Accelerating Block-Matching Disparity Map Computation With Parallelization*

A comparison of CUDA, OpenMP and pthread implementations

Chao Hong Yeh
Institute of Data Science and Engineering
National Yang Ming Chiao Tung University
Hsinchu City, Taiwan
yeh.fela@gmail.com

Yu Siou Lin
Institute of Data Science and Engineering
National Yang Ming Chiao Tung University
Hsinchu City, Taiwan
ericlin66.cs11@nycu.edu.tw

Shih Hsuan Kao
Institute of Computer Science and Engineering
National Yang Ming Chiao Tung University
City State Country
sam8409044601@email.com

INTRODUCTION(motivation)

在電腦視覺領域中，可以透過數張拍攝角度相近的影像，計算出拍攝景象對應的深度圖(Depth map)。研究者發現可以透過計算視差圖(disparity map)以及相機的資訊(如:焦距)得到深度圖。計算視差圖的想法來自人類左右眼觀測景象，景象若離人眼越近，則在左右眼成像的位置差異越大，或稱視差越大；同理可知，距離人眼越遠，則在左右眼成像的位置差異越小，或稱視差越小。

對於電腦而言，還需要判定景象是否相同，才能計算視差。更客觀的說法，電腦如何確定左圖的某個 pixel 位於右圖的何處。研究者指出，可以透過校正左右兩圖，使得左圖的每個 pixel 其在右圖對應的位置一定在相同的高度上，如此便可以減少計算時間。而如何檢驗左右兩個 pixel 的相似度，則需要給定一個 loss function 幫助我們尋找對應的 pixel。在以往的研究中，許多研究者紛紛提出可以採用的 loss function。本篇文章採用的 loss function 稱作 Sum of Absolute difference(後面簡稱 SAD)，其想法為針對左圖的每一個 pixel，將其視為中心，並以其鄰近範圍形成一個區塊，同一時間右圖相同高度的每一個 pixel 也都分別當作中心，形成數個區塊。而 SAD 的想法便是用來評估兩個區塊的相似程度，將兩區塊對應位置的 pixel value 進行相減後取絕對值並將所有差值加總，若加總的值越小，代表兩個區塊越相近。利用此一想法，便可以將左圖的區塊，與右圖所有候選的區塊進行相似度比較，其中相似度最大的候選區塊，其中心位置就視為左圖區塊中心位置對應到右圖的位置。之後將對應 pixel 的水平位置減去左圖 pixel 的水平位置取絕對值，即可得到該 pixel 的視差。

從上述可以得知，SAD 的計算並不複雜，但是將左圖所有 pixel 的視差都計算完，仍需要許多運算，而先前我們曾用 python 實作過該問題，花費時間多要六至九分鐘不等(根據自行定義的區塊大小有所差異)。經過觀察，我們發現每個 pixel 在計算視差時，彼此並沒有 data dependency 的問題，因此我們想要嘗試透過本堂課學習到的平行化工具，幫助我們縮減電腦計算上花費的時間，從而得到更好的效能。

