HW#5 Report
Domain-Specific Accelerator (DSA)

邱振源, 111550100

***Abstract*— 這次的作業主要是透過 Memory Mapped I/O (MMIO) 的方式在 Aquila 實作一個 Domain-Specific Accelerator (DSA)，透過使用 Vivado 在 Aquila 添加 IP 或改寫電路的方式，縮短一個 CNN 模型在 Aquila 上的執行時間，最後加速了 20 倍。**

Keywords— RISC-V; Aquila; DSA; MMIO; floating point; CNN; Vivado ip; converlution layer

I. INTRODUCTION

在觀察 C code 產生的 objdump 檔可以發現，當 Aquila 在遇到 inner product 的時候會轉向運用 __mulsf3 和 __addsf3 這兩個 function 來運行，然而這兩個 function 的本身也需要執行許多指令，因此，這次的作業是要在 Aquila 中實作一個 Domain-Specific Accelerator (DSA)，透過 Memory Mapped I/O (MMIO) 的方式溝通 Aquila 和 DSA，來讓 Aquila 可以透過 load/store 搭配電路來快速執行一些重複性的運算。

而根據老師上課的提示，我嘗試更改程式碼進行加速的部分主要是在 fully_connected_layer.h 中的 fully_connected_layer_forward_propagation()、average_pooling_layer.h 中的 average_pooling_layer_forward_propagation()以及 convolutional_layer.h 中的 conv_3d()，後面針對加速的討論也會主要分析這三個函式。

II. CONCEPT OF DSA

A. MMIO

透過 MMIO 的方式，可以讓我從軟體透過 Aquila 和 DSA 溝通，根據 Aquila 在 soc_top.v 中 Device address decoder 的註解，可以發現 Aquila 將 0xC400_0000 到 0xC4FF_FFFF 這段 memory address 保留給 DSA 使用，也就是說，當 load/store 指令要存取這段 memory address 時，soc_top.v 會藉由定義 dsa_sel 來 select DSA device，並將要 store 的 data 傳送到 DSA，同時將 DSA 要輸出的 data，load 出來。

因此，透過 MMIO，我定義了幾個 address 來傳輸軟體與硬體之間互動的 data，在我的實作中，這些地址搭配了一個 32 bits 的 data input 或 output，data 便是傳輸的資料，而 input 或 output 取決於是做 load 或 store 的指令。

從軟體層面來看，這些地址比較像是一個 pointer，軟體可以透過修改這個地址，或讀取這個地址來傳值與讀值；而硬體層面的 DSA 則是透過這些地址來將值存入暫存器或是將暫存器的值透過 data_o 傳遞出去。透過這樣的方式便可以軟體與硬體透過 Aquila 和預先配置好的 MMIO address 進行溝通與互動。

B. Data Feeder and soc_top.v

在觀察 soc_top.v 之後可以發現 Device address decoder 會決定傳回 Aquila 的值是多少，因此我便模仿 uart 的方式定義 select 和 data 的相關訊號，並模仿 CLINT 和 uart 在 soc_top.v 中的訊號傳遞方式，在 soc_top.v 中加入 data_feeder 的 module 來實作 DSA，主要的不同是將 enable 的部分定義成看 address 是不是在 DSA device 的範圍。

而在 data feeder 的部分，當 en_i 和 we_i 都是 1 的時候，data 的值才可以根據 DSA device 的 address 存到所分配的暫存器，而在觀察 data 可以不可以傳回 Aquila 的部分則是會先看 DSA 的 en_i 和 we_i 訊號，並確保 dsa_ready，data 才會透過 Device address decoder 中的 dev_dout 傳回 Aquila。

C. Floating Point IP

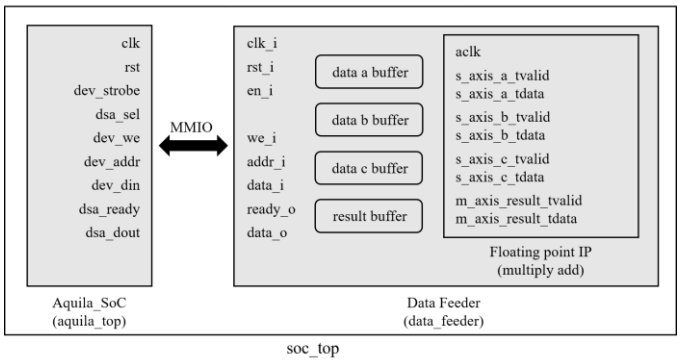

Fig. 1. Architecture of DSA

接著我便根據作業的說明在 data feeder 添加 floating point 的 IP，跟據作業的說明，我先嘗試在做 dot product 的時候使用 IP，所以在 IP 的設定是選 Fused-Multiply-Add，另外我也在 blocking 的模式中選擇 non-blocking 的方式，因為使用 non-blocking 就可以不用處理比較複雜的控制訊號，而在 latency 設定的部分，我一開始嘗試將他設成 0，但發現他會做不完運算，最後將 latency 的部分設定成 2。

