settings
```

這樣一來，這篇 code 就可以用在 u50 上了。

接著編譯這份專案。

>make TARGET=hw DEVICE=xilinx_u50_gen3x16_xdma_5_202210_1 xclbin

這樣就完成了最初步的編譯步驟，那麼接下來是 LAB 的實作。

Lab1 pipeline

在這邊要做 LAB1 PIPELINE

編譯

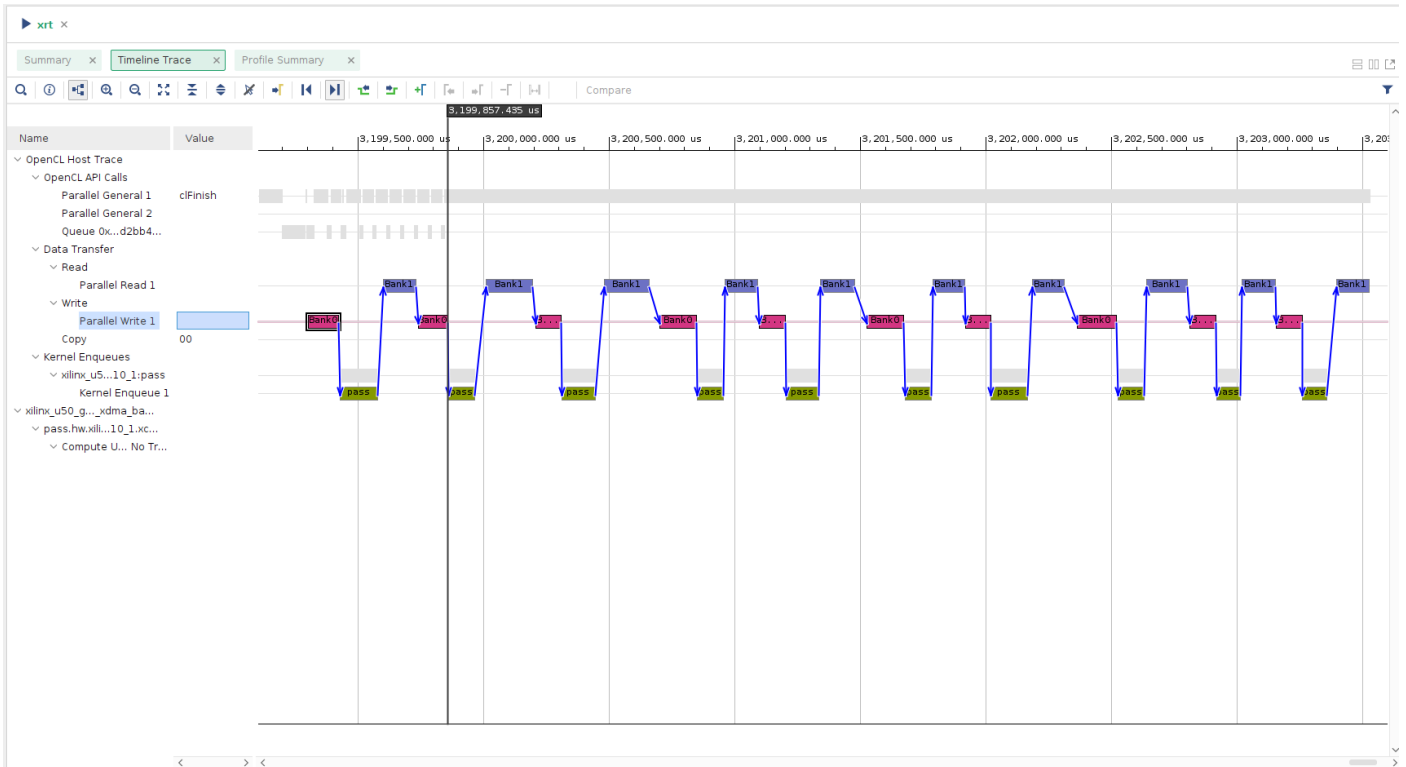
```
>make TARGET=hw DEVICE=xilinx_u50_gen3x16_xdma_5_202210_1 LAB=pipeline
```

然後跑 application :

```
>make run TARGET=hw DEVICE=xilinx_u50_gen3x16_xdma_5_202210_1 LAB=pipeline
```

然後看報告 :

