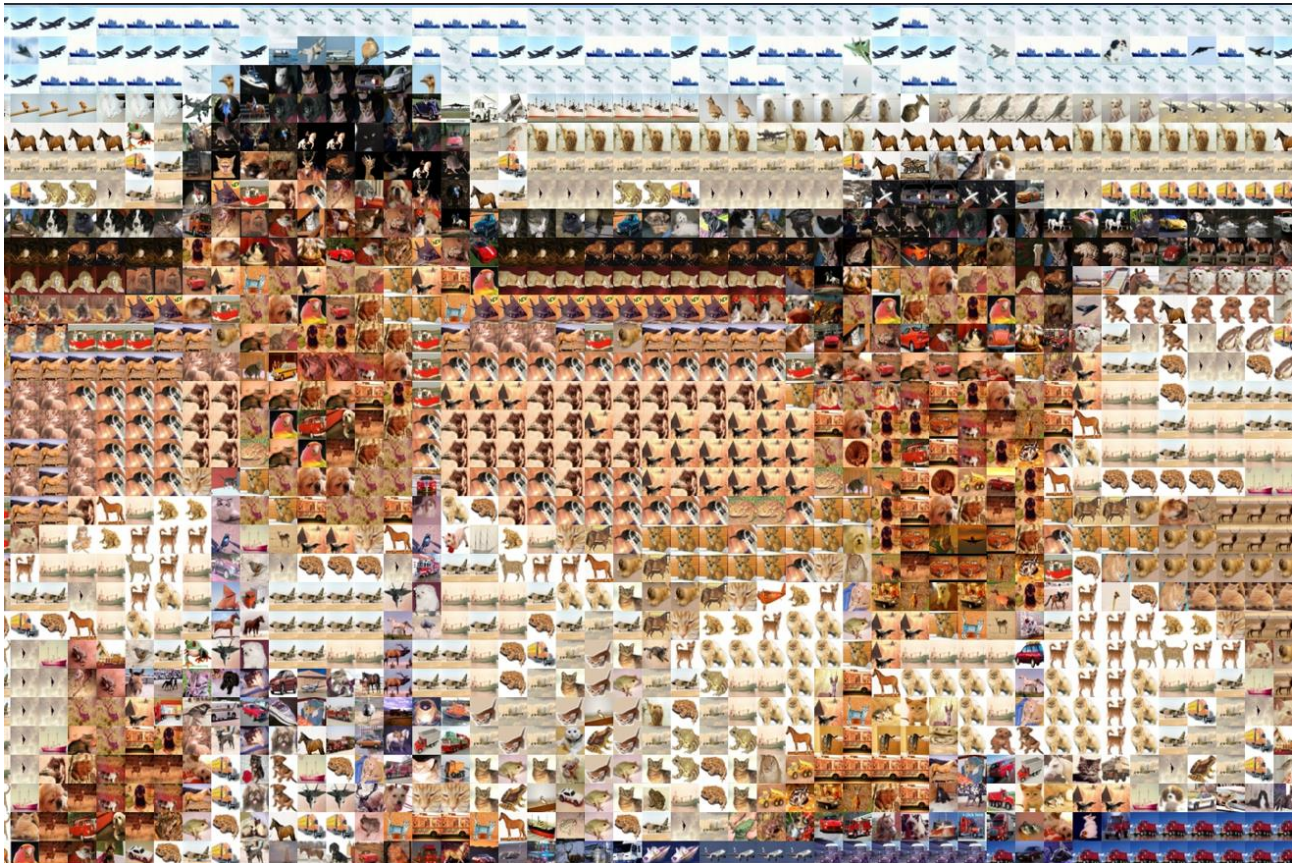


# 計算機結構期末專題

## Photo Mosaic(蒙太奇拼貼)



組員 1: 何祁恩 (0811562)

組員 2: 鄭文洋(109511272)



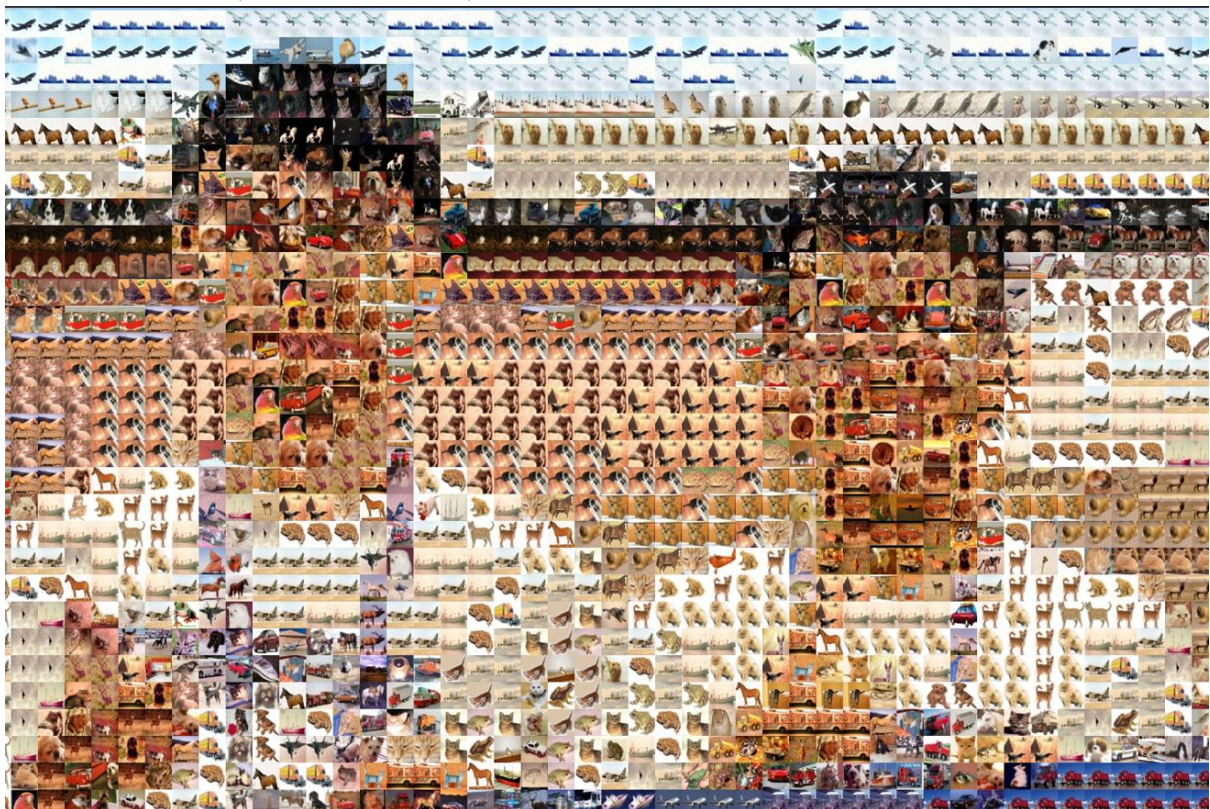
# 題目：Photo Mosaic(蒙太奇拼貼)

