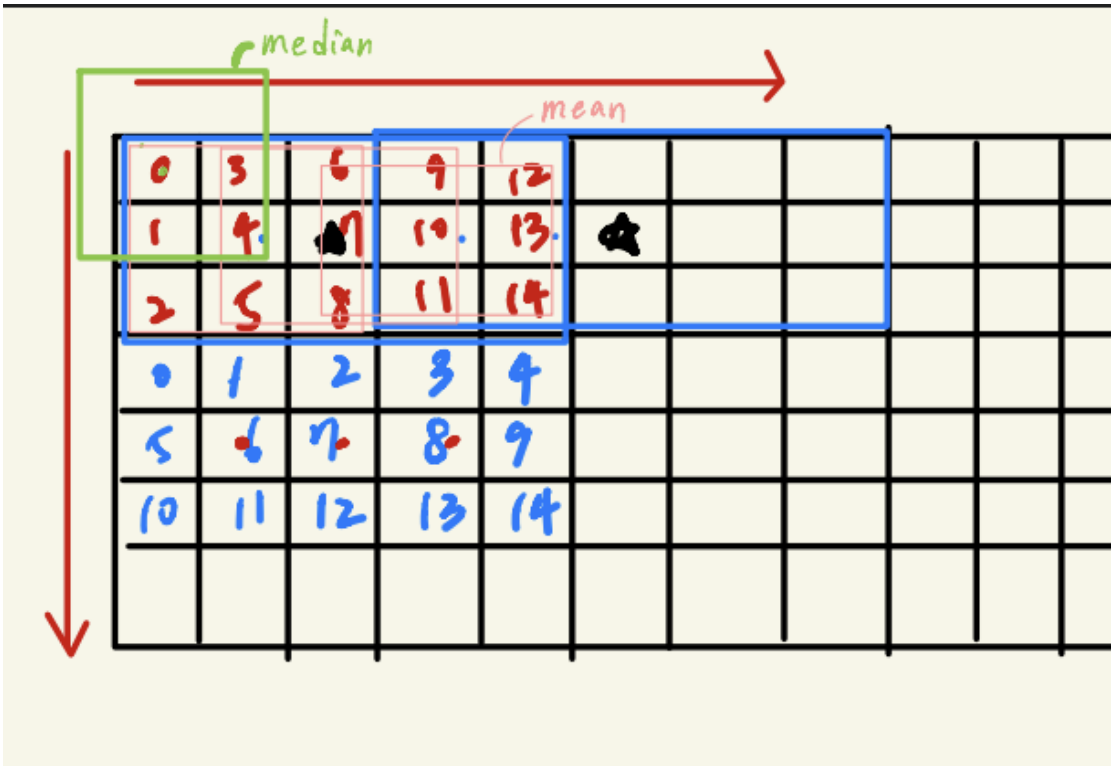
E111064503 通訊所 吳紹齊 HW2

https://github.com/cowboy35927/ESL/tree/main/Hw2

演算法介紹:



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。

1. Median and mean filters with TLM interface

```
Testbench tb("tb");
Median_MeanFilter sobel_filter("sobel_filter");
tb.initiator.i_skt(sobel_filter.t_skt);
```

上圖是 main.cpp 初始化，最下行是 bind initiator socket to target socket。

```
    data.uc[0] = R;
    data.uc[1] = G;
    data.uc[2] = B;
    mask[0] = 0xff;
    mask[1] = 0xff;
    mask[2] = 0xff;
    mask[3] = 0;
    initiator.write_to_socket(SOBEL_FILTER_R_ADDR, mask, data.uc, 4);
    wait(1 * CLOCK_PERIOD, SC_NS);
    // cout << "Now at " << sc_time_stamp() << " TB " << endl; // print current sc_t
  }
}
```

```
int Initiator::write_to_socket(unsigned long int addr, unsigned char mask[],
                               unsigned char wdata[], int dataLen) {
  // Set up the payload fields. Assume everything is 4 bytes.
  trans.set_write();
  trans.set_address((sc_dt::uint64)addr);

  trans.set_byte_enable_length((const unsigned int)dataLen);
  trans.set_byte_enable_ptr((unsigned char *)mask);

  trans.set_data_length((const unsigned int)dataLen);
  trans.set_data_ptr((unsigned char *)wdata);

  // Transport.
  do_trans(trans);

  /* For now just simple non-zero return code on error */
  return trans.is_response_ok() ? 0 : -1;

} // writeUpcall()
```

