AI on Chip 2024

Final - MobilenetV3 Accelerator REPORT

Student name: _胡家豪、蔡明翰、王文楷、陳冠穎、黃雍翔
Student ID: N26122246、N26120579、N26121591、N26121622、N26122238

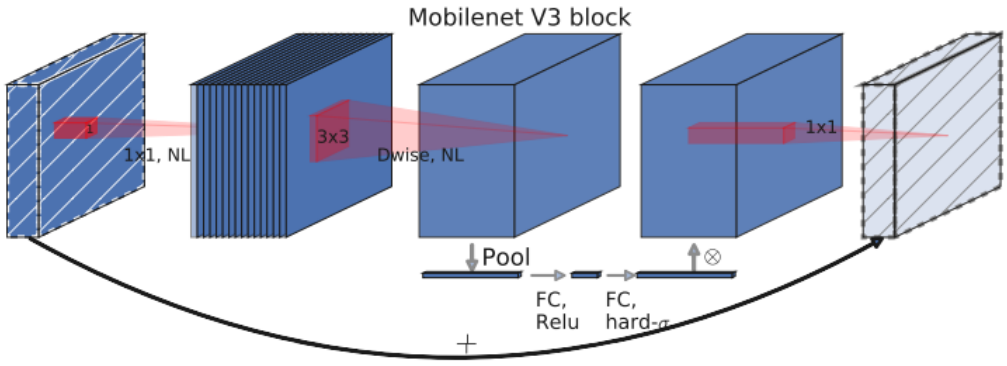
目錄

一.Software

1. Model introduction

a. MobileNetV3 overview

本次實作中，我們以 mobilenetV3 為目標進行實作，其主要是由 pointwise convolution 進行運算，並且使用了 SE layer 與 h swish 增加精確度。

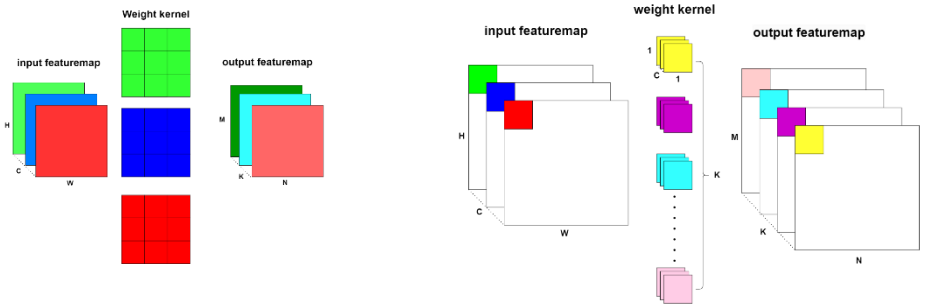


在原有的架構中，一個 Mobilenet V3 block 的流程為：pointwise conv -> depthwise conv -> SE layer -> pointwise conv。

原始的 MobileNet V3 中，總共含有 11 個這樣的 block ，並且每一層這樣的 block 都可以自由選擇是否要用 H-swish 或是 ReLU 作為 activation function 以及是否需要 SE layer。

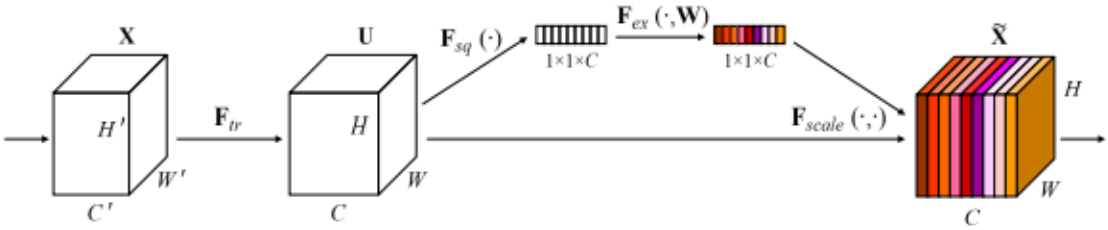
b. Pointwise and Depthwsie

作者使用了 Pointwise 加上 Depthwise 來取代原有的 Convolution，下方左圖為 Depthwise conv 之示意圖、右側則為 Pointwise 之示意圖



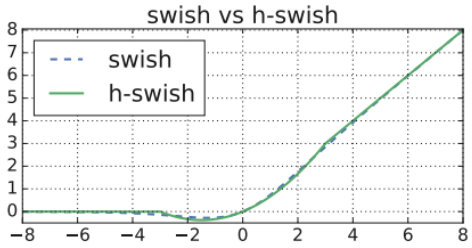
作者透過使用 Depthwise 作為提取特徵、使用 pointwise 作為特徵混和，以此來取代原本的 Convolution。

c. SE layer



SE layer 可以看做是圖片的 attention 機制，將原有的圖片藉由訓練乘上一個權重來得知哪個 channel 比較重要。

d. H-swish

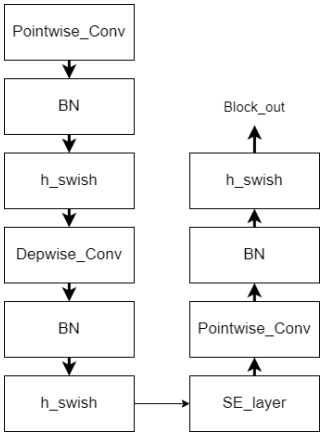


Hswish 是本篇用於取代原有 ReLU 的 activation function，用於解決 ReLU 在負數部分消失的狀況，以此增加精確度。

2. Model Training

由於原本的 Mobilenet 在模型建立上有較高的自由性，為了硬體的設計方便，我們將自己設計了一個 Mobilenet V3 Like 的模型，與原有模型的差異在於我們的 block 每一層的內容都是固定的，在架構比較固定的狀況下比較好映射上硬體。

右側是我們的 block 架構，與原本的 mobilenet block 接近，差異是我們直接固定這個架構在每一層，而不能在特定層決定 activation 與要不要 SE




```
class Our_MobileNetV3_have_bias(nn.Module):
    def __init__(self):
        super(Our_MobileNetV3_have_bias, self).__init__()
        self.conv1 = nn.Conv2d(in_channels=1,out_channels=8,kernel_size=3,stride=1,padding=1,bias=False)
        self.block1 = MobileNetV3_block(infp = 8, outfp=16 , middle_feature=8 ,kernel_size=3,stride=2,padding=1,bias=True)
        self.block2 = MobileNetV3_block(infp = 16, outfp=32 , middle_feature=48 ,kernel_size=3,stride=2,padding=1,bias=True)
        self.block3 = MobileNetV3_block(infp = 32, outfp=32 , middle_feature=64 ,kernel_size=3,stride=2,padding=1,bias=True)
        self.block4 = MobileNetV3_block(infp = 32, outfp=48 , middle_feature=64 ,kernel_size=3,stride=2,padding=1,bias=True)
        self.block5 = MobileNetV3_block(infp = 48, outfp=64 , middle_feature=96 ,kernel_size=3,stride=2,padding=1,bias=True)
        self.block6 = MobileNetV3_block(infp = 64, outfp=64 , middle_feature=96 ,kernel_size=3,stride=2,padding=1,bias=True)
        self.block7 = MobileNetV3_block(infp = 64, outfp=32 , middle_feature=48 ,kernel_size=3,stride=2,padding=1,bias=True)
        self.avgpool = nn.AdaptiveAvgPool2d(1)
        self.linear1 = nn.Linear(32,20,bias=False)
        self.ReLU = nn.ReLU(inplace=True)
        self.linear2 = nn.Linear(20,10,bias=False)
```

