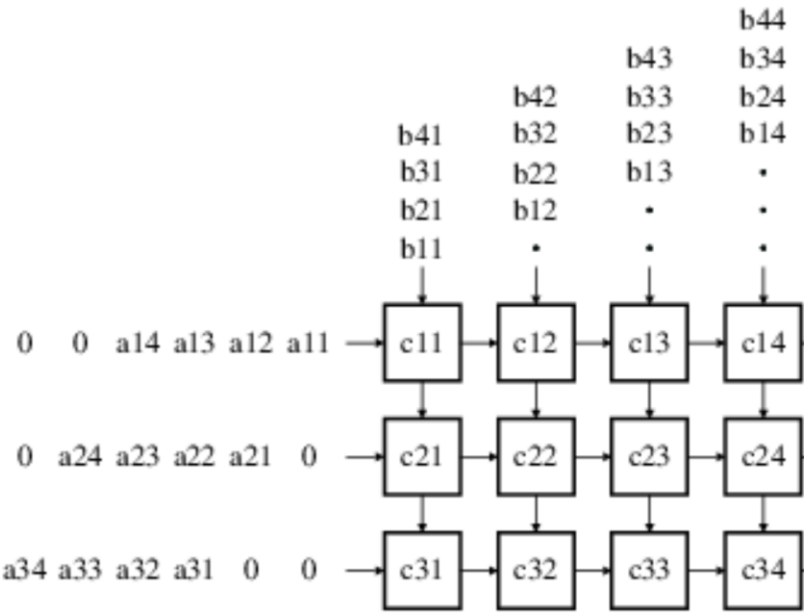
高階合成技術於應用加速 LAB_B Systolic array

電機所 110061555 李冠霈

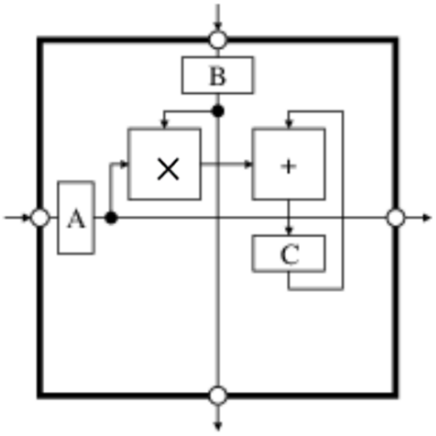
1. Systolic array 介紹

Systolic array 是一種適用於矩陣乘法的硬體架構，可以將已經進行過計算的矩陣數值直接傳遞到下一級的 PE 中，避免再將 matrix A 和 matrix B 存進 SRAM 再讀取出來繼續運算，並且會將每次計算出來的 partial sum 儲存在 PE 內部的 register 中，等下一個 input 進來運算時，再次累加 partial sum，因此可以降低整體運算所消耗的能量。而現在 Systolic array 被大量使用在 AI 加速器中，進行 convolution 運算，達到更高的 data reuse 次數。

圖一為計算 $\begin{bmatrix} c_{11} & c_{12} & c_{13} & c_{14} \\ c_{21} & c_{22} & a_{23} & c_{24} \\ c_{31} & c_{32} & a_{33} & c_{34} \end{bmatrix} = \begin{bmatrix} a_{14} & a_{13} & a_{12} & a_{11} \\ a_{24} & a_{23} & a_{22} & a_{21} \\ a_{34} & a_{33} & a_{32} & a_{31} \end{bmatrix} \cdot \begin{bmatrix} b_{41} & b_{42} & b_{43} & b_{44} \\ b_{31} & b_{32} & b_{33} & b_{34} \\ b_{21} & b_{22} & b_{23} & b_{24} \\ b_{11} & b_{12} & b_{13} & b_{14} \end{bmatrix}$ 示意圖



圖一：利用 systolic array 進行 matrix A 和 matrix B 相乘之示意圖



圖二：PE 內部架構圖

2. 程式碼及 pragma 說明

在本次 Lab 中，會實作將兩個矩陣大小為 16*16 的矩陣相乘，由 host 送入的 matrix A 和 matrix B 皆為一維矩陣(a、b)，會由 read A 及 read B 兩個 block 將 matrix A 和 matrix B 轉換成二為矩陣(local A、local B)的形式，以方便後續運算。

```
readA:
    for (int loc = 0, i = 0, j = 0; loc < a_row * a_col; loc++, j++) {
#pragma HLS LOOP_TRIPCOUNT min = c_size* c_size max = c_size * c_size
        if (j == a_col) {
            i++;
            j = 0;
        }
        localA[i][j] = a[loc];
    }
readB:
    for (int loc = 0, i = 0, j = 0; loc < b_row * b_col; loc++, j++) {
#pragma HLS LOOP_TRIPCOUNT min = c_size* c_size max = c_size * c_size
        if (j == b_col) {
            i++;
            j = 0;
        }
        localB[i][j] = b[loc];
    }
```

接下來的 block 會進行 16*16 的 matrix A 與 16*16 matrix B 相乘，並將相乘的結果存到 matrix C 中，算式如下:

$$\begin{bmatrix} C[0][0] & \cdots & C[0][16] \\ \vdots & \ddots & \vdots \\ C[16][0] & \cdots & C[16][16] \end{bmatrix} = \begin{bmatrix} A[0][0] & \cdots & A[0][16] \\ \vdots & \ddots & \vdots \\ A[16][0] & \cdots & A[16][16] \end{bmatrix} \cdot \begin{bmatrix} B[0][0] & \cdots & B[0][16] \\ \vdots & \ddots & \vdots \\ B[16][0] & \cdots & B[16][16] \end{bmatrix}$$

