

LabB-Convolution_Filtering

111061549 張耀明

1. 本實驗的目的：

Host program：使用 host code optimization 消除 host 和 FPGA 之間的傳輸代價。

Kernel：介紹如何輕鬆地評估用 Vitis HLS 建置的硬體 kernel 的 performance，並且顯示這個評估的準確性和能多接近真實的硬體表現。

2. 實驗介紹：

本實驗共有 N 個 lab。旨在介紹使用 Xilinx Alveo Data Center 加速卡來加速的密集運算 (compute-intensive) 應用。其中

PRELAB：在 Xilinx Alveo Data Center 裡面實現 Video Filter。

LAB1：在設計之前事情評估所需的效能，並構想硬體優化方法

LAB2: 實作硬體並以 vitis-hls 分析。

LAB3 加入 hostcode，實現整體的系統。

3. 事前設定：

租借好板子之後，將 repo clone 到自己的資料夾下：

➤ **git clone** <https://github.com/Xilinx/Vitis-Tutorials.git>

接著輸入以下指令：

➤ **cd** /clone 到的資料夾/Vitis-Tutorials/Hardware_Acceleration/Design_Tutorials/01-convolution-tutorial

➤ **wget** https://www.xilinx.com/bin/public/openDownload?filename=conv_tutorial_files.tar.gz -O conv_tutorial_files.tar.gz

➤ **tar -xvzf** conv_tutorial_files.tar.gz

然後處理環境變數：

➤ **source** /opt/Xilinx/Vitis/2022.1/settings64.sh

➤ **source** "opt/xilinx/xrt/setup.sh"

這樣就完成了環境設定。

接下來輸入指令

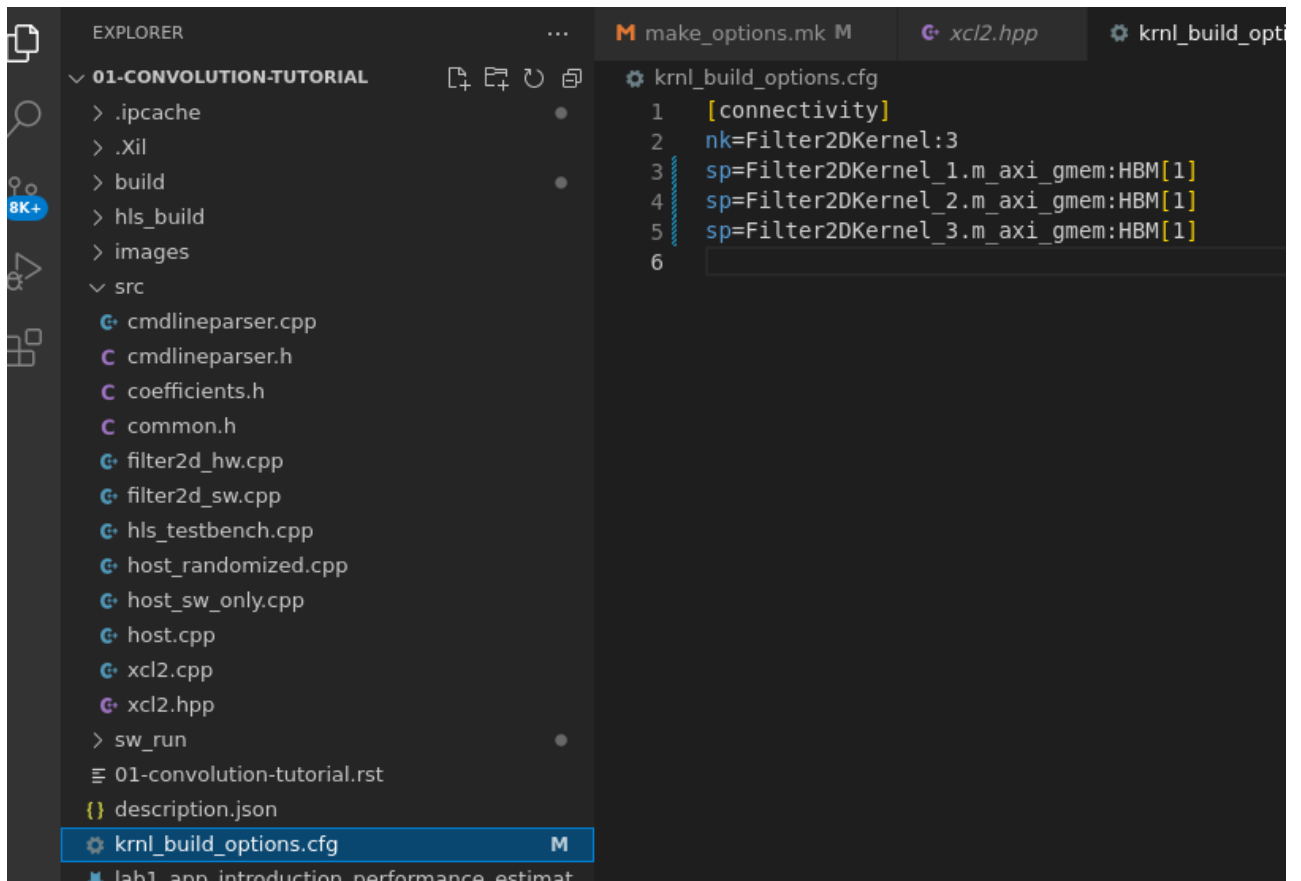
➤ **cd** Vitis-Tutorials/Hardware_Acceleration/Design_Tutorials/01-convolution-tutorial/

➤ **code** .

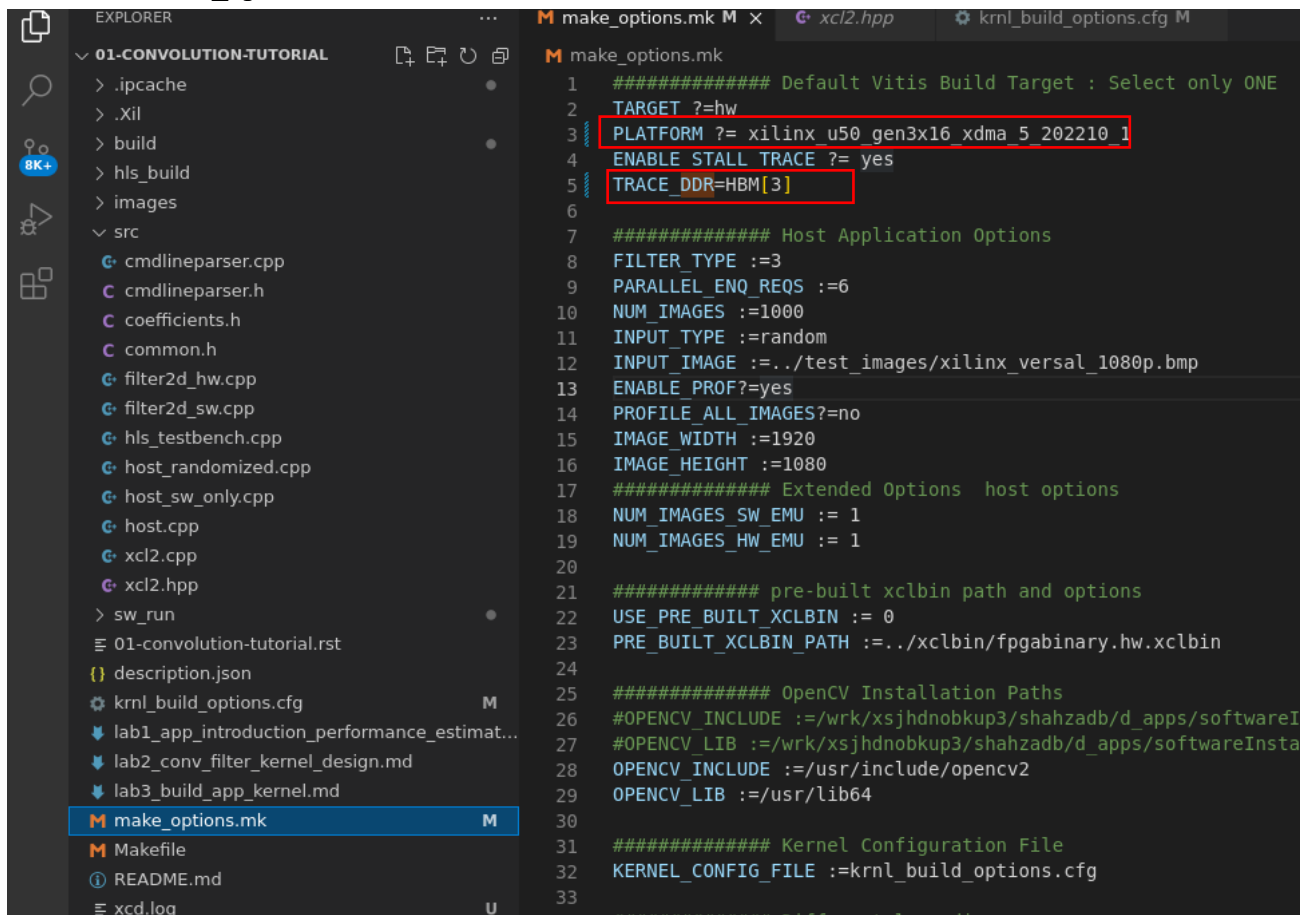
一樣地，這篇 code 也是寫給 U200 的板子，所以需要改一些設定檔和 makefile 以便編譯順利進行。