因為設定成 non-blocking 的方式，所以為了避免混亂，我將這個 IP 的運作方式設定成當三個資料都讀完之後才會有一個 data_valid 的訊號將三個資料送進去 IP，而當 result valid 之後，我會先用一個暫存器先保留運算的結果，完成之後才會接著讀新的三個資料，來讓 IP 的輸入輸出保持以同一組為一個單位。

設定完 floating point 的 IP 之後，一個基本的 DSA 便完成了，Fig. 1 展示了一個基本的 DSA 的架構，aquila top 會將 device 的相關訊號和 clk 傳入 data feeder，data feeder 會

先將資料暫存在被分配到的 buffer，當所有資料完成之後 floating point ip 會開始運算，運算完的結果會先存在另一個暫存器，等 aquila 要 data 的時候再透過 data_o 傳出去。

III. Basic Implementation

這部分主要是分析 CNN 模型中會大量執行 floating point 運算的地方，在這些需要大量運算的地方使用 MMIO，改成在 DSA 的地方用 IP 運算，進而減少模型在 Aquila 上的運行時間。接下來的討論也會著重在我在三個不同 function 使用 DSA 運算之後的結果。

此外，我先測試未經修改的軟體以及 Aquila，得到的結果是 21354 msec，同時我也記錄了三個 function 的 cycle 數，並將它們設為 baseline 來討論之後加速的倍率。

A. *fully_connected_layer_forward_propagation()*

```
for (uint64_t c = 0; c < entry->base.in_size_; c++){
    *((float volatile *)0xC4000000) = W[i*entry->base.in_size_ + c];
    *((float volatile *)0xC4000004) = in[c];
    *((float volatile *)0xC4000008) = a[i];
    a[i] = *((float volatile *)0xC4000010);
}
```

Fig. 2. Software Structure of Original Approach

這個 function 是在 fully_connected_layer.h 中的函式，老師上課的時候說可以先嘗試從這個函式下手，會比較好理解後面的使用，因此最一開始，我沒有特別改變 software 的邏輯，用 Fig. 2 的方式直接將 inner product 的部分交給 DSA 做，每做完一次 multiply-add 之後就將值讀出來，下一次運算的時候再傳進去，這樣的做法讓 fully_connected_layer_forward_propagation()的 cycle 數從 23860231 下降到 4349448，下降了大約 80%的運算時間換算成秒數的話就接近下降了 390 msec。

在觀察 ILA 的結果之後可以發現，原本的方法的 bottleneck 在 data feeding 的時間，calculating 所花的兩個 cycle 的 latency 幾乎可以不計，也因此我就沒有特別計算 calculation time，同時我也在思考有沒有辦法可以減少 data feeding 的時間。

```
for (uint64_t c = 0; c < entry->base.in_size_; c++){
    *((float volatile *)0xC4000000) = W[i*entry->base.in_size_ + c];
    *((float volatile *)0xC4000004) = in[c];
}
a[i] = *((float volatile *)0xC4000008);
```

Fig. 3. Software Structure of New Approach

在觀察這段程式碼之後可以發現，a[i]需要不斷的做 inner product，且 a[i]的初始值是 0，因此，我將這部分的做法修改成如 Fig. 3 所示，將 IP 中的 data C 變為原先 IP 算出的結果，如此便只需要讀入 C = C + (A * B)中 A 和 B 的部分，修改完成之後，fully_connected_layer_forward_propagation()的 cycle 數近一步下降到 3331037，比原先下降了 86%，大約下降了 411 msec。

B. *conv_3d()*

在修改完 fully_connected_layer_forward_propagation()之後，我開始修改 conv_3d()，同樣將最內層 loop 中有關 floating point inner product 運算的部分交給 DSA 執行，同樣用將 inner product 的結果當作 data C 的方式累加 sum。在做完這部分之後，conv_3d()的 cycle 數也從 976625323 下降到 111552686，約下降 89%，減少了 17301 msec。