下方 for 迴圈的運算，會先將 A[0][0]和 B[0][0]到 B[0][16]個別相乘，並將 partial sum 存在 C[0][0]到 C[0][16]中，接下來再將 A[1][0] 和 B[0][0]到 B[0][16]個別相乘並將 partial sum 存在 C[1][0]到 C[1][16]中，直到完成全部矩陣乘法。

```
systolic1:
    for (int k = 0; k < a_col; k++) {
#pragma HLS LOOP_TRIPCOUNT min = c_size max = c_size
    systolic2:
        for (int i = 0; i < MAX_SIZE; i++) {
#pragma HLS UNROLL
        systolic3:
            for (int j = 0; j < MAX_SIZE; j++) {
#pragma HLS UNROLL
                // Get previous sum
                int last = (k == 0) ? 0 : localC[i][j];

                // Update current sum
                // Handle boundary conditions
                int a_val = (i < a_row && k < a_col) ? localA[i][k] : 0;
                int b_val = (k < b_row && j < b_col) ? localB[k][j] : 0;
                int result = last + a_val * b_val;

                // Write back results
                localC[i][j] = result;
            }
        }
    }
```

在 Systolic array 的 block 中，利用 HLS_LOOP_TRIPCOUNT 的 pragma 來計算完成運算所需要的時間，pragma HLS UNROLL 則是可以讓該個 for 進行平行運算，加快運算效率。

Write C 的 block 則是將完成計算的結果，也就是 matrix C，由二維的矩陣轉換成一維的矩陣，直接輸出一維矩陣 C。

```
writeC:
    for (int loc = 0, i = 0, j = 0; loc < c_row * c_col; loc++, j++) {
#pragma HLS LOOP_TRIPCOUNT min = c_size* c_size max = c_size * c_size
        if (j == c_col) {
            i++;
            j = 0;
        }
        c[loc] = localC[i][j];
    }
}
}
```

在這個 LAB 中，我使用了 HLS ARRAY_PARTITION 的 pragma 進行加速，將 matrix A 以 row 的方向切割，matrix B 以 column 的方向切割，增加一次可以讀取的數值，加快運算效率，而 matrix C 的 dim 為 0 指的是最終會將 local C 的數值(16*16*bit 數)全部儲存在一個 register 中，加快傳輸效率。

```
#pragma HLS ARRAY_PARTITION variable = localA dim = 1 complete

    int localB[MAX_SIZE][MAX_SIZE];
#pragma HLS ARRAY_PARTITION variable = localB dim = 2 complete

    int localC[MAX_SIZE][MAX_SIZE];
#pragma HLS ARRAY_PARTITION variable = localC dim = 0 complete
```

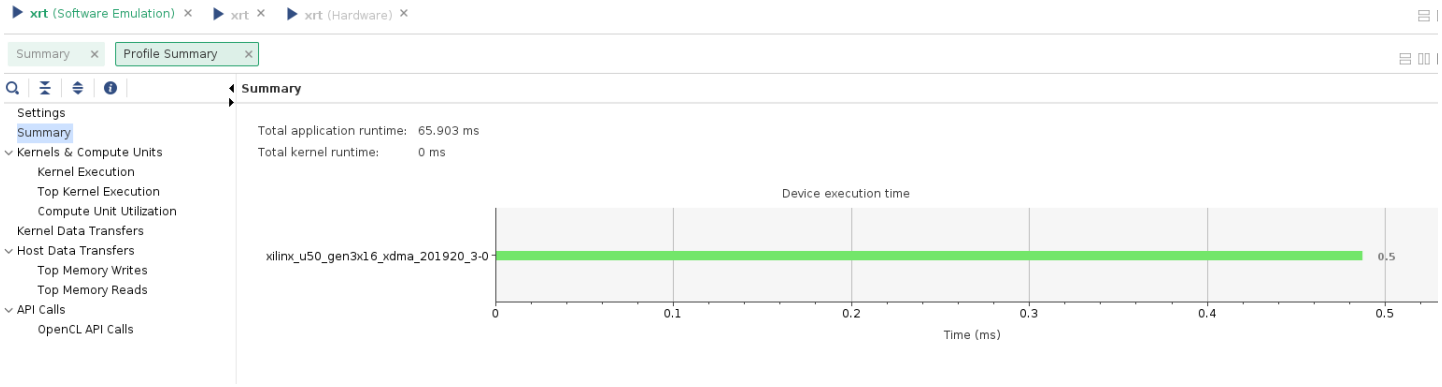
3. Timing and performance

本次 Lab 我實作了兩種狀況，一種是不加入 Unroll 和 array partition 的 pragma，另一種則是兩種 pragma 都加入，不論是哪一種狀況都可以發現使用 CPU 運算的 Software emulation 所花費的 Device execution time 最長;使用硬體加速的 Hardware emulation 時間最短，且 Device execution time 是以 cycle 數乘以週期得到;而 Hardware 的 Device execution time 又比 Hardware emulation 稍微長一些，這是因為在 FPGA 上運行需要更多的資料傳輸時間。而比較是否加入 pragma 的兩種狀況可以發現，使用 pragma 加速的 Hardware emulation 和 Hardware 的 Device execution time 都明顯變短，代表能提供更有效率的運算。