上圖是我們整體架構，我們主要使用了 7 個 block 進行模型建立。

3. Quantization and Hardware mapping

a. BN Fold

在模型訓練完畢，已經在 inference 的階段，由於 BN 層的參數均已經固定，所以我們可以將其透過下列公式併入前一層的 Conv layer

$$\mathbf{y}_k = \mathrm{BatchNorm}(\mathbf{W}_{k,:}\,\mathbf{x})$$

$$= \gamma_k \left(\frac{\mathbf{W}_{k,:}\,\mathbf{x} - \mu_k}{\sqrt{\sigma_k^2 + \epsilon}} \right) + \beta_k \qquad \widetilde{\mathbf{W}}_{k,:} = \frac{\gamma_k \mathbf{W}_{k,:}}{\sqrt{\sigma_k^2 + \epsilon}},$$

$$= \frac{\gamma_k \mathbf{W}_{k,:}}{\sqrt{\sigma_k^2 + \epsilon}} \mathbf{x} + \left(\beta_k - \frac{\gamma_k \mu_k}{\sqrt{\sigma_k^2 + \epsilon}} \right) \qquad \widetilde{\mathbf{b}}_k = \beta_k - \frac{\gamma_k \mu_k}{\sqrt{\sigma_k^2 + \epsilon}}.$$

$$= \widetilde{\mathbf{W}}_{k,:}\,\mathbf{x} + \widetilde{\mathbf{b}}_k$$

在實作上，我們建立了一個專門用來 folding BN 的 function，並且遍歷我們的 model 來進行 BN fold

```python
def bn_folding(conv_w, conv_b, bn_rm, bn_rv, bn_eps, bn_w, bn_b):
    if conv_b is None:
        conv_b = bn_rm.new_zeros(bn_rm.shape)
    bn_var_rsqrt = torch.rsqrt(bn_rv + bn_eps)

    w_fold = conv_w * (bn_w * bn_var_rsqrt).view(-1, 1, 1, 1)
    b_fold = (conv_b - bn_rm) * bn_var_rsqrt * bn_w + bn_b

    return torch.nn.Parameter(w_fold), torch.nn.Parameter(b_fold)
```

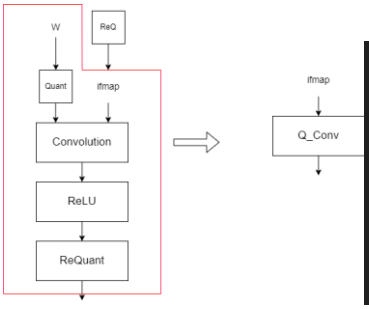
```python
def bn_folding_model(model):

    new_model = copy.deepcopy(model)

    module_names = list(new_model._modules)

    for k, name in enumerate(module_names):
        if len(list(new_model._modules[name]._modules)) > 0:
            new_model._modules[name] = bn_folding_model(new_model._modules[name])


        else:
            if isinstance(new_model._modules[name], nn.BatchNorm2d):
                if isinstance(new_model._modules[module_names[k-1]], nn.Conv2d):

                    # Folded BN
                    folded_conv = fold_conv_bn_eval(new_model._modules[module_names[k-1]], new_model._modules[name])

                    # Replace old weight values
                    new_model._modules.pop(name) # Remove the BN layer
                    new_model._modules[module_names[k-1]] = folded_conv # Replace the Convolutional Layer by the folded version
    return new_model
```

b. Normal Quantization vs Our Quantization

在 Inference-only 這篇的 PTQ 說道，其量化可以透過以下方式進行

$$q_3^{(i,\,k)} = Z_3 + M \sum_{j=1}^{N} \left(q_1^{(i,\,j)} - Z_1 \right) \left(q_2^{(j,\,k)} - Z_2 \right),\ M := \frac{S_1 S_2}{S_3}$$

透過上述公式，可以將其圖像如下：



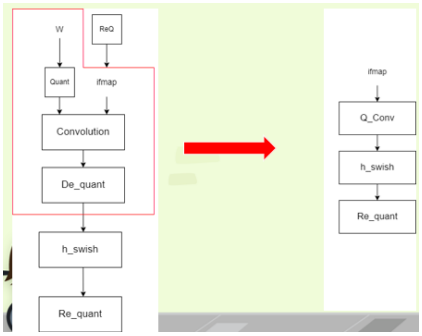

```python
def quantized_linear(input, weights, input_scale, weight_scale, output_scale, input_zero_point, we
    input, weights = input.to(device), weights.to(device)

    ######################################################

    M = input_scale * weight_scale / output_scale
    output = torch.nn.functional.linear((input - input_zero_point ), (weights - weight_zero_point))
    output *= M
    output += output_zero_point

    ######################################################

    #clamp and round
    output = output.round().clamp(-(2**(activation_bitwidth-1)-1), 2**(activation_bitwidth-1)-1)

    return output
```

他可以這樣做是因為一般模型都是使用 ReLU 進行，而 ReLU 在正的部分其實就是 Linear 的。

如果要將此方法應用到我們的模型上並不能直接應用，原因是因為 hswish 其實並不是一個線性函數，因此我們可以將 PTQ 的量化公式寫成如下形式：

$$q_3^{(i,\,k)} = \frac{1}{S_3}(Z_3 + hswish(\, M \sum_{j=1}^{N} \left(q_1^{(i,\,j)} - Z_1\right)\left(q_2^{(j,\,k)} - Z_2\right) \,)),\ M := S_1 S_2$$

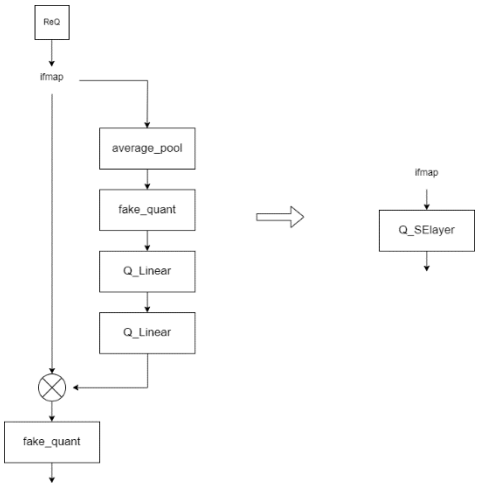
由此可知，Inference-only 的步驟我們必須先進行 DeQuant，經過 hswish 後，再進行 ReQuant



流程如左圖表示。

c. SE layer Quantization

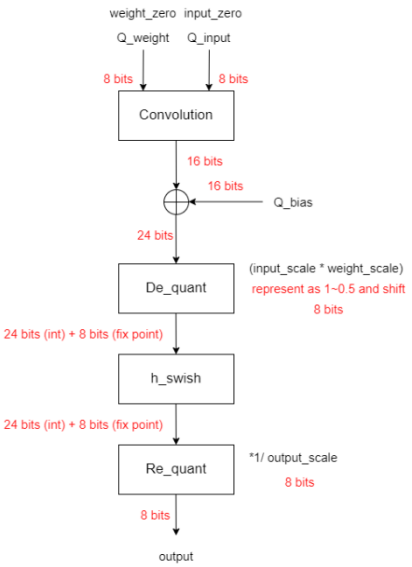
在 SE layer 中，由於有相乘與 pooling 的運算，所以也必須特別處理，下圖是我們的處理方式，在 pool 的後方與相乘的後方加入 fake Quant(或稱 Requant)，而中間的 Linear 則使用一般的 QLinear




```
class Q_SELayer(nn.Module):
  def __init__(self,weights1,input_scale1,weight_scale1,output_scale1, input_zero_point1, weight_zero_point1, output_zero_point1,
               weights2, input_scale2, weight_scale2, output_scale2, input_zero_point2, weight_zero_point2, output_zero_point2,
               input_SE_scale,in_SE_zero_point,
               output_SE_scale,output_SE_zero_point,
               out_pool_scale,out_pool_zero_point):
    super().__init__()
```

4. Hardware mapping

完成量化後，就可以開始針對量化 flow 將需要的運算映射到硬體上面，參考下面的圖，說明在圖片右側：



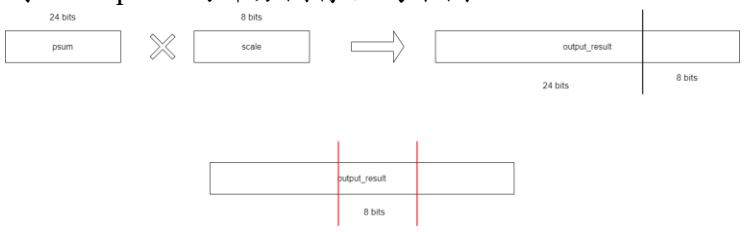
Software preprocess：由於我們的量化有 zero point，所以會在軟體先把需要與 zero point 操作的部分先做好，例如 Q_wieght 與 Q_input，input_scale 與 weight_scale 的相乘，1/output_scale 等

PE array：在 PE array 中主要是執行 Convolution 與 psum 還有 bias 的相加

Postprocess：主要執行 De_quant 與 activation function 與 requant。

- 首先 Convolution 的運算會輸入量化為 8 bits 的權重與輸入，其結果為 16 bits
- 用 24 bits 累加 psum 與 bias
- 進行 Dequant ，也就是乘上 dequant scale，在這邊我們將 scale 表示乘一個 8 bits 的 fix point，其小數點點在最左邊，並且記錄需要的位移。Dequant 的輸出是一個 24bit 整數與 8bit 小數的 fixed point，這個數往右上面的位移數就是結果
- 經過 swish 與 requant，由於 1/output_scale 通常是整數，所以直接乘就可以
- 最後的結果 truncate 掉小數部分即為解答。