write_to_socket 函式會我們設置了 payload。 寫入 SOBEL_FILTER_R_ADDR 地址。

```
void Initiator::do_trans(tlm::tlm_generic_payload &trans) {
  sc_core::sc_time dummyDelay = sc_core::SC_ZERO_TIME;

  // Call the transport and wait for no time, which allows the thread to yield
  // and others to get a look in!

  i_skt->b_transport(trans, dummyDelay);
  //wait(sc_core::SC_ZERO_TIME);
  wait(dummyDelay);

} // do_trans()
```

準備好有效 payload 後，我們調用阻塞傳輸函數來發送數據。do_trans 中的 b_transport()，然後我們調用 wait(delay) 以 target 返回的 dummyDelay 值提前 Systemc 時間。

```
case tlm::TLM_WRITE_COMMAND:
  switch (addr) {
  case SOBEL_FILTER_R_ADDR:
    if (mask_ptr[0] == 0xff) {
      i_r.write(data_ptr[0]);
      //cout << "Now at " << sc_time_stamp() << " R: "<<int(data_ptr[0]) << endl; // print current sc_time
    }
    if (mask_ptr[1] == 0xff) {
      i_g.write(data_ptr[1]);
      //cout << "Now at " << sc_time_stamp() << " G: "<<int(data_ptr[1]) << endl; // print current sc_time
    }
    if (mask_ptr[2] == 0xff) {
      i_b.write(data_ptr[2]);
      //cout << "Now at " << sc_time_stamp() << " B: "<<int(data_ptr[2]) << endl; // print current sc_time
    }
    break;
  default:
    std::cerr << "Error! Median_MeanFilter::blocking_transport: address 0x"
              << std::setfill('0') << std::setw(8) << std::hex << addr
              << std::dec << " is not valid" << std::endl;
    break;
  }
```

上圖就是當 address 是 SOBEL_FILTER_R_ADDR 會執行 tlm::TLM_WRITE_COMMAND 的指令，會將 i_r、i_g、i_b 分別寫入資料。

```
  if (count_y == 2 && count_x == 0)
  {
    bool done=false;
    int output_num=0;
    while(!done){
      initiator.read_from_socket(SOBEL_FILTER_CHECK_ADDR, mask, data.uc, 4);
      output_num = data.sint;
      if(output_num>0) done=true;
    }
    //wait(10 * CLOCK_PERIOD, SC_NS);
    initiator.read_from_socket(SOBEL_FILTER_RESULT_ADDR, mask, data.uc, 4);
    //cout << "Now at " << sc_time_stamp() << " R: "<< data.uint/(256*256)<< endl; // print current sc_time
    //cout << "Now at " << sc_time_stamp() << " G: "<<(data.uint/(256))%256<< endl; // print current sc_time
    //cout << "Now at " << sc_time_stamp() << " B: "<< (data.uint)%(256)  << endl; // print current sc_time
    //total = data.sint_b;
    //if (i_result_b.num_available() == 0)wait(i_result_b.data_written_event());
    *(target_bitmap + bytes_per_pixel * (width * y + x - 2) + 2) = data.uint/(256*256);
    *(target_bitmap + bytes_per_pixel * (width * y + x - 2) + 1) = (data.uint/(256))%256;
    *(target_bitmap + bytes_per_pixel * (width * y + x - 2) + 0) = (data.uint)%(256);

    wait(1 * CLOCK_PERIOD, SC_NS);
    count += 1;

  }
```