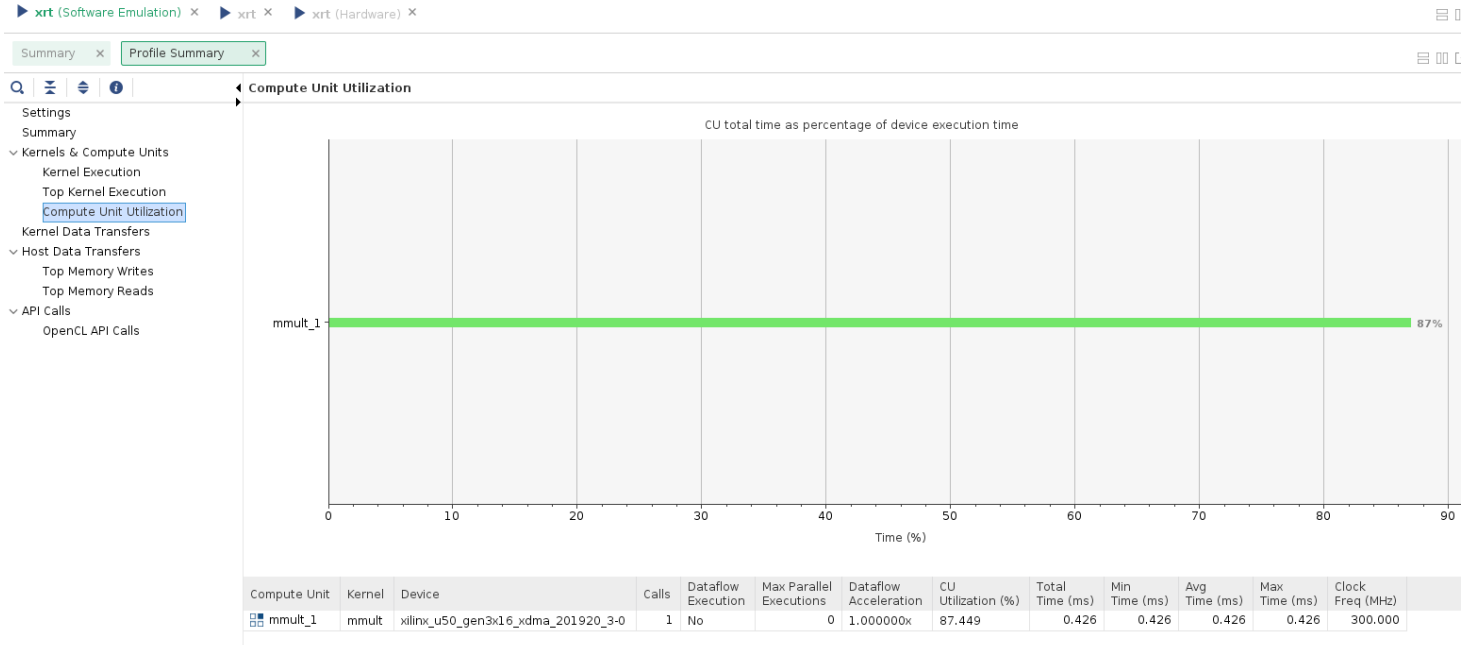
由 CU utilization 則可以看到再把資料從 local buffer 送進 mmult 中，到完成計算時，mmult 計算所花費的時間比例，由於讀取資料的時間不會改變，因此 CU utilization 的比例變低，就可以證明計算所需的時間降低，所以可以說明 Unroll and array partition 大幅降低了計算所需的時間。

mmult		Original	Unroll and array partition
Software emulation	Total application run time	70.6691ms	65.903ms
	Device execution time	0.52ms	0.48ms
	CU utilization	0.469ms (86%)	0.426ms (87%)
Hardware emulation	Total application run time	43069ms	42703.5ms
	Device execution time	0.022ms	0.01ms
	CU utilization	0.018ms (84%)	0.004ms (69%)
Hardware	Total application run time	5674.54ms	4926.1201ms
	Device execution time	0.35ms	0.028ms

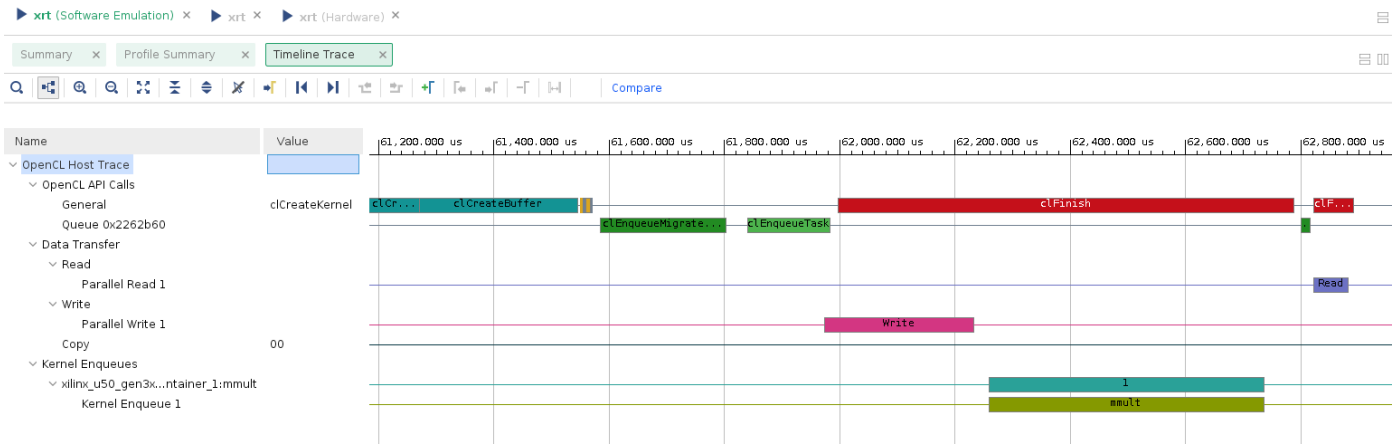
(1) 加入 Unroll and array partition pragma 之 Software emulation 結果:



圖三:Software emulation Device execution time (Unroll and array partition)