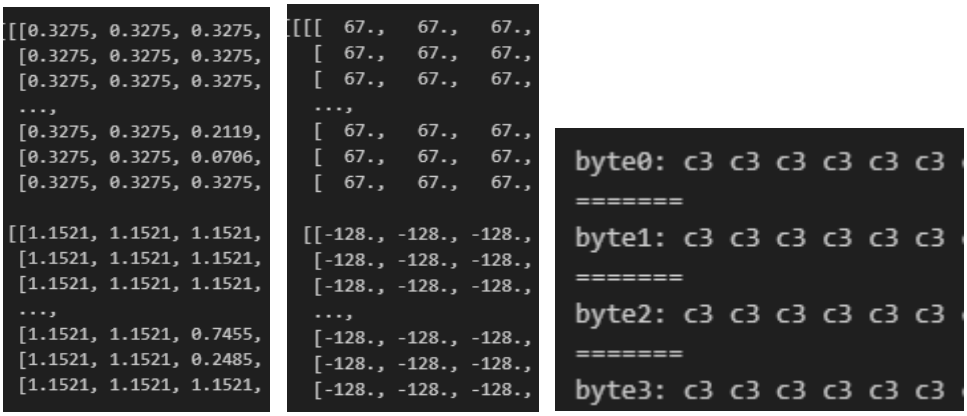
對於 dequant 的部分圖像化為下圖：



5. Golden generation

完成上述部分後，就可以將 model 的 input 以及 weight 轉換成硬體所需要的形式，我們的硬體 input 與 weight 存放在 SRAM 當中，由於使用的 32 位元 CPU 支援 LH 等指令，所以需要一個 Byte 一個 Byte 的存。下面左圖

是量化前的輸入、中圖是量化後的輸入、右圖則是將輸入轉換成硬體可以讀取的十六進位形式。



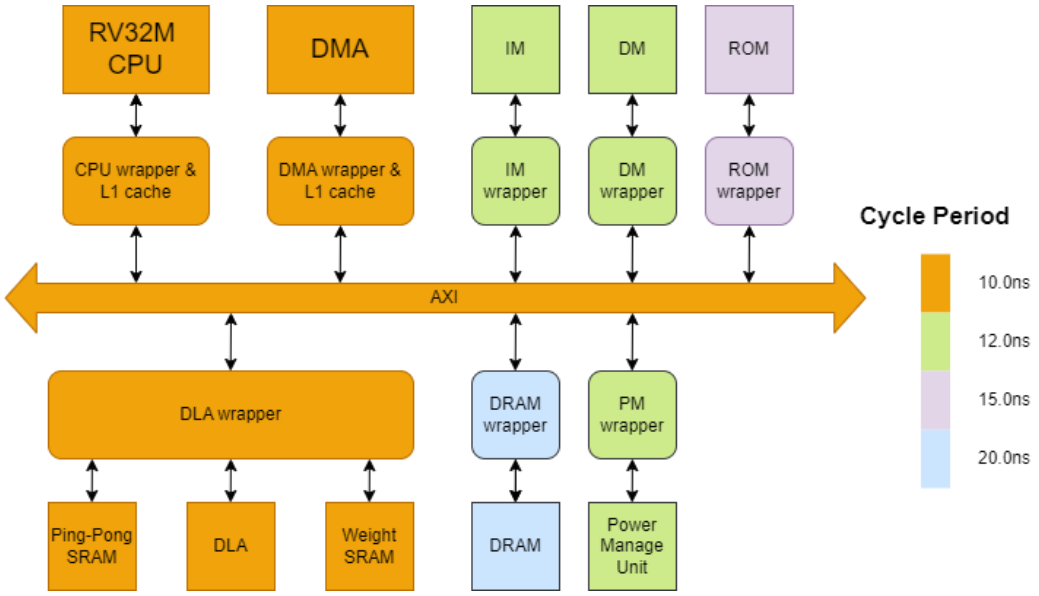
6. Software Result

下面是我們的軟體結果，可以看到我們自己搭建的模型得到了 88.1%的精確度；BN_fold 的模型也獲得同樣的精確度，以此驗證我們的 BN 方法是正確的、最後是我們的模型經過 PTQ 的精確度，由於模型較深且沒有用 QAT 的方式 train，所以最後量化誤差會不斷累積導致精確度下降。因此量化後的模型精確度為 70.1%，還在堪用的範圍內。

Model	accuracy
Our-mobilenet	88.1%
BN_fold_our_mobilenet	88.1%
Q_BN_fold_our_mobilenet	70.1%

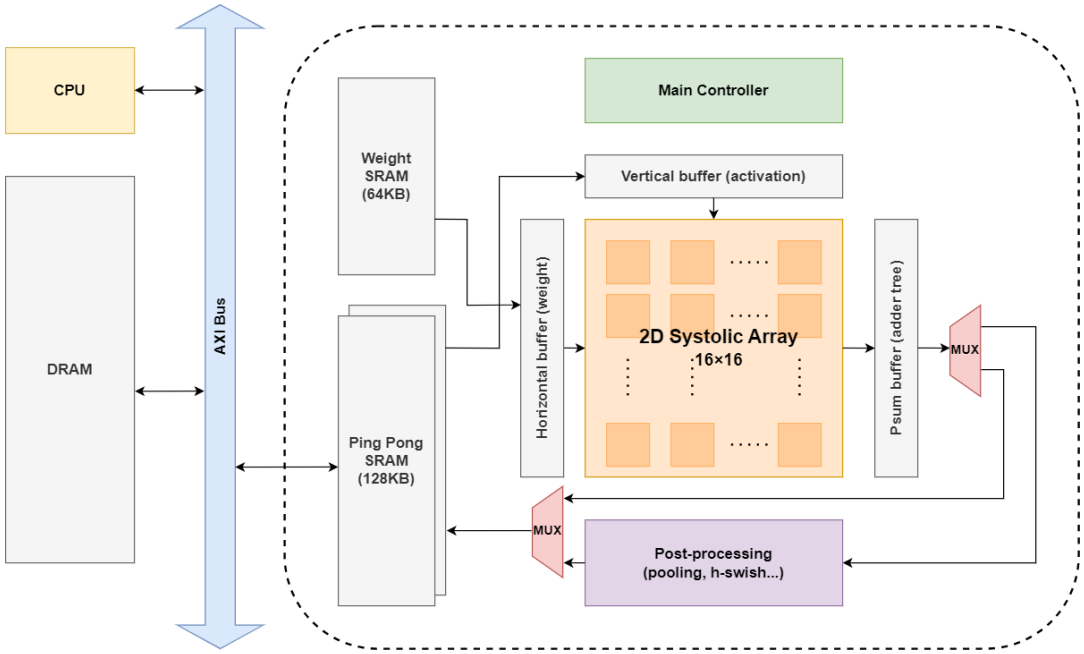
二. Hardware

1. System Overview



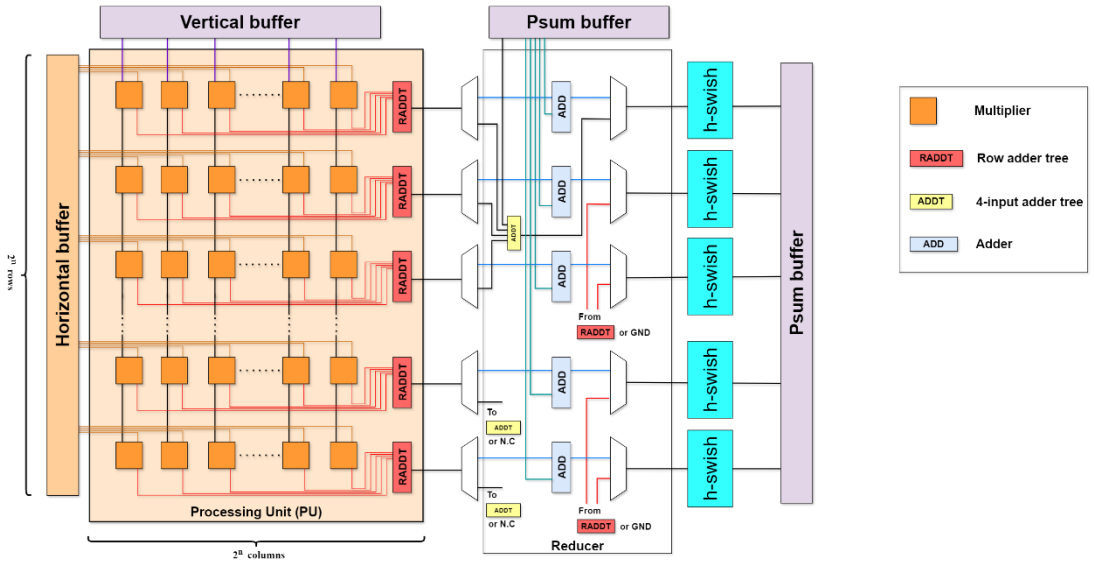
整個系統的流程，CPU 會透過 DMA 將 input data 和 weight data 從 DRAM 分別搬到 DLA 的 ping-pong SRAM 和 weight SRAM。搬完資料後，會寫入特定 address 給 DLA_wrapper，DLA 則會開始進行運算，CPU 則會閒置，不斷去讀另一個特定 address。DLA 運算結束後，會在該特定位址寫值，CPU 則會在該位址讀到數值後，開始將 ping-pong SRAM 的資料搬回 DRAM。另外，power management unit，會判斷哪些 component 處於閒置狀態，降低其操作電壓，以減少功率消耗。

