高階合成技術於應用加速 Final Project Discussion

109061806 羅允辰 110061555 李冠霈 110061575 王睿瑄

I. Introduction

矩陣乘法是非常重要的運算，特別是現在許多重要的人工智慧、圖相關的應用都會使用到大量的矩陣運算。但是矩陣運算的運算量很大，並且這個狀況在矩陣大小增加時會更加嚴峻，因此許多研究探討如何設計出有效率的矩陣加速器。

現有的矩陣乘法加速器無法應對以下三個挑戰：

1. 現在有許許多多低位寬格式，需要特別的電路設計才能支援，彈性的拓展到各種格式對電路系統是非常重要的。
2. 大部分加速器都採用 HDL 來開發，使得他們很難被更改、拓展，因此使用 HLS 來開發加速器是重要的趨勢。
3. 沒有加速器願意公開設計，因此讓社群無法因此受惠。

根據我們的研究，目前只有 ETHZ 的學者公開他們的加速器，我們的 final project 選定期作為我們的基礎。

具體來說，我們想做的事情是：

1. 完整研讀他們的論文 (此文件)
2. 研究他怎麼使用 HLS 架構一個可以輕鬆拓展的加速器
3. 實際將它部署到 U50/PYNQ-Z2 板子上
4. 調整各種參數來詳盡的研究他

5. (有時間的話)提出點子來讓他更好

II. Optimization Goals

一個最佳的矩陣乘法加速器有以下特點：

1. 擁有最高的效能 (運算)
2. 能最少的存取晶片外的資料 (IO)
3. 使用到所有的 FPGA 資源 (資源)

此外，分別在運算、IO、資源上有許多設計時的眉角，以下分別分析他們：

1. 運算：在 FPGA 上，相較於 GPU 等系統，要最佳化舉證運算其實已經是不容易的工作，這篇提出要把問題 decompose 成可以良好反應結果的 module。
2. IO：運算的位置(mapping)和時機點(scheduling)都是重要的，前者會影響 PE 間的資料傳輸、後者則會影響和 memory 的 traffic
3. Resource：在 FPGA 上，很不同的點是需要考量 routing resource 和 chiplet 的特性，前者需注意不要常用 1-to-N 的 broadcast topology 和 N-to-1 的 reduction topology。另外很重要的是盡量不要讓 PE 太大，導致同個電路落在不同的 chiplet 上。

III. Optimization Models

作者主要從三個面向討論優化的方法。

1. Computation Model

   為了優化運算上的 performance，最直接的想法就是減少運算的時間。因此可以表示成下式：

$$minimize\ T\ = \frac{mult - add\ ops}{f \cdot N_c} = \frac{F}{f \cdot N_c} = \frac{mnk}{f \cdot N_p \cdot x_c y_c}$$

   其中，PE 的 bit-width 稱為w_p，因此 PE 內的x_c及y_c須滿足$x_c w_c \le w_{p,max}$，$y_c w_c \le w_{p,max}$。在一個 PE 內，所含的 vector 數量有$r_p + r_c \cdot x_c \cdot y_c$，而 FPGA 最多的 resource 為$r_{max}$，compute units 及 PE 所消耗的資源不應超過可用的總 resource，故須滿足$N_p\left(r_{i,p} + r_{i,c} \cdot x_c \cdot y_c\right) \le r_{i,max}$。

2. I/O model

   作者在 I/O optimal schedule 的宗旨就是 maximize computational intensity，也就是在一個 I/O operation 內，達到最高的運算量，換言之，即增加 data reuse 的次數。

   因此，為了能夠提高 data reuse 的次數，作者先將演算法轉成 computation directed acyclic graph (CDAG)，如 Figure 1(a)所示。並且從 CDAG 可以發現，A 跟 B 矩陣各自帶來$m \times k$跟$k \times n$的 input vertices，並產生$m$nk個 partial sum vertices，而最後一個 k 的$m \times$ n個 partial sum，即為 C 矩陣的 output vertices。

