HW-SW Optimization

湯景彤 , 0316301

I. INTRODUCTION

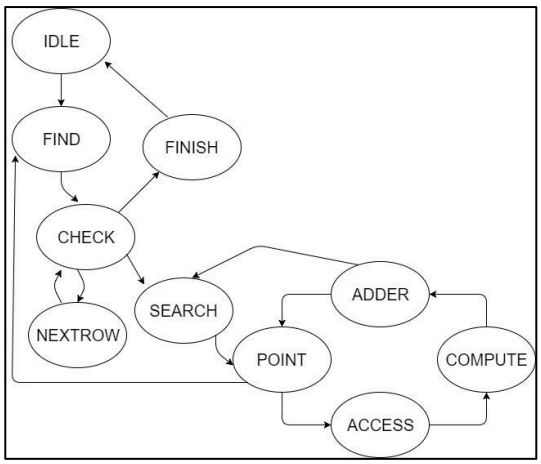
在這次 Lab6 中，我們修改並加速 lab3 的行為，lab3 是在做計算圖片中同上一個點，在兩個時間點移動的距離程式。分析如何修改軟體及硬體程式碼，優化這個程式，使軟體及硬體兩者之間能相互配合，最佳化軟硬體協同設計，讓整個程式能在最短的時間內完成。

II. BEFORE OPTIMIZATION

在尚未修改優化 lab3 的 code 之前，在 release mode 下，執行程式的結果如下：

The motion vectors have a mean of 3.8 pixels.
The motion vectors range between 0.0 and 17.9 pixels.
It took 311 milliseconds to filter the two images.
It took 407326 milliseconds to estimate the motion field.

因為我在做 lab3 的時候是用 block copy function，先用 slave registers 接收 48*48 pixels 的 sa_bank (search window)及 16*16 pixels 的 cb_bank (current block) ，接下來的 finite-state machine 一共開了 10 個 states（如下示意圖 1），在 FIND 時會找到 search window 中的一個 pixel 和對應到的 current block 中的一個 pixel，並在 ACCESS 時用 register 接收這兩個點的值，並在 COMPUTE 時計算 sad，再用 ADDER 把個點的 sad 加總起來，而其他 state 都是用來判斷點的移動。



這樣的 me_match verilg code 執行算完全部的圖要花將近 40 秒，在 lab1 時這個計算如果全部都用軟體做，只需 10 秒，可以知道我在做 lab3 時 finite-state machine 設計的非常不好，完全沒用到硬體能平行計算加速的特性，所以我使用以下幾種方式，嘗試修改軟體及硬體，使計算各點移動距離能在最短的時間完成。

III. HARDWARE _BASED OPTIMIZATION

A. Data Transfer Time

因為原本軟體是用 block copy function 的方式，當被呼叫到後，memmove 才會告訴硬體從記憶體的哪個位置拷貝 bsize 個字元， 並在硬體用 11 個 registers 接收目前要計算的 image 區域。因為每次算完一塊 48*48 的區域，都要等軟體呼叫這個 function 硬體才能載入圖片資料，讀完 data 才開始計算 sad，因此我想要讓硬體載入圖片及計算行為能平行化執行，所以把載入資料的方式改成我們在 lab4 學習的 burst data transfer，設計一個 master IP 讀寫 DRAM 上個資料，在這個 IP 有三個 states，IDLE->INIT_READ->INIT_WRITE，會主動去 DRAM 上抓資料寫到 buffer 裡，我設計一個 output 把 buffer 的 data 傳到 slave IP，而在 slave IP 中用 input wire 接收 data，讀取 search window 及 current block 的方式就變成是用這個 input wire，在我設計的 finite-state machine 執行到 READ_SA/READ_CB state 時，就會把 input wire 的值存到 sa_bank/cb_bank。

```
always @( posedge S_AXI_ACLK )
begin
    if(state==READ_SA&&read_index<12&&rnext) begin
            sa_bank[y][383-(read_index*32)-:32]<=data_in;
    end
    else if(state==READ_CB&&read_index<4&&rnext) begin
            cb_bank[y][128-(read_index*32)-:32]<=data_in;
    end
```

//search window, current block data 存取方式

此外加上 shift registers 的功能，在 SLAVE IP 多設計兩個 states 是當算完一張 48*48 image 時，把原本的 sa/cb_bank 往上移一列，並且 output wire burst_write 告訴 MASTER IP 要開始讀 buffer 中下一列的 data，且把這張圖的 min_x,y,min_sad 傳到一個用來記錄每張圖的 min_sad 值的 sram 中，所以我在 MASTER IP 原本的三個 state 加上了 INIT_WRITE 的 state，是當收到 burst_write 時，就把 SLAVE 傳來的 min data 存到 sram 中，當硬體算完所有 block images 後，再把這個 sram 記錄各點的值印在 SDK 上。