這個蒙太奇拼貼的演算法為使用許多張小圖來盡可能的拼貼成大圖的樣子。換言之，是透過許多張小圖來構成大圖中的像素。

舉例來說，給定目標大圖：



使用許多小圖(cifar10 資料集)構成的結果為：



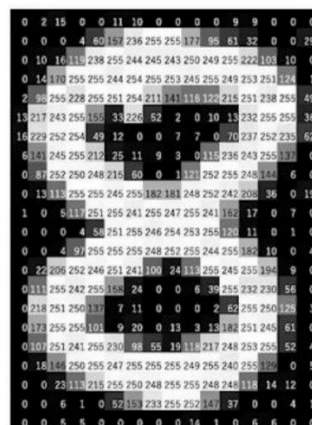
我們在這次專題使用的小圖來自有名公開資料集 cifar10。Cifar10 是電腦視覺領域中廣為人知的資料集，圖片分成飛機、汽車、鳥、貓、鹿、狗、青蛙、馬、船、卡車。大多是拿來訓練多物件辨識的模型。在這個計算機結構的期末專題中，我們使用這個資料集來當成構成大圖的小圖。亦即使用 cifar10 中的  $32 \times 32 \times 3$  的小圖，來拼貼出任何給定的大圖。

# 題目: Photo Mosaic 演算法

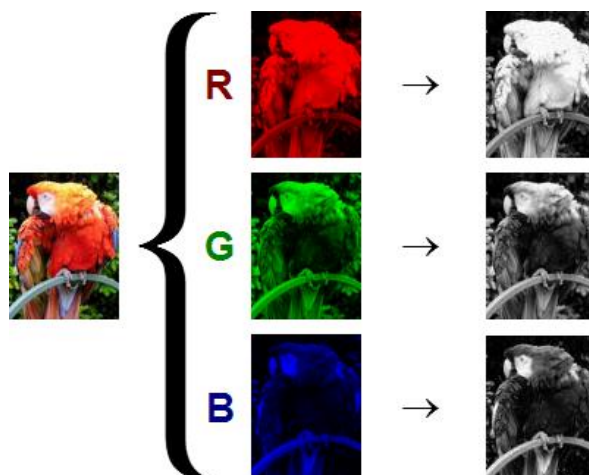
首先，先介紹一下數位影像中是如何構成一張圖片的？

在電腦圖形中表示灰階影像是非常簡單的:使用一個二維陣列，裡面每個數值皆落在 0 到 255 的範圍(8-bit 位元深度為例)。若是彩色圖片，則將其二維陣列擴展為有 RGB 三個 channel 的三維陣列即可。

舉例來說，常見的 mnist dataset:



而至於彩色的圖片，僅需要將原本的二維陣列擴展維三維陣列，第三維分別代表了該圖的紅色、綠色、藍色的通道。更簡單來說，為表示該圖有多紅、有多綠、有多藍，以下為彩圖三通道的示意圖:



這個演算法其實不難。可以把 cifar10 中的小圖片想像成是大圖片中的一個像素。因為遠遠看一個小圖片，其實看不太到細節，僅能大約看得出其代表的顏色而已。將大圖切為許多個小圖，而後將小圖與所有 cifar10 中的候選小圖比較，便可以找到最適合的候選小圖，將其取代大途中的像素值。

# Photo Mosaic 演算法(續)

那究竟如何比較候選的小圖，來選擇哪張小圖比較適合取代給定的大圖中的子圖呢?在這個演算法，我們去計算大圖中子圖的紅色通道的像素平均值  $r_{avg\_grid}$ 、綠色通道的像素平均值  $g_{avg\_grid}$ 、藍色通道的像素平均值  $b_{avg\_grid}$ ，並且與 cifar10 中的所有候選小圖的紅色通道的像素平均值  $r_{avg\_candidate}$ 、綠色通道的像素平均值  $g_{avg\_candidate}$ 、藍色通道的像素平均值  $b_{avg\_candidate}$  來比較。詳細數學式如下：

$$\arg \min_{1 \leq i \leq \#images \text{ in CIFAR10}} \left( \sqrt{(r_{avg\_grid} - r_{avg\_candidate})^2 + (g_{avg\_grid} - g_{avg\_candidate})^2 + (b_{avg\_grid} - b_{avg\_candidate})^2} \right)$$

由上面的算式可以知道會去選擇在 CIFAR10 資料集中，RGB 三通道平均值相差最小的小圖來代表大圖中的子圖。而大圖中會有許多的子圖，需要做相同的事情，彼此間是可以被平行進行的，並不會有 data hazard 或是 race condition 的情況發生，因此這個問題便可以被定義為：

$\forall sub\ img_s_i \in target\ img$ , (對大圖中的每個子圖)

$\forall candidate\ img_s_j \in CIFAR\ 10$ , (在所有候選小圖中)

$$find \left( \arg \min_{1 \leq j \leq \#images \text{ in CIFAR10}} \left( \sqrt{(r_{avg\_grid_i} - r_{avg\_candidate_j})^2 + (g_{avg\_grid_i} - g_{avg\_candidate_j})^2 + (b_{avg\_grid_i} - b_{avg\_candidate_j})^2} \right) \right)$$

$replace(sub\ img_s_i, candidate\ img_j)$  (尋找 RGB 平均值相差最小的候選圖取代)

