LabA-Design optimization

Overall system:

本次 lab 探討了如何優化最基本的 module: 矩陣乘法。在 lab 中，使用了兩個 3*3 的矩陣作乘加運算。

Project 設定:

Device: xc7z020-clg400-1

Period: 10ns

Source code:

```
void matrixmul(
      mat_a_t a[MAT_A_ROWS][MAT_A_COLS],
      mat_b_t b[MAT_B_ROWS][MAT_B_COLS],
      result_t res[MAT_A_ROWS][MAT_B_COLS])
{
  // Iterate over the rows of the A matrix
   Row: for(int i = 0; i < MAT_A_ROWS; i++) {
      // Iterate over the columns of the B matrix
      Col: for(int j = 0; j < MAT_B_COLS; j++) {
         res[i][j] = 0;
         // Do the inner product of a row of A and col of B
         Product: for(int k = 0; k < MAT_B_ROWS; k++) {
            res[i][j] += a[i][k] * b[k][j];
         }
      }
   }

}
```

如上圖所示。新的矩陣 res 總共有 3*3 個位置，對應了最外層的兩層 loop，而每個位置需要作 3 次乘加，對應了最內層的 loop。另外在每個位置計算前會先將值初始為 0。

1. No pipeline

Performance

* Loop:

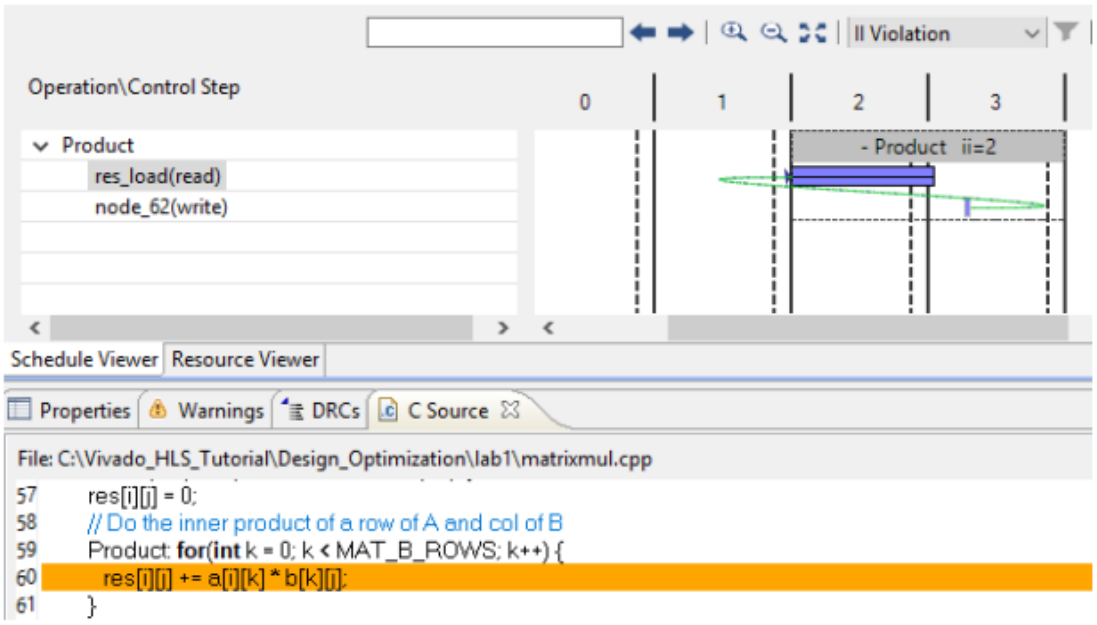
Loop Name	Latency (cycles) min	Latency (cycles) max	Iteration Latency	Initiation Interval achieved	Initiation Interval target	Trip Count	Pipelined
- Row	159	159	53	-	-	3	no
+ Col	51	51	17	-	-	3	no
++ Product	15	15	5	-	-	3	no

Latency: 最裡層的 Product loop，Iteration latency 為 5 個 cycle，共 3 個 Iteration latency，所以總共 5*3=15 個 cycle。而外面一層的 Col loop，在進入

和離開 Product loop 時各需要一個額外的 cycle，故 Iteration latency 為 15+2=17。以此類推到最外層的 Latency 為(17*3+2)*3=159。