C. *average_pooling_layer_forward_propagation()*

老師上課提到，pooling layer 也占了整個運算一大部分的時間，在透過 ILA 計算之後可以發現 average_pooling_layer_forward_propagation()在未修改之前約佔整體的 27%，不同的於前面兩個 function，這個 function 中 floating point 的運算主要是單純的 add 和 multiply，因此，我也在 DSA 中添加 floating point add 與 multiply 的 IP，且 add 的 latency 為 2，multiply 的 lantency 為 1 (確保 WNS > 0)，有關 IP latency 與加速的關係也會在後面討論。

在這邊比較需要注意的是，一開始的時候我一直得到錯誤的結果，在觀察波形圖之後發現前幾次的計算中 add IP 的結果無法順利的更新，因此在 output 的部分，我用了 data_o <= add_result_valid ? add_result_data : add_result_reg; 來處理這樣的狀況，如此便能得到正確的結果。最後，這個 function 的 cycle 數也從 297012178 下降至 71106769，約下降了 76%，減少了 4518 msec。

D. *Result And Analysis*

TABLE I. Result Of Adding IP in DSA (msec)

Approach	Original	Add IP
Total Calculate Time	21354	3326 (15.58%)
fully_conn() Time	477	67 (14.05%)
conv_3d() Time	19533	2225 (11.39%)
avg_pool() Time	5940	1442 (24.28%)

TABLE I 記錄了在添加 IP 之後，各個 function 的執行時間變化，因為現在的方法都只是單純透過 IP 計算，所以我就沒有將 data feeding 的時間和 calculation 的時間分開，但還是可以發現，只是單純將三個 function 中的 floating point 運算改由 DSA 的 IP 執行，並透過累加的方式減少 I/O，便可以提升 6.42 倍的運算速度，由此可見原本 floating point 運算耗時之多。

IV. Advance Implementation

由於一直找需要換成 IP 的地方沒辦法再有效的對這個 CNN 模型加速很多，所以我嘗試將一些 C code 的地方轉成電路來完成，先透過 MMIO 讀入 DSA 計算時需要的參數，再設置 busy waiting 的 trigger，等 DSA 計算完便會將 trigger 設為 0，進而跳出 while 迴圈，進而完成加速。

A. *average_pooling_layer_forward_propagation()*

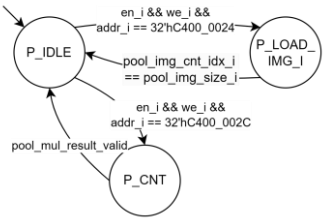

Fig. 4. FSM of Circuit in Pooling Layer

因為 conv_3d()的 index 計算有點複雜，同時 average-_pooling_layer_forward_propagation()也占運算中很大的一部份，所以我決定先從這個 function 下手。在觀察 software 的部分之後，我發現在最外層 loop 的裡面可以簡單看成一個初始值為 0 的變數(a[o])，加上很多透過 dx 和 dy 算出 index 之後的 in[]裡面的值，最後再乘上 entry->scale_factor_。

因此我建立了一個如 Fig. 4 的 FSM，透過 0xC400_0024 這個位置讀入 dxmax*dymax，作為 pool_img_size_i，接著透過一個 counter 去計算要讀入的 in[]資料，in 的 index 會透過 software 計算，讀完之後會回到 P_IDLE，接著，將 0xC400_002C 設為 calculating trigger，被 trigger 開始之後，DSA 會透過 floating point add IP 累加所以有存入的 in[] 資料，加完之後再透過 multiply IP 乘以 scale_factor_，不過因為再 average pooling 的部分，scale_factor_可以直接設成 0.25，因此我直接將其設為 0x3e800000 也就是 0.25 的福點數表示法，當算完之後會再將 trigger 設為 0 進而跳出 while 的 busy waiting 狀態。最後再將 a[o]設為 0xC4000030 的值便完成一次的運算。

B. conv_3d()

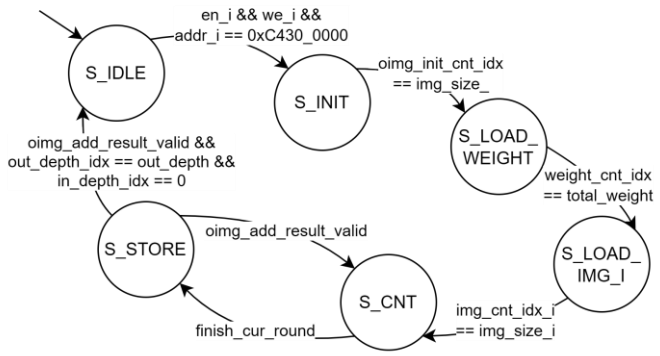

Fig. 5. FSM of Circuit in conv_3d()