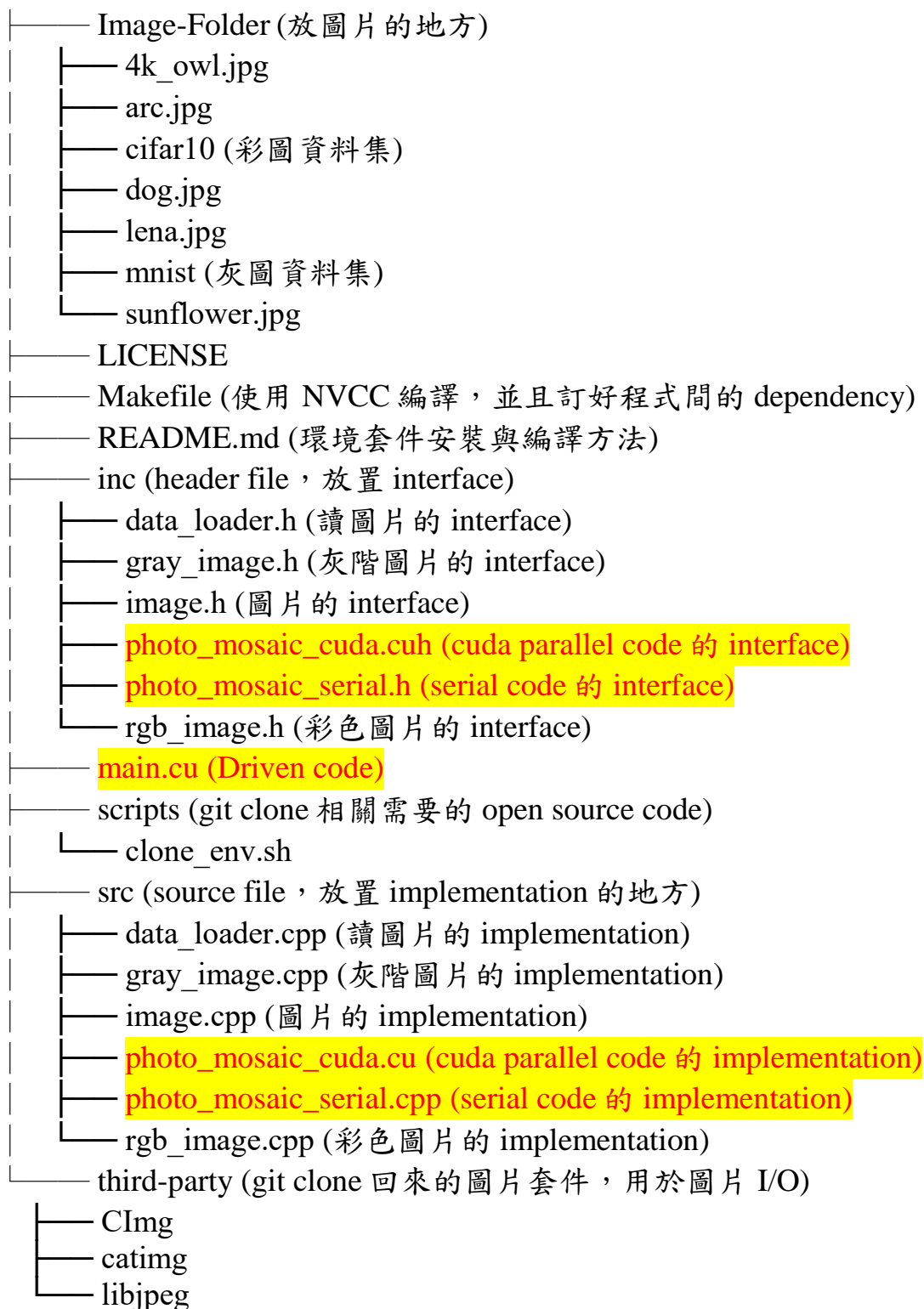
所以以下是這個演算法的流程：

1. 將大圖與候選小圖全部讀進來
2. 計算大圖所有子圖與候選小圖的 RGB 三通道各別的平均值
3. 針對所有大圖，找到最為相符的小圖
4. 將最為相符的小圖取代掉大圖的部分

在這之中，我將使用 CUDA 針對所有大圖，找到最為相符的小圖的步驟使用不同的 block 與 thread 來進行平行運算。使得此演算法最為核心的部分可以運行的更加快速。



## 檔案說明:



整份 project 是以 c/c++/cuda 撰寫而成，並且使用物件導向的風格撰寫而成，在這份 project 中，rgb\_image 與 gray\_image class 繼承 base class image。二維陣列與三維陣列的 pixels 則是存在 gray\_image 與 rgb\_image 中。

# Serial Code

- 計算大圖中所有的子圖的 RGB 三個 channel 的平均值，並存在陣列中

```
44 // calculate the r_avg, g_avg, b_avg of the grid of the target image
45 int target_grid_idx = 0;
46 for(int row = 0; row < target_img.get_h() - SUB_PIC_SIZE; row += SUB_PIC_SIZE){
47     for(int col = 0; col < target_img.get_w() - SUB_PIC_SIZE; col += SUB_PIC_SIZE){
48         double r_avg_target = 0;
49         double g_avg_target = 0;
50         double b_avg_target = 0;
51         for(int i = row; i < row + SUB_PIC_SIZE; i++){
52             for(int j = col; j < col + SUB_PIC_SIZE; j++){
53                 r_avg_target += target_img.pixels[i][j][0];
54                 g_avg_target += target_img.pixels[i][j][1];
55                 b_avg_target += target_img.pixels[i][j][2];
56             }
57         }
58         r_avg_target /= SUB_PIC_SIZE * SUB_PIC_SIZE;
59         g_avg_target /= SUB_PIC_SIZE * SUB_PIC_SIZE;
60         b_avg_target /= SUB_PIC_SIZE * SUB_PIC_SIZE;
61         r_avg_target_grid[target_grid_idx] = r_avg_target;
62         g_avg_target_grid[target_grid_idx] = g_avg_target;
63         b_avg_target_grid[target_grid_idx] = b_avg_target;
64         target_grid_idx++;
65     }
66 }
```

在這邊也可以寫一個 GPU kernel 來加速計算，但是考量到這邊並不是主要耗費時間的地方。根據 Amdahl's law，針對這邊加速對整體程式的效益並不高。額外的 device memory 的 allocation 與 copy 的 overhead 會蠻大的。破壞 spatial locality 的 parallel code 可能會使得整體的效能不增反減。因此這個部分我們留給 CPU 計算。