read_from_socket 函式會我們設置了 payload。 寫入 SOBEL_FILTER_R_ADDR 地址。

```
int Initiator::read_from_socket(unsigned long int addr, unsigned char mask[],
                                unsigned char rdata[], int dataLen) {
  // Set up the payload fields. Assume everything is 4 bytes.
  trans.set_read();
  trans.set_address((sc_dt::uint64)addr);

  trans.set_byte_enable_length((const unsigned int)dataLen);
  trans.set_byte_enable_ptr((unsigned char *)mask);

  trans.set_data_length((const unsigned int)dataLen);
  trans.set_data_ptr((unsigned char *)rdata);

  // Transport.
  do_trans(trans);
  //cout<<int(rdata[0])<<endl;
  //cout<<int(rdata[1])<<endl;
 // cout<<int(rdata[2])<<endl;
  /* For now just simple non-zero return code on error */
  return trans.is_response_ok() ? 0 : -1;

} // read_from_socket()
```

write_to_socket 函式會我們設置了 payload。 寫入 SOBEL_FILTER_CHECK_ADDR 地址。

```
case SOBEL_FILTER_CHECK_ADDR:
  buffer.uint = o_result.num_available();
  break;
```

上圖就是當 address 是 SOBEL_FILTER_R_ADDR 會執行 tlm::TLM_WRITE_COMMAND 的指令，如果 o_result.num_available()不=0 時，就會跳出迴圈，

```
case SOBEL_FILTER_RESULT_ADDR:
  a=o_result.read();
  buffer.uint = a;
  break;
```

寫資料進到 buffer.uint 後，在寫入 target_bitmap。

```
    *(target_bitmap + bytes_per_pixel * (width * y + x - 2) + 2) = data.uint/(256*256);
    *(target_bitmap + bytes_per_pixel * (width * y + x - 2) + 1) = (data.uint/(256))%256;
    *(target_bitmap + bytes_per_pixel * (width * y + x - 2) + 0) = (data.uint)%(256);

    wait(1 * CLOCK_PERIOD, SC_NS);
```

Target socket module 的 wait:

```
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 * CLOCK_PERIOD, SC_NS);
    // cout << "Now at " << sc_time_stamp() << " MEAN " << endl; // print current sc_time
  }
}
```

在每 read 完 rgb 三個 pixel 後，wait 一次

```
//o_result_b.write(mid_b);
total=(mid_r*256*256)+(mid_g*256)+(mid_b);
o_result.write(total);
wait(1* CLOCK_PERIOD, SC_NS);
i = 0;
```

在每 write 完 o_result 後，wait 一次

2. Median and mean filters with quantum keeper

```
tlm_utils::tlm_quantumkeeper m_qk;
```

定義 quantum keeper

```
m_qk.set_global_quantum( sc_time(100, SC_NS) );
m_qk.reset();
```

b_transport() 返回的每個延遲值，我們只需通過調用 inc(delay) 來增加本地時間並檢查本地時間是否超過上述量程（100ns)。 當本地時間達到量程（need_sync() 為真）時，我們調用 sync() ，它將調用 wait(local time) 和 reset() 本地時間為 0。

```
    R = 0;
    G = 0;
    B = 0;
  }
  delay = m_qk.get_local_time();
  data.uc[0] = R;
  data.uc[1] = G;
  data.uc[2] = B;
  mask[0] = 0xff;
  mask[1] = 0xff;
  mask[2] = 0xff;
  mask[3] = 0;

  initiator.write_to_socket(SOBEL_FILTER_R_ADDR, mask, data.uc, 4 ,delay);
  // Increment local time with delay returned from b_transport()
  m_qk.inc( delay );
  // Check if synchronize is necessary
  if (m_qk.need_sync()) m_qk.sync();

  //wait(1 * CLOCK_PERIOD, SC_NS);
```

當本地超過 100ns 後，會進行同步。

```
void Initiator::do_trans(tlm::tlm_generic_payload &trans, sc_time& delay ) {
  //delay=sc_time(1, SC_NS);

  // Call the transport and wait for no time, which allows the thread to yield
  // and others to get a look in!

  i_skt->b_transport(trans, delay);
  //wait(sc_core::SC_ZERO_TIME);
  //wait(dummyDelay);

} // do_trans()
```