(1) 更改記憶體分配

打開一個 `krnl_build_options.cfg` 檔，這邊定義了 FPGA 上的 buffer 要分配在哪個 memory 上。將裡面所有的 `DDR[1]` 改成 `HBM[1]`。



然後來到 make_options.mk



確保框起來的地方如圖所示，若不是則修改。

到這邊為止，改好了不同板子的 memory allocate、板子型號的更改。

接下來要改的是針對環境的改動。

4. 環境修改、Hostcode.cpp 修改、Makefile 修改：

在前面改好板子設定之後，需要修改這份專案的 code 的內容(host.cpp)

code 使用的是舊版的 opencv library 是使用 openCV2.4。但是 boledduser 伺服器端的 linux 系統所使用的 opencv library 是 opencv4，總而言之，因為環境的問題，這篇專案的 host.cpp **是沒辦法運作的**。

Host.cpp 可以真的把一張圖片丟進去處理，然後得到處理完的圖片，如果想要看到圖片結果，就需要 follow 下面的過程，不然其實可以跳過。

- 1 首先，在專案編譯的過程中，C++ hostcode 所需要的 opencv library 的 headerfile 需要在這邊被指定，make_option.mk 裡面，確保這邊是這樣(限於 boledduser)：

```
##### OpenCV Installation Paths
#OPENCV_INCLUDE :=/wrk/xsjhdnobjkup3/shahzadb/d_apps/softwareInstall/anacondaInstall5aug20/envs/opencv2.4/include
#OPENCV_LIB :=/wrk/xsjhdnobjkup3/shahzadb/d_apps/softwareInstall/anacondaInstall5aug20/envs/opencv2.4/lib
OPENCV_INCLUDE :=/usr/local/include/opencv4/
OPENCV_LIB :=/usr/local/lib/
```

然後在 INPUT_IMAGE 選擇一張圖片

"/mnt/HLSNAS/04.HPOmGn/m111061549/LabB_github/Vitis-

Tutorials/Hardware_Acceleration/Design_Tutorials/01-convolution-tutorial/images/inputImage50.jpg"

```
##### Host Application Options
FILTER_TYPE :=3
PARALLEL_ENQ_REQS :=6
NUM_IMAGES :=1000
INPUT_TYPE :=""
INPUT_IMAGE := "/mnt/HLSNAS/04.HPOmGn/m111061549/LabB_github/Vitis-Tutorials/Hardware_Acceleration/Design_Tutorials/01-convolution-tutorial/images/in
ENABLE_PROF?=yes
PROFILE_ALL_IMAGES?=no
IMAGE_WIDTH :=1920
IMAGE_HEIGHT :=1080
##### Extended Options  host options
NUM_IMAGES_SW_EMU := 1
```

2.Host.cpp

(1)裡面有不少函數還在使用 C API，如果只是#include opencv2/opencv.hpp，只能調用裡面對應的 C++ API，所以需要在裡面多 include 其他.h file 確保內部的 C API code 可以被使用。

```
28 #include "coefficients.h"
29 #include "common.h"
30 //-----
31 #include "opencv2/opencv.hpp"
32 #include "opencv2/core/core_c.h"
33 #include "opencv2/videoio/legacy/constants_c.h"
34 #include "opencv2/highgui/highgui_c.h"
35 #include "opencv2/core/core.hpp"
36 #include "opencv2/highgui/highgui.hpp"
37 #include "opencv2/highgui/highgui_c.h"
38 #include "opencv2/videoio.hpp"
39 #include "opencv2/imgproc/imgproc_c.h"
40 #include "opencv2/imgproc/imgproc.hpp"
41 #include "opencv2/imgcodecs/imgcodecs.hpp"
42 #include <opencv2/imgcodecs.hpp>
43 #include "opencv2/highgui.hpp"
44 #include "opencv2/imgproc.hpp"
45 #include "opencv2/imgcodecs.hpp"
46 #include "opencv2/imgproc/types_c.h"
47 #include "opencv2/imgproc/imgproc_c.h"
48 #include "opencv2/core/core_c.h"
49 #include "opencv2/imgcodecs/legacy/constants_c.h"
50 //-----
```

上面做完之後，會發現有兩個 C API 的函數依然不能被使用，我從 header file 裡面確認在新版的 opencv 中，此二功能不見了。分別是 `csSaveImage` 和 `csLoadImage`，他們其實就是 `imwrite` 跟 `imread` 的 C API 版本。

```
84 - // Write to disk
85 - cvSaveImage(filename.c_str(), dst);

298 332
299 333 // Read Input image
300 - IplImage *src;
301 - src = cvLoadImage(srcFileName.c_str()); //format is BGR
```

我在這邊將其改為使用 `imwrite` 和 `imread` 函數。

但由於原本的函數所使用的 datatype 和 `imwrite` `imread` 使用的 datatype 不一樣，所以需要修改 hostcode：

Imwrite：

```
92
93 static void writeRawImage(
94     unsigned width, unsigned height, unsigned stride, unsigned depth, unsigned
95     uchar* y_buf, uchar* u_buf, uchar* v_buf, std::string filename)
96 {
97     IplImage *dst = cvCreateImage(cvSize(width, height), depth, nchannels);
98
99     // Convert processed image from Raw to cvImage
100     Raw2IplImage(y_buf, stride, u_buf, stride, v_buf, stride, dst);
101
102     // Conver to cvMat
103     cvConvert( dst, cvCreateMat(height, width, CV_32FC3 ) );
104
105
106     //Configure for newer Opencv-----
107     Mat img = cvarrToMat(dst);
108     bool isSave = imwrite("OutputImageFromKernel.jpg",img);
109     if(isSave){
110         printf("Writed image successfully");
111     }else{
112         printf("fail to write image");
113     }
114     //-----
115     //cvSaveImage(filename.c_str(), dst);
116 }
```

Imread：

```

327 // -----
328 // Read input image and format inputs
329 // -----
330
331 std::string srcFileName = inputImage;
332
333 // Read Input image
334 //IplImage *src;
335 //src = cvLoadImage(srcFileName.c_str()); //format is BGR
336
337 //Configure-----
338 Mat img2;
339 printf("-----\n");
340 printf("Reading Image\n");
341 printf("-----\n");
342 img2 = imread(srcFileName, IMREAD_UNCHANGED );
343 printf("%s\n", srcFileName.c_str());
344 #if CV_MAJOR_VERSION>3
345 | IplImage ipl = cvIplImage(img2);
346 #else
347 | IplImage ipl = img2;
348 #endif
349 IplImage *src = cvCloneImage(&ipl);
350 //
351

```

3Makefile

因為會用到 opencv 所以在 g++(v++)編譯的時候，會需要-I 來指定到 opencv 的 include path，也就是前面提到的/usr/local/include/opencv4。

```

98 endif
99 #Mycode---
100 CXXFLAGS += -I$(OPENCV_INCLUDE)/opencv2/
101 #-----
102 CXXFLAGS += -I$(SRC_REPO)

```

這樣還不足，因為 imread 會報錯。

error: undefined reference to 'cv::imread(std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char>> const&, int)'

這是因為原版的 code 是使用 opencv2.4。在 opencv3 之後，opencv 多一個 libopencv_imgcodecs 出來，需要在 makefile 裡面確保在編譯的時候有 Link 到這個東西：

如下 -lopencv_imgcodecs

```

109 endif
110 #mycode-----
111 CXXLDFLAGS += -L$(OPENCV_LIB)/
112 CXXLDFLAGS += -lopencv_imgcodecs
113 #-----
114 CXXLDFLAGS += -l/usr/local/lib/

```

這樣就可以了。

5. 實驗：

PRELAB：Baseline the Application Performance

介紹：

在 prelab 中會簡單在 CPU 和 FPGA 上跑一次 filter 的應用，並比較兩者的 performance。在這篇 prelab 裡面主要是實現一個 video filter。

事前設定：

加入常用資料夾

➤ `export CONV_TUTORIAL_DIR=./Vitis-Tutorials/Hardware_Acceleration/Design_Tutorials/01-convolution-tutorial`

這樣每次就可以直接進來

➤ `cd $CONV_TUTORIAL_DIR/`

實驗實作：

Software 分析(跑在 CPU 上)

➤ `cd $CONV_TUTORIAL_DIR/sw_run`

➤ `./run.sh`

並看看報告：

```
-----
Number of runs      : 60
Image width         : 1920
Image height        : 1080
Filter type         : 6

Generating a random 1920x1080 input image
Running Software version on 60 images

CPU   Time          :    17.5911 s
CPU   Throughput    :    20.2350 MB/s
-----
```

這裡顯示的是該應用在 CPU 上運行的 performance

執行 FPGA 加速應用(Hardware)：

接下來要將應用實際跑在 FPGA 上面(System Run)