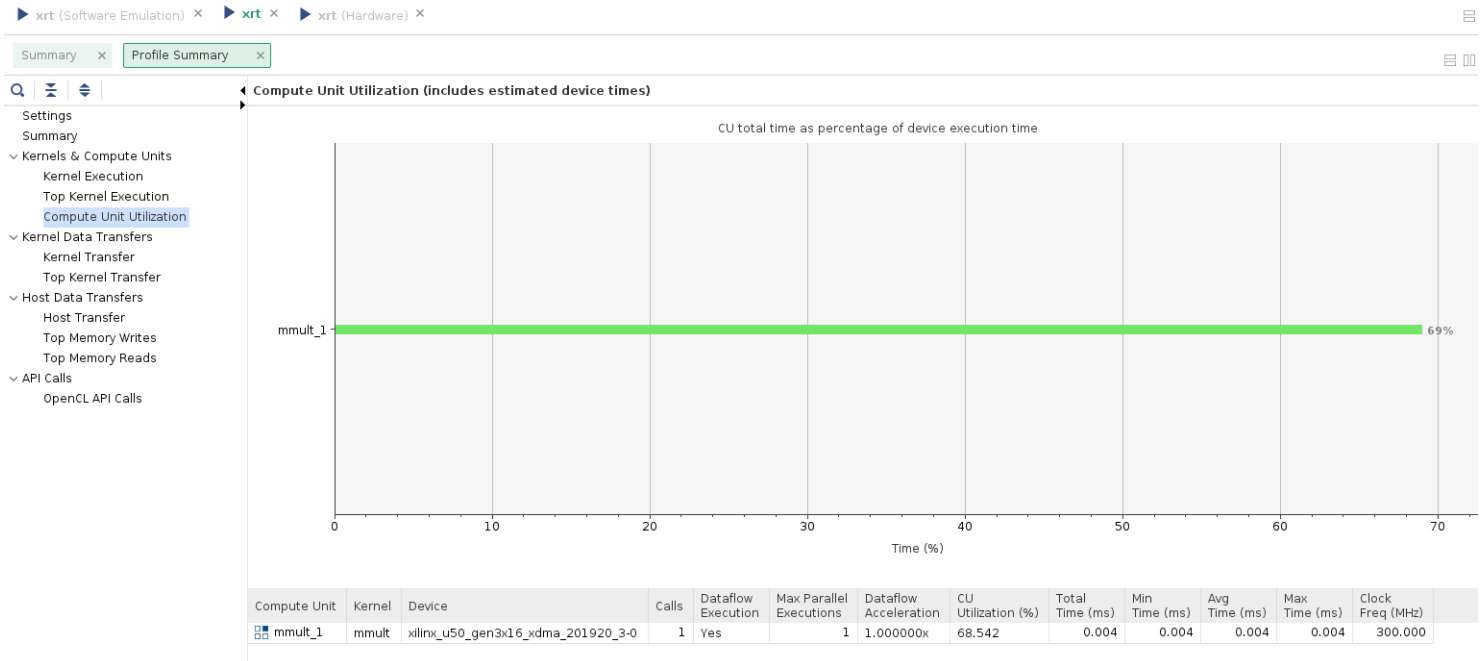
圖四:Software emulation compute unit utilization (Unroll and array partition)



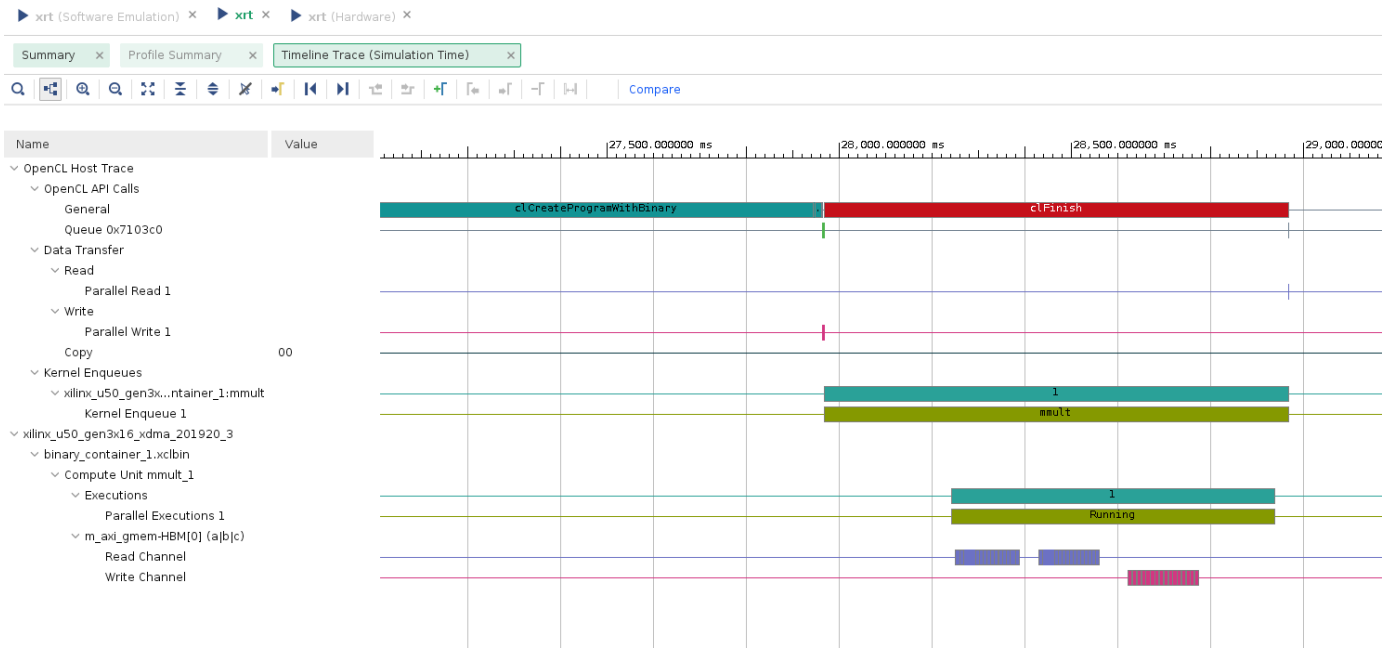
圖五:Software emulation Timeline Trace (Unroll and array partition)

(2) 加入 Unroll and array partition pragma 之 Hardware emulation 結果

圖六:Hardware emulation Device execution time (Unroll and array partition)

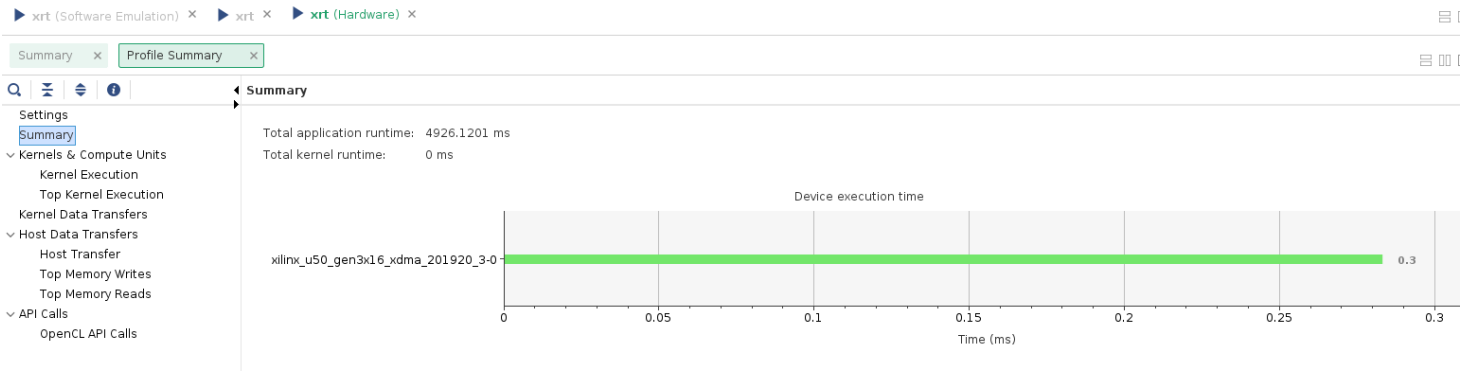


圖七:Hardware emulation compute unit utilization (Unroll and array partition)