但可能因為我不太熟悉 MASTER IP 的設計方式，設計出來的 MASTER, SLAVE IP 雖然跑得動，也能把 data 存到我要的位置，但 sram 存的 min data 一直是錯的，而且使用的 resources 非常大，LUT 及 Flip-Flop 都快五萬多，且合成的時間非常久，timing 也時常爆炸，所以我改變嘗試別的方式，希望能達到快速且 resource 也很

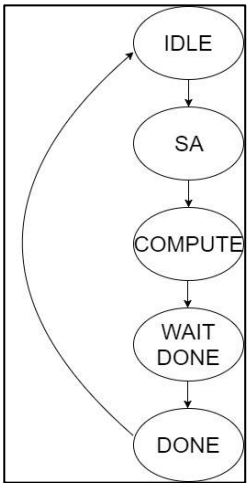
小的做法。

B. Compute sad

因為進入 fsm 一次計算一點的移動距離，再把 256 點的值加起來實在太慢了，所以我把它改成一次能算 256 pixels 的其中一列，也就是 16 個點，所以算一張圖就只要進入 fsm 16 次。在 release mode 下，執行程式的結果如下：

```
The motion vectors have a mean of 3.8 pixels.
The motion vectors range between 0.0 and 17.9 pixels.
It took 311 milliseconds to filter the two images.
It took 78156 milliseconds to estimate the motion field.
```

比一次算一點快了 5.7 倍。接著我利用 lab2 學習的 adder tree 的概念，讓程式進入硬體的時候就能一次算完一張 256 pixels 的圖，也把原本零亂設計不良的 finite-state machine 大幅修改成以下的形式。



新的 fsm 一共有 5 個 states，IDLE->SA->COMPUTE ->WAIT_DONE->DONE，SA state 的功能是把 256 pixels 的 data 從 sa_bank 存到我開的 register sa_buf，當跑到 COMPUTE state 時代表 sa_buf 已經存好我要的 data，且因為我開了 wire [8:0]diff[0:255] 並 assign 他 sa_buf 減對應到 cb_bank 的值，因此他可以及時算出兩者之差，下圖為這部分的程式碼。

```
generate
    for(ii=0;ii<16;ii=ii+1) begin
        for(j=0;j<16;j=j+1) begin
            assign diff[ii*16+j] = sa_buf[ii][127-j*8-:8] - cb_bank[ii][127-j*8-:8];
        end
    end
endgenerate
```

算出來的 256 個 diff 取絕對值後，再用 adder tree 把他們加起來，這裡我有特別去實驗，看一個 clock 最多可以加幾個值，timing 不會爆炸，而我得到的實驗結果是最多可以一次加 16 個值，因此這裡我就讓它用 for 回圈執行 16 次，可得到 16 個部分 sad 值，在下一個 clock 把 16 個值加起來，就可以得到一張 256 pixels 的 sad 值。因為用 adder tree 加總答案一共需要 3 個 clock，所以需要一個 latency_cnt 控制，當它等於 3 時，才代表算出的 total sad 為正確值，才能回傳 min_x,y,sad 到軟體，並且把 x,y 坐標更新到新的點，不然就會是錯的值。 這部分也是我一開始沒有考慮到的地方，所以一直算出錯的值，看了波型圖好久，才找出 bug。

接著 WIAT_DONE state 是確定邊界的點有算到後，就跳到 DONE 表示這一張圖已經算好 min_sad，並用 registers 回傳 data 到軟體。

這樣的修改，在 release mode 下，執行程式的結果如下：

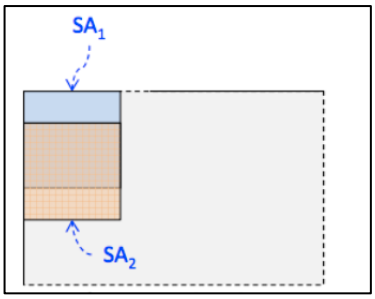
```
The motion vectors have a mean of 3.8 pixels.
The motion vectors range between 0.0 and 17.9 pixels.
It took 311 milliseconds to filter the two images.
It took 1025 milliseconds to estimate the motion field.
```

