二、作品集 (挑選代表作品報告，其餘皆放在 GitHub)

1. 個人專題-Direct Convolution CNN accelerator

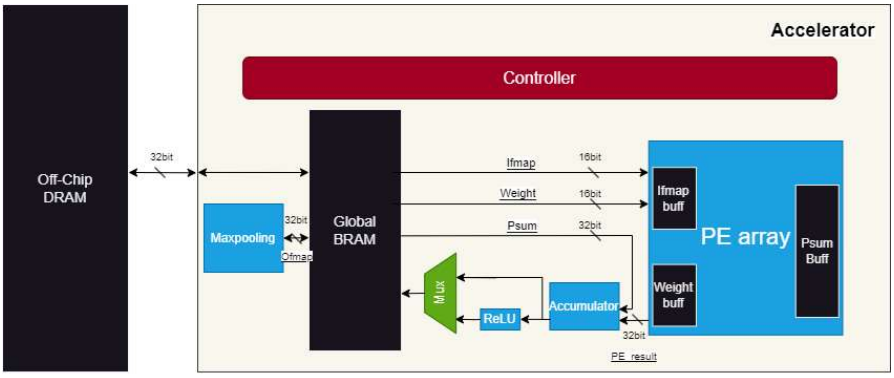
GitHub URL: https://github.com/Chia-Yu-Kuo/Graduation-Project-AI-Accelerator

甲、摘要

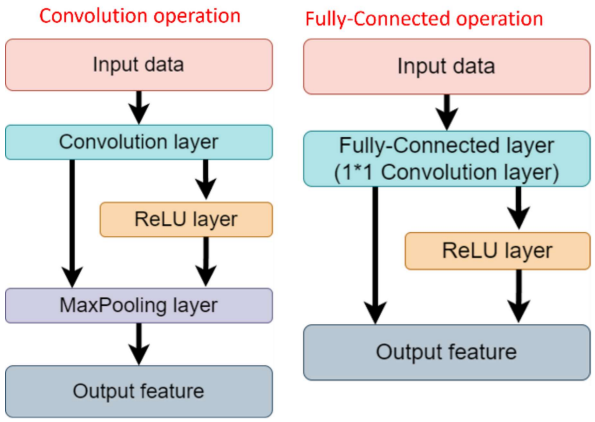
由於傳統 CPU 不擅於特定功能的運算，造成耗時及耗能等缺點。因此設計一款可於嵌入式裝置的 Convolution CNN accelerator，提高 data reuse rate 使得降低 Off-chip DRAM access，以及建立 Memory Hierarchy 的 On-chip buffers，能使加速捲積運算的同時，也可大幅降低能源上的使用。而其中選用 Direct Convolution 的型式而非 Matrix-Matrix-Multiplication 的原因是由於要 implement 的 FPGA (PYNQ-Z2) 硬體資源有限，大量 memory 空間會是一項棘手問題，但缺點是 Flexibility 和 Scalability 低。

乙、想法

整體系統因為要在 FPGA 上面運行，因此 Off-chip DRAM 是透過 PS 端傳入模擬，而 Global Memory 透過 FPGA 上的 BRAM 做模擬(頻寬因此受限，若做 ASIC design 或使用 CDMA 能提高 Bandwidth 可大幅提速)。礙於完成 CNN 加速電路就把 PYNQ-Z2 資源耗盡，因此模型後半部 MLP 也 share CNN 電路進行分類。