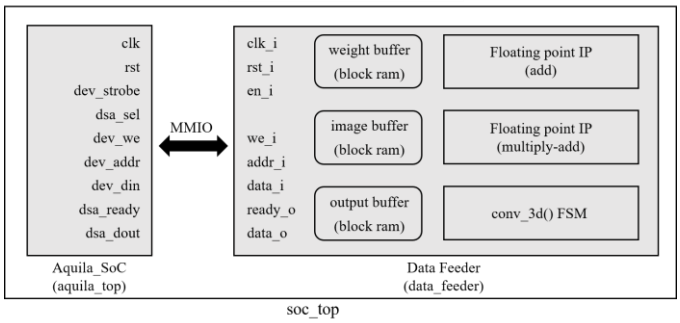

Fig. 6. Block Fiagram of Circuit in conv_3d()

做完將部分的 pooling layer 轉成電路之後，我便嘗試將 conv_3d() 這個 function 也轉成電路，在分析 software 的寫法之後，可以發現我們可以先將兩個主要的 data: input image 和 weight 的部分先 preload 進電路，然而我發現，如果直接將這些資料存進 hardware 中，如果沒有妥善規劃，Vivado 可能會選擇用 LUT 的方式合成，而老師在作業的說明中提到可以用 BRAM 來存這些資料。因此在使用 register 儲存後，我簡化 I/O 的部分，並將 ram style 設為"block"，進而確保儲存這些資料的 register 會用 block ram 的方式合成。

接著，如同 pooling layer，我也用一個 FSM (Fig. 5) 來控制整個 conv_3d()的電路，先讀入需要的參數之後，會在 0xC430000 這個地址開始初始化 output 的暫存器，初始化完成之後，會接續進到 load weight 及 load image data 的 state 讀入相關的資料，接著軟體便會進到 busy waiting 的狀態直到 state 重新回到 S_IDLE，在這之間，由於 weight 的 width 是 5，所以我將做 25 次 inner product 作為一個 round，每做完一個 round 就會到 S_STORE 儲存資料到 output 的暫存器，直到所有 inner product 都做完之後，S 才會回到 S_IDLE 進而解除 busy waiting 的狀態。整個過程中的互動與元件大致如 Fig. 6 所示，其中在硬體實作的部分簡化了一些不必要的計算，如 padding 為 0，stride 是 1，便直接計算完成的結果進而減少計算量。

C. Result And Analysis

TABLE II. RESULT OF ADVANCE IMPLEMENTATION (MSEC)

Counting Area	Baseline	Total Time	Pooling Part	Conv Part
Total Time	21354	1067 (20.01x)	815 (76%)	235 (22%)
Calculating Time	N/A	N/A	204	214
Data Feeding Time	N/A	N/A	311	21

TABLE II 記錄了在將 conv_3d()與部分的 pooling layer 轉成電路之後執行各個部分所需執行的時間，可以發現當將整個 function 轉為電路之後，conv_3d()的時間大幅降低，除了計算的時間 data feeding 的時間也占據了整體的 9%，也是一個 overhead。

相較於 conv_3d()，pooling part 的 caculation time 比 data feeding time 還短，推測是因為我目前只將部分的計算改成電路，計算的本身其實還不是很龐大，因此相較於計算，我目前的 DSA 對於 pooling layer 的加速還是以 data feeding 為 bottle neck。

最後，做完 advance implementation 將一些 software 的地方直接轉成電路執行之後，我的整體的加速為 20.01 倍，但我發現現在的 DSA 使用了大量的資源，因此我嘗試優化現在的 DSA，並在下一部份討論。

V. OTHER DISCUSSION

A. IP Latency amd WNS

TABLE III. RESULT OF DIFFERENT LATENCY

Version	Baseline	Normal	Chang Latency
Spend Time (ms)	21354	1067	990
Speedup Rate	1	20.01	21.60
Clk Rate	50 MHz	50 MHz	50 MHz
IP Latency	2	2	1
WNS	> 0	> 0	-12.59

在決定 IP 的 latency 的時候，我發現調低 latency 可以減少 CNN 模型在當前的 Aquila 上的運行時間，同時，inner product 的計算時間，在初始的 CNN 模型與 DSA 版本中，是決定加速更過的關鍵。因此，在將部分 software 轉換成