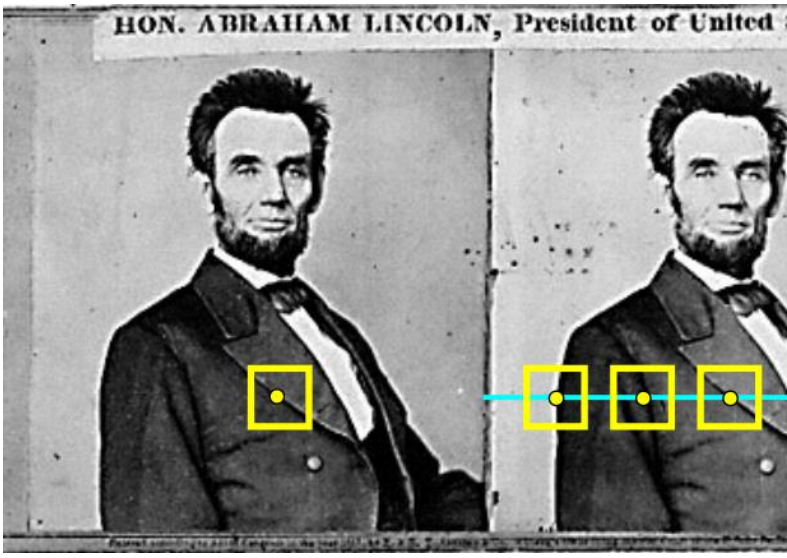

Figure 1: The image shows how to find do the SAD algorithm.

KEYWORDS

Keywords: stereo、cuda、Pthread、OpenMP

Statement of the problem:

stereo 程式與平行化部分介紹

Proposed approaches, which include the block diagram(pseudo code explanation)

Figure 2 是使用 Block-matching method 來計算出 Disparity map 的 pseudo code，可以看到主要有三層的 for loop，我們很直覺的就想到可以分別對這三個 for loop 做平行化，各個平行化方法如下說明：

(1) 方法一：針對第一層 for loop 做平行化， 此 for loop iterate left image 中的 Y 座標，意即我們要將 left image 中不同的 row 來分配給各個 thread 去做計算，每個 thread 對於分配到的 rows 去計算這些 rows 上每個 pixel 的 disparity。

(2) 方法二：針對第二層 for loop 做平行化，此 for loop iterate left image 上某一列的 X 座標，意即我們要將 left image 中某一 row 上的各個 pixel 分配給各個 thread 去做計算，每個 thread 對於分配到的 pixels 去計算這些 pixels 的 disparity。

(3) 方法三：針對第三層 for loop 做平行化，此 for loop iterate right image 上的 candidate pixel，意即我們要將 right image 中的 candidate pixels 分配給各個 thread 去做計算，每個 thread 對於分配到的 candidate pixels 去計算該 candidate pixel 對應 block 與左圖 template 之 SAD 。

```
ComputeDisparity:
  For each Y in ImageL:
    For each X in ImageL:
      Compute the template size
      Compute the number of candidate pixels
      Losses = []
      For each candidate pixel:
        Compute Disparity Metric over candidate pixel neighborhood and root neighborhood
        Append Disparity Matric to Losses

      BestIndex = argmin(Losses)
      CX = candidate pixel[BestIndex].x

      Disparity[X,Y] = ABS(CX - X)
```

Figure 2: Pseudo Code for Disparity Map Computation using Block-Matching Method

Proposed approaches and Language selection

Source Code:

https://github.com/yehfelareborn/Parallel-Programming-NYCU-Final_Project.git

2 Pthread、OpenMP、Cuda 實作

2.1 OpenMP implementation

針對上面三種方法，分別照以下方式實作:

(1) 方法一：在第一個 for 迴圈前加入#pragma omp parallel for，企圖讓每個 row 的所有 pixel 由同一個 thread 來管理，並且每個 thread 分配到的 row 的數量差異不會太大。每個 thread 也會將計算好的結果寫入 disparity map。

(2) 方法二：在第二個 for 迴圈前加入#pragma omp parallel for，企圖讓每個 row 的 pixels 由不同 threads 分工完成計算 disparity。

(3) 方法三：由於每個 pixel 要計算其 disparity 都需要計算多個 block 與左圖該 ixel 形成的 block 進行相似度計算，因此在最後一個回全加入#pragma omp parallel for，企圖讓不同 thread 分別代表不同的 candidate block 與左圖的 block 進行計算 ，最後找到最接近的 block 的中心，並將結果寫入 disparity map。

2.2 Pthread implementation

針對上述三種方法，實作方式分別如下說明：

(1) 方法一：先開始建立 threads，將 left image 中總 row 數除以所指定 thread 數量，計算出平均每個 thread 所需計算 row 數，並將剩餘 rows 分配給 master thread，最後每個 thread 分別執行完成即得 disparity map。