```
>vitis_analyzer pipeline/xrt.run_summary
```

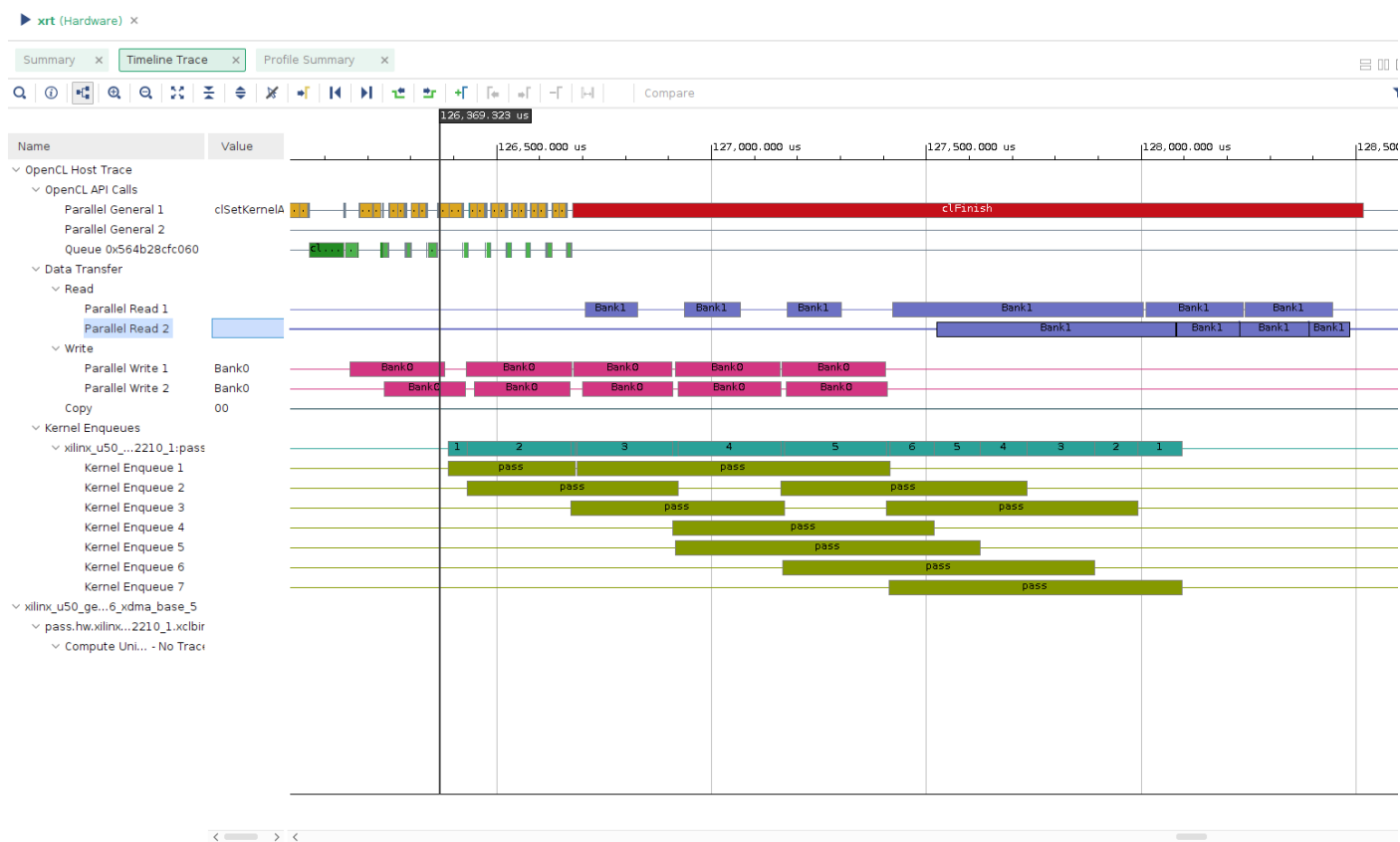


從這張圖中，可以很清楚地知道他是一個標準的 in order excution。前一個做完了才做下一個。

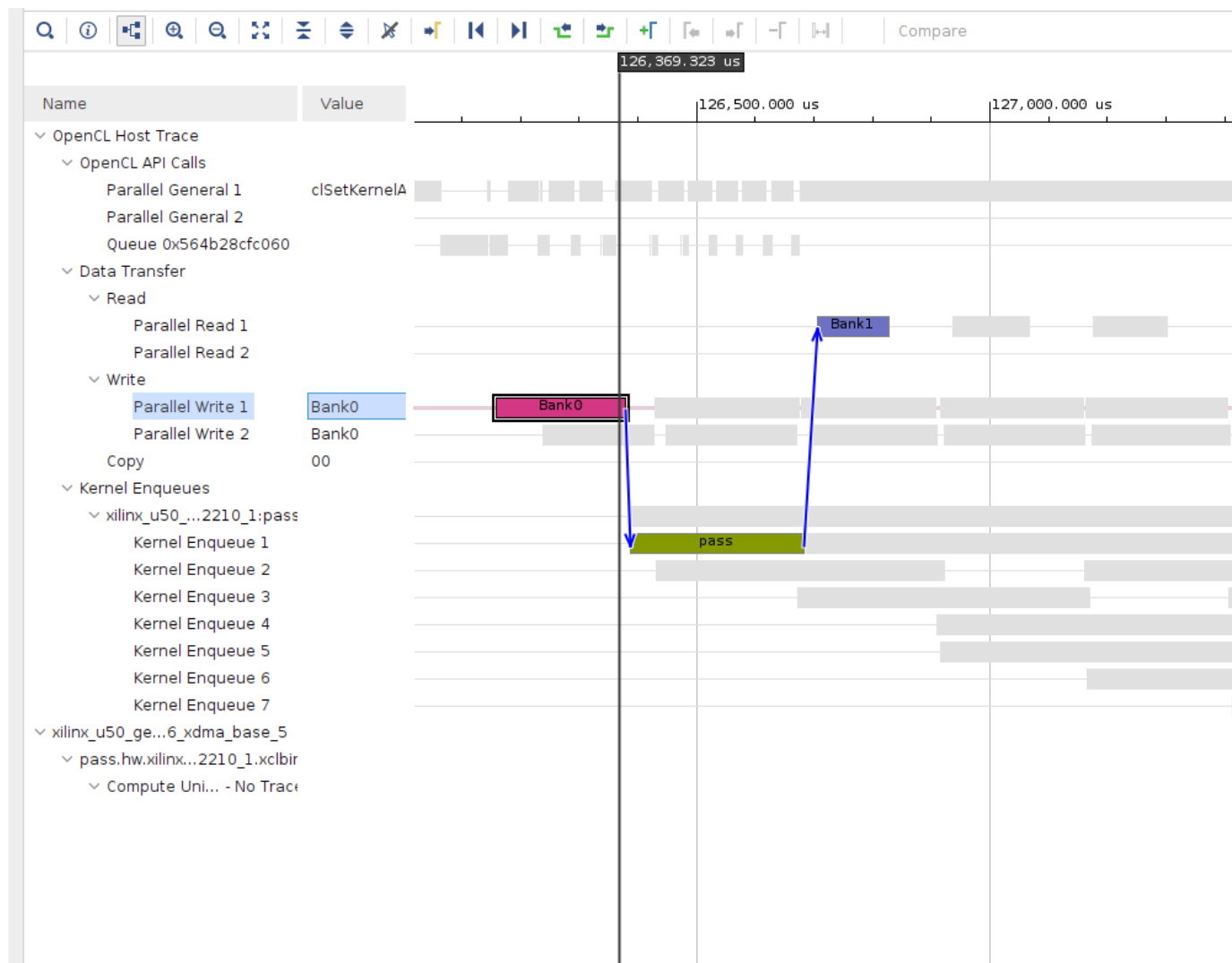
接下來在 src/pipeline_host.cpp。把 27 行的 oooQueue 從 false 改成 true

```
11
12 // -- Environment / Usage Check -----
13
14 char *xcl_mode = getenv("XCL_EMULATION_MODE");
15
16 if (argc != 2) {
17     printf("\nUsage: %s "
18           ".\xclbin/pass.<emulation_mode>.<dsa>.\xclbin ",
19           argv[0]);
20     return EXIT_FAILURE;
21 }
22 char*      binaryName  = argv[1];
23
24 // -- Common Parameters -----
25
26 unsigned int numBuffers      = 10;
27 bool oooQueue = true;
28 unsigned int processDelay    = 1;
29 unsigned int bufferSize      = 8 << 11;
30
31 // -- Setup
```

編譯之後 run 然後打開 Timeline trace



點其中一個 kernel 查看 dependency



設定 `oooQueue=True` 這件事情代表我們啟用了 out of order queue。

接下來將 `pipeline_host.cpp` 的第 26 行的 `numBuffers` 改成 20，得到如下的結果。



然後比較一下有沒有使用 `oooqueue` 的差別

