

Final Project Report (Team 6)

陳揚哲, 電機系, 111061545
葉承泓, 半導體研究學院碩士班(設計部), 112501538

- ✓ Due date: 2024/6/23 23:59

● Describe the whole implementation flow of our final project

1. Target application: optical flow

我們這組選擇的主題為老師所提供的 final project 題目 list 中找到與我們的研究剛好相關的主題——光流(optical flow)的計算。光流演算法的主要目標是「給予一連串連續的影像(每一幀稱為一個 frame) · 我們想要計算圖上的每一個 pixel 隨時間移動的速度 (含量值及方向)」。我們先介紹光流演算法的數學模型：首先，在計算光流時有一個基本假設：「物體移動前後亮度不變。」這是非常重要的前提，因為我們拿來計算速度的資訊只有不同時間點的光強度，若物體的光強度會隨時間變化的話，就無從得知物體變化的位置了。由這個前提假設，我們可以得到下列「光強度(I)守恆方程式」：

$$I(x_0, y_0, t_0) = I(x_0 + \Delta x, y_0 + \Delta y, t_0 + \Delta t)$$

將等式右邊取 Taylor series expansion with respect to t · 可得

$$I(x_0, y_0, t_0) = I(x_0, y_0, t_0) + \frac{\partial I}{\partial x} \frac{\partial x}{\partial t} + \frac{\partial I}{\partial y} \frac{\partial y}{\partial t} + \frac{\partial I}{\partial t} + o(\Delta t)$$

其中 $o(\Delta t)$ 指的是 Taylor remainder。

其中 $o(\Delta t)$ 指的是 Taylor remainder · 由二階以上的項次組合而成。由於其為 2 階 effect，在底下的計算中我們將其忽略。

因此我們可以整理上式得到

$$\frac{\partial I}{\partial x} \frac{\partial x}{\partial t} + \frac{\partial I}{\partial y} \frac{\partial y}{\partial t} + \frac{\partial I}{\partial t} = 0$$

我們令 $\frac{\partial I}{\partial x} = I_x$ 、 $\frac{\partial I}{\partial y} = I_y$ 、 $\frac{\partial I}{\partial t} = I_t$ 、 $\frac{\partial x}{\partial t} = u$ 、 $\frac{\partial y}{\partial t} = v$ · 其中 I_x 表示光強度在 x 軸方向上的變化程度、 I_y 表示光強度在 y 軸方向上的變化程度、 I_t 表示光強度在 t 軸方向上的變化程度、 u 表示物體在 x 方向上的移動速度、 v 表示物體在 y 方向上的移動速度。

因此我們可將上式寫作

$$\mathbf{I}_x \mathbf{u} + \mathbf{I}_y \mathbf{v} + \mathbf{I}_t = \mathbf{0}$$

此式即為**光流基本方程式**。

給定連續幾幀的照片後，我們可由圖片中萃取出每個 frame 的光強度，因此可計算出每個 pixel 位置的 \mathbf{I}_x 、 \mathbf{I}_y 、 \mathbf{I}_t 的資訊，因此這些參數可視為從圖片中已知。故光流基本方程式中剩下我們想求出的 u 、 v 這兩個參數，但我們只有一條方程式、有兩個未知數，並無法找出一組精準解（會出現無限多組解），因此我們還需要更多的 constraint (方程式)。目前已發展出許多種光流演算法，它們各自對光流做了不同種的前提假設，而得到更多的 constraint，最常見的有底下兩種方法：

(1) Lucas-Kanade (LK) method

他假設「我們所正在計算的這個 pixel 附近的 pixel 皆有相同/類似的速度」，也就是這個 pixel 與鄰近 pixel (形成一個 " patch ") 的光流方程式中的 u 、 v 值各自相同。若我們取 3×3 的 patch，我們可以透過鄰近這 9 個 pixel 的光流基本方程式來列出底下的方程組：

$$\left\{ \begin{array}{l} \mathbf{I}_x(\mathbf{P}_1, \mathbf{t}_0) \mathbf{u} + \mathbf{I}_y(\mathbf{P}_1, \mathbf{t}_0) \mathbf{v} + \mathbf{I}_t(\mathbf{P}_1, \mathbf{t}_0) = \mathbf{0} \\ \mathbf{I}_x(\mathbf{P}_2, \mathbf{t}_0) \mathbf{u} + \mathbf{I}_y(\mathbf{P}_2, \mathbf{t}_0) \mathbf{v} + \mathbf{I}_t(\mathbf{P}_2, \mathbf{t}_0) = \mathbf{0} \\ \mathbf{I}_x(\mathbf{P}_3, \mathbf{t}_0) \mathbf{u} + \mathbf{I}_y(\mathbf{P}_3, \mathbf{t}_0) \mathbf{v} + \mathbf{I}_t(\mathbf{P}_3, \mathbf{t}_0) = \mathbf{0} \\ \mathbf{I}_x(\mathbf{P}_4, \mathbf{t}_0) \mathbf{u} + \mathbf{I}_y(\mathbf{P}_4, \mathbf{t}_0) \mathbf{v} + \mathbf{I}_t(\mathbf{P}_4, \mathbf{t}_0) = \mathbf{0} \\ \mathbf{I}_x(\mathbf{P}_5, \mathbf{t}_0) \mathbf{u} + \mathbf{I}_y(\mathbf{P}_5, \mathbf{t}_0) \mathbf{v} + \mathbf{I}_t(\mathbf{P}_5, \mathbf{t}_0) = \mathbf{0} \\ \mathbf{I}_x(\mathbf{P}_6, \mathbf{t}_0) \mathbf{u} + \mathbf{I}_y(\mathbf{P}_6, \mathbf{t}_0) \mathbf{v} + \mathbf{I}_t(\mathbf{P}_6, \mathbf{t}_0) = \mathbf{0} \\ \mathbf{I}_x(\mathbf{P}_7, \mathbf{t}_0) \mathbf{u} + \mathbf{I}_y(\mathbf{P}_7, \mathbf{t}_0) \mathbf{v} + \mathbf{I}_t(\mathbf{P}_7, \mathbf{t}_0) = \mathbf{0} \\ \mathbf{I}_x(\mathbf{P}_8, \mathbf{t}_0) \mathbf{u} + \mathbf{I}_y(\mathbf{P}_8, \mathbf{t}_0) \mathbf{v} + \mathbf{I}_t(\mathbf{P}_8, \mathbf{t}_0) = \mathbf{0} \\ \mathbf{I}_x(\mathbf{P}_9, \mathbf{t}_0) \mathbf{u} + \mathbf{I}_y(\mathbf{P}_9, \mathbf{t}_0) \mathbf{v} + \mathbf{I}_t(\mathbf{P}_9, \mathbf{t}_0) = \mathbf{0} \end{array} \right.$$

將其進一步整理成矩陣型式如下：

$$\begin{bmatrix} I_x(P_1, t_0) & I_y(P_1, t_0) \\ I_x(P_2, t_0) & I_y(P_2, t_0) \\ I_x(P_3, t_0) & I_y(P_3, t_0) \\ I_x(P_4, t_0) & I_y(P_4, t_0) \\ I_x(P_5, t_0) & I_y(P_5, t_0) \\ I_x(P_6, t_0) & I_y(P_6, t_0) \\ I_x(P_7, t_0) & I_y(P_7, t_0) \\ I_x(P_8, t_0) & I_y(P_8, t_0) \\ I_x(P_9, t_0) & I_y(P_9, t_0) \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix} = - \begin{bmatrix} I_t(P_1, t_0) \\ I_t(P_2, t_0) \\ I_t(P_3, t_0) \\ I_t(P_4, t_0) \\ I_t(P_5, t_0) \\ I_t(P_6, t_0) \\ I_t(P_7, t_0) \\ I_t(P_8, t_0) \\ I_t(P_9, t_0) \end{bmatrix}$$

其中的 9×2 矩陣我們將稱其為 A 矩陣、 $\begin{bmatrix} u \\ v \end{bmatrix}$ 稱為 x 矩陣、 9×1 矩陣稱為 b 矩陣。

由於在上述方程組中有 9 個等式，但未知數只有兩個，因此解為 over-determined，可能找不到唯一解，通常會無解。且因為鄰近 pixel 的速度不會完全相同，因此我們將目標改為要盡量使上述方程組越符合（左式的值越接近 0）越好，也就是使左式與 0 之間的 error 越小越好。這個部份我們可以運用「最小平方法(least square fit)」來找出最佳解 \hat{x} ，即

$$\hat{x} = \underset{x}{\operatorname{argmin}} \|Ax - b\|^2$$

藉由「極值會發生在微分為 0 處」的概念，可以解出 \hat{x} 滿足

$$A^T A \hat{x} = A^T b \Rightarrow \hat{x} = (A^T A)^{-1} A^T b$$

將 A 、 b 矩陣代入上式，可得 $\begin{bmatrix} u \\ v \end{bmatrix}$ 最佳解為

$$\begin{bmatrix} u \\ v \end{bmatrix}$$

$$= - \begin{bmatrix} \sum_{i=1}^9 I_x(P_i, t_0) I_x(P_i, t_0) & \sum_{i=1}^9 I_x(P_i, t_0) I_y(P_i, t_0) \\ \sum_{i=1}^9 I_y(P_i, t_0) I_x(P_i, t_0) & \sum_{i=1}^9 I_y(P_i, t_0) I_y(P_i, t_0) \end{bmatrix}^{-1} \begin{bmatrix} \sum_{i=1}^9 I_x(P_i, t_0) I_t(P_i, t_0) \\ \sum_{i=1}^9 I_y(P_i, t_0) I_t(P_i, t_0) \end{bmatrix}$$

In general · 當取 $m \times n$ 的 patch 時 · 方程組將變為

$$\begin{bmatrix} I_x(P_{(1,1)}, t_0) & I_y(P_{(1,1)}, t_0) \\ I_x(P_{(1,2)}, t_0) & I_y(P_{(1,2)}, t_0) \\ \dots & \dots \\ I_x(P_{(1,n)}, t_0) & I_y(P_{(1,n)}, t_0) \\ I_x(P_{(2,1)}, t_0) & I_y(P_{(2,1)}, t_0) \\ I_x(P_{(2,2)}, t_0) & I_y(P_{(2,2)}, t_0) \\ \dots & \dots \\ I_x(P_{(m,n)}, t_0) & I_y(P_{(m,n)}, t_0) \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix} = - \begin{bmatrix} I_t(P_{(1,1)}, t_0) \\ I_t(P_{(1,2)}, t_0) \\ \dots \\ I_t(P_{(1,n)}, t_0) \\ I_t(P_{(2,1)}, t_0) \\ I_t(P_{(2,2)}, t_0) \\ \dots \\ I_t(P_{(m,n)}, t_0) \end{bmatrix}$$

此時對應到 $\begin{bmatrix} u \\ v \end{bmatrix}$ 最佳解為

$$\begin{bmatrix} u \\ v \end{bmatrix} = - \begin{bmatrix} \sum_{P_i \in P} I_x(P_i, t_0)I_x(P_i, t_0) & \sum_{P_i \in P} I_x(P_i, t_0)I_y(P_i, t_0) \\ \sum_{P_i \in P} I_y(P_i, t_0)I_x(P_i, t_0) & \sum_{P_i \in P} I_y(P_i, t_0)I_y(P_i, t_0) \end{bmatrix}^{-1} \begin{bmatrix} \sum_{P_i \in P} I_x(P_i, t_0)I_t(P_i, t_0) \\ \sum_{P_i \in P} I_y(P_i, t_0)I_t(P_i, t_0) \end{bmatrix}$$

為了方便表示 · 我們將上式矩陣中的各項標記如下：

$$\begin{bmatrix} u \\ v \end{bmatrix} = - \begin{bmatrix} A & B \\ C & D \end{bmatrix}^{-1} \begin{bmatrix} E \\ F \end{bmatrix}$$

A
 B
 E
C
 D
 F

因此上述運算的結果可表示為：

$$\begin{bmatrix} u \\ v \end{bmatrix} = - \begin{bmatrix} A & B \\ C & D \end{bmatrix}^{-1} \begin{bmatrix} E \\ F \end{bmatrix}$$

$$\begin{aligned}
 &= -\frac{1}{AD - BC} \begin{bmatrix} D & -B \\ -C & A \end{bmatrix} \begin{bmatrix} E \\ F \end{bmatrix} = -\frac{1}{AD - BC} \begin{bmatrix} DE - BF \\ AF - CE \end{bmatrix} \\
 &= \begin{bmatrix} DE - BF \\ BC - AD \\ AF - CE \\ BC - AD \end{bmatrix} \\
 &= \begin{bmatrix} \Sigma I_y^2 \Sigma I_x I_t - \Sigma I_x I_y \Sigma I_y I_t \\ 2 \Sigma I_x I_y - \Sigma I_x^2 \Sigma I_y^2 \\ \Sigma I_x^2 \Sigma I_y I_t - \Sigma I_x I_y \Sigma I_x I_t \\ 2 \Sigma I_x I_y - \Sigma I_x^2 \Sigma I_y^2 \end{bmatrix}
 \end{aligned}$$

因此當我們從連續的 frames 中得到 I_x 、 I_y 、 I_t 的資訊後，即可帶入此式中得到 (u, v) 的速度結果。

(2) Horn-Schunck (HS) method

此方法在光流基本方程式的前提下，又再進一步加入 constraint。他透過假設光流/速度 (u, v) 在整個 frame 中是 smooth 變化的，也就是 $\frac{\partial u}{\partial x}$ 、 $\frac{\partial u}{\partial y}$ 、 $\frac{\partial v}{\partial x}$ 、 $\frac{\partial v}{\partial y}$ 不應有太劇烈的變化 over the whole frame。故可以得到目標的 error term 為

$$E = \iint (I_x u + I_y v + I_t)^2 dx dy + \alpha \iint \left\{ \left(\frac{\partial u}{\partial x} \right)^2 + \left(\frac{\partial u}{\partial y} \right)^2 + \left(\frac{\partial v}{\partial x} \right)^2 + \left(\frac{\partial v}{\partial y} \right)^2 \right\} dx dy$$

其中可看到他不只考量到光流基本方程式（要盡量使 $I_x u + I_y v + I_t$ 接近 0），也新加入 (u, v) 為 smooth 變化的這個 constraint，將其一起納入 error term 中考量。

我們的目標是希望找到這個 error term 的最小值，此時的 (u, v) 即為最佳解。由於 (u, v) 位於很多項之中，因此無法找到 explicit 的數學解，通常會透過工程學上常用的方法，去迭代計算

$$\begin{cases} \mathbf{u}_{k+1} = \bar{\mathbf{u}}_k - \frac{I_x[I_x\bar{\mathbf{u}}_k + I_y\bar{\mathbf{v}}_k + I_t]}{\alpha^2 + I_x^2 + I_y^2} \\ \mathbf{v}_{k+1} = \bar{\mathbf{v}}_k - \frac{I_y[I_x\bar{\mathbf{u}}_k + I_y\bar{\mathbf{v}}_k + I_t]}{\alpha^2 + I_x^2 + I_y^2} \end{cases}$$

其中 $\bar{\mathbf{u}}_k$ 表示第 k 次 iteration，針對我們要算的 pixel 的附近的 u 值平均 (the neighborhood average of \mathbf{u}_k)。

不斷迭代之後直到 convergence，即為目標速度(u, v)。

在上述兩種 methods 中，又以 LK methos 最為常見，也有最多 paper 在研究如何優化其執行速度或結果的精準度等，例如最後面的 Reference 2. 所提到的 paper 就是以 LK method 為 base，進化成更精準的版本 " multi-scale pyramid LK method "，並將其以 HLS 來實作，以達到更快的執行速度。

因此，我們選擇以 LK method 來做為實作的目標演算法，想透過 HLS 的實作，達到比 software 執行速度更快的 optical flow accelerator！

2. Software implementation (reference code from [rosetta](#))

我們拿來對照用的 reference algorithm C code 是來自 [rosetta](#) 這個 open source，觀察其程式碼後可知他主要是使用 LK 方法來運算，並再使用 LK 的運算式前先對 input data 預先做多層的 filter，以 smooth 亮度/光強度資訊。我們將其主要內容 (typedefs.h、optical_flow_sw.h、optical_flow_sw.cpp) 依照 lab2-2 的 algorithm C code 的格式改寫至/ASoC-Final_project-optical_flow/optical_flow_catapult/cmodel/inc/的 OpticalFlow_defs_software.h 與 OpticalFlow_Algorithm.h 中。對於裡頭的內容修改，其中之一是為了與 HLS 一致，而將其中的 filter coefficient 從原本的

```
const int GRAD_WEIGHTS[] = {1,-8,0,8,-1};
```

改寫為

```
const float GRAD_WEIGHTS_SW[5] = {0.0833,-0.6667,0,0.6667,-0.0833};
```

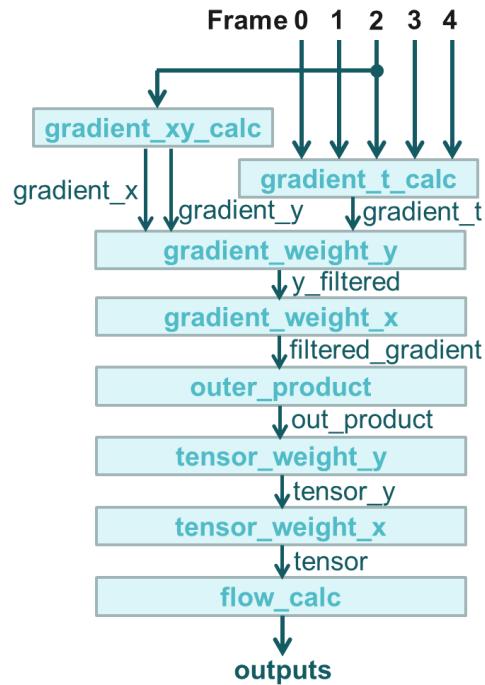
原本的 filter 中的「 $\div 12$ 」的運算是被放在 functions 內部計算，但為了 coefficient 能與 HLS 一致，我們直接將上述的{1,-8,0,8,-1}除以 12 而得到 float type 的 weights。

觀察 rosetta 的 reference algorithm C code 的運算流程，我們可以發現他將整體運算分為多個子運算，並分別放在各個 function 中完成，如下圖中的 " // compute " 部分：

```
// top-level sw function
void optical_flow_sw(pixel_t frame0[MAX_HEIGHT][MAX_WIDTH],
                      pixel_t frame1[MAX_HEIGHT][MAX_WIDTH],
                      pixel_t frame2[MAX_HEIGHT][MAX_WIDTH],
                      pixel_t frame3[MAX_HEIGHT][MAX_WIDTH],
                      pixel_t frame4[MAX_HEIGHT][MAX_WIDTH],
                      velocity_t outputs[MAX_HEIGHT][MAX_WIDTH])
{
    // intermediate arrays
    static pixel_t gradient_x[MAX_HEIGHT][MAX_WIDTH];
    static pixel_t gradient_y[MAX_HEIGHT][MAX_WIDTH];
    static pixel_t gradient_z[MAX_HEIGHT][MAX_WIDTH];
    static gradient_t y_filtered[MAX_HEIGHT][MAX_WIDTH];
    static gradient_t filtered_gradient[MAX_HEIGHT][MAX_WIDTH];
    static outer_t out_product[MAX_HEIGHT][MAX_WIDTH];
    static tensor_t tensor_y[MAX_HEIGHT][MAX_WIDTH];
    static tensor_t tensor[MAX_HEIGHT][MAX_WIDTH];

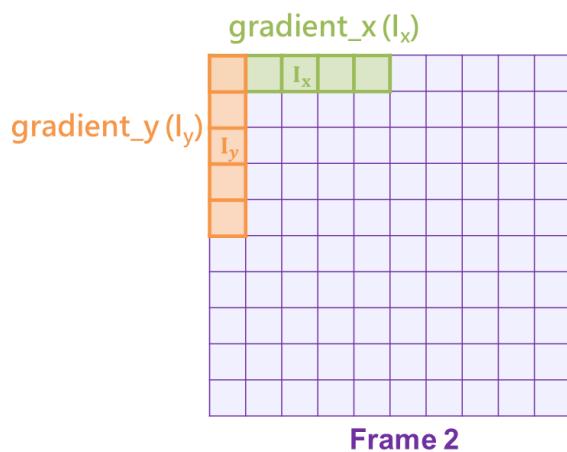
    // compute
    gradient_xy_calc(frame2, gradient_x, gradient_y);
    gradient_z_calc(frame0, frame1, frame2, frame3, frame4, gradient_z);
    gradient_weight_y(gradient_x, gradient_y, gradient_z, y_filtered);
    gradient_weight_x(y_filtered, filtered_gradient);
    outer_product(filtered_gradient, out_product);
    tensor_weight_y(out_product, tensor_y);
    tensor_weight_x(tensor_y, tensor);
    flow_calc(tensor, outputs);
}
```

其運算的 data flow chart 及各 block 如下：

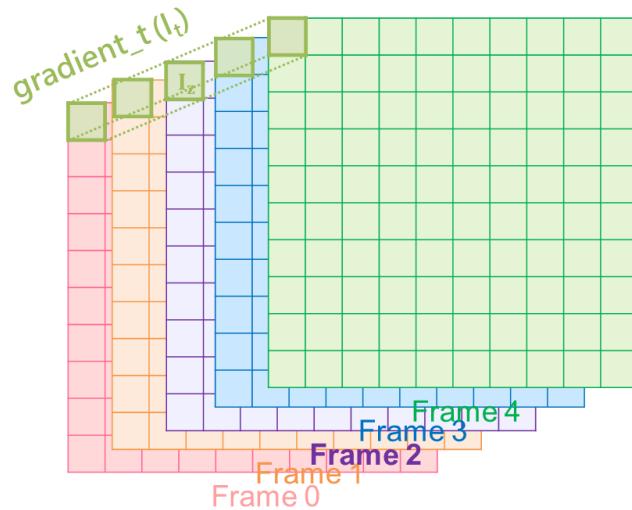


以下簡述其 top function 的計算流程：

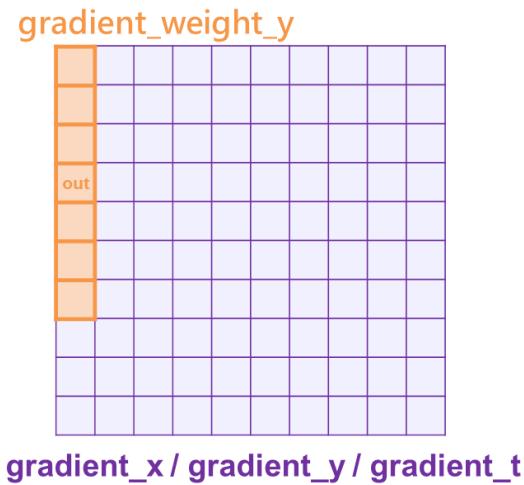
- i. `gradient_xy_calc()`：輸入連續 5 個 frame 中位於正中間的 frame (frame 2)，計算 I_x 及 I_y 的值並輸出。計算過程有如 convolution，只是為 1D kernel。其 kernel value 為 `GRAD_WEIGHTS_SW[5] = {0.0833, -0.6667, 0, 0.6667, -0.0833}`。



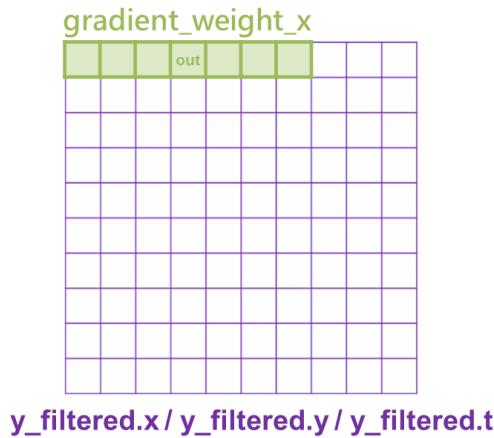
- ii. `gradient_z_calc()`：輸入連續 5 個 frame，計算 I_t 的值並輸出（在 rosetta 的程式碼中的 z 指的皆是我們在第 1 點中介紹的演算法中的 t，因此此處的 output I_z 即為 I_t ）。計算過程跟上述的 I_x 、 I_y 相同，為 t 軸(時間軸)方向的 1D kernel。



- iii. `gradient_weight_y()`：將前面計算所得到的 I_x 、 I_y 、 I_t 透過 filter 做 y 方向上的 smooth，得到 y_{filtered} （有 x , y , t 三個 components）。計算過程同樣有如 convolution，只是為 1D kernel。其 kernel value 為 $\text{GRAD_FILTER_SW}[] = \{0.0755, 0.133, 0.1869, 0.2903, 0.1869, 0.133, 0.0755\}$ 。

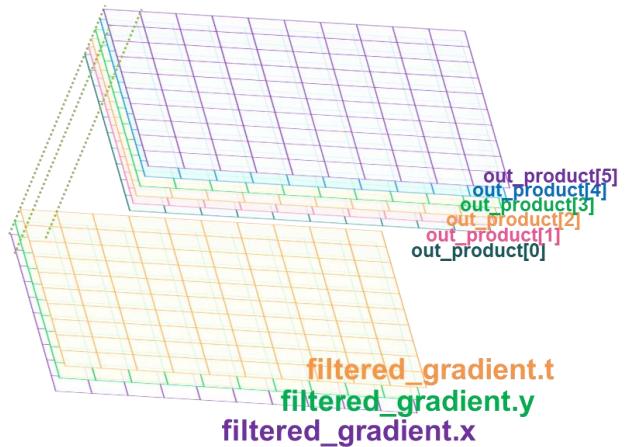


- iv. `gradient_weight_x()`：將 y_{filtered} 的 3 個 components 透過相同的 filter coefficient 做 x 方向上的 smooth，得到 filtered_gradient （亦有 x , y , t 三個 components）。計算過程跟上述的 `gradient_weight_y` 相同，只是為 x 軸方向的 1D kernel。

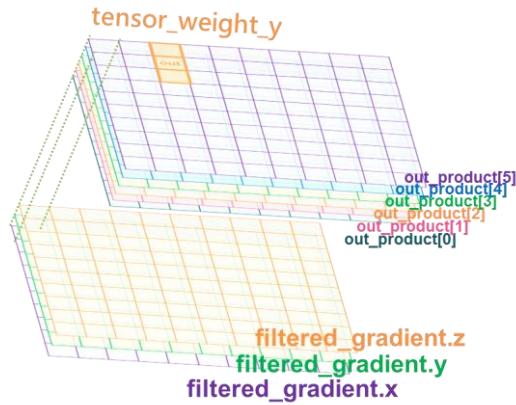


- v. `outer_product()`：由輸入的 I_x 、 I_y 、 I_t (已經過 y 及 x 方向的 smooth) 的資訊求出 I_x^2 、 $I_x I_y$ 、 I_y^2 、 $I_x I_t$ 、 $I_y I_t$ 等值，會在後續的 LK 的矩陣運算處所用。此 function 的輸出為有 6 個 element 的 array (for each pixel)，分別對應如下的計算：

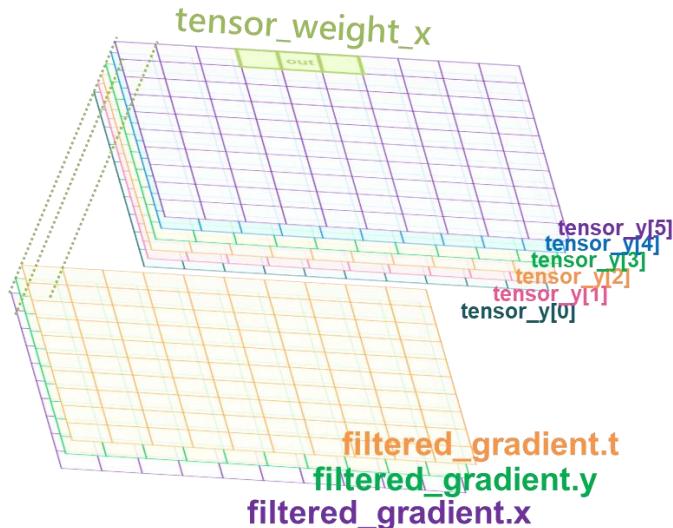
$$\left\{ \begin{array}{l} \text{out_product[0]} = I_x I_x \\ \text{out_product[1]} = I_y I_y \\ \text{out_product[2]} = I_t I_t \\ \text{out_product[3]} = I_x I_y \\ \text{out_product[4]} = I_x I_t \\ \text{out_product[5]} = I_y I_t \end{array} \right.$$



- vi. `tensor_weight_y()`：將 I_x^2 、 $I_x I_y$ 、 I_y^2 、 $I_x I_t$ 、 $I_y I_t$ 透過 filter 做 y 方向上的 smooth，得到 `tensor_y`。計算過程同樣有如 convolution，只是為 1D kernel。其 kernel value 為 `TENSOR_FILTER_SW[3] = {0.3243, 0.3513, 0.3243}`。



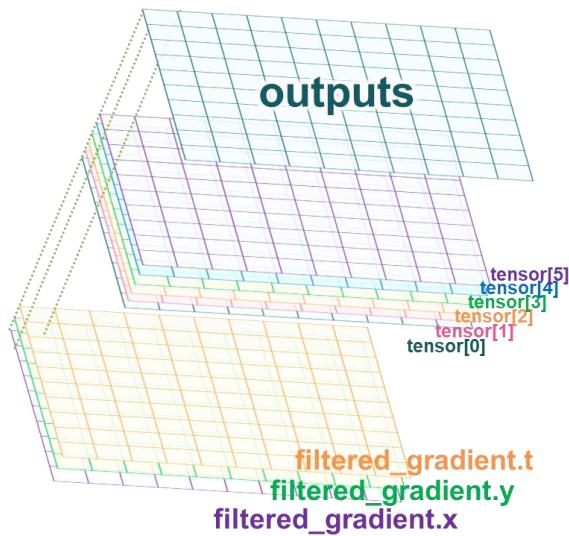
- vii. `tensor_weight_x()` : 將 `tensor_y` 的 6 個 components 透過相同的 tensor filter coefficient 做 x 方向上的 smooth , 得到 tensor (亦有 x, y, t6 個 components) 。計算過程跟上述的 `tensor_weight_y` 相同 , 只是為 x 軸方向的 1D kernel 。



- viii. `flow_calc()` : 將 smooth 後的值帶入 LK 方法的最終結果矩陣運算

$$\begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} \Sigma I_y^2 \Sigma I_x I_t - \Sigma I_x I_y \Sigma I_y I_t \\ 2 \Sigma I_x I_y - \Sigma I_x^2 \Sigma I_y^2 \\ \Sigma I_x^2 \Sigma I_y I_t - \Sigma I_x I_y \Sigma I_x I_t \\ 2 \Sigma I_x I_y - \Sigma I_x^2 \Sigma I_y^2 \end{bmatrix}, \text{ 其中各個顏色所對應到的值即可對應到}$$

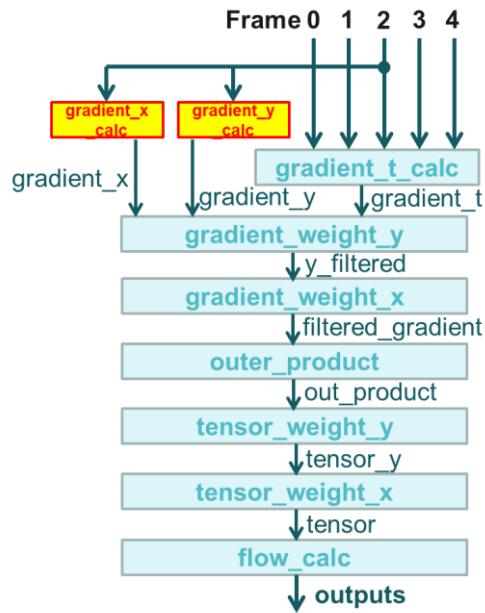
I_x^2 、 $I_x I_y$ 、 I_y^2 、 $I_x I_t$ 、 $I_y I_t$ 這幾個 input 值 , 即可得出最終速度(u,v)並輸出。每個 pixel 皆會輸出一組(u,v)值。



3. HLS implementation

我們以上述 algorithm C 的運算流程為基底，實作出相對應的 HLS 的 module/function，並進一步透過 hardware 可平行運算的優勢嘗試在 FPGA 板有限的資源下優化其運算速度，在第 4.點中也會使用 testbench 驗證其輸出與 algorithm C 的結果相符合。

首先，在 software 中 lx 及 ly 是在同一個 function 中同時計算出來，這是因為在 software 中是透過 2D array 的 address pointer 可直接讀取到 target pixel 的上下左右的 pixel 的值，然而在 HLS 中是使用 stream interface，其輸入通常是由左至右，由上至下去 stream in input value，並無法同時 access 到上下位置的兩個 pixel 值，因此 ly 的計算需仰賴 line buffer 存住前幾個 row 的 input 值，而 lx 的計算只需要 pixel buffer 存住前幾個位置的 input 值即可。這兩個 module 的實作方式也因此不同，故我們將其分作 2 個 HLS module 來實作。此時的 HLS design 設計如下：