- 將小圖中在讀檔就已經算好的 RGB 三通道的平均值存在陣列中

```
68 for(int i = 0; i < num_candidate_imgs; i++){
69     r_avg_candidate[i] = candidate_imgs[i].r_avg;
70     g_avg_candidate[i] = candidate_imgs[i].g_avg;
71     b_avg_candidate[i] = candidate_imgs[i].b_avg;
72 }
```

在一開始使用 data loader 將 cifar10 中所有小圖 load 進來的時候便已經計算過 RGB 三通道的平均值了。因此這邊僅需要把它存成陣列，使得後續運算做 vectorize 可以較為方便與輕鬆。

- 針對大圖中的子圖找到最適合的小圖，紀錄下該小圖的索引

```
78     for(int i = 0; i < tile_width * tile_height; i++){
79         double min_diff = DBL_MAX;
80         for(int j= 0; j < num_candidate_imgs; j++){
81             double r_diff = r_avg_target_grid[i] - r_avg_candidate[j];
82             double g_diff = g_avg_target_grid[i] - g_avg_candidate[j];
83             double b_diff = b_avg_target_grid[i] - b_avg_candidate[j];
84             double diff = sqrt(r_diff * r_diff + g_diff * g_diff + b_diff * b_diff);
85             if(diff < min_diff){
86                 min_idx[i] = j;
87                 min_diff = diff;
88             }
89         }
90     }
```

這邊就是最核心花費最多時間的地方。外面的 for 迴圈是 iterate 大圖的所有子圖，而內部的 for 迴圈則是 iterate CIFAR10 中所有的候選小圖。針對不同的子圖，是需要獨立找最適合的小圖。因此這邊是可以被平行化來增加 performance 的。在後面我將會描述我們如何撰寫 GPU kernel 來平行化這個部分的程式。

- 將大圖中的子圖以前一步找到的最適合的小圖取代

```
99     int count = 0;
100     for(int row = 0; row < target_img.get_h() - SUB_PIC_SIZE; row += SUB_PIC_SIZE){
101         for(int col = 0; col < target_img.get_w() - SUB_PIC_SIZE; col += SUB_PIC_SIZE){
102             for(int i = row; i < row + SUB_PIC_SIZE; i++){
103                 for(int j = col; j < col + SUB_PIC_SIZE; j++){
104                     result->pixels[i][j][0] = candidate_imgs[min_idx[count]].pixels[i-row][j-col][0];
105                     result->pixels[i][j][1] = candidate_imgs[min_idx[count]].pixels[i-row][j-col][1];
106                     result->pixels[i][j][2] = candidate_imgs[min_idx[count]].pixels[i-row][j-col][2];
107                 }
108             }
109             count++;
110         }
111     }
```

這邊也可以使用平行化來做運算，但是同樣不會是最花時間的部分。而且 load 進 device memory 也會花費許多時間，並且 parallel code 會破壞原始 serial code 的 spatial locality。因此這個部分也留給 CPU 來計算，不會使用 GPU kernel 來進一步加速它。

那以上是整個 photo mosaic 演算法的 serial code。全部使用 CPU 來運行。接下來我將會撰寫 GPU kernel 將原本”在所有大圖中子圖找到最適合的候選小圖”的這個 task 分給不同的 block 與 thread 來運算。

# Parallel Cuda Code

- 將大圖中的子圖以前一步找到的最適合的小圖取代的 GPU kernel

```
135     find_min_idx<<<(tile_width * tile_height + THREADS_PER_BLOCK - 1) / THREADS_PER_BLOCK, THREADS_PER_BLOCK>>>(  
136         d_r_avg_target_grid,  
137         d_g_avg_target_grid,  
138         d_b_avg_target_grid,  
139         d_r_avg_candidate,  
140         d_g_avg_candidate,  
141         d_b_avg_candidate,  
142         num_candidate_imgs,  
143         tile_width,  
144         tile_height,  
145         d_min_idx  
146     );
```

在這之前必須確定要傳進 GPU kernel function 的所有 argument 都已經先經過 `cudaMalloc()` 與 `cudaMemcpy()`，將 host 端的值傳入 device 端。那至於如何切割運算的資料就留待下一小節做說明。在這邊我們將深入探討從 Serial Code 變成 Parallel Cuda Code 需要做的改變。

- 在將資料傳入 GPU kernel 前所需要做的準備: allocate, memcpy

```
83     // allocate device memory  
84     cudaMalloc((void **)&d_r_avg_target_grid, sizeof(double) * tile_width * tile_height);  
85     cudaMalloc((void **)&d_g_avg_target_grid, sizeof(double) * tile_width * tile_height);  
86     cudaMalloc((void **)&d_b_avg_target_grid, sizeof(double) * tile_width * tile_height);  
87     cudaMalloc((void **)&d_r_avg_candidate, sizeof(double) * num_candidate_imgs);  
88     cudaMalloc((void **)&d_g_avg_candidate, sizeof(double) * num_candidate_imgs);  
89     cudaMalloc((void **)&d_b_avg_candidate, sizeof(double) * num_candidate_imgs);  
90     cudaMalloc((void **)&d_min_idx, sizeof(int) * tile_width * tile_height);
```

上圖為在 device 端 allocate 資料。值得注意的是，若是在 host 端要直接去寫入該記憶體片段，會出現 segmentation fault。必須使用 `cudaMemcpy` 才可以將值寫入該記憶體片段。

```
122     // copy inputs to device  
123     cudaMemcpy(d_r_avg_target_grid, r_avg_target_grid, sizeof(double) * tile_width * tile_height, cudaMemcpyHostToDevice);  
124     cudaMemcpy(d_g_avg_target_grid, g_avg_target_grid, sizeof(double) * tile_width * tile_height, cudaMemcpyHostToDevice);  
125     cudaMemcpy(d_b_avg_target_grid, b_avg_target_grid, sizeof(double) * tile_width * tile_height, cudaMemcpyHostToDevice);  
126     cudaMemcpy(d_r_avg_candidate, r_avg_candidate, sizeof(double) * num_candidate_imgs, cudaMemcpyHostToDevice);  
127     cudaMemcpy(d_g_avg_candidate, g_avg_candidate, sizeof(double) * num_candidate_imgs, cudaMemcpyHostToDevice);  
128     cudaMemcpy(d_b_avg_candidate, b_avg_candidate, sizeof(double) * num_candidate_imgs, cudaMemcpyHostToDevice);
```