2. System Architecture



整個 DLA 的架構包含用來儲存 weight 的 weight SRAM，還有 Ping-Pong SRAM 用來儲存 input 跟 output feature map，我們會透過加速器的 main controller 控制 vertical buffer 和 horizontal buffer 分別讀取 SRAM 中的 ifmap 和 weight 資料，並且將資料排列成加速器支援計算的格式。

3. Network on Chip



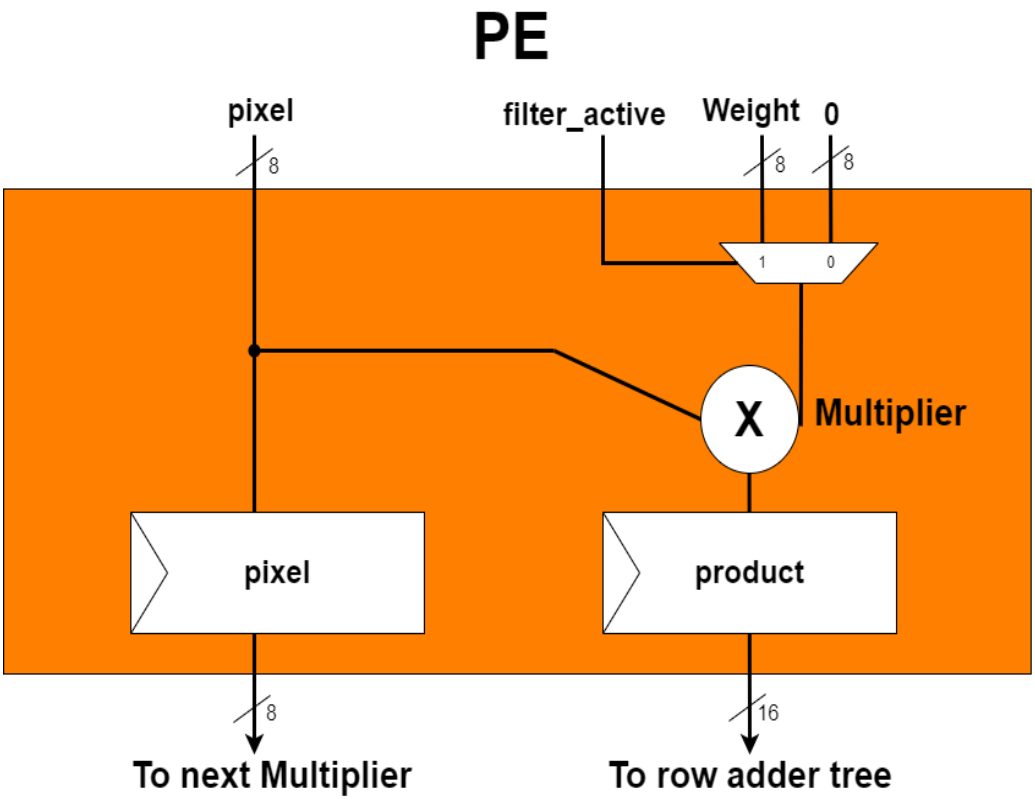
Vertical buffer 中的 input feature map 資料會以 systolic 的方式由上往下傳遞；horizontal buffer 中的 weight 資料則會直接 broadcast 到 PE array 的每個 PE 上。每個 row 的 psum 結果會透過 row adder tree 進行相加，若是 depthwise covolution 的模式下，會將每三個 row adder tree 的 output 在經過一個 adder tree 進行相加。這些 psum data 接著會送入後面的 quantizer、h-swish、re-quantizer，對資料做進一步的處理後，將 psum data 存入 psum-buffer。

DLA Storage size

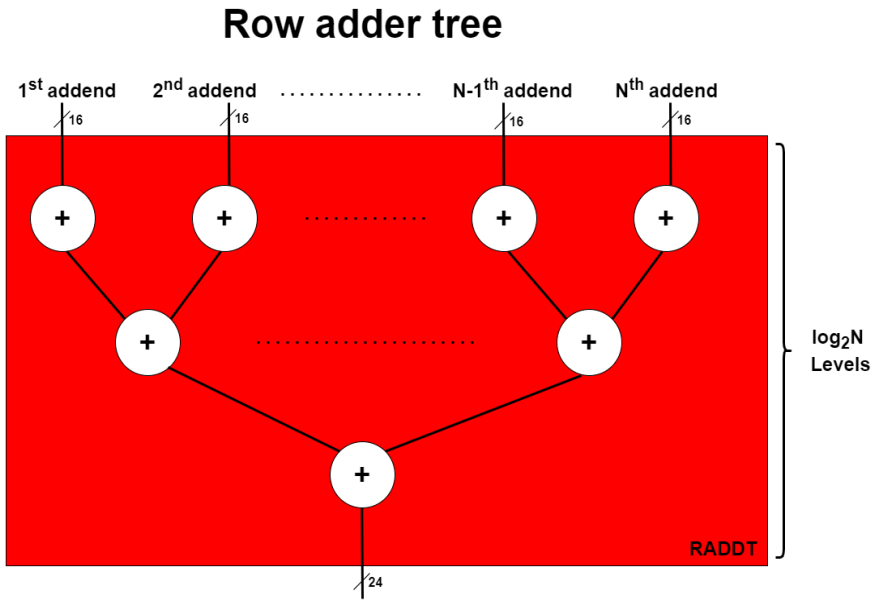
Register	Size
Vertical Buffer	64 Byte
Horizonal Buffer	1024 Byte
PE Array Registers	728 Byte
Psum Buffer	8192 Byte
Pipeline Registers	96 Byte
Total	9.8671875 KB

SRAM	Size
Ping-Pong SRAM	64 Byte
Horizonal Buffer	1024 Byte

4. Processing element

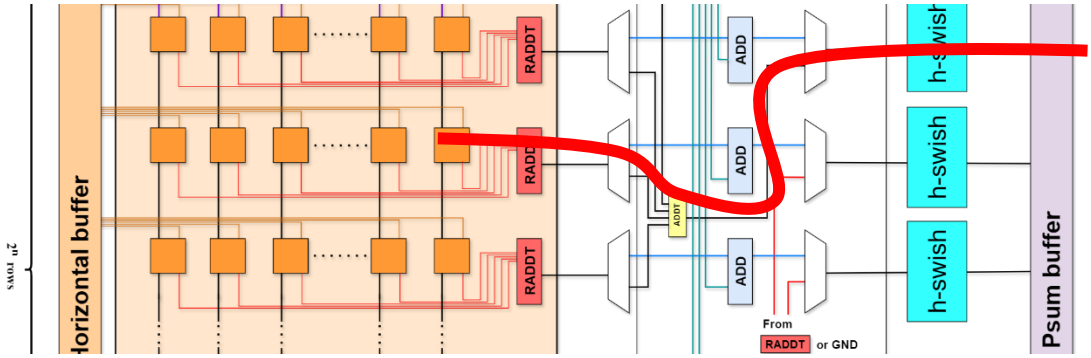


每個 PE 包含 1 個 8 bits * 8 bits 的有號數乘法器、1 個 8 bits registe 用來傳送 ifmap 資料到下一個 PE 和 1 個 16bits 的 register 暫存乘法器的 output 以減少 critical path。並且在 weight input 有一個 filter active 訊號，這個遮蔽設計讓我們的加速器能兼容更多種 kernal size 大小的 convolution 以及避免 row adder tree 有錯誤的非 0 訊號輸入。