接著，由於 FSIC 的設計限制 user project 只能有 32-bit 的 input stream data width 及 32-bit 的 output stream data width。我們發現雖然在 FSIC 的 top module (fsic.v)中可以修改 pDATA_WIDTH 為別的值：

```

module FSIC #(
    parameter BITS=32,
    parameter pUSER_PROJECT_SIDEBOARD_WIDTH = 5,
    `ifdef USER_PROJECT_SIDEBOARD_SUPPORT
        parameter pSERIALIO_WIDTH = 13,
    `else
        parameter pSERIALIO_WIDTH = 12,
    `endif
    parameter pADDR_WIDTH = 15,
    parameter pDATA_WIDTH = 32,
    parameter pRxFIFO_DEPTH = 5,
    parameter pCLK_RATIO = 4
) (

```

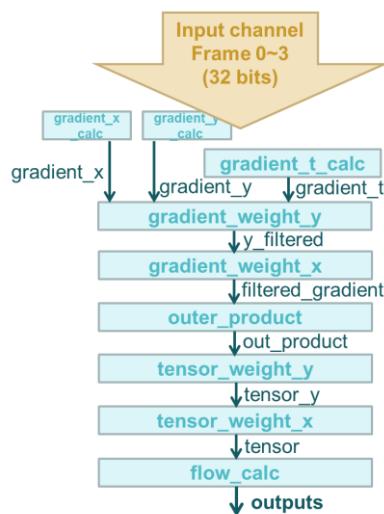
但在底下呼叫更底層的 module 時卻沒有將這個 parameter 傳遞下去，而是直接設定為預設值 32 bit：

```

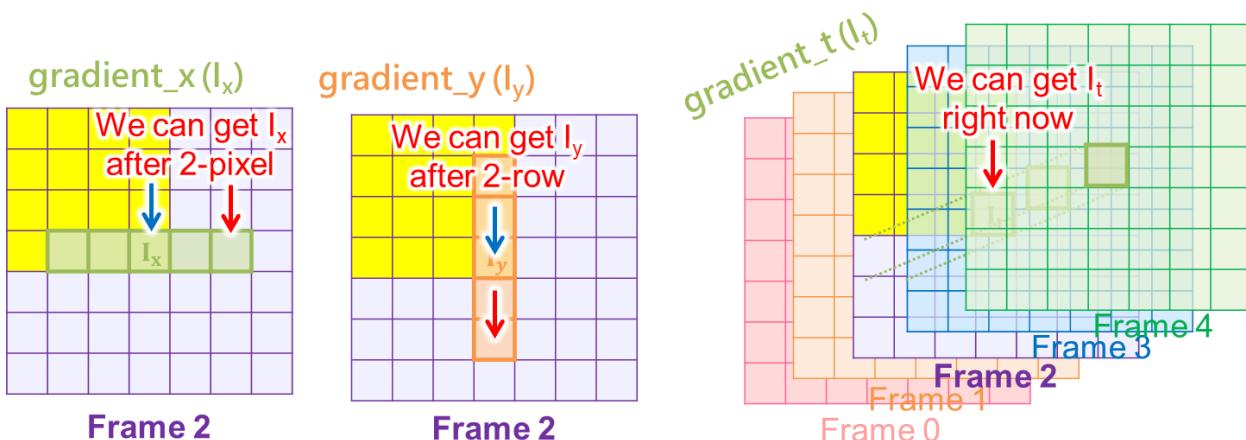
CFG_CTRL #( .pADDR_WIDTH( pADDR_WIDTH ),
            .pDATA_WIDTH( 32 ) ) U_CFG_CTRL0 (
    .aa_cfg_awvalid (m_awvalid_aa_cfg_awvalid), // I
    .aa_cfg_awaddr (m_awaddr_aa_cfg_awaddr), // I 32
    .aa_cfg_wvalid (m_wvalid_aa_cfg_wvalid), // I
    .aa_cfg_wdata (m_wdata_aa_cfg_wdata), // I 32
    .aa_cfg_wstrb (m_wstrb_aa_cfg_wstrb), // I 4
    .aa_cfg_arvalid (m_arvalid_aa_cfg_arvalid), // I
    .aa_cfg_araddr (m_araddr_aa_cfg_araddr), // I 32
    .aa_cfg_rready (m_rready_aa_cfg_rready), // I
    .axi_wready1 (s_wready_axi_wready1), // I
    .axi_awready1 (s_awready_axi_awready1). // I
)

```

因此若需要使用這個功能必須大幅修改 FSIC 的 design。於是我們決定要修改我們的 input stream width，原本因為每個 pixel 使用 8 bit 來儲存其亮度/光強度的資訊 (gray scale value)，因此若想 stream in 連續 5 個 frame 的 pixel 的值，共需要 40 bit。又因為這 5 個 frame 主要會影響到的是 I_t 的運算，我們觀察到 I_t 的運算 kernel 為 GRAD_WEIGHTS_SW[5] = {0.0833, -0.6667, 0, 0.6667, -0.0833}，中間有一個位置是乘以 0，因此似乎可以將其縮減成長度為 4 的 kernel (而 I_y 、 I_x 只會使用到中間的 frame 的數據，因此可以留用長度為 5 的 kernel)。故我們將 input stream 改成每次輸入連續 "4" 個 frame 的 pixel (原本是連續 5 個 frame 的 pixel)，如此一來即可維持每次 32-bit 的輸入，此時的 HLS design 設計如下：



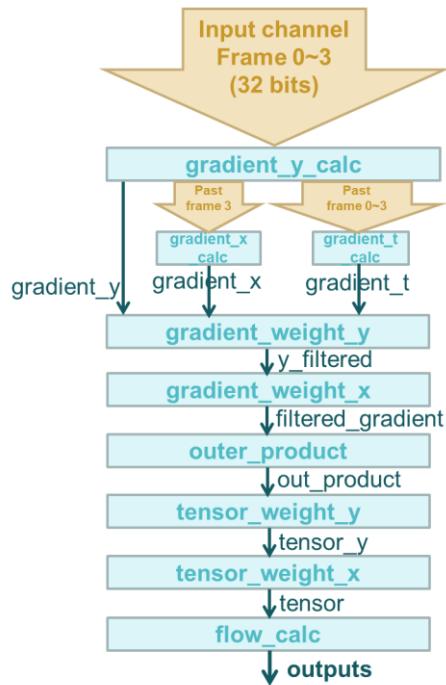
接下來由於 input stream 只能有一個 receiver，必須由 gradient_x_calc 、 gradient_y_calc 、 gradient_t_calc 這三個 module 中選出其中一個，因此我們先來分析他們各自的實作方式，：



假設我們想在這個時間點同時（必須要同時，因為下一個 module 需要連續接收這三個值才能做後續運算）計算出黃色區域右下角位置的 I_x 、 I_y 、 I_t 值，則不同 module 所需要的 pixel 位置各有不同（因為 kernel 方向不同）：

- (1)首先先看 I_y 的運算（上圖中間）：當我們想要圖中藍色箭頭的 I_y 時，表示 input stream 必須已經輸入至紅色箭頭的位置，才能有值乘上 kernel 來得到目標 pixel 的 I_y 。
- (2)接著看 I_x 的運算（上圖左側）：當我們想要圖中藍色箭頭的 I_y 時，表示 input stream 必須已經輸入至紅色箭頭的位置，才能有值乘上 kernel 來得到目標 pixel 的 I_y 。若我們先讓 I_x 接收 input stream，則當 input stream 到上左圖中紅色箭頭位置時，就已算出 I_x ，但必須等到上中圖中紅色箭頭位置時才能算出 I_y ，也就是說，計算出的 I_x 必須經過「 $2*row-2$ 」（其中 row 表示一個 row 的 pixel 數目）這麼大 depth 的 FIFO 才可等到 I_y 輸出再一起輸入至後續 module，而以我們的 test pattern image 而言， $row=1024$ ，因此我們需要 2046 depth 的 FIFO，這在 Catapult 的 tool 中是不支援的（最高只支援到 128 深度的 FIFO）。
- (3)最後來看 I_t 的運算（上圖右側）：當我們想要圖中藍色箭頭的 I_y 時，input stream 只需輸入到同一個 pixel 的位置即可算出，這是因為我們 input stream 會一次 input 4 個 frame 的同一個 pixel 的值，因此 I_t 可以立馬算出。因此若我們先讓 I_t 接收 input stream，則當 input stream 到上右圖中紅色箭頭位置時，就已算出 I_t ，但必須等到上中圖中紅色箭頭位置時才能算出 I_y ，也就是說，計算出的 I_t 必須經過「 $2*row$ 」（其中 row 表示一個 row 的 pixel 數目）這麼大 depth 的 FIFO 才可等到 I_y 輸出再一起輸入至後續 module，也就是我們需要 20468depth 的 FIFO for I_t ，這在 Catapult 的 tool 中是不支援的（最高只支援到 128 深度的 FIFO）。

因此，我們必須先讓 I_y 的 module (也就是 gradient_y_calc) 來接收 input stream，這樣一來可以避免上述 FIFO 過大的問題，不過缺點是它需要存著 I_t 所有的所有 frame 的 input 數據。此時的 HLS design 設計如下：



我們將以這個架構實作我們的 HLS module。

以下為我們 HLS 實作的成果（位於 /ASoC-Final_project-optical_flow/optical_flow_catapult/hls_c/inc 資料夾中）：

(1) OpticalFlow_defs.h :

在此 file 中，我們定義在進行內部運算所會使用的 datatype，首先每次輸入的 input frames 為同一位置 4 個時間點的 frames 值所 packed 起來 data，一個 pixel 為 8 bits，因此剛好能符合 input stream 的 32-bit data 限制。雖然初始的 frames 為 8 bits 的整數，但在經過 kernel 運算後會產生浮點數的 signed output，因此我們仍設定 input_t 的 datatype 為 ac_fixed，以避免後續 ac_int 與 ac_fix 相乘時不相容的可能隱憂。

在 data flow 往 output 方向傳遞的過程中，會有多次與 kernel 相乘的 filter operation，而這些 filter coefficient 皆介於(-1,1)的區間中（整數部分皆為 0），為了計算的精細度我們會將這些 filter 令作 ac_fixed<32>，並將 32-bit 幾乎全部都 allocate 細小數部分，整數部分只留下 sign bit 以及個位數，以達到更精準的乘法結果，也因此在不斷相乘的過程中，我們會需要越來越多 bit 來儲存結果，因此 ac_channel 會越來越寬（bit 數越來越多），可由下圖所見得。直到最後兩個 module (tensor_weight_x 、

flow_calc) , 我們為了滿足 output stream 同樣也需要為 32-bit 的需求，而會使用 bit-shift 的技巧，在 order to 在 bit 數限制和精準度之間作權衡，而使最終 output stream 維持 32 bit。

```
const int MAX_HEIGHT = 436;
const int MAX_WIDTH = 1024;
const int INPUT_T_BIT_WIDTH = 17;
const int INPUT_T_INTEGER_PART = 9;
const int PIXEL_T_BIT_WIDTH = 32;
const int PIXEL_T_INTEGER_PART = 13;

// basic typedefs
typedef ac_fixed<INPUT_T_BIT_WIDTH, INPUT_T_INTEGER_PART, false, AC_TRN, AC_WRAP> input_t; // Integer part: 9 ; Decimal part: 8 ; signed
//typedef ac_fixed<34,18, true, AC_TRN, AC_WRAP> input2x_t; // For ping-pong buffer
typedef ac_int<INPUT_T_BIT_WIDTH> input1x_t; // For ping-pong buffer
typedef ac_int<INPUT_T_BIT_WIDTH*2> input2x_t; // For ping-pong buffer
typedef ac_fixed<PIXEL_T_BIT_WIDTH,PIXEL_T_INTEGER_PART, true, AC_TRN, AC_WRAP> pixel_t; // Integer part: 13 ; Decimal part: 19 ; signed
//typedef ac_fixed<64,26, true, AC_TRN, AC_WRAP> pixel2x_t; // For ping-pong buffer
typedef ac_int<PIXEL_T_BIT_WIDTH> pixel1x_t; // For ping-pong buffer
typedef ac_int<PIXEL_T_BIT_WIDTH*2> pixel2x_t; // For ping-pong buffer

typedef ac_fixed<32,27, true, AC_TRN, AC_WRAP> outer_pixel_t; // Integer part: 27 ; Decimal part: 5 ; signed
typedef ac_fixed<64,56, true, AC_TRN, AC_WRAP> calc_pixel_t; // Integer part: 56 ; Decimal part: 8 ; signed
typedef ac_fixed<32,13, true, AC_TRN, AC_WRAP> vel_pixel_t; // Integer part: 13 ; Decimal part: 19 ; signed
```

另外，在 kernel 計算中會使用到 line buffer，為了 ping-pong buffer 的設計，我們將 line buffer 的 datatype 設定為 ac_int，其 bit width 設定為 frames length 的兩倍，ping-pong buffer 運作中每次更新的 data 的 datatype 也設定為 ac_int，其 bit width 設定為 frames length。在上述計算後所產生的 gradient 值，其 data type 設定為 ac_fixed，data length 設定為 32 bits，使 output 的輸出為 2 的次方。而後續的計算會使用 gradient 值進行 kernel 的運算，也會使用到對應的 ping-pong buffer，最終所產生的 velocity 值，也設定為 32 bits。

(2) OpticalFlow.h :

在此 file 中，我們定義 top function 以及在計算過程中會使用到的 ac_channel 和 channel length，input frames 為同一位置 4 個時間點的 frames 值 packed 而成的 data，設定 ac_channel 的 bit width 為 32 bits，而每個 channel 之間 data 的傳遞都設定 bit width 為 2 的次方，以符合 design rule。

```

// FIFOs connecting the stages
ac_channel<pixel_t> gradient_x; // <-----
ac_channel<pixel_t> gradient_y;
ac_channel<pixel_t> gradient_z;
ac_channel<gradient_t> y_filtered;
ac_channel<gradient_t> filtered_gradient;
ac_channel<outer_t> out_product;
ac_channel<tensor_t> tensor_y;
ac_channel<tensor_int_t> tensor_shift;
ac_channel<shift_t> shift;

// FIFOs for streaming in, just for clarity
//////ac_channel<input_t> frame1_a;
//////ac_channel<input_t> frame2_a;
//////ac_channel<input_t> frame3_a;
//////ac_channel<input_t> frame4_a;
//////ac_channel<input_t> frame5_a;

//Need to duplicate frame3 for the two calculations
//////ac_channel<input_t> frame3_b;
//////ac_channel<input_t> frame3_c;
ac_channel<input_t> frame3;

// Stored input frames for Ix and It
ac_channel<frames_t> input_frames_delayed;

```

```

public:
OpticalFlow_Top() {}

-----
// Function: run
// Top interface for data in/out of class. Combines vertical and
// horizontal derivative and magnitude/angle computation.
#pragma hls_design interface
void CCS_BLOCK(run)(ac_channel<frames_t> &input_frames,
                    maxWType           widthIn,
                    maxHType           heightIn,
                    shift_t             shift_threshold,
                    //ac_channel<pixel_t> &gradient_x, // <-----
                    //ac_channel<gradient_t> &filtered_gradient,
                    //ac_channel<outer_t> &out_product,
                    //ac_channel<tensor_t> &tensor_y,
                    //ac_channel<tensor_t> &tensor,
                    //ac_channel<pixel_t> &denominator,
                    //ac_channel<vel_pixel_t> &denominator,
                    //ac_channel<shift_t> &shift,
                    //ac_channel<velocity_t> &outputs)
                    ac_channel<output_stream_t> &outputs)
{
    // compute
    ////framesplit_inst.run(input_frames, frame1_a, frame2_a, frame3_a, frame3_b, frame3_c, frame4_a, frame5_a, widthIn, heightIn);
    ////gradient_x_calc_inst.run(frame3_b, gradient_x, widthIn, heightIn);
    ////gradient_y_calc_inst.run(frame3_c, gradient_y, widthIn, heightIn);
    ////gradient_z_calc_inst.run(frame1_a, frame2_a, frame3_a, frame4_a, frame5_a, gradient_z, widthIn, heightIn);
    gradient_y_calc_inst.run(input_frames, gradient_y, frame3, input_frames_delayed, widthIn, heightIn);
    gradient_x_calc_inst.run(frame3, gradient_x, widthIn, heightIn);
    gradient_z_calc_inst.run(input_frames_delayed, gradient_z, widthIn, heightIn);
    gradient_weight_y_inst.run(gradient_x, gradient_y, gradient_z, y_filtered, widthIn, heightIn);
    gradient_weight_x_inst.run(y_filtered, filtered_gradient, widthIn, heightIn);
    outer_product_inst.run(filtered_gradient, out_product, widthIn, heightIn);
    tensor_weight_y_inst.run(out_product, tensor_y, widthIn, heightIn);
    ////tensor_weight_x_inst.run(tensor_y, tensor_shift, widthIn, heightIn);
    tensor_weight_x_inst.run(tensor_y, tensor_shift, shift, widthIn, heightIn);
    ////flow_calc_inst.run(tensor, outputs, widthIn, heightIn);
    ////flow_calc_inst.run(tensor_shift, outputs, denominator, widthIn, heightIn);
    ////flow_calc_inst.run(tensor, outputs, denominator, shift, widthIn, heightIn);
    flow_calc_inst.run(tensor_shift, shift, outputs, widthIn, heightIn, shift_threshold));
}

```

以下簡述 Top design 的計算流程：

i. Gradient_y :

首先會進行 Gradient_y 平行計算，使用 input frames 的 middle point 值 (frames3 的 data) · 計算後產生的 gradient_y 的數值。另外也會儲存所輸出的 ly 值所在的那個 pixel 的 4 個 frame 的 data · 以同時輸出至 input_frames_delayed channel 和 frame3 channel 分別供 Gradient_z 、Gradient_x 作為 input stream 。

ii. Gradient_x :

與 gradient_z 平行計算，也會使用 input frames 的 middle point 值 (frames3 的 data) · 計算後產生 gradient_x 的數值。

iii. Gradient_z :

也就是 Gradient_t · 使用 4 個時間點的 frames 同時進行計算，產生 gradient_z 的數值。

iv. Gradient_weight_y :

從此步驟開始的運算為進行 Flow_calc 的前處理，將先前計算產生的 gradient 使用 kernel 的運算，將 gradient 的值 smooth 後再給 Flow_calc 計算。此步驟將 gradient_x 、gradient_y 、gradient_z 計算後產生 y_filtered 。

v. Gradient_weight_x :

此步驟會使用 y_filtered 產生 filtered_gradient 。

vi. Outer_product :

此步驟會使用 filtered_gradient 產生 out_product 。

vii. Tensor_weight_y :

自此步驟開始為第二次前處理，將先前計算產生的 out_product 進行 kernel 的運算(與上述計算類似) · 將 out_product 的值 smooth 後再給 Flow_calc 計算。此步驟會使用 filtered_gradient 產生 tensor_y 。

viii. Tensor_weight_x :

此步驟會使用 tensor_y 產生 tensor。在此會執行第一次的 bit-shift operation 以增加精準度，並減少 interface 的 bit 數目。

ix. Flow_calc :

此步驟會使用 tensor 產生最後的 outputs (velocity)。在此會執行第二次的 bit-shift operation 以增加精準度，並限制 output stream 為 32-bit 的 width。

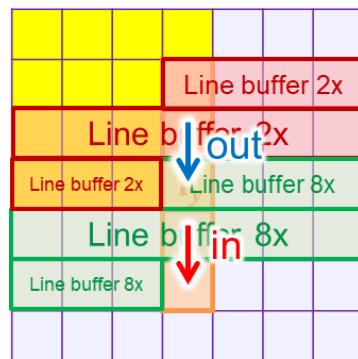
(3) OpticalFlow_gradient_y_calc.h :

此步驟是將 32 bits 的 frames 讀取進入，分離出並計算出 frame3 的各 pixel 的 ly 值，並將 32 bits data 暫存起來以供後續 It 的 module 使用。

由於 data 的讀取是沿 x 軸方向進行更新，因此在計算 y gradient 時需要使用 line buffer，將 4 個 row 的數據儲存起來，在讀取到第 5 列 row 數據時才能計算出 gradient_y 的數值。因為使用的是比較省 resource 的 " 1-port " SRAM，不支援同時作 Read/Write，故我們是使用 ping-pong buffer 的形式來存/取值，因此 WIDTH 只需要圖片寬度的一半，但會需要 2 倍 input data width 的寬度，而且為了使用 bit select 的 function，我們必須將 data type 設定為 ac_int(使用 ac_fixed 無法使用此 function)。

```
typedef ac_int<INPUT_T_BIT_WIDTH> input1x_t; // For ping-pong buffer
typedef ac_int<INPUT_T_BIT_WIDTH*2> input2x_t; // For ping-pong buffer
typedef ac_int<INPUT_T_BIT_WIDTH*4> input4x_t; // For ping-pong buffer
typedef ac_int<INPUT_T_BIT_WIDTH*8> input8x_t; // For ping-pong buffer
```

再者，在輸出 gradient_y 的數值時，會同時輸出「此 ly 所對應到的位置的 4 個 frame 的數據」給後續的 It module，以及「此 ly 所對應到的位置的 frame 3 的數據」給與 It module 平行運算的 Ix module。由於在第 n 個 row 時會輸出第(n-2)個 row，同一個 column 位置的 ly 值，並同時輸出 4 個 frame 給 It，之後就只需要存住 ly 所需要用到的 frame3 數據了，因此只需要 2 組 8 倍 (2×4 倍) input data width 的寬度的 line buffer，後面 2 組只需存住 frame3 的值即可，因此這 2 組 line buffer 只需使用 2 倍 input data width 的寬度即可，如下圖所示：

gradient_y (I_y)**Frame 2**

在 HLS code 中，我們是透過下列方式宣告並使用 line buffer：

```
// Line buffers store pixel line history - Mapped to RAM
input2x_t line_buf3[MAX_WIDTH/2];
input2x_t line_buf2[MAX_WIDTH/2];
input8x_t line_buf1[MAX_WIDTH/2];
input8x_t line_buf0[MAX_WIDTH/2];

input2x_t rdbuf2_pix, rdbuf3_pix;
input2x_t rdbuf1_pix_slice;
input8x_t rdbuf0_pix, rdbuf1_pix;
input8x_t wrbuf0_pix; //, wrbuf1_pix, wrbuf2_pix, wrbuf3_pix;

input_t pix0, pix1, pix2, pix3, pix4;
input1x_t pix0_buf, pix1_buf, pix2_buf, pix3_buf, pix4_buf;
```

由於 ac_fixed 的 data 在轉換至 ac_int 的過程中，為了避免 bit 在轉換的過程中遺失 (Catapult 中不支援直接將同樣深度的 ac_fixed 的數據 bitwise 對應給 ac_int，以及 vice versa，故我們只能透過 for loop 的方式 bit by bit 指定)，因此使用 for loop 將儲存的每一 bit 對應至 pix0 中。

```
// transform back into ac_fixed type
for(uint i=0; i<INPUT_T_BIT_WIDTH; i++){
    pix4[i] = pix4_buf[i];
    pix3[i] = pix3_buf[i];
    pix2[i] = pix2_buf[i];
    pix1[i] = pix1_buf[i];
}
```

將 data 儲存至 pix0 後，會將每兩筆 frames 的值儲存 line buffer 的每一個 element 當中。

```

// Write data cache, write lower 32 on even iterations of COL loop, upper 32 on odd (when "INPUT_T_BIT_WIDTH=8")
if ( (x&1) == 0 ) {
    //////////////wrbuf0_pix.set_slc(0,input_frames_40bits);
    wrbuf0_pix.set_slc(0,input_frames_value);
} else {
    //////////////wrbuf0_pix.set_slc(INPUT_T_BIT_WIDTH*5,input_frames_40bits);
    wrbuf0_pix.set_slc(INPUT_T_BIT_WIDTH*4,input_frames_value);
}
// Read line buffers into read buffer caches on even iterations of COL loop
if ( (x&1) == 0 ) {
    // vertical window of pixels
    rdbuf3_pix = line_buf3[x/2];
    rdbuf2_pix = line_buf2[x/2];
    rdbuf1_pix = line_buf1[x/2];
    rdbuf0_pix = line_buf0[x/2];
    //if (x==50){
    // cout << rdbuf0_pix << " out, ";
    //}
} else { // Write line buffer caches on odd iterations of COL loop
    line_buf3[x/2] = rdbuf2_pix; // copy previous line
    rdbuf1_pix_slice.set_slc(0,rdbuf1_pix.slc<INPUT_T_BIT_WIDTH>(16));
    rdbuf1_pix_slice.set_slc(INPUT_T_BIT_WIDTH,rdbuf1_pix.slc<INPUT_T_BIT_WIDTH>(INPUT_T_BIT_WIDTH*4+16));
    line_buf2[x/2] = rdbuf1_pix_slice; // copy previous line
    line_buf1[x/2] = rdbuf0_pix; // copy previous line
    line_buf0[x/2] = wrbuf0_pix; // store current line
    //if (x==51){
    // cout << wrbuf0_pix << ", ";
    //}
}

```

雖然此時 data 可能還未準備完畢，我們先將計算的過程寫出，由於 pix 會進行小數的乘法運算，因此仍須將 pix_buf(ac_int)轉換回 ac_fixed。若此時其中有些 pix 的 data 尚未 assign 數值，此時 gradient_y 的數值尚未準備好輸出，因此計算上不會產生問題，直到每一個 pix 的 data 都準備完畢後，data 即可輸出。

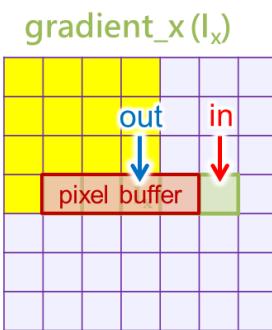
```

// Get 32-bit data from read buffer caches, lower 32 on even iterations of COL loop (when "INPUT_T_BIT_WIDTH=8")
pix4_buf = ((x&1)==0) ? rdbuf3_pix.slc<INPUT_T_BIT_WIDTH>(0) : rdbuf3_pix.slc<INPUT_T_BIT_WIDTH>(INPUT_T_BIT_WIDTH);
pix3_buf = ((x&1)==0) ? rdbuf2_pix.slc<INPUT_T_BIT_WIDTH>(0) : rdbuf2_pix.slc<INPUT_T_BIT_WIDTH>(INPUT_T_BIT_WIDTH);
pix2_buf = ((x&1)==0) ? rdbuf1_pix.slc<INPUT_T_BIT_WIDTH>(16) : rdbuf1_pix.slc<INPUT_T_BIT_WIDTH>(INPUT_T_BIT_WIDTH*4+16);
pix1_buf = ((x&1)==0) ? rdbuf0_pix.slc<INPUT_T_BIT_WIDTH>(16) : rdbuf0_pix.slc<INPUT_T_BIT_WIDTH>(INPUT_T_BIT_WIDTH*4+16);

input_frames_delayed_value = ((x&1)==0) ? rdbuf1_pix.slc<INPUT_T_BIT_WIDTH*4>(0) : rdbuf1_pix.slc<INPUT_T_BIT_WIDTH*4>(INPUT_T_BIT_WIDTH*4);

```

(4) OpticalFlow_gradient_x_calc.h :



此步驟也是將 8 bits 的 frame 讀取進入此 module 當中，由於 data 的讀取是沿 x 軸方向進行更新，因此計算不需使用 line buffer，只需要使用 shift register 即可。每一 cycle 更新時，只需要讀取新的 input，並更新每一個 buffer 的值，即可計算出 gradient_x。

```

// read input value
if (x <= widthIn-1) {
    pix0 = frame3_b.read(); // Read from input stream
} else {
    pix0 = pix_buf0;
}

pix1 = pix_buf0;
pix2 = pix_buf1;
pix3 = pix_buf2;
pix4 = pix_buf3;

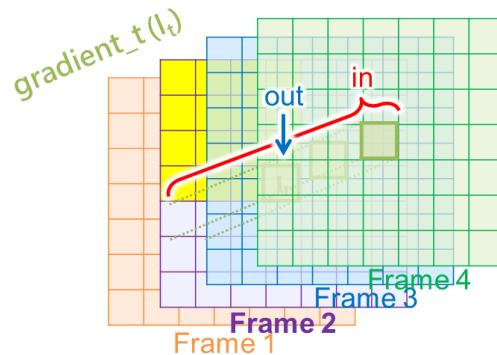
// Calculate Ix
gradient_x_value = pix0*GRAD_WEIGHTS[4] + pix1*GRAD_WEIGHTS[3] + pix2*GRAD_WEIGHTS[2] + pix3*GRAD_WEIGHTS[1] + pix4*GRAD_WEIGHTS[0];

pix_buf3 = pix_buf2;
pix_buf2 = pix_buf1;
pix_buf1 = pix_buf0;
pix_buf0 = pix0;

// Write output Ix streaming interface
if ((x >= 4) && (x < widthIn)) {
    gradient_x.write(gradient_x_value);
    //cout << "HLS: " << gradient_x_value << endl;
} else if (x >= 2) {
    gradient_x.write(0);
}

```

(5) OpticalFlow_gradient_z_calc.h :



此步驟將 32bits 的 input frames 讀取進入此 module 當中，每一個 cycle 所 input 的 frames 對應到 $t = 1, 2, 3, 4$ 。即可使用此 input 進行 output data 的計算，因此每一 cycle 即可輸出對應的 gradient_z。

```

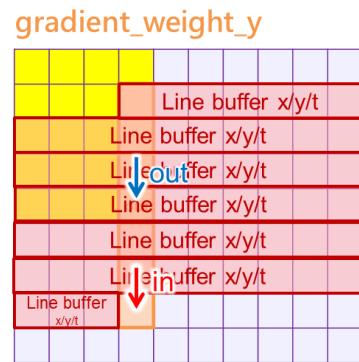
// read input channels
input_frames_delayed_value = input_frames_delayed.read();
frame1_value = (input_t)(input_frames_delayed_value.slc<8>(0));
frame2_value = (input_t)(input_frames_delayed_value.slc<8>(8));
frame3_value = (input_t)(input_frames_delayed_value.slc<8>(16));
frame4_value = (input_t)(input_frames_delayed_value.slc<8>(24));

// Calculate Iz
gradient_z_value = frame1_value*GRAD_WEIGHTS_Z[0] + frame2_value*GRAD_WEIGHTS_Z[1] + frame3_value*GRAD_WEIGHTS_Z[2] + frame4_value*GRAD_WEIGHTS_Z[3];

// Write output Iz streaming interface
gradient_z.write(gradient_z_value);

```

(6) OpticalFlow_gradient_weight_y.h :



gradient_x / gradient_y / gradient_t

此步驟進行的計算與 gradient_y 非常相似，但由於會需要同時使用 gradient_x, gradient_y, gradient_z，因此會需要使用對應 gradient_x, gradient_y, gradient_zn 所需的 line buffer，且因為 kernel length = 7，因此需要開三組，每組 7 個 line buffer。

```
// Line buffers store pixel line history - Mapped to RAM
pixel2x_t line_buf6_Ix[MAX_WIDTH/2];
pixel2x_t line_buf5_Ix[MAX_WIDTH/2];
pixel2x_t line_buf4_Ix[MAX_WIDTH/2];
pixel2x_t line_buf3_Ix[MAX_WIDTH/2];
pixel2x_t line_buf2_Ix[MAX_WIDTH/2];
pixel2x_t line_buf1_Ix[MAX_WIDTH/2];
pixel2x_t line_buf0_Ix[MAX_WIDTH/2];
pixel2x_t line_buf6_Iy[MAX_WIDTH/2];
pixel2x_t line_buf5_Iy[MAX_WIDTH/2];
pixel2x_t line_buf4_Iy[MAX_WIDTH/2];
pixel2x_t line_buf3_Iy[MAX_WIDTH/2];
pixel2x_t line_buf2_Iy[MAX_WIDTH/2];
pixel2x_t line_buf1_Iy[MAX_WIDTH/2];
pixel2x_t line_buf0_Iy[MAX_WIDTH/2];
pixel2x_t line_buf6_Iz[MAX_WIDTH/2];
pixel2x_t line_buf5_Iz[MAX_WIDTH/2];
pixel2x_t line_buf4_Iz[MAX_WIDTH/2];
pixel2x_t line_buf3_Iz[MAX_WIDTH/2];
pixel2x_t line_buf2_Iz[MAX_WIDTH/2];
pixel2x_t line_buf1_Iz[MAX_WIDTH/2];
pixel2x_t line_buf0_Iz[MAX_WIDTH/2];
```

