CA Final Project

電物系 碩一

311651052 吳挺宇；311651055 林柏宇

一、 題目簡介:

灰階處理（Gray scale)，在一般數位影像圖片由多個像素單位組成，分別有他們的 RGB 值，改為每個像素只有一個採樣顏色的影像。其中，灰階影像常用於影像識別，通常是以某個特定函式，將三維 RGB 值轉換為一個二進制 8bits 的一維數值，總共可以有 256 種灰階。在醫學影像與遙感影像則會運用 10~12bits 左右的灰階來提高傳感器精度，避免計算時的近似誤差。如下圖；左圖為一般數位影像圖片，右圖為經過灰階處理後的圖片。

origin image

grayscale image

二、 實驗環境:

- 課程助教所提供之模擬器
- 基本記憶體: 2048MB
- 處理器: 1xCPU

三、 檔案說明:

檔案名稱	說明
main.cu	Host 端程式,使用 CPU 執行灰階。
cuda1.cu	Device 端程式，使用 GPU 執行灰階。（第一種資料切割方式）
cuda2.cu	Device 端程式，使用 GPU 執行灰階。（第二種資料切割方式）

四、 演算法內容:

通過隨機取樣方式，生成各個像素格點中的數值並將其設定為 RGB 值。通過設定的 RGB 值由(式 1)轉換成灰階。

由 RGB 到灰階透過以下的函數轉換：

H[i][j] = R[i][j] * a + G[i][j] * b + B[i][j] * c （式 1）

其中(a,b,c)為轉換的係數，一般 (a,b,c) = (0.299,0.587,0.114)

H[i][j]為圖片第 i 列、第 j 行的輸出、R[i][j]、G[i][j]、B[i][j]為圖片第 i 列、第 j 行本身的 RGB 值。

如下圖：

R=140 G=167 B=120		

154		

五、資料分割方式:

1. 方法一：如下圖 a；將列上的 i/n 與行的 j/n，將資料切為 n 塊小矩陣，當區域 1、2、3、4 分別都執行完各自的[0][0]，才能執行下一個(follow memory row major order)，也就是各自的[0][1]。以 n=4 來說如圖 a 所示，若無法整除則將最後一行或列放入最接近一塊 pipeline 執行。例如:底下多出來的部分，分別分配到 3 & 4 執行。

```
__global__ void grayscaleConversion(unsigned char *inputImage, unsigned char *outputImage, int width, int height)
{
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;


    //allocate each pixel to threads
    if (x < width && y < height) {
        int tid = y * width + x;
        unsigned char r = inputImage[3 * tid];
        unsigned char g = inputImage[3 * tid + 1];
        unsigned char b = inputImage[3 * tid + 2];
        outputImage[tid] = 0.299f * r + 0.587f * g + 0.114f * b;
    }
}
```

Device code:

在每次平行運算中，同時計算第 1、2、3、4 區域內的[0][0]像素的 RGB 值。同時此程式的 block 與 grid 維度設置如下圖；其中 blockDim 是為了讓我足以能夠 loading 我像素大小的 Thread 數去配置的；而 gridDim 則是為了以防超過而預留了一點空位。另外與方法二相比，方法一運行速度較

快，可知可利用此概念加速運算。

```
dim3 blockDim(4,4);
dim3 gridDim((width + blockDim.x - 1) / blockDim.x, (height + blockDim.y - 1) / blockDim.y);
```

2. 方法二：如下圖 b；將 n 個像素分別交給 n 個 Thread 執行灰階運算。參考底下的程式碼，我的 Input 檔編號%3 = 0 的矩陣格位為該像素 RGB 的 R，%3 = 1、%3 = 2 分別為 RGB 的 G 與 B。一個 Thread 一次負責執行一組 RGB 的灰階運算： 0.299 * R + 0.587 * G + 0.114 * B。

```
__global__ void grayscaleConversion(unsigned char *inputImage, unsigned char *outputImage, int width, int height)
{
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;

    if (x < width && y < height) {
        int tid = y * width + x;
        unsigned char r = inputImage[3 * tid];
        unsigned char g = inputImage[3 * tid + 1];
        unsigned char b = inputImage[3 * tid + 2];
        outputImage[tid] = 0.299f * r + 0.587f * g + 0.114f * b;
    }
}
```

其中 block、grid 參數與法一相同，其中 blockDim 是為了讓我足以能夠 loading 我像素大小的 Thread 數去配置的，而 gridDim 則是為了以防超過而預留了一點空位。

```
dim3 blockDim(4,4);
dim3 gridDim((width + blockDim.x - 1) / blockDim.x, (height + blockDim.y - 1) / blockDim.y);
```

