國立交通大學

資訊科學與工程研究所

碩士論文

利用圖形處理晶片研究共同資訊量的影像對位

Mutual Information Image Registration Applications On Graphic Processing Unit

研 究 生：李枘旭

指導教授：荊宇泰　教授

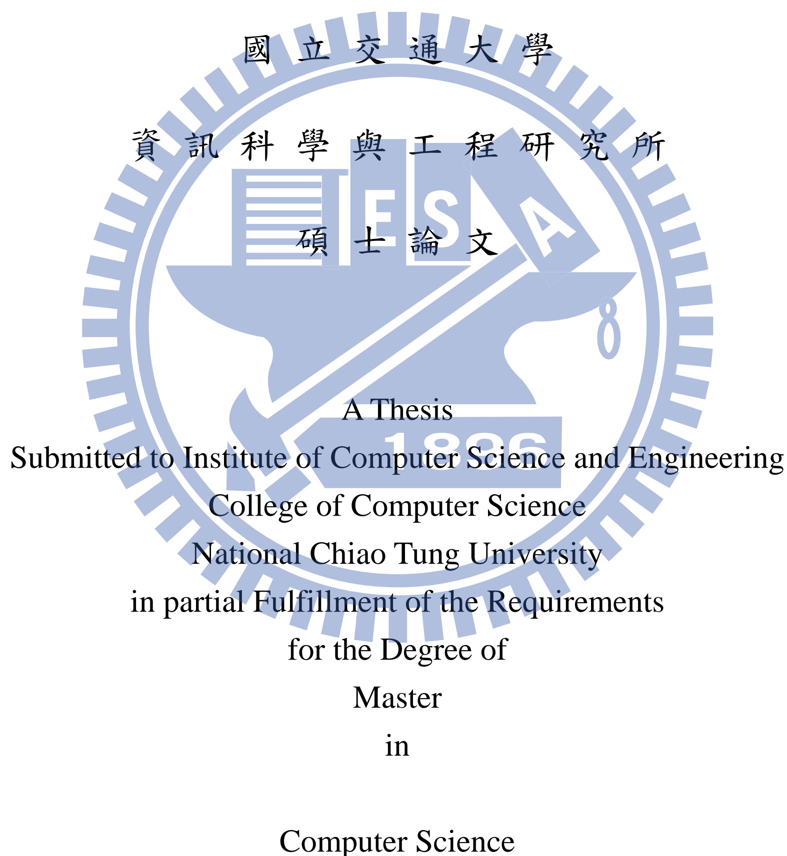
中華民國九十八年十月

利 用 圖 形 處 理 晶 片 研 究 共 同 資 訊 量 的 影 像 對 位

Mutual Information Image Registration Applications On Graphic Processing Unit

研 究 生：李柟旭 Student：Nan-Shiu Lee

指導教授：荊宇泰 Advisor：Yu-Tai Ching



國 立 交 通 大 學

資 訊 科 學 與 工 程 研 究 所

碩 士 論 文

A Thesis
Submitted to Institute of Computer Science and Engineering
College of Computer Science
National Chiao Tung University
in partial Fulfillment of the Requirements
for the Degree of
Master
in

Computer Science

October 2009

Hsinchu, Taiwan, Republic of China

中華民國九十八年十月

利用圖形處理晶片研究共同資訊量的影像對位

學生 ；李柟旭　　　　　　　　　　指導教授:荊宇泰 博士

國立交通大學資訊科學與工程研究所

摘　要

近年來圖形處理晶片的計算速度非常快的增加,用圖形處理晶片加速各種計算是一個很好的選擇,而影像對位是一個很重要的影像處理工具，其主要的應用是在匹配兩張或是多張影像上，在許多大型的影像處理系統都使用著這樣的技術。在本篇論文中，我們藉由 “共同資訊量” 的最大化來做影像對位的研究，並將該技術用在生物影像處理。我們的實驗資料來自果蠅腦以及生物影像的細胞，用圖形顯示卡實作 2d 和 3d 的影像對位。

Mutual Information Image Registration Applications On Graphic Processing Unit

Student：Nan-Shiu Lee Advisor：Yu-Tai Ching

Institute of Computer Science and Engineering

National Chiao Tung University

ABSTRACT

In the recent decade, the computing capacities of the graphics processor units (GPUs) has a great improvement . GPU-accelerated computation is a good option for many applications. Registration is a important tool in image processing that is applied to match regions in two or more pictures. Virtually all large systems which evaluate images require the registration of images. In this thesis, image registration technique by maximization of mutual information was studied. The technique was applied to biological images. Our experimental data are fly brain and cell of biological image . We use graphics processor units to implement 2d and 3d image registration.

目錄

第一章 緒論 1
1.1 簡介........ 1
1.2 論文架構........ 2
第二章 影像對位和 cuda 簡介 3
2.1 前言 3
2.2 cuda 簡介........ 3
2.3 影像處理簡介........ 8
第三章 Mutual Information 11
3.1 前言 11
3.2 Entropy(熵) 11
3.3 共同資訊(Mutual Information) 12
3.4 MI 特性........ 14
第四章 CUDA 實作流程 15
4.1 實作簡介........ 15
4.2 計算 entropy 的概念 15
4.3 計算 MI 與影像對位的實作 19
4.4 細胞的 Tomography Reconstruction 25
第五章 實驗結果 27
5.1 實驗環境........ 27
5.2 部份腦對位........ 27
5.3 細胞........ 28
第六章 未來展望 36

圖表

圖 2-1　GPU 效能和 CPU 效能對比 4
圖 2.2　CUDA 執行架構 5
圖 2.3　CUDA 記憶體架構 7
圖 2.4　CUDA 軟體架構....................8
圖 2-5　registration 流程圖 10
圖 4-1 MI 流程示意圖....................20
表 4-1 Table1 鄰近偏移量....................26
表 4-2 Table2 的偏移量為累積偏移量相對於 Reference Image 要做的偏移量....................26
圖 5-1 (a) 特徵點框架 (b) 特徵點對位結果....................27
圖 5-2 (a) 細胞一第 10 張 (b) 第 71 張....................29
圖 5-3 (a) 細胞二第 10 張 (b) 第 71 張 31
圖 5-4 (a) 細胞三第 10 張 (b) 第 71 張....................33
圖 5-5 (a) 細胞四第 10 張 (b) 第 71 張....................34

第一章 緒論

1.1 簡介

近年來圖形顯示卡的計算速度非常快的增加，用圖形顯示卡加速各種計算是一個很好的選擇，最近幾十年，影像對位相關演算法的研究被廣泛的發表，影像對位(Image Registration)是一個很重要的影像處理工具。本篇論文主要應用共同資訊(Mutual Information)的演算法來做影像對位，其主要應用在匹配(match)兩張或是多張影像上，在許多大型的影像處理系統都使用著這樣的技術，常見的例子像是；醫學上影像臨床觀察、即時影像追蹤、衛星影像觀測等[1]。

我們的實驗資料是果蠅腦的共軛焦顯微鏡(confocal microscopy)影像以及生物影像的細胞。在果蠅腦上我們圈出特徵點進行 3d 影像對位，用來做果蠅腦之間特徵點的對齊。在生物影像的細胞我們進行 2d 影像對位，用來對齊細胞多角度投影影像的圖形，解決影像的偏移震動。

1.2 論文架構

本篇論文共分為六個章節。第一章為緒論，其中描述我們的研究的動機以及研究目標。第二章將說明影像對位(Image Registration)以及 cuda 的架構。第三章將詳細介紹共同資訊(Mutual Information)演算法來應用在影像對位上。第四章為利用 cuda 實作的流程與步驟。第五章則是實驗結果。而第六章為結論與未來展望。