在此處我們一樣使用 ping-pong buffer 的 design，由於一次會有 3 個 input stream，因此使用 3 組 ping-pong buffer，而且由於上一步驟所產生的 gradient 皆為 32 bits 的 output，因此我們的 ping-pong buffer 為 64 bits，每次更新的 element 為 32 bits。

```

pixel2x_t rdbuf0_Ix, rdbuf1_Ix, rdbuf2_Ix, rdbuf3_Ix, rdbuf4_Ix, rdbuf5_Ix, rdbuf6_Ix;
pixel2x_t rdbuf0_Iy, rdbuf1_Iy, rdbuf2_Iy, rdbuf3_Iy, rdbuf4_Iy, rdbuf5_Iy, rdbuf6_Iy;
pixel2x_t rdbuf0_Iz, rdbuf1_Iz, rdbuf2_Iz, rdbuf3_Iz, rdbuf4_Iz, rdbuf5_Iz, rdbuf6_Iz;
pixel2x_t wrbuf0_Ix;
pixel2x_t wrbuf0_Iy;
pixel2x_t wrbuf0_Iz;

pixel_t Ix0, Ix1, Ix2, Ix3, Ix4, Ix5, Ix6;
pixel_t Iy0, Iy1, Iy2, Iy3, Iy4, Iy5, Iy6;
pixel_t Iz0, Iz1, Iz2, Iz3, Iz4, Iz5, Iz6;
pixel1x_t Ix0_buf, Ix1_buf, Ix2_buf, Ix3_buf, Ix4_buf, Ix5_buf, Ix6_buf;
pixel1x_t Iy0_buf, Iy1_buf, Iy2_buf, Iy3_buf, Iy4_buf, Iy5_buf, Iy6_buf;
pixel1x_t Iz0_buf, Iz1_buf, Iz2_buf, Iz3_buf, Iz4_buf, Iz5_buf, Iz6_buf;

gradient_t y_filtered_value;

```

接下來的部分與 gradient_y 部分雷同，將 data 讀取進入後，按每一 bit 的位置 assign 到 l_buf (datatype 為 ac_int)。

```

// read input channel
if (y <= heightIn-1) {
    Ix0 = gradient_x.read();
    Iy0 = gradient_y.read();
    Iz0 = gradient_z.read();

    // transform into ac_int type, in order to set write_data_buffer
    for(uint i=0; i<PIXEL_T_BIT_WIDTH; i++){
        Ix0_buf[i] = Ix0[i];
        Iy0_buf[i] = Iy0[i];
        Iz0_buf[i] = Iz0[i];
    }
}

```

將 buffer 當中的 data 儲存至 line buffer 當中，並且依照 ping-pong buffer 的格式寫入。

```

// Read line buffers into read buffer caches on even iterations of COL loop
if ( (x&1) == 0 ) {
    // vertical window of pixels
    rdbuf6_Ix = line_buf6_Ix[x/2];
    rdbuf5_Ix = line_buf5_Ix[x/2];
    rdbuf4_Ix = line_buf4_Ix[x/2];
    rdbuf3_Ix = line_buf3_Ix[x/2];
    rdbuf2_Ix = line_buf2_Ix[x/2];
    rdbuf1_Ix = line_buf1_Ix[x/2];
    rdbuf0_Ix = line_buf0_Ix[x/2];
    rdbuf6_Iy = line_buf6_Iy[x/2];
    rdbuf5_Iy = line_buf5_Iy[x/2];
    rdbuf4_Iy = line_buf4_Iy[x/2];
    rdbuf3_Iy = line_buf3_Iy[x/2];
    rdbuf2_Iy = line_buf2_Iy[x/2];
    rdbuf1_Iy = line_buf1_Iy[x/2];
    rdbuf0_Iy = line_buf0_Iy[x/2];
    rdbuf6_Iz = line_buf6_Iz[x/2];
    rdbuf5_Iz = line_buf5_Iz[x/2];
    rdbuf4_Iz = line_buf4_Iz[x/2];
    rdbuf3_Iz = line_buf3_Iz[x/2];
    rdbuf2_Iz = line_buf2_Iz[x/2];
    rdbuf1_Iz = line_buf1_Iz[x/2];
    rdbuf0_Iz = line_buf0_Iz[x/2];
} else { // Write line buffer caches on odd iterations of COL loop
    line_buf6_Ix[x/2] = rdbuf5_Ix; // copy previous line
    line_buf5_Ix[x/2] = rdbuf4_Ix; // copy previous line
    line_buf4_Ix[x/2] = rdbuf3_Ix; // copy previous line
    line_buf3_Ix[x/2] = rdbuf2_Ix; // copy previous line
    line_buf2_Ix[x/2] = rdbuf1_Ix; // copy previous line
    line_buf1_Ix[x/2] = rdbuf0_Ix; // copy previous line
    line_buf0_Ix[x/2] = wrbuf0_Ix; // store current line
    line_buf6_Tw[x/2] = wrbuf0_Tw; // copy previous line
}

```

再依照 ping-pong buffer 的格式將 data 讀出(top 32 bit and bottom 32 bit)。

```
// Get 32-bit data from read buffer caches, lower 32 on even iterations of COL loop (when "PIXEL_T_BIT_WIDTH=32")
Ix6_buf = ((x&1)==0) ? rdbuf5_Ix.slc<PIXEL_T_BIT_WIDTH>(0) : rdbuf5_Ix.slc<PIXEL_T_BIT_WIDTH>(PIXEL_T_BIT_WIDTH);
Ix5_buf = ((x&1)==0) ? rdbuf4_Ix.slc<PIXEL_T_BIT_WIDTH>(0) : rdbuf4_Ix.slc<PIXEL_T_BIT_WIDTH>(PIXEL_T_BIT_WIDTH);
Ix4_buf = ((x&1)==0) ? rdbuf3_Ix.slc<PIXEL_T_BIT_WIDTH>(0) : rdbuf3_Ix.slc<PIXEL_T_BIT_WIDTH>(PIXEL_T_BIT_WIDTH);
Ix3_buf = ((x&1)==0) ? rdbuf2_Ix.slc<PIXEL_T_BIT_WIDTH>(0) : rdbuf2_Ix.slc<PIXEL_T_BIT_WIDTH>(PIXEL_T_BIT_WIDTH);
Ix2_buf = ((x&1)==0) ? rdbuf1_Ix.slc<PIXEL_T_BIT_WIDTH>(0) : rdbuf1_Ix.slc<PIXEL_T_BIT_WIDTH>(PIXEL_T_BIT_WIDTH);
Ix1_buf = ((x&1)==0) ? rdbuf0_Ix.slc<PIXEL_T_BIT_WIDTH>(0) : rdbuf0_Ix.slc<PIXEL_T_BIT_WIDTH>(PIXEL_T_BIT_WIDTH);
Iy6_buf = ((x&1)==0) ? rdbuf5_Iy.slc<PIXEL_T_BIT_WIDTH>(0) : rdbuf5_Iy.slc<PIXEL_T_BIT_WIDTH>(PIXEL_T_BIT_WIDTH);
Iy5_buf = ((x&1)==0) ? rdbuf4_Iy.slc<PIXEL_T_BIT_WIDTH>(0) : rdbuf4_Iy.slc<PIXEL_T_BIT_WIDTH>(PIXEL_T_BIT_WIDTH);
Iy4_buf = ((x&1)==0) ? rdbuf3_Iy.slc<PIXEL_T_BIT_WIDTH>(0) : rdbuf3_Iy.slc<PIXEL_T_BIT_WIDTH>(PIXEL_T_BIT_WIDTH);
Iy3_buf = ((x&1)==0) ? rdbuf2_Iy.slc<PIXEL_T_BIT_WIDTH>(0) : rdbuf2_Iy.slc<PIXEL_T_BIT_WIDTH>(PIXEL_T_BIT_WIDTH);
Iy2_buf = ((x&1)==0) ? rdbuf1_Iy.slc<PIXEL_T_BIT_WIDTH>(0) : rdbuf1_Iy.slc<PIXEL_T_BIT_WIDTH>(PIXEL_T_BIT_WIDTH);
Iy1_buf = ((x&1)==0) ? rdbuf0_Iy.slc<PIXEL_T_BIT_WIDTH>(0) : rdbuf0_Iy.slc<PIXEL_T_BIT_WIDTH>(PIXEL_T_BIT_WIDTH);
Iz6_buf = ((x&1)==0) ? rdbuf5_Iz.slc<PIXEL_T_BIT_WIDTH>(0) : rdbuf5_Iz.slc<PIXEL_T_BIT_WIDTH>(PIXEL_T_BIT_WIDTH);
Iz5_buf = ((x&1)==0) ? rdbuf4_Iz.slc<PIXEL_T_BIT_WIDTH>(0) : rdbuf4_Iz.slc<PIXEL_T_BIT_WIDTH>(PIXEL_T_BIT_WIDTH);
Iz4_buf = ((x&1)==0) ? rdbuf3_Iz.slc<PIXEL_T_BIT_WIDTH>(0) : rdbuf3_Iz.slc<PIXEL_T_BIT_WIDTH>(PIXEL_T_BIT_WIDTH);
Iz3_buf = ((x&1)==0) ? rdbuf2_Iz.slc<PIXEL_T_BIT_WIDTH>(0) : rdbuf2_Iz.slc<PIXEL_T_BIT_WIDTH>(PIXEL_T_BIT_WIDTH);
Iz2_buf = ((x&1)==0) ? rdbuf1_Iz.slc<PIXEL_T_BIT_WIDTH>(0) : rdbuf1_Iz.slc<PIXEL_T_BIT_WIDTH>(PIXEL_T_BIT_WIDTH);
Iz1_buf = ((x&1)==0) ? rdbuf0_Iz.slc<PIXEL_T_BIT_WIDTH>(0) : rdbuf0_Iz.slc<PIXEL_T_BIT_WIDTH>(PIXEL_T_BIT_WIDTH);
```

並轉換回 ac_fixed 的 datatype，此時即可進行 gradient 的運算。

```
// transform back into ac_fixed type
for(uint i=0; i<PIXEL_T_BIT_WIDTH; i++){
    Ix6[i] = Ix6_buf[i];
    Ix5[i] = Ix5_buf[i];
    Ix4[i] = Ix4_buf[i];
    Ix3[i] = Ix3_buf[i];
    Ix2[i] = Ix2_buf[i];
    Ix1[i] = Ix1_buf[i];
    Iy6[i] = Iy6_buf[i];
    Iy5[i] = Iy5_buf[i];
    Iy4[i] = Iy4_buf[i];
    Iy3[i] = Iy3_buf[i];
    Iy2[i] = Iy2_buf[i];
    Iy1[i] = Iy1_buf[i];
    Iz6[i] = Iz6_buf[i];
    Iz5[i] = Iz5_buf[i];
    Iz4[i] = Iz4_buf[i];
    Iz3[i] = Iz3_buf[i];
    Iz2[i] = Iz2_buf[i];
    Iz1[i] = Iz1_buf[i];
}
```

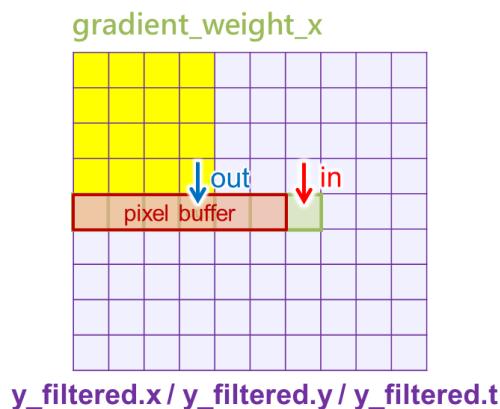
與 gradient_y 相同，雖然此時 gradient 的 data 可能尚未準備完畢，但因為 data 若並非 available 時，stream 的 valid 訊號並不會拉起，因此可以將後續的計算與 output 直接寫出，同時也須注意 kernel 與圖片邊界的條件。

```
if ((y >= 6) && (y < heightIn)) {
    // Calculate y_filtered_value
    y_filtered_value.x = Ix0*GRAD_FILTER[0] + Ix1*GRAD_FILTER[1] + Ix2*GRAD_FILTER[2] + Ix3*GRAD_FILTER[3] + Ix4*GRAD_FILTER[4] + Ix5*GRAD_FILTER[5] + Ix6*GRAD_FILTER[6];
    y_filtered_value.y = Iy0*GRAD_FILTER[0] + Iy1*GRAD_FILTER[1] + Iy2*GRAD_FILTER[2] + Iy3*GRAD_FILTER[3] + Iy4*GRAD_FILTER[4] + Iy5*GRAD_FILTER[5] + Iy6*GRAD_FILTER[6];
    y_filtered_value.z = Iz0*GRAD_FILTER[0] + Iz1*GRAD_FILTER[1] + Iz2*GRAD_FILTER[2] + Iz3*GRAD_FILTER[3] + Iz4*GRAD_FILTER[4] + Iz5*GRAD_FILTER[5] + Iz6*GRAD_FILTER[6];
    //if ((x==990) && (y==435)){
    //    cout << Ix0 << ", " << Ix1 << ", " << Ix2 << ", " << Ix3 << ", " << Ix4 << ", " << Ix5 << ", " << Ix6 << endl;
    //    cout << GRAD_FILTER[0] << ", " << GRAD_FILTER[1] << ", " << GRAD_FILTER[2] << ", " << GRAD_FILTER[3] << ", " << GRAD_FILTER[4] << ", " << GRAD_FILTER[5] << ", " << GRAD_FILTER[6] << endl;
    //    cout << y_filtered_value.x << endl;
    //}

    // Write output y_filtered_value streaming interface
    y_filtered.write(y_filtered_value);
} else if (y >= 3) {
    // Calculate y_filtered_value
    y_filtered_value.x = 0;
    y_filtered_value.y = 0;
    y_filtered_value.z = 0;

    // Write output y_filtered_value streaming interface
    y_filtered.write(y_filtered_value);
}
```

(7) OpticalFlow_gradient_weight_x.h :



此 function 的實作方法與 OpticalFlow_gradient_x_calc.h (從 input I 值計算出 I_x 值) 非常類似，因為它們都是要將 input data 做 x 方向的 filtering，指示在此因為 filter coefficient (kernel 寬度)為 7 個，因此 pixel buffer 數量需增加為 6 個。因此我們仿照 OpticalFlow_gradient_x_calc.h 的寫法，先定義 piuxel buffer 及要拿來做運算的變數：

```

17     // gradient buffers store gradient history
18     gradient_t gradient_buf0 = {0,0,0};
19     gradient_t gradient_buf1 = {0,0,0};
20     gradient_t gradient_buf2 = {0,0,0};
21     gradient_t gradient_buf3 = {0,0,0};
22     gradient_t gradient_buf4 = {0,0,0};
23     gradient_t gradient_buf5 = {0,0,0};
24     gradient_t gradient_buf6 = {0,0,0};
25
26     gradient_t gradient0;
27     gradient_t gradient1;
28     gradient_t gradient2;
29     gradient_t gradient3;
30     gradient_t gradient4;
31     gradient_t gradient5;
32     gradient_t gradient6;
33
34
35     gradient_t filtered_gradient_value;
36

```

接著重複每個 pixel，將 input 的 stream data 讀取進來，若已超出 pixel array 範圍則不讀取 input stream。

```

37     Gradient_weight_x_ROW: for (maxHType y=0; ; y++) {
38         Gradient_weight_x_COLUMN: for (maxWType x=0; ; x++) {
39             // read input value
40             if (x <= widthIn-1) {
41                 gradient0 = y_filtered.read(); // Read from input stream
42             } else {
43                 gradient0 = gradient_buf0;
44             }
45         }

```

接下來將過去存在 buffer 中的值取出來，放到計算用的變數：

```

46     gradient1 = gradient_buf0;
47     gradient2 = gradient_buf1;
48     gradient3 = gradient_buf2;
49     gradient4 = gradient_buf3;
50     gradient5 = gradient_buf4;
51     gradient6 = gradient_buf5;
52

```

將目前的變數存回 buffer，以供未來使用

```

54     gradient_buf6 = gradient_buf5;
55     gradient_buf5 = gradient_buf4;
56     gradient_buf4 = gradient_buf3;
57     gradient_buf3 = gradient_buf2;
58     gradient_buf2 = gradient_buf1;
59     gradient_buf1 = gradient_buf0;
60     gradient_buf0 = gradient0;
61

```

接著就進行與 filter kernel 之間的 convolution 運算，並輸出至 filtered_gradient 這個 output stream，而邊界 kernel 超出的部分則直接 pending 為 0。

```

62     if ((x >= 6) && (x < widthIn)) {
63         // Calculate filtered_gradient
64         filtered_gradient_value.x = gradient0.x*GRAD_FILTER[0] + gradient1.x*GRAD_FILTER[1] + gradient2.x*GRAD_FILTER[2] + gradient3.x*GRAD_FILTER[3] + gradient4.x*GRAD_FILTER[4] + gradient5.x*GRAD_FILTER[5] + gradient6.x*
65         filtered_gradient_value.y = gradient0.y*GRAD_FILTER[0] + gradient1.y*GRAD_FILTER[1] + gradient2.y*GRAD_FILTER[2] + gradient3.y*GRAD_FILTER[3] + gradient4.y*GRAD_FILTER[4] + gradient5.y*GRAD_FILTER[5] + gradient6.y*
66         filtered_gradient_value.z = gradient0.z*GRAD_FILTER[0] + gradient1.z*GRAD_FILTER[1] + gradient2.z*GRAD_FILTER[2] + gradient3.z*GRAD_FILTER[3] + gradient4.z*GRAD_FILTER[4] + gradient5.z*GRAD_FILTER[5] + gradient6.z*
67
68         // Write output filtered_gradient streaming interface
69         filtered_gradient.write(filtered_gradient_value);
70     } else if (x >= 3) {
71         // Calculate filtered_gradient
72         filtered_gradient_value.x = 0;
73         filtered_gradient_value.y = 0;
74         filtered_gradient_value.z = 0;
75
76         // Write output filtered_gradient streaming interface
77         filtered_gradient.write(filtered_gradient_value);
78     }
79

```

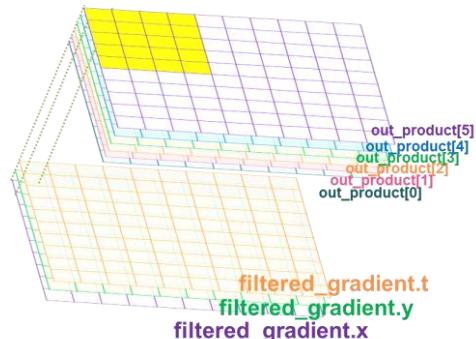
最後是 boundary condition 的設定，由於我們從 $x \geq 3$ 才開始寫 channel，所以最後需要到 $x == widthIn+2$ 才停止此 row 的輸出，並且最後幾個 pixel 就輸出 0：

```

80         // programmable width exit condition
81         if (x == widthIn+2) {
82             | break;
83         }
84     }
85     // programmable height exit condition
86     if (y == heightIn-1) {
87         | break;
88     }

```

(8) OpticalFlow_outer_product.h :



將 Ix、Iy、It 3 個 channel 同時讀入，並計算出 6 個 outer product 的值。

```

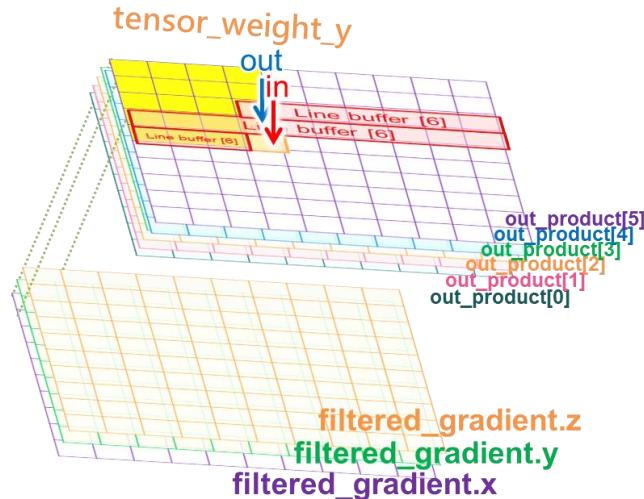
// read input channels
filtered_gradient_value = filtered_gradient.read();

// Calculate out_product_value
out_product_value.val[0] = filtered_gradient_value.x * filtered_gradient_value.x;
out_product_value.val[1] = filtered_gradient_value.y * filtered_gradient_value.y;
out_product_value.val[2] = filtered_gradient_value.z * filtered_gradient_value.z;
out_product_value.val[3] = filtered_gradient_value.x * filtered_gradient_value.y;
out_product_value.val[4] = filtered_gradient_value.x * filtered_gradient_value.z;
out_product_value.val[5] = filtered_gradient_value.y * filtered_gradient_value.z;

// Write output Iz streaming interface
out_product.write(out_product_value);

```

(9) OpticalFlow_tensor_weight_y.h :



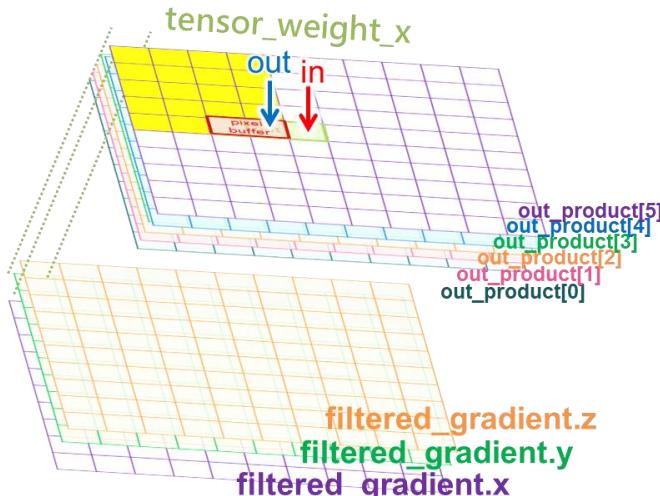
此 module 與前面 y 方向的 kernel 方法相同，只是在此需要將每個 pixel 所對應的 6 個值都用 line buffer 存起來，因此會需要非常多的 line buffer。

```
// Get 32*6-bit data from read buffer caches, lower 32*6 on even iterations of COL loop (when "OUTER_PIXEL_T_BIT_WIDTH=32")
out_product2_buf = ((x&1)==0) ? rdbuf1.slc<OUTER_PIXEL_T_BIT_WIDTH*6>(0) : rdbuf1.slc<OUTER_PIXEL_T_BIT_WIDTH*6>(OUTER_PIXEL_T_BIT_WIDTH*6);
out_product1_buf = ((x&1)==0) ? rdbuf0.slc<OUTER_PIXEL_T_BIT_WIDTH*6>(0) : rdbuf0.slc<OUTER_PIXEL_T_BIT_WIDTH*6>(OUTER_PIXEL_T_BIT_WIDTH*6);

// split into 6 components & transform back into ac_fixed type
for(uint component=0; component<6; component++){
    for(uint i=0; i<OUTER_PIXEL_T_BIT_WIDTH; i++){
        out_product2.val[component][i] = out_product2_buf[OUTER_PIXEL_T_BIT_WIDTH*component+i];
        out_product1.val[component][i] = out_product1_buf[OUTER_PIXEL_T_BIT_WIDTH*component+i];
    }
}

if ((y >= 2) && (y < heightIn)) {
    // Calculate tensor_y_value
    tensor_y_value.val[0] = out_product0.val[0]*TENSOR_FILTER[0] + out_product1.val[0]*TENSOR_FILTER[1] + out_product2.val[0]*TENSOR_FILTER[2];
    tensor_y_value.val[1] = out_product0.val[1]*TENSOR_FILTER[0] + out_product1.val[1]*TENSOR_FILTER[1] + out_product2.val[1]*TENSOR_FILTER[2];
    tensor_y_value.val[2] = out_product0.val[2]*TENSOR_FILTER[0] + out_product1.val[2]*TENSOR_FILTER[1] + out_product2.val[2]*TENSOR_FILTER[2];
    tensor_y_value.val[3] = out_product0.val[3]*TENSOR_FILTER[0] + out_product1.val[3]*TENSOR_FILTER[1] + out_product2.val[3]*TENSOR_FILTER[2];
    tensor_y_value.val[4] = out_product0.val[4]*TENSOR_FILTER[0] + out_product1.val[4]*TENSOR_FILTER[1] + out_product2.val[4]*TENSOR_FILTER[2];
    tensor_y_value.val[5] = out_product0.val[5]*TENSOR_FILTER[0] + out_product1.val[5]*TENSOR_FILTER[1] + out_product2.val[5]*TENSOR_FILTER[2];
```

(10) OpticalFlow_tensor_weight_x.h :



```
// Calculate tensor_shift_value
tensor_value.val[0] = tensor0.val[0]*TENSOR_FILTER[0] + tensor1.val[0]*TENSOR_FILTER[1] + tensor2.val[0]*TENSOR_FILTER[2];
tensor_value.val[1] = tensor0.val[1]*TENSOR_FILTER[0] + tensor1.val[1]*TENSOR_FILTER[1] + tensor2.val[1]*TENSOR_FILTER[2];
tensor_value.val[2] = tensor0.val[2]*TENSOR_FILTER[0] + tensor1.val[2]*TENSOR_FILTER[1] + tensor2.val[2]*TENSOR_FILTER[2];
tensor_value.val[3] = tensor0.val[3]*TENSOR_FILTER[0] + tensor1.val[3]*TENSOR_FILTER[1] + tensor2.val[3]*TENSOR_FILTER[2];
tensor_value.val[4] = tensor0.val[4]*TENSOR_FILTER[0] + tensor1.val[4]*TENSOR_FILTER[1] + tensor2.val[4]*TENSOR_FILTER[2];
tensor_value.val[5] = tensor0.val[5]*TENSOR_FILTER[0] + tensor1.val[5]*TENSOR_FILTER[1] + tensor2.val[5]*TENSOR_FILTER[2];
```

```
for (uint i=0; i<TENSOR_LONG_PIXEL_T_BIT_WIDTH; i=i+1) {
    tensor_value_compare_to_sign_bit.val[0][i] = tensor_value.val[0][TENSOR_LONG_PIXEL_T_BIT_WIDTH-1]*tensor_value.val[0][i];
    tensor_value_compare_to_sign_bit.val[1][i] = tensor_value.val[1][TENSOR_LONG_PIXEL_T_BIT_WIDTH-1]*tensor_value.val[1][i];
    tensor_value_compare_to_sign_bit.val[3][i] = tensor_value.val[3][TENSOR_LONG_PIXEL_T_BIT_WIDTH-1]*tensor_value.val[3][i];
    tensor_value_compare_to_sign_bit.val[4][i] = tensor_value.val[4][TENSOR_LONG_PIXEL_T_BIT_WIDTH-1]*tensor_value.val[4][i];
    tensor_value_compare_to_sign_bit.val[5][i] = tensor_value.val[5][TENSOR_LONG_PIXEL_T_BIT_WIDTH-1]*tensor_value.val[5][i];
    tensor_value_compare_to_sign_bit_bitwise_OR[i] = tensor_value_compare_to_sign_bit.val[0][i] | tensor_value_compare_to_sign_bit.val[1][i] |
    tensor_value_compare_to_sign_bit.val[3][i] | tensor_value_compare_to_sign_bit.val[4][i] | tensor_value_compare_to_sign_bit.val[5][i];
}

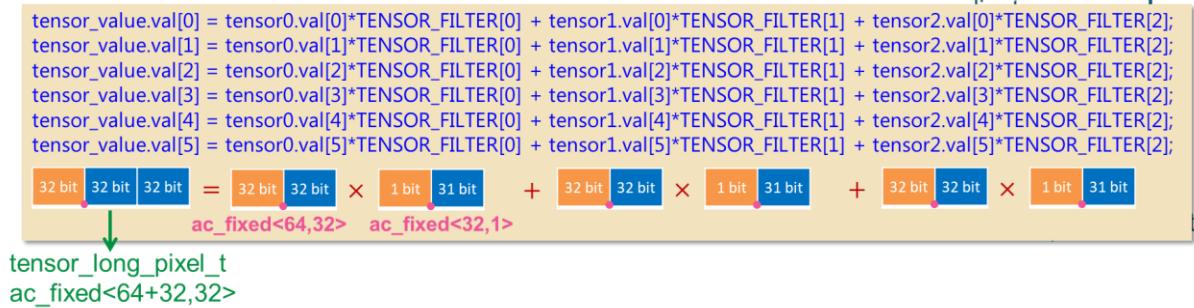
for (uint i=0; i<TENSOR_LONG_PIXEL_T_BIT_WIDTH; i=i+1) {
    tensor_value_compare_to_sign_bit_bitwise_OR_complement[i]=tensor_value_compare_to_sign_bit_bitwise_OR[TENSOR_LONG_PIXEL_T_BIT_WIDTH-1-i];
}

rightmost_1 = tensor_value_compare_to_sign_bit_bitwise_OR_complement & (~tensor_value_compare_to_sign_bit_bitwise_OR_complement);
for (uint i=1; i<TENSOR_LONG_PIXEL_T_BIT_WIDTH; i=i+1) {
    shift_value = shift_value + i*rightmost_1[i];
}
shift_value = shift_value-1;
//cout << "shift_value: " << shift_value << endl;
tensor_shift_value.val[0] = (tensor_value.val[0]<<shift_value).to_int();
tensor_shift_value.val[1] = (tensor_value.val[1]<<shift_value).to_int();
tensor_shift_value.val[2] = (tensor_value.val[2].to_int());
tensor_shift_value.val[3] = (tensor_value.val[3]<<shift_value).to_int();
tensor_shift_value.val[4] = (tensor_value.val[4]<<shift_value).to_int();
tensor_shift_value.val[5] = (tensor_value.val[5]<<shift_value).to_int();
```

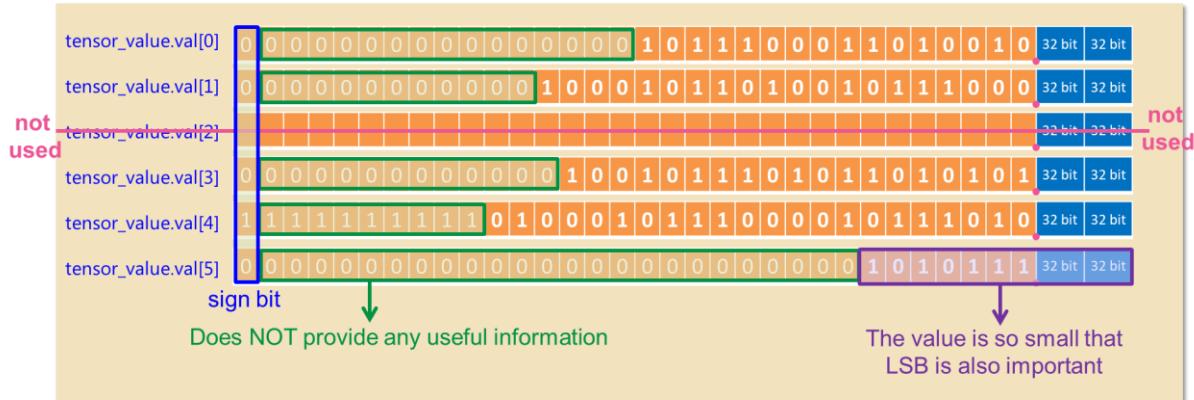
此 module 與前面 x 方向的 kernel 方法相同，只是在此的 pixel buffer 需要存住 pixel 所對應的 6 個值。在此 module 中，為了減省 bit 的浪費，我們使用 bit-shift 的技巧，並輸出 shift 的數目至下一個 module 以作 threshold 的判斷：

● Bit-shift technique

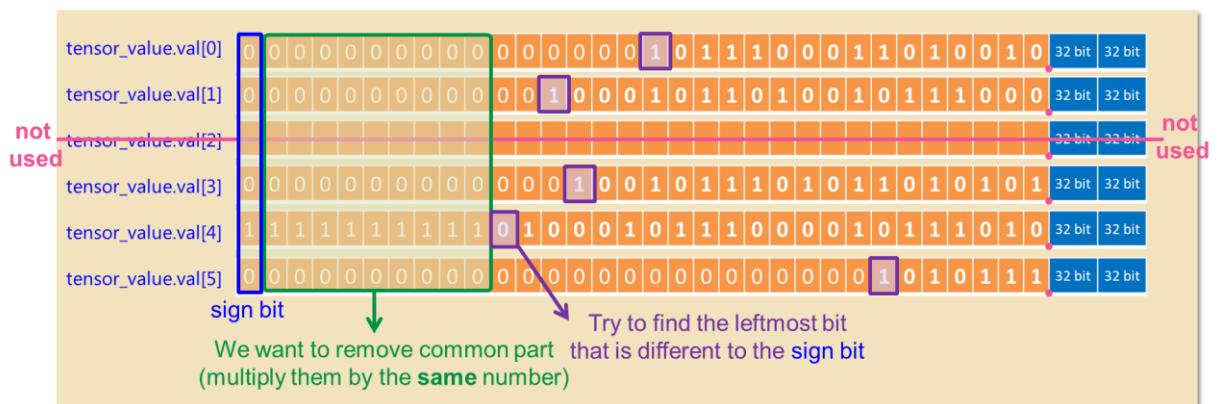
在此 module 中，由於已經經過多層 kernel 運算，而使得數值可能差異很大（有的 pixel 值已經非常大，有些 pixel 的值卻非常小），此時若我們再使用更多 bit 來儲存差距如此大的值，可能會很浪費 bit 數，以及 interface 的線寬。對於很大的數值，其實 LSB 的值並不重要，因為他頂多影響小數點後的值；而對於很小的值（例如 10^{-15} ），其實 MSB 的值並不重要，因為整數位數皆為 0。因此，若能將這些不必要的 bit 部分刪除，即可省下後續很多運算量。我們想出了 " Bit-shift technique " 來解決這個 issue，以此 module 為例：



