多核心嵌入式系統與軟體專題作業一報告

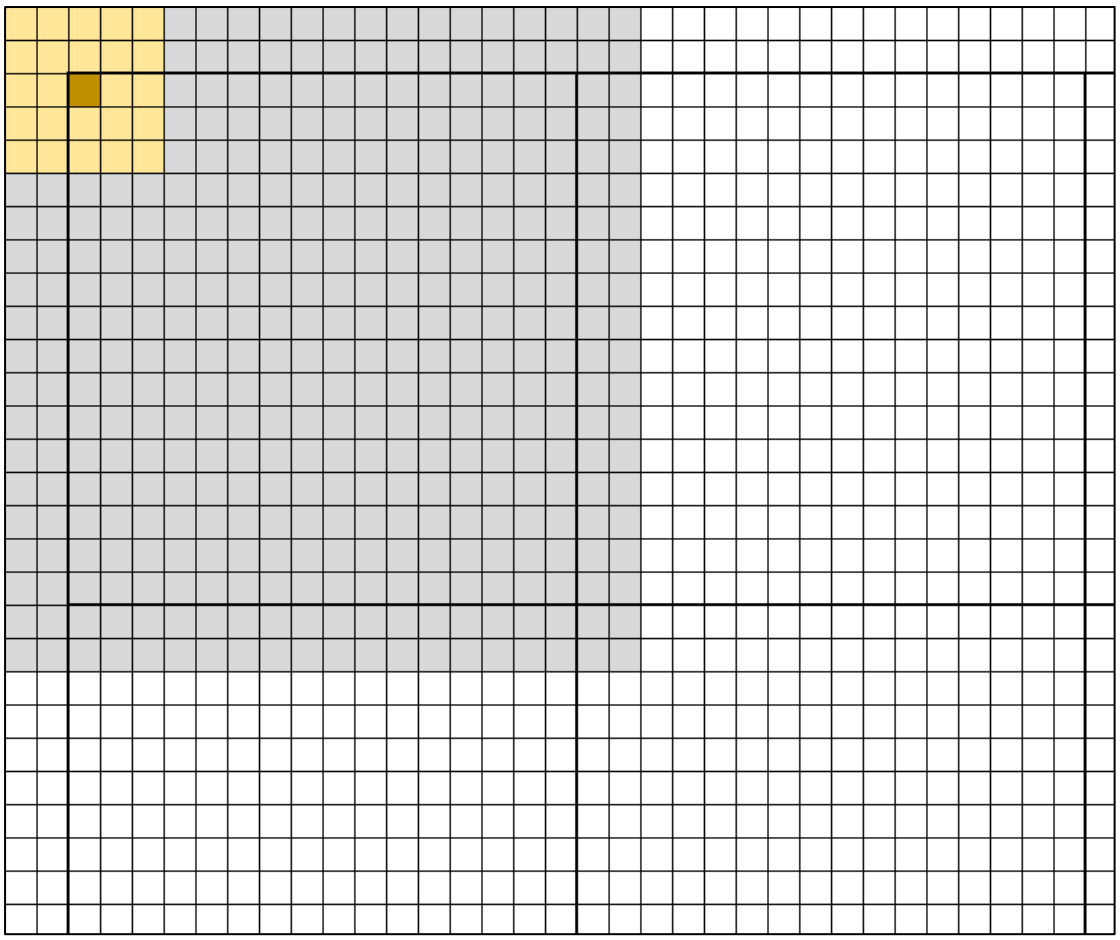
A1085512 林嘉軒

壹、摘要

本專題作業以 C++程式實作特徵匹配（Template Matching）機制，並利用 Nvidia CUDA 套件庫來實現平行加速運算。該程式支援皮爾森積差相關係數（Pearson Correlation Coefficient）與差方和（Sum of Squared Differences）兩種相似度指標，並以分塊（Blocking）技巧提升記憶體存取效率。