將先前所準備好的值透過 `cudaMemcpy` 複製進 device memory，使 GPU kernel 可以對這些資料進行運算。

將資料準備好，也複製資料進去 device memory 了，那就可以將焦點放到 GPU kernel 內部的 implementation 了。



- [GPU kernel]找到最適合大圖的子圖的小圖

```
3  __global__ void find_min_idx(  
4      const double *r_avg_target_grid,  
5      const double *g_avg_target_grid,  
6      const double *b_avg_target_grid,  
7      const double *r_avg_candidate,  
8      const double *g_avg_candidate,  
9      const double *b_avg_candidate,  
10     int num_candidate_imgs,  
11     int tile_width,  
12     int tile_height,  
13     int *min_idx  
14 )  
15 {  
16     int i = blockIdx.x * blockDim.x + threadIdx.x;  
17  
18     if (i < tile_width * tile_height) {  
19         double min_diff = DBL_MAX;  
20         for (int j = 0; j < num_candidate_imgs; ++j) {  
21             double r_diff = r_avg_target_grid[i] - r_avg_candidate[j];  
22             double g_diff = g_avg_target_grid[i] - g_avg_candidate[j];  
23             double b_diff = b_avg_target_grid[i] - b_avg_candidate[j];  
24             double diff = sqrt(r_diff * r_diff + g_diff * g_diff + b_diff * b_diff);  
25  
26             if (diff < min_diff) {  
27                 min_diff = diff;  
28                 min_idx[i] = j;  
29             }  
30         }  
31     }  
32 }
```

上圖為我們的 GPU kernel 實作方式，大致上與 serial code 相同，只不過將最外面的迴圈拆掉，將大圖中的每個子圖分到不同的 block 中，而每個 block 中的 thread 則會針對每一個大圖中的子圖去尋找最適合的候選小圖。最後將最適合的小圖存進 min\_idx 陣列中。

值得注意的是在程式中的第 16 行為計算該 thread 要針對第幾個子圖去尋找最適合的小圖。而後再將找到的結果寫到 min\_idx 中相對的位置。由於不同的 thread 中不會有 race condition 或是 data hazard 的問題。因此這邊是不用加上 \_\_syncthreads 等所有 thread 都完成才結束這個 GPU kernel 的。

但是這邊仍然有許多的優化空間。例如第 4 到 9 行傳進來的陣列是 const，在 GPU kernel 中是 read only 的，因此可以將其放進 share memory。讓同一個 block 的 thread 都可以看到同一份的資料，使得程式可以跑得更快(GPU kernel 在 access share memory 會有更快得存取時間)。

# 資料切割方法

在 cuda 中有兩層階層式的平行關係。第一層是將資料切割成 block，而第二層是每個 block 中又有多個 thread。這樣階層式的平行關係使得同個 block 的 thread 可以互相 access 相同的 share memory[以\_\_share\_\_為 keyword]，可以減少記憶體存取的時間。此外，同一個 block 間的 thread 也可以互相 synchronize，與 pthread 的 join 類似，等待所有的 thread 做到這邊才可以繼續往後做，以避免 data race 的可能性。

在這次的期末專題中，我單純將資料分到多個 block，而每個 block 有 512 個 threads。所以我的資料切割方式詳細如下：

- 先計算大圖的長寬(wxh)是 cifar10 小圖(32x32)的幾倍
  - A.  $\text{tile\_width} = \text{大圖的 } w / 32$
  - B.  $\text{tile\_height} = \text{大圖的 } h / 32$
- 大圖總共切成  $\text{tile\_width} \times \text{tile\_height}$  這麼多個子圖

將  $\text{tile\_width} \times \text{tile\_height}$  這個多個子圖分給每個 block，而每個 block 中有 512 個 thread。因此共需要

$$\#blocks = \text{ceil}\left(\frac{\text{tile\_width} \times \text{tile\_height}}{\text{THREADS\_PER\_BLOCK}}\right)$$

這麼多個 block，而為甚麼需要取 ceil() 呢？若是需要運算的子圖不是 THREADS\_PER\_BLOCK 的倍數，那麼會沒有辦法整除，為了要處理剩下的資料，就必須要多開一個 block 來計算剩餘的資料。也因此 GPU kernel 中的第 18 行需要判斷算出來的 index 是否在需求範圍內，如果超過那就不用算了。透過這個 if 來接受各式各樣長寬的大圖，使得 GPU kernel 內部可以執行 arbitrary size 的 vector。

總而言之，我將每個大圖中的子圖分給 GPU 的某個 thread 來計算，每 512 個 thread 會組成一個 block。但我認為這也不會是一個很好切割資料的方式。比較好的方式應該為在事前先 run 一個 configuration 的程式，知道該執行環境的設定，將這個設定自動帶入跑 GPU kernel 時的設定，這樣寫出來的 GPU kernel 的程式才可以隨著硬體最佳化，而不會需要人工去針對不同的執行平台去更改 run 某個 GPU kernel code 需要多少個 block 或是多少個 thread。

# 模擬環境(Lab Server)

```
Model: NVIDIA L4
IRQ: 150
GPU UUID: GPU-d6dfd91f-1633-8811-2d44-861315f58a4c
Video BIOS: 95.04.29.00.06
Bus Type: PCIe
DMA Size: 47 bits
DMA Mask: 0x7fffffffffffff
Bus Location: 0000:5b:00.0
Device Minor: 0
GPU Firmware: 535.104.05
GPU Excluded: No
```