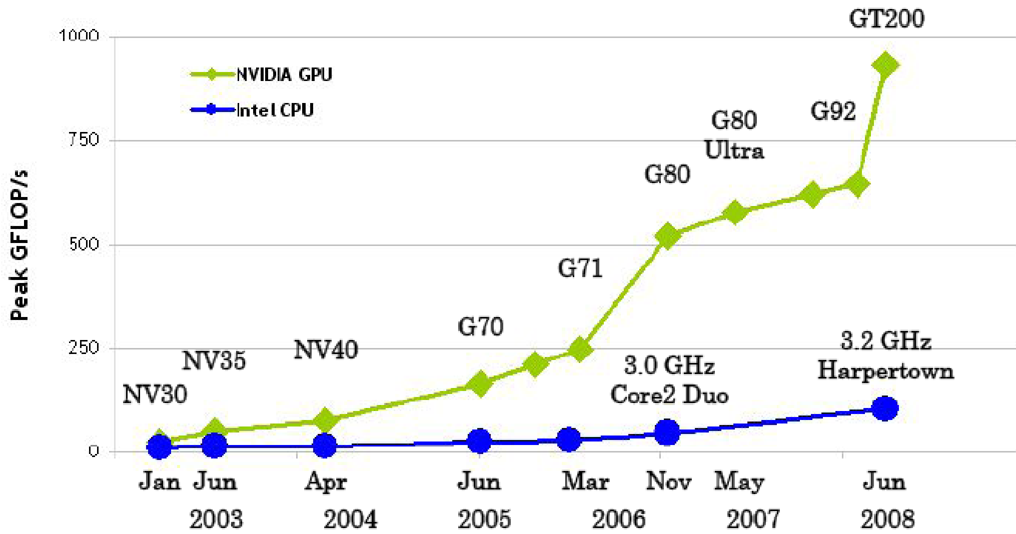
第二章 影像對位和 CUDA 簡介

2.1 前言

這個章節我們將介紹 CUDA 的 GPGPU 模型，並且我們將描述影像對位(Image Registration)的基本概念，並且介紹他們的重要性。

2.2 CUDA簡介

GPGPU(General-Purpose computation on GPU)是以 GPU(Graphic Processing Unit)來進行通用的計算，而不是只局限在把顯示卡用在於圖形的顯示上，隨著市面上的 GPU 可程式化程度的提升，這些晶片原本被設計作為特定的圖形運算用途，到現在則可以被用來解決許多數學運算上的問題，在某些方面的應用上，甚至已經大幅的超越 CPU 了，圖 2.1(取自 cuda 官方文件)可以看出來 GPU 和 CPU 的速度演進，單就浮點數的計算來說，GPU 的成長速度已經算是 CPU 的數倍了。原則上，由於 GPU 的架構設計，GPU 最適合拿來做的是平行化的處理，適用於 GPGPU 的問題，大多是用在可以把一個問題大量的拆解彼此資料上不相關、且可平行化計算的情況。在這種情況下，用 GPGPU 的方法，就可以把我們的運算用 GPU 來大量平行化的處理，藉此達到加速的效果。nVidia 研發的 CUDA(Compute Unified Device Architecture) 是一種 GPGPU 的技術；cuda 平台是透過 C 語言的函式庫和一些 CUDA 的函式延伸出來的，因為不用使用圖形函式庫，而不會被傳統的 render pipeline 綁住，使得在程式設計上更方便。



GT200 = GeForce GTX 280	G71 = GeForce 7900 GTX	NV35 = GeForce FX 5950 Ultra
G92 = GeForce 9800 GTX	G70 = GeForce 7800 GTX	NV30 = GeForce FX 5800
G80 = GeForce 8800 GTX	NV40 = GeForce 6800 Ultra	

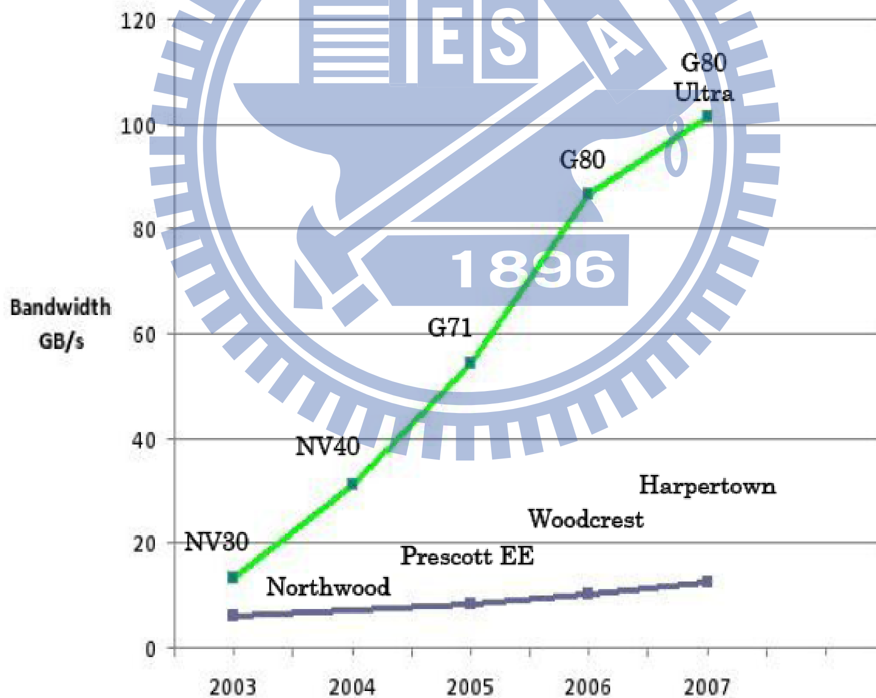
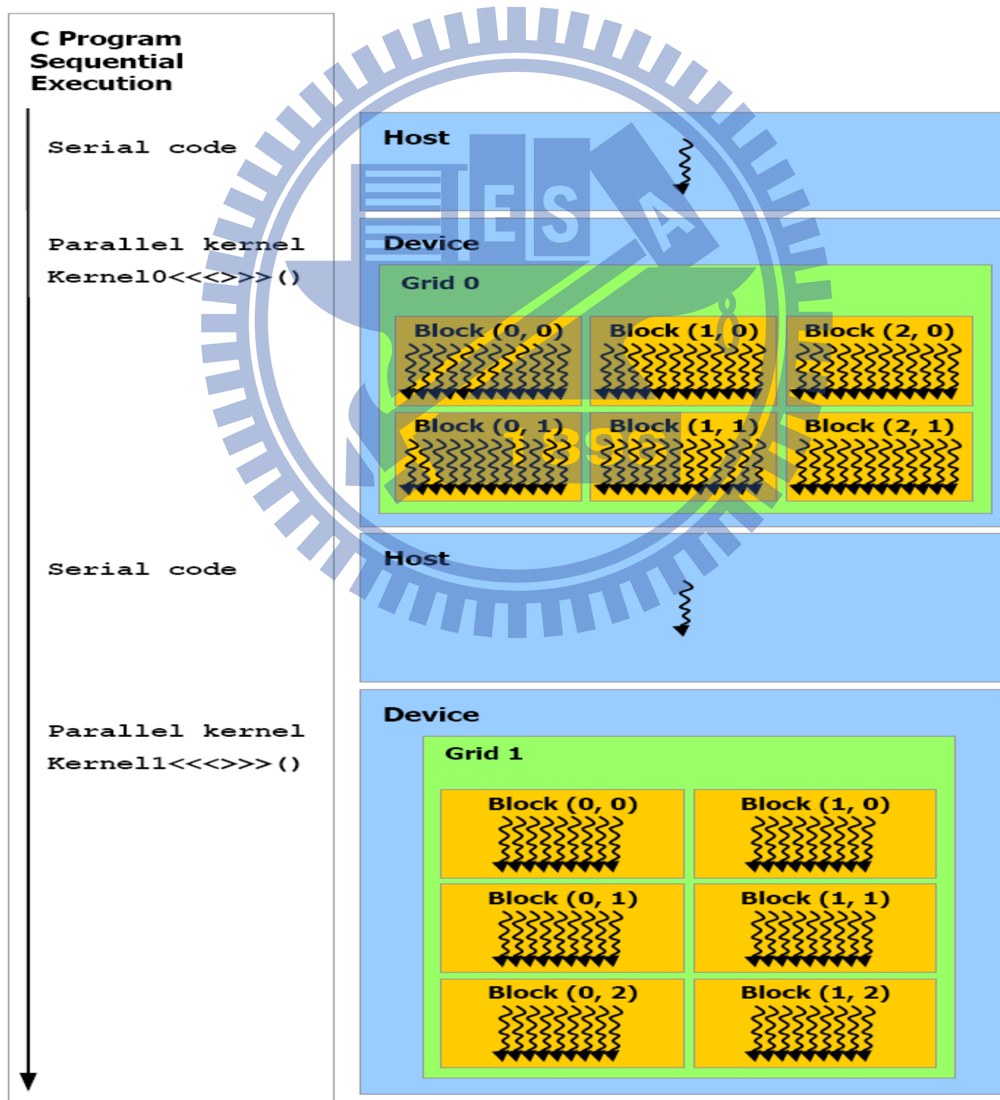


圖 2-1　GPU 效能和 CPU 效能對比

CUDA 的 GPGPU 是採取 SIMD 架構[2]，也就是不同的資料，要用同樣的程式來做計算。而在這種情況下，就可以透過把每一組資料的計算，都用一個 thread

去做，在 device 中要執行的 thread 又可以分成好幾個 block，其中 thread block 的型式最多可以到三維。以此平行化來加速計算。其中，演算法的平行度也要高，如此才能避免平行化的負擔大於平行化的加速。

在 CUDA 的程式架構裡，程式執行的區域會分成 Host 和 Device 其中，host 指的就是 CPU，而「device」就是 GPU 了在 CUDA 的程式架構中，主程式還是由 CPU 來執行(圖 2.2 取自官方文件) 。



Serial code executes on the host while parallel code executes on the device.