貳、實作

存取共享記憶體（Shared Memory）比全域記憶體（Global Memory）要來得快上許多，但共享記憶體的大小又不足以塞入整個要被比對的矩陣，所以我在實作中使用分塊技巧，將一個區塊（Block）中被存取的部分寫入共享記憶體，以達到類似快取（Cached）的作用。



參考上圖，以 16x16 的區塊大小（256 個線程，一個線程計算其中一個元素）來說，在一個區塊中，實際要進行運算的範圍為粗框線所圍起來之 16x16 的矩陣，而黃色標示之區域為卷積核（範例為 5x5），灰色標示之區域則是被寫入共享記憶體的矩陣。灰色區域較實際運算範圍大一點，以解決不同分塊間，卷積運算有範圍交疊的問題。

該實作將特徵匹配分成兩部分來進行，分別由 GPU 進行卷積計算相似度，再由 CPU（如下函示宣告）針對結果找尋最相似的目標。

```
std::vector<std::pair<int, int>> findBestMatch(const float *sum, int
    width, int height, bool findMax)
```

本程式支援計算皮爾森積差相關係數與差方和，兩種相似度指標。為了程式實作方便，我將該部分計算寫成函式（參見以下函式宣告與定義），傳入卷積核與相同大小之矩陣，再回傳對應的結果值。該方法的缺點是，必須從較大的矩陣切出要計算的區域並存入一個變數，才能傳入函式進行計算。雖然額外增加了這個步驟，但可提升程式的可閱讀性。

```
__device__ __host__ float PCC(const float *matX, const float *matY)
```

```
__device__ __host__ float SSD(const float *matX, const float *matY) {
    float sum = 0;
    for (int i = 0; i < KERNEL_HEIGHT; i++) {
        for (int j = 0; j < KERNEL_WIDTH; j++) {
            sum += (matX[i * KERNEL_HEIGHT + j] -
                    matY[i * KERNEL_HEIGHT + j]) *
                   (matX[i * KERNEL_HEIGHT + j] -
                    matY[i * KERNEL_HEIGHT + j]);
        }
    }
    return sum;
}
```

下圖為 GPU 計算皮爾森積差相關係數的核心函式（Kernel Function）

```
__global__ void CalculatePCC(const float *matrix, size_t ldm, const
    float *kernel, size_t ldk, float *sum, size_t lds) {
    __shared__ float filter[KERNEL_WIDTH * KERNEL_HEIGHT];
    __shared__ float tile[(BLOCK_SIZE + KERNEL_WIDTH / 2 * 2) *
                          (BLOCK_SIZE + KERNEL_HEIGHT / 2 * 2)];
    if (threadIdx.x < KERNEL_HEIGHT && threadIdx.y < KERNEL_WIDTH) {
        filter[threadIdx.x * KERNEL_WIDTH + threadIdx.y] =
            kernel[threadIdx.x * ldk + threadIdx.y];
    }                                          將卷積核寫入共享記憶體
    for (int i = (int) threadIdx.x;
        i < (BLOCK_SIZE + KERNEL_HEIGHT / 2 * 2) &&
        i < MATRIX_HEIGHT; i += BLOCK_SIZE) {z
        for (int j = (int) threadIdx.y;
            j < (BLOCK_SIZE + KERNEL_WIDTH / 2 * 2) &&
            j < MATRIX_WIDTH; j += BLOCK_SIZE) {
            tile[i * (BLOCK_SIZE + KERNEL_WIDTH / 2 * 2) + j] =
                    matrix[(blockIdx.x * BLOCK_SIZE + i) * ldm +
                           (blockIdx.y * BLOCK_SIZE + j)];
        }
    }                                          將分塊矩陣寫入共享記憶體
    __syncthreads();
    float image[KERNEL_WIDTH * KERNEL_HEIGHT];
    for (int i = 0; i < KERNEL_HEIGHT; i++) {
        for (int j = 0; j < KERNEL_WIDTH; j++) {
            image[i * KERNEL_WIDTH + j] =
                tile[(threadIdx.x + i) *
                     (BLOCK_SIZE + KERNEL_WIDTH / 2 * 2) +
                     (threadIdx.y + j)];
        }
    }                                          切出要進行計算之區域
    if (blockIdx.x * BLOCK_SIZE + threadIdx.x < SUM_HEIGHT &&
        blockIdx.y * BLOCK_SIZE + threadIdx.y < SUM_WIDTH) {
        sum[(blockIdx.x * BLOCK_SIZE + threadIdx.x) * lds +
            (blockIdx.y * BLOCK_SIZE + threadIdx.y)] =
            PCC(image, filter);
    }                                          計算該區域之 PCC 並儲存結果
}
```