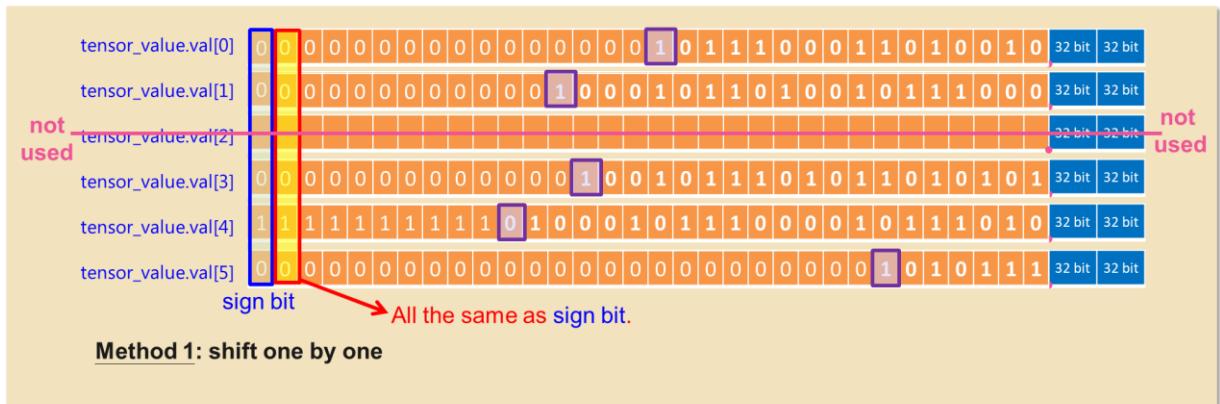
首先，我們先將這些 bit 都先不拿掉，而是作底下的判斷：



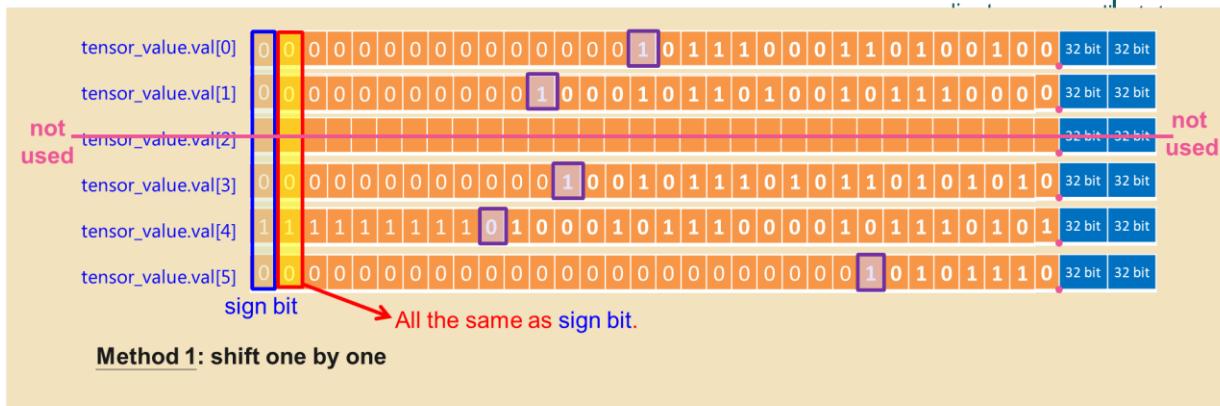
`tensor_value.val[2]`在後面的運算中不會使用到，因此可以先不管。我們可發現，綠色的部分都算是多餘的 bit，但由於後續運算，我們必須要讓這個值都 shift 相同的量，再 truncate 成 MSB 的 32 bit。



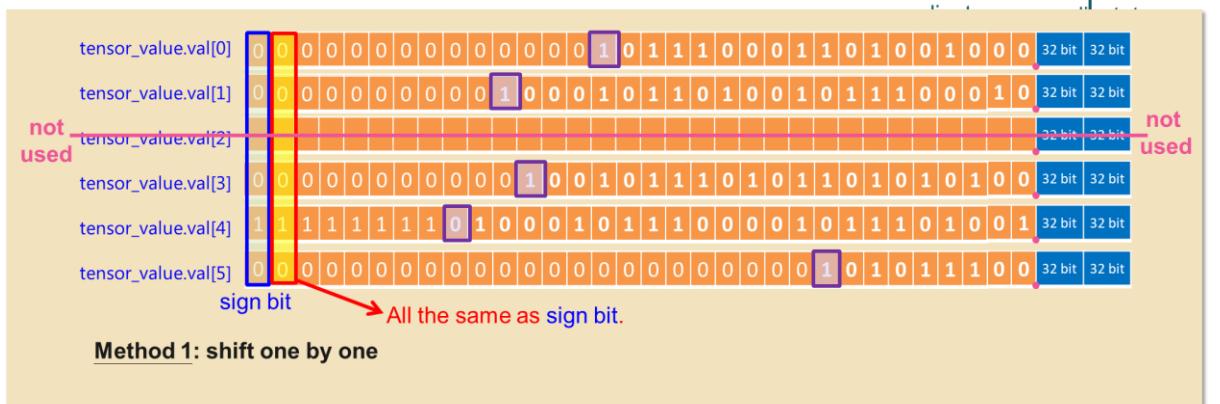
第一個方法為透過 while loop 花費許多 cycle 來判斷是否要向左 shift 1 bit：



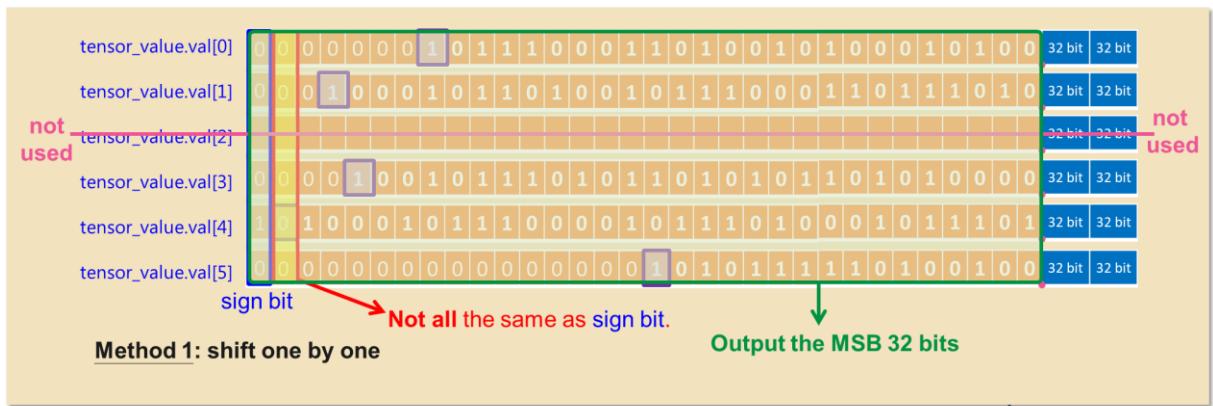
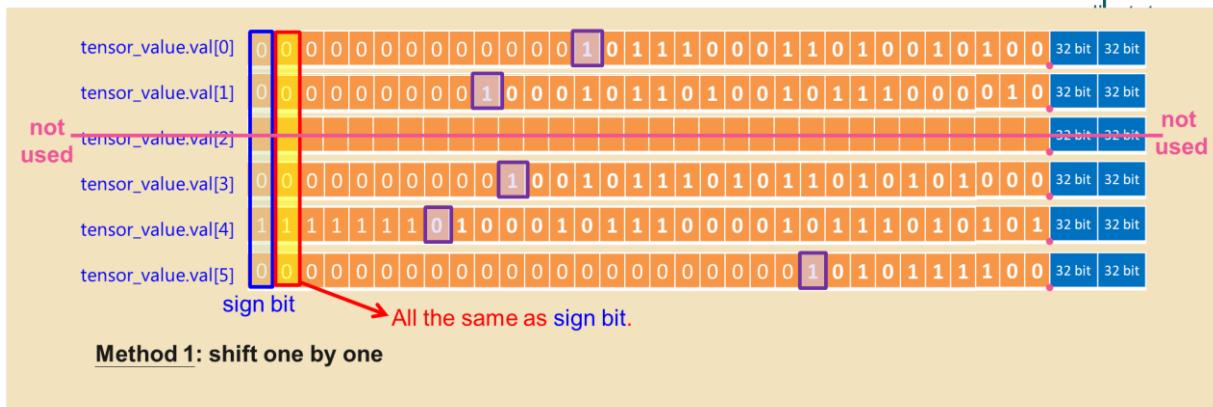
向左 shift 1 bit:



再向左 shift 1 bit:



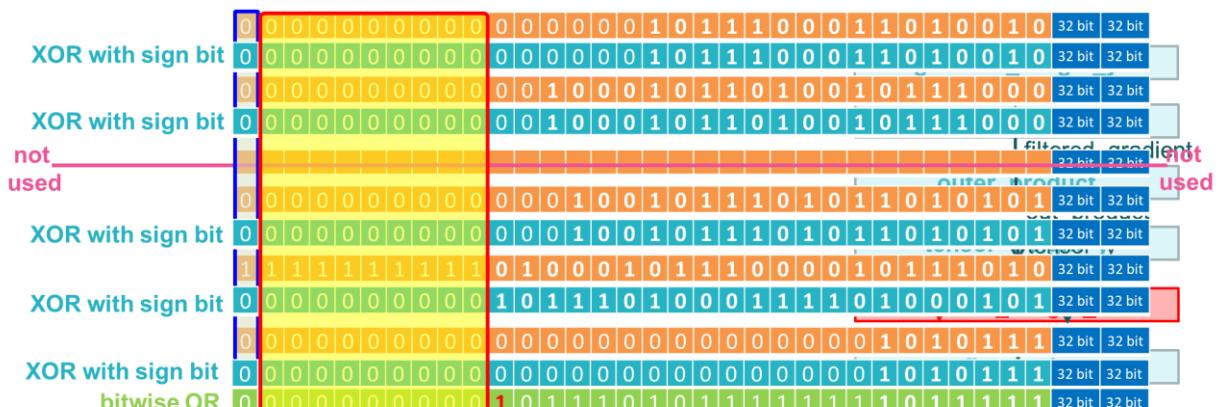
再向左 shift 1 bit:



即可得知 shift 量，並輸出最有意義的部分。

然而，由於使用 while loop，因此在 Catapult 中會因為 unbounded 而無法 unroll，而導致 cycle 數極高。因此我們想出了第二種方法：使用 hardware logic gates 來在同一個 cycle 中找出要 shift 的數量，方法如下：

先取每個 bit 的 XOR with sign bit，並將同一個位置的 5 值再做 OR。



接著要找出 leftmost 1 的位置，我們先將每個 bit 做 flip (MSB1u04t/6)

LSB、LSB 變成 MSB) 後，即可使用現成的做法，透過 A & (-A)的方式找出 rightmost bit 的位置，此位置即為 leftmost 1 距離 sign bit 的距離了！

bitwise OR 0 0 0 0 0 0 0 0 0 0 1 0 1 1 1 0 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 32 bit | 32 bit

◆ We want to find the position of the “leftmost 1”:

flip 32 bit | 32 bit 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1 0 1 0 1 1 1 1 0 1 0 0 0 0 0 0 0 0 0 0 0

Then to find the position of the “rightmost 1”:

~flip 32 bit | 32 bit 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 1 0 1 0 0 0 1 0 1 1 1 1 1 1 1 1 1 1 1

~flip (=~flip+1) 32 bit | 32 bit 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 1 0 1 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0

flip & (~flip) 32 bit | 32 bit 0 1 0 0 0 0 0 0 0 0 0 0

The position of the only “1” is our goal.

最後可透過與常數相乘得到此位置的數值（因為只有 1 bit 為 1）：

◆ We want to find the position of the “leftmost 1”:

flip & (~flip) 32 bit | 32 bit 0 1 0 0 0 0 0 0 0 0 0

The position of the only “1” is our goal.

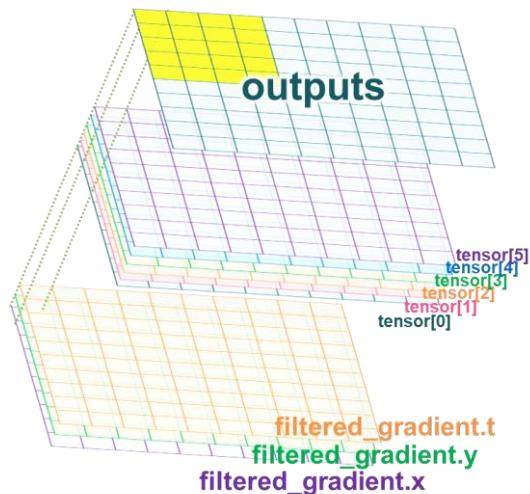
→ But we cannot use “while” loop to find the position; instead, we use the following technique:

flip & (~flip) 32 bit | 32 bit 0 1 0 0 0 0 0 0 0 0 0
 × 31 7 6 5 4 3 2 1 0

→ Then we can get the position without using “while” loop !!

在 shift 之後明顯會更接近軟體的運算結果，執行過程如下：

(11)OpticalFlow_flow_calc.h :



此部分為 LK 演算法的核心運算部分，也就是將 smooth 後的值帶入 LK 方法的最終結果矩陣運算：

$$\begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} \Sigma I_y^2 \Sigma I_x I_t - \Sigma I_x I_y \Sigma I_y I_t \\ 2 \Sigma I_x I_y - \Sigma I_x^2 \Sigma I_y^2 \\ \Sigma I_x^2 \Sigma I_y I_t - \Sigma I_x I_y \Sigma I_x I_t \\ 2 \Sigma I_x I_y - \Sigma I_x^2 \Sigma I_y^2 \end{bmatrix}$$

由於使用硬體實作除法運算(division operation)會非常消耗 resources，並且 Catapult 不支援除法器的合成，因此我們決定將分子和分母分別輸出，再到 testbench 中做運算（未來可以將這個運算送到 Caravel SoC 中的 SoC 使用 CPU 來分擔運算），也因此需要輸出「u 的分子」、「v 的分子」、「共同的分母」這三個輸出值 for each pixel，但由於 user project 對外(FSIC)的 output stream 只能有一條，且限制在 32-bit，也因此必須分為 3 個 cycle 輸出這三筆 data。

```
// Write output optical flow (velocity) streaming interface
output_value = denominator_value_after_shift.slc<32>(VEL_PIXEL_T_BIT_WIDTH-32);
output.write(output_value);
output_value = velocity_value_after_shift.x.slc<32>(VEL_PIXEL_T_BIT_WIDTH-32);
output.write(output_value);
output_value = velocity_value_after_shift.y.slc<32>(VEL_PIXEL_T_BIT_WIDTH-32);
output.write(output_value);
```

這導致此 module 為整個 design 的 bottleneck，會需要至少 3 個 cycle 才能輸出 1 個 pixel 的資料，導致 II 不能做到 1。這個部分會在底下介紹 Catapult tool 合成的部分介紹到我們的處理方法。

另外，由於 input 為 32-bit，經過相乘後會形成 64 bit 的 data，但 output stream 却只能 32 bit，雖然可以分成 2 個 cycle 輸出，但這樣又會使速度進一步降成 0.5 倍，對執行時間的 performance 不是個好結果，因此我們也在這裡使用與前面相同的 bit-shift 的技巧，只取出 common part 的有意義的部分，以盡量保留精準的結果。

```

denominator_value = (tensor_shift_value.val[0]*tensor_shift_value.val[1] - tensor_shift_value.val[3]*tensor_shift_value.val[3]);
velocity_value.x = (tensor_shift_value.val[5]*tensor_shift_value.val[0] - tensor_shift_value.val[4]*tensor_shift_value.val[1]); // / denominator_value;
velocity_value.y = (tensor_shift_value.val[4]*tensor_shift_value.val[3] - tensor_shift_value.val[5]*tensor_shift_value.val[0]); // / denominator_value;

for (uint i=0; i<VEL_PIXEL_T_BIT_WIDTH; i=i+1) {
    denominator_value_compare_to_sign_bit[i] = denominator_value[VEL_PIXEL_T_BIT_WIDTH-1]*denominator_value[i];
    velocity_value_compare_to_sign_bit.x[i] = velocity_value.x[VEL_PIXEL_T_BIT_WIDTH-1]*velocity_value.x[i];
    velocity_value_compare_to_sign_bit.y[i] = velocity_value.y[VEL_PIXEL_T_BIT_WIDTH-1]*velocity_value.y[i];
    velocity_value_compare_to_sign_bit_bitwise_OR[i] = denominator_value_compare_to_sign_bit[i] | velocity_value_compare_to_sign_bit.x[i] | velocity_value_compare_to_sign_bit.y[i];
}

for (uint i=0; i<VEL_PIXEL_T_BIT_WIDTH; i=i+1) {
    velocity_value_compare_to_sign_bit_bitwise_OR_complement[i] = velocity_value_compare_to_sign_bit_bitwise_OR[VEL_PIXEL_T_BIT_WIDTH-1-i];
}
rightmost_1 = velocity_value_compare_to_sign_bit_bitwise_OR_complement & (~velocity_value_compare_to_sign_bit_bitwise_OR_complement);
for (uint i=1; i<VEL_PIXEL_T_BIT_WIDTH; i=i+1) {
    shift_value_here = shift_value_here + i*rightmost_1[i];
}
shift_value_here = shift_value_here-1;
//cout << "shift_value_here: " << shift_value_here << endl;
denominator_value_before_threshold = denominator_value<<shift_value_here;
velocity_value_before_threshold.x = velocity_value.x<<shift_value_here;
velocity_value_before_threshold.y = velocity_value.y<<shift_value_here;

```

由於這裡的運算為將 input 值相乘，因此前面的 shift 量會變成 2 倍（可想像成原本每個 input 都乘以 2^N ，因此兩個 input 相乘的值就變為原來的 2^{2N} ），將其加上這裡的 shift 量後，「 $2 \times \text{shift in tensor_weight_x} + \text{shift in flow_calc}$ 」即為整體 shift 的值。我們利用此值可以來判斷是否 shift 過多，若 shift 量大於由 testbench 外給的 shift_threshold 時，就表示原本的值太小了，往 input 方向追溯可知此現象表示這個 pixel 附近的亮度變化都很小，也就是幾乎不移動，故當 shift 量大於 shift_threshold 時，我們就設計讓 HLS 設置輸出值為 0

```

// Calculate velocity_value
if ((shift_value*2 + shift_value_here) > shift_threshold) {
    denominator_value_after_shift = 0;
    velocity_value_after_shift.x = 0;
    velocity_value_after_shift.y = 0;
} else {
    denominator_value_after_shift = denominator_value_before_threshold;
    velocity_value_after_shift.x = velocity_value_before_threshold.x;
    velocity_value_after_shift.y = velocity_value_before_threshold.y;
}

```

如此一來我們就可以透過 testbench 來決定我們要容忍多小的變化量，並在 testbench 接收 output stream 時透過判別分母為 0 來判斷是否變化過小。

4. Testbench between algorithm C & HLS

在/ASoC-Final_project-optical_flow/optical_flow_catapult/hls_c/src/OpticalFlow_tb.cpp 中，我們仿照 lab2-1 的 EdgeDetect testbench 範例 code 的寫法，改寫成測試我們的 HLS function 的 testbench。其主要是為了提供 algorithm C 及 HLS code 所需的 input data，並讀出 output data，再將兩種算法的結果作比較，並 print 在螢幕上。首先，除了一些基本的 library 要 include 外，這裡也要 include algorithm C 所在的 OpticalFlow_Algorithm.h，以及 HLS 所在的 OpticalFlow.h：

```
18 #include "OpticalFlow.h"
19 #include "OpticalFlow_Algorithm.h"
```

接著我們宣告一些會使用到的變數後，就可以將 input image 透過 bmp_read() 的函式讀進相對應的 array。

```
80     bmp_read((char*)input_location1.c_str(), &width, &height, &rarray1, &garray1, &barray1);
81     bmp_read((char*)input_location2.c_str(), &width, &height, &rarray2, &garray2, &barray2);
82     bmp_read((char*)input_location3.c_str(), &width, &height, &rarray3, &garray3, &barray3);
83     bmp_read((char*)input_location4.c_str(), &width, &height, &rarray4, &garray4, &barray4);
84     bmp_read((char*)input_location5.c_str(), &width, &height, &rarray5, &garray5, &barray5);
85     assert(width==iW);
86     assert(height==iH);
```

我們所使用的 input test pattern 是來自 rosetta 裡現有的，為如下的 4 個連續 frame 的圖片：





若連續播放，可看出他的手正往上舉當中。

接著也要設定 shift_threshold 的值，在此設定為 107，可以使得 performance 最佳，也讓 software 及 HLS 所濾掉的太小值的範圍趨近一致：

```
int shift_threshold = 107; //105;
```

然後先宣告 for HLS 的 input channel 及 output channel，以及 for algorithm C 的 input array 及 output array（由於在 rosetta 的 C code 裡面是使用 input argument by array reference 的方式，因此我們就使用 array pointer 來做輸入、輸出）：

```
//ac_channel<input_t> frame1_channel;
//ac_channel<input_t> frame2_channel;
//ac_channel<input_t> frame3_channel;
//ac_channel<input_t> frame4_channel;
//ac_channel<input_t> frame5_channel;
ac_channel<frames_t> frames_channel;
//////ac_channel<pixel_t> gradient_x_HLS; // <-----
//////ac_channel<gradient_t> gradient_x_HLS;
//////ac_channel<outer_t> gradient_x_HLS;
//////ac_channel<tensor_t> gradient_x_HLS;
////////ac_channel<velocity_t> output_HLS_channel;
ac_channel<output_stream_t> output_HLS_channel;
//ac_channel<pixel_t> denominator_HLS_channel;
////////ac_channel<vel_pixel_t> denominator_HLS_channel;
////////ac_channel<shift_t> shift_HLS_channel;
```

```

static float frame1[iH][iW];
static float frame2[iH][iW];
static float frame3[iH][iW];
static float frame4[iH][iW];
/////////static float frame5[iH][iW];
/////static float gradient_x_algorithm[iH][iW]; // -----
/////static gradient_t_sw gradient_x_algorithm[iH][iW];
/////static outer_t_sw gradient_x_algorithm[iH][iW];
/////static tensor_t_sw gradient_x_algorithm[iH][iW];
static velocity_t_sw output_algorithm[iH][iW];
static pixel_t_sw denom_algorithm[iH][iW];

```

當宣告好 input/output 的 interface 後，即可輸入 data 至 input channel/input array (原本的圖片有 RGB 三個 pixel array，但由於演算法一次只能處理一個顏色 (灰階)，因此在此我們挑選紅色 array 來計算光流)：

```

unsigned cnt = 0;
for (int y = 0; y < iH; y++) {
    for (int x = 0; x < iW; x++) {
        // input for HLS
        //////////////////////////////frame1_channel.write((input_t)(rarray1[cnt])); // just using red component (pseudo monochrome)
        //////////////////////////////frame2_channel.write((input_t)(rarray2[cnt]));
        //////////////////////////////frame3_channel.write((input_t)(rarray3[cnt]));
        //////////////////////////////frame4_channel.write((input_t)(rarray4[cnt]));
        //////////////////////////////frame5_channel.write((input_t)(rarray5[cnt]));
        //////////////////frames_channel.write(((frames_t)rarray5[cnt]) << 32) + (((frames_t)rarray4[cnt]) << 24) + (((frames_t)rarray3[cnt]) << 16) + (((frames_t)rarray2[cnt]) << 8) + (frames_t)rarray1[cnt];
        //printf("%x\n", rarray1[cnt]);
        //printf("%x\n", rarray2[cnt]);
        //printf("%x\n", rarray3[cnt]);
        //printf("%x\n", rarray4[cnt]);
        //printf("%x\n", rarray5[cnt]);
        //printf("%x\n", (((frames_t)rarray5[cnt]) << 32) + (((frames_t)rarray4[cnt]) << 24) + (((frames_t)rarray3[cnt]) << 16) + (((frames_t)rarray2[cnt]) <<
        // input for algorithm
        frame1[y][x] = float(rarray1[cnt]);
        frame2[y][x] = float(rarray2[cnt]);
        frame3[y][x] = float(rarray3[cnt]);
        frame4[y][x] = float(rarray4[cnt]);
        //////////////////frame5[y][x] = float(rarray5[cnt]);
        cnt++;
    }
}

```

開始執行 algorithm C 以及 HLS，並接上 interface 的 channel / array pointer：

29	OpticalFlow_Algorithm<iW,iH> ref_inst;
30	OpticalFlow_Top dut;

```

cout << "Running" << endl;

/////ref_inst.run(frame1,frame2,frame3,frame4,frame5,output_algorithm);
/////ref_inst.run(frame1,frame2,frame3,frame4,frame5,denom_algorithm,output_algorithm);
start_time_algorithm = clock();
ref_inst.run(frame1,frame2,frame3,frame4,denom_algorithm,output_algorithm);
stop_time_algorithm = clock();
/////ref_inst.run(frame1,frame2,frame3,frame4,frame5,gradient_x_algorithm,output_algorithm); // -----
/////dut.run(frames_channel,widthIn,heightIn,output_HLS_channel);
/////dut.run(frames_channel,widthIn,heightIn,denominator_HLS_channel,shift_HLS_channel,output_HLS_channel);
dut.run(frames_channel,widthIn,heightIn,(shift_t)shift_threshold,output_HLS_channel);
/////dut.run(frames_channel,widthIn,heightIn,denominator_HLS_channel,output_HLS_channel);
/////dut.run(frames_channel,widthIn,heightIn,gradient_x_HLS,output_HLS_channel); // -----

```

最後將 output 結果一個 pixel 接著一個 pixel 取出來，並轉換為相同 type 進行比較。

```
cnt = 0;
float sumErr_magnitude = 0;
//double magnitude_algorithm;
for (int y = 0; y < heightIn; y++) {
    for (int x = 0; x < iW; x++) {
```

由於在 HLS 中我們是一個 pixel 會輸出 u 的分子、v 的分子、分母這三項 (shift 之後的數據，但由於分子與分母皆 shift 相同的量級，因此相除之後不需再 sfiht 回來即為正確值)，因此我們會輪流讀出同一個 output channel，並指定給不同的 double 值，接著在 testbench 中作相除的動作。（若分母值輸出 0 表示超過 shift_threshold 而使 HLS 強迫輸出 0，此時表示附近的影像幾乎不變，也就因此應該不會有光流的產生，故直接指定 u=0 且 v=0）。

```
///////////////////////////////
double tolerable_error_threshold = 1;
double denominator_threshold = 1e-15;
/////////double denominator_threshold = pow(10,(-1)*2*shift_threshold);

/////////velocity_t final_velocity_HLS = output_HLS_channel.read();
output_stream_t final_velocity_HLS = output_HLS_channel.read();
double denominator_HLS = final_velocity_HLS.to_double();
final_velocity_HLS = output_HLS_channel.read();
double final_velocity_x_HLS = final_velocity_HLS.to_double();
final_velocity_HLS = output_HLS_channel.read();
double final_velocity_y_HLS = final_velocity_HLS.to_double();

/////////int shift_HLS = shift_HLS_channel.read().to_int();
/////////int shift2x_HLS = shift_HLS*2; // This is because in OpticalFlow_flow_calc.h,
//printf("%f * %f --> ", final_velocity_x_HLS,denominator_HLS);
double thresholded_final_velocity_x_HLS;
double thresholded_final_velocity_y_HLS;
/////////if (abs(denominator_HLS/pow(2,shift2x_HLS))<denominator_threshold){
/////////    thresholded_final_velocity_x_HLS = 0;
/////////    thresholded_final_velocity_y_HLS = 0;
/////////} else {
/////////    thresholded_final_velocity_x_HLS = final_velocity_x_HLS/denominator_HLS;
/////////    thresholded_final_velocity_y_HLS = final_velocity_y_HLS/denominator_HLS;
/////////}
if (denominator_HLS==0){
    thresholded_final_velocity_x_HLS = 0;
    thresholded_final_velocity_y_HLS = 0;
} else {
    thresholded_final_velocity_x_HLS = final_velocity_x_HLS/denominator_HLS;
    thresholded_final_velocity_y_HLS = final_velocity_y_HLS/denominator_HLS;
}
```

接著讀取出 algorithm 的結果（會存在 array 中），並透過對應到 shift_threshold 的 threshold 值來判斷是否要令 u=0、v=0：

```
if (abs(denom_algorithm[y][x])<denominator_threshold){
    output_algorithm[y][x].x = 0;
    output_algorithm[y][x].y = 0;
}
```

然後就可以比較 HLS 與 algorithm 的 output 結果：首先在圖片的 boundary 部分，在 software 中由於 kernel 超出圖片範圍，會直接令這個 module 在此位置的 output 值為 0；而在 HLS 的實作中，由於 interface 是 channel 的關係，我們實作上較不易判別而處理成 0，而是使用 duplicate 最近的 boundary 值的方式來實作，如此一來會使得 boundary 處的結果有較大的差異，但我們認為主要計算重點應在圖片偏中間處，因此我們就直接透過 if condition 過濾掉 boundary 的差異，只在螢幕上 print 出圖片中間有差異的部分。此外，由於軟體中是使用 float type 的變數，但在 HLS 中是用 ac_fix<> 的 data type，故在計算結果的精確度上會稍微有差異（因為 quantization error），算出來的值也有些許的不同，在這裡我們的比較方式為將兩者結果相減，並取絕對值後，當差異(error)小過我們所定的 threshold 值就算過關，例如我們在這裡定 threshold 值為 1，當兩者的差異大過 1 時，才會在螢幕上 print 出 error 相關資訊。

```
double tolerable_error_threshold = 1;
if ((x==0) && (y==0)) {
    printf("\n\nReport those pixels with error value > %f as following:\n", tolerable_error_threshold);
}
if (>x5 && x<1019 && y>5 && y<430) {
    if (abs(output_algorithm[y][x].x-thresholded_final_velocity_x_HLS) > tolerable_error_threshold) { // We can set "threshold=0.1" before OpticalFlow_gradient_weight_x.h, and set "thres
        printf("(%, %d), ", x, y);
        printf("u: (algorithm, HLS) = (%f, %f), error = %f\n", output_algorithm[y][x].x, thresholded_final_velocity_x_HLS, abs(output_algorithm[y][x].x-thresholded_final_velocity_x_HLS));
    }
    if (abs(output_algorithm[y][x].y-thresholded_final_velocity_y_HLS) > tolerable_error_threshold) { // We can set "threshold=0.1" before OpticalFlow_gradient_weight_y.h, and set "thres
        printf("(%, %d), ", x, y);
        printf("v: (algorithm, HLS) = (%f, %f), error = %f\n", output_algorithm[y][x].y, thresholded_final_velocity_y_HLS, abs(output_algorithm[y][x].y-thresholded_final_velocity_y_HLS));
    }
}
//if ((output_algorithm[y][x].x==0) && (output_algorithm[y][x].y==0) && (thresholded_final_velocity_x_HLS!=0) && (thresholded_final_velocity_y_HLS!=0)){
//    printf("denominator: (algorithm, HLS (after shift)) = (%.32f, %.32f)\n", denom_algorithm[y][x], denominator_HLS/pow(2,shift2x_HLS));
//}
```