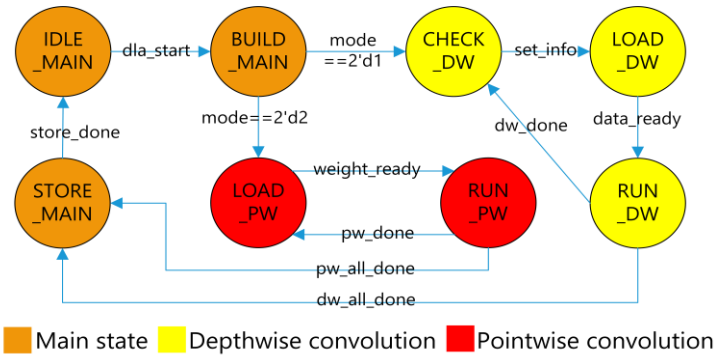
因為 PE array 大小為 16*16，一個 row adder tree 需要將 16 筆 16bits 的資料進行累加，故 adder tree 架構需要 4 層。

Critical Path



根據合成時 timing analysis 的 report 結果，我們發現電路的 critical path 是在 adder tree 到 h-swish，為了解決這個問題，我們決定將這段路徑切成 2 stage pipeline 的架構，將 delay time 壓到 10ns 以下，並成功合成。

5. Finite state machine diagram

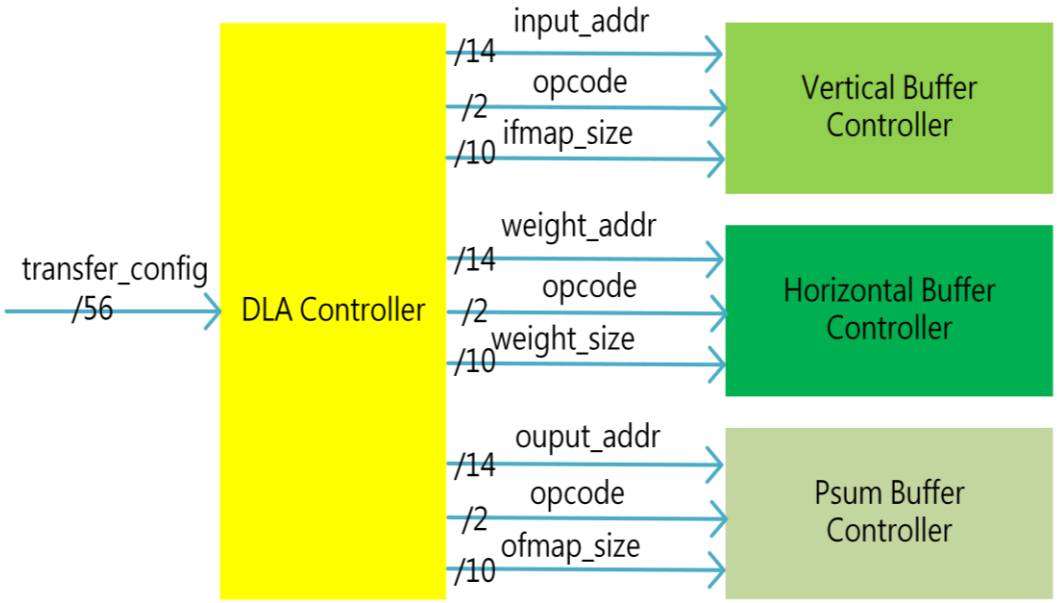


當 DLA 接收到 dla_start 訊號，會進入 build 模式，並將該層 layer 運算的資訊給各個 controller，包括 ifmap、weight、ofmap 的起始 addr、ifmap, ofmap & kernal 的 size、進行運算的模式。之後根據當下是要進行 depthwise convolution 還是 pointwise convolution 進入對應的 state。

Depthwise convolution: 在 state CHECK_DW，檢查和設定該次運算的資訊到 control registers。設定好資訊後，進入 LOAD state，depthwise convlotuiun 不僅需要等待 weight data ready，同時需要等待 vertical buffer 是否準備好一個 kernal size 的資料，這邊我們是直接使用 fifo 的 full 訊號，以簡化設計。只有當 vertical buffer 和 horizontal buffer 都發出 data_ready 訊號後，才會進入運算的 state RUN_DW，將資料送進 PE array 內，並將計算完的 ofmap 存入 psum buffer。RUN_DW 運算時，會檢查計算結果的次數和 ofmap size，以拉起結束訊號 dw_all_done，進入 state STORE_MAIN。

Pointwise convolution: 因為我們有做 pipeline PE array 的設計，可以容許 vertical buffer 沒有準備好的情況，在 LOAD_PW 僅需要確認 horizontal buffer 有準備好，就會進入運算 state RUN_PW，邊載入資料和運算，並將結果存入 sum buffer。若有超過 16 channels 需要計算，則會回到 LOAD_PW，準備好 weight，在進入 RUN_PW。RUN_PW 運算時，會檢查計算結果的次數和 ofmap size，以拉起結束訊號 pw_all_done，進入 state STORE_MAIN。STOE_MAIN: 將之前存在 psum buffer 的資料寫入 ping-pong sram。

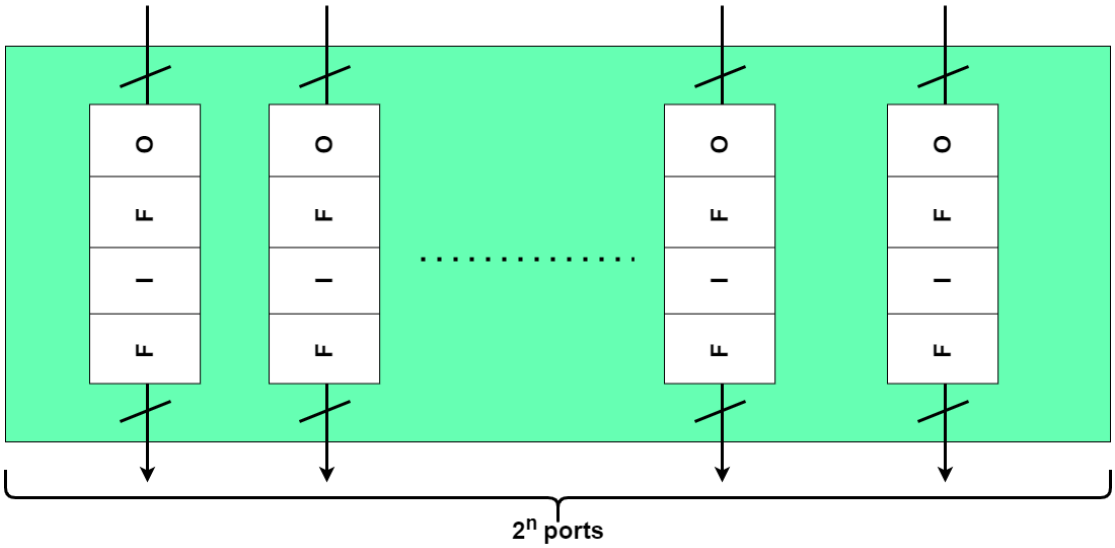
6. Programable Controller



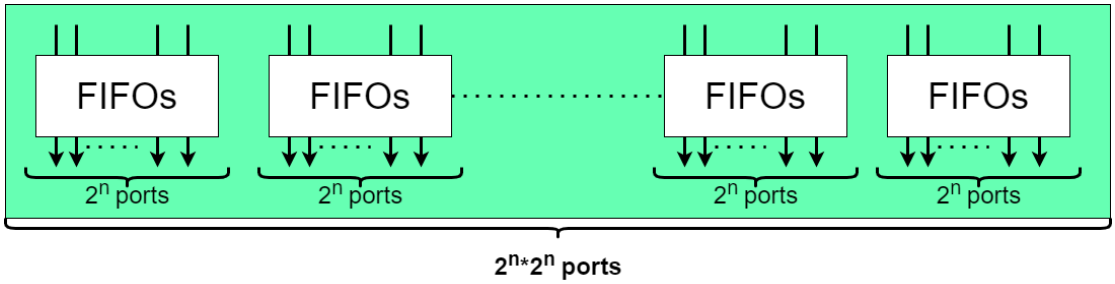
因為 DLA 內部的 module 是多人進行開發的緣故，所以有 4 個 controller 分別去控制 PE array、vertical buffer、horizontal buffer 和 psum buffer，但最主要是控制 PE array 的 DLA controller 會先從 config register 讀取 transfer_confog 即該層 layer 的資訊，並將對應的資訊分派給各個 controller，以 generalize 不同大小的和模式的 convolution，達到 programable 的效果。