圖 2.2 CUDA 執行架構

CUDA的程式是遇到了資料要平行化處理的部分，就會將要在 GPU 跑的程式編譯成 device 能執行的程式，再丟給 device 執行了，而這個程式在 CUDA 裡把他叫做kernel。 CUDA 的是標準的single instruction multiple data (SIMD)架構，這代表了會產生許多在 device 上執行的 thread，每一個 thread 都會去執行 kernel 這個程式；雖然程式都是同一份，但是會因為其 thread index 的不同，而取得不同的資料來計算。而在同一個 block 裡的 thread，有部分的記憶體（shared memory）是共用的，（shared memory）是cuda加快整體效能的核心；shared memory是內建在晶片上(built on-chip) ，而且可以非常快速的被存取，可以大幅的增加記憶體的存取速度，而由於 thread block 的最大大小是有限制的，大約是512個，所以一般來說不會把所有的 thread 都塞到同一個 block 裡。所以在同一個kernel裡面，必須用同樣維度和大小的 thread block，來組成一個 grid 做批次處理。這個 grid是指block的數目，同樣的可以是一維或二維陣列的形式。所以，實際上 CUDA 在執行一個 kernel 程式的時候，必須要指定他的 grid 和 block 的相關資訊（維度和大小）；而會產生的 thread 數目，就會由這兩者的資訊來決定，這必須是固定的，同一個kernel不能夠變換他的block跟threads的數目。

一個CUDA實作大致上包含了以下五個步驟:

1.在GPU上宣告出我們想要的記憶體空間，並且初始化GPU上的資料

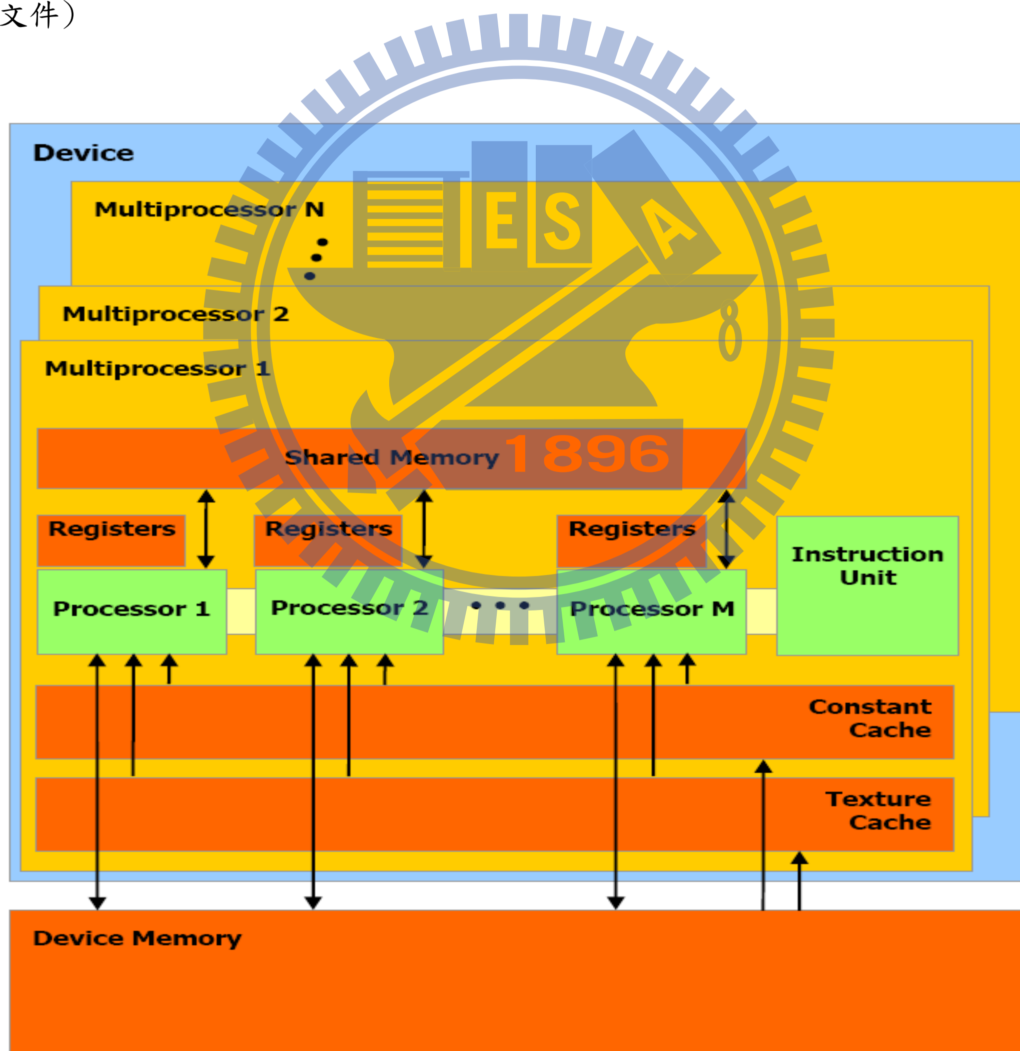
2.把資料從CPU的記憶體上傳到GPU的記憶體空間

3.決定kernel的gridDim (要有多少個blocks)和blockDim(每個block有多少個thread)

4.執行kernel結果會存在GPU的memory內

5.把我們想要的結果從GPU的記憶體送回到CPU的記憶體

在 CUDA 中，要讓 thread 可以使用的變數，都必須要先把資料傳送到 device 的記憶體裡，而 device 的記憶體，記憶體架構可分為 DRAM 和 chip 上的記憶體兩種。在使用的時候，global memory 是存在於 DRAM，可以由整個 kernel 裡面的所有 thread 讀取跟存入，但是讀取跟存入的速度較慢。local memory 如果用的記憶體少則會存取在是在 chip 上，如果用到的記憶體多則是存在於 DRAM 上。shared memory 是內建在晶片上(built on-chip) ，shared memory 的記憶體空間只有 16KB，可以非常快速的被存取，並且大幅的增加記憶體的存取速度，texture 則是唯獨的記憶體，也是在 DRAM 上但是讀取速度較快。(圖 2.3 取自官方文件)



A set of SIMT multiprocessors with on-chip shared memory.