比一次算一列，快了將近 7 倍，接著我利用 shift registers 的概念，修改 SA state 的資料存取方式，以及 COMPUTE state，下圖為這部分的程式碼。

```
assign updata_data = sa_bank[updata_data_y][383-updata_data_x*8-:8*16];

always @ (posedge S_AXI_ACLK)
begin
    if(st==SA || st==COMPUTE)begin
        sa_buf[15] <= updata_data ;
        for(i=0;i<=14;i=i+1)begin
            sa_buf[i] <= sa_buf[i+1] ;
        end
    end
end
```

就是當在 SA 或 COMPUTE state 時，sa_buf 不用再去一個一個存取 sa_bank 的值，而是用 for 回圈把第一列到十四往上 shift 一列，到第 0 列到第 14 列，並只有第 15 列是要存取新的 data，updata_data_x,y 是控制現在要去 sa_bank 的哪個位置取值，下圖為 shift register 示意圖：



而且在 COMPUTE state 的時候，它在等 adder tree 加總的同時，因為已經運用目前的 sa_buf 值完畢，所以也可以同時更新新的 sa_buf data，可以達到硬體平行化運算的優點，但是因為我把 search window 的移動方式，從原本的一次往右移一格 block image，變成往下移一個，才可以使用 shift register 的方法，所以雖然算出來的 min_sad 是對的，但是 min_x,y 一直有部分點是錯的，後來 debug 許久，才知道是因為我把坐標控制方式改成往下移，所以當有兩個點是一樣小得值時，要加上以下判斷式，最小的 x,y 值才不會被更新成錯的。

```
    else if(total_sad==min_sad) begin
        if((real_y > min_y)|| ((real_y == min_y) && (real_x >= min_x))) begin
            min_sad<=total_sad;
            min_x<=real_x;
            min_y<=real_y;
        end
    end
end
```

以上這樣的修改，加上 shift register 以及加上 pipeline 運算，在 release mode 下，執行程式的結果如下：

The motion vectors have a mean of 3.8 pixels.
The motion vectors range between 0.0 and 17.9 pixels.
It took 311 milliseconds to filter the two images.
It took 738 milliseconds to estimate the motion field.

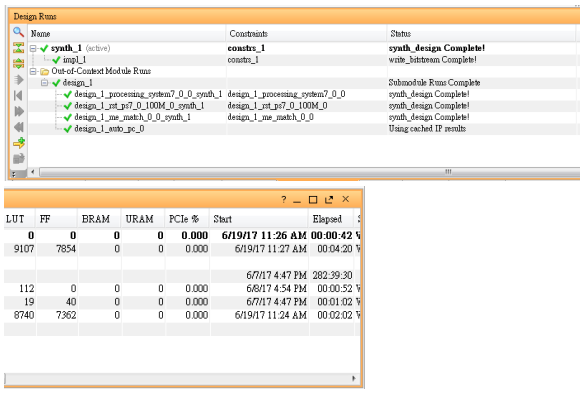
終於把計算時間縮短到一秒之內，但是這時候 resource 還是很大，我想可能是我 sa,cb_bank 的儲存方式沒有設計好，因此我決定修改 data transfer 的方式。原本的資料存取方式是這樣，先把 S_AXI_WDATA 存到 slv_reg0-11，在開 register 存到 bank 裡，下圖為部分程式碼：

```
if (slv_reg_wren) begin
    if(slv_reg12<=47) begin
        sa_bank[0 ][slv_reg12] <= slv_reg0[7:0];
        sa_bank[1 ][slv_reg12] <= slv_reg0[15:8];
        sa_bank[2 ][slv_reg12] <= slv_reg0[23:16];
        sa_bank[3 ][slv_reg12] <= slv_reg0[31:24];
        sa_bank[4 ][slv_reg12] <= slv_reg1[7:0];
        sa_bank[5 ][slv_reg12] <= slv_reg1[15:8];
        sa_bank[6 ][slv_reg12] <= slv_reg1[23:16];
        sa_bank[7 ][slv_reg12] <= slv_reg1[31:24];
        sa_bank[8 ][slv_reg12] <= slv_reg2[7:0];
        sa_bank[9 ][slv_reg12] <= slv_reg2[15:8];
```

Cb_bank 也是相同做法，我想修改的方式是把 S_AXI_WDATA 直接寫到對應的 bank 位置，修改完後的程式碼如下圖：

```
if (slv_reg_wren)
  begin
    case ( axi_awaddr[ADDR_LSB+OPT_MEM_ADDR_BITS:ADDR_LSB] )
      5'h00: begin
        for ( byte_index = 0; byte_index <= (C_S_AXI_DATA_WIDTH/8)-1; byte_index = byte_index+1 )
          if ( S_AXI_WSTRB[byte_index] == 1 ) begin
            slv_reg0[(byte_index*8) +: 8] <= S_AXI_WDATA[(byte_index*8) +: 8];
            if(slv_reg12<=47) begin
                sa_bank[slv_reg12][383-(byte_index*8)-:8] <= S_AXI_WDATA[(byte_index*8+7) -: 8];
            end
            else if(slv_reg12>47&&slv_reg12<64) begin
                cb_bank[slv_reg12-48][127-(byte_index*8)-:8] <= S_AXI_WDATA[(byte_index*8+7) -: 8];
            end
          end
      end
```