2. Pipeline Product loop

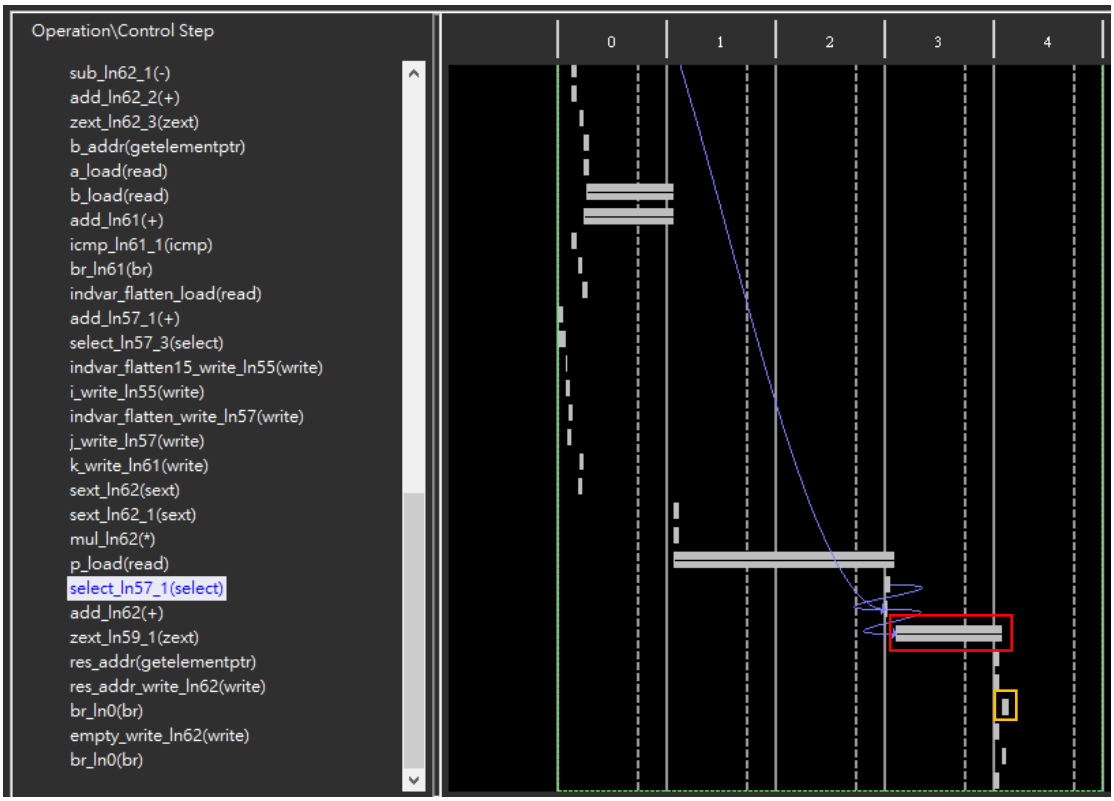
在最內層的 Product loop 加上 pragma HLS pipeline。和官方的教學不同，這個版本沒有 Waring 且 II 為 1，iteration latency 為 5 個 cycle。根據官網的教學，此設定有兩個問題。一是在 Product loop 外還有一個將 res 初始化的操作，導致無法將 Product loop flatten；二是因為 data dependency 的問題，因為 res 作了+=的操作，故在第一個 iteration 時還在 write 時，第二個 iteration 還沒有辦法 read，導致 II 為 2，如下圖。



猜測可能是因為版本或設定的不同，讓 tool 作了不同的優化。

對於第一個問題，看了 Cosim 產生的 Verilog code，發現 tool 把這個初始化的值利用控制信號加到 MAC 裡面。

第二個問題，相關的 schedule 如下圖:

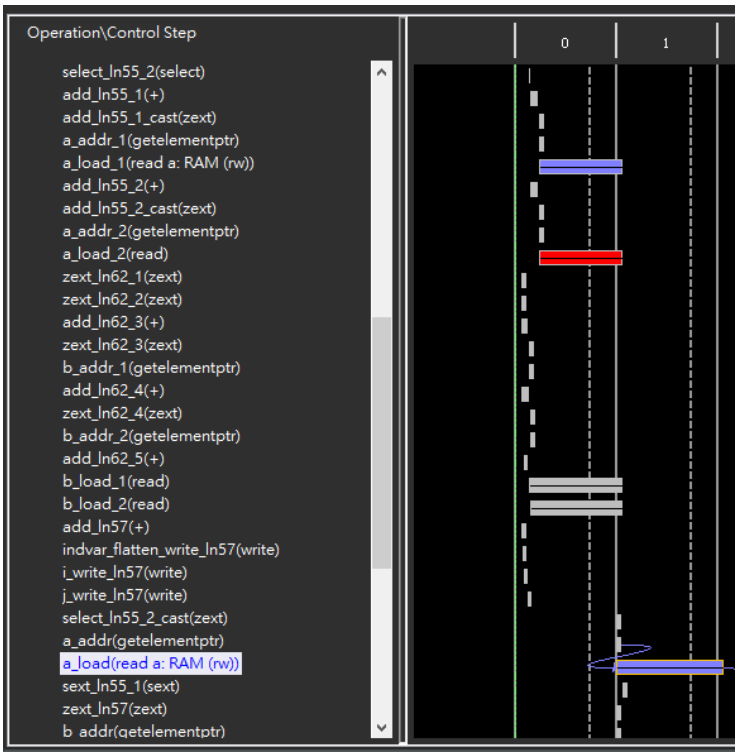


紅框內的操作為加法，輸入分別為乘法的輸出和一個 multiplexer。而 multiplexer 的輸入又分別可能是初始化的值或是 res read 的值。另外黃框內的操作為 res write，可以看到相較官網的早了許多，可能因為如此所以避免了 data dependency 的問題。

3. Pipeline Col loop

在 Col loop 加上 pragma HLS pipeline。因為會 unroll 裡層的 Product loop，故會一次做完 Product loop 裡的操作。因為其中的 a 和 b matrix 是用 block RAM 實作，最多支援兩個 port 的讀寫，而 res 的計算需要 3 個 a 和 b matrix 的值，故 RAM 的頻寬不夠大，無法使得 II 為 1，如下圖所示。

```
* Loop:
+-----------+---------+---------+----------+-----------+-----------+------+----------+
|           |  Latency (cycles) | Iteration|  Initiation Interval  | Trip |          |
| Loop Name |   min   |   max   |  Latency |  achieved |   target  | Count| Pipelined|
+-----------+---------+---------+----------+-----------+-----------+------+----------+
|- Row_Col  |       21|       21|         6|          2|          1|     9|       yes|
+-----------+---------+---------+----------+-----------+-----------+------+----------+
```



其中最一開始的紅藍為 RAM 的兩個 read port，讀取了兩個 a matrix 裡的值。第二個藍色代表讀取第三個值。

4. Reshape the Arrays

上個方法的問題可以使用 reshape 來解決。此方式能夠將 array 切割到不同的記憶體以增加頻寬。

Directive:

```
############################################################
## This file is generated automatically by Vitis HLS.
## Please DO NOT edit it.
## Copyright 1986-2022 Xilinx, Inc. All Rights Reserved.
############################################################
set_directive_top -name matrixmul "matrixmul"
set_directive_array_reshape -type complete -dim 2 "matrixmul" a
set_directive_array_partition -type complete -dim 1 "matrixmul" b
set_directive_pipeline "matrixmul/Col"
```

其中 a 和 b 切割的 dimension 分別是 2 和 3。

```
* Loop:
+-----------+---------+---------+----------+-----------+-----------+------+----------+
|           |  Latency (cycles) | Iteration|  Initiation Interval  | Trip |          |
| Loop Name |   min   |   max   |  Latency |  achieved |   target  | Count| Pipelined|
+-----------+---------+---------+----------+-----------+-----------+------+----------+
|- Row_Col  |       12|       12|         5|          1|          1|     9|       yes|
+-----------+---------+---------+----------+-----------+-----------+------+----------+
```

可以看到增加頻寬後 II 變成 1，latency 也減少了。

5. Apply FIFO Interfaces

另外我們也可以嘗試使用 FIFO 的方式來傳送資料。然而目前的 C code 對資料的提取是 random access 的形式，故若直接下 FIFO 的 pragma 無法達成預