圖 2.3 CUDA 記憶體架構

CUDA 的軟體架構大概分為 Library、runtime、Driver 三個部分；而我們在開發程式的時候，可以透過這三個部分，來使用 GPU 的計算能力。(圖 2.4 取自官方文件)

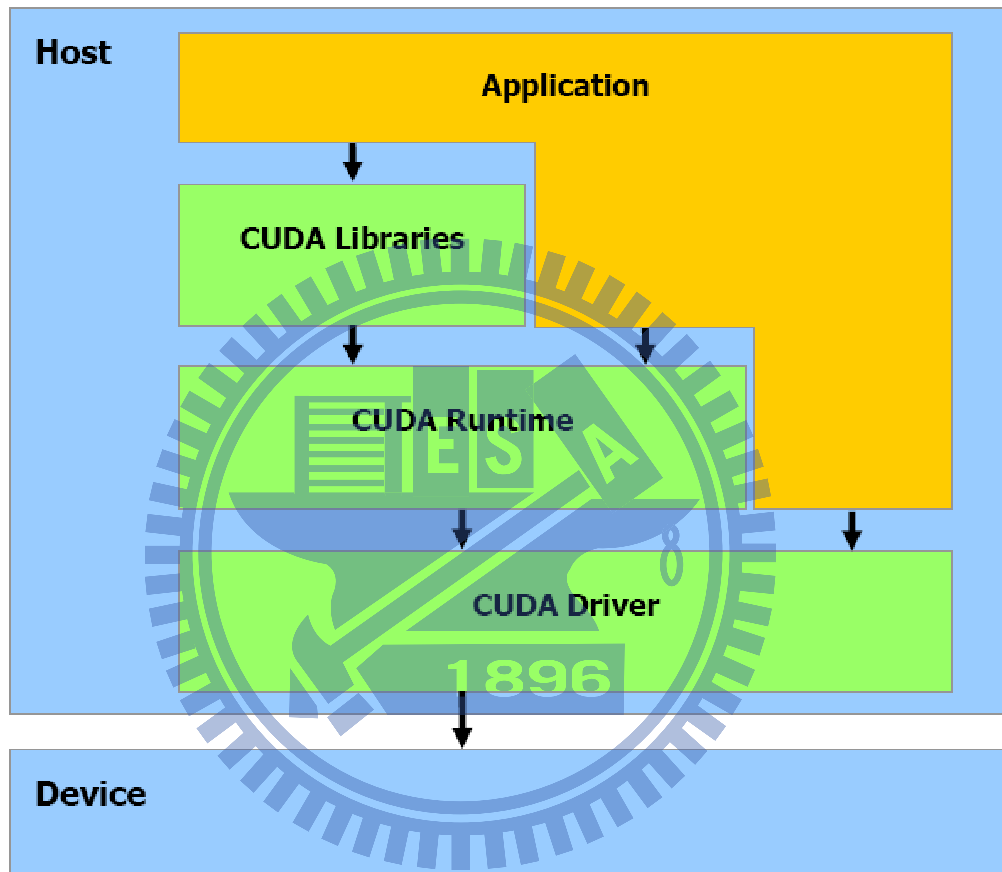


圖 2.4　CUDA 軟體架構

2.3 影像對位簡介

影像對位(Image Registration)是一種方法用來了解兩張影像間彼此特徵的對應關係，在許多大型的影像處理系統都使用著這樣的技術。影像對位被認為是一個基本的技術用來分析醫學影像，因為當空間上的相對應關係被清楚的定義之後，我們可以整合它所獲得的資訊，以利於很多影像的研究。

在醫學影像中，由於影像資料的成像可能由多種方式產生，這樣的分類主要來自於利用不同的儀器，產生出相對應的影像，影像模型大概可以分為二個種類：結構型(Anatomical)與功能型(Functional)。結構型影像就例如；X-ray、電腦斷層掃描(Computed tomography)、核磁共振(Magnetic Resonance Imaging)之類的影像結構；功能型影像；正電子發射電腦斷層掃描(Positron Emission Tomography)單光子射出電腦斷層掃描(Single Photon Emission Computed Tomography)等，包含了器官或是組織功能的內部資訊。

三維(3D)影像對位的流程是將其中一組影像資料 A(稱為 Floating Image or Transformed Image)選出一組空間轉換的參數與另一組資料 B(稱為 Reference Image)找出最相似的轉換過程[3]。這個演算法將不斷的調整空間上的參數去做影像 A、B 相似度(Similarity Measure)的測量，循序的找出最大的相似度，這篇論文中我們是以 Mutual Information 來判斷相似度。

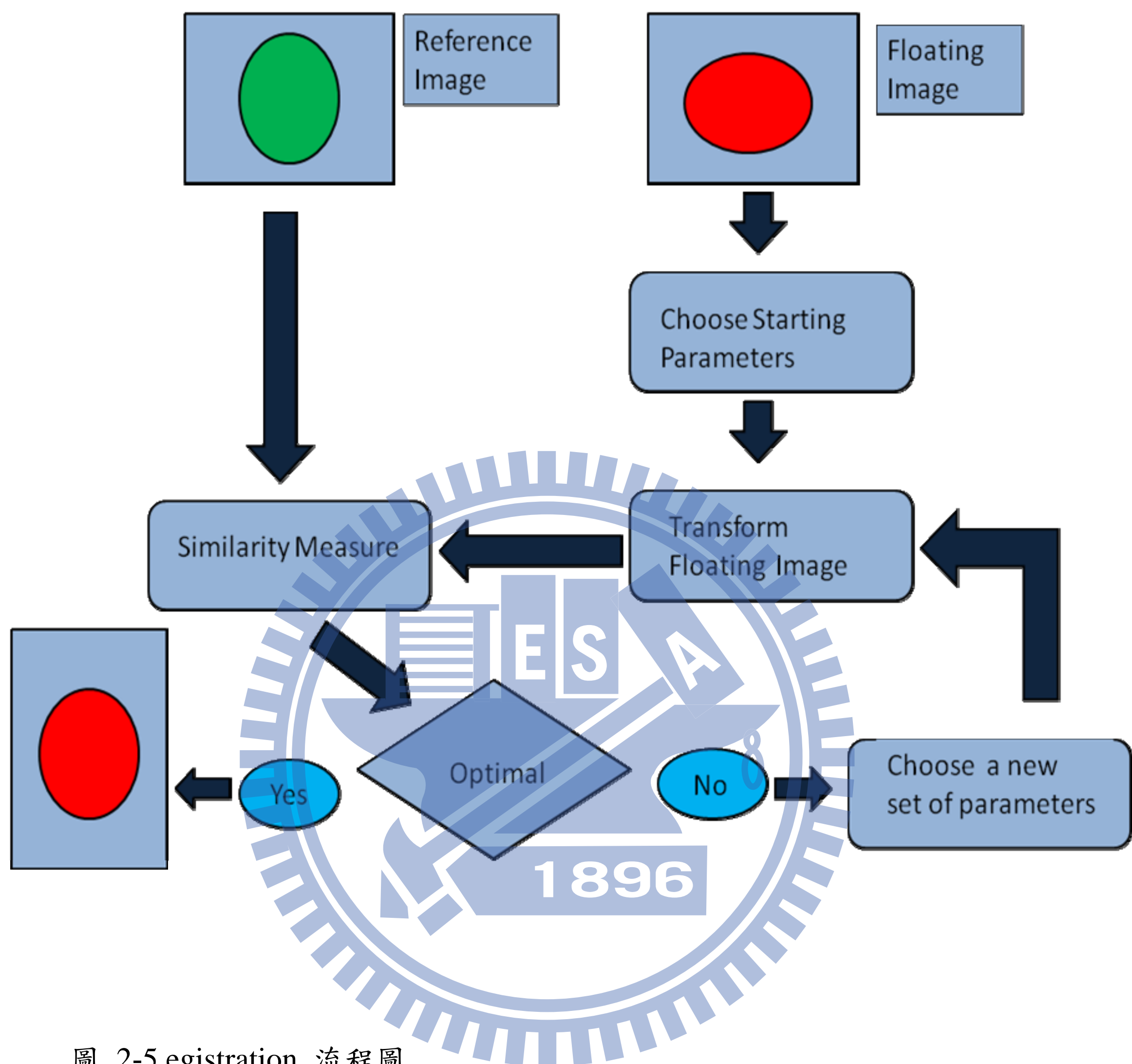


圖 2-5 egistration 流程圖

在最近十年，越來越多研究者對於資訊理論的測量感到興趣，我們可以把一張影像當做一種資訊，而影像對位(Image Registration)可以被想成求取最大的兩張影像的共有資訊，最常被使用到的資訊量測量方法為起源於通訊理論的 Shannon Entropy，我們將在第三章會詳細介紹。

第三章 Mutual Information 簡介

3.1 前言

利用最求取最大的 Mutual Information (MI)的方法來完成 Image Registration 將這章節詳述。使用 MI 很大的好處就是不需先對影像做任何處理的動作，不必先做影像分割或是灰階值的調整，可以直接拿 raw data 做 MI 的運算。一開始我們先簡介 Entropy 理論，接著介紹 MI 跟 Entropy 的關係，以及 MI 的特性。

