

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：

$$H[i][j] = R[i][j] * a + G[i][j] * b + B[i][j] * c \quad (\text{式 1})$$

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。

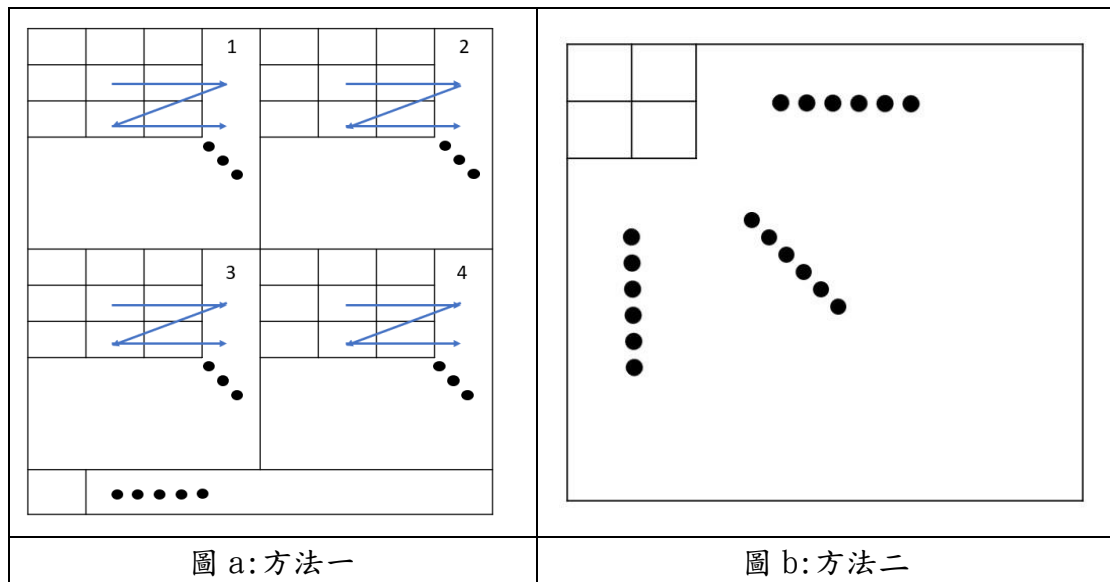
參考底下的程式碼，我的 Input 檔編號%3 = 0 的矩陣格位為該像素 RGB 的 R，%3 = 1、%3 = 2 分別為 RGB 的 G 與 B。一個 Thread 一次負責執行一組 RGB 的灰階運算： $0.299 * R + 0.587 * G + 0.114 * B$ 。

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



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。

單一像素的 RGB 值

117	177	178	205	43	196	241	61	105	246	149	220	217	214	214	45	255	104	72	160	120	62	211	153
202	44	144	166	81	1	35	232	6	147	9	240	76	148	234	36	28	144	71	256	138	37	120	247
189	71	214	102	165	86	136	178	52	169	145	50	120	152	38	8	190	254	82	202	159	189	11	0
225	164	170	37	209	194	111	115	58	212	33	230	255	45	225	64	212	235	244	227	92	82	109	130
47	191	173	132	145	125	64	182	39	28	97	30	10	207	65	99	106	93	42	50	191	199	211	126
155	99	108	100	160	40	112	94	202	189	248	159	88	226	129	86	102	99	59	141	231	199	24	44
154	43	85	215	4	1	165	206	25	29	56	86	104	201	28	67	163	234	133	21	126	3	169	185
26	58	16	107	111	95	127	66	57	215	54	151	238	86	211	32	114	95	209	138	223	51	9	37
25	60	116	32	42	159	74	200	78	83	122	245	81	107	109	249	247	223	165	234	0	2	217	255
168	76	45	119	243	51	52	161	140	40	154	111	48	134	171	256	32	203	60	226	4	107	137	85
20	136	51	233	244	96	9	102	2	152	173	50	224	227	98	91	27	58	56	209	133	197	7	9
231	154	49	8	113	243	121	249	192	145	4	193	184	256	157	132	169	98	54	234	194	190	194	91
79	218	223	95	93	230	24	113	156	17	50	148	202	129	247	3	81	75	97	45	46	169	21	210
243	236	19	16	108	82	235	111	150	92	126	115	56	185	183	73	9	83	174	252	134	45	207	166
204	89	175	154	248	253	206	143	217	59	239	195	76	232	249	20	18	0	36	129	95	157	237	218
153	86	206	252	231	65	1	83	185	61	72	216	206	181	66	116	129	31	94	216	45	9	249	159
216	50	141	97	187	144	8	202	79	170	167	57	39	154	88	142	226	150	183	132	246	7	211	145
153	129	127	40	239	78	87	80	68	232	161	29	114	126	19	106	212	47	8	223	146	235	185	144
85	40	121	189	211	239	68	179	237	98	219	82	242	40	80	84	137	181	16	248	113	76	155	117
76	6	46	124	127	186	110	204	81	70	79	91	197	203	214	206	213	224	226	222	180	129	211	91
219	86	3	62	92	115	114	59	105	12	212	142	32	173	255	9	109	153	153	183	64	161	36	107
164	217	197	31	145	228	208	202	202	17	225	256	11	241	121	66	106	240	97	30	102	231	235	107
151	220	198	63	13	231	69	202	228	51	107	88	8	149	37	207	253	250	112	11	219	200	75	14
86	108	81	202	7	234	11	174	56	84	14	184	49	188	222	141	165	7	158	141	137	64	239	62

圖 c: 24×24 之輸入影像矩陣

159	102	122	182	145	116	80	43
108	97	137	155	138	128	66	107
119	147	151	107	130	111	173	83
186	76	141	108	68	220	51	113
214	136	129	128	131	173	152	145
174	43	142	170	102	96	142	87
129	187	161	216	63	126	66	168
159	109	62	103	197	78	121	24

圖 d: 運算結果之 8×8 輸出矩陣