接下來我們把 algorithm 及 HLS 的 u、v 結果運算出 magnitude= $\sqrt{u^2 + v^2}$ ，並將這些值寫至圖片檔中：

```

float u_algorithm = (double)(output_algorithm[y][x].x);
float v_algorithm = (double)(output_algorithm[y][x].y);
double magnitude_algorithm = sqrt(u_algorithm*u_algorithm + v_algorithm*v_algorithm);

float u_HLS = thresholded_final_velocity_x_HLS;
float v_HLS = thresholded_final_velocity_y_HLS;
double magnitude_HLS = sqrt(u_HLS*u_HLS + v_HLS*v_HLS);

float Absdiff_magnitude = abs(magnitude_algorithm-magnitude_HLS);
sumErr_magnitude += Absdiff_magnitude;

garray1[cnt] = (int)u_algorithm; // repurposing 'green' array to the original algorithmic output
garray2[cnt] = (int)v_algorithm;
garray3[cnt] = (int)magnitude_algorithm;

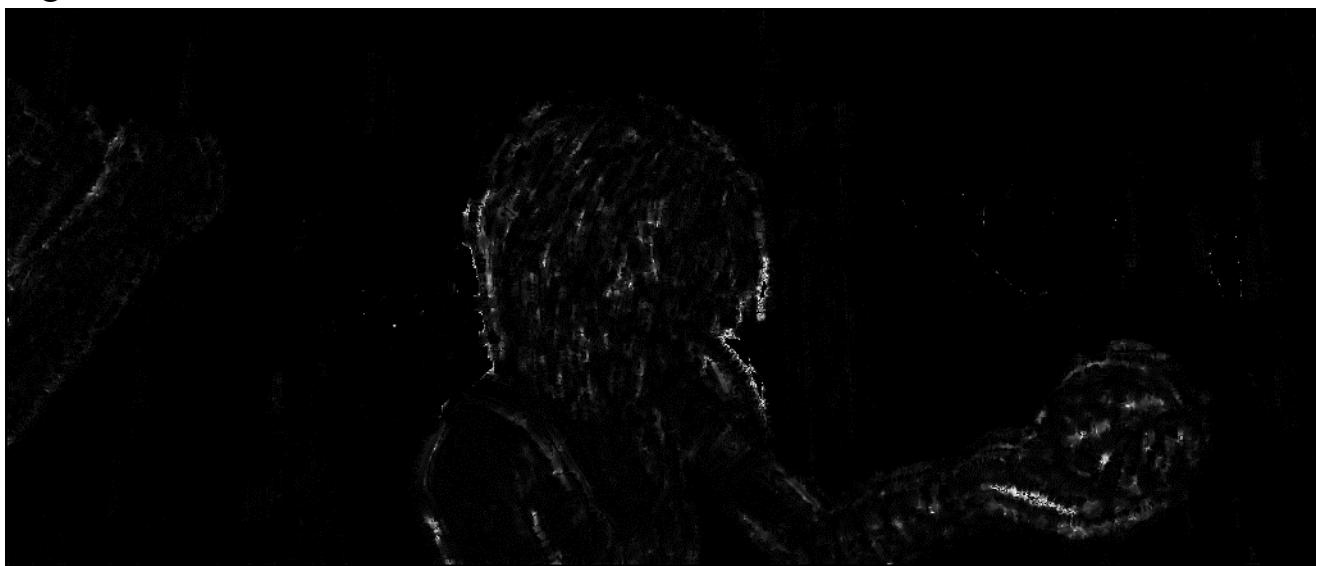
barray1[cnt] = (int)u_HLS; // repurposing 'blue' array to the original HLS output
barray2[cnt] = (int)v_HLS;
barray3[cnt] = (int)magnitude_HLS;

fprintf(file_pointer_frame1, "%x\n", rarray1[cnt]);
fprintf(file_pointer_frame2, "%x\n", rarray2[cnt]);
fprintf(file_pointer_frame3, "%x\n", rarray3[cnt]);
fprintf(file_pointer_frame4, "%x\n", rarray4[cnt]);
/////////fprintf(file_pointer_frame5, "%x\n", rarray5[cnt]);
fprintf(file_pointer_output_u_algorithm, "%x\n", (int)u_algorithm);
fprintf(file_pointer_output_v_algorithm, "%x\n", (int)v_algorithm);
fprintf(file_pointer_output_magnitude_algorithm, "%x\n", (int)magnitude_algorithm);
fprintf(file_pointer_output_u_HLS, "%x\n", (int)u_HLS);
fprintf(file_pointer_output_v_HLS, "%x\n", (int)v_HLS);
fprintf(file_pointer_output_magnitude_HLS, "%x\n", (int)magnitude_HLS);
/////////fprintf(file_pointer_channel_output_u_before_threshold_HLS, "%llx\n", (signed long long int)final_velocity_x_HLS);
/////////fprintf(file_pointer_channel_output_v_before_threshold_HLS, "%llx\n", (signed long long int)final_velocity_y_HLS);
/////////fprintf(file_pointer_channel_output_denominator_HLS, "%llx\n", (signed long long int)denominator_HLS);
fprintf(file_pointer_channel_output_u_before_threshold_HLS, "%x\n", (int)final_velocity_x_HLS);
fprintf(file_pointer_channel_output_v_before_threshold_HLS, "%x\n", (int)final_velocity_y_HLS);
fprintf(file_pointer_channel_output_denominator_HLS, "%x\n", (int)denominator_HLS);
/////////fprintf(file_pointer_channel_output_shift_HLS, "%x\n", (int)shift_HLS);

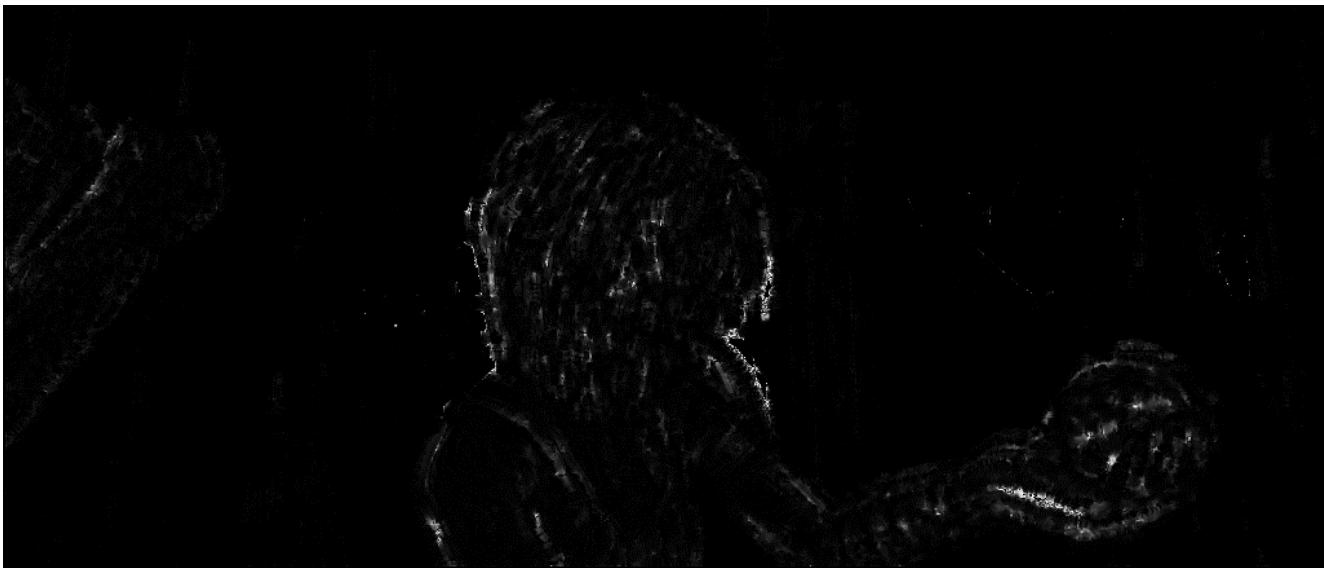
```

最終輸出圖片比較：

Algorithm :



HLS :



比對後可知兩者幾乎一模一樣！

Testbench 也將比較結果 print 在螢幕中：

```

19 Report those pixels with error value > 1.000000 as following:
20 (1018, 89), v: (algorithm, HLS) = (0.516886, -0.679200), error = 1.196086
21 (1018, 90), u: (algorithm, HLS) = (-1.168788, 0.006828), error = 1.175617
22 (1018, 90), v: (algorithm, HLS) = (1.404909, -0.757867), error = 2.162776
23 (1018, 91), u: (algorithm, HLS) = (-1.707085, 0.026652), error = 1.733737
24 (1018, 91), v: (algorithm, HLS) = (2.424565, -0.866157), error = 3.290722
25 (1018, 92), u: (algorithm, HLS) = (-1.227341, -0.045206), error = 1.182135
26 (1018, 92), v: (algorithm, HLS) = (1.383813, -0.795784), error = 2.179596
27 (343, 128), u: (algorithm, HLS) = (0.000000, -23.188396), error = 23.188396
28 (343, 128), v: (algorithm, HLS) = (0.000000, -25.026038), error = 25.026038
29 (344, 129), u: (algorithm, HLS) = (0.000000, -11.353325), error = 11.353325
30 (344, 129), v: (algorithm, HLS) = (0.000000, -2.397898), error = 2.397898
31 (345, 130), u: (algorithm, HLS) = (0.000000, -2.595953), error = 2.595953
32 (345, 130), v: (algorithm, HLS) = (8.000000, 6.262345), error = 1.737655
33 (346, 131), u: (algorithm, HLS) = (0.000000, -2.195232), error = 2.195232
34 (591, 139), v: (algorithm, HLS) = (-1976.888428, -1981.823439), error = 4.935012
35 (591, 140), v: (algorithm, HLS) = (-6546.304199, -6548.806204), error = 2.502005
36 (377, 152), u: (algorithm, HLS) = (564.058350, 562.777244), error = 1.281106
37 (582, 152), v: (algorithm, HLS) = (18235.078125, 18227.533512), error = 7.544613
38 (584, 152), u: (algorithm, HLS) = (-16395.356484, -16381.128599), error = 14.267885
39 (584, 152), v: (algorithm, HLS) = (8624.936523, 8620.340865), error = 4.595659
40 (378, 153), u: (algorithm, HLS) = (2287.164062, 2285.643862), error = 1.520201
41 (378, 153), v: (algorithm, HLS) = (3343.235352, 3341.449891), error = 1.785460
42 (379, 153), u: (algorithm, HLS) = (-187.500153, 190.988869), error = 3.485716
43 (379, 153), v: (algorithm, HLS) = (-158.316330, -160.572373), error = 2.256043
44 (582, 153), u: (algorithm, HLS) = (-15526.722656, -15532.224654), error = 5.501998
45 (583, 153), u: (algorithm, HLS) = (-2899.090576, -2895.994817), error = 3.095759
46 (379, 154), u: (algorithm, HLS) = (14616.552734, 14480.880402), error = 135.672332
47 (379, 154), v: (algorithm, HLS) = (9952.260742, 9873.882281), error = 78.378461
48 (580, 155), u: (algorithm, HLS) = (-2233.261230, -2318.767694), error = 85.506463
49 (580, 155), v: (algorithm, HLS) = (216672.656250, 224478.039353), error = 7805.383103
50 (579, 156), u: (algorithm, HLS) = (13603.999023, 13607.966121), error = 3.967098
51 (579, 156), v: (algorithm, HLS) = (-32241.501953, -32252.424470), error = 10.922517
52 (574, 163), u: (algorithm, HLS) = (0.000000, 250.867255), error = 250.867255
53 (574, 163), v: (algorithm, HLS) = (0.000000, -46.585631), error = 46.585631
54 (571, 167), u: (algorithm, HLS) = (0.000000, -53.933615), error = 53.933615
55 (571, 167), v: (algorithm, HLS) = (0.000000, -28.918402), error = 28.918402
56 (569, 168), u: (algorithm, HLS) = (0.000000, 105.904771), error = 105.904771
57 (569, 169), v: (algorithm, HLS) = (0.000000, 63.753189), error = 63.753189
58 (568, 170), u: (algorithm, HLS) = (0.000000, -17.668464), error = 17.668464
59 (568, 170), v: (algorithm, HLS) = (0.000000, 302.755111), error = 302.755111
60 (567, 171), u: (algorithm, HLS) = (0.000000, 6.762858), error = 6.762858
61 (567, 171), v: (algorithm, HLS) = (0.000000, 388.149182), error = 388.149182
62 (562, 176), v: (algorithm, HLS) = (10589.887695, 10592.813339), error = 2.925643
63 (373, 191), v: (algorithm, HLS) = (-20391.718750, -20407.185100), error = 15.466350
64 (373, 192), v: (algorithm, HLS) = (-19236.683594, -19300.476494), error = 63.792900
65 (587, 197), v: (algorithm, HLS) = (11825.548828, 11834.195316), error = 8.646488
66 (588, 197), v: (algorithm, HLS) = (8859.022461, 8861.760302), error = 2.737841
67 (588, 200), v: (algorithm, HLS) = (-2636.878418, -2638.669568), error = 1.791550
68 (591, 208), u: (algorithm, HLS) = (-9129.3554492, -9131.285033), error = 1.930601
69 (589, 209), v: (algorithm, HLS) = (18156.841797, 18553.199576), error = 396.357779
70 (590, 209), v: (algorithm, HLS) = (18947.361328, 18988.683765), error = 41.322437
71 (591, 209), u: (algorithm, HLS) = (-8037.385176, -8038.731810), error = 1.426634
72 (272, 213), u: (algorithm, HLS) = (-1.000017, 0.000000), error = 1.000017
73 (590, 214), u: (algorithm, HLS) = (278.492157, 280.451234), error = 1.959077
74 (590, 214), v: (algorithm, HLS) = (-130166.453125, -132974.771635), error = 2808.318510
75 (590, 214), v: (algorithm, HLS) = (-1085.974499, 1082.777297), error = 5.402572

```

在 $1024 \times 436 = 446464$ 個 pixel 中，只有不到 100 個 pixel 有超過 1 的 error (大部分是因為分母很小而導致的 quantization error) 。

其 Manhattan norm per pixel=0.063405，非常小，這表示我們的結果非常接近 algorithm！

5. HLS synthesis (using Catapult HLS tool)

(1) OpticalFlow_gradient_y.h :

因為 input data 是沿 x 軸輸入，因此我們需要使用 line buffer 來儲存所需要計算 gradient 的 data，在這個 module 當中，我們把 input data type 設定為 ccs_ioprot.ccs_in_wait_coupled，Configuration Port 則被設定為 [Direct Input]。因為 Kernel Size = 5，因此 line buffer 至少需要儲存 4 行圖片的資訊，也就是 $4 * \text{Width}$ 的 Depth 儲存資料，其中 Width = 436。

Resource: Input_frames:rsc

Resource Type: ccs_ioprot.ccs_in_wait_coupled

Input Delay

Library Delay: 0 ns
Inherited Delay: 0 ns

Resource: widthIn:rsc ...

Resource Type: [DirectInput]

Block Size: 0

Input Delay

Library Delay: 0 ns
Inherited Delay: 0 ns

因為 width = 436，但 Depth 與 Address 對應的關係會優先被合成為 2 的次方大小的 BRAM。此處 BRAM 被合成為 depth = 512。

line_buf3:rsc (512x16)
line_buf2:rsc (512x16)
line_buf1:rsc (512x64)
line_buf0:rsc (512x64)

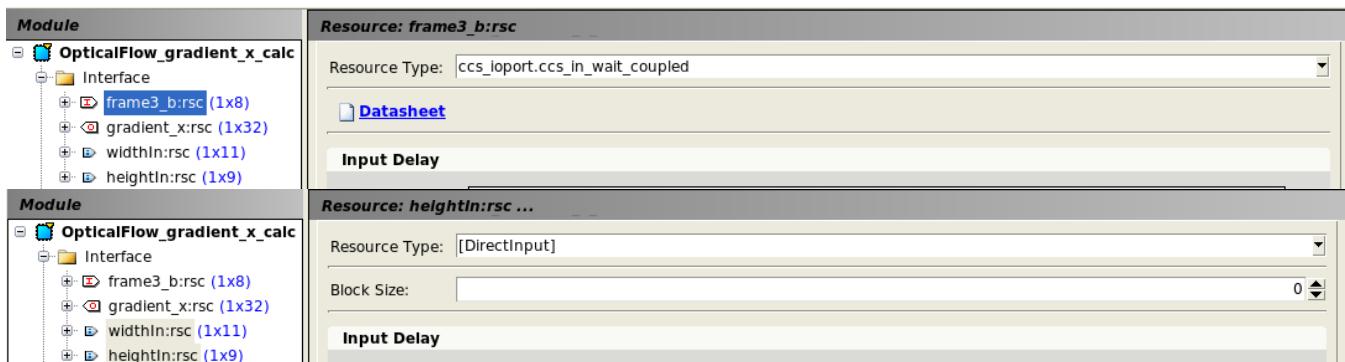
Resource Type: Xilinx_RAMS.BLOCK_SPRAM_RBW

Externalize
 Generate External Enable

(2) OpticalFlow_gradient_x.h :

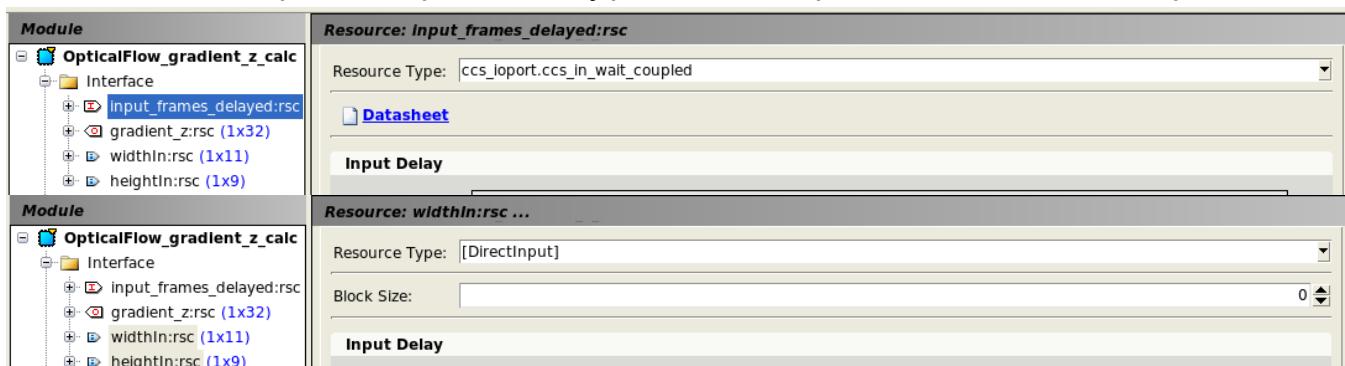
和 OpticalFlow_gradient_y.h 相比，由於 input data 是由 x 軸方向輸入，因此運算上較為單純，但為了協調 FIFO size(catapult 最高設定為 128)，

以及和 gradient_y 和 gradient_z 的 delay，因此 input data 是先藉由 gradient_y 的 line buffer 暫存，並在 output gradient_y 的時候與此 module 的 input data 同時輸出。此 module 的 kernel size = 5，只需要使用 buffer 儲存前 4 frames 的 data 即可運算出結果，總共只使用 4 個 buffer。設定與 gradient_y 相同，Configuration Port 設定為[Direct Input]，input data type 設定為 ccs_ioprot.ccs_in_wait_couple。



(3) OpticalFlow_gradient_z.h :

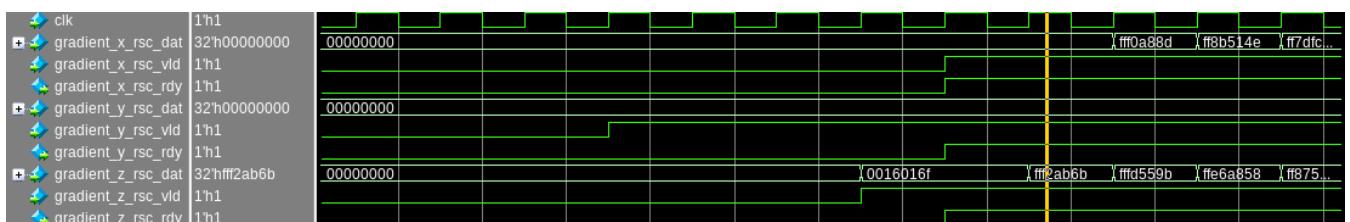
每當 input data 到 gradient_z module 時，得到 data 的當下即可進行運算，但與 gradient_x 為了協調 FIFO size(catapult 最高設定為 128)，以及和 gradient_y 和 gradient_x 的 delay，因此 input data 一樣是藉由 gradient_y 的 line buffer 暫存，並在 output gradient_y 的時候與此 module 的 input data 同時輸出。此 module 的 kernel size = 4，受限於 stream input 的 bit 數。設定與 gradient_y 相同，Configuration Port 為 [Direct Input]，input data type 為 ccs_ioprot.ccs_in_wait_couple。



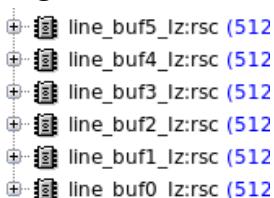
(4) OpticalFlow_gradient_weight_y.h :

與 OpticalFlow_gradient_y.h module 架構上完全相同，因為 input data 是沿 x 軸輸入，因此我們需要使用 line buffer 來儲存所需要計算 gradient

的 data，把 input data type 設定為 ccs_ioprot.ccs_in_wait_couple，Configuration Port 則被設定為[Direct Input]。因為 Kernel Size = 7，因此對於每種 input data 而言，line buffer 至少需要儲存 6 行圖片的資訊，也就是 $6 * \text{Width}$ 的 Depth 儲存資料，其中 Width = 436。另外此 module 需要使用到 gradient_x, gradient_y, gradient_z，總共使用到 18 個 line buffer。另外，由下方的 waveform 可知，為了協同 gradient_x, gradient_y, gradient_z，需要設定 gradient_y FIFO = 4, gradient_z FIFO = 1。

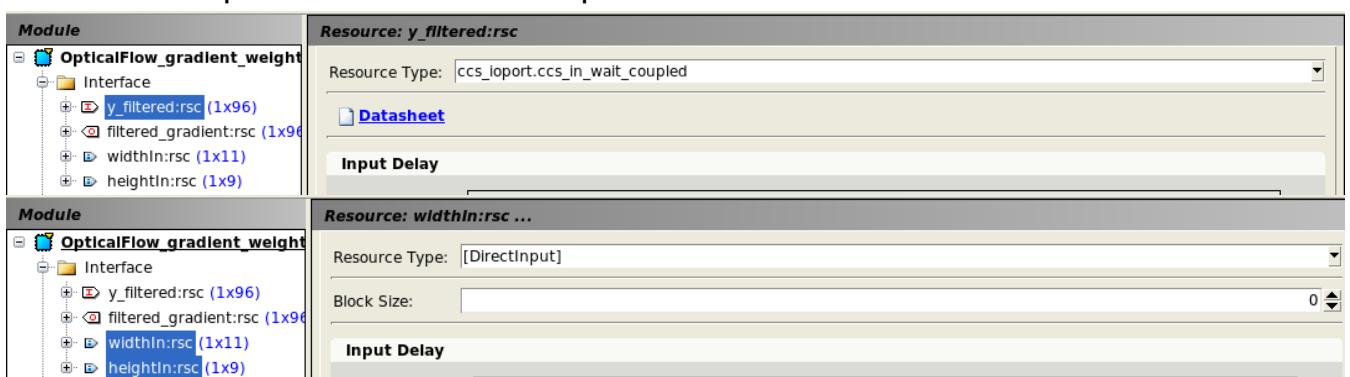


以 gradient_z 為例，需要 6 個 line buffer。



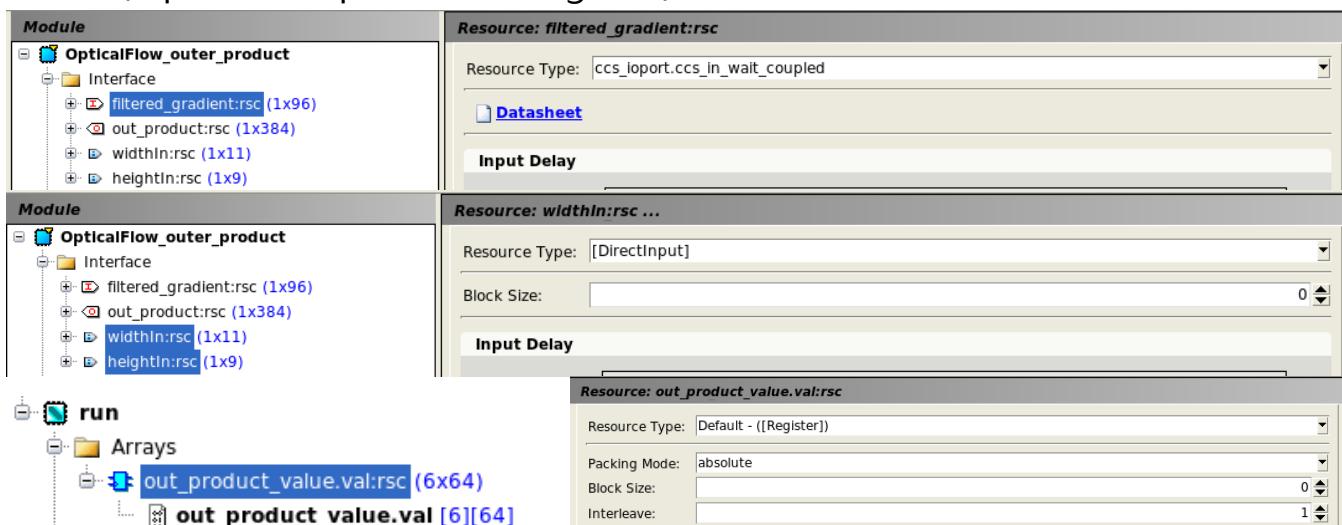
(5) OpticalFlow_gradient_weight_x.h :

與 gradient_x 相同，由於 input data 是由 x 軸方向輸入，因此運算上較為單純。此 module 的 kernel size = 7，只需要使用 buffer 儲存前 6 frames 的 data 即可運算出結果，總共只使用 6 個 buffer。設定與 gradient_x 相同，Configuration Port 設定為[Direct Input]，input data type 設定為 ccs_ioprot.ccs_in_wait_couple。



(6) OpticalFlow_outer_product.h :

Input data 是從 OpticalFlow_gradient_weight_x.h 的 output 而來，設定上與前述的 module 相同，Configuration Port 設定為[Direct Input]，input data type 設定為 ccs_ioprot.ccs_in_wait_couple。另外由於 output 為一 array，因此 Catapult 提供 Register 和 BRAM 的 interface 做 output 的儲存，在這邊我們設定為 register，可在後續的 input interface 一樣使用 register，可以把 II 降為 1 並節省使用的 resource (input 和 output 可共用 register)。



(7) OpticalFlow_tensor_weight_y.h :

Input data 是一 array，若我們使用 SPRAM 作為 input 的 interface，會無法將 II 降為 1(同一 cycle 同時寫入和讀出)，因此將其設定為 register，與此可對應到 outer_product 的 output，可共用節省 register 的資源。此 module 的 kernel size = 3，與前述計算 y 的 gradient module 相同，需要使用 2 個 line buffer。I/O 的設定與其他 module 相同。Configuration Port 設定為[Direct Input]，input data type 設定為 ccs_ioprot.ccs_in_wait_couple。

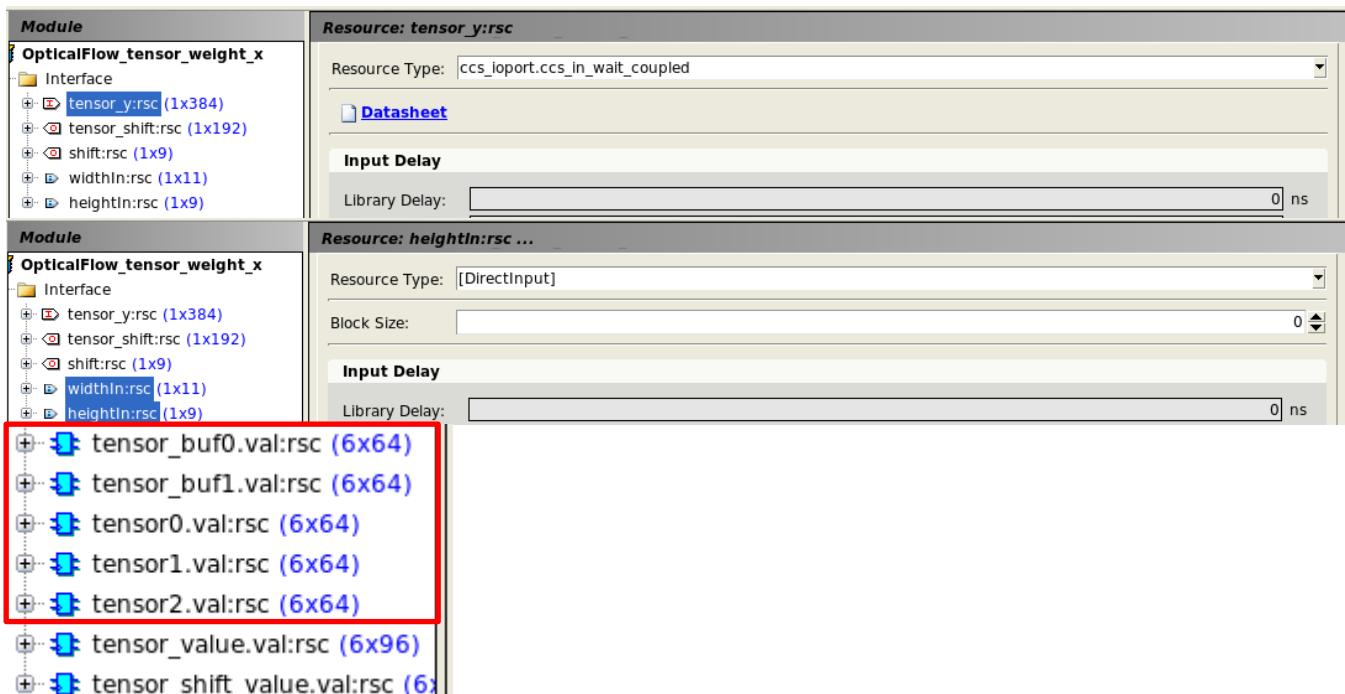




(8) OpticalFlow_tensor_weight_x.h :

Input data 為一 array，與 tensor_weight_y 相同，input interface 需要使用 register 才可使 II 降為 1，為了共用 register 將 tensor_weight_y 的 output 也設定為 register。I/O 的設定與其他 module 相同。

Configuration Port 設定為[Direct Input]，input data type 設定為 ccs_ioprot.ccs_in_wait_couple。



(9) OpticalFlow_flow_calc.h :

Input data 為前一級 module 的 output array，因此需要將 input interface 設定為 register 才能把 II 降至 1，然而因為我們總共需要 output 3 筆 data (1 筆 input 對應 3 筆 output)，如此導致我們的 II 最低只能降低至 3 (每 3 cycle input data, output stream 連續不間斷)。

6. Integrate RTL (after HLS synthesis) into FSIC

To integrate the RTL design, we should follow these steps :

- (1) Check the design of HLS (I/O)
- (2) Direct Input type channel should be set as AXI-lite interface (configuration)
- (3) Ac_channel is set as stream interface
- (4) Assign the external SPRAM for HLS design

因為我們使用到非常多 line buffer，因此總共會產生 24 個 SPRAM 的 interface，首先我們先 specify 各個訊號屬於何種 interface，再依序將訊號 assign 給各自的 module。

```

OpticalFlow_Top U_OpticalFlow(
    .clk          (axi_clk),
    .rst          (reg_rst),
    .arst_n      (axi_reset_n),

    .widthIn     (reg_widthIn), //I
    .heightIn    (reg_heightIn), //I
    .shift_threshold (reg_shift_threshold), //I

    .input_frames_rsc_dat (dat_in_rsc_dat),
    .input_frames_rsc_vld (ss_tvalid),
    .input_frames_rsc_rdy (ss_tready),
    .outputs_rsc_dat (dat_out_rsc_dat),
    .outputs_rsc_vld (sm_tvalid),
    .outputs_rsc_rdy (sm_tready),

    .line_buf3_rsc_clken (ram0_en),
    .line_buf3_rsc_q (ram0_q),
    .line_buf3_rsc_we (ram0_we),
    .line_buf3_rsc_d (ram0_d),
    .line_buf3_rsc_addr (ram0_addr),

    .line_buf2_rsc_clken (ram1_en),
    .line_buf2_rsc_q (ram1_q),
    .line_buf2_rsc_we (ram1_we),
    .line_buf2_rsc_d (ram1_d),
    .line_buf2_rsc_addr (ram1_addr),
);

```

AXI-lite interface

Stream interface

SRAM interface

SRAM interface

對於 AXI-lite interface，根據我們所設定的 address map，分別對應定義給不同的 register。

`reg_rst = 12' h0000_0000_0000`

`Width_In = 12' h0000_0000_0100`

Height_In = 12' h0000_0000_0200

shift_threshold = 12' h0000_0000_0300

AXI-lite interface

```

OpticalFlow_Top U_OpticalFlow(
    .clk                (axi_clk),
    .rst                (reg_rst),
    .arst_n             (axi_reset_n),

    .widthIn            (reg_widthIn), //I
    .heightIn           (reg_heightIn), //I
    .shift_threshold    (reg_shift_threshold), //I

    .input_frames_rsc_dat (dat_in_rsc_dat),
    .input_frames_rsc_vld (ss_tvalid),
    .input_frames_rsc_rdy (ss_tready),
    .outputs_rsc_dat   (dat_out_rsc_dat),
    .outputs_rsc_vld   (sm_tvalid),
    .outputs_rsc_rdy   (sm_tready),

    .line_buf3_rsc_clken (ram0_en),
    .line_buf3_rsc_q    (ram0_q),
    .line_buf3_rsc_we   (ram0_we),
    .line_buf3_rsc_d    (ram0_d),
    .line_buf3_rsc_adr  (ram0_adr),

    .line_buf2_rsc_clken (ram1_en),
    .line_buf2_rsc_q    (ram1_q),
    .line_buf2_rsc_we   (ram1_we),
    .line_buf2_rsc_d    (ram1_d),
    .line_buf2_rsc_adr  (ram1_adr),

    //write register
    always @(posedge axi_clk or negedge axi_reset_n) begin
        if ( !axi_reset_n ) begin
            reg_widthIn      <= 1024;
            reg_heightIn     <= 436;
            reg_shift_threshold <= 105;
            reg_rst          <= 0;
        end else begin
            if ( awvalid_in && wvalid_in ) begin
                if ( awaddr[11:2] == 10'h000 ) begin //offset 0
                    if ( wstrb[0] == 1) reg_rst      <= wdata[0];
                end else if ( awaddr[11:2] == 10'h001 ) begin //offset 1
                    if ( wstrb[0] == 1) reg_widthIn[7:0] <= wdata[7:0];
                    if ( wstrb[1] == 1) reg_widthIn[10:8] <= wdata[10:8];
                end else if ( awaddr[11:2] == 10'h002 ) begin //offset 2
                    if ( wstrb[0] == 1) reg_heightIn[7:0] <= wdata[7:0];
                    if ( wstrb[1] == 1) reg_heightIn[8]  <= wdata[8];
                end else if ( awaddr[11:2] == 10'h003 ) begin //offset 3
                    if ( wstrb[0] == 1) reg_shift_threshold[7:0] <= wdata[7:0];
                    if ( wstrb[1] == 1) reg_shift_threshold[8]  <= wdata[8];
                end
            end
        end
    end
    always @* begin
        if ( araddr[11:2] == 10'h000) rdata_tmp = reg_rst;
        else if ( araddr[11:2] == 10'h001) rdata_tmp = reg_widthIn;
        else if ( araddr[11:2] == 10'h002) rdata_tmp = reg_heightIn;
        else if ( araddr[11:2] == 10'h003) rdata_tmp = reg_shift_threshold;
        else
            rdata_tmp = 0;
    end