➤ `make run`

然後看報告。

```

-----
Xilinx 2D Filter Example Application (Randomized Input Version)

FPGA binary      : ./fpgabin/hw.xclbin
Number of runs   : 60
Image width      : 1920
Image height     : 1080
Filter type      : 3
Max requests     : 6
Compare perf.    : 1

Programming FPGA device
XRT build version: 2.13.466
Build hash: f5505e402c2ca1ffe45eb6d3a9399b23a0dc8776
Build date: 2022-04-14 17:43:11
Git branch: 2022.1
PID: 269361
UID: 1059
[Sun Mar 26 09:32:20 2023 GMT]
HOST: HLS04
EXE: /mnt/HLSNAS/04.HP0mGn/m111061549/LabB/Vitis-Tutorials/Hardware_Acceleration/Design_Tutorials/01-convolution-tutorial/build/host.exe
[XRT] WARNING: Trace Buffer size is too big. The maximum size of 4095M will be used.
[XRT] WARNING: Trace buffer size for 0th. TS2MM is too big for memory resource. Using 268435456 instead.
Generating a random 1920x1080 input image
Running FPGA accelerator on 60 images
Running Software version
Comparing results

Test PASSED: Output matches reference

FPGA Time       : 0.4198 s
FPGA Throughput : 847.9343 MB/s
CPU Time        : 17.3046 s
CPU Throughput  : 20.5701 MB/s
FPGA Speedup    : 41.2216 x
-----
04.HP0mGn@HLS04: /mnt/HLSNAS/04.HP0mGn/m111061549/LabB/Vitis-Tutorials/Hardware_Acceleration/Design_Tutorials/01-c

```

總結與討論：

Prelab 的用意僅僅是要用 CPU 和 FPGA 各跑一次應用。從報告可以顯示：使用 FPGA 來加速應用之後，可以很大幅度的提升運算速度。以本報告使用的板子 U50 為例，FPGA 加速了 41 倍左右。而使用 U200(Tutorial)的例子，則是加速到 68x。

LAB1 :

介紹：

Lab1 要來實現 2D video convolution filter 這個東西。並且在 host 端來分析他的 performance，並建立一個 performance baseline。

這個 LAB 的目的如下：

- 了解 video convolution filters
- 分析測量在軟體上實現的 convolution filter 的效能
- 計算軟體實現的 filter 在給定的效能限制下計算其所需之加速。
- 在硬體實現前，估計在硬體上的表現。

在本實驗中要對 1080p HD 的 Video 做處理。

那在實現之前，可以估算其所需的 performance 需求：

我們需要 1080pHD(1920 x 1080)，60FP 的影片，RGB 三色且每個位元深度為 8bit。

Requirement：

Video Resolution = 1920 x 1080

Frame Width (pixels) = 1920

Frame Height (pixels) = 1080

Frame Rate(FPS) = 60

Pixel Depth(Bits) = 8

Color Channels(YUV) = 3

計算若要達到要求所需的系統 Throughput。

Throughput(Pixel/s) = Frame Width * Frame Height * Channels * FPS

Throughput(Pixel/s) = 1920*1080*3*60

Throughput (MB/s) = 373 MB/s

可以得到所需的 Throughput 為 373 MB/s。

實驗實作：

Software Implementation

在本 lab 中，有兩篇 code：

- src/host_randomized.cpp
- src/filter2d_sw.cpp //定義卷積濾波器的軟體實現

其中實現濾波器的 code 如下：

```
void Filter2D(
    const char      coeffs[FILTER_V_SIZE][FILTER_H_SIZE],
    float           factor,
    short           bias,
    unsigned short   width,
    unsigned short   height,
    unsigned short   stride,
    const unsigned char *src,
    unsigned char    *dst)
{
    for(int y=0; y<height; ++y)
    {
        for(int x=0; x<width; ++x)
        {
            // Apply 2D filter to the pixel window
            int sum = 0;
            for(int row=0; row<FILTER_V_SIZE; row++)
            {
                for(int col=0; col<FILTER_H_SIZE; col++)
                {
                    unsigned char pixel;
                    int xoffset = (x+col-(FILTER_H_SIZE/2));
                    int yoffset = (y+row-(FILTER_V_SIZE/2));
                    // Deal with boundary conditions : clamp pixels to 0 when outside of image
                    if ( (xoffset<0) || (xoffset>=width) || (yoffset<0) || (yoffset>=height) ) {
                        pixel = 0;
                    } else {
                        pixel = src[yoffset*stride+xoffset];
                    }
                    sum += pixel*coeffs[row][col];
                }
            }

            // Normalize and saturate result
            unsigned char outpix = MIN(MAX((int)(factor * sum)+bias), 0), 255);

            // Write output
            dst[y*stride+x] = outpix;
        }
    }
}
```

跑跑看：

```
-----
Number of runs      : 60
Image width         : 1920
Image height        : 1080
Filter type         : 6

Generating a random 1920x1080 input image
Running Software version on 60 images

CPU Time           : 17.5978 s
CPU Throughput     : 20.2274 MB/s
-----
```

得到以上的結果。

調查了一下遠端電腦所使用的硬體設備

```
Model: 167
Model name: 11th Gen Intel(R) Core(TM) i7-11700 @ 2.50GHz
Stepping: 1
CPU MHz: 801.372
CPU max MHz: 4900.0000
CPU min MHz: 800.0000
```

也就是說剛剛產生的結果是由這棵 CPU 跑出來的。

我們可以發現，這樣一張圖片，CPU 要處理這麼長的時間。換算成 FPS 就是 $60/17.6=3.41\text{FPS}$ 。

也就是說，我們需要使用硬體加速。可是，加速要加速多少？

可以用以下的公式來計算：

Acceleration Factor = Throughput (Required)/Throughput(SW only)

可以算出： $373/20.2274=18.44x$

如果要達到系統要求，需要 18.44 倍的加速。

Hardware Implementation

觀察 code，每一次的 filter 運算都是對一個 1920X1080 的圖片做運算。做完一次 convolution 運算會產生一個 pixel 的 output，而一次的 convolution 運算就是 filter 跟 input matrix 的其中一塊 submatrix 做完一次 pixel 對 pixel 相乘(一個 pixel 有 8bits 的位元深度)後全部加總。在這篇 code 中，filter size 是 15X15，所以一次的運算需要做 225 次的 SOP。或者稱為 225 次的 multiply-accumulate(MAC)

Baseline Hardware Implementation Performance：

首先要來計算一下基本的硬體實現的 performance。

```
for(int row=0; row<FILTER_V_SIZE; row++)
{
    for(int col=0; col<FILTER_H_SIZE; col++)
    {
        unsigned char pixel;
        int xoffset = (x+col-(FILTER_H_SIZE/2));
        int yoffset = (y+row-(FILTER_V_SIZE/2));
        // Deal with boundary conditions : clamp pixels to 0 when outside of image
        if ( (xoffset<0) || (xoffset>=width) || (yoffset<0) || (yoffset>=height) ) {
            pixel = 0;
        } else {
            pixel = src[yoffset*stride+xoffset];
        }
        sum += pixel*coeffs[row][col];
    }
}
```

上圖的巢狀迴圈，執行一次就是執行 225 次的 SOP，即一次的 MAC

最簡單粗暴的加速方法就是將 kernel code 放到 Vitis HLS tool(Baseline hardware implementation)。它會使用 pipeline 優化上圖的 loop 使得 II=1，那麼一個 cycle 就只會執行一次的 MAC。可以估計一下 performance。

➤ platforminfo -p xilinx_u50_gen3x16_xdma_5_202210_1

```

=====
Clock Information
=====
Default Clock Index: 0
Clock Index:         0
Frequency:            300.000000
Clock Index:         1
Frequency:            500.000000
Clock Index:         2
Frequency:            100.000000

```

可以看到板子的時脈大概是 300MHz。

MACs per Cycle = 1

Hardware Fmax(MHz) = 300

Throughput = $300/225 = 1.33$ (MPixels/s) = 1.33 MB/s(一個 pixel 有 8bits 的位元深度)

可以估算出若將原本的 source code 的 filter loop 做 pipeline 使其 II=1，那麼一次的 cycle 做一次的 MAC。那麼一秒鐘總共處理 1.33MB 的資料。

接著估算所需的 memory throughput。輸出的部分，一秒可以做 1.33MB(一秒 133 萬個 pixels)。然後寫出 memory。那麼

Output Memory Bandwidth = Throughput = 1.33 MB/s

但是輸出一個 pixel(8bits)，需要做 225 次的 MAC。那麼

Input Memory Bandwidth = Throughput * 225 = 300 MB/s

要達成前述之要求(Throughput (MB/s) = 373 MB/s)，那麼

Acceleration Factor to Meet 60FPS Performance = $373/1.33 = 280x$

Acceleration Factor to Meet SW Performance = $20.2274/1.33 = 15.2x$

這非常地清楚，Baseline hardware implementation，也就是基本的硬體實現，必須要加速 280 倍才能達到 60FPS 的要求。

Performance Estimation for Optimized Hardware Implementations

顯而易見地，Baseline hardware implementation 要達成系統要求還遠遠不足，必須要針對 kernel code 做優化。我們可以好好想想如何優化 convolution 的計算。

比如說，將 Filter2D 的最裡面的兩層 loop 做 unroll。也就是把 15*15 次的 MAC 做 unroll，使得一個 cycle 就可以處理好一個 pixel。這意味著 225x 的加速。