Initiator.cpp 的 do_trans()可以把 wait 去掉。

還有 Median_MeanFilter.cpp 的 wait 都可以拿掉。

所以原本每 1ns 就會 call wait，現在是每 100ns call wait 大幅減少 call wait 的次數，以達到節省 simulation 的時間。

3. Median and mean filters with TLM interconnect

唯一的區別是，現在 Testbench 通過 bus 模塊 SimpleBus 向 Median_MeanFilter 寫入像素和讀取結果。 我們只能關注 bus 如何將 transactions from an initiator to a target module.。

```
Testbench tb("tb");
SimpleBus<1, 1> bus("bus");
bus.set_clock_period(sc_time(CLOCK_PERIOD, SC_NS));
Median_MeanFilter sobel_filter("sobel_filter");
tb.initiator.i_skt(bus.t_skt[0]);
bus.setDecode(0, SOBEL_MM_BASE, SOBEL_MM_BASE + SOBEL_MM_SIZE - 1);
bus.i_skt[0](sobel_filter.t_skt);
```

在代碼中，我們首先通過“SimpleBus<1, 1>”模板參數實例化一個具有一個 target socket 和一個 initiator socket 的 bus。 initiator socket 通過“tb.initiator.i_skt(bus.t_skt[0]);”連接到 Testbench，其中“bus.t_skt[0]”是 bus 的第一個（也是唯一一個）initiator socket。initiator socket 通過“bus.i_skt[0](sobel_filter.t_skt);”連接到 Median_MeanFilter，其中“bus.i_skt[0]”是 bus 上的第一個（也是唯一一個）initiator socket。

```
// Sobel filter Memory Map
// Used between SimpleBus & SobelFilter
const int SOBEL_MM_BASE = 0x90000000;
const int SOBEL_MM_SIZE = 0x0000000C;
const int SOBEL_MM_MASK = 0x0000000F;
```

記憶體位置和大小

```
void setDecode(int portId, Addr lo, Addr hi) {
  if (portId >= static_cast<int>(no_of_targets())) {
    printf("Error configuring TLM decoder %s: portId (%d) cannot be greater "
           "than the number of targets (%d)\n",
           memory_map_name().c_str(), portId, no_of_targets());
    return;
    abort();
  }
  if (lo > hi) {
    printf("Error configuring TLM decoder %s: lo (0x%llX) cannot be greater "
           "than hi (0x%llX)\n",
           memory_map_name().c_str(), lo, hi);
    return;
  }
  icmPortMapping *decode = new icmPortMapping(lo, hi);
  decode->setNext(decodes[portId]);
  decodes[portId] = decode;
 }
};
```

使用“bus.setDecode()”將全局內存映射地址“SOBEL_MM_BASE”設置為端口 ID“0”。這意味著對該地址的任何 blocking transport 調用都將被發送到端口 ID 為“0”的 initiator socktet。同樣在將總線 blocking transport 調用轉發到端口 ID“0”時，將從轉發地址中減去“SOBEL_MM_BASE”。例如，如果我們想通過 Bus 調用“SOBEL_MM_BASE+0x04”處的 blocking transport，目標（Median_MeanFilter）將僅接收到“0x04”地址

```
initiator.write_to_socket(SOBEL_MM_BASE + SOBEL_FILTER_R_ADDR, mask,data.uc, 4,delay);
initiator.read_from_socket(SOBEL_MM_BASE +SOBEL_FILTER_CHECK_ADDR, mask, data.uc, 4,delay);
initiator.read_from_socket(SOBEL_MM_BASE +SOBEL_FILTER_RESULT_ADDR, mask, data.uc, 4,delay);
```