```

對於 Stream interface 的部分，我們可以只需要把外界的 stream interface 接到 OpticalFlow module 的 I/O 即可。其中的 sm_tlast，由於我們的 HLS design 中沒有輸出這個訊號，因此我們是在 user project 2 中加入 counter 數出當下以輸

出幾筆 data，當其值接近 widthIn*heightIn*3（因為一個 pixel 會對應輸出 3 筆 data）後表示整張圖片皆已輸出完畢，此時就會 assert sm_tlast 的訊號，讓 userDMA 得知此為最後一筆 output stream 訊號。

Stream interface

```
OpticalFlow_Top U_OpticalFlow(
    .clk          (axi_clk      ),
    .rst          (reg_rst      ),
    .arst_n       (axi_reset_n ),
    .widthIn      (reg_widthIn   ), //I
    .heightIn     (reg_heightIn  ), //I
    .shift_threshold (reg_shift_threshold), //I

    .input_frames_rsc_dat (dat_in_rsc_dat  ),
    .input_frames_rsc_vld (ss_tvalid      ),
    .input_frames_rsc_rdy (ss_tready      ),
    .outputs_rsc_dat    (dat_out_rsc_dat ),
    .outputs_rsc_vld   (sm_tvalid      ),
    .outputs_rsc_rdy   (sm_tready      ),

    .line_buf3_rsc_clken (ram0_en),
    .line_buf3_rsc_q    (ram0_q),
    .line_buf3_rsc_we   (ram0_we),
    .line_buf3_rsc_d    (ram0_d),
    .line_buf3_rsc_addr (ram0_addr),

    .line_buf2_rsc_clken (ram1_en),
    .line_buf2_rsc_q    (ram1_q),
    .line_buf2_rsc_we   (ram1_we),
    .line_buf2_rsc_d    (ram1_d),
    .line_buf2_rsc_addr (ram1_addr),
```

最後在 SRAM interface 的部分，我們根據 line buffer 的需要調整各種 SRAM word size。

SRAM interface

```
OpticalFlow_Top U_OpticalFlow(
    .clk          (axi_clk      ),
    .rst          (reg_rst      ),
    .arst_n       (axi_reset_n ),
    .widthIn      (reg_widthIn   ), //I
    .heightIn     (reg_heightIn  ), //I
    .shift_threshold (reg_shift_threshold), //I

    .input_frames_rsc_dat (dat_in_rsc_dat  ),
    .input_frames_rsc_vld (ss_tvalid      ),
    .input_frames_rsc_rdy (ss_tready      ),
    .outputs_rsc_dat    (dat_out_rsc_dat ),
    .outputs_rsc_vld   (sm_tvalid      ),
    .outputs_rsc_rdy   (sm_tready      ),

    .line_buf3_rsc_clken (ram0_en),
    .line_buf3_rsc_q    (ram0_q),
    .line_buf3_rsc_we   (ram0_we),
    .line_buf3_rsc_d    (ram0_d),
    .line_buf3_rsc_addr (ram0_addr),
```

SRAM interface

```
//SRAM
SPRAM #(.data_width(16), .addr_width(9), .depth(512)) U_SPRAM_0(
    .adr (ram0_addr),
    .d   (ram0_d),
    .en  (ram0_en),
    .we  (ram0_we),
    .clk (axi_clk),
    .q   (ram0_q)
);

SPRAM #(.data_width(64), .addr_width(9), .depth(512)) U_SPRAM_2(
    .adr (ram2_addr),
    .d   (ram2_d),
    .en  (ram2_en),
    .we  (ram2_we),
    .clk (axi_clk),
    .q   (ram2_q)
);

SPRAM #(.data_width(768), .addr_width(9), .depth(512)) U_SPRAM_22(
    .adr (ram22_addr),
    .d   (ram22_d),
    .en  (ram22_en),
    .we  (ram22_we),
    .clk (axi_clk),
    .q   (ram22_q)
);
```

7. FSIC simulation

在 FSIC simulation 的部分，整個實作的流程與 lab1 大致相符，我們將 test 的過程分為兩個 test，分別從 SoC side 及 FPGA side 發出 AXI-Lite 的請求，並透過 FPGA side 輸入 stream input 以及接收 stream output。主要實作檔案為 "/ASoC-Final_project-optical_flow/rtl/user/testbench/tb_fsic.v"，主要流程如下：

Test#1: AXI-Lite from SoC side

(1) 先做 reset：透過呼叫 `test0_INITIALIZATION()` 這個 task 即可：

```
//////// SoC & FPGA reset //////////
test0_INITIALIZATION();
```

其中 test0_initialization() 與 lab 1 時相同寫法。主要是將 SoC 及 FPGA side 皆做 reset，並透過 configuration write 寫入 IS (io-serdes) 使 SoC 與 FPGA side 的 Tx 和 Rx 皆開啟，才能開始與 FSIC 內部互相傳遞資料。

- (2) 由於經過 reset，故所選擇的 user project 編號回到 default 值 user_project_0，因此要再透過 soc_change_UP_configuration_write(32'h02); 來將 user_prj_sel 訊號值重新 program 成 2 (我們這次將我們的 HLS 合成後的 design 放置於 lab 4 的 user project area，此時 user_project_0 為 edge detect IP；user_project_1 為 FIR engine)。

```
///////// Enable user_project_2 (Optical flow engine) ///////
soc_change_UP_configuration_write(32'h02);
```

- (3) 仿照 lab2 的 EdgeDetect IP 的範例 testbench 的流程，先將 reset program 成 1，以將 HLS design 做 reset，並且讀取此位址的值以確認有被 program 為 1

```
/// modified from test001_up_soc_cfg() ///
//////// Set rst = 1 (address=0x00) //////
//soc_up_cfg_read(12'd0, 4'b1111);
soc_up_cfg_write(12'h0, 4'b0001, 32'd1);

//////// read-back and check //////
soc_UP_configuration_read_and_check(12'h0, 4'b0001, 32'd1);
```

- (4) 接著將 reset 的值 program 為 0，表示 reset 結束，並且讀取此位址的值以確認有被 program 回 0

```
//////// Set rst = 0 (address=0x00) //////
soc_up_cfg_write(12'h0, 4'b0001, 32'd0);

//////// read-back and check //////
soc_UP_configuration_read_and_check(12'h0, 4'b0001, 32'd0);
```

(5) 接著透過 AXI-Lite 輸入圖片的 width 和 height，以及拿來判別分母是否過小的比較基準值 shift_threshold。接著再讀回這些值並 check 是否成功 program。

```
///////// Program widthIn and heightIn ///////
soc_up_cfg_write(12'h04, 4'b1111, TST_FRAME_WIDTH);
soc_up_cfg_write(12'h08, 4'b1111, TST_FRAME_HEIGHT);
soc_up_cfg_write(12'h0c, 4'b1111, shift_threshold);

///////// read-back and check ///////
soc_UP_configuration_read_and_check(12'h04, 4'b1111, TST_FRAME_WIDTH);
soc_UP_configuration_read_and_check(12'h08, 4'b1111, TST_FRAME_HEIGHT);
soc_UP_configuration_read_and_check(12'h0c, 4'b1111, shift_threshold);
```

到目前為止就完成了 configuration 的步驟。

(6) 接著依照 lab1 的方式，透過 mailbox 來通知 FPGA side 可以開始執行 stream in 了。因此保留 lab1 的這段程式碼，以完成 SoC side to FPGA side 的 mailbox write。

```
///////// Use Mailbox to notify FPGA side to start input stream transfer /////// <-----
// modified from test005_aa_mailbox_soc_cfg() ///
soc_to_fpga_mailbox_write_addr_expect_value = SOC_to_FPGA_MailBox_Base; // which is the same as AA_MailBox_Reg_Offset in our setting
soc_to_fpga_mailbox_write_addr_BE_expect_value = 4'b1111; //
soc_to_fpga_mailbox_write_data_expect_value = 32'h5a5a_5a5a; //
soc_aa_cfg_write(AA_MailBox_Reg_Offset, soc_to_fpga_mailbox_write_addr_BE_expect_value, soc_to_fpga_mailbox_write_data_expect_value); //
@ (soc_to_fpga_mailbox_write_event); //wait for fpga to get the mail box written from soc. <- FPGA will be "notified" at this time -----
$display($time, "> got soc_to_fpga_mailbox_write_event");

//Address part
check_cnt = check_cnt + 1;
if (soc_to_fpga_mailbox_write_addr_expect_value !== soc_to_fpga_mailbox_write_addr_captured[27:0]) begin
    $display($time, "> [ERROR] soc_to_fpga_mailbox_write_addr_expect_value=%x, soc_to_fpga_mailbox_write_addr_captured[27:0]=%x",
    soc_to_fpga_mailbox_write_expect_value, soc_to_fpga_mailbox_write_addr_captured[27:0]);
    error_cnt = error_cnt + 1;
end
else
    $display($time, "> [PASS] soc_to_fpga_mailbox_write_addr_expect_value=%x, soc_to_fpga_mailbox_write_addr_captured[27:0]=%x",
    soc_to_fpga_mailbox_write_expect_value, soc_to_fpga_mailbox_write_addr_captured[27:0]);

//BE part
check_cnt = check_cnt + 1;
if (soc_to_fpga_mailbox_write_addr_BE_expect_value !== soc_to_fpga_mailbox_write_addr_captured[31:28]) begin
    $display($time, "> [ERROR] soc_to_fpga_mailbox_write_addr_BE_expect_value=%x, soc_to_fpga_mailbox_write_addr_captured[31:28]=%x",
    soc_to_fpga_mailbox_write_expect_value, soc_to_fpga_mailbox_write_addr_captured[31:28]);
    error_cnt = error_cnt + 1;
end
else
    $display($time, "> [PASS] soc_to_fpga_mailbox_write_addr_BE_expect_value=%x, soc_to_fpga_mailbox_write_addr_captured[31:28]=%x",
    soc_to_fpga_mailbox_write_expect_value, soc_to_fpga_mailbox_write_addr_captured[31:28]);

//data part
check_cnt = check_cnt + 1;
if (soc_to_fpga_mailbox_write_data_expect_value !== soc_to_fpga_mailbox_write_data_captured) begin
    $display($time, "> [ERROR] soc_to_fpga_mailbox_write_data_expect_value=%x, soc_to_fpga_mailbox_write_data_captured=%x", soc_to_fpga_mailbox_write_data_expect_value,
    soc_to_fpga_mailbox_write_data_captured);
    error_cnt = error_cnt + 1;
end
else
    $display($time, "> [PASS] soc_to_fpga_mailbox_write_data_expect_value=%x, soc_to_fpga_mailbox_write_data_captured=%x", soc_to_fpga_mailbox_write_data_expect_value,
    soc_to_fpga_mailbox_write_data_captured);
$display("-----");
```

(7) 接著是 FPGA side 要處理 stream in data 的流程：首先先將 input frame 的 data 以及 output frame 的 golden value 透過 file read 讀取進來。

```
//////// Optical flow input stream data from FPGA side ///////
/// modified from test002() ///
$readmemh("./pattern/frame1.hex", test_image_in_frame1_buf);
$readmemh("./pattern/frame2.hex", test_image_in_frame2_buf);
$readmemh("./pattern/frame3.hex", test_image_in_frame3_buf);
$readmemh("./pattern/frame4.hex", test_image_in_frame4_buf);
//$/readmemh("./pattern/frame5.hex", test_image_in_frame5_buf);
$readmemh("./pattern/channel_output_u_before_threshold_HLS.hex", test_image_golden_u_HLS_buf);
$readmemh("./pattern/channel_output_v_before_threshold_HLS.hex", test_image_golden_v_HLS_buf);
$readmemh("./pattern/channel_output_denominator_HLS.hex", test_image_golden_denominator_HLS_buf);
////$/readmemh("./pattern/channel_output_shift_HLS.hex", test_image_golden_shift_HLS_buf);
```

這些資訊是在 HLS simulation 時，在 OpticalFlow_tb.cpp 中將其寫入檔案中的：

```
fprintf(file_pointer_frame1, "%x\n", rarray1[cnt]);
fprintf(file_pointer_frame2, "%x\n", rarray2[cnt]);
fprintf(file_pointer_frame3, "%x\n", rarray3[cnt]);
fprintf(file_pointer_frame4, "%x\n", rarray4[cnt]);
//...
fprintf(file_pointer_channel_output_u_before_threshold_HLS, "%x\n", (int)final_velocity_x_HLS);
fprintf(file_pointer_channel_output_v_before_threshold_HLS, "%x\n", (int)final_velocity_y_HLS);
fprintf(file_pointer_channel_output_denominator_HLS, "%x\n", (int)denominator_HLS);
```

此處的 golden data 為 HLS 的輸出值，因為我們希望 HLS synthesis 後的 RTL 在 simulation 結果應該要和 HLS simulation 時的結果一致才對。

- (8) 透過 TST_TOTAL_LOOP_NUM 的值可設定要重複執行多少次 loop，而在此呼叫 test1_fpga_axis_req()這個 task，以輸入 stream input data。

```
for (test_loop_count=0; test_loop_count<TST_TOTAL_LOOP_NUM; test_loop_count=test_loop_count+1) begin
    $display("++++++ test loop No. %02d ++++++", test_loop_count);
    soc_to_fpga_axis_expect_count = 0;
    test1_fpga_axis_req(); //target to Axis Switch
```

- (9) 在 test1_fpga_axis_req()這個 task 中，最多會重複執行 total pixel 次，也就是圖片的 row*column 次，每次都會將前面讀取存起來的 4 個 input frame 的同一 pixel 的值 pack 起來，形成 32 bit 的 data，並附加上 start of frame (sof)、end of line (eol)等 flag，一起送至 fpga_axis_req_modified()這個 task 中，即可傳送出此筆 data 的 stream 給 FSIC。

```
for(vcnt=0; vcnt < TST_FRAME_HEIGHT; vcnt += 1) begin
    for(hcnt=0; hcnt < TST_FRAME_WIDTH; hcnt += 1) begin
        index = vcnt * TST_FRAME_WIDTH + hcnt;
        //input_data = {test_image_in_frame5_buf[index], test_image_in_frame4_buf[index], test_image_in_frame3_buf[index], test_image_in_frame2_buf[index],
        test_image_in_frame1_buf[index]};
        input_data = {test_image_in_frame4_buf[index], test_image_in_frame3_buf[index], test_image_in_frame2_buf[index], test_image_in_frame1_buf[index]};
        expect_data = {test_image_golden_denominator_HLS_buf[index], test_image_golden_u_HLS_buf[index], test_image_golden_v_HLS_buf[index]};
        sof = (vcnt==0 && hcnt==0);
        eol = (hcnt== TST_FRAME_WIDTH-1);
        //
```

```

`ifdef USER_PROJECT_SIDEBOARD_SUPPORT
    upsb = {eol,sof};
    fpga_axis_req_modified(input_data, TID_DN_UP, 0, upsb, vcnt, hcnt); //target to User Project
`else
    fpga_axis_req_modified(input_data, TID_DN_UP, 0, vcnt, hcnt);      //target to User Project
`endif

```

- (10)並將對應的 golden value 存至 soc_to_fpga_axis_expect_value 這個 array 中，以方便未來比對結果。

```

`ifdef USER_PROJECT_SIDEBOARD_SUPPORT
    if ((vcnt==TST_FRAME_HEIGHT-1) && (hcnt==TST_FRAME_WIDTH-1)) begin
        soc_to_fpga_axis_expect_value[soc_to_fpga_axis_expect_count] <= {upsb, 4'b0000, 4'b0000, 1'b1, expect_data};
    end
    else begin
        soc_to_fpga_axis_expect_value[soc_to_fpga_axis_expect_count] <= {upsb, 4'b0000, 4'b0000, 1'b0, expect_data};
    end
`else
    if ((vcnt==TST_FRAME_HEIGHT-1) && (hcnt==TST_FRAME_WIDTH-1)) begin
        soc_to_fpga_axis_expect_value[soc_to_fpga_axis_expect_count] <= {4'b0000, 4'b0000, 1'b1, expect_data};
    end
    else begin
        soc_to_fpga_axis_expect_value[soc_to_fpga_axis_expect_count] <= {4'b0000, 4'b0000, 1'b0, expect_data};
    end
`endif
    soc_to_fpga_axis_expect_count <= soc_to_fpga_axis_expect_count+1;

```

- (11)最後因為我們拿來測試的圖片為 1024*436，總共 446464 個 pixel，模擬時間會非常非常久，因此我們在 FSIC testbench 中多設定了一個變數 fpga_axis_test_length，可以在輸出這麼多筆 input 後就 break，跳出迴圈，只檢查到目前為止的 output 結果。

```

    if (vcnt*TST_FRAME_WIDTH+hcnt==fpga_axis_test_length) begin
        break;
    end
end
if (vcnt*TST_FRAME_WIDTH+hcnt==fpga_axis_test_length) begin
    break;
end

```

- (12)最後 return 回到 test1_initialization_from_SoC_side() 的 task，會檢查 capture 到的 data 是否與 golden value 一致。

```

$display($time, "> wait for soc_to_fpga_axis_event");
if (fpga_axis_test_length==TST_TOTAL_PIXEL_NUM) begin
|  @(soc_to_fpga_axis_event);
end
$display($time, "> soc_to_fpga_axis_expect_count = %d", soc_to_fpga_axis_expect_count);
$display($time, "> soc_to_fpga_axis_captured_count = %d", soc_to_fpga_axis_captured_count);

check_cnt = check_cnt + 1;
if ( soc_to_fpga_axis_expect_count != fpga_axis_test_length) begin
  $display($time, "> [ERROR] soc_to_fpga_axis_expect_count = %d, soc_to_fpga_axis_captured_count = %d", soc_to_fpga_axis_expect_count,
  soc_to_fpga_axis_captured_count);
  error_cnt = error_cnt + 1;
end
else
  $display($time, "> [PASS] soc_to_fpga_axis_expect_count = %d, soc_to_fpga_axis_captured_count = %d", soc_to_fpga_axis_expect_count,
  soc_to_fpga_axis_captured_count);

```

而在上述的 captured data 部分，則是也有修改如下圖，主要是由於我們的 design 的 1 筆 input 會對應到 3 筆 output，因此 capture count 需要每收到 3 筆再+1，並且需要把這 3 筆 output data pack 在一起，以方便之後值與 expected value 的比較：

```

initial begin      //get upstream soc_to_fpga_axis - for loop back test
  soc_to_fpga_axis_captured_count = 0;
  soc_to_fpga_axis_event_triggered = 0;
  output_stream_element_counter = 0;
  check_index = 0;
  while (1) begin
    //ifdef USE_EDGEDECTECT_IP // added by me
    @posedge fpga_coreclk;
    `ifdef USER_PROJECT_SIDEBOARD_SUPPORT
    if (fpga_is_as_tvalid == 1 && fpga_is_as_tid == TID_UP_UP && fpga_is_as_tuser == TUSER_AXIS) begin
      if (output_stream_element_counter==0) begin
        captured_denominator_buffer = fpga_is_as_tdata;
        output_stream_element_counter = output_stream_element_counter+1;
      end
      else if (output_stream_element_counter==1) begin
        captured_u_buffer = fpga_is_as_tdata;
        output_stream_element_counter = output_stream_element_counter+1;
      end
      else if (output_stream_element_counter==2) begin
        captured_v_buffer = fpga_is_as_tdata;
        //$/display($time, "> get soc_to_fpga_axis be : soc_to_fpga_axis_captured_count=%d, soc_to_fpga_axis_captured[%d]=%x, fpga_is_as_tupsb=%x, fpga_is_as_tstrb=%x, fpga_is_as_tkeep=%x",
        //fpga_is_as_tlast=%x, fpga_is_as_tdata=%x", soc_to_fpga_axis_captured_count, soc_to_fpga_axis_captured[soc_to_fpga_axis_captured_count], fpga_is_as_tupsb,
        //fpga_is_as_tstrb, fpga_is_as_tkeep, fpga_is_as_tlast, fpga_is_as_tdata);
        soc_to_fpga_axis_captured[soc_to_fpga_axis_captured_count] = {fpga_is_as_tupsb, fpga_is_as_tstrb, fpga_is_as_tkeep , fpga_is_as_tlast, captured_denominator_buffer, captured_u_buffer,
        captured_v_buffer}; //use block assignment
        //$/display($time, "> get soc_to_fpga_axis af : soc_to_fpga_axis_captured_count=%d, soc_to_fpga_axis_captured[%d]=%x, fpga_is_as_tupsb=%x, fpga_is_as_tstrb=%x, fpga_is_as_tkeep=%x,
        //fpga_is_as_tlast=%x, fpga_is_as_tdata=%x", soc_to_fpga_axis_captured_count, soc_to_fpga_axis_captured[soc_to_fpga_axis_captured_count], fpga_is_as_tupsb,
        //fpga_is_as_tstrb, fpga_is_as_tkeep , fpga_is_as_tlast, fpga_is_as_tdata);
        $display($time, "> get soc_to_fpga_axis : soc_to_fpga_axis_captured[%d]=%x, fpga_is_as_tupsb=%x, fpga_is_as_tstrb=%x, fpga_is_as_tkeep=%x,
        fpga_is_as_tlast=%x, captured_denominator_buffer=%x, captured_u_buffer=%x", soc_to_fpga_axis_captured_count, soc_to_fpga_axis_captured[soc_to_fpga_axis_captured_count],
        fpga_is_as_tupsb, fpga_is_as_tstrb, fpga_is_as_tkeep , fpga_is_as_tlast, captured_denominator_buffer, captured_u_buffer,
        captured_v_buffer);
        soc_to_fpga_axis_captured_count = soc_to_fpga_axis_captured_count+1;
        output_stream_element_counter = 0;
      end
      // Check the result with golden value (expected value) from HLS testbench
      check_cnt = check_cnt + 1;
      if (soc_to_fpga_axis_expect_value[check_index][95:0] != soc_to_fpga_axis_captured[check_index][95:0] ) begin
        $display($time, "> [ERROR] index=%d, soc_to_fpga_axis_expect_value[%d] = %x, soc_to_fpga_axis_captured[%d] = %x", check_index, check_index, soc_to_fpga_axis_expect_value
        [check_index], check_index, soc_to_fpga_axis_captured[check_index]);
        //##Finish;
        error_cnt = error_cnt + 1;
      end
    end
  end
end

```

啟用 Windows

Test#2: AXI-Lite from FPGA side

(1) 與 Test 1 的流程類似，同樣先做 reset：透過呼叫 test0_initialization() 這個 task 即可：

```

//////// SoC & FPGA reset //////////
test0_initialization();

```

(2) 與 Test 1 的流程類似，同樣透過

`soc_change_UP_configuration_write(32'h02);` 指令來將 user_prj_sel 訊號值重新 program 成 2

```
///////// Enable user_project_2 (Optical flow engine) ///////
soc_change_UP_configuration_write(32'h02);
```

(3) 先確認 AA (AXIS-AXIL) 的 internal register 在 reset 後的 default 值是否正確。如下圖所示，首先將 fpga_as_is_tready 這個訊號寫為 1，表示可以接收回傳的訊號，接著去 read AA internal register 的位址的值，並與 golden value (default 值的 golden value 為 32' h0) 比對，將結果 print 在螢幕上。

```
///////// Start to initialize FIR from FPGA side, which is modified from "task test003_fpga_to_soc_cfg_read()" & "task
test006_fpga_to_soc_cfg_write()" ///////
@(posedge fpga_coreclk);
fpga_as_is_tready <= 1;

/// step 1. check default value
$display($time, "> start checking SoC AA internal register default value (after reset), which should be 0");
cfg_read_data_expect_value = 32'h0;           //default value after reset = 0
soc_aa_cfg_read(AA_Internal_Reg_Offset, 4'b1111);

check_cnt = check_cnt + 1;
if (cfg_read_data_captured != cfg_read_data_expect_value) begin
    $display($time, "> [ERROR] cfg_read_data_expect_value=%x, cfg_read_data_captured=%x", cfg_read_data_expect_value,
    cfg_read_data_captured);
    error_cnt = error_cnt + 1;
end
else
    $display($time, "> [PASS] cfg_read_data_expect_value=%x, cfg_read_data_captured=%x", cfg_read_data_expect_value,
    cfg_read_data_captured);
$display("-----");
```

其中會用到 `soc_aa_cfg_read()` 這個 task，其寫法與 `soc_up_cfg_read()` 非常類似，只是寫入的位址不同（為 AA_BASE = 32'h3000_2000）：

```
task soc_aa_cfg_read;
    input [11:0] offset;          //4K range
    input [3:0] sel;

begin
    @(posedge soc_coreclk);
    wbs_addr <= AA_BASE;
    wbs_addr[11:2] <= offset[11:2]; //only provide DW address

    wbs_sel <= sel;
    wbs_cyc <= 1'b1;
    wbs_stb <= 1'b1;
    wbs_we <= 1'b0;

    @(posedge soc_coreclk);
    while(wbs_ack==0) begin
        @(posedge soc_coreclk);
    end
    $display($time, "> soc_aa_cfg_read : wbs_addr=%x, wbs_sel=%b", wbs_addr, wbs_sel);
    //#
    //add delay to make sure cfg_read_data_captured get the correct data
    @(soc_cfg_read_event);
    $display($time, "> soc_aa_cfg_read : got soc_cfg_read_event");
end
endtask
```

(4)與 Test 1 的流程類似，同樣先將 reset program 成 1，以將 HLS design 做 reset，並且讀取此位址的值以確認有被 program 為 1

```
/// step 2. FPGA issues FPGA-to-SoC configuration read/write request to SoC
//////// Set rst = 1 (address=0x00) //////////
//////fpga_axilite_read_req(FPGA_to_SOC_UP_BASE + 32'd0); // or {4'b0000,FPGA_to_SOC_UP_BASE}
//////(soc_to_fpga_axilite_read_cpl_event); //wait for FPGA get the read completion
//////while (soc_to_fpga_axilite_read_cpl_captured[2]==0) begin // which means "ap_idle_done_start[2]==0"
//////    fpga_axilite_read_req(FPGA_to_SOC_UP_BASE + 32'd0);
//////    @(soc_to_fpga_axilite_read_cpl_event); //wait for FPGA get the read completion
//////end
FPGA_to_SoC_configuration_write(28'h0, 32'd1);

//////// read-back and check //////////
FPGA_to_SoC_configuration_read_and_check(32'h0, 32'd1);
```

(5)接著將 reset 的值 program 為 0，表示 reset 結束，並且讀取此位址的值以確認有被 program 回 0

```
//////// Set rst = 0 (address=0x00) //////////
FPGA_to_SoC_configuration_write(28'h0, 32'd0);

//////// read-back and check //////////
FPGA_to_SoC_configuration_read_and_check(32'h0, 32'd0);
```

(6)接著透過 AXI-Lite 輸入圖片的 width 和 height，以及拿來判別分母是否過小的比較基準值 shift_threshold。接著再讀回這些值並 check 是否成功 program。

```
//////// Program widthIn and heightIn //////////
FPGA_to_SoC_configuration_write(28'h04, TST_FRAME_WIDTH);
FPGA_to_SoC_configuration_write(28'h08, TST_FRAME_HEIGHT);
FPGA_to_SoC_configuration_write(28'h0c, shift_threshold);

//////// read-back and check //////////
FPGA_to_SoC_configuration_read_and_check(32'h04, TST_FRAME_WIDTH);
FPGA_to_SoC_configuration_read_and_check(32'h08, TST_FRAME_HEIGHT);
FPGA_to_SoC_configuration_read_and_check(32'h0c, shift_threshold);
```

到目前為止就完成了 configuration 的步驟。

(7)接著由於與 Test 1 是在同一次 testbench 模擬下，因此 global array 與 global 變數會互相影響到，因此要將一些 register 的值先做 reset，再繼續執行 Test 2。

```

task test2_reset_some_register;
begin
    soc_to_fpga_axis_captured_count=0;
    check_index = 0;
    output_stream_element_counter = 0;
    for(vcnt=0; vcnt < TST_FRAME_HEIGHT; vcnt += 1) begin
        for(hcnt=0; hcnt < TST_FRAME_WIDTH; hcnt += 1) begin
            index = vcnt * TST_FRAME_WIDTH + hcnt;
            soc_to_fpga_axis_expect_value[index] = 0;
            soc_to_fpga_axis_captured[index] = 0;
        end
    end
    $display("Before starting transfer input data, reset the following register: soc_to_fpga_axis_captured_count, check_index,
    output_stream_element_counter, soc_to_fpga_axis_expect_value[(all_pixel)], soc_to_fpga_axis_captured[(all_pixel)]");
end
endtask

```

(8) Stream data from FPGA side 的部分則跟 Test 1 相同：

```

/// modified from test002() ///
$readmemh("./pattern/frame1.hex", test_image_in_frame1_buf);
$readmemh("./pattern/frame2.hex", test_image_in_frame2_buf);
$readmemh("./pattern/frame3.hex", test_image_in_frame3_buf);
$readmemh("./pattern/frame4.hex", test_image_in_frame4_buf);
//$/readmemh("./pattern/frame5.hex", test_image_in_frame5_buf);
$readmemh("./pattern/channel_output_u_before_threshold_HLS.hex", test_image_golden_u_HLS_buf);
$readmemh("./pattern/channel_output_v_before_threshold_HLS.hex", test_image_golden_v_HLS_buf);
$readmemh("./pattern/channel_output_denominator_HLS.hex", test_image_golden_denominator_HLS_buf);
////$/readmemh("./pattern/channel_output_shift_HLS.hex", test_image_golden_shift_HLS_buf);

for (test_loop_count=0;test_loop_count<TST_TOTAL_LOOP_NUM;test_loop_count=test_loop_count+1) begin
    $display("++++++ test loop No. %02d ++++++", test_loop_count);
    soc_to_fpga_axis_expect_count = 0;
    test1_fpga_axis_req();           //target to Axis Switch
    $display($time, "> wait for soc_to_fpga_axis_event");
    if (fpga_axis_test_length==TST_TOTAL_PIXEL_NUM) begin
        @(soc_to_fpga_axis_event);
    end
    $display($time, "> soc_to_fpga_axis_expect_count = %d", soc_to_fpga_axis_expect_count);
    $display($time, "> soc_to_fpga_axis_captured_count = %d", soc_to_fpga_axis_captured_count);

    check_cnt = check_cnt + 1;
    if ( soc_to_fpga_axis_expect_count != fpga_axis_test_length) begin
        $display($time, "> [ERROR] soc_to_fpga_axis_expect_count = %d, soc_to_fpga_axis_captured_count = %d",
        soc_to_fpga_axis_expect_count, soc_to_fpga_axis_captured_count);
        error_cnt = error_cnt + 1;
    end
    else
        $display($time, "> [PASS] soc_to_fpga_axis_expect_count = %d, soc_to_fpga_axis_captured_count = %d",
        soc_to_fpga_axis_expect_count, soc_to_fpga_axis_captured_count);