參、實驗

測試平台資訊

CPU	Intel Core i5-8400 @ 2.80GHz
RAM	16 GB
GPU	NVIDIA GeForce GTX 1050 Ti
OS	Ubuntu 20.04.6 LTS
CUDA Driver	12.0
CUDA Runtime	11.6

實驗結果

測資	函式	線程數	執行時間（秒）
卷積核：3×3 圖像：3750×4320	PCC	64	0.06544978
		256	0.06519269
		1024	0.06520767
	SSD	64	0.06333082
		256	0.06331473
		1024	0.06332835
卷積核：5×5 圖像：7750×1320	PCC	64	0.04027000
		256	0.04024140
		1024	0.04023060
	SSD	64	0.04160310
		256	0.04143880
		1024	0.04170100
卷積核：3×3 圖像：8140×9925	PCC	64	0.31351540
		256	0.31466180
		1024	0.31442290
	SSD	64	0.32333700
		256	0.32277550
		1024	0.33099710
卷積核：5×5 圖像：50×50	PCC	64	0.00020407
		256	0.00021441
		1024	0.00020212
	SSD	64	0.00020788
		256	0.00021508
		1024	0.00021261

註：表格中的每筆數據皆為執行程式 10 次所取得之平均運算時間。

本實驗主要討論不同線程數量，對於執行速度之影響。不同線程數量，會使得寫入共享記憶體的分塊大小不同，也會使網格維度（Grid Dimension）改變。以側資一為例，一個區塊 64 個線程，需要使用 253,260（469x540）個區塊；一個區塊 1024 個線程，則只需使用 15,930（118x135）個區塊。理論上不同數量的區塊與線程數量，會影響 GPU 對區塊分配到 SM 的排程及運算，但就實驗數據來看，即便以最大筆的測資來看，使用 64、256 或 1024 個線程，僅有大約 1 毫秒的差距，對執行時間沒有很明顯的影響。Nvidia 官網有說明，對於沒有特別考量最佳化的情況下，採用 128 或 256 個線程，是較適當的選擇。

肆、討論

對於本次作業，主要的困難在於規劃一個區塊所要運算的範圍，以及計算與其相關矩陣的索引值。稍有不甚，便很容易存取到未知的空間，也得特別注意邊界，同時因為 GPU 是一個高度平行化的運算平台，相當不容易進行偵錯，幸好利用 cuda-gdb 與 Nsight 能夠提供一些基本的偵錯功能，讓程式可以順利編譯。

如果說要進一步提升效能，可能有以下幾種方向：(1) 讓每個線程運算多個元素，這可以讓我們將更大的分塊寫入共享記憶體，而且一個線程也不會太快完成工作，造成過多的溝通與調度成本。(2) 在本實作中，使用額外函式計算單一元素的相似度值，需要先另外準備好一個空間存放要與卷積核運算的矩陣，若能將該部分整合到核心函式之中，僅需利用迴圈搭配陣列索引值，即可進行運算，進而減少不必要的函式呼叫成本。

伍、參考

[1] CUDA C++ Programming Guide

[2] Two Applications Using CUDA

[3] 課程簡報