3.2 Entropy(熵)

在資訊理論上，entropy 是一種度量資訊量的標準，也就是代表一個隨機變數的資訊量的含量[4]，entropy 所描述的資訊量可以被視為是不確定性(uncertainty)，當資訊量越大時，代表我們會有更大的不確定性。在 1948 年 Shannon 定義出一種的資訊量測量的方法[5]，成為現在 entropy 的定義，如果有一個系統 S 內存在多個事件 $S = \{E_1,...,E_n\}$，每個事件的機率分布 $P = \{p_1, ..., p_n\}$，Shannon Entropy 的定義如下：

$$H = \sum_i pi \log \frac{1}{pi} = -\sum_i pi \log pi$$

Shannon Entropy 的的定義中，$\log \frac{1}{pi}$這項表示著當 pi 越大的時候獲得的資料量越小[1]，這也就是說發生機率較低的事件提供了更多的資訊量，這是因為發生機率較低的事件我們比較難預測他什麼時候會發生，不確定性也就增加了，而想反的發生機率較高的事件則很容易預期他們什麼時候會發生，所以不確定性就下降，要用來表示他們的資料量也就可以降低了。當一組隨機變數當中如果每

個事件發生的機率愈平均，我們就比較不容易預期哪個事件會發生，不確定性就上升，entropy 也就比較大，如果其中幾個事件發生的機率很大，那不確定就會隨之下降，entropy 也就會比較小。

在影像處理上我們把影像當作一個隨機變數，把不同的灰階值當作不同的事件，利用計算影像的 histogram 來求出每個灰階值出現的機率，如此就可以利用 Shannon Entropy 的定義去求出影像的 entropy。

3.3 共同資訊(Mutual Information)

我們在 3.2 提到了 entropy 的定義，利用 entropy 我們可以進而求出 Mutual Information，計算 Mutual Information 主要表示兩個隨機變數的相關聯程度，Mutual Information 愈大表示兩個隨機變數的關聯程度愈大。MI (Mutual Information)起源於通訊理論的發展，後來才應用在影像對位上，我們可以把影像當成隨機變數。首先，兩個隨機變數 A、B 由 Shannon Entropy 熵 $H(A)$和 $H(B)$可定義成：

$$H(A)=-\sum_{a\in A} P_A(a)\log P_{A(a)}$$

$$H(B)=-\sum_{b\in B} P_B(b)\log P_{B(b)}$$

$P_{A(a)}$及 $P_{B(b)}$分別代表影像 A 與 B 之臨界機率分佈(Marginal Probability Distribution，MPD)。

計算聯合熵(Joint Entropy)則是當 A 影像與 B 影像之聯合機率密度函數 $P_{AB(a,b)}$已知，則其聯合熵可利用以下公式獲得

$$H(A,B)=-\sum_{a\in A}\sum_{b\in B} P_{AB(a,b)}\log P_{(a,b)}$$

計算 Joint Entropy 需要的是統計 joint histogram，也就是統計兩張影像中相對應點的灰階值對(gray level pair)累積的次數，比如說在同一個位置 A 圖的像素值是 5，而 B 圖的像素值是 10，我們就把(5,10)這一個 pair 存進我們的 joint histogram。

計算共同資訊（Mutual Information）

共同資訊可用來量測一個變數包含在另一個變數的資訊量。得到 A 影像與 B 影像之臨界熵及聯合熵後，A 與 B 影像之共同資訊即可依下式計算求得。

$$MI(A,B) = H(A) + H(B) - H(A,B)$$

藉由取共同資訊的最大值，而找到使得影像對位的空間轉換T_α最理想的參數 t：

$$T_\alpha = \mathrm{argmax} I(A,B^t)$$

$I(A,B^t)$為影像 A 和 B 使用參數 t 轉換的共同資訊， MI 為最大值時也就表示這是我們要轉換的空間參數的最佳值，也就是T_α。

3.4 MI特性

我們現在可以瞭解 MI 與 Entropy 之間的關係，簡單來說 Entropy 可以用來測量影像包含的資訊量，也就是影像的灰階值分布情形，而 MI 則是測量兩個影像間彼此包含多少資訊量。MI 會有以下特性[6]；

1. Non-negativity $I(A,B) \geq 0$

2. Symmetry $I(A,B) = I(B,A)$

3. Independence $I(A,B) = 0 \leftrightarrow P_{AB(a,b)} = P_{A(a)} \cdot P_{B(b)}$

4. Self $I(A,A) = H(A)$

5. Bounded

$$I(A,B) \leq \min(H(A),H(B)) \leq (H(A),H(B))/2 \leq \max(H(A),H(B)) \leq H(A) + H(B)$$

上述的特性我們都可以了解到，共同資訊量是不會為負的，A 和 B 的 MI 跟 B 和 A 的 MI 是一樣的，若是當 MI 為 0 的時候，也就是代表彼此的隨機變數是獨立無相關聯，自己跟自己的 MI 也就是自己的資訊量。

第四章 Cuda 實作流程

4.1 實作簡介

這個章節是這篇論文中最重要的部分，我們利用 CUDA 平行的算出 histogram 值，進而用 CUDA 平行的算出 entropy 跟 joint entropy，並且從 GPU 的記憶體傳回 CPU 的記憶體，求出最大的 MI 值。

4.2 計算entropy的概念

在這個部分將介紹 CUDA 實作的概念，我們將先求出 histogram，並且在同一個 kernel 中算出 entropy，利用 CPU 的 sequential algorithm 計算 histogram 是很容易的，我們只要循序的讀取 volume data 中每個 pixel 的像素值，然後讀取到的每個像素值所對應的存取位置就加 1，便可得出 histogram。

用 CPU 實作的演算法如下:

```
for(int i = 0; i < totaldatanumber; i++)
    temp=data[i];
    result[temp]= result[temp]+1;
```

利用 CUDA 實作 histogram，由於是利用 GPU 平行運算的架構，首先 Concurrent Read 是 CUDA 記憶體讀取的架構，所以平行的讀取資料是可以的 (temp=data[i];)，但是在寫入 result 的時候就會產生 Write collisions (result[temp]= result[temp]+1;)，例如兩個 threads

同時對 result[temp]進行加 1，result[temp]只會加 1 一次而不是兩次，這樣與我們想要的結果不同，這就是產生了 Write collisions。

為了要解決 Write collisions 的問題，由於 CUDA 的 shared memory 並沒有 hardware 支援的 atomic operaion，所以我們必須用程式邏輯的方式去解決 shared memory 的 Write collisions。由於 CUDA 的 device 在執行的時候，會以 block 為單位，把每個的 block 分配給各別的 multiprocessor 進行目前 CUDA 的 warp 大小都是 32，也就是 32 個 thread 會被群組成一個 warp 來一起執行，並且享有共同的 shared memory range，所以我們以一個 warp 為基礎去解決 shared memory 的 write collisions，整個流程如下:

1.給每一個不同的 thread 都有一個獨立的編號，用 threadtag 這個變數中最高位的 5 個 bits 去儲存。
2.讀取 histogram 中已經累積的數目，然後存到 count 這個變數的低位的 27 個 bits。
3. count 的上面五個 bits 則存入 threadtag，然後 thread 對 count 進行加 1。
4.將 count 存回我們的 histogram 中。
5.判斷是否該 thread 進行加 1 的 count 最後存回 histogram 中，如果是則跳出迴圈，如果是另外的 thread 存回 histogram 覆蓋掉的情形下，則繼續迴圈，直到是該 thread 的 count 最後存入為止。

演算法的要點如下:

```
    unsigned int threadTag = threadIdx.x << 27;//  每一個 thread 有自己的編號

addData256(result, temp , threadTag);

    unsigned int count;
    do{
            count = result [temp] & 0x07FFFFFFU;
            count = threadTag | (count + 1);
            result [temp] = count;//存取
    }while(result [temp] != count);
```