```

最終模擬結果 print 在螢幕上的資訊非常多，底下只截出 summary 的部分：

```

s_as_tkeep=0 , fpga_is_as_tlast=0, captured_denominator_buffer=d7a45f8, captured_u_buffer=d0be8939, captured_v_buffer=e6333fc9
1438895=> [PASS] index= 3821, soc_to_fpga_axis_expect_value[ 3821] = 000067a245f8d1be8939e6333fc9, soc_to_fpga_axis_captured[ 3821] = 000067a245f8d1be8939e6333fc9
s_as_tkeep=0 , fpga_is_as_tlast=0, captured_denominator_buffer=d7a45f8, captured_u_buffer=d0be8939e6333fc9, soc_to_fpga_axis_expect_value[ 3822] = 000064a073f4ccdae50de466688b, fpga_is_as_tupsb=0, fpga_is_as_tstrb=0, fpga_i
s_as_tkeep=0 , fpga_is_as_tlast=0, captured_denominator_buffer=d7a45f8, captured_u_buffer=d0be8939e6333fc9, soc_to_fpga_axis_expect_value[ 3822] = 000064a073f4ccdae50de466688b, fpga_is_as_tupsb=0, fpga_is_as_tstrb=0, fpga_i
1439205=> get soc_to_fpga_axis : soc_to_fpga_axis_captured_count= 3823, soc_to_fpga_axis_expect_value[ 3823] = 000064a073f4ccdae50de466688b, fpga_is_as_tupsb=0, fpga_is_as_tstrb=0, fpga_i
1439205=> get soc_to_fpga_axis : soc_to_fpga_axis_captured_count= 3823, soc_to_fpga_axis_expect_value[ 3823] = 000064a073f4ccdae50de466688b, fpga_is_as_tupsb=0, fpga_is_as_tstrb=0, fpga_i
1439205=> [PASS] index= 3823, soc_to_fpga_axis_expect_value[ 3823] = 000064a073f4ccdae50de466688b, fpga_is_as_tupsb=0, fpga_is_as_tstrb=0, fpga_i
1439205=> get soc_to_fpga_axis : soc_to_fpga_axis_captured_count= 3824, soc_to_fpga_axis_expect_value[ 3824] = 000064a073f4ccdae50de466688b, fpga_is_as_tupsb=0, fpga_is_as_tstrb=0, fpga_i
1439205=> [PASS] index= 3824, soc_to_fpga_axis_expect_value[ 3824] = 000064a073f4ccdae50de466688b, fpga_is_as_tupsb=0, fpga_is_as_tstrb=0, fpga_i
1439205=> get soc_to_fpga_axis : soc_to_fpga_axis_captured_count= 3825, soc_to_fpga_axis_expect_value[ 3825] = 000064a073f4ccdae50de466688b, fpga_is_as_tupsb=0, fpga_is_as_tstrb=0, fpga_i
1439205=> [PASS] index= 3825, soc_to_fpga_axis_expect_value[ 3825] = 000064a073f4ccdae50de466688b, fpga_is_as_tupsb=0, fpga_is_as_tstrb=0, fpga_i
1439205=> get soc_to_fpga_axis : soc_to_fpga_axis_captured_count= 3826, soc_to_fpga_axis_expect_value[ 3826] = 000064a073f4ccdae50de466688b, fpga_is_as_tupsb=0, fpga_is_as_tstrb=0, fpga_i
1439205=> [PASS] index= 3826, soc_to_fpga_axis_expect_value[ 3826] = 000064a073f4ccdae50de466688b, fpga_is_as_tupsb=0, fpga_is_as_tstrb=0, fpga_i
1439205=> get soc_to_fpga_axis : soc_to_fpga_axis_captured_count= 3827, soc_to_fpga_axis_expect_value[ 3827] = 000064a073f4ccdae50de466688b, fpga_is_as_tupsb=0, fpga_is_as_tstrb=0, fpga_i
1439205=> [PASS] index= 3827, soc_to_fpga_axis_expect_value[ 3827] = 000064a073f4ccdae50de466688b, fpga_is_as_tupsb=0, fpga_is_as_tstrb=0, fpga_i
-----finish the data input(AXI-Stream ss) & data output(AXI-Stream sm) with checking-----
----- [Success] Congratulations! Pass test 1 ! ! !-----
```

由此可知 Test#1 pass，並無 error！並且其開始執行 Test 2。

```

s_as_tkeep=0 , fpga_is_as_tlast=0, captured_denominator_buffer=b7a45f8, captured_u_buffer=d0be8939, captured_v_buffer=e6333fc9
2901445=> [PASS] index= 3821, soc_to_fpga_axis_expect_value[ 3821] = 000067a245f8d1be8939e6333fc9, soc_to_fpga_axis_captured[ 3821] = 000067a245f8d1be8939e6333fc9
s_as_tkeep=0 , fpga_is_as_tlast=0, captured_denominator_buffer=d7a45f8, captured_u_buffer=d0be8939e6333fc9, soc_to_fpga_axis_expect_value[ 3822] = 000064a073f4ccdae50de466688b, fpga_is_as_tupsb=0, fpga_is_as_tstrb=0, fpga_i
2901605=> get soc_to_fpga_axis : soc_to_fpga_axis_captured_count= 3822, soc_to_fpga_axis_expect_value[ 3822] = 000064a073f4ccdae50de466688b, fpga_is_as_tupsb=0, fpga_is_as_tstrb=0, fpga_i
2901605=> [PASS] index= 3822, soc_to_fpga_axis_expect_value[ 3822] = 000064a073f4ccdae50de466688b, fpga_is_as_tupsb=0, fpga_is_as_tstrb=0, fpga_i
2901605=> get soc_to_fpga_axis : soc_to_fpga_axis_captured_count= 3823, soc_to_fpga_axis_expect_value[ 3823] = 000064a073f4ccdae50de466688b, fpga_is_as_tupsb=0, fpga_is_as_tstrb=0, fpga_i
2901765=> [PASS] index= 3823, soc_to_fpga_axis_expect_value[ 3823] = 000064a073f4ccdae50de466688b, fpga_is_as_tupsb=0, fpga_is_as_tstrb=0, fpga_i
2901765=> get soc_to_fpga_axis : soc_to_fpga_axis_captured_count= 3824, soc_to_fpga_axis_expect_value[ 3824] = 000064a073f4ccdae50de466688b, fpga_is_as_tupsb=0, fpga_is_as_tstrb=0, fpga_i
2901925=> [PASS] index= 3824, soc_to_fpga_axis_expect_value[ 3824] = 000064a073f4ccdae50de466688b, fpga_is_as_tupsb=0, fpga_is_as_tstrb=0, fpga_i
2901925=> get soc_to_fpga_axis : soc_to_fpga_axis_captured_count= 3825, soc_to_fpga_axis_expect_value[ 3825] = 000064a073f4ccdae50de466688b, fpga_is_as_tupsb=0, fpga_is_as_tstrb=0, fpga_i
2902085=> [PASS] index= 3825, soc_to_fpga_axis_expect_value[ 3825] = 000064a073f4ccdae50de466688b, fpga_is_as_tupsb=0, fpga_is_as_tstrb=0, fpga_i
2902085=> get soc_to_fpga_axis : soc_to_fpga_axis_captured_count= 3826, soc_to_fpga_axis_expect_value[ 3826] = 000064a073f4ccdae50de466688b, fpga_is_as_tupsb=0, fpga_is_as_tstrb=0, fpga_i
2902085=> [PASS] index= 3826, soc_to_fpga_axis_expect_value[ 3826] = 000064a073f4ccdae50de466688b, fpga_is_as_tupsb=0, fpga_is_as_tstrb=0, fpga_i
-----Finish the data input(AXI-Stream ss) & data output(AXI-Stream sm) with checking-----
----- [Success] Congratulations! Pass test 2 ! ! !-----
```

```

2902405=> get soc_to_fpga_axis : soc_to_fpga_axis_captured_count= 3827, soc_to_fpga_axis_expect_value[ 3827] = 000067a245f8d1be8939e6333fc9, soc_to_fpga_axis_captured[ 3827] = 000067a245f8d1be8939e6333fc9
2902405=> [PASS] index= 3827, soc_to_fpga_axis_expect_value[ 3827] = 000067a245f8d1be8939e6333fc9, soc_to_fpga_axis_captured[ 3827] = 000067a245f8d1be8939e6333fc9
2902565=> get soc_to_fpga_axis : soc_to_fpga_axis_captured_count= 3828, soc_to_fpga_axis_expect_value[ 3828] = 000064a5a7b13da7e89ad2b10ff2, fpga_is_as_tupsb=0, fpga_is_as_tstrb=0, fpga_i
2902565=> [PASS] index= 3828, soc_to_fpga_axis_expect_value[ 3828] = 000064a5a7b13da7e89ad2b10ff2, fpga_is_as_tupsb=0, fpga_is_as_tstrb=0, fpga_i
2902725=> Final result [PASS], check_cnt = 7673, error_cnt = 0000
=====
```

由此部分可知 Test#2 成功，並無 error！且最終統計結果為總共執行了 7673 次的與 golden value 的比對，而錯誤數目為 0 次！

8. FSIC-FPGA simulation

此部分的整個實作流程與 lab4 大致相符，主要實作檔案為 "/ASoC-Final_project-optical_flow/vivado/fsic_tb.v" 以及 "/ASoC-Final_project-optical_flow/vivado/fsic_tb.sv"

optical_flow/vivado/vvd_caravel_fpga_fsic_sim.tcl" · 和"/ASoC-Final_project-optical_flow/vivado/vitis_prj/hls_userdma/userdma.cpp" · 主要流程如下：

(1) 關於 userDMA 的部分：我們是從 lab4 的為基礎進行修改，當時已解決跳出迴圈的 BUF_LEN 更改為 input assign 的 in_s2m_len 的 issue，但由於應用情境與 EdgeDetect IP 不同，而註解掉關於底下的這幾行：

```
////////// Modified --> uncomment in final project ///////////
if((width_count==in_Img_width-1)&&(in_val.user(3,3)!=1))
    s2m_err=1;

if(width_count==in_Img_width-1)
    width_count = 0;
else
    width_count++;
////////// Modified --> uncomment in final project ///////////
if(Img_width_count == in_Img_width-1){
    out_val.upsb(3,3) = 1;
    Img_width_count=0;
}else{
    Img_width_count++;
}
////////// Modified --> uncomment in final project ///////////
```

由於我們的應用與 EdgeDetect IP 很類似，實作上也滿類似的，因此我將這幾行程式碼去掉註解。

(2) Add opticalFlow-related files into user_project_2 in FSIC

我們按照 lab4 時的做法，將相關檔案放到 user project 2 area，位於 /vivado/vvd_srcs/caravel_soc/rtl/user/user_subsys/user_prj/user_prj2/rtl/，並在執行 run_vivado_fsic_sim 與 run_vivado_fsic 時分別將這些檔案 include 到 vvd_caravel_fpga_fsic_sim.tcl 與 vvd_caravel_fpga_fsic.tcl 中。

(3) FSIC FPGA simulation in Vivado — fsic_tb.sv

仿照 lab4-1 的 EdgeDetect 的 userDMA 寫法，我們將其 modify 如下：

1. 寫入 stream_in data 至 memory 中 (以 slave_agent3 的 memory (mem_model) 來儲存) :

```

task SocUp2DmaPath;
begin
    $display($time, "-> Starting SocUp2DmaPath() test...");
    $display($time, "-> =====");

    $readmemh("../.../.../test_pattern/frame1.hex", updma_img_frame1);
    $readmemh("../.../.../test_pattern/frame2.hex", updma_img_frame2);
    $readmemh("../.../.../test_pattern/frame3.hex", updma_img_frame3);
    $readmemh("../.../.../test_pattern/frame4.hex", updma_img_frame4);

    fd = $fopen("../.../.../updma_input_data.log", "w");
    for (index = 0; index < TST_FRAME_WIDTH*TST_FRAME_HEIGHT; index +=1) begin
        updma_data_stream_in = updma_img_frame1[index];
        updma_data_stream_in |= updma_img_frame2[index] << 8;
        updma_data_stream_in |= updma_img_frame3[index] << 16;
        updma_data_stream_in |= updma_img_frame4[index] << 24;
        slave_agent3.mem_model.backdoor_memory_write_4byte(addr+i+index*4,updma_data_stream_in,4'b1111);
        updma_data_stream_in = 0;
        $fdisplay(fd, "%08h", slave_agent3.mem_model.backdoor_memory_read_4byte(addr+i+index*4));
    end
    $fclose(fd);

```

首先先讀取與 FSIC simulation 相同的 frames data · 接著我們透過 for loop 的方式 · 將 stream_in data (即程式碼中的 updma_data) 寫入 slave_agent3.mem_model 的相對應位址 addri+index*4 · 其中 index 要乘以 4 是因為我們的每筆 input data 是 32-bit · 因此需要 4 個 address 的大小才夠儲存 · 故下一筆 data 需要隔 4 個 address 再存 · 另外為了方便我們模擬結束後查看 input data · 因此將這些值也寫到 updma_input_data.log 這個 file 中 ·

```

$readmemh("../.../.../test_pattern/channel_output_denominator_HLS.hex", updma_img_golden_denominator_HLS);
$readmemh("../.../.../test_pattern/channel_output_u_before_threshold_HLS.hex", updma_img_golden_u_HLS);
$readmemh("../.../.../test_pattern/channel_output_v_before_threshold_HLS.hex", updma_img_golden_v_HLS);

fd = $fopen("../.../.../updma_output_gold.log", "w");
for (index = 0; index < TST_FRAME_WIDTH*TST_FRAME_HEIGHT; index +=1) begin
    for (index_inner = 0; index_inner < 3; index_inner +=1) begin
        if (index_inner==0) begin
            updma_data_golden_out = updma_img_golden_denominator_HLS[index];
        end
        else if (index_inner==1) begin
            updma_data_golden_out = updma_img_golden_u_HLS[index];
        end
        else begin
            updma_data_golden_out = updma_img_golden_v_HLS[index];
        end
        //slave_agent2.mem_model.backdoor_memory_write_4byte(addr+index*3+index_inner,updma_data_golden_out,4'b1111);
        //updma_data_golden_out = 0;
        //$fdisplay(fd, "%08h", slave_agent2.mem_model.backdoor_memory_read_4byte(addr+index*3+index_inner));
        $fdisplay(fd, "%08h", updma_data_golden_out);
    end
end
$fclose(fd);

```

在此我們也去讀取 output golden data · 並寫至 file 中 · 形成 golden file · 以方便之後的結果比對 ·

2. Configure user_DMA : 在透過 ap_start 訊號使 user_DMA 開始運作之前，需要先對其做前置作業(configuration)。在此我們仿照 lab4-1 及 lab4-3 的作法，但 output buffer length 要設定為 input length 的 3 倍，相關程式碼及對應的用途如下，其中參考了 UserDMA IRS.docx 中對於各 register 用途的敘述：

- Configure offset 0x20、data=0x0 : s2m exit clear

```
//Setup userdma
$display($time, "-> FpgaLocal_Write: PL_UPDMA, s2m exit clear...");
offset = 32'h0000_0020;
data = 32'h0000_0000;
axil_cycles_gen(WriteCyc, PL_UPDMA, offset, data, 1);
//#20us
axil_cycles_gen(ReadCyc, PL_UPDMA, offset, data, 1);
//#20us
if(data == 32'h0000_0000) begin
    $display($time, "-> Fpga2Soc_Write PL_UPDMA offset %h = %h, PASS", offset, data);
end else begin
    $display($time, "-> Fpga2Soc_Write PL_UPDMA offset %h = %h, FAIL", offset, data);
    ->> error_event;
end
```

此 offset 對應到下圖所述的 register：

s2m Buffer transfer done status clear register ..	0x20 ..	bit 0 – clear s2m buffer transfer done status (Read/Write) .. Set 1 to clear s2m buffer done status/s2m error status and reset internal state, then set 0 if finish to clear buffer done status .. Note: Before set this register to 1 to clear s2m buffer transfer done status/s2m error status, Clear status control register must set to 1. After buffer transfer done status is clear, this register needs to be cleared for next operation. ..
---	---------	---

我們將此 register 的值 program 成 0，代表已將 s2m buffer done 的 register 狀態(status)清除(clear)完成。

- Configure offset 0x30、data=0x0 : s2m disable to clear

```
$display($time, "-> FpgaLocal_Write: PL_UPDMA, s2m disable to clear...");
offset = 32'h0000_0030;
data = 32'h0000_0000;
axil_cycles_gen(WriteCyc, PL_UPDMA, offset, data, 1);
//#20us
axil_cycles_gen(ReadCyc, PL_UPDMA, offset, data, 1);
//#20us
if(data == 32'h0000_0000) begin
    $display($time, "-> Fpga2Soc_Write PL_UPDMA offset %h = %h, PASS", offset, data);
end else begin
    $display($time, "-> Fpga2Soc_Write PL_UPDMA offset %h = %h, FAIL", offset, data);
    ->> error_event;
end
```

此 offset 對應到下圖所述的 register：

s2m Clear Status Control register	0x30	bit 0 –Enable to clear s2m buffer transfer done status (Read/Write)
-----------------------------------	------	---

我們將此 register 的值 program 成 0，表示要 disable 將 s2m buffer transfer 完成狀態的 register clear 的動作。

➤ Configure offset 0x78、data=0x0 : m2s exit clear

```
$display($time, "> FpgaLocal_Write: PL_UPDMA, m2s exit clear...");  
offset = 32'h0000_0078;  
data = 32'h0000_0000;  
axil_cycles_gen(WriteCyc, PL_UPDMA, offset, data, 1);  
//#20us  
axil_cycles_gen(ReadCyc, PL_UPDMA, offset, data, 1);  
//#20us  
if(data == 32'h0000_0000) begin  
    $display($time, "> Fpga2Soc_Write PL_UPDMA offset %h = %h, PASS", offset, data);  
end else begin  
    $display($time, "> Fpga2Soc_Write PL_UPDMA offset %h = %h, FAIL", offset, data);  
    -> error_event;  
end
```

此 offset 對應到下圖所述的 register：

m2s Buffer transfer done status clear register	0x78	bit 0 – clear m2s buffer transfer done status (Read/Write) Set 1 to clear m2s buffer done status/s2m error status and reset internal state, then set 0 if finish to clear buffer done status. Note: Before set this register to 1 to clear m2s buffer transfer done status/m2s error status, Clear status control register must set to 1. After buffer transfer done status is clear, this register needs to be cleared for next operation.
--	------	---

我們將此 register 的值 program 成 0，代表已將 m2s buffer done 的 register 狀態(status)清除(clear)完成。

➤ Configure offset 0x88、data=0x0 : m2s disable to clear

```
$display($time, "> FpgaLocal_Write: PL_UPDMA, m2s disable to clear...");  
offset = 32'h0000_0088;  
data = 32'h0000_0000;  
axil_cycles_gen(WriteCyc, PL_UPDMA, offset, data, 1);  
//#20us  
axil_cycles_gen(ReadCyc, PL_UPDMA, offset, data, 1);  
//#20us  
if(data == 32'h0000_0000) begin  
    $display($time, "> Fpga2Soc_Write PL_UPDMA offset %h = %h, PASS", offset, data);  
end else begin  
    $display($time, "> Fpga2Soc_Write PL_UPDMA offset %h = %h, FAIL", offset, data);  
    -> error_event;  
end
```

此 offset 對應到下圖所述的 register：

m2s Clear Status Control register	0x88	bit 0 –Enable to clear m2s buffer transfer done status (Read/Write)
-----------------------------------	------	---

我們將此 register 的值 program 成 0，表示要 disable 將 m2s buffer transfer 完成狀態的 register clear 的動作。

- Configure offset 0x28、data=output data length=3×input data length，在此與 FSIC simulation 時類似，為了避免 simulation 跑超級久，而設定為 3×test_length，理想情況下 test_length=width×height : s2m set buffer length

```
$display($time, "-> FpgaLocal_Write: PL_UPDMA, s2m set buffer length..."); // output length
offset = 32'h0000_0028;
data = test_length*3; //32'd1339392; // because we have "3" outputs for 1 input, so output data length should be "1024*436*3"
axil_cycles_gen(WriteCyc, PL_UPDMA, offset, data, 1);
//#20us
axil_cycles_gen(ReadCyc, PL_UPDMA, offset, data, 1);
//#20us230400
if(data == test_length*3) begin
    $display($time, "-> Fpga2Soc_Write PL_UPDMA offset %h = %h, PASS", offset, data);
end else begin
    $display($time, "-> Fpga2Soc_Write PL_UPDMA offset %h = %h, FAIL", offset, data);
    ->> error_event;
end
```

此 offset 對應到下圖所述的 register：

s2m Buffer Length	0x28	Set s2m buffer length, must set to 640*480/4.
-------------------	------	---

我們設定 output data length，由於我們 input 了 test_length 筆 data，因此在 FIR 運算完後我們期望會收到 3×input data length 筆 output data（因為每筆 input data 對應到分母、u 分子、v 分子共 3 個 output data），因此此 register 設定為 3×input data length。

- Configure offset 0x38、data=0x4508_0000 : s2m set buffer low

```
$display($time, "-> FpgaLocal_Write: PL_UPDMA, s2m set buffer low...");
offset = 32'h0000_0038;
data = 32'h4508_0000;
axil_cycles_gen(WriteCyc, PL_UPDMA, offset, data, 1);
//#20us
axil_cycles_gen(ReadCyc, PL_UPDMA, offset, data, 1);
//#20us
if(data == 32'h4508_0000) begin
    $display($time, "-> Fpga2Soc_Write PL_UPDMA offset %h = %h, PASS", offset, data);
end else begin
    $display($time, "-> Fpga2Soc_Write PL_UPDMA offset %h = %h, FAIL", offset, data);
    ->> error_event;
end
```

此 offset 對應到下圖所述的 register :

s2m Buffer Lower base address register ..	0x38 ..	bit 31~0 – The memory base address [31:0] of s2m buffer (Read/Write) ..
---	---------	---

觀察下圖 Vivado 中的 address map / address editor，我們發現 AXI_VIP_3(我們拿來作為 s2m 的 memory)的 memory address 被指定為 0x0000_0000_4508_0000，因此 s2m buffer lower base 為 0x4508_0000，故將此 register 寫入此值。

Name	Interface	Slave Segment	Master Base Address	Range	Master High Address
Network 0					
/axi_vip_0					
/ps_axil_0/axi_vip_0/Master_AXI (32 address bits : 4G)					
/ps_axil_0/ladma_s	ladma_s	reg0	0x6000_8000	4K	0x6000_8FFF
/ps_axil_0/s_axi	s_axi	reg0	0x6000_0000	32K	0x6000_7FFF
/ps_axil_0/s_axi_control	s_axi_control	Reg	0x6000_9000	4K	0x6000_9FFF
Network 1					
/ps_axil_0					
/ps_axil_0/ps_axil_0/ladma_mm (64 address bits : 16E)					
/ps_axil_0/ps_axil_0/axi_vip_1/S_AXI	S_AXI	Reg	0x0000_0000_44A0_0000	32K	0x0000_0000_44A0_7FFF
Network 2					
/Userdma_0					
/Userdma_0/Data_m_axi_gmem0 (64 address bits : 16E)					
/ps_axil_0/ps_axil_0/ps_axil_0/axi_vip_2/S_AXI	S_AXI	Reg	0x0000_0000_4500_0000	512K	0x0000_0000_4507_FFFF
Network 3					
/Userdma_0					
/Userdma_0/Data_m_axi_gmem1 (64 address bits : 16E)					
/ps_axil_0/ps_axil_0/ps_axil_0/axi_vip_3/S_AXI	S_AXI	Reg	0x0000_0000_4508_0000	512K	0x0000_0000_450F_FFFF

➤ Configure offset 0x3C、data=0x0 : s2m set buffer high

```
$display($time, "-> FpgaLocal_Write: PL_UPDMA, s2m set buffer high...");  
offset = 32'h0000_003C;  
data = 32'h0000_0000;  
axil_cycles_gen(WriteCyc, PL_UPDMA, offset, data, 1);  
//#20us  
axil_cycles_gen(ReadCyc, PL_UPDMA, offset, data, 1);  
//#20us  
if(data == 32'h0000_0000) begin  
  $display($time, "-> Fpga2Soc_Write PL_UPDMA offset %h = %h, PASS", offset, data);  
end else begin  
  $display($time, "-> Fpga2Soc_Write PL_UPDMA offset %h = %h, FAIL", offset, data);  
  ->> error_event;  
end
```

此 offset 對應到下圖所述的 register :

s2m Buffer Upper base address register ..	0x3C ..	bit 31~0 – The memory base address [63:32] of s2m buffer (Read/Write) ..
---	---------	--

由於 AXI_VIP_3 (我們拿來作為 s2m 的 memory) 的 memory address 被指定為 0x0000_0000_4508_0000 , 因此 s2m buffer upper base 為 0x0000_0000 , 故將此 register 寫入此值 :

Name	Interface	Slave Segment	Master Base Address	Range	Master High Address
Network 0					
/axi_vip_0					
/ps_axil_0/Master_AXI (32 address bits : 4G)					
/ps_axil_0/ladma_s	ladma_s	reg0	0x6000_8000	4K	0x6000_BFFF
/ps_axil_0/s_axi	s_axi	reg0	0x6000_0000	32K	0x6000_7FFF
/ps_axil_0/s_axi_control	s_axi_control	Reg	0x6000_9000	4K	0x6000_9FFF
Network 1					
/ps_axil_0					
/ps_axil_0/ladma_mm (64 address bits : 16E)					
/axi_vip_1/S_AXI	S_AXI	Reg	0x0000_0000_44A0_0000	32K	0x0000_0000_44A0_7FFF
Network 2					
/userdma_0					
/userdma_0/Data_m_axi_gmem0 (64 address bits : 16E)					
/axi_vip_2/S_AXI	S_AXI	Reg	0x0000_0000_4500_0000	512K	0x0000_0000_4507_FFFF
Network 3					
/userdma_0					
/userdma_0/Data_m_axi_gmem1 (64 address bits : 16E)					
/axi_vip_3/S_AXI	S_AXI	Reg	0x0000_0000_4508_0000	512K	0x0000_0000_450F_FFFF

- Configure offset 0x54 、 data= TST_FRAME_WIDTH : image width

```
$display($time, "-> FpgaLocal_Write: PL_UPDMA, set image width...");  
offset = 32'h0000_0054;  
data = TST_FRAME_WIDTH; //32'h0000_00A0;  
axil_cycles_gen(WriteCyc, PL_UPDMA, offset, data, 1);  
//#20us  
axil_cycles_gen(ReadCyc, PL_UPDMA, offset, data, 1);  
//#20us  
if(data == TST_FRAME_WIDTH) begin  
  $display($time, "-> Fpga2Soc_Write PL_UPDMA offset %h = %h, PASS", offset, data);  
end else begin  
  $display($time, "-> Fpga2Soc_Write PL_UPDMA offset %h = %h, FAIL", offset, data);  
  ->> error_event;  
end
```

此 offset 對應到下圖所述的 register :

Image width	0x54	bit 31~0 – <u>Image_width[31:0]</u> (Read/Write) Note: The value of this register is DW unit, so the value should be real width divided by 4. This value must be 160, due to Image is 640(width)*480(height).
-------------	------	--

輸入 test image 的 width 即可。

- Configure offset 0x5C 、 data=0x4500_0000 : m2s set buffer low

```

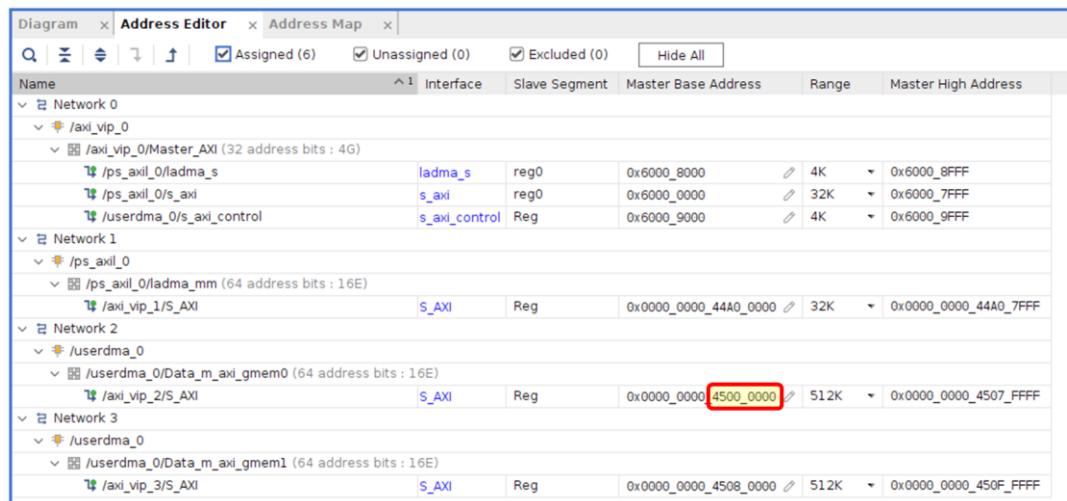
$display($time, "-> FpgaLocal_Write: PL_UPDMA, m2s set buffer low...");
offset = 32'h0000_005C;
data = 32'h4500_0000;
axil_cycles_gen(WriteCyc, PL_UPDMA, offset, data, 1);
//#20us
axil_cycles_gen(ReadCyc, PL_UPDMA, offset, data, 1);
//#20us
if(data == 32'h4500_0000) begin
    $display($time, "-> Fpga2Soc_Write PL_UPDMA offset %h = %h, PASS", offset, data);
end else begin
    $display($time, "-> Fpga2Soc_Write PL_UPDMA offset %h = %h, FAIL", offset, data);
    -> error_event;
end

```

此 offset 對應到下圖所述的 register :

m2s Buffer Lower base address register ..	0x5C ..	bit 31~0 – The memory base address [31:0] of m2s buffer (Read/Write) ..
---	---------	---

觀察下圖 Vivado 中的 address map / address editor，我們發現 AXI_VIP_2(我們拿來作為 m2s 的 memory)的 memory address 被指定為 0x0000_0000_4500_0000，因此 m2s buffer lower base 為 0x4500_0000，故將此 register 寫入此值。



- Configure offset 0x60、data=0x0 : m2s set buffer high

```

$display($time, "> FpgaLocal_Write: PL_UPDMA, m2s set buffer high...");  

offset = 32'h0000_0060;  

data = 32'h0000_0000;  

axil_cycles_gen(WriteCyc, PL_UPDMA, offset, data, 1);  

//#20us  

axil_cycles_gen(ReadCyc, PL_UPDMA, offset, data, 1);  

//#20us  

if(data == 32'h0000_0000) begin  

    $display($time, "> Fpga2Soc_Write PL_UPDMA offset %h = %h, PASS", offset, data);  

end else begin  

    $display($time, "> Fpga2Soc_Write PL_UPDMA offset %h = %h, FAIL", offset, data);  

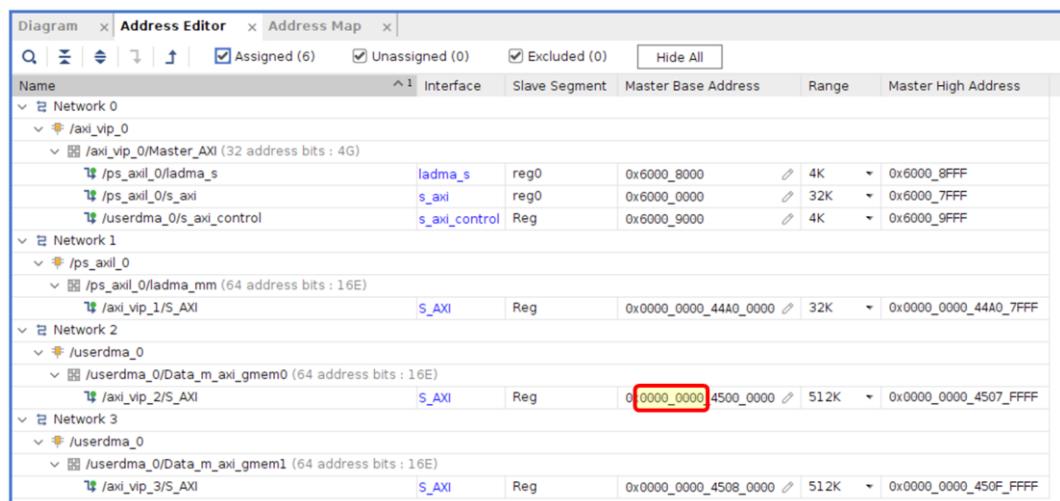
    -> error_event;
end

```

此 offset 對應到下圖所述的 register :

m2s Buffer Upper base address register ..	0x60 ..	bit 31~0 – The memory base address [63:32] of m2s buffer (Read/Write) ..
---	---------	--

由於 AXI_VIP_2 (我們拿來作為 m2s 的 memory) 的 memory address 被指定為 0x0000_0000_4500_0000 , 因此 m2s buffer upper base 為 0x0000_0000 , 故將此 register 寫入此值 :



- Configure offset 0x80 、 data=input data length=input data length=width×height : m2s set buffer length

```

$display($time, "-> FpgaLocal_Write: PL_UPDMA, m2s set buffer length..."); // input length
offset = 32'h0000_0080;
data = 32'd446464; // which is "1024*436"
axil_cycles_gen(WriteCyc, PL_UPDMA, offset, data, 1);
//#20us
axil_cycles_gen(ReadCyc, PL_UPDMA, offset, data, 1);
//#20us
if(data == 32'd446464) begin
    $display($time, "-> Fpga2Soc_Write PL_UPDMA offset %h = %h, PASS", offset, data);
end else begin
    $display($time, "-> Fpga2Soc_Write PL_UPDMA offset %h = %h, FAIL", offset, data);
    ->> error_event;
end

```

此 offset 對應到下圖所述的 register :

m2s Buffer Length	0x80	Set m2s buffer length, must set to 640*480/4.
-------------------	------	---

在此我們設定 input (Stream_in, ss) data length=1024*436=446464 筆 data。即使因為模擬時間太長而停止 simulation 也沒關係，主要是看 output 的數目來決定何時停止模擬，input stream 則是一直不斷給予新的值以避免沒有新 data 進來 stall 住的情形。