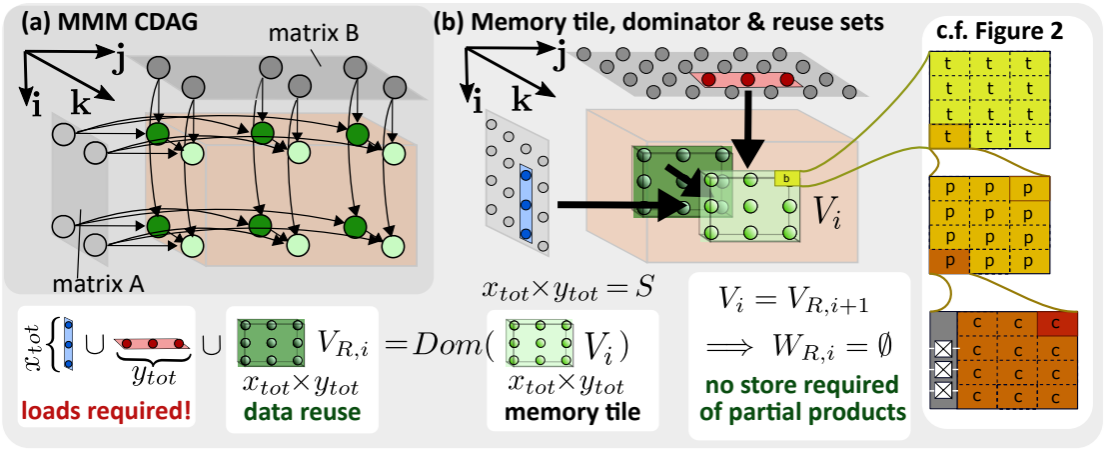

Figure 1: (a) MMM CDAG, and (b) subcomputation Vi

完成 MMM 最理想的狀況，就是每一個 input 都只讀取一次，並且中間的 partial sum 可以保留在 on-chip memory 而不需要寫回 off-chip memory 再讀。因此，作者將 CDAG 切成 h 份，並將每一次 I/O operation 可以進行的運算量作為優化目標寫成下式：

$$maximize \ \frac{|V_i|}{|Dom(V_i)| - |V_{R,i}| + |W_{B,i}|}$$

$V_i, i = 1 \ldots h$	CDAG 圖被分成 h 份的 vertices
$\|V_i\|$	在此份V_i有多少 value 被 update
$\|Dom(V_i)\|$	此V_i的 input 數量
$\|V_{R,i}\|$	在 input 中有多少數字在 on-chip memory 裡面，也就是 data reuse 的數量
$\|W_{B,i}\|$	要存回去 off-chip memory 的數量

也就是說，$|Dom(V_i)| - |V_{R,i}| + |W_{B,i}|$代表的是 I/O 的數量，$|V_i|$則代表運算量。因此，我們的目標是要最大化 on-chip memory 的 utilization，以增加

$|V_{R,i}|$ reuse 的數量。

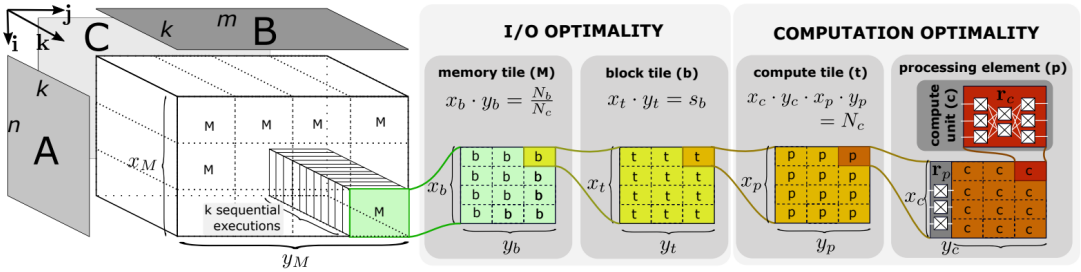

Figure 2: Decomposition of MMM achieving both performance and I/O optimality in terms of hardware resources.

Figure 2: Decomposition of MMM achieving both performance and I/O optimality in terms of hardware resources.

假設 FPGA 有N_b on-chip memory block，每一個 block 有s_b個 word，由此可知 FPGA 一次可以儲存的容量即為$N_b \cdot s_b$。同時，每個 memory block 一次讀寫的 port bandwidth 為w_b bits。

每一個V_i會看作一個 memory tile，按照 Fig 2 的切法，設$x_{tot} = x_c \cdot x_p \cdot x_t \cdot x_b$，$y_{tot} = y_c \cdot y_p \cdot y_t \cdot y_b$，$|V_i| = x_{tot} y_{tot}$。一次會從 A 矩陣 load x_{tot}個 element，從 B 矩陣 load y_{tot}個 element，並作 outer product。

此時我們原本上面的優化式就可以換成：

$$maximize \frac{x_{tot} y_{tot}}{x_{tot} + y_{tot}}, x_{tot} + y_{tot} \leq N_b \cdot s_b, x_{tot} y_{tot} \leq N_b \cdot s_b$$