7. Horizontal buffer

a. FIFO



b. horizontal buffer



horizontal buffer 架構如圖所示，a 圖的 FIFO 總共有 16 個,b 圖的 FIFOs 的內部架構則是 a 圖，b 圖總共會有 16 個 FIFOs，所以我們的 Horizontal buffer 總共有 256 個 FIFOs，以對應 256 個 PE array，horizontal buffer 的功能是將 filter 的值從 memory 讀出以後暫存在 buffer 內，當 filter 所需的值都從 memory 搬到 buffer 後，Horizontal buffer 會向 PE Array 發出 ready 信號，當 PE Array 接收到 ready 信號以後，再根據目前所要執行的 convoltion 方式，向 horizontal buffer 發出 valid 信號，此時 PE Array 就可以讀取所需要的值。

8. Vertical buffer

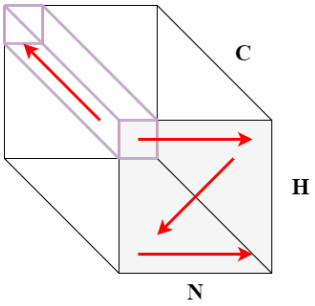


Fig1. Pointwise data access order

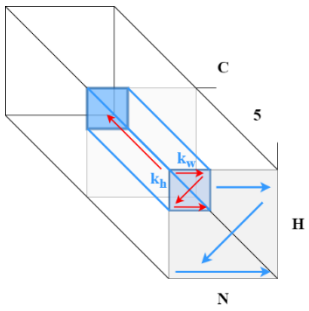


Fig2. depthwise data access order

Vertical controller 主要的功能是將 ifmap 從 ping-pong SRAM 拿出來，並且將資料排列成 PE array 支援計算的格式，存入 vertical buffer 中。

以下說明 vertical controller 至 SRAM 拿資料的順序，因為要計算 pointwise 或是 depthwise 取資料與排列資料的方式都不一樣 ，因此一開始 vertical controller 需要接收到這層要計算的 layer 資訊。

首先是 pointwise，ifmap 會以 NHWC 的格式存放在 SRAM 當中，而 controller 會先從 channel 方向開始拿資料，如 Fig1.的紫色方塊所示，因為我們的 PE array 大小為 16*16，一次可以對 ifmap 的 16 個 channel 做計算，因此一次會先拿 16 個 channel 的資料出來，並且以列方向排列，存入 vertical buffer 中，接著會往 ifmap 的寬方向移動，一樣取 16 個 channel 的資料排列好存入 vertical buffer，最後才往寬方向移動，重複上述步驟直到移動完整個 ifmap 平面，會再往後 16 個 channel 取資料，直到拿完所有的 input feature map。

再來介紹 depthwise，ifmap 存放在 SRAM 的格式與 pointwise 不同，是以一個 convolutional block 為單位來存放，一個 convolutional block 又會以 NCHW 的方式儲存。因為我們 PE array 支援一次對 5 個 convolution kernel 做計算，因此 Controller 拿資料也是以 convolutional block 為單位，差別在於每個 convolutional block 的 channel 為 5，如 Fig2. 的藍色方塊所示。針對 convolutional block 的資料存取順序則會先以寬度方向再往 channel 方向，最後則是高度方向，這樣的順序，與 PE array 如何計算 depthwise 有關，當計算完一次 depthwise，會再往寬度方向移動 stride 的大小，存取下一塊 convolutional block，寬度方向都計算完畢，再往高度方向移動，重複上述步驟直到計算完整個 ifmap 的平面後，會再往後 5 個 channel 做存取，直到拿完所有的 input feature map。

9. Memory mapping

a. Memory mapping-pointwise convolution

Filter1							
a8	a7	a6	a5	a4	a3	a2	a1
Filter2							
b8	b7	b6	b5	b4	b3	b2	b1
Filter3							
c8	c7	c6	c5	c4	c3	c2	c1
Filter4							
d8	d7	d6	d5	d4	d3	d2	d1
Filter5							
e8	e7	e6	e5	e4	e3	e2	e1
Filter6							
f8	f7	f6	f5	f4	f3	f2	f1
Filter7							
g8	g7	g6	g5	g4	g3	g2	g1
Filter8							
h8	h7	h6	h5	h4	h3	h2	h1

我們硬體實作的部分為 pointwise convolution 和 depthwise convolution,在 pointwise convolution 的部分，我們採用的是 channel first，即把 filter1 的第一個 channel(a1)放在 memory 第一個位置，filter1 的第二個 channel(a2)放在 memory 第二個位置，假設第一個 filter 有 8 個 channel，那 memory 的第 8 個位置就會是 filter1 的第 8 個 channel 的值，接者再放 filter2 第一個 channel 的值，以此類推，範例如下圖所示，featue map 也是按照 channel first 的方法排列。

b. Memory Mapping–Depthwise Convolution

Channel 1(3x3)		
a1	a2	a3
a4	a5	a6
a7	a8	a9

Channel 2(3x3)		
b1	b2	b3
b4	b5	b6
b7	b8	b9

Channel 3(3x3)		
c1	c2	c3
c4	c5	c6
c7	c8	c9

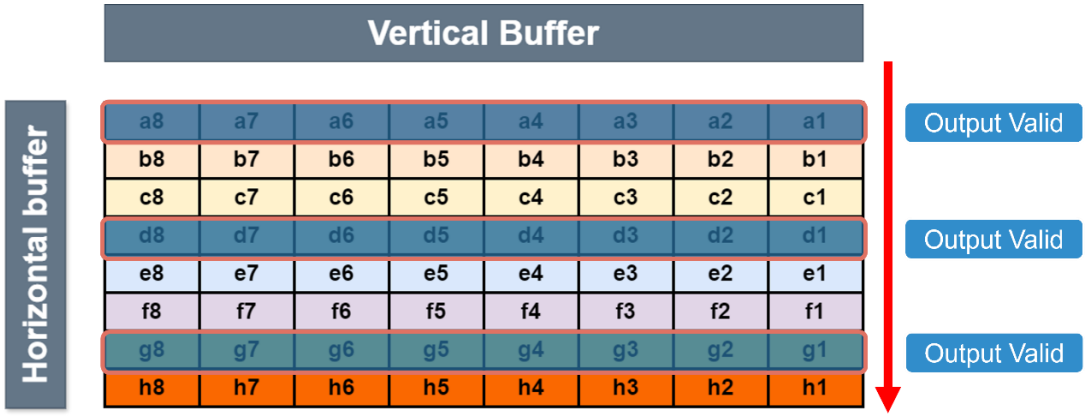
PE Array								
0	0	0	0	0	0	a9	a8	a7
0	0	0	0	0	0	a6	a5	a4
0	0	0	0	0	0	a3	a2	a1
0	0	0	b9	b8	b7	0	0	0
0	0	0	b6	b5	b4	0	0	0
0	0	0	b3	b2	b1	0	0	0
c9	c8	c7	0	0	0	0	0	0
c6	c5	c4	0	0	0	0	0	0
c3	c2	C1	0	0	0	0	0	0

而 depthwise convolution 部分，我們使用的方法為 row major，我們排放的順序為 a1~a9,b1~b9,c1~c9…..以此類推,如下圖所示，feature map 的排放方式也是 row major。

10. Improvement:

相對於參考論文的架構，我們在架構上添加了一些新的設計:

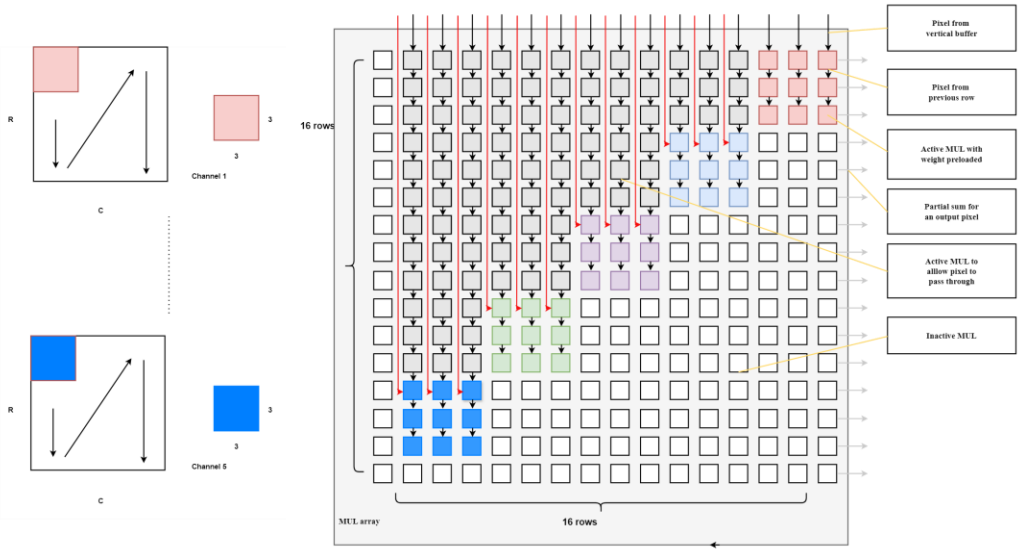
a. Ifmap data bubble tolerate design



首先是 bubble tolerate design，無論是 Pointwise 或 Depthwise Convolution，在 vertical buffer 準備資料時，我們的 PE 是能夠 pipeline 將 ifmap 往下傳遞做計算的 (上圖紅色箭頭為傳遞方向)，但在 ifmap 往下傳遞的同時會有 bubble 出現，也就是在傳遞真正需要的 data (上圖中間 3 個藍色矩形框起來處) 時會有不必要的 data 也一起往下傳遞進行運算並輸出，因此我們在電路上添加了 output valid 這條訊號線。

