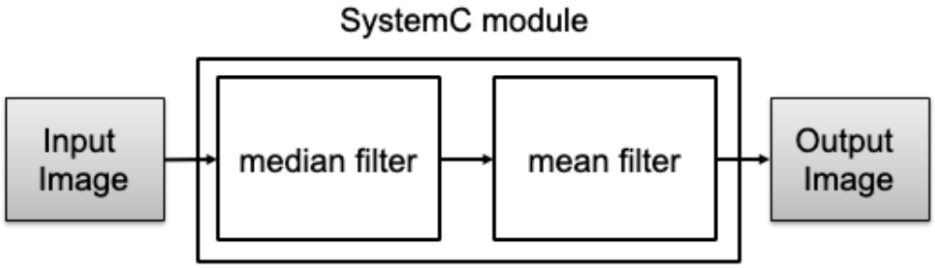
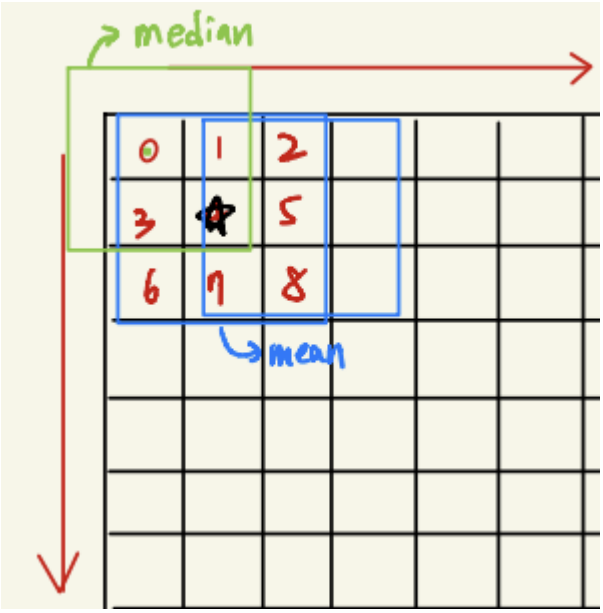
1. Median and Weighted Mean Filter





圖一

上圖是示意圖綠色框是做 median filter，藍色框是 mean filter，藍色框移動順序是做完 mean filter 後向右移一個，整排做完後再往下移一格。

因為 systemC module 先做 median filter 再做 mean filter，所以我設計需要先算出 9 個經過 median filter 所得出的 median value pixel，再將這 9 個 pixel 加起來平均後，得

到 1 個 mean value pixel 後，輸出到 Testbench。所以要得到 1 個 mean value pixel，必須讀取 81 個 pixel。

```
for (y = 0; y != height; ++y) {
  for (x = 0; x != width ; ++x) {
    adjustX = 1 ;// 1
    adjustY = 1 ; // 1
    xBound = 1;            // 1
    yBound = 1;            // 1
    for(int count_y=0;count_y<3;count_y++){
      for(int count_x=0;count_x<3;count_x++){
        for (v = -yBound; v != yBound + adjustY; ++v) {   //-1, 0, 1
          for (u = -xBound; u != xBound + adjustX; ++u) { //-1, 0, 1
            if (x+count_x + u >= 0 && x+count_x + u < width && y+count_y + v >= 0 && y+count_y + v < height) {
              R = *(source_bitmap +
                    bytes_per_pixel * (width * (y+count_y + v) + (x+count_x + u)) + 2);
              G = *(source_bitmap +
                    bytes_per_pixel * (width * (y+count_y + v) + (x+count_x + u)) + 1);
              B = *(source_bitmap +
                    bytes_per_pixel * (width * (y+count_y + v) + (x+count_x + u)) + 0);
            } else {
              R = 0;
              G = 0;
              B = 0;
            }
            o_r.write(R);
            o_g.write(G);
            o_b.write(B);
            wait(1); //emulate channel delay
            //cout << "Now at " << sc_time_stamp() <<" TB "<< endl; //print current sc_time
          }
        }
      }
    }
```

圖二(Testbench)

可以看到上面的迴圈最外面的兩層是控制藍色框的 mean filter 的移動順序。

中間兩個迴圈是控制藍色框裡的 pixel 點，順序如圖一紅色數字的順序。

最內層是控制綠色框的 median filter 的順序。

中間四個迴圈總共會寫入 81 個 pixel 得到一個 mean pixel

```
for (unsigned int v = 0; v < MASK_Y; ++v)
{
  for (unsigned int u = 0; u < MASK_X; ++u)
  {
    val[0][v * 3 + u] = i_r.read();
    val[1][v * 3 + u] = i_g.read();
    val[2][v * 3 + u] = i_b.read();
    wait(1);
    // cout << "Now at " << sc_time_stamp() <<" MEAN "<< endl; //print current sc_time
  }
}

mergeSort(val[0], 0, val[0].size() - 1);
mergeSort(val[1], 0, val[1].size() - 1);
mergeSort(val[2], 0, val[2].size() - 1);
```