就可以發現，若$x_{tot} = y_{tot} = \sqrt{N_b \cdot s_b}$，就可以將所有 partial sum $|V_i|$儲存在 on-chip memory，而不需要寫回。

3. Resource Model

考慮 I/O model 的做法，作者整理每個層級所需的 resource 數量。

hardware	resource
Compute unit (c)	r_c
PE (p)	$r_p + x_c y_c r_c$

Compute tile (t)	$x_p y_p (r_p + x_c y_c r_c)$
Block tile (b)	$x_t y_t x_p y_p (r_p + x_c y_c r_c) = s_b x_p y_p (r_p + x_c y_c r_c)$
Memory tile (M)	$x_b y_b s_b x_p y_p (r_p + x_c y_c r_c) = \left\lfloor \frac{N_b}{N_{b,min}} \right\rfloor s_b x_p y_p (r_p + x_c y_c r_c)$

其中$N_{b,min}$是考慮到每個 memory tile 一次讀寫只能 access $w_b$bits，為了使每個 compute unit 能夠在每個 cycle 讀取和寫入一個 C 矩陣的 element，至少需要$N_{b,min} = x_p y_p \left\lceil \frac{w_c x_c y_c}{w_b} \right\rceil$個 memory block 平行運算所有 compute unit。$N_{b,max}$是 FPGA 提供的 memory block tile，如果$N_c > 0.5 N_{b,max}$的話，代表 FPGA 提供的 memory block tile 有一部分一定會空下來，使用的N_b比例就會等於 $\frac{N_c}{N_{b,max}}$；相對的，如果$N_{b,max}$是$N_{b,min}$的倍數，代表我們可以很好的使用到每個 memory block tile，在提高 on-chip memory utilization 的同時，也能提升我們的 performance。

Algorithm 如下，一個 cycle 會執行一個 compute tile，PE 及 compute unit 的 for loop 使用 HLS UNROLL 優化，其他 for loop 使用 HLS PIPELINE 優化，且 II=1。

```
1 // Memory tiles m
2 for (i_m = 1; i_m ≤ n; i_m = i_m + x_tot)
3   for (j_m = 1; j_m ≤ m; j_m = j_m + y_tot)
4     for (k = 1; k ≤ k; k = k + 1) // Full dimension k
5 // [Sequential] Block tiles b in memory tile
6       for (i_b = i_m; i_b ≤ i_m + x_tot; i_m = i_m + x_t x_c x_p)
7         for (j_b = j_m; j_b ≤ j_m + y_tot; j_m = j_m + y_t y_p y_c)
8 // [Sequential] Compute tiles t in block tile
9           for (i_t = i_b; i_t ≤ i_b + x_t x_p x_c; i_b = i_b + x_c x_p)
10            for (j_t = y_b; j_t ≤ j_b + y_t y_p y_c; j_t = j_t + y_c y_p)
11 // [Parallel] Processing elements p in compute tile
12              forall (i_p = i_t; i_p ≤ i_t + x_p x_c; i_t = i_t + x_c)
13                forall (j_p = j_t; j_p ≤ j_t + y_p y_c; j_p = j_p + y_c)
14 // [Parallel] Compute units c in processing element
15                  forall (i_c = i_p; i_c ≤ i_p + x_c; i_c = i_c + 1)
16                    forall (j_c = j_p; j_c ≤ j_p + y_c; j_c = j_c+1)
17                      C(i_c, j_c) = C(i_c, j_c) + A(i_c, k) · B(k, j_c)
```

IV. Hardware Architecture

那在 Hardware mapping 的部份，Figure 2 的 compute tile 的部份會全部同時運算，也就是一次會執行$x_p \cdot y_p \cdot x_c \cdot y_c$個乘法與加法，在此比較了兩種不同的 PE 排列方式，來進行以 outer product 方式運算的矩陣乘法:

1. 2D grid PE array:

這邊以 Matrix A 為 4×4，Matrix B 為 4×2 為例，Matrix A 和 B 會以藍色、黃色、、綠色、橘色的順序送進 Feed A 和 Feed B 中，而$A_0 \sim A_{\frac{c_n}{h_n}-1}$會讀取 A column 中的不同 element，並將讀出的數值往右邊的 PE 傳遞；$B_0 \sim B_{\frac{c_n}{h_n}-1}$則會讀取 B row 中的不同 element，並將讀出的數值往下方的 PE 傳遞，每一次送進 PE 的 A、B 相乘的結果會和之前的計算結果累加，等全部計算完後，最終的結果