我們將 output valid 與資料一起往下作傳遞，這樣做除了能夠準確地將真正所需的 data 做輸出，還能避免輸出 bubble 算出來的 data，並且達到 pipeline 的效果。

b. Optimize Depthwise Convolution - Shortcut Strategy

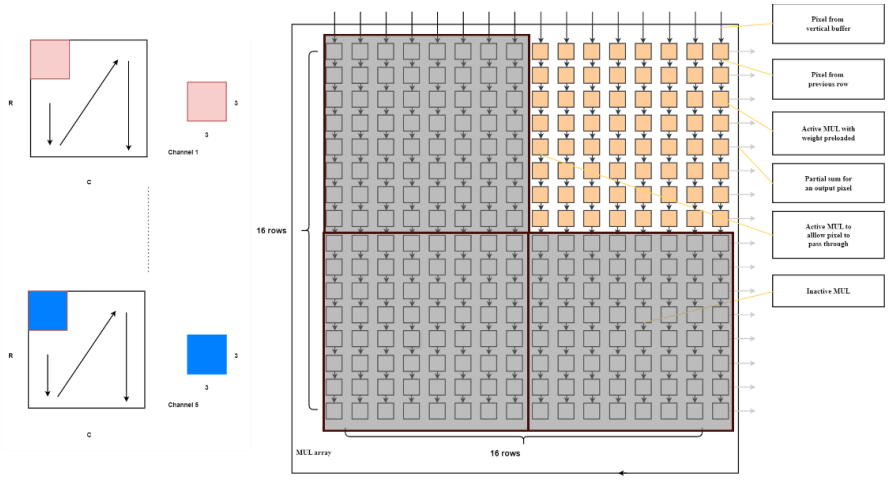


在原始論文中，PE 之間傳遞 data 的方式是由上往下作傳遞，這在做 Depthwise Convolution 時，會花費較多的 clock cycle 數準備資料，因為此架構在 Depthwise Convolution 時須等到每個所需的 PE 都準備好資料才會做運算並輸出。比如上圖 PE array 裡藍色 3*3 大小矩形的最下方列需要共 15 個 cycles 才能將資料從最上方的 input 傳遞到該位置做運算並輸出。

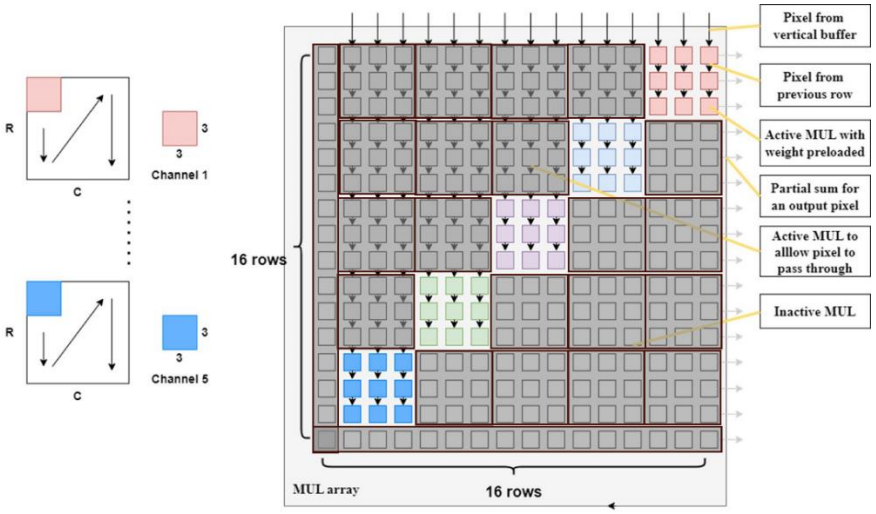
因此我們在架構上使用了 Shortcut Strategy (PE array 裡紅色箭頭)，因為添加了 short cut 的電路，讓原本花 15 cycles 的傳輸時間減少為 3 個 cycles，大幅減少在 Depthwise Convolution 搬運資料所需的時間。

c. Weight active design

- Pointwise Convolution:



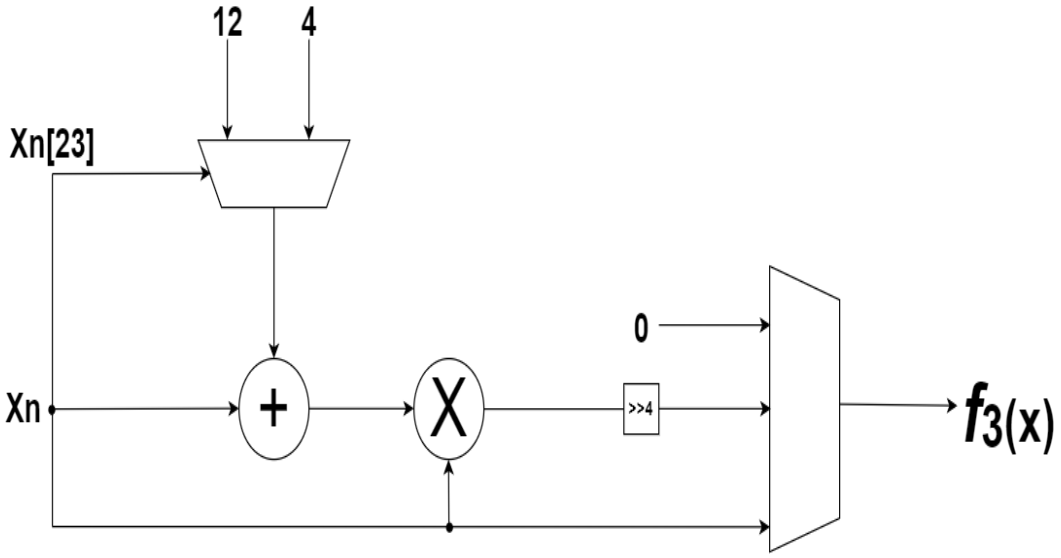
- Depthwise Convolution:



在我們的架構中，會使用 Row adder tree 將 PE array 每列 16 個 data 做加總，但若該列裡有未使用的 PE，會導致 Row adder tree 連同不需要的 data 也一起加總。因此我們添加了 Weight active 的設計，這個設計能夠將沒有使用的 PE 的 Weight 設為 0，因為 weight 為 0 與任何 ifmap 做乘法都為 0，這麼做能夠避免 Row adder tree 將不必要的 data 也一起加總，上方兩張分別為 Pointwise Convolution 在 ifmap channel = 8、ofmap channel = 8 與 Depthwise Convolution 的 Weight active design (灰色區域為 Weight active)。

11. h-swish module

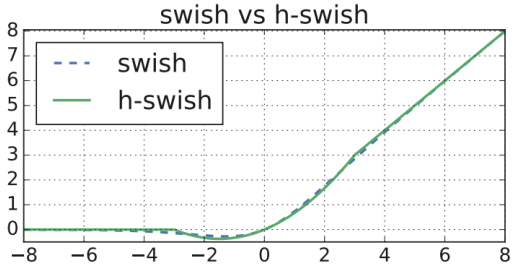
a. Approximation h-swish Architecture

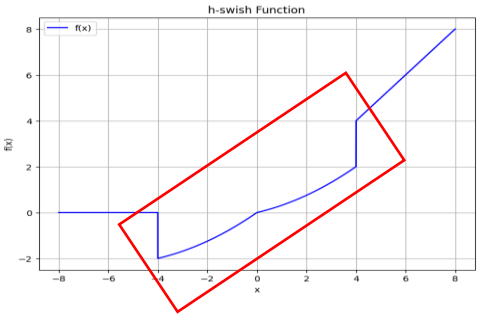


b. h-swish approximation formula

$$f_3(x) = \begin{cases} 0 & (x \le -4) \\ \dfrac{12x + x^2}{16} & (-4 < x < 0) \\ \dfrac{4x + x^2}{16} & (0 \le x < 4) \\ x & (x \ge 4) \end{cases}$$

c. h-swish vs approximation h-swish





我們實作的 h-swish 架構是參考 K. Choi, S. Kim, J. Kim and I. -C. Park, "Hardware-

Friendly Approximation for Swish Activation and Its Implementation" 這篇論文的作法，但我們沒有注意到本篇論文使用的數學近似式和原本的 h-swish 相差非常大，導致我們做出來的結果無法和軟體的結果相匹配，我們太晚才發現這個問題，來不及更正，最直接的改善方法就是如同老師給的建議:使用 look up table 就能解決了!

