多核心嵌入式系統與軟體專題作業二報告

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 POSIX Threads 套件庫來實現在 CPU 的平行加速運算。該程式支援皮爾森積差相關係數（Pearson Correlation Coefficient，PCC）與差方和（Sum of Squared Differences，SSD）兩種相似度指標。

貳、實作

程式具體執行的邏輯之說明：先分配每個線程（Thread）要計算的範圍，利用 ThreadData 結構傳入參數，計算完成後直接修改全域變數，最後再利用序列化部分（Serial Part）的程式來搜尋最相似的位置。後面兩段即針對如何切割不同線程運算範圍、以及存取全域變數詳加說明。

本程式需要使用者指定欲使用之最大線程數量 MAX_THREAD_NUM，之後將要進行比對的矩陣，切分成線程數個等大的小矩陣，在我的實作中為求方便，是直接對列切分，也就是矩陣列數除以線程數量個列為一組，送入新增的線程中進行比對運算。該作法可能會遇到一個問題：當指定線程數量比矩陣列數還多時，會有冗餘的線程被建立，故本程式進行了判斷，若指定線程數量比矩陣列數還多，則最多只建立矩陣列數個線程。

Pthreads 套件庫實現了共享記憶體模型（Shared Memory Model），宣告在全域的變數會成為共享變數，可以在不同線程之間存取，但因為平行化的特性，需要特別注意共享變數在不同線程中的同步。在本程式中，不同矩陣元素的計算是獨立的，沒有產生相依性，因此不需要進行特別的同步處理。

ThreadData 結構及全域（共享）變數之宣告如下

```
struct ThreadData {
    int rowStart;
    int rowEnd;
    int colStart;
    int colEnd;
    Method method;
};

// Global variables
MatrixDimension kernelDim, sourceDim, resultDim;
float *kernel, *source, *result;
```

建立線程及集合線程之程式碼如下

```
// Create threads
int createdThreadCount = std::min(MAX_THREAD_NUM, resultDim.height);
int rowsPerThread = ceil((double) resultDim.height / MAX_THREAD_NUM);
auto *threads = new pthread_t[createdThreadCount];
auto *params = new ThreadData[createdThreadCount];
for (int i = 0; i < createdThreadCount; i++) {
    params[i].rowStart = i * rowsPerThread;
    params[i].rowEnd = (i + 1) * rowsPerThread;
    if (params[i].rowStart > resultDim.height)
        params[i].rowStart = resultDim.height;
    if (params[i].rowEnd > resultDim.height)
        params[i].rowEnd = resultDim.height;
    params[i].colStart = 0;
    params[i].colEnd = resultDim.width;
    params[i].method = method;
    pthread_create(&threads[i], nullptr, run, &params[i]);
}

// Join threads
for (int i = 0; i < createdThreadCount; i++) {
    pthread_join(threads[i], nullptr);
}
```

線程函式：負責計算在指定範圍中之矩陣元素的 PCC 或 SSD 數值

```
// Thread function
void *run(void *arg) {
    auto data = (ThreadData *) arg;
    for (int i = data->rowStart; i < data->rowEnd; i++) {
        for (int j = data->colStart; j < data->colEnd; j++) {
            // Extract the image to be compared
            float image[kernelDim.width * kernelDim.height];
            for (int m = 0; m < kernelDim.height; m++) {
                for (int n = 0; n < kernelDim.width; n++) {
                    image[m * kernelDim.width + n] =
                        source[(i + m) * sourceDim.width + (j + n)];
                }
            }

            // Calculate the result based on the specified method
            if (data->method == PCC)
                result[i * resultDim.width + j] =
                    calculatePCC(kernel, image, kernelDim);
            else if (data->method == SSD)
                result[i * resultDim.width + j] =
                    calculateSSD(kernel, image, kernelDim);
        }
    }
    pthread_exit(nullptr);
    return nullptr;
}
```

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：3x3 圖像：3750x4320	PCC	1	2.0800170
		4	0.9157979
		16	0.3688751
	SSD	1	1.1269650
		4	0.4969767
		16	0.1995338
卷積核：5x5 圖像：7750x1320	PCC	1	3.1366060
		4	1.3094990
		16	0.5530632
	SSD	1	1.6856580
		4	0.7473268
		16	0.3136339
卷積核：3x3 圖像：8140x9925	PCC	1	10.388530
		4	4.2156500
		16	1.8174690
	SSD	1	5.6225300
		4	1.4437530
		16	0.9861535
卷積核：5x5 圖像：50x50	PCC	1	0.0008049
		4	0.0003449
		16	0.0005331
	SSD	1	0.0004844
		4	0.0003072
		16	0.0004150

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。此次實驗分別採用最多 1、4、16 個線程來執行，從上面的數據可以觀察出，**利用平行化可以提速約 5.7**（測資 3，PCC，1 vs 16 線程）。越多的線程看似可以提升速度愈多，但經過測試開啟超過 20 個以上的線程，不會再有明顯的提速，並且還可能有延長執行時間的狀況，事實上這與測資大小有關係，如測資 4 數據中，使用 16 個線程反而較 4 個來得更久，因此需要針對不同測資調整最大線程數，不過就一般情況來說，使用 16 個線程即可。

參、討論

相較於上次作業以 CUDA 來實作 GPU 平行化程式，本次作業可以較容易完成，主要的困難為：需理解並特別注意指標的運用，稍有不甚，便很容易存取到錯誤的位置，或是產生意外的結果。

本次作業平行化了計算 PCC 或 SSD 的部分，而找出最相似位置則是採普通無平行化的方法進行，主要考量到最相似位置可能不只一個，所以使用 C++標準容器 vector 來儲存位置。持續維護及更新最佳位置的部分其實也可以放到線程函式中達成平行化，但是會在不同線程間產生相依性，需要透過互斥鎖及條件變數等方案來避免衝突，不過這也使得計算時間大幅增加，故最終捨棄了這個實作方案。

肆、參考

[1] 課程簡報

[2] <pthread.h>