圖三(Median filte)

每讀進 9 個 pixel 後，經過 median filter 得到 median pixel

```
if (i == 4)
{
  mean_r += val[0][4] / 5;
  mean_g += val[1][4] / 5;
  mean_b += val[2][4] / 5;
}
else
{
  mean_r += val[0][4] / 10;
  mean_g += val[1][4] / 10;
  mean_b += val[2][4] / 10;
}

i = i + 1;
if (i == 9)
{
  // cout << "Now at " << sc_time_stamp() <<" 5164165165,i= "<< i<< endl; /
  // mid_r=MeanFiter(mean[0]);
  // mid_g=MeanFiter(mean[1]);
  // mid_b=MeanFiter(mean[2]);

  o_result_r.write(mean_r);
  o_result_g.write(mean_g);
  o_result_b.write(mean_b);
  i = 0;
  mean_r = 0, mean_g = 0, mean_b = 0;
  // cout << "Now at CPP85454123132 " << sc_time_stamp() <<" i "<< i<< endl
  // vector<vector<unsigned char>> mean(3, vector<unsigned char>(9, 0));
}
```

將 median pixel 平均後累加得到 mean pixel 並輸出。

```
user@ubuntu:~/ee6470/docker-images/EE6470/Hw1/Median and mean filters with FIFO channels/test_1/build$ make run
Consolidate compiler generated dependencies of target sobel
[ 80%] Built target sobel
[100%] Generating out.bmp

        SystemC 2.3.3-Accellera --- Mar  2 2023 02:09:42
        Copyright (c) 1996-2018 by all Contributors,
        ALL RIGHTS RESERVED
Image width=256, height=256

Info: /OSCI/SystemC: Simulation stopped by user.
Simulated time == 5308416 ns
[100%] Built target run
```

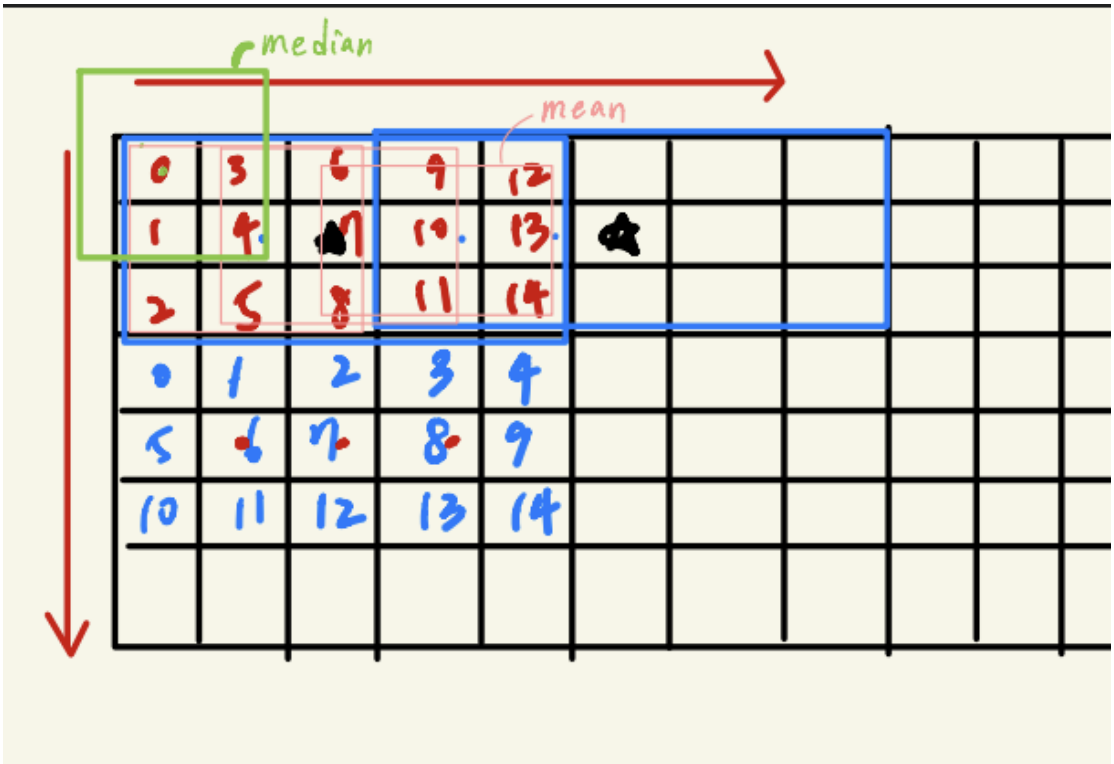
256*256 的圖片花了 5388416ns

```
user@ubuntu:~/ee6470/docker-images/EE6470/Hw1/Median and mean filters with FIFO channels/test_2/build$ make run
Consolidate compiler generated dependencies of target sobel
[ 80%] Built target sobel
[100%] Generating out.bmp

        SystemC 2.3.3-Accellera --- Mar  2 2023 02:09:42
        Copyright (c) 1996-2018 by all Contributors,
        ALL RIGHTS RESERVED
Image width=512, height=512

Info: /OSCI/SystemC: Simulation stopped by user.
Simulated time == 21233664 ns
[100%] Built target run
```