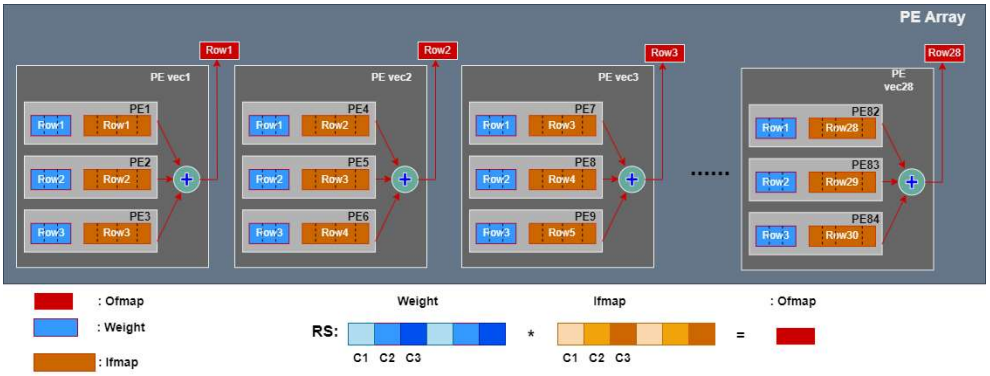


圖一、系統架構



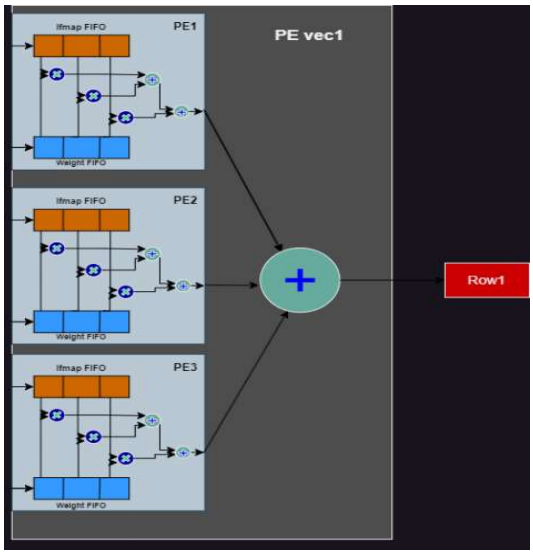
圖二、Data flow diagram

i. 採用 Row Stationary(Multiple fmap+Muliple weight+Multiple channel)來使在 register file 中 data reuse 最大化。



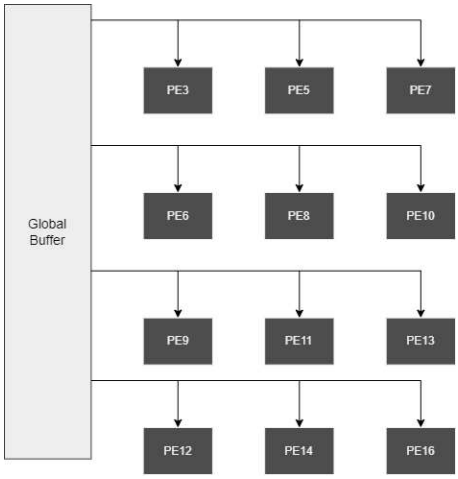
圖三、PE array 採用 RS 設計

ii. 採用 Short-lived intermediate results 技術設計 PE 及 PE_vec，來對 memory 優化。



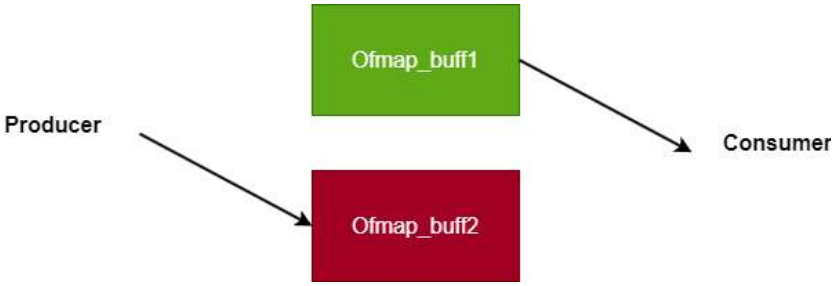
圖四、PE_vec 及 PE 架構

iii. 採用 1D Multicast NOC 來降低 FPGA BRAM 頻寬太低所造成的影響。



圖五、Network on chip 用 1D Multicast 設計

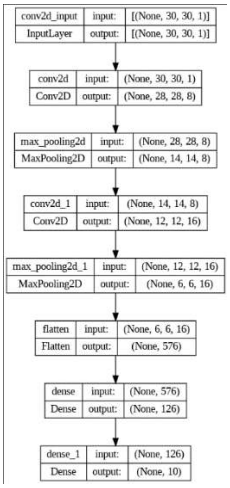
iv. 採用 Ping-Pong-buffering SRAM 設計減緩計算與傳輸重疊。