其中 temp 是讀取到的 intensity，result [temp]是要存入的位置，threadTag 則是把一個 thread 的 thread id 存入到高位的五個 bits，如果有兩個以上的 threads 要存入同一個記憶體位置(result [temp])，由於每一個 thread 的 thread id 不同，count 高位的 5 個 bits 就會不同，存入 result [temp]之後高位的 5 個 bits 就會不同。接下來利用判斷式(result [data] != count)，如果是目前的 threads 存取的 result [temp]，這個 thread 就可以跳出迴圈，如果不是則在迴圈內繼續跑，這樣就可以確保每一個讀取到同一個 intensity 的 thread 都會對 result[temp]的位置加一次，shared memory 的 write collisions 就不會影響到最後的結果，這個方法如果像素值越平均，threads 就不會都集中在同一個迴圈內，write collisions 就比較能在平行處理中解決，所以執行速度就越快，如果像素值集中在同一個數值，程式就會卡在同一個迴圈，就會從 parallel 的演算法變成 linear 的情況，這樣執行時間會大幅上升。由於 cuda 的 shared memory 的空間只有 16KB，存每一個像素值的數目要 4 個 bytes，如果是 1D histogram 通常可以存在 shared memory 內。而 2D 的 joint

histogram 就要動用到 global memory 來實作，由於 shared memory 的存入跟讀取的時間較快，我們還是會將 2D 的 joint histogram 像素值比較集中的部分由 shared memory 存入跟讀取，其他的部分再由 global memory 來存入。

如果要解決 global memory 的 write collisions 問題，在 cuda 中通常 8600GT 以上的顯卡會支援 atomic 函式，也就是硬體支援的 atomic operation 可以確保一次只能有一個 thread 對指定的記憶體位置存入，另外的 thread 必須等待正在寫入的 thread 對指定的記憶體存入後，才能對指定的記憶體進行存入，這樣可以避免 Write collisions，並且達到我們想要的存取結果。

在 cuda 的架構中，一個 kernel 的 grid 是沒有 syntronization 的機制的，但是同一個 block 裡面的 threads 是可以透過__syncthreads 函式進行同步的，用__syncthreads 可以確保在同一個 block 裡面的所有 threads 同樣執行到程式的那一行，同一個 block 中所有的 threads 都要做完那一行之前的指令後，才會往下執行。如果 block 中的其中任何一個 thread 沒有完成__syncthreads 之前的指令，那麼整個 block 中的所有 threads 都不會執行__syncthreads 後面的指令，這是 cuda 所提供的 block level syntronization，在 cuda 架構下同一個 block 裡面的 thread 有 syntronization 的機制，但是不同的 block 之間，是沒有 syntronization 的機制的，這也影響到我們實作上的設計。

由於果蠅腦的影像，有很廣大的背景，也就是像素值會集中在 0-5 之間，特別是集中在 0，這樣會讓 GPU 在計算 histogram 時執行速度大幅變慢，要解決上述的問題，我們特別針對 intensity 較集中的部份，我們會給每一個 thread 自己的 shared memory 的位置，專門存入各別 thread 讀取到較集中的像素值的數目，__syncthreads 後再合併到同一個 shared memory 的位置，由於 shared memory 有 16KB，我們可以在 shared memory 內解決這樣的問題，這樣可以平行的解決

出現次數最多的像素值，剩下的出現次數較少的像素值，就可以透過原本 atomic 的方式來解決。

，

最後我們要提及的是我們要運算 entropy 必須要用到 log 函式，cuda 在 kernel 裡是有內建的__log()函式的，不過由於是在 kernel 內使用，是利用 gpu 來計算，gpu 在處理浮點數上的精確度比 cpu 要來的稍微低，所以跟 cpu 相比較會有一點點的誤差，由於差距非常小，不會影響到影像對位的結果。

4.3 計算MI與影像對位的實作

在實作影像對位上，我們的方法是在 source 圖中個別圈出特徵點當作 source volume data 記作 window，然後針對 target 圖中選取與 source volume data 相同大小的 target volume data，也就是 target voxel。圖 4.1 是 MI 整體流程的示意圖，其中 MI 的計算是 MI(A,B)=H(A)-H(B)-H(A,B)，要計算 H(A)以及 H(B)要計算 1D histogram，要計算 H(A,B)則是要計算 joint histogram，所以我們在 CPU 及 GPU 的實作都是把 H(A)和 H(B)以及 H(A,B)分開計算，再求出最後的 MI 值。我們也針對醫學影像的特殊 histogram 的分佈，做出了特殊的處理以利於 GPU 的平行運算結構。

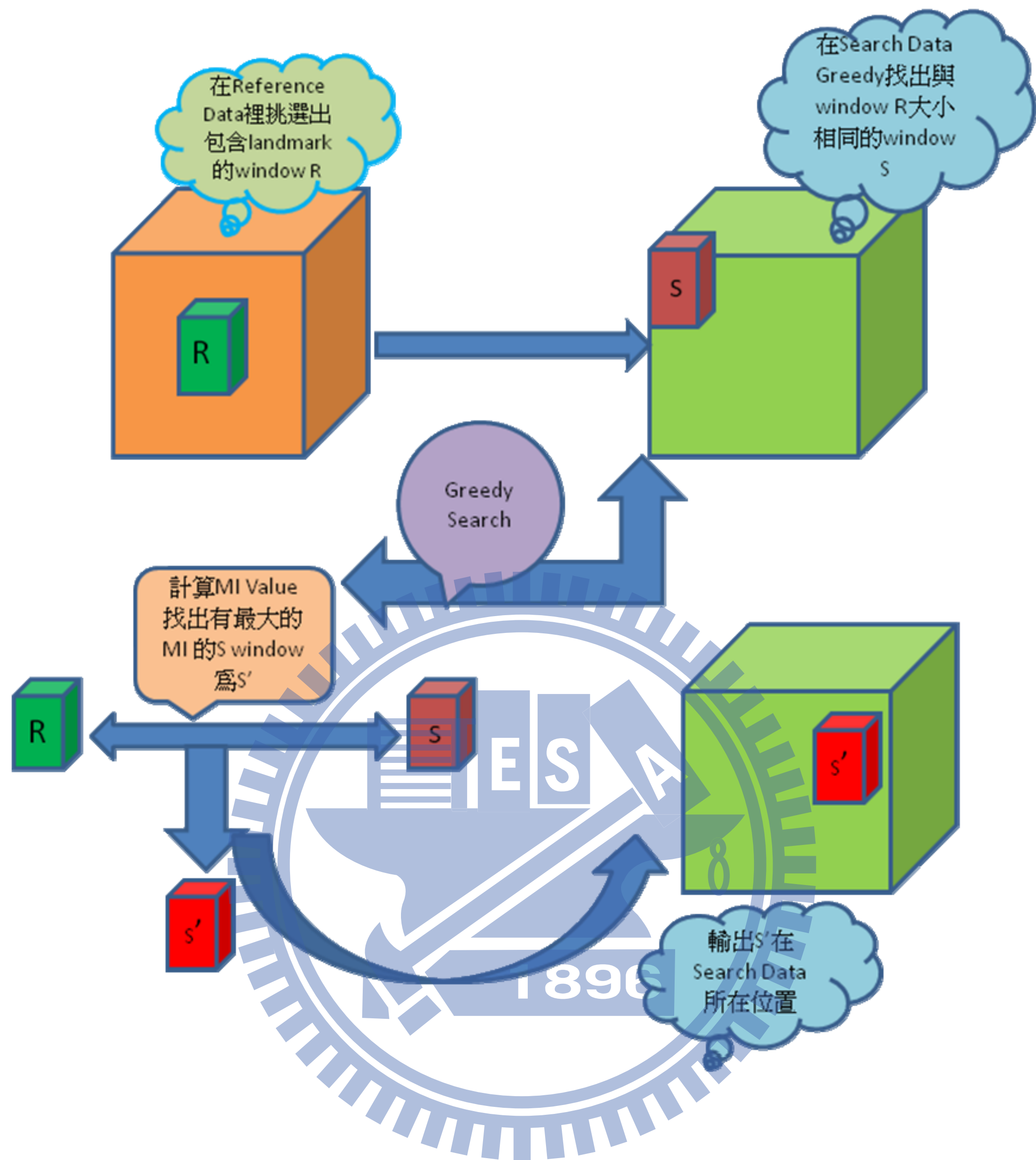


圖 4-1 MI 流程示意圖

我們用CUDA實作H(A)和H(B)包含了以下五個步驟:

1.在GPU上宣告A圖和B圖的記憶體空間，還有GPU上儲存entropy的記憶體空間

2.把A圖和B圖從CPU的記憶體上傳到GPU的記憶體空間

3.決定kernel的gridDim (要有多少個blocks)和blockDim(每個block有多少個thread)