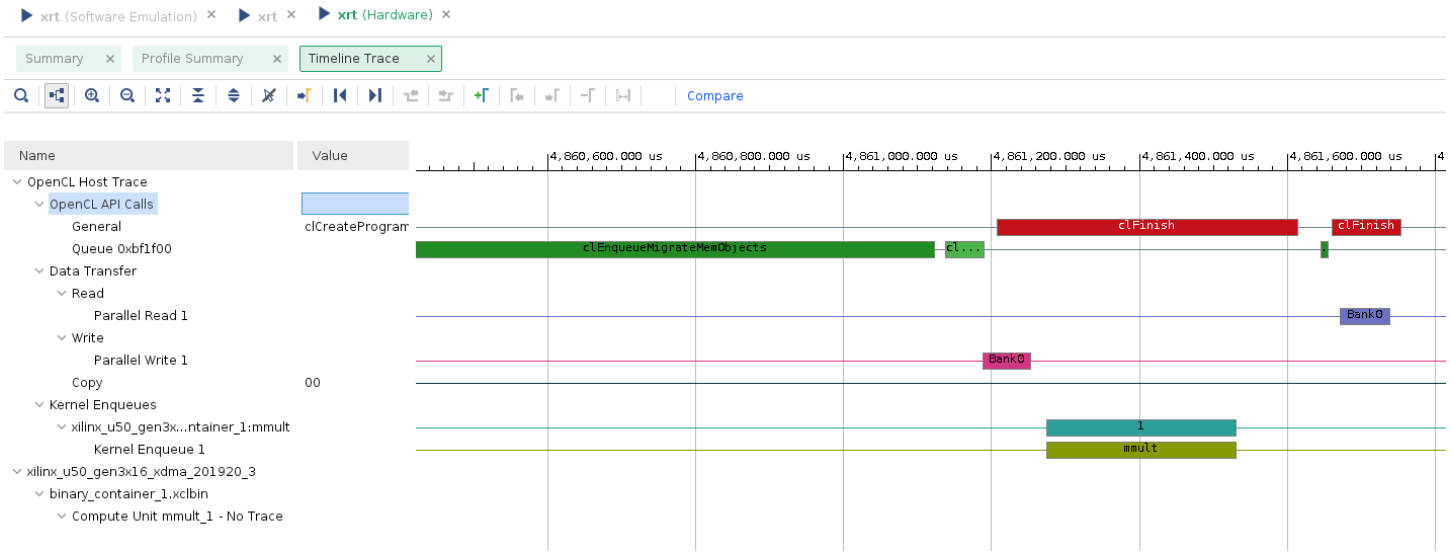


圖八:Hardware emulation Timeline Trace (Unroll and array partition)

(3) 加入 Unroll and array partition pragma 之 Hardware 結果

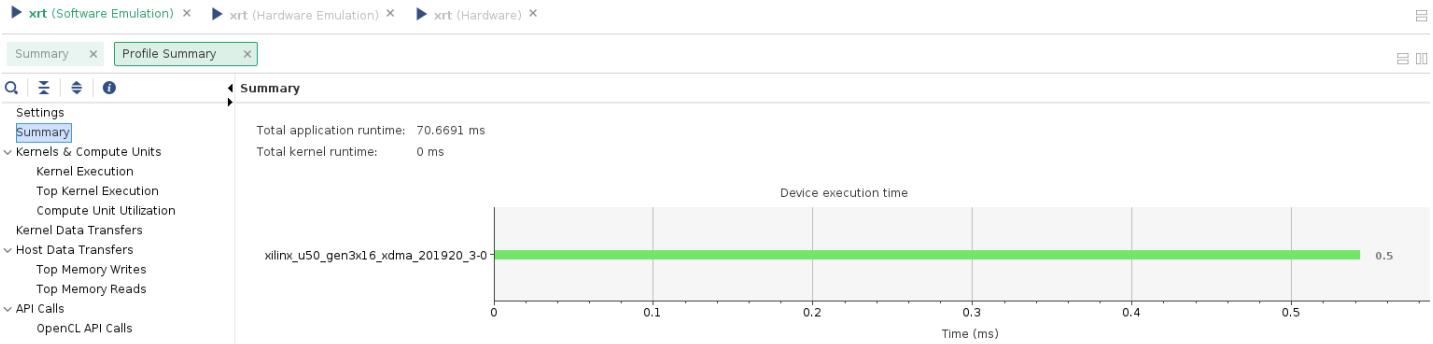


圖九:Hardware Device execution time (Unroll and array partition)