圖六、Ofmap buffer 用 Double-buffering 設計

v. 平行化使 PE utilization 在 CONV2 layer 提高，不會受到 input size 因 Downsample 縮小而 PE underutilized。

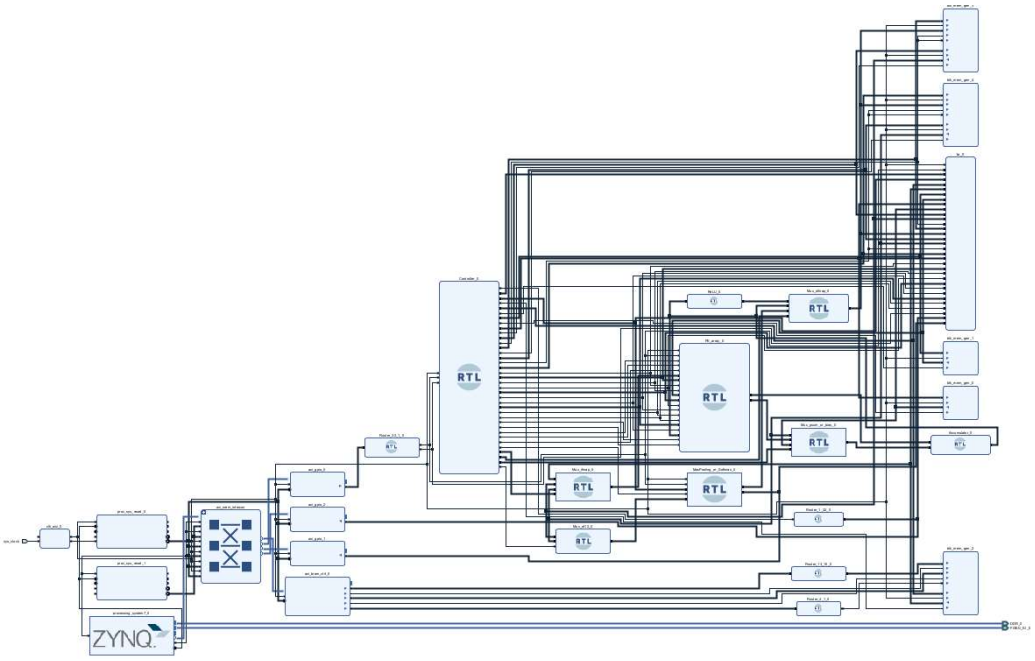
丙、實體化




```
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 28, 28, 8)         80

max_pooling2d (MaxPooling2D  (None, 14, 14, 8)         0
)

conv2d_1 (Conv2D)            (None, 12, 12, 16)        1168

max_pooling2d_1 (MaxPooling  (None, 6, 6, 16)          0
2D)

flatten (Flatten)            (None, 576)               0

dense (Dense)                (None, 126)               72702

dense_1 (Dense)              (None, 10)                1270

=================================================================
Total params: 75,220
Trainable params: 75,220
Non-trainable params: 0
_________________________________________________________________
None
```

```
for layer in model.layers:
    if 'conv' in layer.name:   # Assuming you want to extract weights from convolutional layers
        weights = layer.get_weights()
        if weights:   # Check if there are weights in this layer
            weight_matrix = weights[0]   # Get the weight matrix
            # Rearrange the weight matrix according to the specified order
            weight_matrix = np.transpose(weight_matrix, (3, 2, 0, 1))   # output_channels -> input_channels -> row -> column
            # Flatten the weight matrix for conversion
            flattened_weights = weight_matrix.flatten()
            # Convert and append the weights to the sorted list
            #sorted_weights.extend(float_to_hex(value) for value in flattened_weights)
            for value in flattened_weights:
                fixed_point_value = float_to_fixed_point(value)
                hex_value = format(fixed_point_value & 0xFFFF, '04x')   # Mask to 16 bits (2 bytes)
                coe_data.append(hex_value)
    else:
        weights = layer.get_weights()
        '''
        if(layer.name=='dense_1'):
            #print(weights[0])
            dim1 = len(weights[0])
            print(dim1)
            dim2 = len(weights[0][0])
            print(dim2)
        '''
        if weights:   # Check if there are weights in this layer
            weight_matrix = weights[0]   # Get the weight matrix
            # Rearrange the weight matrix according to the specified order
            weight_matrix = np.transpose(weight_matrix, (1, 0))   # output_channels -> input_channels -> row -> column
            # Flatten the weight matrix for conversion
            flattened_weights = weight_matrix.flatten()
            #sorted_weights.extend(float_to_hex(value) for value in flattened_weights)
            for value in flattened_weights:
                fixed_point_value = float_to_fixed_point(value)
                hex_value = format(fixed_point_value & 0xFFFF, '04x')   # Mask to 16 bits (2 bytes)
                coe_data.append(hex_value)

with open('/content/drive/My Drive/py/weights_v5.coe', 'w') as coe_file:
    coe_file.write('memory_initialization_radix=10;\n')
    coe_file.write('memory_initialization_vector=\n')
    coe_file.write(',\n'.join(map(str, coe_data)))
    coe_file.write(';')
```