Throughput = Fmax * Pixels produced per cycle = $300 * 1 = 300$ MB/s

(Again, one pixel contains 8 bits)

Output Memory Bandwidth = Fmax * Pixels produced per cycle = 300 MB/s

Input Memory Bandwidth

= Fmax * Input pixels read per output pixel = $300 * 225 = 67.5$ GB/s

這樣一來，輸出的 output throughput 提升了。但 input 的提升卻來到一個極其可怕的數值。67.5 GB/s 對於 FPGA 來說，顯然是力不從心的。

透過觀察 code，其實並不用每次都讀進 15*15 的 filter。或許可以利用 caching 的技巧，來優化 filter 運算。

這邊所使用的 filter 屬於 stencil kernels(一種種類的 filter)，可以被優化來提升 input data 的 reuse。透過 caching 的技巧，可以讓 input 所需的 bandwidth 降至跟 output 一樣 level。也就是差不多 300MB/s。那麼每個 cycle 產生一個 output pixel，就只需要平均讀入一個 input pixel。

可是，300MB/s 還是沒辦法達到要求的 **373MB**。或許還可以增加運算的單元。可以透過異構計算 heterogeneous computing 來增加運算單元來讓資料平行運算。以卷積濾波為例子，何不讓三個 color channel(YUV)分開平行運算呢？三個計算單元，則可以再加速三倍：

Throughput(estimated) =

Performance of Single Compute Unit * # of Compute Units = 300 x 3 = 900 MB/s

Acceleration Against Software Implementation = 900/14.5 = 62x

Kernel Latency (per image on any color channel) = (1920*1080) / 300 = 6.9 ms

Video Processing Rate = (1/Kernel Latency) = 144 FPS

結論：

在這個 lab1，除了了解了一個基本的 convolution filter 架構，並實際分析了軟體實現的 performance，並了解只讓 code 跑在 CPU 上，並不能達成要求。所以接著我們估算了在硬體上實現的 performance，了解到 baseline 的硬體加速的限制(Baseline Hardware Implementation Performance)，從而思考可能的優化，提出方案之後加以分析，估算其 performance(Performance Estimation for Optimized Hardware Implementations)\

接下來就是實驗的重點了：要如何實現我們在 Performance Estimation for Optimized Hardware Implementations。

LAB2：

介紹：

在 LAB1 裡不斷地計算、估計系統的需求，可以發現 baseline implementation 是遠遠不夠的，我們並沒有辦法將現有的 kernel(software)的 code 直接利用 hls 的 pipeline 讓他自己得到所需求的系統效能。因此在 LAB1 的結尾，有幾個方法被提出：

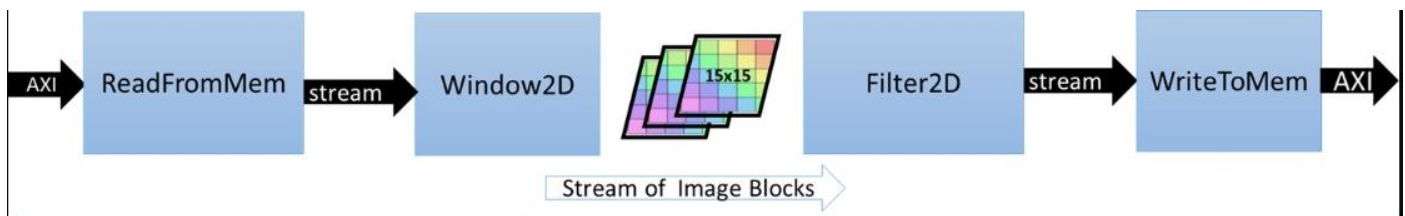
(1)使用 caching 的技巧，降低 read 所需的頻寬，使得 Input Memory Bandwidth 和 output Memory Bandwidth 在同樣等級。

(2)使用三個運算單元，將圖片的三個 channel 分開運算。

在這個 LAB，將會講述如何做出這 **caching** 方法，並輔以分析工具來加以驗證。

Kernel Code 分析：

Kernelcode 內部有三個主要的功能：



- ReadFromMem：負責將資料從 Global memory 讀進來
- Window2D：在每一個 cycle，提供 15X15 的 input pixel sample 給下一階做 filter2D，**並在這邊實現 Caching**。
- Filter2D：使用 15X15 的 filter，來對前一級輸入的 15X15 的 input pixel sample 做卷積運算，並輸出一個 pixel。
- WriteToMem 將算好的 pixel 寫回 global memory。

以上是整個 kernel 的資料流，並且使用 #pragma HLS DATAFLOW 來做 dataflow 的優化。因此，內部 block 之間的資料傳輸是使用 stream。

Caching 技巧：

在 LAB1 提到，若沒有特別作優化，input memory bandwidth requirement 會是一個龐大的數字。Caching 的目的就是讓 Input memory bandwidth 和 output memory bandwidth 持平。接下來要來解釋這個優化演算法：

Line Buffer：

Line buffer 是整個演算法的最核心的東西。假設 filter size 為 $w \times h$ ，input image width 為 $W \times H$ ，則 line buffer size = $(h-1) \times W$ 。以本專案為例，15X15 的 filter，1920 的 input image width，line buffer 大小就是 $(15-1) \times (1920)$

Line buffer 的作用如下圖：

想像現在在 Window2D 這個 block，假設有一個 input image，width=8，且 filter size 為 3x3。那 line buffer 為 2×8 。這個 window2D 的作用就是每次從 input 中取樣 3X3 大小的 window 丟給下一級作運算。Window 可以想成是一個 3x3 的 output buffer。首先從 A 開始，A 當中，為了輸出 10 這個 pixel，要將周圍一圈的 pixel 和 filter 做 MAC 運算，此時這個 line buffer 所存的值如紅色框框所示：

A

1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16
17	18	19					

再來 filter 右移，如 B

B

1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16
17	18	19	?				

在 A 的時候，右邊兩個 column 在 B 時會左移一格，也就是 2、3、10、11、18、19，也就是已經在 window 裡面了。Window 的新的值會從右邊的 column 近來，其中 4、12 已經在 line buffer 裡面，那麼只剩下?是還沒有輸入的值，也就是說這時候只需要向前一級讀入一個值，就可以傳下一份 3X3 的 input。再來移動完之後，更新 line buffer 的值，來應付下一次的運算。

B

1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16
17	18	19	?				

Kernel Code：

框起的地方就是 line buffer。

```

void Window2D(
    unsigned short    width,
    unsigned short    height,
    hls::stream<U8>    &pixel_stream,
    hls::stream<window> &window_stream)
{
    // Line buffers - used to store [FILTER_V_SIZE-1] entire lines of pixels
    U8 LineBuffer[FILTER_V_SIZE-1][MAX_IMAGE_WIDTH];
    #pragma HLS ARRAY_PARTITION variable=LineBuffer dim=1 complete
    #pragma HLS DEPENDENCE variable=LineBuffer inter false
    #pragma HLS DEPENDENCE variable=LineBuffer intra false

    // Sliding window of [FILTER_V_SIZE][FILTER_H_SIZE] pixels
    window Window;

    unsigned col_ptr = 0;
    unsigned ramp_up = width*((FILTER_V_SIZE-1)/2)+(FILTER_H_SIZE-1)/2;
    unsigned num_pixels = width*height;
    unsigned num_iterations = num_pixels + ramp_up;

    const unsigned max_iterations = MAX_IMAGE_WIDTH*MAX_IMAGE_HEIGHT + MAX_IMAGE_WIDTH*((FILTER_V_SIZE-1)/2)+(FILTER_H_SIZE-1)/2;

    // Iterate until all pixels have been processed
    update_window: for (int n=0; n<num_iterations; n++)
    {
        #pragma HLS LOOP_TRIPCOUNT max=max_iterations
        #pragma HLS PIPELINE II=1

        // Read a new pixel from the input stream
        U8 new_pixel = (n<num_pixels) ? pixel_stream.read() : 0;

        // Shift the window and add a column of new pixels from the line buffer
        for(int i = 0; i < FILTER_V_SIZE; i++) {
            for(int j = 0; j < FILTER_H_SIZE-1; j++) {
                Window.pix[i][j] = Window.pix[i][j+1];
            }
            Window.pix[i][FILTER_H_SIZE-1] = (i<FILTER_V_SIZE-1) ? LineBuffer[i][col_ptr] : new_pixel;
        }

        // Shift pixels in the column of pixels in the line buffer, add the newest pixel
        for(int i = 0; i < FILTER_V_SIZE-2; i++) {
            LineBuffer[i][col_ptr] = LineBuffer[i+1][col_ptr];
        }
        LineBuffer[FILTER_V_SIZE-2][col_ptr] = new_pixel;

        // Update the line buffer column pointer
        if (col_ptr==(width-1)) {
            col_ptr = 0;
        } else {
            col_ptr++;
        }
    }
}

```

Hardware implementation

在專案裡面輸入指令：

- `cd $CONV_TUTORIAL_DIR/hls_build`
- `vitis_hls -f build.tcl`

其實就是打開 `vitis_hls`，然後把 kernel file 加進去，並且執行模擬而已。

過一陣子之後她會產生一個報告。

```

INFO: [COSIM 212-316] Starting C post checking ...
-----
HLS Testbench for Xilinx 2D Filter Example
* Image info
- Width      :      1000
- Height     :        30
- Stride     :     1024
- Bytes      :     30720
* Running FPGA accelerator
* Comparing results
Test PASSED: Output matches reference
-----