cat /proc/driver/nvidia/gpus/0000:5b:00.0/information

```
Architecture: x86_64
CPU op-mode(s): 32-bit, 64-bit
Byte Order: Little Endian
CPU(s): 160
On-line CPU(s) list: 0-159
Thread(s) per core: 2
Core(s) per socket: 20
Socket(s): 4
NUMA node(s): 4
Vendor ID: GenuineIntel
CPU family: 6
Model: 85
Model name: Intel(R) Xeon(R) Gold 6230 CPU @ 2.10GHz
Stepping: 7
CPU MHz: 2799.987
CPU max MHz: 3900.0000
CPU min MHz: 800.0000
BogoMIPS: 4200.00
Virtualization: VT-x
L1d cache: 32K
L1i cache: 32K
L2 cache: 1024K
L3 cache: 28160K
NUMA node0 CPU(s): 0,4,8,12,16,20,24,28,32,36,40,44,48,52,56,60,64,68,72,76,80,84,88,92,96,100,104,108,112,116,120,124,128,132,136,140,144,148,152,156
NUMA node1 CPU(s): 1,5,9,13,17,21,25,29,33,37,41,45,49,53,57,61,65,69,73,77,81,85,89,93,97,101,105,109,113,117,121,125,129,133,137,141,145,149,153,157
NUMA node2 CPU(s): 2,6,10,14,18,22,26,30,34,38,42,46,50,54,58,62,66,70,74,78,82,86,90,94,98,102,106,110,114,118,122,126,130,134,138,142,146,150,154,158
NUMA node3 CPU(s): 3,7,11,15,19,23,27,31,35,39,43,47,51,55,59,63,67,71,75,79,83,87,91,95,99,103,107,111,115,119,123,127,131,135,139,143,147,151,155,159
```

lscpu

```
=====NVSMI LOG=====

Timestamp                               : Thu Jun  6 22:35:47 2024
Driver Version                           : 535.104.05
CUDA Version                             : 12.2

Attached GPUs                             : 1
GPU 00000000:5B:00.0
  Product Name                           : NVIDIA L4
  Product Brand                           : NVIDIA
  Product Architecture                     : Ada Lovelace
```

nvidia-smi -q

```
Device 0: NVIDIA L4
Total Global Memory: 23583784960
Multiprocessors: 58
Compute Capability: 8.9
```



```
cudaDeviceProp deviceProp;
cudaGetDeviceProperties(&deviceProp, device);
std::cout << "Device " << device << ": " << deviceProp.name << std::endl;
std::cout << "  Total Global Memory: " << deviceProp.totalGlobalMem << std::endl;
std::cout << "  Multiprocessors: " << deviceProp.multiProcessorCount << std::endl;
std::cout << "  Compute Capability: " << deviceProp.major << "." << deviceProp.minor << std::endl;
```

Embedded c++ cuda checkout function



# 模擬結果

## Testcase 1: 貓頭鷹

原始照片	
成果照片	
tile width(寬由幾張小圖構成)	187
tile height(高由幾張小圖構成)	116
大圖被切分成幾等分	187x116=21692
Serial Code 運行時間	3616.4(ms)
Cuda Code 運行時間	21.51(ms)
效率提升	168 X Improvement

```
##### Test case1 #####  
[Serial]Elapsed time: 3616.406494 ms  
[Serial](Tile_Width, Tile_Height) = (187, 116)  
[CUDA]Elapsed time: 21.510592 ms  
[CUDA](Tile_Width, Tile_Height) = (187, 116)
```

Run Time Output Log

## Testcase 2: 向日葵

原始照片	
成果照片	
tile width(寬由幾張小圖構成)	125
tile height(高由幾張小圖構成)	187
大圖被切分成幾等分	125x187=23375
Serial Code 運行時間	3893.71(ms)
Cuda Code 運行時間	21.26(ms)
效率提升	183 X Improvement

```
##### Test case2 #####
[Serial]Elapsed time: 3873.975586 ms
[Serial](Tile_Width, Tile_Height) = (125, 187)
[CUDA]Elapsed time: 21.260384 ms
[CUDA](Tile_Width, Tile_Height) = (125, 187)
```

Run Time Output Log

### Testcase 3: 狗



原始照片	
成果照片	
tile width(寬由幾張小圖構成)	162
tile height(高由幾張小圖構成)	121
大圖被切分成幾等分	162x121=19602
Serial Code 運行時間	3235.33(ms)
Cuda Code 運行時間	21.26(ms)
效率提升	152 X Improvement

```
##### Test case3 #####
[Serial]Elapsed time: 3250.671143 ms
[Serial](Tile_Width, Tile_Height) = (162, 121)
[CUDA]Elapsed time: 21.259552 ms
[CUDA](Tile_Width, Tile_Height) = (162, 121)
```

Run Time Output Log



## Testcase 4: 組員合照

原始照片	
成果照片	
tile width(寬由幾張小圖構成)	93
tile height(高由幾張小圖構成)	125
大圖被切分成幾等分	$93 \times 125 = 11625$
Serial Code 運行時間	1919.59(ms)
Cuda Code 運行時間	10.89(ms)
效率提升	176 X Improvement

```
##### Test case4 #####
[Serial]Elapsed time: 1919.592407 ms
[Serial](Tile_Width, Tile_Height) = (93, 125)
[CUDA]Elapsed time: 10.885632 ms
[CUDA](Tile_Width, Tile_Height) = (93, 125)
```

Run Time Output Log

在這四個 case 中，我們使用 cuda GPU kernel 所帶來的相較單純使用 CPU 來計算平均快了 170 倍!