三. Analysis

1. Computation Utilization

Input data size: 8*8*1, kernal 大小均是 3*3

Layer	Computation type	Ofmap size	Utilization
1	Standard convolution	8*8*8	14.0625 %
2	Pointwise convolution	16*16*8	25 %
2	Depthwise convolution	16*16*8	14.0625 %
3	Pointwise convolution	32*32*48	50 %
3	Depthwise convolution	32*32*48	16.875 %
4	Pointwise convolution	32*32*64	100 %
4	Depthwise convolution	32*32*64	17.307 %
5	Pointwise convolution	48*48*64	100 %
5	Depthwise convolution	48*48*64	17.307 %
6	Pointwise convolution	64*64*96	100 %
6	Depthwise convolution	64*64*96	16.875 %
7	Pointwise convolution	32*32*48	100 %
7	Depthwise convolution	32*32*48	16.875 %

2. Performance

Clock period	10 ns
Area	13.3909 mm^2
Power	261.9147 mW

pointwise convolution 執行 cycles，實際和理想比較，以一層運算為例。

Test : ifmap size 28*28*8，ofmap size 26*26*8

Max utilization: ifmap size 28*28*16，ofmap size 26*26*16

Pointwise	Test ideal	Max utilization	Test real
Cycles	1576	3152	6287

depthwise convolution 執行 cycles，實際和理想比較，以一層運算為例。

Test: ifmap size 30*30*8，stride 3，ofmap size 10*10*8

Max utilization: ifmap size 30*30*10，stride 3，ofmap size 10*10*10

depthwise	Test ideal	Max utilization	Test real
Cycles	3200	3200	9425

BUS transfer: 75.2MB/s

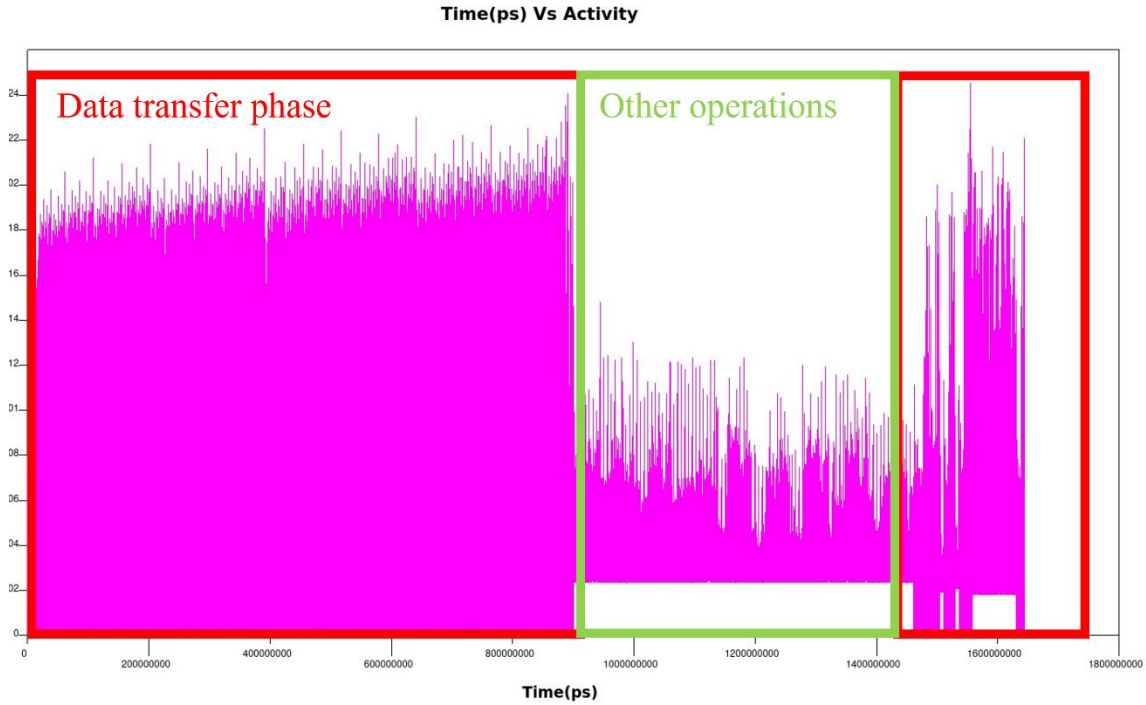
MAC：800M FLOP (28*28*8*8 MAC in 62875 ns)

BUS transfer: 400MB/s(Best case)

MAC：1600M FLOP (28*28*8*16 MAC)

I=75.2×106800×106=75.2800≈10.64 FLOP/byte

3. Time vs Activity



我們針對系統搬運資料的 dataflow 利用 sypglass 進行 power 的分析，可以在 spyglass 產出的 report 中看到，當系統在存取資料時，power 的 activity 會比較高，而進行其他操作時，power activity 則會比較低，我們可以分析系統在哪裡有比較高的功率消耗，進而對該部分進行優化，以減少 power consumption。

四.Share your thoughts

1. 胡家豪

   經過了本次專題我學習到了很多，主要我是負責軟體的部分，包括模型搭建與量化。除了對量化的 flaw 更加認識以外，我也理解到其實在做量化有一部份就是在設計 model。必須不斷的與硬體端溝通到底要做那些運算，還必須考慮到位寬到底夠不夠，會不會溢位等。雖然這次專題很累，但是收獲也很大！

2. 蔡明翰

   這次的 project 我負責的是 DLA controller 和 PE array 的部分，在刻硬體的過程中，會發現 paper 跟實際在實作會有很大的出入，並且可能會因為誤解 paper 的意思，而不斷的進行修正。所以一開始在制定硬體規格時，可能就需要想好整個 data flow 如何運作，甚至可能需要透過軟體先進行模擬。另外就是軟硬體溝通的問題，在這次的 project 中，我們遇到很多次軟體的結果迥異於硬體，原本以為是量化或其他原因的損失，後來在用 testbench 確認硬體功能應該是正常的後，才發現是軟體和硬體的資料排序不同。另外就是量化的問題，原本以為量化僅僅只是將數字縮放 range 而已，但其實不然，量化需要可慮到硬體可以表示的範圍，以及硬體運算後的結果，其實是非常複雜的。經過這次 project，算是跑過個簡單的 AI 加速器的 flow，要 programable，甚至需要 compiler 的輔助，以避免都人工計算，也深刻體會到，AI 加速器最難的其實是整個 data flow 並如何想出好的控制訊號，收益良多。

3. 王文楷

   在這次的 final project 中，我們組解決了需多困難，也收穫的非常多。首先在技術層面，我們克服了許多難題，包括硬體架構設計、以及優化我們的 dataflow。每一項克服的困難都表示我們的硬體設計的實力又邁進了一步，此外，這次 final project 需要互相溝通才能有效的解決遇到的難題，這提升了我與他人共同合作的能力。

   設計這次 final project 硬體的過程也讓我們體會到團隊合作的重要性，只有在組員的共同努力下，我們才能克服遇到的困難，這次經歷讓我們學到了

以後在面對新的挑戰時，只要保持這種合作精神和不懈追求的態度，一定能解決問題。

4. 陳冠穎

   這次的 final project 讓我深入了解了加速器的開發過程，從硬體架構的構想、資料運算的 dataflow 討論，到硬體設計的實現。實作過程中，軟硬體的整合以及硬體單元之間的溝通與合併，是我們投入最多時間的部分，透過大家合作，最後才能夠順利的完成這次的 project。

5. 黃雍翔

   透過這次的 final project 我學習到了許多關於硬體設計相關的知識，還有設計 AI 加速器的基本流程，也了解到在設計不同的硬體、軟體和硬體之間需要多溝通才會更有效率的解決問題，透過這次的經驗，我了解到自身還有許多的不足，需要多努力才能增進自身的實力。

五.Demo and Source code link

Demo：https://youtu.be/nx5mJhMWBNw

Source code：https://github.com/kevin199907/AOC