```


輸入指令來啟動 HLS GUI 來分析他的 performance。

來看一些分析：







Modules & Loops	Issue Type	Violation Type	Distance	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LUT	URAM
[-] Filter2DKernel				0.00	2211165	7.370E6	-	2211166	-	dataflow	14	139	30282	28234	0
[-] ReadFromMem				0.00	2211165	7.370E6	-	2211165	-	no	0	1	2467	1484	0
[-] Window2D				0.00	2087055	6.956E6	-	2087055	-	no	14	1	2237	492	0
[-] entry_proc				-	0	0.0	-	0	-	no	0	0	3	47	0
[-] Filter2D				0.00	2073859	6.912E6	-	2073859	-	no	0	136	16365	14319	0
[-] WriteToMem				0.00	2210837	7.369E6	-	2210837	-	no	0	1	889	1299	0

Bind Storge Report :

如下圖，有 14 個 BRAM 被分配來當作 LineBuffer，我猜因為 filter size 是 15X15，所以生出 14 個 LineBuffer，並且一個儲存一整個 input width 的值。

The screenshot shows the Vivado IDE interface. On the left, the Project Explorer displays the project structure for 'filter2d_hw.cpp'. The project is organized into a hierarchy: 'filter2d_hw.cpp' (root) contains 'ReadFromMem', 'Window2D', and 'Window2D_Pipeline_update_window'. 'ReadFromMem' contains 'ReadFromMem_Pipeline_read_coefs' and 'ReadFromMem_Pipeline_read_image'. 'Window2D' contains 'LineBuffer_U' through 'LineBuffer_13_U' and 'Window2D_Pipeline_update_window'. 'Window2D_Pipeline_update_window' contains 'update_window'. The 'LineBuffer_U' through 'LineBuffer_13_U' are listed with their respective ports and data types. The 'Window2D' component is highlighted in orange. On the right, the Source window displays the C++ code for 'Window2D'. The code includes comments and pragmas for HLS optimization. The code defines a 'Window2D' function that takes a stream of unsigned shorts and returns a stream of unsigned shorts. It uses a sliding window of size [FILTER_V_SIZE][FILTER_H_SIZE] pixels to process the input stream. The code includes comments and pragmas for HLS optimization, such as 'HLS ARRAY_PARTITION variable=LineBuffer dim=1 complete' and 'HLS DEPENDENCE variable=LineBuffer intra false'. The code also includes a loop for iterating until all pixels have been processed, with a maximum of 'MAX_IMAGE_WIDTH*MAX_IMAGE_HEIGHT' iterations. The code uses a 'const unsigned max_iterations' variable to define the loop limit. The code also includes a 'prAGMA HLS PIPELINE II=1' directive to enable pipeline parallelism. The code uses a 'const unsigned max_iterations' variable to define the loop limit. The code also includes a 'prAGMA HLS PIPELINE II=1' directive to enable pipeline parallelism. The code uses a 'const unsigned max_iterations' variable to define the loop limit. The code also includes a 'prAGMA HLS PIPELINE II=1' directive to enable pipeline parallelism.

Co-simulation report

Modules & Loops	Avg II	Max II	Min II	Avg Latency	Max Latency	Min Latency
▼  Filter2DKernel				38455	38455	38455
▶  ReadFromMem				31127	31127	31127
▶  Window2D				38242	38242	38242
 entry_proc				0	0	0
▶  Filter2D				38268	38268	38268
▶  WriteToMem				38454	38454	38454

Synthesis report

Performance & Resource Estimates ⓘ												
Modules & Loops												
Issue Type	Violation Type	Distance	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	
		0.00	2211165	7.370E6	-	2211166	-	2211165	-	dataflow	14	139 300
		0.00	2211165	7.370E6	-	2211165	-	2211165	-	no	0	1 2
		0.00	259	863.000	-	259	-	259	-	no	0	0 1
		-	257	857.000	3	1	256	256	yes	-	-	-
		0.00	2210763	7.368E6	-	2210763	-	2210763	-	no	0	0 1
		-	2210761	7.368E6	3	1	2210760	2210760	yes	-	-	-
		0.00	2087055	6.956E6	-	2087055	-	2087055	-	no	14	1 2
		-	2087051	6.956E6	-	2087051	-	2087051	-	no	0	0 2
		-	2087049	6.956E6	3	1	2087047	2087047	yes	-	-	-
		-	0	0.0	-	0	-	0	-	no	0	0
		0.00	2073859	6.912E6	-	2073859	-	2073859	-	no	0	136 16
		-	227	757.000	-	227	-	227	-	no	0	0 1
		-	225	750.000	2	1	225	225	yes	-	-	-
		-	2073627	6.911E6	-	2073627	-	2073627	-	no	0	135 14
		-	2073625	6.911E6	27	1	2073600	2073600	yes	-	-	-
		0.00	2210837	7.369E6	-	2210837	-	2210837	-	no	0	1
		0.00	2210763	7.368E6	-	2210763	-	2210763	-	no	0	0
		-	2210761	7.368E6	3	1	2210760	2210760	yes	-	-	-

結論：

在這個 LAB 中，我們實現了一個使用 Caching 技巧的 Convolution 演算法。如同報告所顯示的，的確透過運用 Line Buffer 實現的 caching 技巧，讓整體的 latency 下降。

LAB3：

介紹：

最後要來實作的是，使用 host program 來管理、調用這個 kernel。並且整體來分析系統效能。

在./src/底下，有兩個 hostcode file:

host.cpp //會實際把圖片丟進去跑，並產生結果

host_randomized.cpp //只是生出一個沒有意義的隨機圖片去跑

在以下的實驗中，會用到這兩個 hostcode。這兩個做的事情沒有不同，差別只在於一個真的有把圖片輸入放進去跑並輸出。

由於兩個 hostcode 做的事情一樣，所以只介紹 host_randomize 在干嘛：

Host Code：

Hostcode 有三個部分：

Filter2DRequest：物件型態，每個物件都定義了一個運算單一 color channel 的 convolution 的 host code execution flow(setkernelArg、transfer data、enqueue task...)。

Filter2DDispatcher：處理所有的 task request(1 request 代表一次的 convolution 運算)

Main：整體運行。

首先先定義整體系統，注意 outstanding requests 是 3(代表三個 channel)

```
parser.addSwitch("--nruns", "-n", "Number of times to image is processed", "1");
parser.addSwitch("--fpga", "-x", "FPGA binary (xclbin) file to use");
parser.addSwitch("--width", "-w", "Image width", "1920");
parser.addSwitch("--height", "-h", "Image height", "1080");
parser.addSwitch("--filter", "-f", "Filter type (0-6)", "0");
parser.addSwitch("--maxreqs", "-r", "Maximum number of outstanding requests", "3");
parser.addSwitch("--compare", "-c", "Compare FPGA and SW performance", "false", true);
```

再來宣告 Dispatcher 物件：

```
// -----
// Make requests to kernel(s)
// -----

printf("Running FPGA accelerator on %d images\n", numRuns);

// Dispatcher of requests to the kernel
// 'maxReqs' controls the maximum number of outstanding requests to the kernel
// and equates to the depth of SW pipelining.
Filter2DDispatcher Filter2DKernel(context, program, queue, maxReqs);
```

那麼在 constructor 這邊：會將三個 Filter2DRequest 物件宣告並推入 vector，req 這個 vector 其實就是一次執行的 request(I channel 2D convolution)總數

```
class Filter2DDispatcher
{
    std::vector<Filter2DRequest> req;
    int max;
    int cnt;

public:
    Filter2DDispatcher(cl::Context &context, cl::Program &program, cl::CommandQueue &queue, int nreqs)
    {
        cnt = 0;
        max = nreqs;
        for(int i=0; i<max; i++) {
            req.push_back( Filter2DRequest(context, program, queue) );
        }
    };
};
```

來看看 Filter2DRequest 的 constructor，每一個 Filter2DRequest 被宣告時，都會調用一個 kernel 資源，如紅色框框所框。因為總共在 Dispatcher 宣告了三次 Filter2DRequest，所以總共會調用 3 個 Kernel(CU)。