4.執行kernel後entropy的結果會存在GPU的memory內

5.把entropy的結果從GPU的記憶體送回到CPU的記憶體

由於圖A跟圖B都是只要讀取且不會去存入或改變影像的數值，也就是唯讀的，這表示圖A和圖B是只需要讀取的device memory，我們可以把這塊記憶體當作texture來使用，可以大幅增進我們的讀取速度。

在CPU的實作我們是循序移動每一個Voxel，計算target volume data跟source volume data的MI，然後找尋最大MI的位置當作我們的對位作標，CPU的作法是sequential的去search data中找尋相同大小的window，用函式的參數去控制空間上的位置，循序的計算MI後再比較大小。在CUDA的實作中，我們是用一樣的想法，但是利用每一個block去作一個window，也就是volume data，由於一個kernel可以分成非常多個block，我們也就可以平行的處理好幾塊volume data的MI，也就是平行一次計算好幾塊的MI，然後比較彼此間MI的大小，而block index可以控制空間上的位置。

由於桌上型電腦的GPU最少需要處理桌面的更新，如果顯示卡執行同一個kernel時間過久，可能會使得 Windows 螢幕的畫面沒辦法更新。我們的做法是變成一個 iterative algorithms，不會把所有的volume data都在同一個kernel內執行，而是利用iterative的方式一次計算一些volume data，這樣也不必反覆的在GPU和CPU之間傳輸影像資料。因為圖A和圖B是已經傳送到GPU裡面，可以一直給kernel使用，我們只需要把entropy的結果從GPU傳回CPU就好。

在kernel的部份，這是cuda最重要的地方，kernel也是實際在GPU上執行的程式，我們想法是一個grid平行的處理很多個target window，由一個block去處理一個targer window，我們分配給一個block 32 個thread，以利於我們利用intra-warp的方式去處理shared memory的write collisions，並且以一個block去

處理一個 window 我們就能用__syncthreads 去處理 block level 的 syntronization，我們才能確定所有 thread 已經存好了 histogram，然後在 kernel 內把 entropy 算出來，我們的 kernel 大概是這樣規劃的:

H(A)和 H(B)的部份

1. 初始化 shared memory 空間以及設定好每個 thread 的 threadTag。

__syncthreads

2. 框出該 block 要處理的 volume data window，並且平行的由 block 中的 thread 去讀取圖的 intensity，利用 shared memory 儲存 histogram 的結果，藉由 4.2 敘述的方式處理 shared memory 的 write collisions。至於 intensity 較集中的部份則是每一個 thread 都有自己的 shared memory 空間去存取。

__syncthreads(同一個 block 裡面的 threads 都要完成了 histogram 的計算)

3. 將每個 thread 中儲存 intensity 是較集中的部份的 shared memory 空間分別累加起來。

__syncthreads

4.利用計算出來的 histogram 去計算 Marginal Probability Distribution，進而利用 cuda 內建的_log 函式來求出 entropy value。

我們用CUDA實作H(A,B)的想法跟實作h(A)和H(B)有些類似，只是在處理joint histogram 需要的記憶體空間非常大，例如說一個256 bins的影像如果算1d histogram只要用到256x4個bytes(大約1KB)去存，但是joint histogram就要用到256x256x4個bytes(大約256KB)的記憶體空間，所以我們就要用到global memory來儲存。global memory是有硬體支援的atomic函式的，由於shared memory的存入跟讀取的時間較快，我們還是會將2D的joint histogram像素值比較集中的部分由shared memory存入跟讀取，其他的部分再由global memory來存入。我們計算H(A,B)分成兩個kernel去做，其中一個kernel把H(A,B)中像素值較集中的部份用shared memory去存，另外一個kernel存取其他部份用global memory，由於joint entropy的算法是H(A,B)= $-\sum_{a\in A}\sum_{b\in B} P_{AB(a,b)} \log P_{(a,b)}$，是分別計算各別intensity的機率後再取log然後加起來的，所以可以分成兩個kernel分別去運算，我們的kernel大概是這樣規劃的:

H(A,B)中 A 或者 B 其中一個像素值較集中的部份

1. 初始化 shared memory 空間以及設定好每個 thread 的 threadTag。

__syncthreads

2. 框出該 block 要處理的 volume data window，並且平行的由 block 中的 thread 去讀取 A 圖跟 B 圖中的資料，如果 gray level pair 較集中則利用 shared memory 儲存 histogram 的結果。藉由 4.2 敘述的方式處理 shared memory 的 write collisions，至於 gray level pair 最集中的部分(0,0)則是跟 4.2 提及的一樣，每一個 thread 都有自己的 shared memory 空間去存 gray level pair (0,0)的部份。

__syncthreads(同一個 block 裡面的 threads 都要完成了 histogram 的計算)

3. 合併 gray level pair(0,0)的部份的 shared memory 空間。

__syncthreads

4.利用計算出來的 histogram 去計算 Probability density function，進而利用 cuda 內建的_log 函式來求出 entropy value。

計算 H(A,B)中的其他部分

1. 框出該 block 要處理的 volume data window，並且平行的由 block 中的 thread 去讀取 A 圖跟 B 圖中的資料，然後可以得到 gray level pair，如果 gray level pair 的像素值都不是 0 則利用 global memory 儲存 histogram 的結果，藉由 CUDA 內建的 atomic 函式，可以計算出我們想要的 histogram。

。

__syncthreads(同一個 block 裡面的 threads 都要完成了 histogram 的計算)。

2.利用計算出來的 histogram 去計算 Probability density function，進而利用 cuda 內建的_log 函式來求出 entropy value。

4.4 細胞的Tomography Reconstruction

我們可以由同步輻射中心所提供的細胞多角度投影影像，來做影像對位，由於分析 sinogram 大概會呈現不規律的上下左右偏移震動，所以我們利用 MI 的方法來做 Alignment。

整個實驗的流程如下(本篇論文的實驗資料是 Time Series Image 為 1~171):

Step1:首先由 Time Series Image 中挑選出一張影像當做 Reference Image(本論文為第 71 張記作 Reference Image)。

Step2:框取適當的範圍來做 MI 計算，循序算出每一張跟下一張之間的彼此之間的 local 偏移量，*MI Value* 最大者的偏移量即為最佳的 Alignment，我們將其偏移量記錄在一個陣列，當作接下來要累積偏移量的數據。

Step3:再將每一張彼此之間的偏移量累加成為真正的 global 偏移量，I1~ I70 對齊 I70，同樣的 I71~ I141 也對齊 I70。

Step4:利用算出的偏移量進行影像對位的對齊。

假設是有二十張影像的影像，並且取第 10 張記作 Reference Image。

範例如 Table1:其中偏移量為該影像與鄰近影像經由 MI 計算出來的值，假設 Reference Image 為 I10:

影像編號	I1	I2	I3	I4	I5	I6	I7	I8	I9	I10
偏移量	X=-1 Y=0	X=4 Y=3	X=-1 Y=-2	X=-1 Y=-2	X=1 Y=-2	X=-3 Y=2	X=3 Y=2	X=1 Y=2	X=1 Y=1	X=0 Y=0

表 4-1 Table1 鄰近偏移量

影像編號	I1	I2	I3	I4	I5	I6	I7	I8	I9	I10
偏移量	X=4 Y=4	X=5 Y=4	X=1 Y=1	X=2 Y=3	X=3 Y=5	X=2 Y=7	X=5 Y=5	X=2 Y=3	X=1 Y=1	X=0 Y=0

表 4-2 Table2 的偏移量為累積偏移量相對於 Reference Image 要做的偏移量

由 Table2 我們可以知道 I1 相對於 Reference Image I10 要 Translation X=4 Y=4 即可完成 I1 Alignment 的動作，依此類推，完成 Time series Image 的 Alignment 。