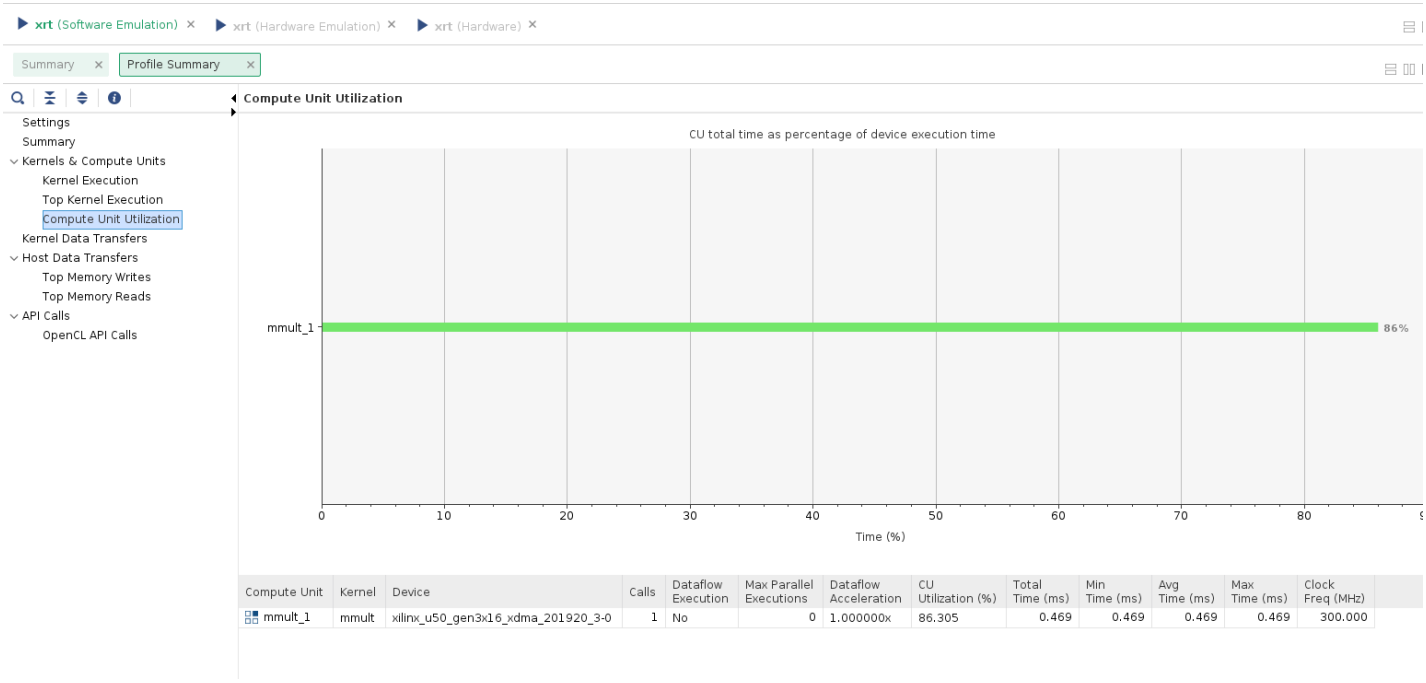


圖十: Hardware Timeline Trace (Unroll and array partition)

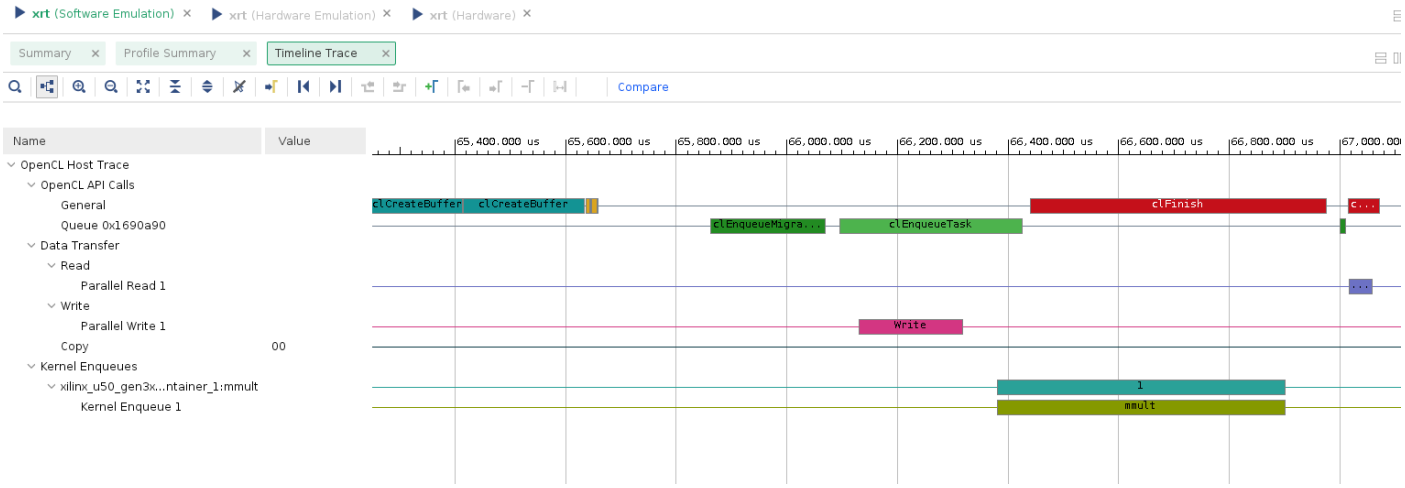
(4) 沒有加入 Unroll and array partition pragma 之 Software emulation 結果