```
Create Kernel: pass
Create Sequential Queue
Setup Complete

Total number of buffers: 10
  BufferSize: 16384
  Bits per Element: 512
  Bytes per Transfer: 1048576
  processDelay: 1
  Out of Order Queue: false

Running FPGA

Total data: 80 Mbits
  FPGA Time: 0.00525168 s
  FPGA Throughput: 15233.2 Mbits/s
  FPGA PCIe Throughput: 30466.5 Mbits/s

PASS: Simulation
04.HP0mGn@HLS04:~/m11061549/LabA/Vitis-Tutorials/Hardware_Acceleration/Design_Tutorials/07-h
ost-code-opt/reference-files$
```



```
Create Kernel: pass
Create Out of Order Queue
Setup Complete

Total number of buffers: 10
      BufferSize: 16384
      Bits per Element: 512
      Bytes per Transfer: 1048576
      processDelay: 1
      Out of Order Queue: true

Running FPGA

      Total data: 80 MBits
      FPGA Time: 0.00262055 s
      FPGA Throughput: 30527.9 MBits/s
      FPGA PCIe Throughput: 61055.8 MBits/s

PASS: Simulation
make: warning: Clock skew detected. Your build may be incomplete.
04.HP0mGn@HLS04:~/m111061549/LabA/Vitis-Tutorials/Hardware_Acceleration/Design_Tutorials/07-h
ost-code-opt/reference-files$
```

從這邊的報告可以發現，在啟用 oooqueue 之前，需要跑 0.0052s。啟用之後只要 0.0026s。

比較與討論：

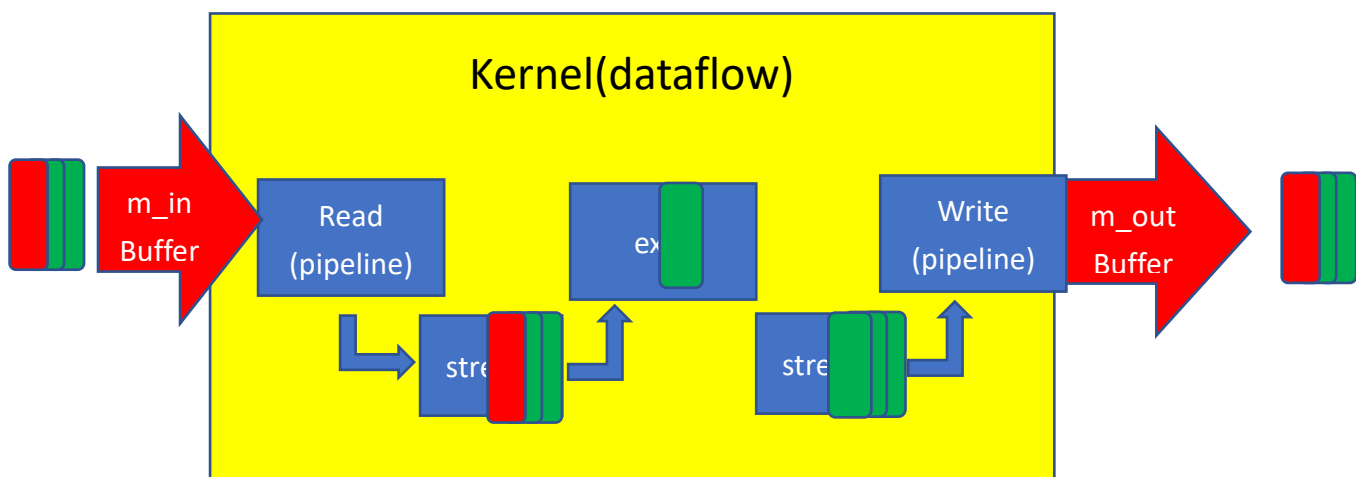
如上面的圖表我們可以知道，將 out of order queue 給 enable 從整個系統執行的角度來看，可以提高他的 throughput，並降低總共的執行時間。因為 inorder 會等資料先讀進 FPGA 讀完之後讀回 host，這一個流程完畢之後才從 host 送下一筆資料。所以每一次的 operation 除了 kernel 的執行時間之外，從 host 送到 FPGA 所需的 PCIE transfer time 也會浪費一堆時間。這段時間 kernel 是 idle 的。所以 throughput 會降低，utilization 也不高。使用 out of order 之後就沒這問題，host 有資料就直接寫入，FPGA 有結果就直接讀回來。這樣的話就可以一邊 pcie 傳資料一邊讀回資料。

如同前面所提到的，我有觀察到一個蠻有意思的狀況，就是當啟用 out of order queue 的時候，kernel(就是 pass)的執行時間不但變長。但是從尚未啟用 out of order queue 的分析圖表來看，其實 kernel 執行的時間所需非常短。為甚麼呢？我認為這跟我前面提到的 dataflow 有關。

我的推測如下：當不管執行順序的 order 之後，變成 host 資料就是有資料就一直送到 FPGA 的 buffer 裡面，但是 read exec write 三個 block 的執行時間可能不一樣(特別是 exec 需要讀資料然後再做加法，且沒有 Pipeline 優化，至少 exec 應該會慢於 Read)，然而所謂的 kernel 執行的時間(應該)是 host 讀進來之後開始，經過 read+exec+write 直到讀回 host 的總和時間，。所以當資料源源不斷輸入 Kernel 時，來不及被 exec 處理的資料就會被堆在 stream 之間。

也有可能是因為把 m_inBuffer 讀進 Read block 和讀出 m_out Buffer 這兩件事情需要花時間，畢竟讀寫 memory 本來就是很花時間的事情，所以從 host 端來的資料就堆在 m_inBuffer，來不及讀出去的資料則會被堆在 stream 裡面。