512*512 的圖片花了 21233664ns

上圖是比較圖

2.Prepare the input images



圖四

上圖是示意圖綠色框是做 median filter，粉色框是 mean filter，粉色框移動順序是做完 mean filter 後向右移一個，藍色框是 buffer 儲存 pixel 的數量，一共存 15 個 pixel，藍色框移動順序是中心點向右移 3 格，做完整排後，往下移一格。

因為 systemC module 先做 median filter 再做 mean filter，所以我設計需要先算出 9 個經過 median filter 所得出的 median value pixel，再將這 9 個 pixel 加起來平均後，得到 1 個 mean value pixel 後，輸出到 Testbench。但我至

這邊與上一題的儲存順序不太一樣，buffer 的 index 是由上到下後由左到右的順序去儲存，因讀入 pixel 是由左到右後由上到下，舉例來說:讀入 pixel 為

先讀第一行 buffer index 為:0、3、6、9、12

再讀第二行 buffer index 為:1、4、7、10、13

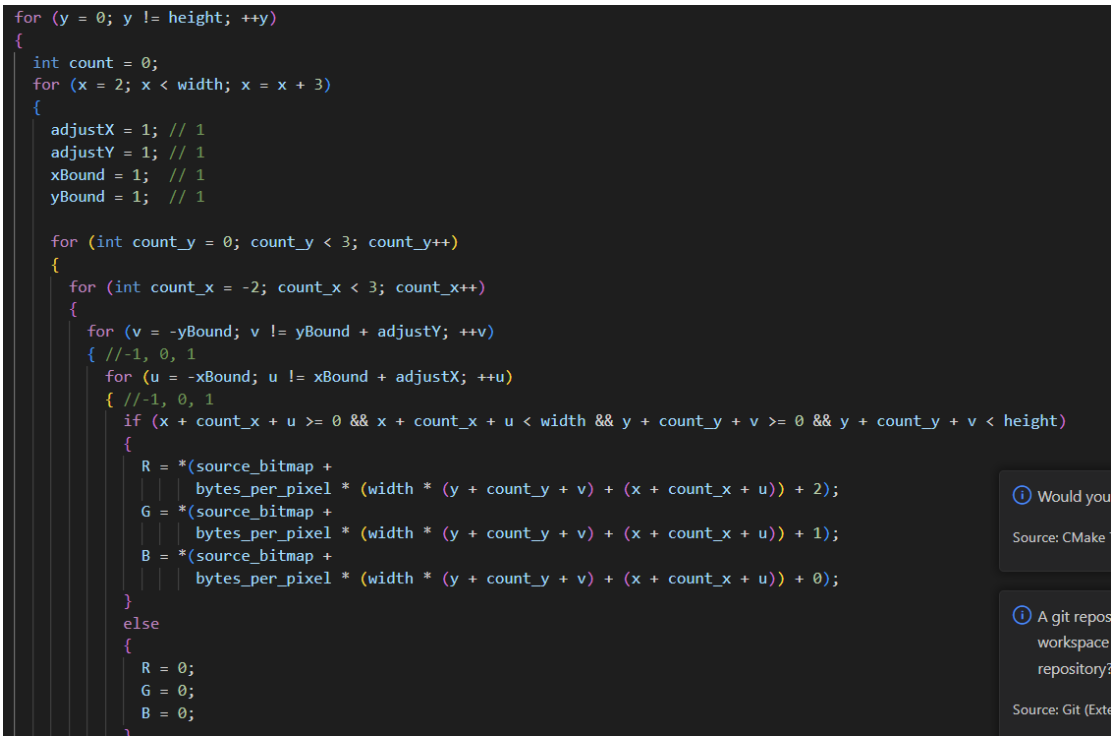
再讀第三行 buffer index 為:2、5、8、11、14

當讀到第三行的 8 時，可以看到第一個 mean filter 出現了可以算第一個 mean pixel，可以直接讀 buffer[0]~buffer[8]的 mean pixel。