會依序傳回C_0~$C_{\frac{c_n}{h_n}-1}$，再傳回 Off-chip memory 中。最終每個 PE 會完成$\frac{x_{tot}\cdot y_{tot}}{N_p}$次的運算。

由 figure 3 可以觀察到以 2D gird 的架構進行乘法運算，總共會需要$3\cdot x_p\cdot y_p$個 inter-module connections，也就是 inter-module connections 的數量會正比於 PE 的數量，且在讀取 Matrix A 和 Matrix B 的部份，inter-module connections 的數量會正比於 PE array 的周長， 因此使用越多的 PE 運算，資料傳遞的複雜度越高。

$$A=\begin{bmatrix} A_{00} & A_{01} & A_{02} & A_{03} \\ A_{10} & A_{11} & A_{12} & A_{13} \\ A_{20} & A_{21} & A_{22} & A_{23} \\ A_{30} & A_{31} & A_{32} & A_{33} \end{bmatrix}, \quad B=\begin{bmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \\ B_{20} & B_{21} \\ B_{30} & B_{31} \end{bmatrix}$$

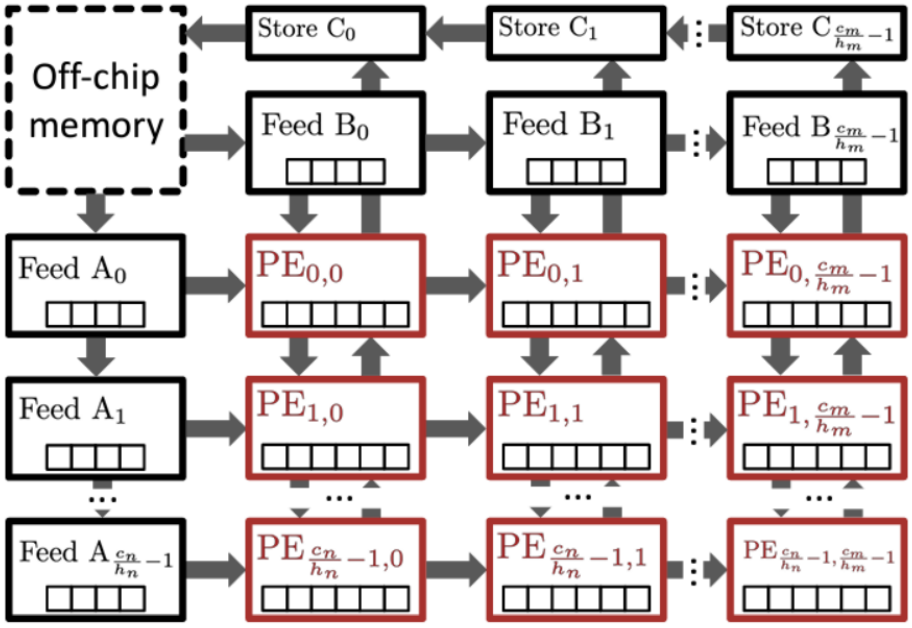

Figure 3: Compute arranged in a 2D grid

2. 1D PE array:

為了可以有效率地把運算分配給不同的晶片或是 FPGA 處理，因此提出了將 Figure 3 的 2D grid 轉換為 1D 的形式， 會將y_p和x_c設定為 1，拉成 1 維的 PE

形式，也就是x_p就等於 PE 的個數，並以y_c來控制 CU 數量。從 OFF-chip memory 會一次讀出整個 Matrix A 和 Matrix B，且讀出的 Matrix A 會儲存在 PE 中的 register 中，傳遞給不同 PE，且是以 double buffer 的形式設計，當在計算 outer product 時，會先將下一組 column 存進 PE 中，因此總共需要$3 \cdot N_p$個 register 儲存 matrix A 的數值。而 Matrix B 會儲存在 Feed B 中，以 stream 的形式傳遞，每個 tile 計算出的結果都會被儲存在 PE 中，也就是每個 PE 中會儲存$\frac{x_{tot} \cdot y_{tot}}{N_p}$個 output C 的數值。

由 figure 4，則可以觀察到改為 1D 架構後，雖然在進行乘法運算，同樣會需要$3 \cdot N_p$個 inter-module connections，但在讀取 Matrix A 和 Matrix B 和傳送 output C 的部份，inter-module connections 的數量會保持為 3，不會受到 PE 數量的影響，因此在小晶片有限的 width 下，可以更方便傳遞數值。