在 LAB1 當中我額外提高了 numBuffer 的大小，可以發現當我提高之後 kernel 的執行時間又拉得更長了。雖然不能確定原因，但我確定一定是有資料塞在裡面。



Lab2

Kernel and Host Synchronization

在 hostcode 裡面。有這樣一段的 code：

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

這是被稱為 free-running pipeline 的東西。

Name	Stalled Pipeline (HLS default)	Free running Pipeline	Traditional flushable pipeline
Pragma/Directive	#HLS pragma pipeline style= stp set_directive_pipeline -style stp	#HLS pragma pipeline style= frp set_directive_pipeline -style frp	#HLS pragma pipeline style= flp set_directive_pipeline -style flp Deprecated: #HLS pragma pipeline enable_flush
Global default	config_compile -pipeline_style stp (default)	config_compile -pipeline_style frp config_compile -frp_max_fanout \$value (default is 2000)	N/A
Description	Stalling architecture: runs only when input data is available otherwise stalls	Flushable frequency-optimized architecture: runs even when input data is not available	Traditional flushable architecture.
Advantages	<ul style="list-style-type: none">No usage constraintLeast overall resource usage (typically, but not always)	<ul style="list-style-type: none">Better timing due to:<ul style="list-style-type: none">Less fanoutSimpler pipeline control logicFlushable	Flushable
Disadvantages	<ul style="list-style-type: none">Not flushable, hence it might:<ul style="list-style-type: none">Cause more deadlocks in dataflowPrevent already computed	<ul style="list-style-type: none">Moderate resource increase due to FIFOs added on outputsUsage constraints:<ul style="list-style-type: none">Mainly used for dataflow	<ul style="list-style-type: none">May increase IIResource increase due to less sharing (II>1)Usage constraints:

也就是說並沒有 synchronization。即使沒有 input data 也會跑。是 frequency-optimized 的架構。雖然說這樣可以產生很高效的 pipeline，但是在 buffer allocation 和執行順序上會有一些問題。這是因為 buffer 只在他們不被需要時才會被 flush。

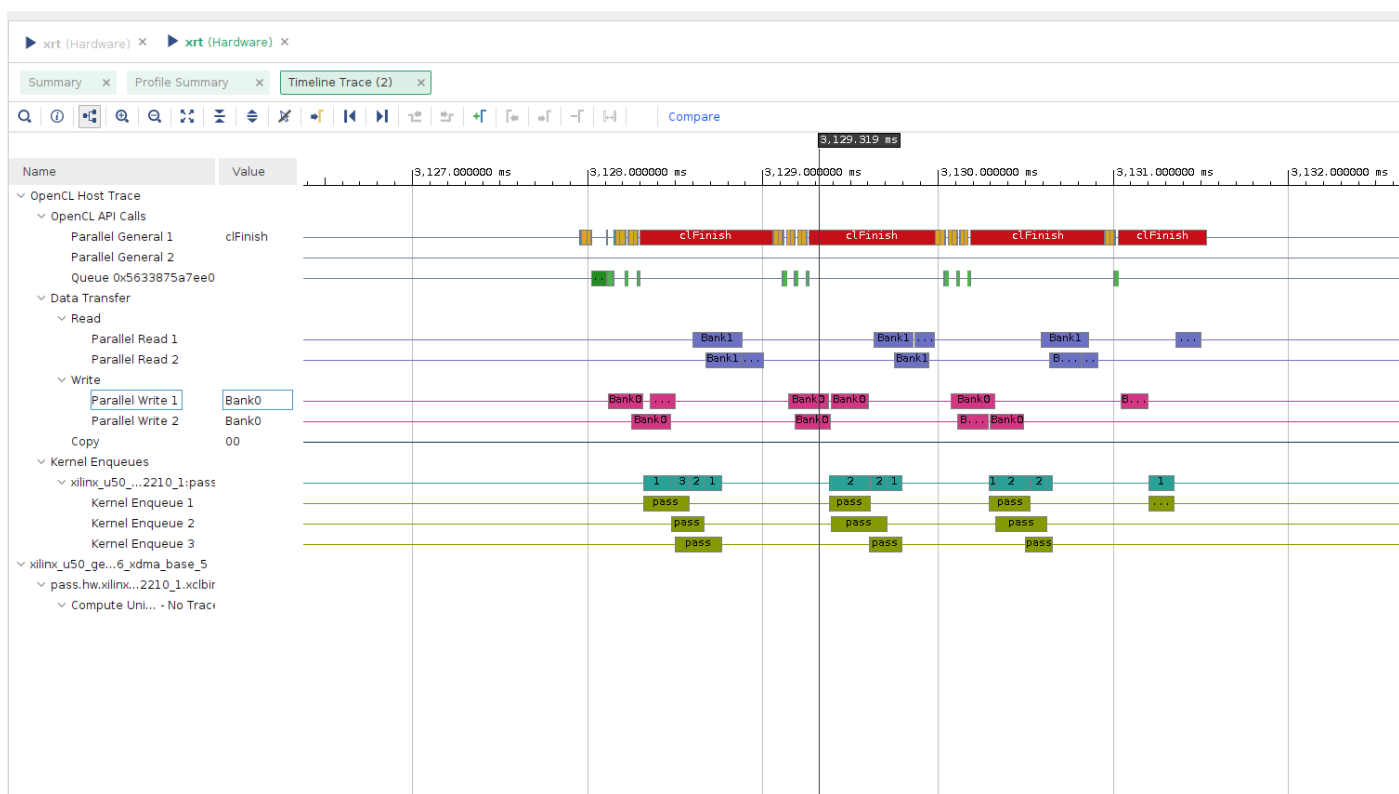
在上一個 lab，我將 numbuffer 調到 20 也就是說要跑 20 次 kernel function。我發現的情況是 enqueue 會越來越多，只要有資料，input 就會不斷 read 進來，而且是有空位就會放。那麼這衍伸了甚麼問題呢？就是假設 numbuffer 調到很大，FPGA 的 memory 可能會被用光。再來就是他是有空位就放(out of order queue)，雖然單純在速度上看，他執行大量的 kernel function 很快，但是它就不會是按照順序執行。

這個 lab 就是要解決這個問題：如果我需要有序執行，但卻想要啟用 oooqueue 來加速度的話要怎麼辦？

首先將 sync 的 host code 做修改(src/sync_host.cpp)。