所以調用複數個 Kernel，是在 host 端調用的。

```
class Filter2DRequest
{
    cl::Kernel        kernel;
    cl::CommandQueue  q;
    cl::Buffer        coef_buffer;
    cl::Buffer        src_buffer;
    cl::Buffer        dst_buffer;
    std::vector<cl::Event> events;

public:
    Filter2DRequest(cl::Context &context, cl::Program &program, cl::CommandQueue &queue)
    {
        cl_int err;

        q = queue;

        OCL_CHECK(err, kernel = cl::Kernel(program, "Filter2DKernel", &err));

        // Allocate input and output buffers
        OCL_CHECK(err, coef_buffer = cl::Buffer(context, CL_MEM_READ_ONLY, (FILTER_V_SIZE*FILTER_V_SIZE)*sizeof(char), nullptr, &err));
        OCL_CHECK(err, src_buffer  = cl::Buffer(context, CL_MEM_READ_ONLY, (1920*1080)*sizeof(char), nullptr, &err));
        OCL_CHECK(err, dst_buffer  = cl::Buffer(context, CL_MEM_WRITE_ONLY, (1920*1080)*sizeof(char), nullptr, &err));

        // Set kernel arguments - this pins the buffers to specific global memory banks
        OCL_CHECK(err, err = kernel.setArg(0, coef_buffer));
        OCL_CHECK(err, err = kernel.setArg(6, src_buffer));
        OCL_CHECK(err, err = kernel.setArg(7, dst_buffer));

        // Make buffers resident in the device
        // If done after setArg, then buffers are pinned and runtime knows in which bank they should be made resident,
        // removing the need for using the vendor extensions to explicitly map to DDR.
        OCL_CHECK(err, err = q.enqueueMigrateMemObjects({coef_buffer, src_buffer, dst_buffer}, CL_MIGRATE_MEM_OBJECT_CONTENT_UNDEFINED));

        // Make sure buffers are migrated before continuing
        q.finish();
    };
};
```

Software Emulation :

make run TARGET=sw_emu

```
f 3 -r 6 -n 1 -w 1920 -h 1080
-----
Xilinx 2D Filter Example Application (Randomized Input Version)

FPGA binary      : ./fpgabinary.sw_emu.xclbin
Number of runs   : 1
Image width      : 1920
Image height     : 1080
Filter type      : 3
Max requests     : 6
Compare perf.    : 1

Programming FPGA device
Kernel Name: Filter2DKernel_1, CU Number: 0, Thread creation status: success
Kernel Name: Filter2DKernel_2, CU Number: 1, Thread creation status: success
Kernel Name: Filter2DKernel_3, CU Number: 2, Thread creation status: success
Generating a random 1920x1080 input image
Running FPGA accelerator on 1 images
  finished Filter2DRequest
  finished Filter2DRequest
  finished Filter2DRequest
Running Software version
Comparing results

Test PASSED: Output matches reference
```

在 make_options.mk 設定 input type 為 ramdon :

```
FILTER_TYPE :=3
PARALLEL_ENQ_REQS :=6
NUM_IMAGES :=1000
INPUT_TYPE :=""
INPUT_IMAGE :=../test_
ENABLE_PROF?=yes
PROFILE_ALL_IMAGES?=no
```

Unrecognized shortcut key passed -t

=====

Usage: application.exe -[-h-n-x-w-h-f-r-c]

--help, -h	prints this help list	Default: [false]
--nruns, -n	Number of times to image is processed	Default: [1]
--fpga, -x	FPGA binary (xclbin) file to use	
--width, -w	Image width	Default: [1920]
--height, -h	Image height	Default: [1080]
--filter, -f	Filter type (0-6)	Default: [0]
--maxreqs, -r	Maximum number of outstanding requests	Default: [3]
--compare, -c	Compare FPGA and SW performance	Default: [false]

FPGA binary : ./fpgabinary.sw_emu.xclbin
Number of runs : 1
Image width : 1920
Image height : 1080
Filter type : 3
Max requests : 6
Compare perf. : 1

Programming FPGA device

Kernel Name: Filter2DKernel_1, CU Number: 0, Thread creation status: success

Kernel Name: Filter2DKernel_2, CU Number: 1, Thread creation status: success

Kernel Name: Filter2DKernel_3, CU Number: 2, Thread creation status: success

Generating a random 1920x1080 input image

Running FPGA accelerator on 1 images

finished Filter2DRequest

finished Filter2DRequest

finished Filter2DRequest

Running Software version

Comparing results

Test PASSED: Output matches reference

device process sw_emu device done

Hardware Emulation :

先跑：host.cpp

```
cp xrt.ini ./build_hw_emu;
cd ./build_hw_emu && XCL_EMULATION_MODE=hw_emu ./host.exe -x ./fpgabinary.hw_emu.xclbin -c -f 3 -r 6 -n 1 -i ../test_images/xilinx_versal_1080p.bmp

-----
Xilinx 2D Filter Example Application

FPGA binary      : ./fpgabinary.hw_emu.xclbin
Input image      : ../test_images/xilinx_versal_1080p.bmp
Number of runs   : 1
Filter type      : 3
Max requests     : 6
Compare perf.    : 1

-----
Reading Image
-----
/mnt/HLSNAS/04.HP0mGn/m111061549/LabB/Vitis-Tutorials/Hardware_Acceleration/Design_Tutorials/01-convolution-tutorial/images/inputImage50.jpg
Programming FPGA device
INFO: [HW-EMU 01] Hardware emulation runs simulation underneath. Using a large data set will result in long simulation times. It is recommended that a small
dataset is used for faster execution. The flow uses approximate models for Global memories and interconnect and hence the performance data generated is app
ximate.
configuring dataflow mode with ert polling
scheduler config ert(1), dataflow(1), slots(16), cudma(0), cuivr(0), cdma(0), cus(3)
INFO: [HW-EMU 07-0] Please refer the path "/mnt/HLSNAS/04.HP0mGn/m111061549/LabB/Vitis-Tutorials/Hardware_Acceleration/Design_Tutorials/01-convolution-tuto
al/build_hw_emu/.run/194659/hw_em/device0/binary_0/behav_waveform/xsim/simulate.log" for more detailed simulation infos, errors and warnings.
Running FPGA accelerator on 1 images
finished Filter2DRequest
finished Filter2DRequest
finished Filter2DRequest
Wrote image successfullyRunning Software version
Wrote image successfullyComparing results

Test PASSED: Output matches reference
-----
INFO::[ Vitis-EM 22 ] [Time elapsed: 3 minute(s) 45 seconds, Emulation time: 0.999817 ms]
Data transfer between kernel(s) and global memory(s)
Filter2DKernel_1:m_axi_gmem-HBM[1]      RD = 169.000 KB      WR = 168.750 KB
Filter2DKernel_2:m_axi_gmem-HBM[1]      RD = 169.000 KB      WR = 168.750 KB
Filter2DKernel_3:m_axi_gmem-HBM[1]      RD = 169.000 KB      WR = 168.750 KB

INFO: [HW-EMU 06-0] Waiting for the simulator process to exit
INFO: [HW-EMU 06-1] All the simulator processes exited successfully
INFO: [HW-EMU 07-0] Please refer the path "/mnt/HLSNAS/04.HP0mGn/m111061549/LabB/Vitis-Tutorials/Hardware_Acceleration/Design_Tutorials/01-convolution-tuto
al/build_hw_emu/.run/194659/hw_em/device0/binary_0/behav_waveform/xsim/simulate.log" for more detailed simulation infos, errors and warnings.
04.HP0mGn@HLS04:~/m111061549/LabB/Vitis-Tutorials/Hardware_Acceleration/Design_Tutorials/01-convolution-tutorial$
```

產生圖片結果



然後跑 host_randomize.cpp :

```
-----start random-----FPGA binary      : ./fpgabinaty.hw_emu.xclbin
Number of runs      : 1
Image width         : 1920
Image height        : 100
Filter type         : 3
Max requests        : 6
Compare perf.       : 1

Programming FPGA device
INFO: [HW-EMU 01] Hardware emulation runs simulation underneath. Using a large data set will result in long simulation times. It is recommended that a small
dataset is used for faster execution. The flow uses approximate models for Global memories and interconnect and hence the performance data generated is appro
ximate.
INFO: [HW-EMU 07-0] Please refer the path "/mnt/HLSNAS/04.HP0mGn/m11061549/LabB/Vitis-Tutorials/Hardware_Acceleration/Design_Tutorials/01-convolution-tutori
al/build_hw_emu/.run/198734/hw_em/device0/binary_0/behav_waveform/xsim/simulate.log" for more detailed simulation infos, errors and warnings.
configuring dataflow mode with ert polling
scheduler config ert(1), dataflow(1), slots(16), cudma(0), cuisr(0), cdma(0), cus(3)
Generating a random 1920x100 input image
Running FPGA accelerator on 1 images
finished Filter2DRequest
INFO: [Vitis-EM 22 ] [Time elapsed: 5 minute(s) 32 seconds, Emulation time: 1.4072 ms]
Data transfer between kernel(s) and global memory(s)
Filter2DKernel_1:m_axi_gmem-HBM[1]      RD = 187.750 KB      WR = 187.500 KB
Filter2DKernel_2:m_axi_gmem-HBM[1]      RD = 187.750 KB      WR = 187.500 KB
Filter2DKernel_3:m_axi_gmem-HBM[1]      RD = 187.750 KB      WR = 180.000 KB

finished Filter2DRequest
finished Filter2DRequest
Running Software version
Comparing results

Test PASSED: Output matches reference
-----
INFO: [Vitis-EM 22 ] [Time elapsed: 5 minute(s) 47 seconds, Emulation time: 1.52761 ms]
Data transfer between kernel(s) and global memory(s)
Filter2DKernel_1:m_axi_gmem-HBM[1]      RD = 187.750 KB      WR = 187.500 KB
Filter2DKernel_2:m_axi_gmem-HBM[1]      RD = 187.750 KB      WR = 187.500 KB
Filter2DKernel_3:m_axi_gmem-HBM[1]      RD = 187.750 KB      WR = 187.500 KB

INFO: [HW-EMU 06-0] Waiting for the simulator process to exit
INFO: [HW-EMU 06-1] All the simulator processes exited successfully
INFO: [HW-EMU 07-0] Please refer the path "/mnt/HLSNAS/04.HP0mGn/m11061549/LabB/Vitis-Tutorials/Hardware_Acceleration/Design_Tutorials/01-convolution-tutorial/build
run/198734/hw_em/device0/binary_0/behav_waveform/xsim/simulate.log" for more detailed simulation infos, errors and warnings.
make: warning: Clock skew detected. Your build may be incomplete.
04.HP0mGn@HLS04:~/m11061549/LabB/Vitis-Tutorials/Hardware_Acceleration/Design_Tutorials/01-convolution-tutorial$ Connection to 140.112.207.200 closed by remote host
Connection to 140.112.207.200 closed.
```

