1. Code 解說:

Case2:

```
#include "spmv.h"

void spmv(int rowPtr[NUM_ROWS+1], int columnIndex[NNZ],
        DTYPE values[NNZ], DTYPE y[SIZE], DTYPE x[SIZE])
{
    // Loop L1
    L1: for (int i = 0; i < NUM_ROWS; i++) {
        DTYPE y0 = 0;

        // Loop L2
        L2: for (int k = rowPtr[i]; k < rowPtr[i+1]; k++) {
            #pragma HLS PIPELINE
            y0 += values[k] * x[columnIndex[k]];
        }
        y[i] = y0;
    }
}
```

Loop L1 就是對矩陣的行進行遍歷。 將矩陣當前的行與向量 x 相乘，得到輸出的結果。

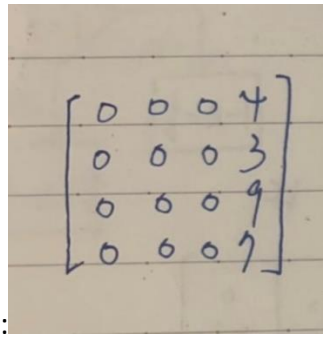
Loop L2 實現對矩陣 **M** 中每列元素的遍歷。 L2 迴圈反覆運算每一行非 0 元素的個數。 每次迴圈計算，能從 **value** 陣列中讀取矩陣 M 的非 0 元素然後對應的從 x 陣陣列中取得被乘向量 x 的值，對應相乘。然後我們又對 L2 進行 pipeline 使其變成使用一個乘法器跟加法器，依次執行乘法及加法。

Case4:

```
#include "spmv.h"

const static int S = 3;

void spmv(int rowPtr[NUM_ROWS+1], int columnIndex[NNZ],
          DTYPE values[NNZ], DTYPE y[SIZE], DTYPE x[SIZE])
{
  L1: for (int i = 0; i < NUM_ROWS; i++) {
      DTYPE y0 = 0;

      L2_1: for (int k = rowPtr[i]; k < rowPtr[i+1]; k += S) {
#pragma HLS pipeline II=S
          DTYPE yt = values[k] * x[columnIndex[k]];
          L2_2: for(int j = 1; j < S; j++) {
                  if(k+j < rowPtr[i+1]) {
                      yt += values[k+j] * x[columnIndex[k+j]];
                  }
          }
        y0 += yt;
      }

    y[i] = y0;
  }
}
```

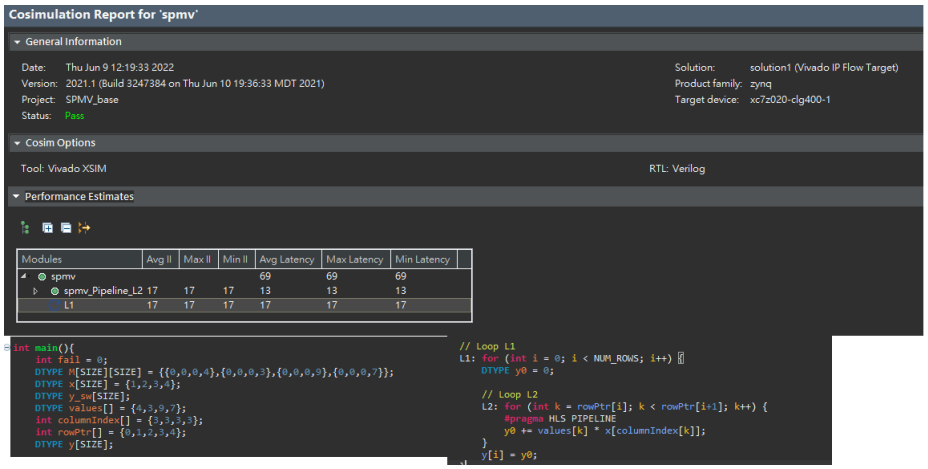
在 loop L2 把它拆分成 L2_1、L2_2，最內層的迴圈 **L2_2** 執行的次數由參數 **S** 確定。 內部迴圈包含了最原始的 **L2** 迴圈，其中循環邊界是由最原始的 **L2** 迴圈決定的。**L2_1** 迴圈包含了不確定次數的乘法和加法操作，運算次數由參數 **S** 確定，和一次遞歸完成累加。

2. 測試結果:
   測試矩陣 1:

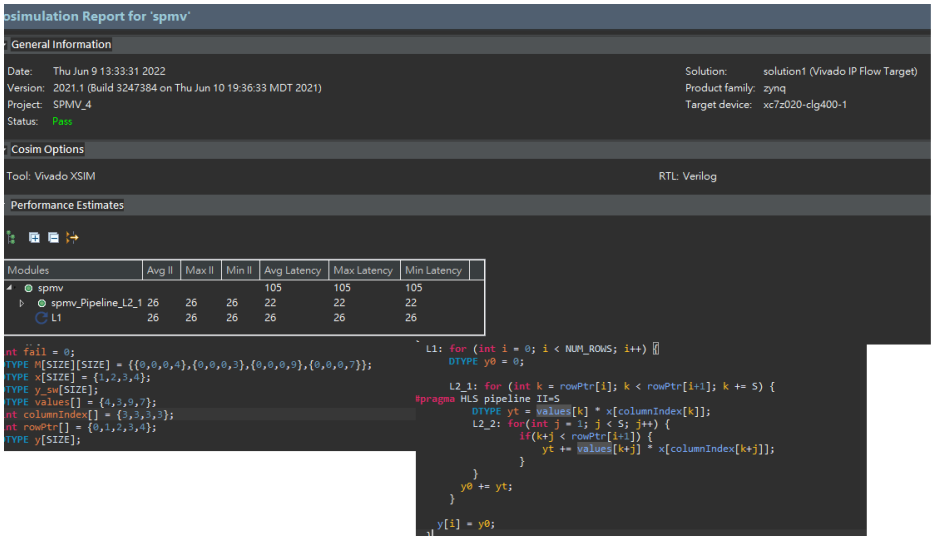


:

Case2:



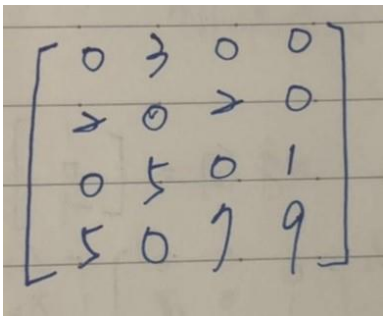
Case4:



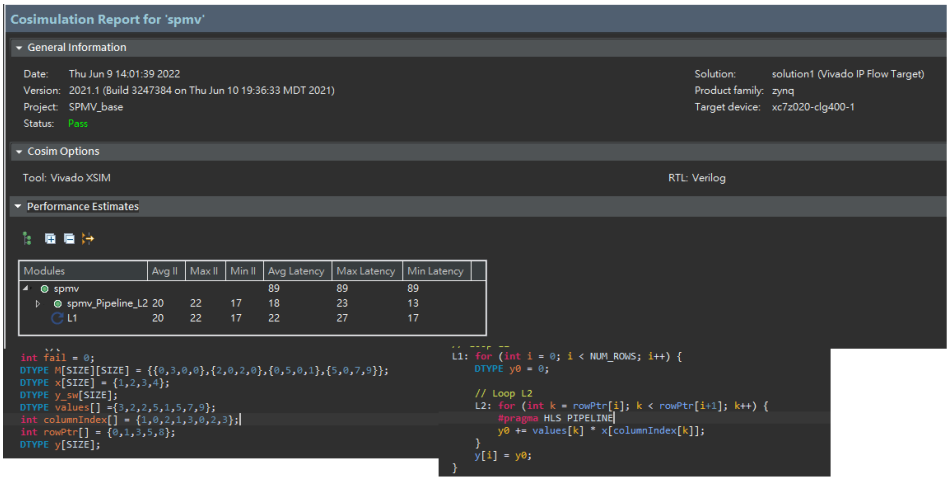
測試矩陣 1 結論:

第一個測試矩陣我使用非零數佔 1/4 去進行測試，結果發現真的如理論所想，當非零數很少的時候 case4 並不會比較快，因為 case4 在計算時一行有個 3 個元素和一行有一個元素計算的時間是相同的。 剩下的運算也需要在迴圈流水線中執行，即使他們的結果沒有用，所以其實反而是浪費的。

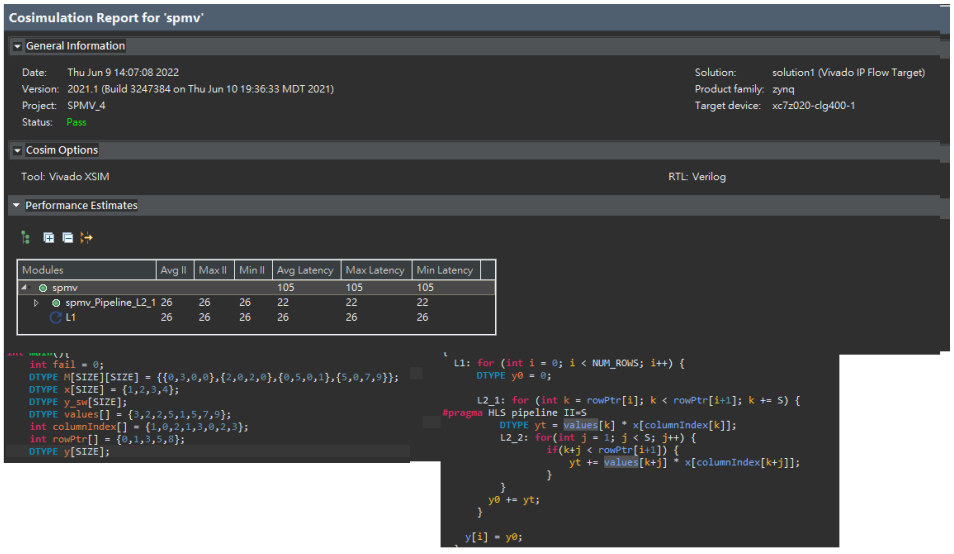
測試矩陣 2:



Case2:



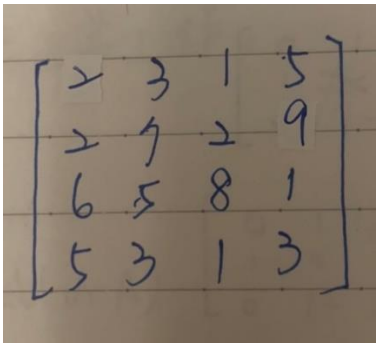
Case4:



測試矩陣 2 結論:

第二個測試矩陣我使用非零數佔 1/2 去進行測試，結果發現雖然非零數變多，但依舊是 case2 使用的時間會比較少。

測試矩陣 3:



Case2:

Cosimulation Report for 'spmv'

General Information

Date: Thu Jun 9 16:57:05 2022
Version: 2021.1 (Build 3247384 on Thu Jun 10 19:36:33 MDT 2021)
Project: SPMV_base
Status: Pass

Solution: solution1 (Vivado IP Flow Target)
Product family: zynq
Target device: xc7z020-clg400-1

Cosim Options

Tool: Vivado XSIM

RTL: Verilog

Performance Estimates

Modules	Avg II	Max II	Min II	Avg Latency	Max Latency	Min Latency
spmv				129	129	129
spmv_Pipeline_L2	32	32	32	28	28	28
L1	32	32	32	32	32	32

```
int main(){
    int fail = 0;
    DTYPE M[SIZE][SIZE] = {{2,3,1,5},{2,7,2,9},{6,5,8,1},{5,3,1,3}};
    DTYPE x[SIZE] = {1,2,3,4};
    DTYPE y_sw[SIZE];
    DTYPE values[] ={2,3,1,5,2,7,2,9,6,5,8,1,5,3,1,3};
    int columnIndex[] = {0,1,2,3,0,1,2,3,0,1,2,3,0,1,2,3};
    int rowPtr[] = {0,4,8,12,16};
    DTYPE y[SIZE];
```

```
// Loop L1
L1: for (int i = 0; i < NUM_ROWS; i++) {
    DTYPE y0 = 0;

    // Loop L2
    L2: for (int k = rowPtr[i]; k < rowPtr[i+1]; k++) {
        #pragma HLS PIPELINE
        y0 += values[k] * x[columnIndex[k]];
    }
    y[i] = y0;
}
```

Case4:

Cosimulation Report for 'spmv'

General Information

Date: Thu Jun 9 16:59:29 2022
Version: 2021.1 (Build 3247384 on Thu Jun 10 19:36:33 MDT 2021)
Project: SPMV_4
Status: Pass

Solution: solution1 (Vivado IP Flow Target)
Product family: zynq
Target device: xc7z020-clg400-1

Cosim Options

Tool: Vivado XSIM

RTL: Verilog

Performance Estimates

Modules	Avg II	Max II	Min II	Avg Latency	Max Latency	Min Latency
spmv				121	121	121
spmv_Pipeline_L2_1	30	30	30	26	26	26
L1	30	30	30	30	30	30

```
int main(){
    int fail = 0;
    DTYPE M[SIZE][SIZE] = {{2,3,1,5},{2,7,2,9},{6,5,8,1},{5,3,1,3}};
    DTYPE x[SIZE] = {1,2,3,4};
    DTYPE y_sw[SIZE];
    DTYPE values[] ={2,3,1,5,2,7,2,9,6,5,8,1,5,3,1,3};
    int columnIndex[] = {0,1,2,3,0,1,2,3,0,1,2,3,0,1,2,3};
    int rowPtr[] = {0,4,8,12,16};
    DTYPE y[SIZE];
```

```
L1: for (int i = 0; i < NUM_ROWS; i++) {
    DTYPE y0 = 0;

    L2_1: for (int k = rowPtr[i]; k < rowPtr[i+1]; k += S) {
#pragma HLS pipeline II=S
        DTYPE yt = values[k] * x[columnIndex[k]];
        L2_2: for(int j = 1; j < S; j++) {
            if(k+j < rowPtr[i+1]) {
                yt += values[k+j] * x[columnIndex[k+j]];
            }
        }
      y0 += yt;
    }

  y[i] = y0;
```

測試矩陣 3 結論:

第三個測試矩陣我使用非零數佔 100%，結果發現當沒有零的時候，case4 的優勢就發揮出來了。

結論:

透過多次嘗試，最後發現當非零數開始變多時兩者差距會開始縮小，而當非零數幾乎佔滿整個矩陣時，case4 的優勢才會被發揮出來，所以我認為如果今天非零數很少，那不用多說肯定是選 case2，反之非零數非常多，那就會是選擇 case4，但若是今天可能非零數佔整體矩陣可能接近一半或超過一半一點時，我認為就要兩種方法都實際去測試才能知道說對於這個矩陣用甚麼樣的方法會比較好了。