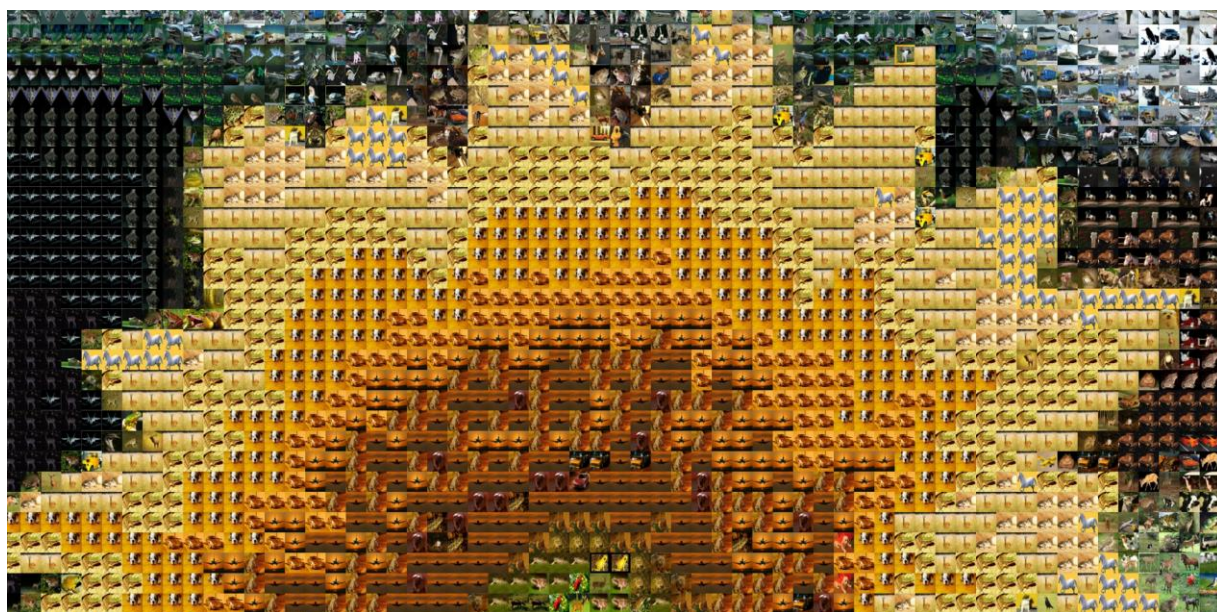
# 量測方法

在前一步計算運行時間我們是使用 cuda 內建的 timing profile 的函式，如下：

```
130 // for each grid image in target image, find the best fit candidate image from cifar10
131 cudaEvent_t start, stop;
132 cudaEventCreate(&start);
133 cudaEventCreate(&stop);
134 cudaEventRecord(start, 0);
135 find_min_idx<<<<(tile_width * tile_height + THREADS_PER_BLOCK - 1) / THREADS_PER_BLOCK, THREADS_PER_BLOCK>>>(<
136     d_r_avg_target_grid,
137     d_g_avg_target_grid,
138     d_b_avg_target_grid,
139     d_r_avg_candidate,
140     d_g_avg_candidate,
141     d_b_avg_candidate,
142     num_candidate_imgs,
143     tile_width,
144     tile_height,
145     d_min_idxs
146 );
147
148 cudaEventRecord(stop, 0);
149 cudaEventSynchronize(stop);
150 float elapsedTime;
151 cudaEventElapsedTime(&elapsedTime, start, stop);
152 printf("[CUDA]Elapsed time: %f ms\n", elapsedTime);
153 printf("[CUDA](Tile_Width, Tile_Height) = (%d, %d)\n", tile_width, tile_height);
```

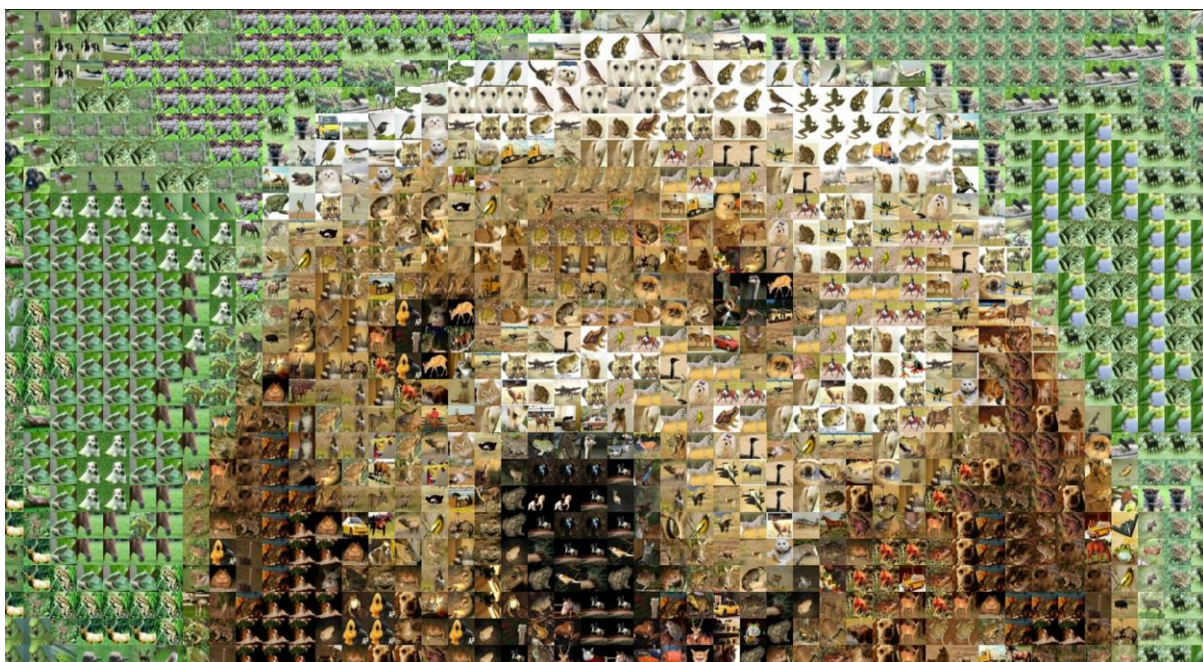
透過 cudaEvent 將 GPU kernel 運行前後包起來，相差的時間即為該 GPU kernel 的運行時間。在這邊我們為求簡單，不考慮資料搬運的時間，單純量測該 GPU kernel 運行所需要花費的時間。