圖七八九、將訓練好的模型參數萃取出來



圖十、AOC Block design

丁、結果

i. Accuracy and Time cost

軟體端測試是將 tensorflow、keras 安裝到 FPGA 的 ubuntu 中，用純 CPU 去跟電路做比較；硬體端則是透過 PYNQ-z2 的 SD card 當 OS 用 python 測試電路。

SW accuracy

```
In [19]: CPU_start = time.process_time()
         pre=model.predict(xtestpad)
         CPU_end = time.process_time()

         print("CPU use", CPU_end - CPU_start, "sec")

         CPU use 50.453142642 sec

In [20]: right = 0
         for i in range(pre.shape[0]):
             if(np.argmax(pre[i]) == ytest[i]):
                 right=right+1
         print("accuracy:", right/pre.shape[0] )

         accuracy: 0.9778
```

HW accuracy

```
In [288]: from tqdm import tqdm

In [289]: predict = []
          for i in tqdm(range(xtestpad.shape[0])):
              load_test(xtestpad[i])
              RST_START_MMIO.write(0,0x00000001)
              RST_START_MMIO.write(0,0x00000002)
              RST_START_MMIO.write(0,0x00000000)
              while(True):
                  IS_DONE = DONE_MMIO.read(0)
                  if(IS_DONE == 1):
                      break

              predict.append(INDEX_MMIO.read(0))

          100%|████████| 10000/10000 [08:08<00:00, 20.86it/s]

In [300]: pre = np.array(predict)
          right = 0
          for i in range(pre.shape[0]):
              if(pre[i] == ytest[i]):
                  right=right+1
          print("accuracy:", right/pre.shape[0] )

          accuracy: 0.9799
```

圖十一十二、可發現 SW 與 HW 測試的準確度差不多

FPGA processor speed

```
In [3]: import tensorflow as tf
        from tensorflow.keras.datasets import mnist
        import numpy as np
        import time

In [4]: model = tf.keras.models.load_model('FPGA_project.h5')

In [5]: (xtrain, ytrain), (xtest, ytest) = mnist.load_data()

In [6]: xtestpad=np.pad(xtest, pad_width=((0,0),(1,1),(1,1)), mode='constant', constant_values=0)
        xtestpad= xtestpad.reshape(-1,30,30,1)

In [14]: pix = np.array(gray_image)
         pix= pix.reshape(1,30,30,1)

In [17]: CPU_start = time.process_time()
         pre=model.predict(pix)
         CPU_end = time.process_time()

         print(np.argmax(pre))
         print("CPU use", CPU_end - CPU_start, "sec")

         0
         CPU use 0.5847233129999978 sec
```