圖十一:Software emulation Device execution time

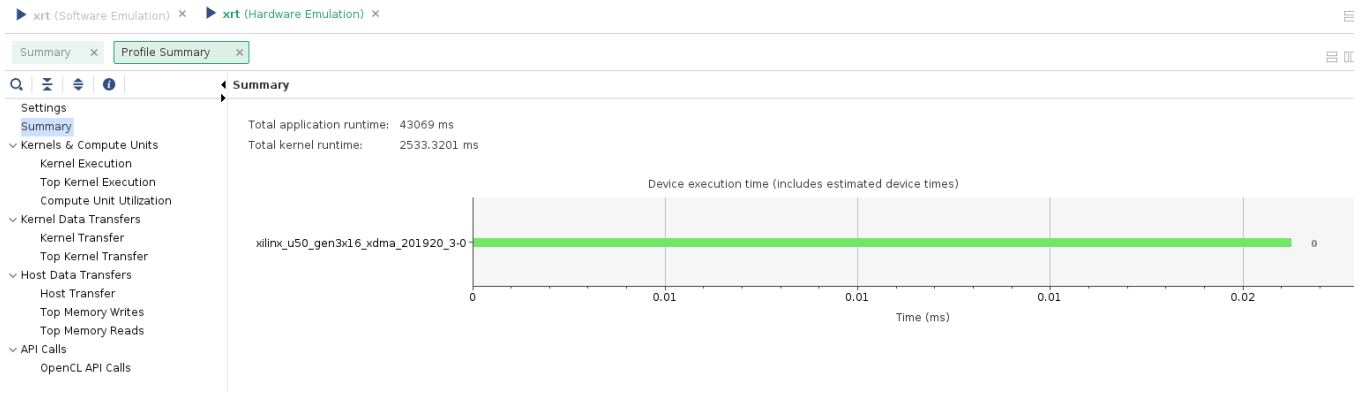


圖十二: Software emulation compute unit utilization

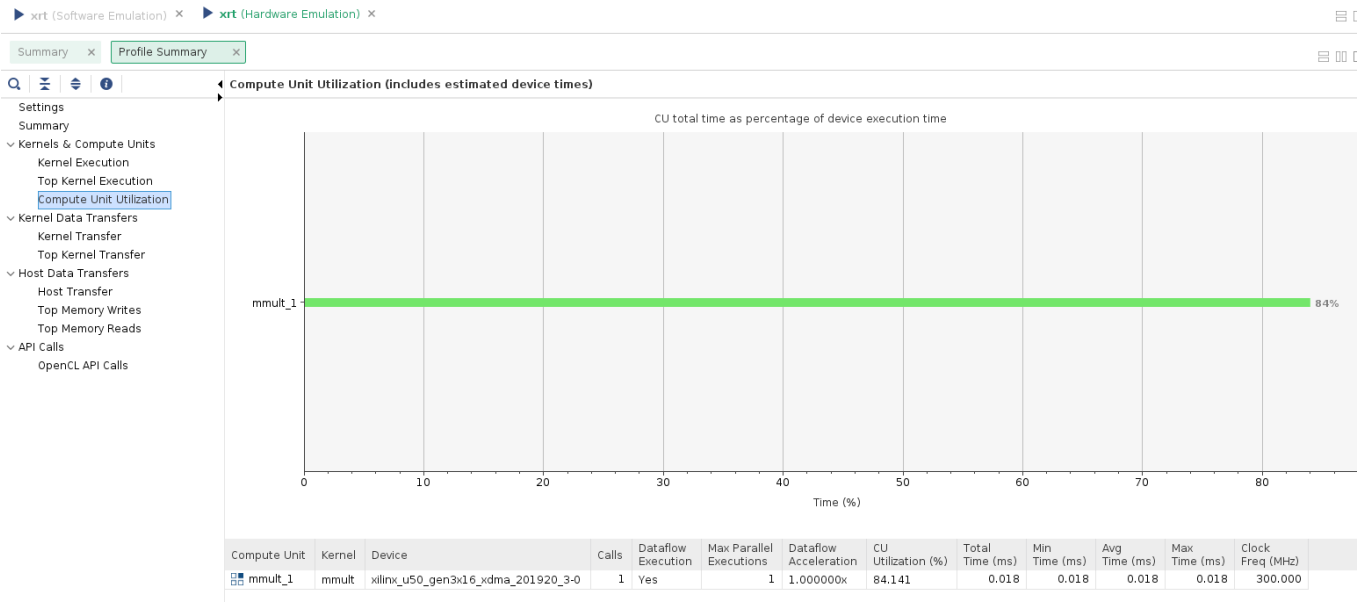


圖十三: Software emulation Timeline Trace

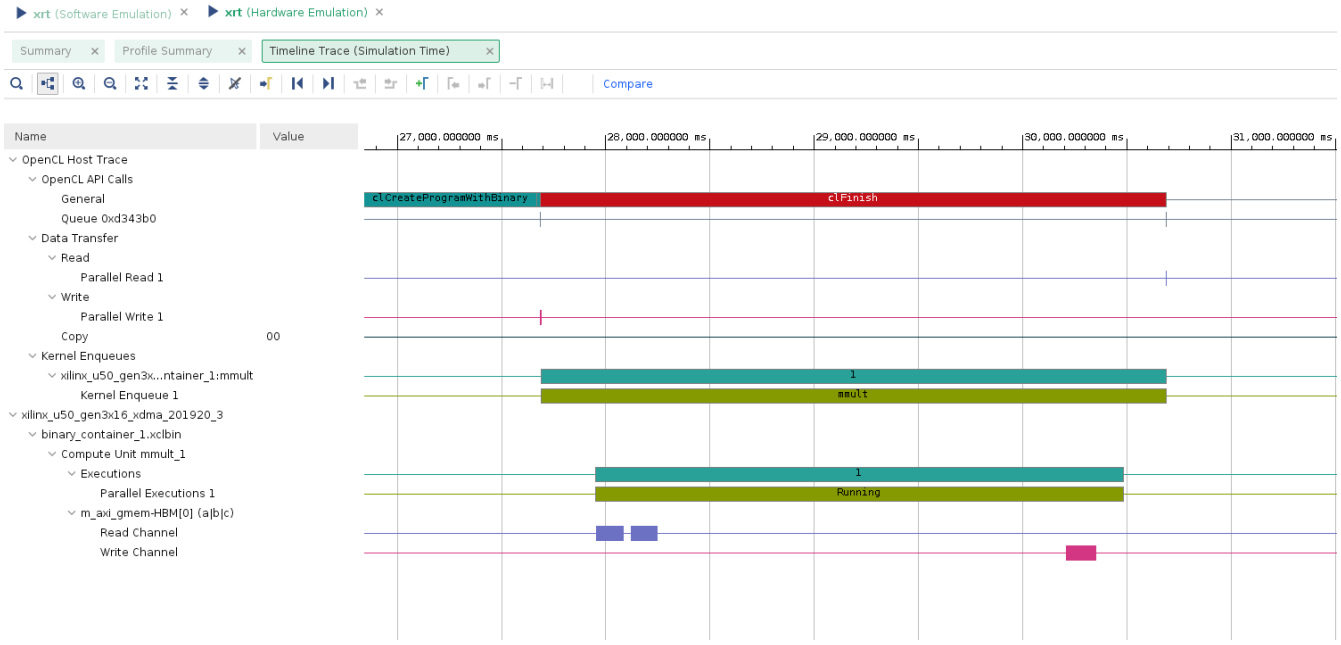
(5) 沒有加入 Unroll and array partition pragma 之 Hardware emulation 結果