## 一些有趣的結果

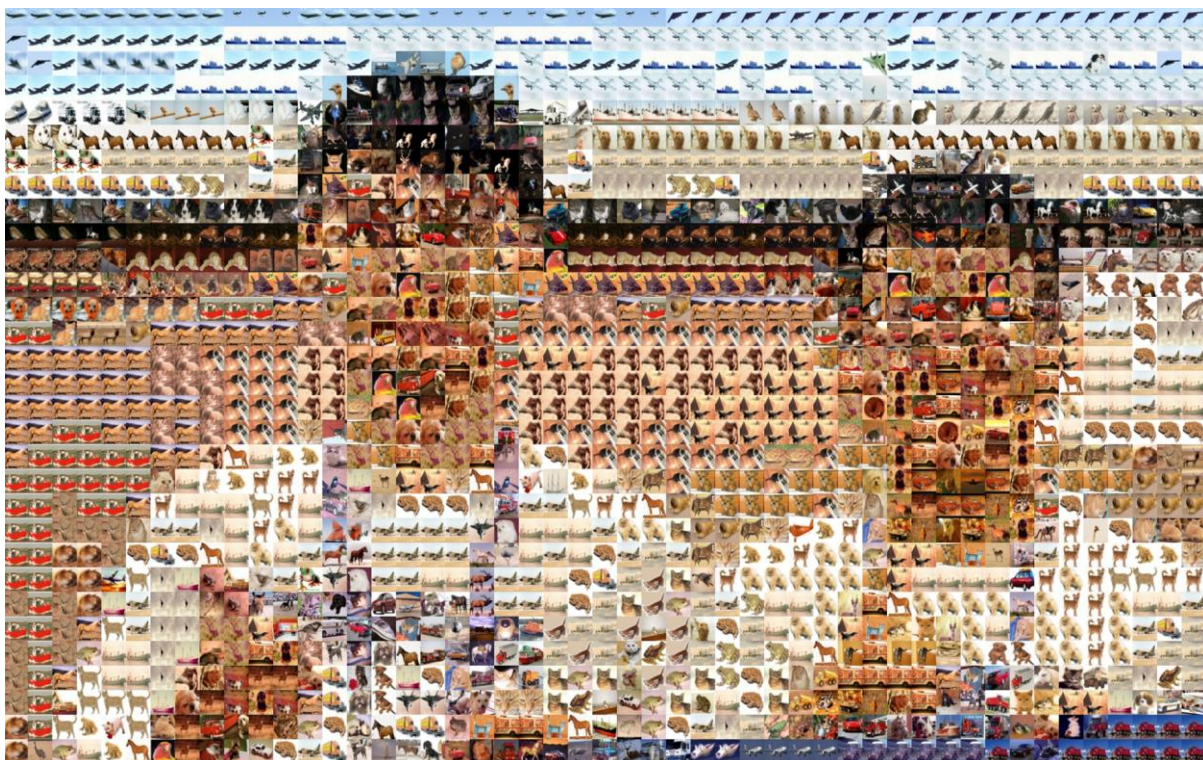


向日葵放大後，可以看到狗狗與、飛機跟馬！





狗狗的臉部放大可以看到青蛙、狗、小鳥、貓!



較低解析度的照片，或是照片中比較小的物體，經過這個演算法後，結果會比較不理想。人物變的很糊，五官都不見了!



# 程式運行

\$ git clone <https://github.com/coherent17/PhotoMosaic-Cuda-Optimization>

\$ cd PhotoMosaic-Cuda-Optimization

\$ make install #抓取第三方 open source project

\$ make -j #parallel compile

```
@coherent17 → PhotoMosaic-Cuda-Optimization git(main) make -j
NVCC      obj/data_loader.o
NVCC      obj/gray_image.o
NVCC      obj/image.o
NVCC      obj/photo_mosaic_serial.o
NVCC      obj/rgb_image.o
NVCC      obj/photo_mosaic_cuda.o
LD         PhotoMosaic
```

有撰寫好 makefile，並且使用 NVCC 來編譯。

\$ make VERBOSE=1 #秀出所有 compile 過程

```
@coherent17 → PhotoMosaic-Cuda-Optimization git(main) make VERBOSE=1
nvcc -I ./inc -I ./third-party/Cimg -I ./third-party/libjpeg -I ./Data-Loader -std=c++11 -xptxas -O3 -MMO -c src/data_loader.cpp -o obj/data_loader.o
nvcc -I ./inc -I ./third-party/Cimg -I ./third-party/libjpeg -I ./Data-Loader -std=c++11 -xptxas -O3 -MMO -c src/gray_image.cpp -o obj/gray_image.o
nvcc -I ./inc -I ./third-party/Cimg -I ./third-party/libjpeg -I ./Data-Loader -std=c++11 -xptxas -O3 -MMO -c src/image.cpp -o obj/image.o
nvcc -I ./inc -I ./third-party/Cimg -I ./third-party/libjpeg -I ./Data-Loader -std=c++11 -xptxas -O3 -MMO -c src/photo_mosaic_serial.cpp -o obj/photo_mosaic_serial.o
nvcc -I ./inc -I ./third-party/Cimg -I ./third-party/libjpeg -I ./Data-Loader -std=c++11 -xptxas -O3 -MMO -c src/rgb_image.cpp -o obj/rgb_image.o
nvcc -I ./inc -I ./third-party/Cimg -I ./third-party/libjpeg -I ./Data-Loader -std=c++11 -xptxas -O3 -MMO -c src/photo_mosaic_cuda.cu -o obj/photo_mosaic_cuda.o
nvcc -I ./inc -I ./third-party/Cimg -I ./third-party/libjpeg -I ./Data-Loader -std=c++11 -xptxas -O3 main.cu obj/data_loader.o obj/gray_image.o obj/image.o obj/photo_mosaic_serial.o obj/rgb_image.o obj/photo_mosaic_cuda.o -o PhotoMosaic -L/usr/X11R6/lib -ln -lX11 -L./third-party/libjpeg -ljpeg -lpng
```

\$ ./PhotoMosaic #執行 binary executable

```
@coherent17 → PhotoMosaic-Cuda-Optimization git(main) ./PhotoMosaic
Device 0: NVIDIA GeForce MX350
Total Global Memory: 2147352576
Multiprocessors: 5
Compute Capability: 6.1
##### Test case1 #####
[Serial]Elapsed time: 1165.619141 ms
[Serial](Tile_Width, Tile_Height) = (187, 116)
[CUDA]Elapsed time: 190.699493 ms
[CUDA](Tile_Width, Tile_Height) = (187, 116)
##### Test case2 #####
[Serial]Elapsed time: 1321.745239 ms
[Serial](Tile_Width, Tile_Height) = (125, 187)
[CUDA]Elapsed time: 166.791840 ms
[CUDA](Tile_Width, Tile_Height) = (125, 187)
##### Test case3 #####
[Serial]Elapsed time: 1088.349365 ms
[Serial](Tile_Width, Tile_Height) = (162, 121)
[CUDA]Elapsed time: 164.732285 ms
[CUDA](Tile_Width, Tile_Height) = (162, 121)
##### Test case4 #####
[Serial]Elapsed time: 672.573975 ms
[Serial](Tile_Width, Tile_Height) = (93, 125)
[CUDA]Elapsed time: 122.195648 ms
[CUDA](Tile_Width, Tile_Height) = (93, 125)
```

在筆電跑的結果

## 分工合作

何祁恩: 環境建置 Serial Code & 報告

鄭文洋: Cuda Code & 報告