當讀到第三行的 11 時，可以看到第二個 mean filter 出現了可以算第二個 mean pixel，可以直接讀 buffer[3]~buffer[11]的 mean pixel。

當讀到第三行的 14 時，可以看到第三個 mean filter 出現了可以算第三個 mean pixel，可以直接讀 buffer[6]~buffer[14]的 mean pixel。

每做出一個 mean pixel 就輸出一次，每輸出三個 pixel 需要讀 135 個 pixel。



```
for (y = 0; y != height; ++y)
{
  int count = 0;
  for (x = 2; x < width; x = x + 3)
  {
    adjustX = 1; // 1
    adjustY = 1; // 1
    xBound = 1;  // 1
    yBound = 1;  // 1

    for (int count_y = 0; count_y < 3; count_y++)
    {
      for (int count_x = -2; count_x < 3; count_x++)
      {
        for (v = -yBound; v != yBound + adjustY; ++v)
        { //-1, 0, 1
          for (u = -xBound; u != xBound + adjustX; ++u)
          { //-1, 0, 1
            if (x + count_x + u >= 0 && x + count_x + u < width && y + count_y + v >= 0 && y + count_y + v < height)
            {
              R = *(source_bitmap +
                    bytes_per_pixel * (width * (y + count_y + v) + (x + count_x + u)) + 2);
              G = *(source_bitmap +
                    bytes_per_pixel * (width * (y + count_y + v) + (x + count_x + u)) + 1);
              B = *(source_bitmap +
                    bytes_per_pixel * (width * (y + count_y + v) + (x + count_x + u)) + 0);
            }
            else
            {
              R = 0;
              G = 0;
              B = 0;
```

圖五(Testbench)

可以看到上面的迴圈最外面的兩層是控制藍色框的 mean filter 的移動順序，X 方向是移動 3 格，Y 是一格。

中間兩個迴圈是控制藍色框裡的 pixel 點，順序如圖四粉紅色數字的順序。

最內層是控制綠色框的 median filter 的順序。

中間四個迴圈總共會寫入 135 個 pixel 得到三個 mean pixel

```
vector<vector<unsigned char>> buffer(3, vector<unsigned char>(15, 0));
```

上式是我所加的 buffer，一共可以存 15 個 pixel。

```
user@ubuntu:~/ee6470/docker-images/EE6470/Hw1/Data buffers/test_1/build$ make run
Consolidate compiler generated dependencies of target sobel
[ 80%] Built target sobel
[100%] Generating out.bmp

        SystemC 2.3.3-Accellera --- Mar  2 2023 02:09:42
        Copyright (c) 1996-2018 by all Contributors,
        ALL RIGHTS RESERVED
Image width=256, height=256

Info: /OSCI/SystemC: Simulation stopped by user.
Simulated time == 3002880 ns
```

256*256 的圖片花了 3002880ns

```
user@ubuntu:~/ee6470/docker-images/EE6470/Hw1/Data buffers/test_2/build$ make run
Consolidate compiler generated dependencies of target sobel
[ 80%] Built target sobel
[100%] Generating out_1.bmp

        SystemC 2.3.3-Accellera --- Mar  2 2023 02:09:42
        Copyright (c) 1996-2018 by all Contributors,
        ALL RIGHTS RESERVED
Image width=512, height=512

Info: /OSCI/SystemC: Simulation stopped by user.
Simulated time == 12011520 ns
```

512*512 的圖片花了 12011520ns

結論:

第一題沒有加 buffer 的架構會比有加 buffer 的架構更花時間，因為會需要讀到重覆算過得 pixel，第一題算 1 個 mean pixel 需要讀 81 個 pixel，第二題 135 個 pixel 可以得到 3 個 mean pixel，也就是每 45 個 pixel 可以得到 1 個 mean pixel，可以看出第一題要多讀兩倍的 pixel 才可以得到一個 mean pixel，所以 data transfer between testbench and filter module 可以少一半的傳輸量，所花的時間也可以少接近一倍。