3. Select user project 2：由於我們將 OpticalFlow IP 運算用的.v 檔都放在 user project 2 裡，因此若要執行 optical flow 運算，則要將 SOC_CC=0x30005000 位址的值 program 成 2。此位址對應到 FPGA side 的 0x60005000 (base address 為 0x60000000(即程式碼中的 base_addr)) 在加上 SOC_CC 的 offset=0x5000)，因此我們要將 "2" 寫入到 base_addr+0x5000 的位址：

```

//Select OpticalFlow IP user project
$display($time, "-> Fpga2Soc_Write: SOC_CC");
offset = 0;
data = 32'h0000_0002;
axil_cycles_gen(WriteCyc, SOC_CC, offset, data, 1);
//#20us
axil_cycles_gen(ReadCyc, SOC_CC, offset, data, 1);
//#20us
if(data == 32'h0000_0002) begin
    $display($time, "-> Fpga2Soc_Write SOC_CC offset %h = %h, PASS", offset, data);
end else begin
    $display($time, "-> Fpga2Soc_Write SOC_CC offset %h = %h, FAIL", offset, data);
    ->> error_event;
end

```

4. 接著就可以開始 configure OpticalFlow IP 的前置作業了，按照前面 FSIC simulation 的流程，這裡只需完成 AXI-Lite 的 configuration 相關任務即可，而 stream input/output 則是在前面 configure user_DMA 時就已交給 user_DMA 來完成：

A. 將其 reset 值 program 為 1 並讀回檢查

```
//////// Set rst = 1 (address=0x00) ///////
offset = 0;
data = 32'd1;
axil_cycles_gen(WriteCyc, SOC_UP, offset, data, 1);
axil_cycles_gen(ReadCyc, SOC_UP, offset, data, 1);
//#20us
if(data == 32'd1) begin
    $display($time, "> Fpga2Soc_Write SOC_UP offset %h = %h, PASS", offset, data);
end else begin
    $display($time, "> Fpga2Soc_Write SOC_UP offset %h = %h, FAIL", offset, data);
    ->> error_event;
end
```

B. 將 reset 的值 program 為 0，表示 reset 結束，並且讀取此位址的值以確認有被 program 回 0

```
//////// Set rst = 0 (address=0x00) ///////
offset = 0;
data = 32'd0;
axil_cycles_gen(WriteCyc, SOC_UP, offset, data, 1);
//#20us
axil_cycles_gen(ReadCyc, SOC_UP, offset, data, 1);
//#20us
if(data == 32'd0) begin
    $display($time, "> Fpga2Soc_Write SOC_UP offset %h = %h, PASS", offset, data);
end else begin
    $display($time, "> Fpga2Soc_Write SOC_UP offset %h = %h, FAIL", offset, data);
    ->> error_event;
end
```

C. 輸入圖片的 width 和 height，以及拿來判別分母是否過小的比較基準值 shift_threshold。接著再讀回這些值並 check 是否成功 program

```

///////// Program widthIn ///////
offset = 12'h4;
data = TST_FRAME_WIDTH;
axil_cycles_gen(WriteCyc, SOC_UP, offset, data, 1);
//#20us
axil_cycles_gen(ReadCyc, SOC_UP, offset, data, 1);
//#20us
if(data == TST_FRAME_WIDTH) begin
    $display($time, "> Fpga2Soc_Write SOC_CC offset %h = %h, PASS", offset, data);
end else begin
    $display($time, "> Fpga2Soc_Write SOC_CC offset %h = %h, FAIL", offset, data);
    ->> error_event;
end

///////// Program heightIn ///////
offset = 12'h8;
data = TST_FRAME_HEIGHT;
axil_cycles_gen(WriteCyc, SOC_UP, offset, data, 1);
//#20us
axil_cycles_gen(ReadCyc, SOC_UP, offset, data, 1);
//#20us
if(data == TST_FRAME_HEIGHT) begin
    $display($time, "> Fpga2Soc_Write SOC_CC offset %h = %h, PASS", offset, data);
end else begin
    $display($time, "> Fpga2Soc_Write SOC_CC offset %h = %h, FAIL", offset, data);
    ->> error_event;
end

///////// Program shift_threshold ///////
offset = 12'h0C;
data = shift_threshold;
axil_cycles_gen(WriteCyc, SOC_UP, offset, data, 1);
//#20us
axil_cycles_gen(ReadCyc, SOC_UP, offset, data, 1);
//#20us
if(data == shift_threshold) begin
    $display($time, "> Fpga2Soc_Write SOC_CC offset %h = %h, PASS", offset, data);
end else begin
    $display($time, "> Fpga2Soc_Write SOC_CC offset %h = %h, FAIL", offset, data);
    ->> error_event;
end

```

5. Program user_DMA 的 ap_start : 使 user_DMA 開始運作

```

///////// Program "userDMA" ap_start = 1 ///////
$display($time, "> FpgaLocal_Write: PL_UPDMA, set ap_start..."); 
offset = 12'h000;
data = 32'h0000_0001;
axil_cycles_gen(WriteCyc, PL_UPDMA, offset, data, 1);

```

6. 等待直到 user_DMA 的任務完成 : user_DMA 會將 input data 從 memory 取值並透過 m2s 的方式餵給 Caravel SoC 中的 FIR engine，並將 FIR 計算過後透過 AXI-Stream (sm) interface 傳出來的 output

data 透過 s2m 的方式儲存在 AXI_VIP_2 的 memory 中。這個過程需要一段時間，因此我們等待直到 user_DMA done，才結束這個 task。

```

fork
|   CheckuserDMADone();
|   join_none

|   @(userdma_done);

|   $display($time, "-> End SocUp2DmaPath() test..."); 
|   $display($time, "-> =====");
|

```

CheckuserDMADone() 這個 task 主要是等待 stream_in 及 stream_out 的任務完成，完成後 user_DMA 會將 offset=0x10 位址的值寫入 1，因此我們可以透過讀取這個位址的方式來得知是否已完成儲存所有 output data。User_DMA 完成後， $3 \times \text{test_length}$ 筆 output data 會被儲存在 AXI_VIP_2 的 memory 中，因此我們可以透過 slave_agent2.mem_model.backdoor_memory_read_4byte() 這個 function 將 AXI_VIP_2 的 memory 中的值讀取出來，並寫至 updma_output.log 這個檔案，方便我們查看 OpticalFlow IP 結果。

```

task CheckuserDMADone;
begin
    $display($time, "-> Starting CheckuserDMADone()..."); 
    $display($time, "-> =====");
    $display($time, "-> FpgaLocal_Read: PL_UPDMA");

    keepChk = 1;
    offset = 32'h0000_0010;
    $display($time, "-> Waiting buffer transfer done..."); 
    while (keepChk) begin
        #10us
        axil_cycles_gen(ReadCyc, PL_UPDMA, offset, data, 0);
        if(data == 32'h0000_0001 || timeout_flag==1) begin
            if ( timeout_flag ) $display($time, "-> ERROR: Time Out - force quit!!!");
            else $display($time, "-> Buffer transfer done. offset %h = %h, PASS", offset, data);
            keepChk = 0;
        end
        fd = $fopen("../../../../../updma_output.log", "w");
        for (index = 0; index < test_length; index +=1) begin
            for (index_inner = 0; index_inner < 3; index_inner +=1) begin
                //slave_agent2.mem_model.backdoor_memory_write_4byte(addr+(index*3+index_inner)*4,updma_data_golden_out,4'b1111);
                $fdisplay(fd, "%08h", slave_agent2.mem_model.backdoor_memory_read_4byte(addr+(index*3+index_inner)*4));
            end
        end
        $fclose(fd);
    end
    ->> userdma_done;
    $display($time, "-> End CheckuserDMADone()..."); 
    $display($time, "-> =====");

```

由於 1 筆 input 對應到 3 筆 output，故在此我們需使用 index_inner 來數出 3 次 output，並分別存入 updma_output.log 這個檔案中。

在/vivado 資料夾中執行 " ./run_vivado_fsic_sim " 進行 Vivado 環境的 simulation 後，等待一段時間完成模擬，可以在/vivado 資料夾找到「updma_input.log」、「updma_output_gold.log」、「updma_output.log」、「run_vivado_fsic_sim.log」，而波形檔則是放在 /ASoC-Final_project-optical_flow/vivado/vvd_caravel_fpga_sim/vvd_caravel_fpga_sim.sim/sim_1/behav/xsim/FSIC_tb_optical_flow.vcd。

執行結果 log 檔如下：(由於檔案非常長，因此在此只截最後一小部分，表示有完成 simulation，未出現 deadlock 的情形)

```

38799998.000 ns MSG fsic_tb, +100000 cycles, finish_flag=0, repeat_cnt=0097
39199998.000 ns MSG fsic_tb, +100000 cycles, finish_flag=0, repeat_cnt=0098
39599998.000 ns MSG fsic_tb, +100000 cycles, finish_flag=0, repeat_cnt=0099
39999998.000 ns MSG fsic_tb, +100000 cycles, finish_flag=0, repeat_cnt=0100
40399998.000 ns MSG fsic_tb, +100000 cycles, finish_flag=0, repeat_cnt=0101
40799998.000 ns MSG fsic_tb, +100000 cycles, finish_flag=0, repeat_cnt=0102
41199998.000 ns MSG fsic_tb, +100000 cycles, finish_flag=0, repeat_cnt=0103
41599998.000 ns MSG fsic_tb, +100000 cycles, finish_flag=0, repeat_cnt=0104
41999998.000 ns MSG fsic_tb, +100000 cycles, finish_flag=0, repeat_cnt=0105
42399998.000 ns MSG fsic_tb, +100000 cycles, finish_flag=0, repeat_cnt=0106
42799998.000 ns MSG fsic_tb, +100000 cycles, finish_flag=0, repeat_cnt=0107
43199998.000 ns MSG fsic_tb, +100000 cycles, finish_flag=0, repeat_cnt=0108
    43366658=> Buffer transfer done, offset 010 = 00000001, PASS
    43366658=> End CheckuserDMADone() ...
    43366658=> =====
    43366658=> End SocUp2DmaPath() test...
    43366658=> =====
43599998.000 ns MSG fsic_tb, +100000 cycles, finish_flag=0, repeat_cnt=0109
    43866658=> End of the test ...
Executing Ax14 End Of Simulation checks
$finish called at time : 43866658 ns : File "/home/ubuntu/Advanced_SoC/final_project/ASoC-Final_project-optical_flow/vivado/fsic_tb.sv" Line 149
run: Time (s): cpu = 00:05:40 ; elapsed = 00:16:08 . Memory (MB): peak = 3048.219 ; gain = 42.000 ; free physical = 121 ; free virtual = 6989
# exit
INFO: xsimkernel Simulation Memory Usage: 878924 KB (Peak: 878924 KB), Simulation CPU Usage: 954290 ms
INFO: [Common 17-206] Exiting Vivado at Tue Jun 18 19:43:53 2024 ...
=====
vivado complete
=====
```

updma_output_gold.log 與 updma_output.log 內容如下所示，可發現與 golden data (HLS)相符，因此計算結果也正確無誤！

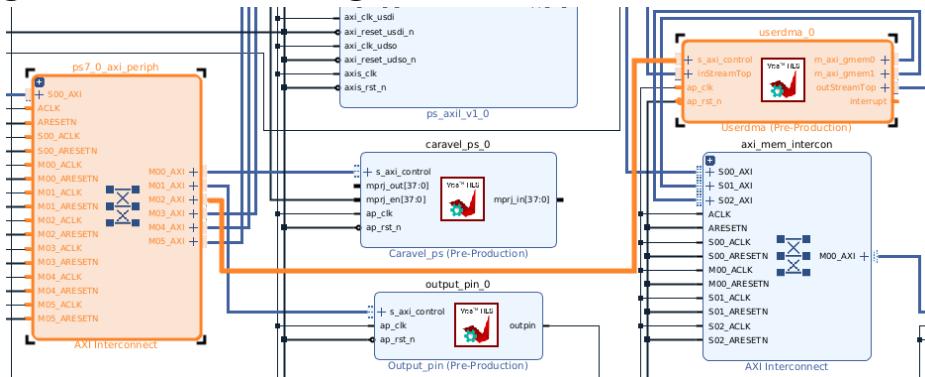
updma_output_gold.log 內容		updma_output.log 內容	
6125	00000000	6125	00000000
6126	00000000	6126	00000000
6127	00000000	6127	00000000
6128	00000000	6128	00000000
6129	00000000	6129	00000000
6130	00000000	6130	00000000
6131	00000000	6131	00000000
6132	00000000	6132	00000000
6133	00000000	6133	00000000
6134	00000000	6134	00000000
6135	00000000	6135	00000000
6136	00000000	6136	00000000
6137	00000000	6137	00000000
6138	00000000	6138	00000000
6139	00000000	6139	00000000
6140	00000000	6140	00000000
6141	00000000	6141	00000000
6142	00000000	6142	00000000
6143	00000000	6143	00000000
6144	00000000	6144	00000000
6145	00000000	6145	00000000
6146	00000000	6146	00000000
6147	00000000	6147	00000000
6148	00000000	6148	00000000
6149	00000000	6149	00000000
6150	00000000	6150	00000000
6151	bcf35d78	6151	bcf35d78
6152	c6b7d7b2	6152	c6b7d7b2
6153	3757668a	6153	3757668a
6154	3543480e	6154	3543480e
6155	c85d547b	6155	c85d547b
6156	ac26d907	6156	ac26d907
6157	29fbf87d	6157	29fbf87d
6158	bb8c2bf8	6158	bb8c2bf8
6159	b244cda7	6159	b244cda7
6160	3160d5db	6160	3160d5db
6161	36164ff7	6161	36164ff7
6162	466ec53f	6162	466ec53f
6163	77dbe802	6163	77dbe802
6164	1272d63f	6164	1272d63f
6165	42e6a655	6165	42e6a655
6166	56e37032	6166	56e37032
6167	e78e38b3	6167	e78e38b3
6168	11dff8af	6168	11dff8af
6169	7ed6f126	6169	7ed6f126
6170	ce56c66b	6170	ce56c66b
6171	11588ddb	6171	11588ddb
6172	6a7d29d5	6172	6a7d29d5

9. FSIC-FPGA validation (onlineFPGA)

在 validation 的部分，整個實作的流程與 lab4 大致相符，我們需要設定好 userDMA 用以輸入 input data 以及讀取並暫存 output data。

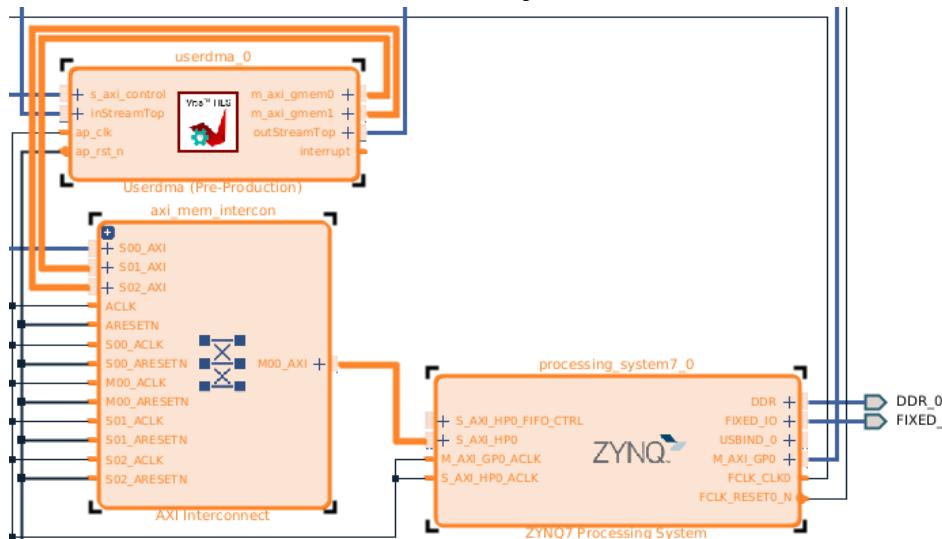
以下為使用 userDMA 的流程：

(1) Assign address to config userDMA

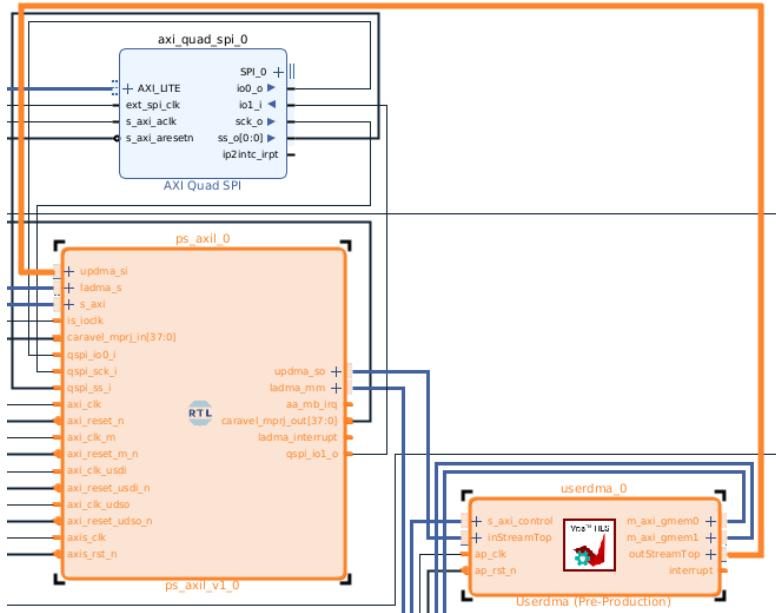


/userdma_0/s_axi_control s_axi_control Reg 0x6000_9000 4K 0x6000_9FFF

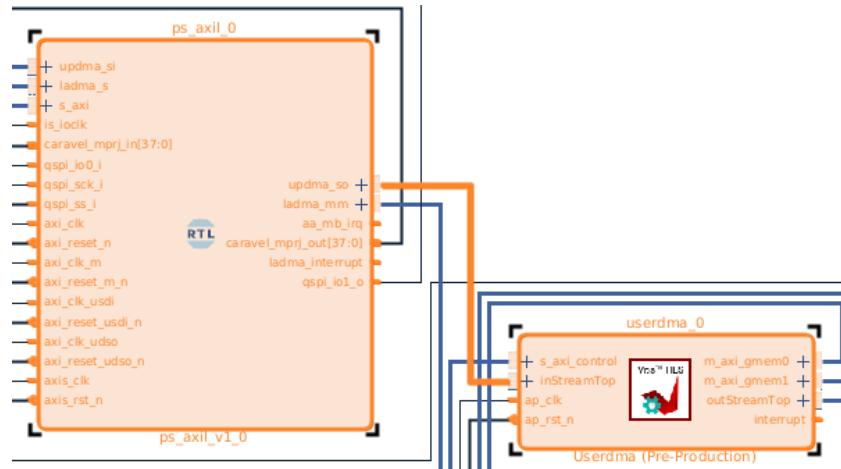
(2) Connect the userDMA to memory (for stream in/out interface)



(3) Connect the stream_in data for our user project



(4) Read the stream_out data to dump the result



將上述改動完畢並燒製 bitstream 之後，即可到 OnlineFPGA 的平台上進行 FPGA 的 validation。首先我們需要先將下列檔案上傳至 FPGA。

1. **caravel_fpga.bit**
2. **caravel_fpga.hwh**
3. **fsic.hex**
4. **caravel_fpga_fsic.ipynb**
5. **frame1.hex**
6. **frame2.hex**
7. **frame3.hex**
8. **frame4.hex**

以下為我們 ipynb 的執行流程，首先會有一段設定 FSIC 的部分在此省略，以下僅敘述我們所做的改動。

1. 先選擇 user project 2 已確定是在執行 opticalflow

```
In [40]: # Check Caravel-CC Config
ADDRESS_OFFSET = SOC_CC # 0x5000
print("mmio.read(ADDRESS_OFFSET): ", hex(mmio.read(ADDRESS_OFFSET)))

mmio.read(ADDRESS_OFFSET): 0x0

In [41]: # Caravel-CC Config
ADDRESS_OFFSET = SOC_CC # 0x5000
# select user project 2
mmio.write(ADDRESS_OFFSET, 0x00000002)

In [42]: # Check new Caravel-CC Config
ADDRESS_OFFSET = SOC_CC # 0x5000
print("mmio.read(ADDRESS_OFFSET): ", hex(mmio.read(ADDRESS_OFFSET)))

# The result should be "2" !!

mmio.read(ADDRESS_OFFSET): 0x2
```

2. 確定有正確的 reset 以啟動 user project

```
# Caravel-UP Config (configure OpticalFlow IP)
ADDRESS_OFFSET = SOC_UP # 0x0000

# Set rst = 1 (address=0x00)
offset = 0x0
write_data = 0x1
mmio.write(ADDRESS_OFFSET + offset, write_data)

# Read back & check
read_value = mmio.read(ADDRESS_OFFSET + offset)
if read_value == write_data:
    print("[PASS] Read OpticalFlow address ", hex(ADDRESS_OFFSET + offset), ": read out data = ", hex(read_value), "; golden = 0x1")
else:
    print("[ERROR] Read OpticalFlow address ", hex(ADDRESS_OFFSET + offset), ": read out data = ", hex(read_value), "; golden = 0x1")

# Set rst = 0 (address=0x00)
offset = 0x0
write_data = 0x0
mmio.write(ADDRESS_OFFSET + offset, write_data)

# Read back & check
read_value = mmio.read(ADDRESS_OFFSET + offset)
if read_value == write_data:
    print("[PASS] Read OpticalFlow address ", hex(ADDRESS_OFFSET + offset), ": read out data = ", hex(read_value), "; golden = 0x0")
else:
    print("[ERROR] Read OpticalFlow address ", hex(ADDRESS_OFFSET + offset), ": read out data = ", hex(read_value), "; golden = 0x0")
```

3. 設定 image height 和 width 以及 shift_threshold。

```
TST_FRAME_WIDTH = 1024
TST_FRAME_HEIGHT = 436
shift_threshold = 105

# Program widthIn, heightIn, and shift_threshold
mmio.write(ADDRESS_OFFSET + 0x04, TST_FRAME_WIDTH)
mmio.write(ADDRESS_OFFSET + 0x08, TST_FRAME_HEIGHT)
mmio.write(ADDRESS_OFFSET + 0x0C, shift_threshold)

# Read back widthIn, heightIn, shift_threshold & check
print("Read OpticalFlow address ", hex(ADDRESS_OFFSET + 0x04), ": read out data = ", hex(mmio.read(ADDRESS_OFFSET + 0x04)))
print("Read OpticalFlow address ", hex(ADDRESS_OFFSET + 0x08), ": read out data = ", hex(mmio.read(ADDRESS_OFFSET + 0x08)))
print("Read OpticalFlow address ", hex(ADDRESS_OFFSET + 0x0C), ": read out data = ", hex(mmio.read(ADDRESS_OFFSET + 0x0C)))
```

4. 設定 userDMA 的 input data width 和 output data width 並且將 input data load 進入 userDMA 中。

```

test_length = TST_FRAME_WIDTH*TST_FRAME_HEIGHT

# Allocation userDMA s2m memory
updma_buf_s2m = allocate(shape=(TST_FRAME_WIDTH*TST_FRAME_HEIGHT*3,), dtype=np.uint32)
print("updma_buf_s2m.device_address: ", hex(updma_buf_s2m.device_address))

s2m_BASE_ADDRESS = updma_buf_s2m.device_address
s2m_ADDRESS_RANGE = 0x1000
s2m_buf_mmio = MMIO(s2m_BASE_ADDRESS, s2m_ADDRESS_RANGE)

# userDMA Configuration
ADDRESS_OFFSET = PL_UPDMA # 0x9000
# exit clear operation
mmio.write(ADDRESS_OFFSET + 0x20, 0x00000000)
# disable to clear
mmio.write(ADDRESS_OFFSET + 0x30, 0x00000000)
# set buffer length
mmio.write(ADDRESS_OFFSET + 0x28, test_length*3)
# set buffer low
mmio.write(ADDRESS_OFFSET + 0x38, updma_buf_s2m.device_address)
# set buffer high
mmio.write(ADDRESS_OFFSET + 0x3C, 0x00000000)

```

```

# Allocation userDMA m2s memory
updma_buf_m2s = allocate(shape=(TST_FRAME_WIDTH*TST_FRAME_HEIGHT,), dtype=np.uint32)
print("updma_buf_m2s.device_address: ", hex(updma_buf_m2s.device_address))

# set input data
frame1_pointer = open("frame1.hex", "r")
frame2_pointer = open("frame2.hex", "r")
frame3_pointer = open("frame3.hex", "r")
frame4_pointer = open("frame4.hex", "r")
for i in range(TST_FRAME_WIDTH*TST_FRAME_HEIGHT):
    frame1_value = int(frame1_pointer.readline().strip('\n'),16)
    frame2_value = int(frame2_pointer.readline().strip('\n'),16)
    frame3_value = int(frame3_pointer.readline().strip('\n'),16)
    frame4_value = int(frame4_pointer.readline().strip('\n'),16)
    updma_buf_m2s[i] = (frame4_value<<24) + (frame3_value<<16) + (frame2_value<<8) + frame1_value
# print(hex((frame4_value<<24) + (frame3_value<<16) + (frame2_value<<8) + frame1_value))

frame1_pointer.close()
frame2_pointer.close()
frame3_pointer.close()
frame4_pointer.close()

m2s_BASE_ADDRESS = updma_buf_m2s.device_address
m2s_ADDRESS_RANGE = 0x1000
m2s_buf_mmio = MMIO(m2s_BASE_ADDRESS, m2s_ADDRESS_RANGE)

# userDMA Configuration
ADDRESS_OFFSET = PL_UPDMA # 0x9000
# exit clear operation
mmio.write(ADDRESS_OFFSET + 0x78, 0x00000000)
# disable to clear
mmio.write(ADDRESS_OFFSET + 0x88, 0x00000000)
# set buffer length
mmio.write(ADDRESS_OFFSET + 0x80, TST_FRAME_WIDTH*TST_FRAME_HEIGHT)
# set buffer low
mmio.write(ADDRESS_OFFSET + 0x5C, updma_buf_m2s.device_address)
# set buffer high
mmio.write(ADDRESS_OFFSET + 0x60, 0x00000000)

```

5. 接著 program ap_start 後，即可開始等待最後的 output done signal (由 userDMA 而來)，確認 output data length 已經達到預期的數值。

```
# PL_UPDMA, set ap_start
mmio.write(PL_UPDMA, 0x00000001)

# Check userDMA Done
keepChk = True
while keepChk:
    if mmio.read(PL_UPDMA + 0x10) == 0x00000001:
        keepChk = False
print("[Congratulations] Read userDMA buffer transfer done, address ",hex(PL_UPDMA + 0x10), ": read out data = 1")
```

6. 最後即可將所有 output data print 出來，用以和 golden 值做比較

```
# print out updma_buf_s2m
for i in range(test_length*3):
    print("updma_buf_s2m[", i, "] = ", hex(updma_buf_s2m[i]))
```

由 Output Result 可以確認，output data 皆正確。

updma_buf_s2m[6141] = 0x0	6149 00000000
updma_buf_s2m[6142] = 0x0	6150 00000000
updma_buf_s2m[6143] = 0x0	6151 bcf35d78
updma_buf_s2m[6144] = 0x0	6152 c6b7d7b2
updma_buf_s2m[6145] = 0x0	6153 3757668a
updma_buf_s2m[6146] = 0x0	6154 3543480e
updma_buf_s2m[6147] = 0x0	6155 c85d547b
updma_buf_s2m[6148] = 0x0	6156 ac26d907
updma_buf_s2m[6149] = 0x0	6157 29fbf87d
updma_buf_s2m[6150] = 0xbcf35d78	6158 bb8c2bf8
updma_buf_s2m[6151] = 0xc6b7d7b2	6159 b244cda7
updma_buf_s2m[6152] = 0x3757668a	6160 3160d5db
updma_buf_s2m[6153] = 0x3543480e	6161 36164ff7
updma_buf_s2m[6154] = 0xc85d547b	6162 466ec53f
updma_buf_s2m[6155] = 0xac26d907	6163 77dbe802
updma_buf_s2m[6156] = 0x29fbf87d	6164 1272d63f
updma_buf_s2m[6157] = 0xbb8c2bf8	6165 42e6a655
...	6166 56e37032
...	6167 e78e38b3

10. Performance comparison

我們在 HLS 的 testbench 中加入 timer 測量 algorithm C 的執行時間，並 print 在螢幕上，發現約為 0.31 秒。

```
++++++ (Report execution timing) ++++++
algorithm: 0.310000 seconds
++++++
```

而 rosetta (底下的 Reference 1.) 中也有寫出一種 HLS implementation，其 performance 如下：

On ZC706

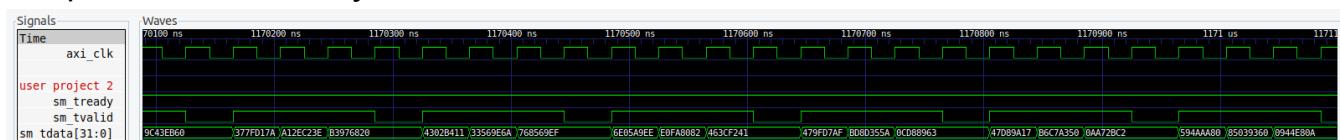
Benchmark	#LUTs	#FFs	#BRAMs	#DSPs	Runtime (ms)	Throughput
Floating_point	42878	61078	54	454	24.3	41.2 frames/s
Fixed_point	27869	47860	72	354	19.7	50.8 images/s

而在 Reference 2. 的 paper 中，他們同樣使用 HLS 來實作更複雜的 LK 法光流，其 performance table 為：

COMPARISON OF THE HARDWARE RESOURCE CONSUMPTION AND PERFORMANCE BETWEEN THE PROPOSED HARDWARE ARCHITECTURE AND OTHER STATE-OF-THE-ART FPGA-BASED OPTICAL FLOW IMPLEMENTATION

Implementation	With Pyramid	Image Resolution	Frame Rate (fps)	Throughput (Mpixels/s)	Platform	Clock Frequency (MHz)	Average Power (W)	LUT	FF	DSPs	BRAMs
Our work (Multi-Scale)	Yes	752×480	93	33.6	Xilinx Zynq XC7z100	180	0.69	66440	64630	49	16.5
Barranco [8]	Yes	640×480	32	9.8	Xilinx Virtex-4	44	N/A	51879	46122	124	244
Tomasi [9]	Yes	640×480	31.5	9.7	Xilinx Virtex-4	45	4.35	40073	60564	132	106
He [33]	Yes	1280×720	60	55.3	Xilinx Zynq 7020	667	0.356	41000	26000	127	15
Blachut [34]	Yes	1280×720	50	46.1	Xilinx Virtex-7	N/A	N/A	41167	52324	N/A	186
Smets [35]	Yes	640×480	51	15.7	Xilinx XC7A100T	20	0.024	41853	12599	50	43
Our work (Mono-scale)	No	752×480	243	87.7	Xilinx Zynq XC7z100	180	0.27	29249	37011	19	6.5
Kunz [36]	No	640×512	30	9.8	Altera Stratix IV	294.9	N/A	16997	66220	476	63
Mahalingam [37]	No	640×480	30	9.2	Xilinx Virtex-2	55	N/A	11086	N/A	23	20
Barranco [8]	No	640×480	270	82.9	Xilinx Virtex-4	83	N/A	50000	7000	N/A	N/A

我們由於實作出 II=2、最後一個 module II=4 的 design，故平均每個 pixel 的 output 之間差 4 個 cycle，此現象也可在 FSIC simulation 的波形中看到：



每 4 個 cycle 可輸出 1 個 pixel 的 3 個 data。

而合成時可知我們的 clock frequency 為 40MHz 左右，也就是 period 為 25ns。故我們的 design 處理一張 output image (size=1024×436) 所需的時間為

$$4 \times 25\text{ns} \times 1024 \times 436 = 44.64\text{ms}$$

此 performance 較 software 快了約 7 倍。Throughput 為其倒數，約為 22.401 frames/s。

而 resource 的使用量約為：

Name	Slice LUTs (53200)	Slice Registers (106400)	F7 Muxes (26600)	F8 Muxes (13300)	Block RAM Tile (140)	DSPs (220)	Bonded IOPADs (130)	BUFGCT (32)
> U_USRPRJ2 (de)	24372	19423	1	0	43	216	0	

比上述兩個 reference HLS 都少。因此我們認為我們的 design 雖然執行速度沒有他們快，但使用的資源量是比較少的！

● Reference

1. Open source (reference algorithm C code): <https://github.com/cornell-zhang/rosetta/tree/master/optical-flow>
2. Y. Gong et al., "A Real-Time and Efficient Optical Flow Tracking Accelerator on FPGA Platform," in *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 70, no. 12, pp. 4914-4927, Dec. 2023, doi: 10.1109/TCSI.2023.3298969.

● Github link for our work about final project

https://github.com/whywhytellmewhy/ASoC-Final_project-optical_flow

在上述 Github 連結中有關於這次 lab 及 final project 的相關檔案。

關於上述 Github 中的檔案及模擬方式的更多說明可至下方連結中的 README.md 查看：<https://github.com/whywhytellmewhy/Advanced-SoC-design>