修改完後的 resource 果然變小許多，LUT 從原本的四萬多變成 8740，而 FF 也縮減成 7362，下圖為 vivado 合成完後的數據：



而且這樣的存法把 LUT 變成 Distributed RAM，合成的速度也快非常多，以上這種做法是我實驗的各種方式後，能最快執行完計算，以及花費最少的 resource 的做法，下圖為 LUT 的數據分析。

```
1. Slice Logic
--------------

+----------------------------+------+-------+-----------+-------+
|          Site Type         | Used | Fixed | Available | Util% |
+----------------------------+------+-------+-----------+-------+
| Slice LUTs*                | 8740 |     0 |     53200 | 16.43 |
|   LUT as Logic             | 7204 |     0 |     53200 | 13.54 |
|   LUT as Memory            | 1536 |     0 |     17400 |  8.83 |
|     LUT as Distributed RAM | 1536 |     0 |           |       |
```

之後又嘗試了以下一些修改，原本預期要比較快，但結果並沒有。

C. *Two same module*

在把原本的 me_match 硬體加速到能在 0.7 秒完成計算行為後，因為使用的 resource 很少，所以我想說開兩個一樣的硬體 module，一個計算奇數直行的 block image，一個則是算偶數直行，因此在軟體 match function，則是改為其中一個硬體進入 hw_active 後，就告訴另一個硬體要開始傳送 data，並且兩個硬體可以同時運算，下圖為軟體改完後的程式碼：

```
int match(int *x0, int *y0, int *x1, int *y1, int posx, int posy, uint8
    int min_sad;
    /* Copy CB and SA into the HW IP internal buffers */
    block_bcopy(IP_SA, 48, prev, width, posx-16, posy-16);
    block_bcopy(IP_CB, 16, curr, width, posx, posy);
    *hw_active_0 = 1;

    block_bcopy_1(IP_SA, 48, prev, width, posx+1-16, posy-16);
    block_bcopy_1(IP_CB, 16, curr, width, posx+1, posy);
    *hw_active_1 = 1;

    while (*hw_active_0);
    *x0 = *ip_mvx_0;
    *y0 = *ip_mvy_0;

    while (*hw_active_1);
    *x1 = *ip_mvx_1;
    *y1 = *ip_mvy_1;
```

原本以為兩個硬體一起算，應該會變快沒想到執行程式的結果如下：

It took 779 milliseconds to estimate the motion field.

比原本的慢了 0.04 秒，我想可能是因為用 block copy 的方式傳 data 的關係，雖然兩個硬體同時計算，但是因為光是單一一個硬體載入 image 就須花許多時間，這段時間遠比 finite-state machine 計算 sad 的時間長，因此開兩個 module 並無法讓速度變快，反而因為 data transfer 時間，速度變慢了，如果是開兩個 module，並且是利用 burst 的傳送方式，一定能大幅減少整個程式的計算時間。

D. *Median filter function*

修改完硬體後，我也研究了軟體程式碼，我發現其中 median 3x3 function 也是每一次計算一點就會執行一次

的 function，所以我覺得如果改善這裡的速度，應該也會對整體執行時間，加速不少的。這個函式的功能是把一個傳進來的矩陣，先轉成 array，排序完成後，再把矩陣中間那格，替換成 array 的中間值，達到中間值濾波器的功能。原本的程式碼是用 insertion sort，我把它改成 quick sort，希望能加速程式執行時間，而結果是只有加速 0.01 秒，對整體的速度並沒有太大的幫助，我覺著如果這裡把他寫成硬體，讓硬體去做中值濾波器的功能一定會比較快。

IV. EXPERIMENT REVIEW

這次的 final project，是要結合這學期前面五次 lab 所學，加速 lab3 的程式，以上各種方法是我所嘗試的方法，原本也有想到開兩個 cpu 並用硬體設計 mutex，讓兩個 cpu 去搶執行動作，但是我比較喜歡 fully hardware 的設計，所以只用了上述幾種嘗試方法。我學習到硬體 pipeline 的優點，是軟體無法完成的事，一開始覺得看波型圖很困難，但是當碰到 de 不出來的 bug 時，真的覺得看波型很好玩。