Hardware :

- make build TARGET=hw
- make run TARGET=hw

```
-----start random-----FPGA binary      : ./fpgabinaty.hw.
xclbin
Number of runs      : 60
Image width         : 1920
Image height        : 100
Filter type         : 3
Max requests        : 6
Compare perf.       : 1

Programming FPGA device
XRT build version: 2.13.466
Build hash: f5505e402c2ca1ffe45eb6d3a9399b23a0dc8776
Build date: 2022-04-14 17:43:11
Git branch: 2022.1
PID: 388027
UID: 1059
[Fri Mar 31 17:49:02 2023 GMT]
HOST: HLS04
EXE: /mnt/HLSNAS/04.HP0mGn/m11061549/LabB/Vitis-Tutorials/Hardware_Acceleration/Design_Tutorials/01-co
nvolution-tutorial/build/host.exe
[XRT] WARNING: Trace Buffer size is too big. The maximum size of 4095M will be used.
[XRT] WARNING: Trace buffer size for 0th. TS2MM is too big for memory resource. Using 268435456 instea
d.
Generating a random 1920x100 input image
Running FPGA accelerator on 60 images
Running Software version
Comparing results

Test PASSED: Output matches reference

FPGA Time          :    0.0417 s
FPGA Throughput    :   789.7792 MB/s
CPU Time           :    1.6315 s
CPU Throughput     :   20.2011 MB/s
FPGA Speedup       :   39.0959 x
-----
04.HP0mGn@HLS04:~/m11061549/LabB/Vitis-Tutorials/Hardware_Acceleration/Design_Tutorials/01-convolution-tut
```

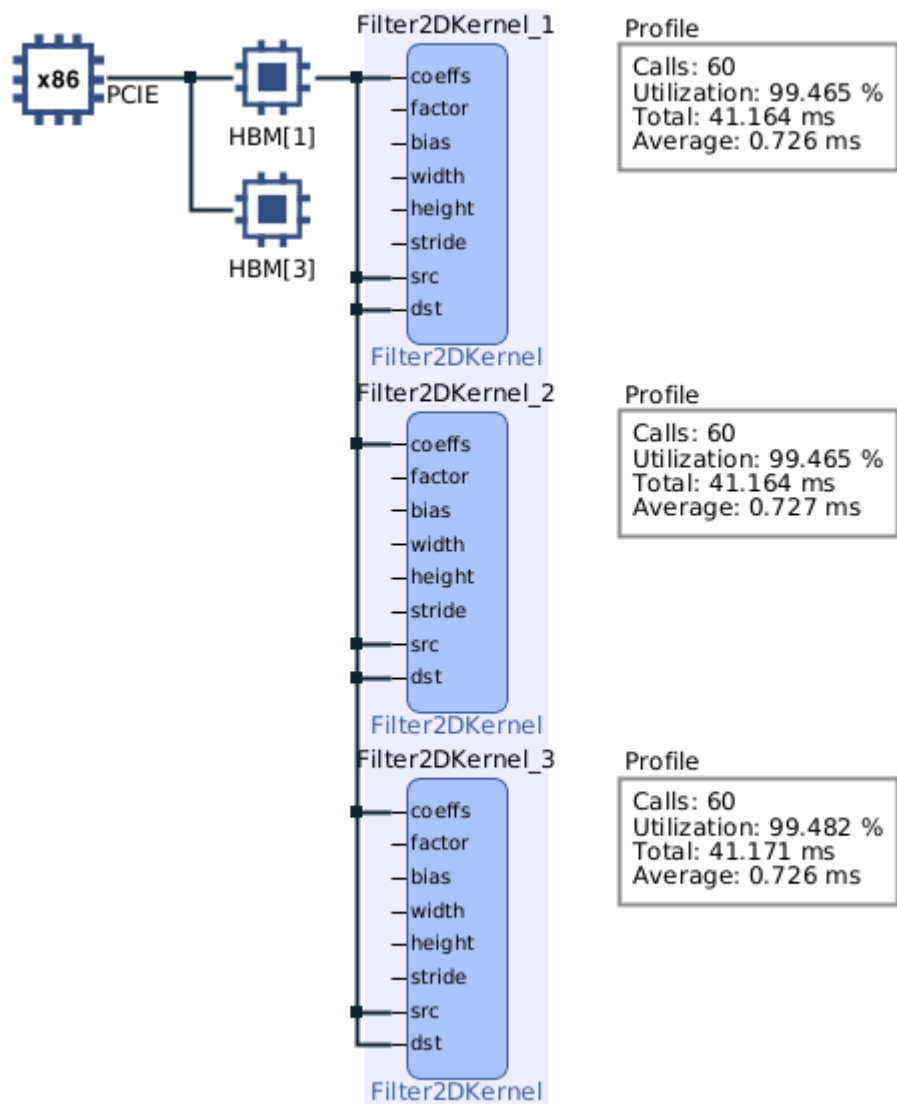
從這個報告我們可以看到。789MB/s，這和先前所要求的 FPGA 運算單元的 Throughput 的 900MB 其實，相去並不算太遙遠。整體來講，FPGA 提升了 39x，這在 Prelab 所列出的要求

中。看一下 prelab 所需要的 FPGA 加速：41x，其實和 39 相去不遠。

```
FPGA Time      : 0.4198 s
FPGA Throughput : 847.9343 MB/s
CPU Time       : 17.3046 s
CPU Throughput  : 20.5701 MB/s
FPGA Speedup    : 41.2216 x
```

System Diagram :

首先來看一下系統合成了什麼樣的硬體：的確如先前設計的，我們使用三個 CU 來分別跑三個不同的 color channel，所以總共用到了三份 Convolution Kernel。



Profile Summary :

再來是 summary，由於系統有做 host code optimization，使用 out of order queue 輔 synchronization，這相當程度地提升了 utilization。

Kernels & compute Unit

從這個 summary 也可以明確地看到，在 compute unit utilization 那裡，的確生了三份硬體來處理圖片，並各自都有 99% 以上的 CU utilization。這歸功於 kernel 的設計，因為使用 caching 技巧，kernel 不用每次都要等待 15×15 的 window 被讀入才能做運算，這大大減少的 idle 的時間。關於這點會在 timeline 上更詳細地分析說明。

Kernels & Compute Units

Kernel Execution

Kernel	Enqueues	Total Time (ms)	Min Time (ms)	Avg Time (ms)	Max Time (ms)
Filter2DKernel	180	221.129	0.716	1.228	1.288

Top Kernel Execution

Kernel	Kernel Instance Address	Context ID	Command Queue ID	Device	Start Time (ms)	Duration (ms)
Filter2DKernel	0x564b943261a0	0	0	xilinx_u50_gen3x16_xdma_base_5-0	4245.270	1.288
Filter2DKernel	0x564b943261a0	0	0	xilinx_u50_gen3x16_xdma_base_5-0	4248.030	1.276
Filter2DKernel	0x564b9441a190	0	0	xilinx_u50_gen3x16_xdma_base_5-0	4241.340	1.273
Filter2DKernel	0x564b942dfd20	0	0	xilinx_u50_gen3x16_xdma_base_5-0	4255.750	1.273
Filter2DKernel	0x564b9441a190	0	0	xilinx_u50_gen3x16_xdma_base_5-0	4235.850	1.273
Filter2DKernel	0x564b942dfd20	0	0	xilinx_u50_gen3x16_xdma_base_5-0	4258.500	1.273
Filter2DKernel	0x564b9441a190	0	0	xilinx_u50_gen3x16_xdma_base_5-0	4255.070	1.273
Filter2DKernel	0x564b9441a190	0	0	xilinx_u50_gen3x16_xdma_base_5-0	4244.090	1.273
Filter2DKernel	0x564b9441a190	0	0	xilinx_u50_gen3x16_xdma_base_5-0	4245.460	1.272
Filter2DKernel	0x564b942dfd20	0	0	xilinx_u50_gen3x16_xdma_base_5-0	4229.680	1.271

Compute Unit Utilization

Compute Unit	Kernel	Device	Calls	Dataflow Execution	Max Parallel Executions	Dataflow Acceleration	CU Device Utilization (%)	CU Kernel Utilization (%)	Total Time (ms)	Min Time (ms)	Avg Time (ms)	Max Time (ms)	Clock Freq (MHz)
Filter2DKernel_1	Filter2DKernel	xilinx_u50_gen3x16_xdma_base_5-0	60	Yes	2	1.058539x	99.465	18.615	41.164	0.688	0.726	0.730	300.000
Filter2DKernel_2	Filter2DKernel	xilinx_u50_gen3x16_xdma_base_5-0	60	Yes	2	1.059030x	99.465	18.615	41.164	0.688	0.727	0.730	300.000
Filter2DKernel_3	Filter2DKernel	xilinx_u50_gen3x16_xdma_base_5-0	60	Yes	2	1.057572x	99.482	18.619	41.171	0.690	0.726	0.730	300.000