圖十四:Hardware emulation Device execution time

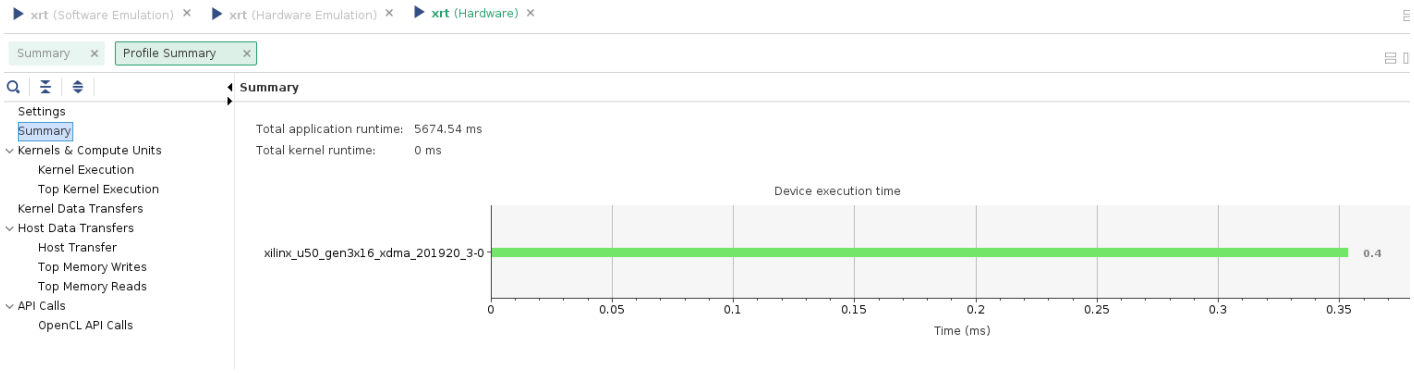


圖十五: Hardware emulation compute unit utilization

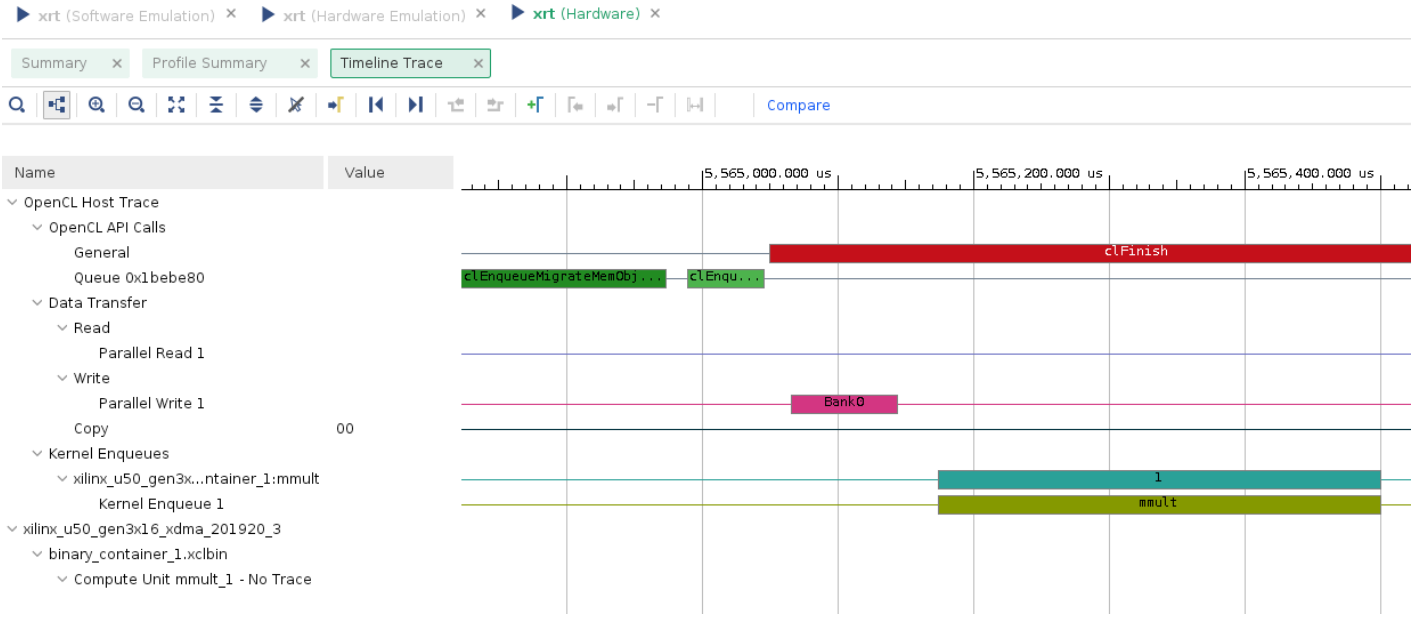


圖十六: Hardware emulation Timeline Trace

(6) 沒有加入 Unroll and array partition pragma 之 Hardward 結果



圖十七:Hardware Device execution time



圖十八: Hardware Timeline Trace

4. 心得

在本次 Lab 中，我參考了 Lab 3 的實作步驟，並多加入了 hardware 的結果與 Software emulation 及 Hardware emulation 進行比較，也藉由本次 Lab 驗證了 pragma 確實可以優化電路的計算效率。

在一開始實作時，設定 Program Arguments 的部分我使用了和 Lab 3 相同的 arguments，但在 emulation 時一直沒有結果，最後發現在 Lab B 時只有一個參數，因此只需填寫”binary_container_1.xclbin”即可，而在模擬 Original 和 Unroll and array partition 兩種狀況時，當修改了 mmult 中的程式碼，再重新模擬時，會一直跳出錯誤，後來發現好像系統會讀到未修改過的程式碼，而造成未知的錯誤，因此建立兩個 project 分開模擬是比較保險的方式。

5. Github 網址:

https://github.com/kuanpei/Lab_B_Systolic_array