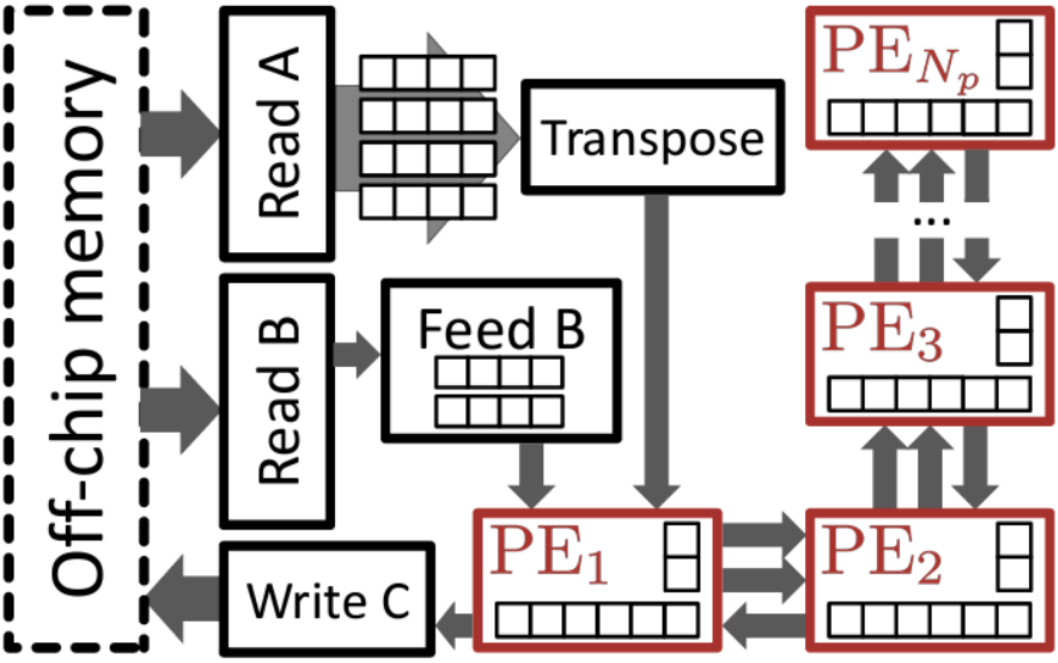

Figure 4: Module layout of final kernel architecture

在進行 outer product 的運算時，由於 Matrix A 必須由 column 的方向讀取，但在 DDR 是以 row-major 的方式儲存，因此會造成讀取緩慢的狀況，因此加入 Transpose block，可以將讀出的 Matrix A 放進與 column 個數相同的多個 FIFO 中，就可以以 Transpose 的形式輸出 Matrix A。

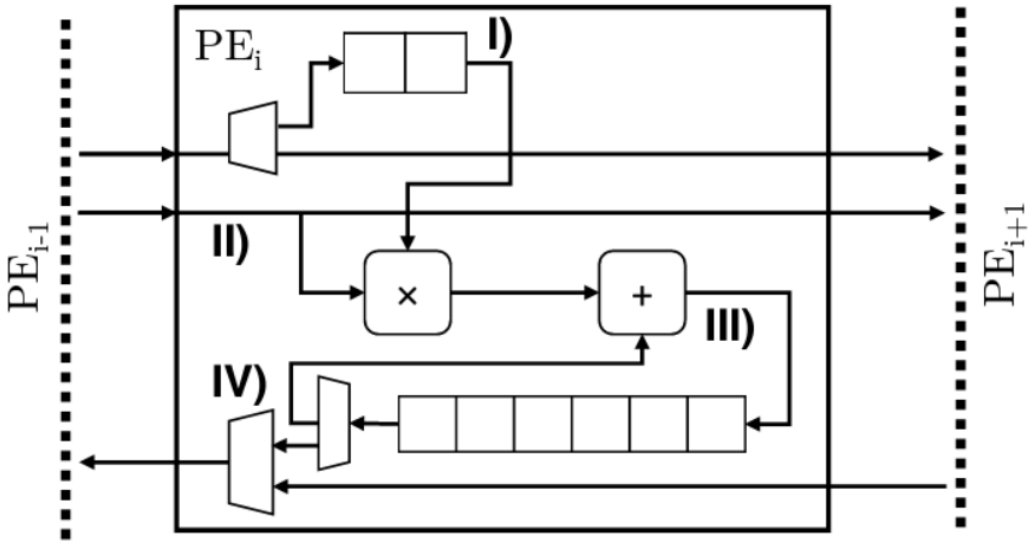

Figure 5: Architecture of a single PE

References:

1. https://github.com/spcl/gemm_hls
2. Flexible Communication Avoiding Matrix Multiplication on FPGA with High-Level Synthesis