```
// -- 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());
```

然後編譯並跑專案，然後打開 analyer 觀察



這個方法是在 host 端每執行三次的 task 就會做一次 clFinish。

這個 clFinish 就會做為一個同步點，他所造成的效益就是當 clFinish 發生開始到下一個 command 進入 queue 中間這段時間，host 端不會再增加新的 command 進入 queue。clFinish 結束之前，在 command queue 內的所有 command 都會跑完，包含 data transfer 的部分。

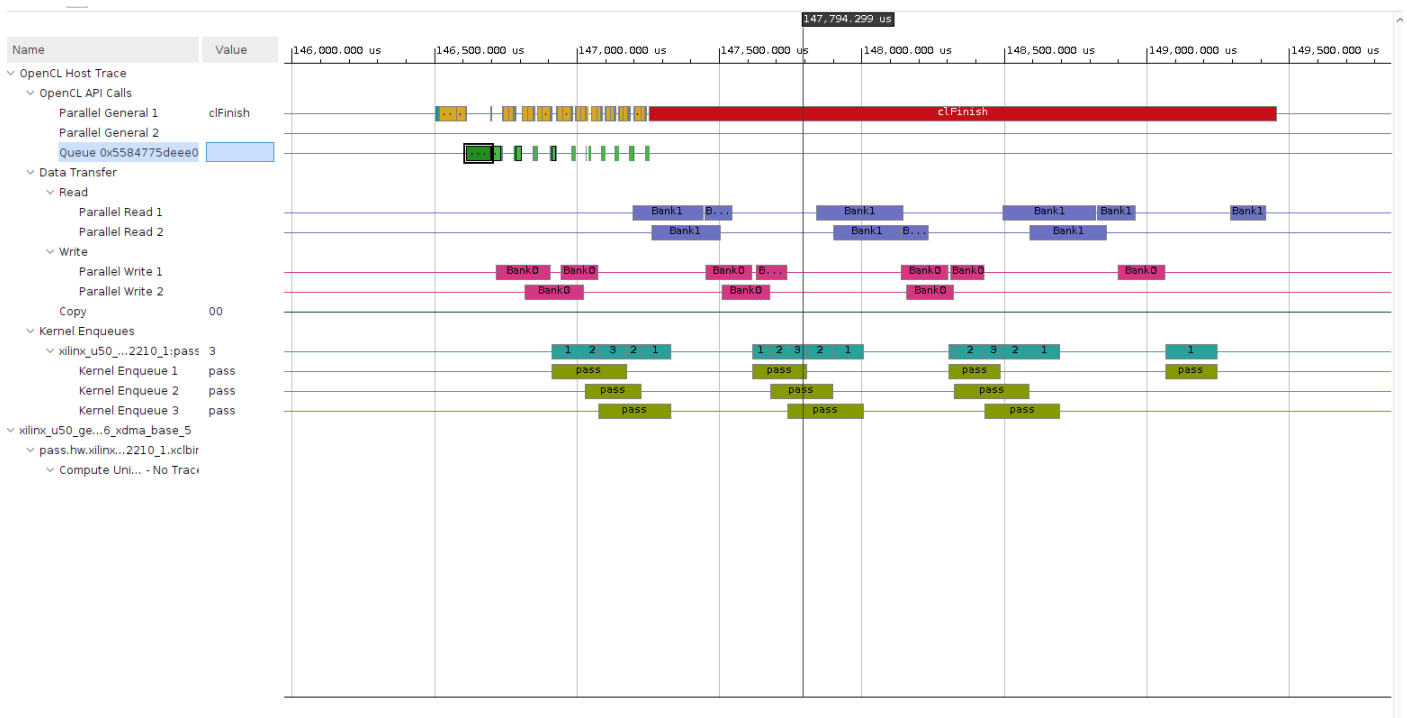
從 time line 上看，這樣做確實讓 task 有 in order 地執行，但是可以發現每三個 kernel 執行之間的 gap 其實蠻大的。當 kernel 全部執行完到下一筆資料讀進來這之間完全不會有動作，因為 clFinish 的強制性。這算是一個可以接受的辦法，但並非最上解。

我們將 host code 做如下更改

```
// -- 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());
```



這個方法是當執行到第三個 task 之後，在 run 裡面傳入 getDoneEv() 這個東西。

按照這篇 code 所提供的 API 來看，getDoneEv() 其實就是讓整個 task 回傳一個 done 的 event。

如下所示：

```
public:
    cl_event* getDoneEv() { return &m_doneEv; }
```

然後在 task 的流程控制中有這樣一段 code：

srcCommon/ApiHandle.h

```

//sent data(&m_inBuffer[0]) from host to device(50)
// host->u50
if(prevEvent != nullptr) {
    clEnqueueMigrateMemObjects(api.getQueue(), 1, &m_inBuffer[0],
    | 0, 1, prevEvent, &m_inEv);
} else {
    clEnqueueMigrateMemObjects(api.getQueue(), 1, &m_inBuffer[0],
    | 0, 0, nullptr, &m_inEv);
}

```

這個地方是執行 task 之前，我要將 m_in m_out 的資料搬到 fpga 的 global memory 上面。也是 kernel execution time 開始計算的地方(應該是)。也就是 host-->FPGA。

在 lab2 的第一個辦法(以及 lab1)，prevEvent 都是=nullptr，也就是我在 lab1 所說的：hostcode 會不斷地塞資料進去到 FPGA 上。那當我在 run 的時候傳入 getDoneEv() 的時候，就是強迫從第四個 task 開始，要接到 m_doneEv 這個 Event 才能開始搬下一個資料。然而

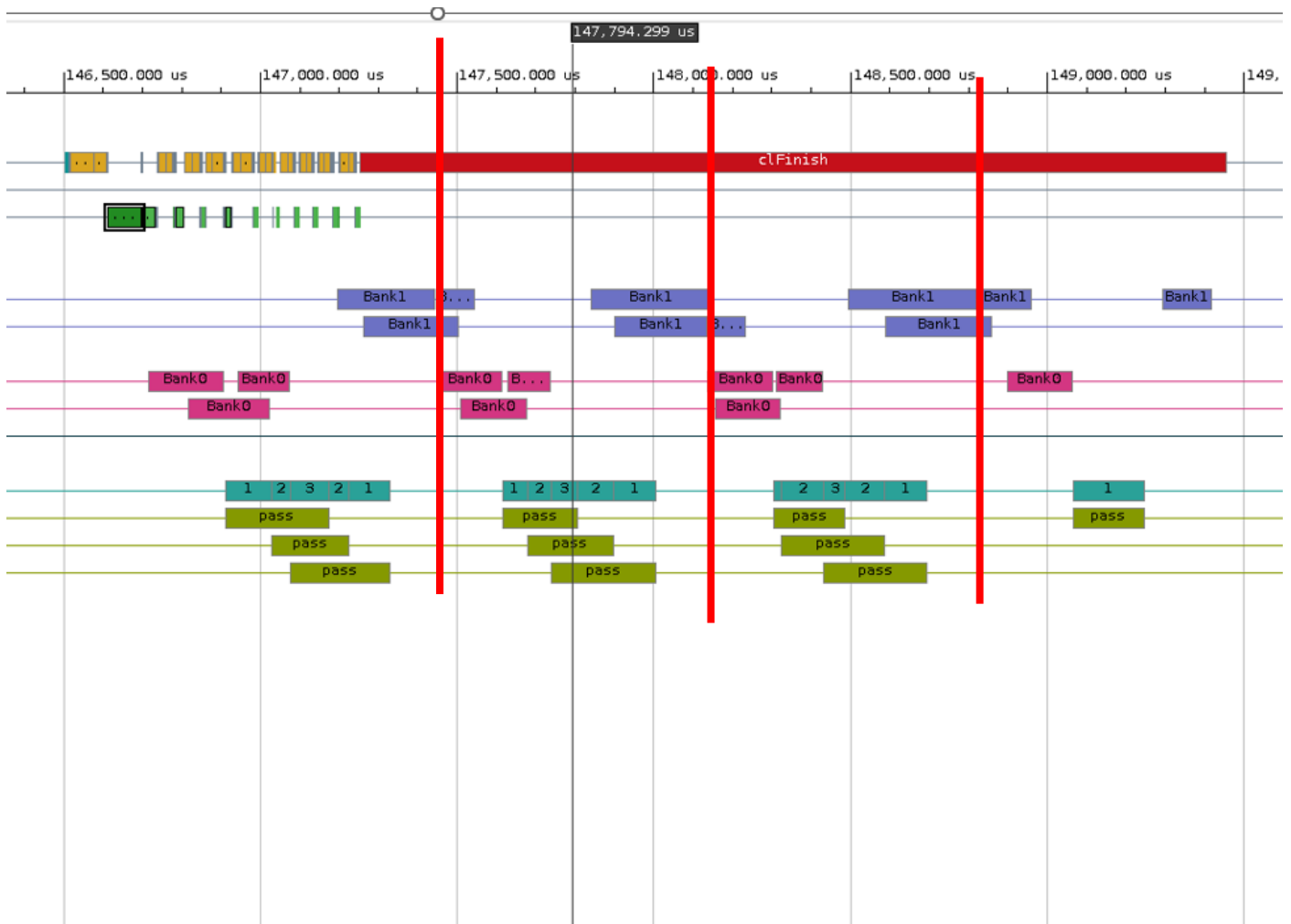
```