Initiator 的讀寫的地址需是 SOBEL_MM_BASE+指令地址。

下兩圖是計算對目標模塊的 read/write 次數，我在 Simplebus.h 的 initiatorBTransport 放置 counter，來計算 read/write 總數。
我還有在 initiator.cpp 分別在 read_from_socket 和 write_to_socket 放置 counter 去分別計算 read 和 write 的次數來做驗證。

```
class Initiator : public sc_module {
public:
  tlm_utils::simple_initiator_socket<Initiator> i_skt;
  tlm_utils::tlm_quantumkeeper m_qk;
  SC_HAS_PROCESS(Initiator);
  Initiator(sc_module_name n);
  int count_read=0;
  int count_write=0;
```

```
void initiatorBTransport(int SocketId, transaction_type &trans,
                         sc_core::sc_time &t) {
  Addr orig = trans.get_address();
  Addr offset;
  int portId = getPortId(orig, offset);

  if (portId < 0) {
    std::cout << "ERROR: " << name() << ": initiatorBTransport()"
              << ": Invalid (undefine memory mapped) address == "
              << tshsu::print(trans.get_address()) << std::endl;
    assert(false);
  }

  if (m_trace) {
    printf("TLM: %s decode:0x%llX -> i_skt[%d]\n", name(), orig, portId);
  }
  // It is a static port ?
  initiator_socket_type *decodeSocket = &i_skt[portId];
  if (m_is_address_masked) {
    trans.set_address(offset);
  }
  t = t + delay(trans); //add interconnect delay
  (*decodeSocket)->b_transport(trans, t);
  counter+=1;
  //cout<<"counter:"<<counter<<endl;
}
```

```
tb.read_bmp(argv[1]);
sc_start();
std::cout << "Simulated time == " << sc_core::sc_time_stamp() << std::endl;
cout<<"Simplebus:"<<endl;
cout<<"the number of read/write to the target module:"<<bus.counter<<endl;
cout<<"Initiator socket:"<<endl;
cout<<"read:"<<tb.initiator.count_read<<endl;
cout<<"write:"<<tb.initiator.count_write<<endl;
cout<<"result:"<<tb.initiator.count_read+tb.initiator.count_write<<endl;
tb.write_bmp(argv[2]);
```

最後印出結果。

結果和比較:

256*256 的圖片:

(1)

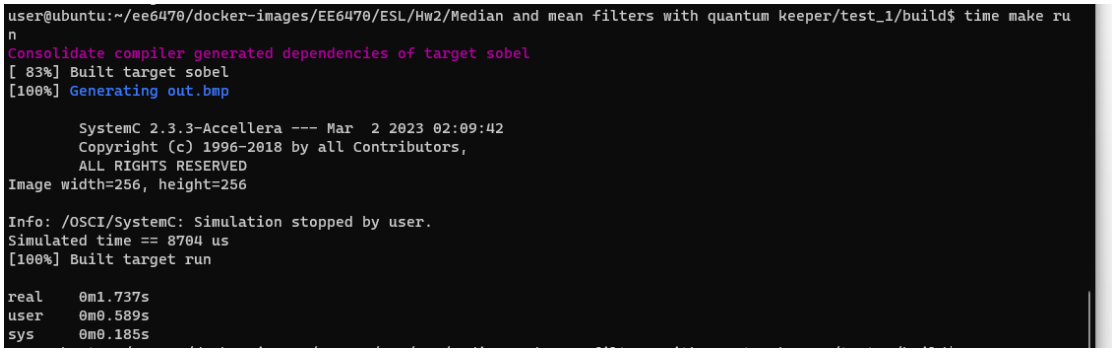
```
user@ubuntu:~/ee6470/docker-images/EE6470/ESL/Hw2/Median and mean filters with TLM interface/test_1/build$ time make run
Consolidate compiler generated dependencies of target sobel
[ 83%] Built target sobel
[100%] Generating out.bmp

        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
[100%] Built target run

real    0m2.916s
user    0m1.603s
sys     0m0.215s
```

(2)



(3)