(2) 方法二：在每次選定 row 後，開始建立 threads ，將 left image 中總 column 數除以所指定 thread 數量，計算出平均每個 thread 所需計算 column 數，並將剩餘 columns 分配給 master thread，最後每個 thread 分別執行完成後，再依相同方式繼續進行 left image 中下一 row 的計算，待所有 row 都計算完成後即得 disparity map。

(3) 方法三：在每次選定 left image 中的某一 pixel 後，開始建立 threads，將其所對應 right image 中的 candidate pixel 數除

以所指定 thread 數量，計算出平均每個 thread 所需計算 candidate pixel 數，並將剩餘 candidate pixel 分配給 master thread，每個 thread 分別計算其所分配到的 candidate pixel 所對應 block 與 left image 中 template 的 SAD，並依此求出最小 SAD 值及其對應之 candidate pixel，最後每個 thread 執行完成後，再從所有 thread 的最小 SAD 值取出最小值及其對應之 candidate pixel，再依此 candidate pixel 來計算出 disparity，依相同方式繼續進行 left image 中下一 pixel 的計算，待所有 pixel 都計算完成後即得 disparity map。

2.3 Parallel with Cuda (GPU)

Cuda 部分能夠以資料為單位進行最大的資料平行化，同時，在 index 上需要更多的著墨。

首先，在 Linux 系統上執行 C++ 的 OpenCV library 時，無法以 .cu 檔的方式 import opencv，需要另外建立一個 stereo.cu 檔並 link 進 cmake 的 compile 過程中。

接下來是把 block match 以 parallel 形式在 GPU 上執行並且加速。起初，我們試著以 **Pthread method3** 的形式，在兩個不同 window 的部分進行平行計算，發現到 data allocation 的時間過長，甚至達到了 **serial 的 4 倍時間**，於是更改為把兩張目標圖片放進 GPU 後，以 threadID 的形式作為 index 進行計算

```
// 根據 CUDA 模型，算出當下 thread 對應的 x 與 y
const int idx_x = blockIdx.x * blockDim.x + threadIdx.x;
const int idx_y = blockIdx.y * blockDim.y + threadIdx.y;
int idx = idx_y * 881 + idx_x;
if(idx_y>2&&idx_y<402){
    if(idx_x>2&&idx_x<879){
    int cxDstMin = max(idx_x - halfSearchRange, halfWinSize); //left edge
    int cxDstMax = min(idx_x + halfSearchRange, 881-halfWinSize);//right edge
```

Figure 3: Kernel function and its threadID to compute block matching

```
void stereoMatch(const cv::Matlf &imgSrc, const cv::Matlf &imgDst, cv::Matlf &disparity) {
    float* d_imgSrc; float* d_imgDst; float* d_disparity;
    cudaMalloc(&d_imgSrc,  imgSrc.total() * sizeof(float));
    cudaMalloc(&d_imgDst,  imgDst.total() * sizeof(float));
    disparity = cv::Matlf::zeros( imgSrc.size() );
    cudaMalloc(&d_disparity,  disparity.total() * sizeof(float));
    cudaMemcpy(d_imgSrc,  imgSrc.ptr<float>(0), imgSrc.total() * sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpy(d_imgDst,  imgDst.ptr<float>(0), imgDst.total() * sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpy(d_disparity,  disparity.ptr<float>(0), disparity.total() * sizeof(float), cudaMemcpyHostToDevice);

    dim3 blockSize(BLOCK_SIZE, BLOCK_SIZE);
    dim3 numBlock(881 / BLOCK_SIZE+1, 400 / BLOCK_SIZE+1);
    Calculate<<<numBlock, blockSize>>>(d_imgSrc, d_imgDst, d_disparity, halfWinSize, halfSearchRange);
    cudaDeviceSynchronize();
    cudaMemcpy(disparity.ptr(), d_disparity,  disparity.total() * sizeof(float), cudaMemcpyDeviceToHost);

    cudaFree(d_imgDst);
    cudaFree(d_imgSrc);
    cudaFree(d_disparity);
}
```