電路之後，我便試著降低 multiply-add IP 的 latency，並記錄他造成的影響。

TABLE III 紀錄了在降低 multiply-add IP 的 latency 之後的結果，可以發現，降低 latency 可以在正確率仍是 95%的情況下，成功加速 DSA 的運作，然而，在將 IP 的 latency 設為 1 之後，Setup time 的 WNS (Worse Negative Slack)變為-12.59，這意味著當前的運作沒辦法在該 cycle 完成，而這樣可能使雖然正確率不變，但其實有一些 operation failed。

B. Resource Utilzation

TABLE IV. RESOURCE USAGE OF DATA FEEDER IN DIFFERENT VERSION

Version	Baseline	Normal	Optimize
Spend Time (ms)	21354	1067	1183
Speedup Rate	1	20.01	18.05
Slice LUTs	N/A	8568	6837
Slice	N/A	2621	2064
LUT as Logic	N/A	3660	3337
LUT as Memory	N/A	4908	3500
Block RAM	N/A	5	5

在做完將部分 pooling 跟 conv_3d()轉成電路之後，我發現現在的做法會消耗很多的資源，所以我便嘗試縮減一些原本設計的時候使用的暫存器與 state 和 flag。

TABLE IV 紀錄了不同版本之間硬體資源的使用狀況，可以發現其 Block RAM 為 5，驗證了上面敘述將一些 weight 和 image data 轉到 block ram 的做法。此外，可以發現優化完的版本中，LUT 的使用大幅下降，無論是 Slice 或是 LUT as Memory 都下降到 7 至 8 成。然而，優化完的版本的加速狀況變少，推測可能是因為在刪除一些多餘的 state 中，刪掉了原本 trigger 一些計算的 flag，這種特殊判斷的 flag 可能可以讓運算提早進行但沒有其他作用，且不影響原本的計算邏輯，因而被刪除卻增加了一點運算時間，而我最後交上 e3 的程式碼也是 optimize 之後的版本。

VI. FUTURE WORK

A. Integration

目前的做法中，我只有將 conv_3d()和部分的 pooling layer 轉成在 DSA 中用電路的方式進行，之後也可以將更多 function，如：完整的 pooling、relu、fully connected layer……轉在 DSA 執行，變能更有效的提升執行速度。

此外，當前的做法更像是抓取某一塊軟體的地方換在 DSA 執行，若有機會可以試著將整個 CNN 模型轉成電路，變可以不需藉由太多的軟體輔助而直接在電路完成。

B. Quantization

老師上的的時候有提到，在這種 CNN 的運算，可能可以不用到 floating point 那麼精確的運算，用整數 integer 就可能很足夠了。

因此就可以將 image 和 weight 的資料轉成用 8 bits 的 integer 值，因此，只要確保正確率依舊，就可以將 4 個 8 bits 的 integer 資料轉成一個 32 bits 的資料。這樣可以大幅減少 I/O 的傳遞次數，同時，DSA 也可以用整數運算的方式對資料進行運算，而不需使用大量的 floating point IP，DSA 的設計上也會更加簡單，因此，混和精度可能是一個可以嘗試的方向。

C. Parallel Operations

在現在的做法中，雖然我會在執行運算之前就先將所有參數和 image 以及 weight 的資料讀完，但我現在在執行運算的時候還是會將資料一筆一筆的讀入同一個 IP，之後可以嘗試將 index 一次算好之後，開多個 IP 並將這些資料同時塞進去 IP 中，變可以在一個 clock cyle 執行更多的運算，進而提升速度。

VII. CONCLUTION

這次的作業中，從加入 IP 到最後改寫部分的電路，發現了很多在寫軟體的時候不會特別注意，但在硬體層面上可能會大幅度影響效能的地方，像是 IP latency 的設定，或是自己撰寫電路的 resource 使用狀況，改變 latency 雖然可以在保有正確率的情況下提升效能，但可能會使 WNS 變成負值，而 resource 的使用也同樣十分重要，雖然我最後交上的檔案縮減了很多 LUT 的部分，但我覺得應該可以再將更多資源轉成 BRAM 的形式，也是之後可以嘗試的方向。

最後，在這次的功課中我成功將 CNN 模型的運作透過 DSA 加速 20 倍，從軟體一步一步改，加入 FSM 的控制，到最後為了可以改成電路去更了解 CNN 的運算，我覺得在做這個功課的時候蠻有成就感的，也謝謝老師給我們這樣子的軟體與硬體環境。