//host receive data(&m_outBuffer[0]) from device(50)
//host<-u50
clEnqueueMigrateMemObjects(api.getQueue(), 1, &m_outBuffer[0],
|  |  |  | CL_MIGRATE_MEM_OBJECT_HOST,
|  |  |  | 1, &m_outEv, &m_doneEv);
m_hasRun = true;
}
bool outputOk() {

```

這個步驟是 task 的最後步驟，也就是傳算好的資料回 host 的步驟。只有這個步驟會產生 m_doneEv。由於 host 下 command 的速度遠快於傳送 PCIE 的速度，所以可以想成這樣：

從第四個 task 執行開始，都必須等到自己往前數三個的 task 結束，資料傳回來才能開始搬資料到 kernel 運算，地 4 個等第 1 個，第 5 個等第 2 個，第 i 個等第 i-3 個 task 結束。我們可以再來仔細地觀察這張 timeline



除了最右邊之外，前面兩個紅色線由左至右分別代表第一個 task 的結束跟第 4 個 task 的開始，以及第四個 task 的結束跟第 7 個 task 的開始。他們都剛好銜接在一起。也非常明顯地，每三個 kernel 之間的 gap 也縮短，這不但提高了 utilization，也縮短了總時間。

接著看一下兩者的執行時間

前者：

```
Create out of order queue
Setup Complete

Total number of buffers: 10
  BufferSize: 16384
  Bits per Element: 512
  Bytes per Transfer: 1048576
  processDelay: 1
  Out of Order Queue: true

Running FPGA

Total data: 80 MBits
  FPGA Time: 0.00368019 s
  FPGA Throughput: 21738 MBits/s
  FPGA PCIe Throughput: 43476 MBits/s

PASS: Simulation
04_HP0mGp@HL S04:~/m111061549/LabA/Vitis-Tutorials/H
```

後者

```
Create Kernel: pass
Create Out of Order Queue
Setup Complete

Total number of buffers: 10
      BufferSize: 16384
      Bits per Element: 512
      Bytes per Transfer: 1048576
      processDelay: 1
      Out of Order Queue: true

Running FPGA

      Total data: 80 Mbits
      FPGA Time: 0.0030106 s
      FPGA Throughput: 26572.7 Mbits/s
      FPGA PCIe Throughput: 53145.5 Mbits/s
```

前者的方法跑了 0.0036s 後者則是 0.0030s。後者比較快。

問題與討論

這個 lab 的概念其實就是在改善 lab1 oooqueue 所可能帶來的問題：

1. Out of memory issue

在 lab1 用了蠻大的篇幅討論為甚麼 kernel 執行時間會那麼長可能是因為資料塞在 buffer 或是 stream 裡面。如果 numBuffer(執行的 task 數量)拉到很大的話，那極度有可能發生 out of memory 的狀況。

2. Out of order execution

其次的問題是 lab1 只是把效率最大最大化，但並未考慮執行順序。萬一我的 task 是要承接前一次 task 的輸出來做為輸入的話，這麼做就會出錯。

其實可以發現並沒有比 lab1 的第二個快，反而慢了一點。我認為這是一個 trade off。我需要利用 synchronization 來讓資料有序執行，已得到好的效果，只是需要犧牲一些執行時間。

Lab3

首先研究一下他的 hostcode

src/buf_host.cpp

在做 synchronization 的前提下，將 numBuffers(也就是 task 的總執行次數)加到 100。

```
unsigned int numBuffers      = 100;
bool        oooQueue        = false;
unsigned int processDelay    = 1;
unsigned int bufferSize      = 1 << atoi(argv[2]);
```

```
// -- 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());
```

由於這篇 code 是寫給 U200 跑的，所以需要做一些修改：

1. makefile 需要改，第 18 行的 DSA 原本是被註解的，需要把它取消註解。

```
14  DEVICE := xilinx_u50_gen3x16_xdma_5_202210_1
15  LAB := pipeline
16  #BUILDIR := $(LAB)
17  XCLBIN := ./xclbin
18  DSA := $(call device2sandsa, $(DEVICE))
19
```

並且確認這邊的內容。