期的結果。後面會提出解法。

6. Pipeline the Function

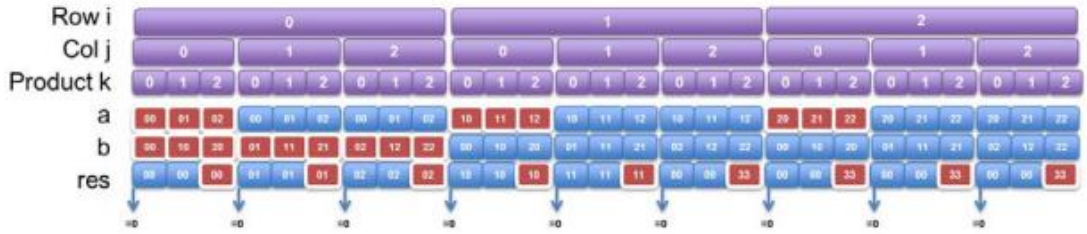
最後我們嘗試在 function level 作 pipeline，並與前面的結果作比較。

	Latency	Resource		
		DSP	FF	LUT
No pipeline	159	1	46	186
Product loop	32	1	238	406
Col loop	14	2	171	292
Function	8	18	463	816

可以看到在 no pipeline 和整個 function 作 pipeline 都太過極端，而在 inner loop 作適當的 pipeline 才有機會作合理的加速。

7. C Code Optimized for I/O Accesses

最後根據上面提到的 FIFO 問題，用下面這張圖作解釋。



FIFO 的條件是 data 必須是 Sequence 的進來，因次不能索要重複的資料，必須在 function 裡面創建 cache 自己儲存起來。上途中是整個 function 在計算過程中需要資料的流程圖，其中紅色是各個資料第一次進來的時候，可以讀取，其餘藍色的部分則只能讀取自己預先儲存好的。

修改後的 C code 如下圖

```
  mat_a_t a_row[MAT_A_ROWS];
  mat_b_t b_copy[MAT_B_ROWS][MAT_B_COLS];
  int tmp = 0;

  // Iterate over the rowa of the A matrix
  Row: for(int i = 0; i < MAT_A_ROWS; i++) {
    // Iterate over the columns of the B matrix
    Col: for(int j = 0; j < MAT_B_COLS; j++) {
#pragma HLS PIPELINE rewind
      // Do the inner product of a row of A and col of B
      tmp=0;
      // Cache each row (so it's only read once per function)
      if (j == 0)
        Cache_Row: for(int k = 0; k < MAT_A_ROWS; k++)
          a_row[k] = a[i][k];

       // Cache all cols (so they are only read once per function)
     if (i == 0)
           Cache_Col: for(int k = 0; k < MAT_B_ROWS; k++)
              b_copy[k][j] = b[k][j];

      Product: for(int k = 0; k < MAT_B_ROWS; k++) {
        tmp += a_row[k] * b_copy[k][j];
      }
      res[i][j] = tmp;
    }
  }
}
```

修改的部分為增加了額外的 cache: a_row, b_copy。先將讀取到的資料儲存在裡面，計算的時候再從裡面拿取。因為是自己的 cache，所以可以 random access。

和前面的 Col pipeline 作比較

	Latency	Resource		
		DSP	FF	LUT
Col loop	14	2	171	292
FIFO	14	2	399	453

雖然整體用了更多硬體資源而 latency 維持一致，但對於外部資料的提取變成了 FIFO/Stream 的形式，這種方式在某些情況可以減少 block 間的溝通，kernel 也只需要作 Steam in, Stream out，是比較好的資料傳輸方式。

Problem:

由於版本不太相同導致有些內容無法和官方教學的一致，例如最一開始 no pipeline 的部分，若都不加 pragma，tool 會自己作優化，須加上 pragma HLS PIPELINE off 才能合成出想要的電路。

Observed and Learned:

在這次 lab 學會了基本 pipeline 的用法，以及如何解決不同層級下 pipeline 會遇到的問題，例如增加頻寬。過程中 debug 時，也還是會忍不住去看 Cosim 產生的 Verilog，這樣或許就失去使用 HLS 的意義了。若要真的利用 HLS 來加快硬體設計的速度，還是要學會如何對 C code 下正確的 pragma。