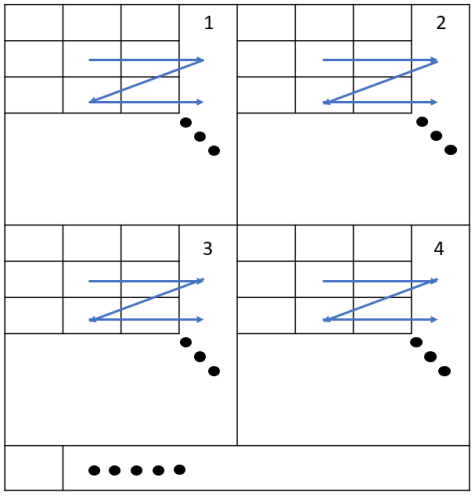

圖 a:方法一

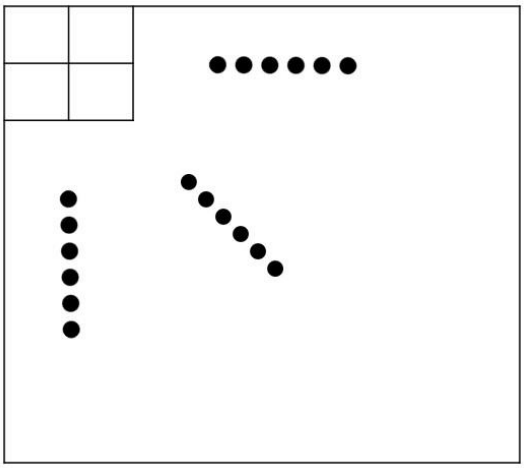

圖 b:方法二

表一

六、模擬結果:

以下為使用 matlab 驗證灰階運算的結果,圖 c 為 24×24 之輸入影像矩陣,而圖 d 是運算結果之 8×8 輸出矩陣,運算結果正確,可發現輸出矩陣數值經由(式 1)的數學運算後可驗證為正確結果。

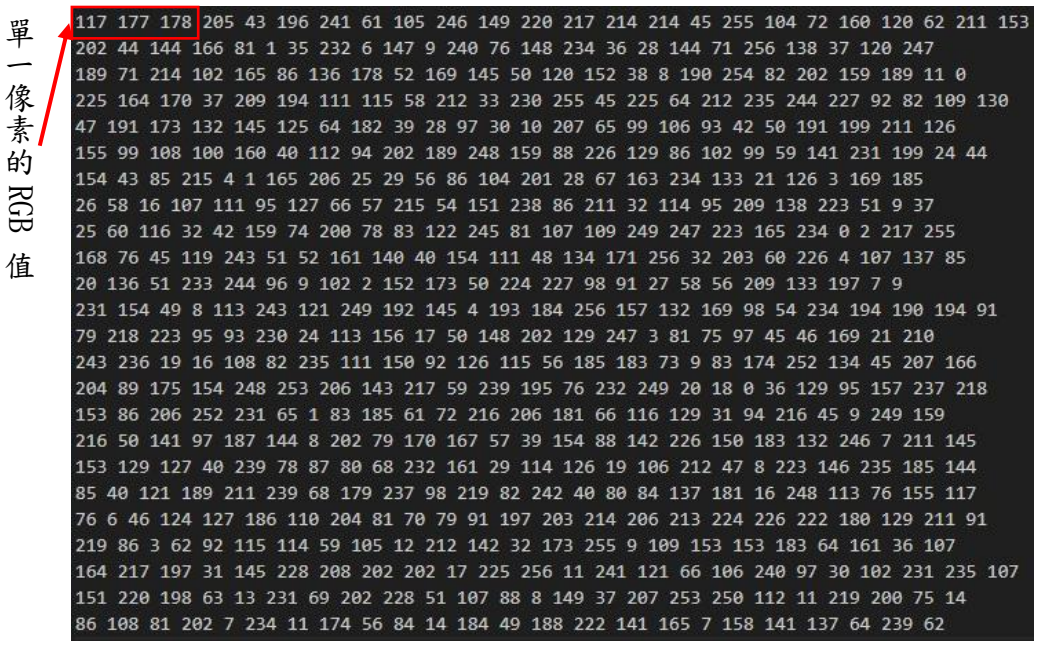

圖 c: 24×24 之輸入影像矩陣

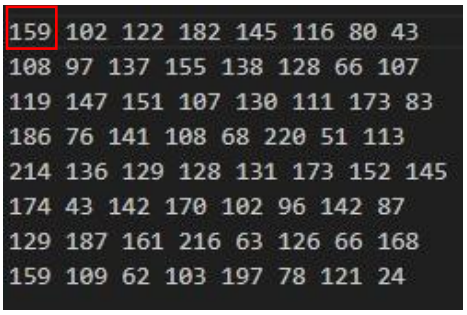

圖 d:運算結果之 8×8 輸出矩陣