```
66
67  # add code to check for gnuplot and if it exists try to pot otherwise just l
68  bufRunSweep:
69      cp auxFiles/xrt.ini buf
70      cp auxFiles/run.py buf
71      cd buf; ./run.py $(DSA)
72      more buf/results.csv
73      if hash gnuplot 2>/dev/null; then gnuplot -p -c auxFiles/plot.txt; fi;
74
```

2. 然後需要修改一些設定檔

auxFiles/plot.txt 的內容需要做修改：

第八行的路徑需要是 buf/results.csv，原本的 code 並不是。

```

auxFiles > ≡ plot.txt
1  set key autotitle columnhead
2  set logscale x 2
3  set style line 1 \
4  |   linecolor rgb '#0060ad' \
5  |   linetype 1 linewidth 2 \
6  |   pointtype 7 pointsize 1.5
7
8  plot "buf/results.csv" with linespoints linestyle 1
9

```

3. 再來是 auxFiles/run.py

24 行的 range(8, 20)改成(8, 15)。如果不這麼做會出現

[XRT]error: bad_alloc 報錯。推測是 U50 記憶體被吃光了。

```

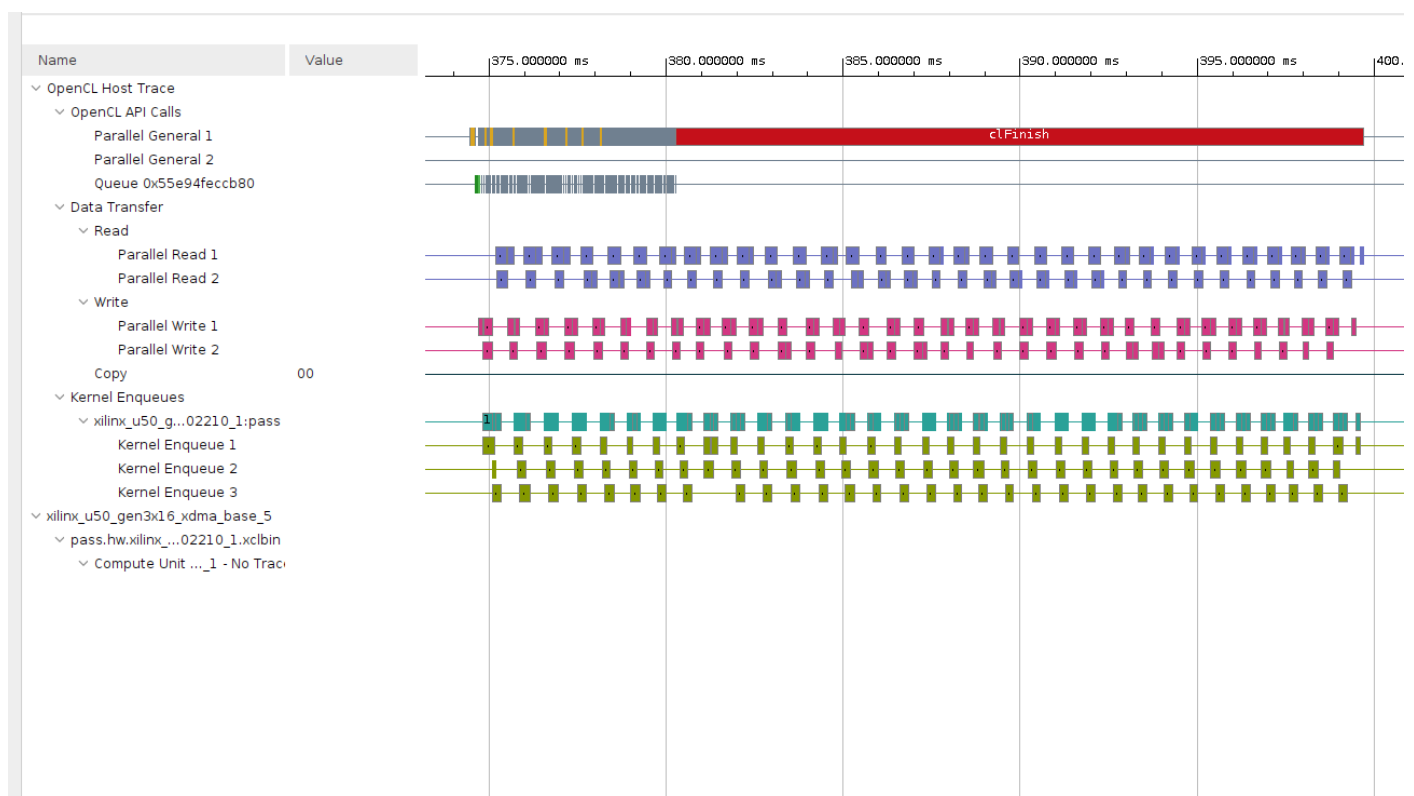
23
24  for i in range(8, 15):
25      fields = {}
26      buffersize = 1 << i

```

接下來就可以跑了。

> make TARGET=hw DEVICE=xilinx_u50_gen3x16_xdma_5_202210_1 SIZE=14 LAB=buf

> make run TARGET=hw DEVICE=xilinx_u50_gen3x16_xdma_5_202210_1 SIZE=14 LAB=buf



```
Total number of buffers: 100
      BufferSize: 16384
      Bits per Element: 512
      Bytes per Transfer: 1048576
      processDelay: 1
      Out of Order Queue: true

Running FPGA

      Total data: 800 Mbits
      FPGA Time: 0.0252285 s
      FPGA Throughput: 31710.2 Mbits/s
      FPGA PCIe Throughput: 63420.5 Mbits/s
```

和 lab2 第二個方法比較

```
Setup Complete

Total number of buffers: 10
      BufferSize: 16384
      Bits per Element: 512
      Bytes per Transfer: 1048576
      processDelay: 1
      Out of Order Queue: true