AOC speed

```
In [52]: import time

In [53]: arr = xtest[0]

In [60]: start = time.process_time()
         load_test(arr)
         RST_START_MMIO.write(0,0x00000001) #rst
         RST_START_MMIO.write(0,0x00000002) #START
         RST_START_MMIO.write(0,0x00000000) #CLEAR
         while(True):
             IS_DONE = DONE_MMIO.read(0)
             if(IS_DONE == 1):
                 break
         print(INDEX_MMIO.read(0))
         end = time.process_time()

         print("AOC use", end - start, "sec")

         0
         AOC use 0.057337041000000255 sec
```

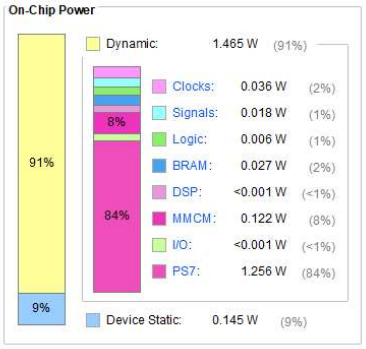
圖十三十四、可看出 AOC 比 FPGA 純 CPU 快 10 倍左右

ii. Power consumption

Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

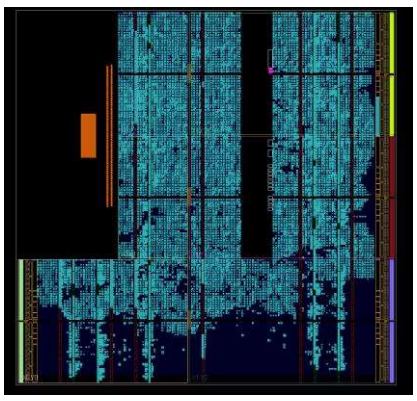
Total On-Chip Power:	**1.61 W**
Design Power Budget:	**Not Specified**
Power Budget Margin:	**N/A**
Junction Temperature:	**43.6°C**
Thermal Margin:	41.4°C (3.4 W)
Effective ϑJA:	11.5°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Medium

Launch Power Constraint Advisor to find and fix invalid switching activity

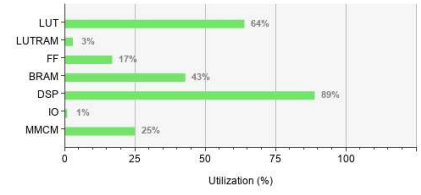


圖十五、AOC power consumption

iii. HW Cost

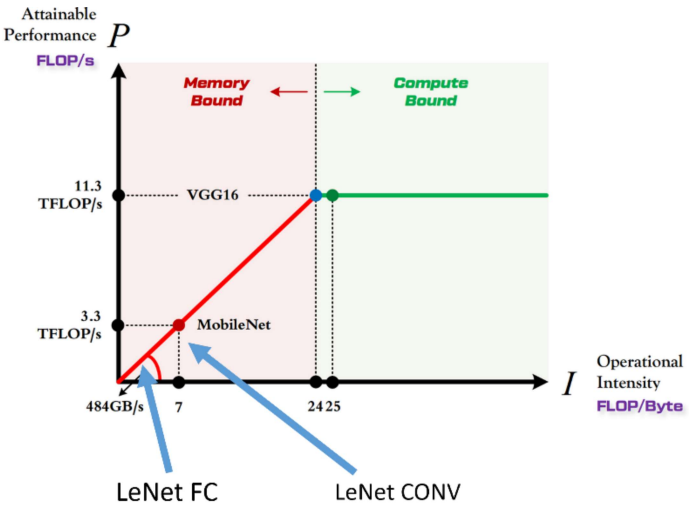


Resource	Utilization	Available	Utilization %
LUT	34073	53200	64.05
LUTRAM	608	17400	3.49
FF	17908	106400	16.83
BRAM	59.50	140	42.50
DSP	196	220	89.09
IO	1	125	0.80
MMCM	1	4	25.00



圖十六十七、AOC 在 FPGA 上硬體使用資源

iv. Roofline model



圖十八、分析設計加速器於 Roofline model 中所遇到的限制