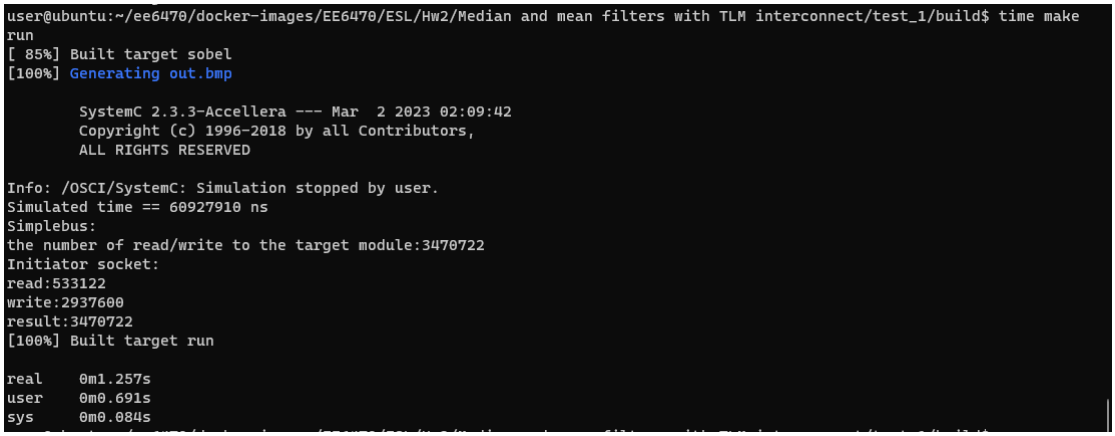
Simplebus 的 read/write 總和次數為 3470722

Initiator 的 read 次數為 533122

Initiator 的 write 次數為 2937600

Initiator 的 read0/write 次數為 3470722

兩者一致

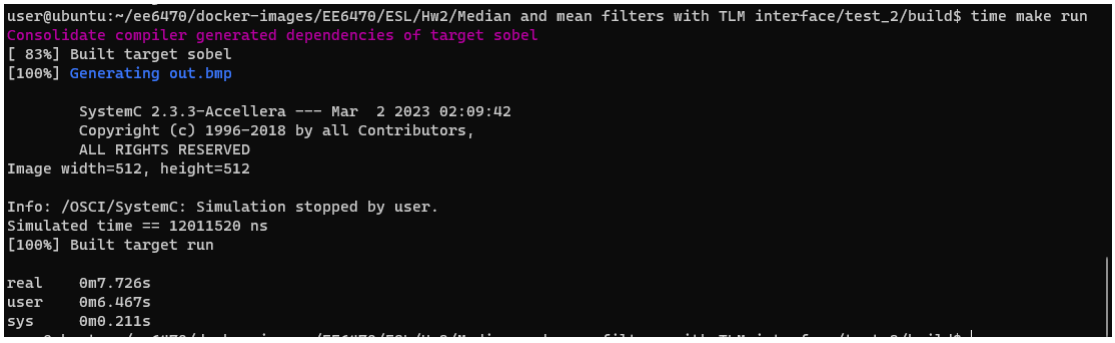


執行時間: (1) > (2) >= (3)

(1)需要一直 call wait，而(2)使用 quantum keeper 減少了 call wait 的次數達成減少執行時間的功效，最後(3)使用 quantum keeper 減少 call wait 的次數還有指定 address。

512*512 的圖片:

(1)



(2)

```
user@ubuntu:~/ee6470/docker-images/EE6470/ESL/Hw2/Median and mean filters with quantum keeper/test_2/build$ time make ru
n
Consolidate compiler generated dependencies of target sobel
[ 83%] 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 == 34816 us
[100%] Built target run

real    0m3.410s
user    0m2.163s
sys     0m0.189s
```

(3)

```
user@ubuntu:~/ee6470/docker-images/EE6470/ESL/Hw2/Median and mean filters with TLM interconnect/test_2/build$ time make
run
Consolidate compiler generated dependencies of target sobel
[ 85%] 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

Info: /OSCI/SystemC: Simulation stopped by user.
Simulated time == 243711910 ns
Simplebus:
the number of read/write to the target module:13882882
Initiator socket:
read:2132482
write:11750400
result:13882882
[100%] Built target run

real    0m3.688s
user    0m2.412s
sys     0m0.197s
```

執行時間: (1) > (2) >= (3)

(1)需要一直 call wait，而(2)使用 quantum keeper 減少了 call wait 的次數達成減少執行時間的功效，最後(3)使用 quantum keeper 減少 call wait 的次數還有指定 address。