Running FPGA

      Total data: 80 Mbits
      FPGA Time: 0.0030106 s
      FPGA Throughput: 26572.7 Mbits/s
      FPGA PCIe Throughput: 53145.5 Mbits/s
```

可以發現我提到 numbuffer 之後，其實 FPGA 和 PCIE 的 throughput 都提升了。

這件事情很好證明了就是當 host 傳的東西越大，我的 throughput 是會上升的。也就是說 FPGA 的資源有被更有效率的使用，使其 utilization 上升。

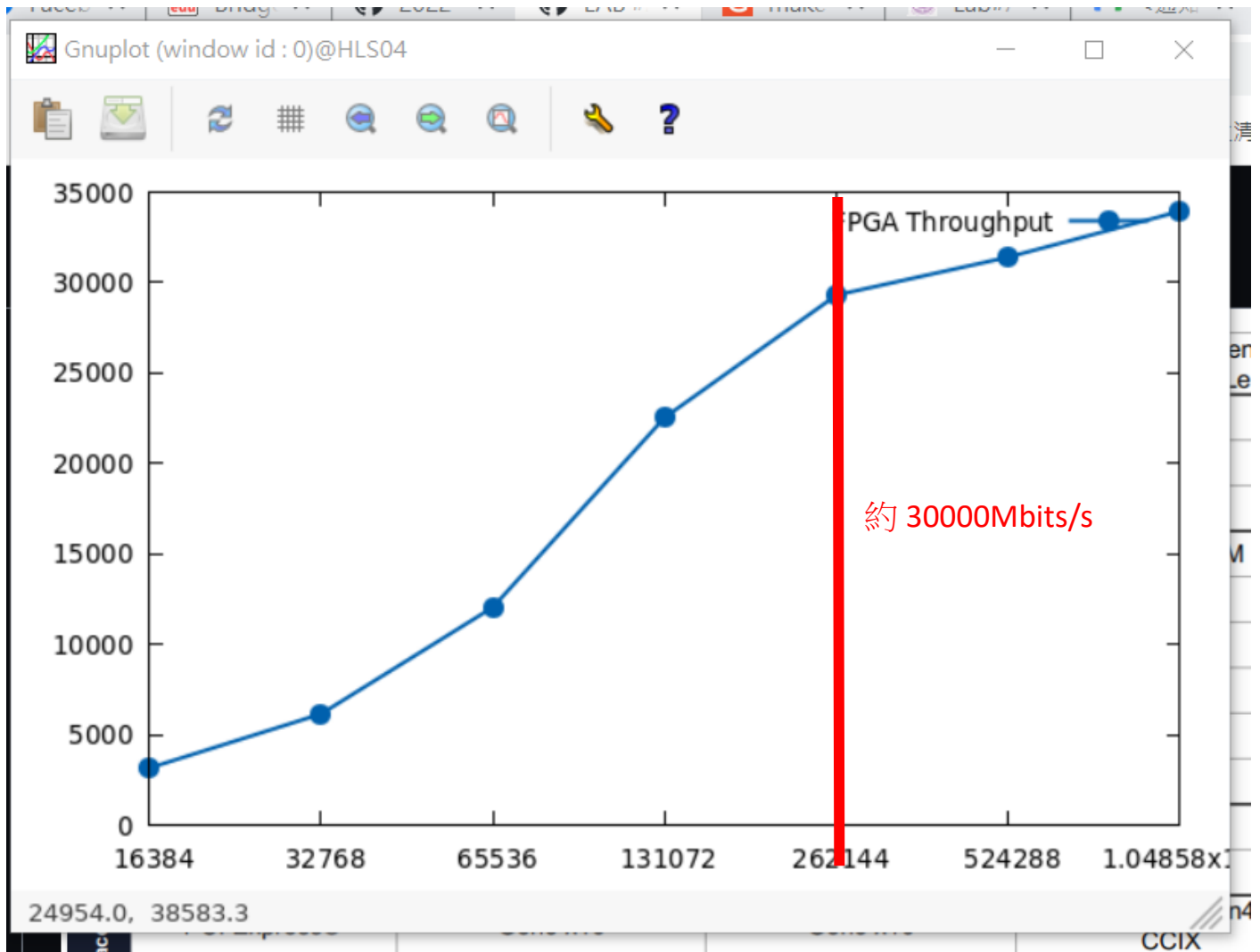
但事實總是如此嗎？

跑這段 code

```
>make TARGET=hw DEVICE=xilinx_u50_gen3x16_xdma_5_202210_1 bufRunSweep
```

這邊其實是在將 buffer size 不斷加大，每次兩倍。看 numbuffer 和 throughput 的關係。

4 然後就會有如下的圖跑出來



其中，縱軸為 Bytes per Transfer 縱軸為 FPGA Throughput。

從這邊可以知道，buffer 逐漸增加，我們希望 throughput 也會線性上升，但實際上 throughput 會逐漸到一個飽和，過了一個點之後它的上升曲線會趨緩。以這邊為例，FPGA 的最高 throughput 大概在 34000Mbits/s 左右而大概在 30000 開始上升時間就會趨緩。

結論：

結語

雖然這堂課是應用加速，但是由於 xrt 系統的 pcie bandwidth 限制，使得資料從 host 到 fpga 的傳輸也變成一個需要考慮問題。當在開發一個 Kernel 應用的時候，如果這個 Kernel 需要被不斷地呼叫的話，那麼就需要去 overlap host to FPGA 的 data transfer 和 kernel 的執行時間。這樣才能達到更好的 utilization 和 throughput。總之在 xrt 上做應用加速的開發，需要不斷反覆且有序執行 Kernel 時 host 端的優化也會是需要考慮的問題。

我學到了甚麼：

如果要搞懂這個 host code 怎麼做 optimization，需要把整個 hostcode 和 kernel code 翻個底朝天，從頭到尾 trace 一遍知道每一個動作在幹嘛，因為 tutorial 其實甚麼也沒講。所以要一邊對著 timeline 一邊一行一行 code 看才知道到底哪一個步驟是關鍵。我認為這個主題是個非常好的教材，多虧了這個教材，我幾乎快要把 hostcode 給摸熟了。

參考資料：

Github 教學

https://github.com/bol-edu/2022-fall-ntu/tree/main/LabA/Host_Code_Optimization

課本

<https://www.boledu.org/textbooks/hls-textbook/application-acceleration-development-flow/host-application-development---opencl>

github discussion

<https://github.com/bol-edu/HLS-Discussions/discussions>

台大上課影片：

https://www.youtube.com/watch?v=qQT9kRYt_yg&list=PL5CoDA0gt0HWDIbsYu0j10_ocqetwU1WF&index=11

去年台大的 LABA ch6 報告影片

https://youtu.be/yNhj550uoZU?list=PL5CoDA0gt0HWDIbsYu0j10_ocqetwU1WF&t=4430

U50 資料手冊

https://www.xilinx.com/content/dam/xilinx/support/documents/data_sheets/ds965-u50.pdf

OpenCL reference pages

<https://registry.khronos.org/OpenCL/sdk/1.2/docs/man/xhtml/>

Github 連結：<https://github.com/s095339/LabA-host-code-optimization>