<

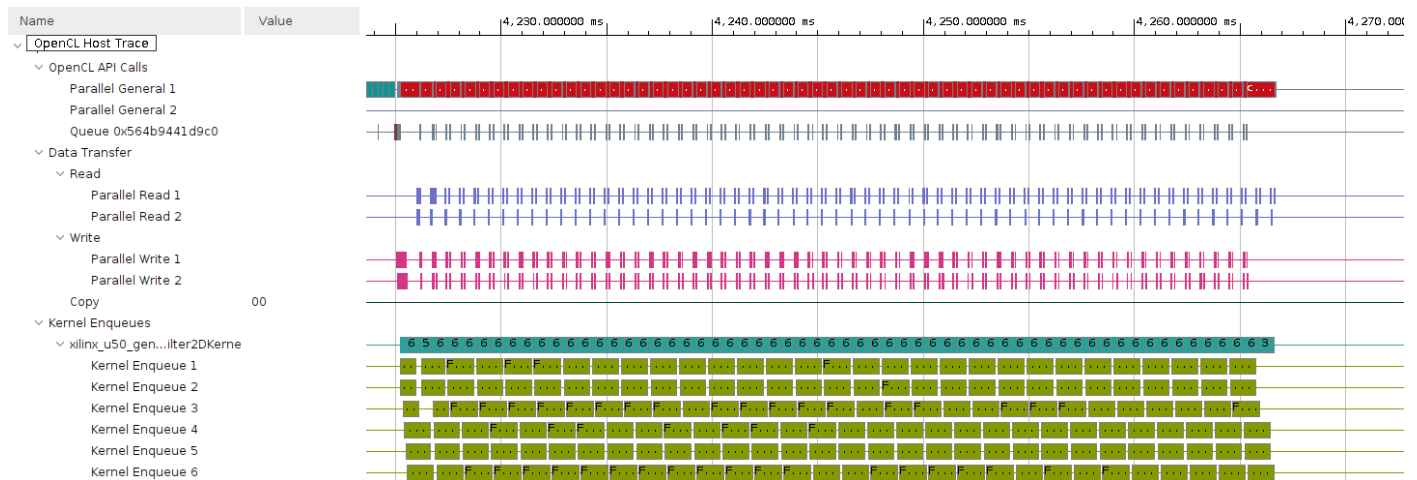
Compute Unit Stalls

Compute Unit	Execution Count	Running Time (ms)	Intra-Kernel Dataflow Stalls (ms)	External Memory Stalls (ms)	Inter-Kernel Pipe Stalls (ms)
Filter2DKernel_1	60	43.574	2.720	0.000	0.000
Filter2DKernel_2	60	43.594	2.721	0.000	0.000
Filter2DKernel_3	60	43.541	2.721	0.000	0.000

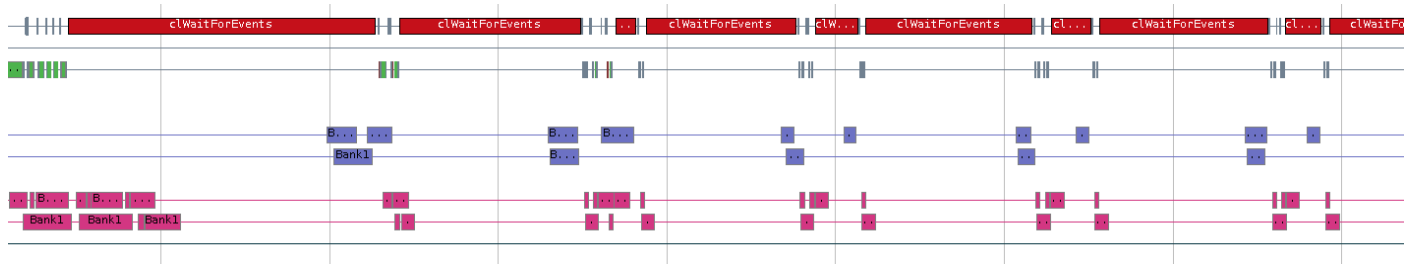
Timeline :

總共跑了 60 張圖片，擷取 timeline 的一小部分來分析

從 host 端看的波形：可以看到他有做 hostcode optimization。



可以看到他每固定一段時間就會使用 `waitForEvent` 產生同步點，使得 global memory 不會被塞滿

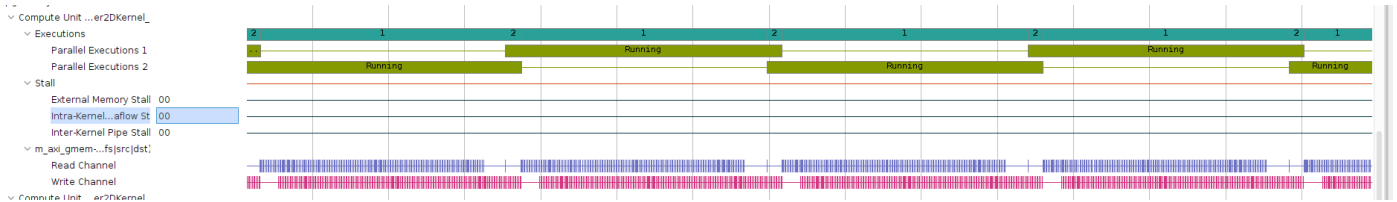


kernel 的波形

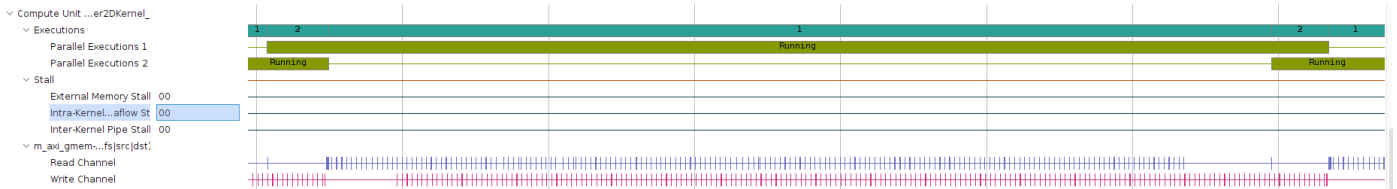
可以明顯看到有三個 kernel 在一起跑。



擷取其中一個 kernel 看：



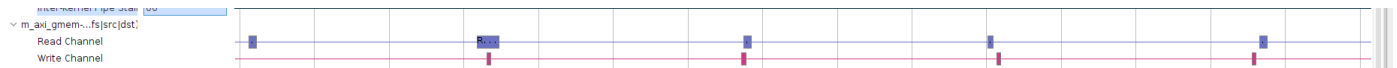
拉大一部份：



我們單看 kernel 的一次運算，下面藍色的虛線是將 pixel 讀進 kernel，紅色是寫回 memory。

把它放大：

可以發現除了運算最一開始會需要一次性讀入很多 pixel 來初始化 Linebuffer 跟 window 之外，其餘的 read write 的寬度是差不多的。這裡正是體現了把 input memory bw 和 output memory bw 保持在同一個等級這個優化，也側面證明了 line buffer 確實做好了他的職責。



總結：

這個 LABB tutorial 由一個 video filter 作為要加速的應用，從 lab1 開始：

1. 計算 performance requirements，從軟體層面來計算出整體系統所需的 throughput 等等來定義系統 SPEC。
2. 再來從 baseline implementation 分析出現有演算法的問題，從硬體層面分析整體硬體的 IO throughput(input memory bandwidth limit)並提出優化演算法做 kernel code 的 optimization 來平衡 IO 以及做增加硬體資源達到平行運算的決策。
3. LAB2 開始實際做 kernel code 優化，將 caching、linebuffer 的技巧實際帶入演算法，並實際已 Vitis-HLS 來做 kernel 效能分析。
4. LAB3 撰寫 host code，從 host code 調用三份 kernel 硬體資源，並將整體系統優化(host code optimization)來消除 PCIE 傳輸頻寬的限制。
5. 使用 Tool 做整體系統功能的分析與驗證，證實實現所需的加速應用。

從頭到尾帶我走完一個開發流程，雖然 tutorial 越後面寫得越來越水，但依然可以自己從相關報告中獲取很多的分析資訊。

學到的東西：

1. 大致上熟悉整體的應用加速開發流程。
2. 更加熟悉各個 tool 的應用
3. 了解 Line buffer 的 caching 技巧如何優化 convolution 計算。

Github 連結：<https://github.com/s095339/LAB->

[B_Convolution_Filtering/blob/main/lab1_app_introduction_performance_estimation.md](https://github.com/s095339/LAB-B_Convolution_Filtering/blob/main/lab1_app_introduction_performance_estimation.md)

課程講義

<https://basile.be/2019/03/18/a-tutorial-on-non-separable-2d-convolutions-in-vivado-hls/>