Figure 4: allocate space on GPU and call the kernel function

3 效能比較與評估 (含表格)

Pthread 和 OpenMP 的方式中，我們實作了三種不同的 method，以及兩種形式的 threads，如同 Table 1和 Table 2 所呈現，加速結果為將執行五次之平均時間與 serial 版本時間做比較所得。

3.1 Pthread

從 Table 1 中可以看到 Method 1 的 speedup 是最佳的，Method 2 次之，最差的為 Method 3。經過分析後，我們發現 Method 2 較 Method 1 慢的可能原因為其平行化方式是每個 thread 分別計算每個 pixel，然而每個 pixel 所對應的 search range 並不相同，而造成有些 thread 的 workload 較重，成為效能瓶頸。而 Method 3 在 thread 數量為 4 時，執行時間幾乎與 serial 一致，沒有達到加速效果，甚至在 thread 數量為 8 時，執行時間比 serial 多了一倍左右，造成此現象的可能原因為其平行化方式是每個 thread 分別計算 candidate pixel 與 template 的 SAD，然而此計算的 workload 是非常小的，因此導致 create thread 和 destroy thread 的 overhead 遠大於平行化所帶來的效能提升，這也說明了為何 method 3 與其他 method 不同，thread 的數量上升反而導致其 speedup 下降。

Table 1: Thread and Method Effects on Pthread Implementation

	# of threads	Speedup
Method 1	4	31.26
	8	57.99
Method 2	4	28.45
	8	48.28
Method 3	4	1.03
	8	0.44

3.2 OpenMP

Table 2: Thread and Method Effects on OpenMP Implementation

	# of threads	Speedup
Method 1	4	3.84
	8	6.90
Method 2	4	3.75
	8	7.40
Method 3	4	3.59

	8	5.77

由 Table 1、Table 2
中，可看出不同地方的平行化，會導致結果上很大的不同。Method1、Method 2分別對 row、column 作平行化，在 pthread 中就有著很顯著的提升，然而，method3 針對 window 計算的方式，由於計算上需要多次的資料移動，速度上有著明顯的拖沓，**甚至時間還比 serial 的運行時間要來的久**

在 OpenMP 的方式中，也如同 pthread
會遇到的平行化區域不同所帶來的問題，Method 3 相較於 Method 1, 2 要來的慢。

最後是 cuda 的部分

3.3 CUDA

Table 3: Block Size Effects on CUDA Implementation

BLOCK_SIZE	Speedup
4	70.7
32	95.7

可以由圖中看出，cuda 的平行效果最好，甚至能獲得近乎 100 倍的加速，同時，他也是對於圖片中的所有 pixel 去進行平行化

Conclusion

本次專題結果發現，由於block mathcing的計算方式沒有data dependency，因此使用平行化工具加速普遍可以獲得顯著的效能提升，其中以CUDA的加速效果最為明顯。惟時間有限，該專題尚有許多部份可以深入探討，例如:區塊大小不斷增加的情形下，平行化工具都依然保有穩定的效能提升嗎、能否與openCV的API進行效能上的比較，不同的loss function是否都能有顯著的效能提升等等，都是可以延續研究下去的主題。

Reference

$$SAD\left(B^L, B^R\right) = \sum_{i=1}^{M}\sum_{j=1}^{N}\left|B_{ij}^L - B_{ij}^R\right|$$

關於SAD的演算法如上面式子展示，左邊的block與右邊block對應位置進行相減取絕對值，並將所有block內的差值加總，即可得兩個block的loss為何。

REFERENCES

[1] How to find and link CUDA libraries using CMake 3.15? https://stackoverflow.com/questions/66327073/how-to-find-and-link-cuda-libraries-using-cmake-3-15
[2] Block-matching algorithm “https://en.wikipedia.org/wiki/Block-matching_algorithm”