第五章 實驗結果

5.1 實驗環境

CPU：Intel Core(TM)2 Quad CPU Q6600 @ 2.40GHz

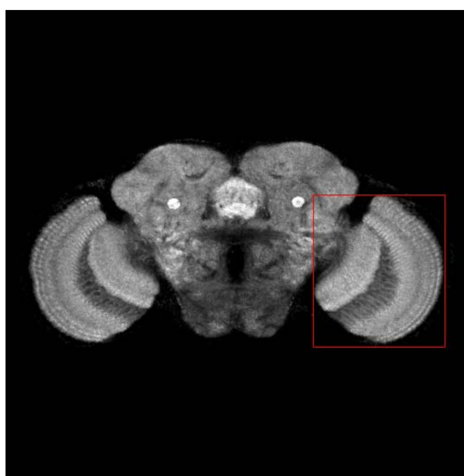
RAM：2GB

GPGPU：Nvidia GeForce 9800GTs

作業系統：Microsoft Windows XP Professional Release mode

5.2 部份腦對位

我們挑選果蠅的其中一個特徵點進行對位，比較 GPU 和 CPU 的效能，GPU 和 CPU 的對位結果完全一樣，MI 的總和因為 GPU 的不同略有誤差但是並不影響到結果。由於 0 這個像素值特別集中，所以我們特別用 shared memory 存入。我們用 texture memory 去讀圖也大幅增進了我們的整體效能。我們還比較了各種不同的顯示卡的執行效能。



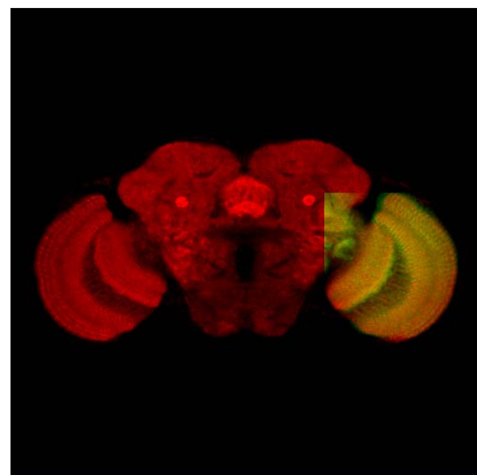


圖 5-1 (a) 特徵點框架　(b) 特徵點對位結果

果蠅特徵點處理時間

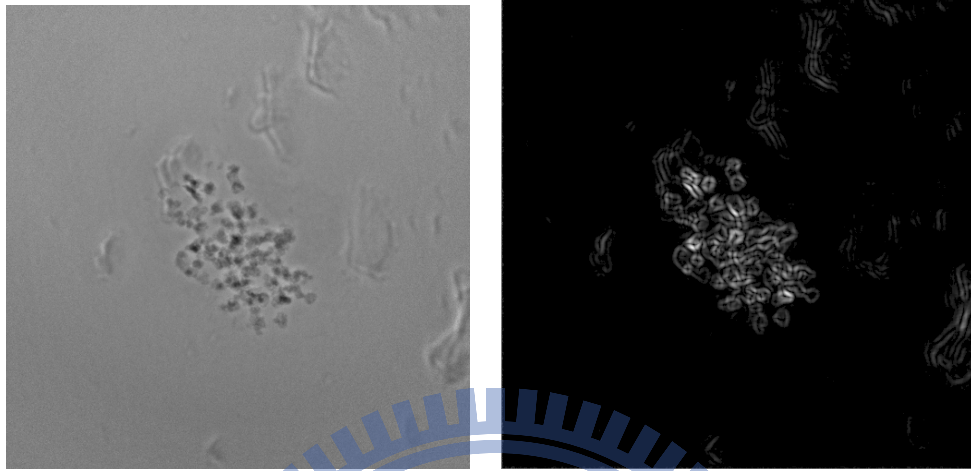
GPU 9800GT	CPU
平均跑一整張 z 軸約 34.579 秒	平均跑一整張 z 軸約 490.23 秒
GPU GTX 260	CPU
平均跑一整張 z 軸約 16.2 秒	平均跑一整張 z 軸約 490.23 秒
GPU 9600GT	CPU
平均跑一整張 z 軸約 37.395 秒	平均跑一整張 z 軸約 490.23 秒
GPU 8600GT	CPU
平均跑一整張 z 軸約 98.976 秒	平均跑一整張 z 軸約 490.23 秒

5.3 細胞

我們由同步輻射中心獲取得的細胞之多角度投影的影像來做對位，由於分析 sinogram 除了有一些雜訊之外，大概會呈現不規律的上下左右偏移震動。我們先將原始影像做 mean filter 去除雜訊，然後在利用 sobel filter 對影像做 edge detection，得到處理後的影像，將像素值二十以下的背景部分設定成 0，其他部份是邊界輪廓就保留下來，把有 0 的 joint histogram 存放在 shared memory 內，我們再利用 MI 的方法來做 Alignment，一共有四個不同的細胞影像。

細胞一

原始的影像:　　　　　　　　　　　　　　sobel filter 後的影像:



對位的結果:

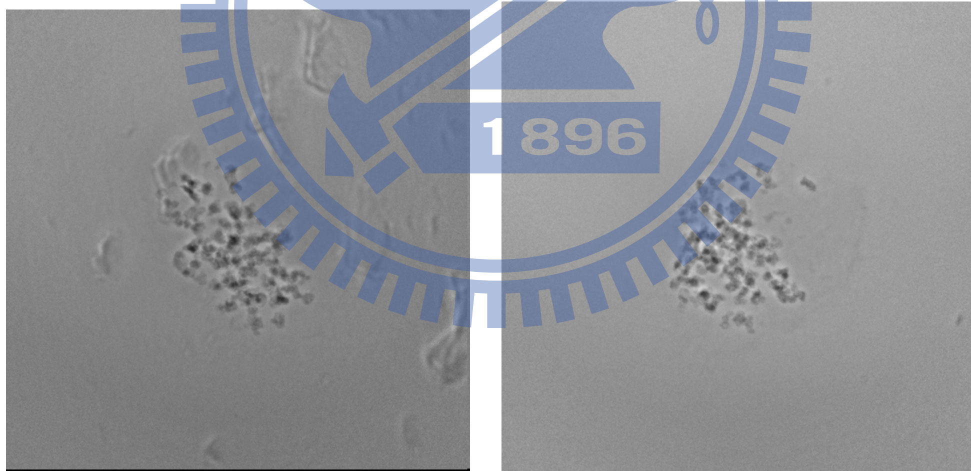


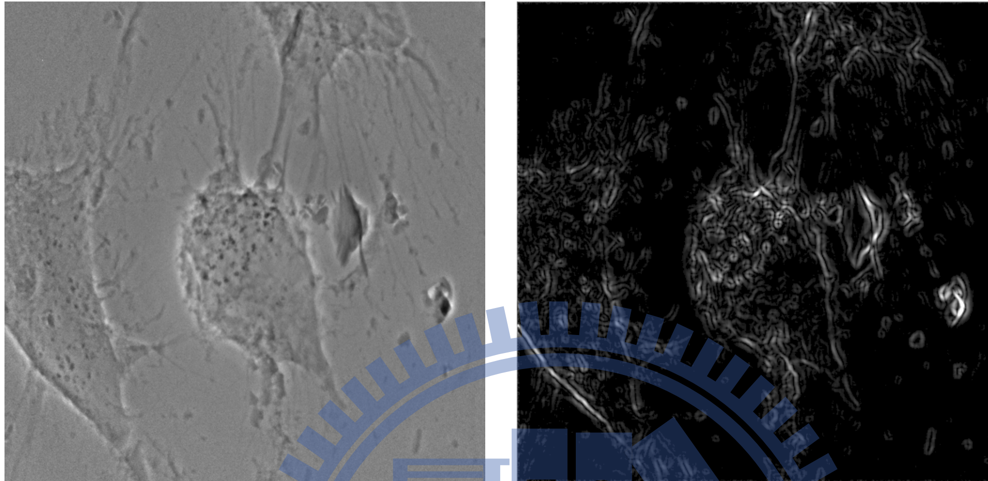
圖 5-2 (a) 細胞一第 10 張　　　　　　　　(b) 第 71 張

細胞一處理時間

GPU 9800GT	CPU
平均跑一個 slice 約 13 秒	平均跑一個 slice 約 209 秒
GPU GTX 260	CPU
平均跑一個 slice 約 8.5 秒	平均跑一個 slice 約 209 秒
GPU 9600GT	CPU
平均跑一個 slice 約 15 秒	平均跑一個 slice 約 209 秒
GPU 8600GT	CPU
平均跑一個 slice 約 35 秒	平均跑一個 slice 約 209 秒

細胞二

原始的影像:　　　　　　　　　　　　sobel filter 後的影像:



對位的結果:

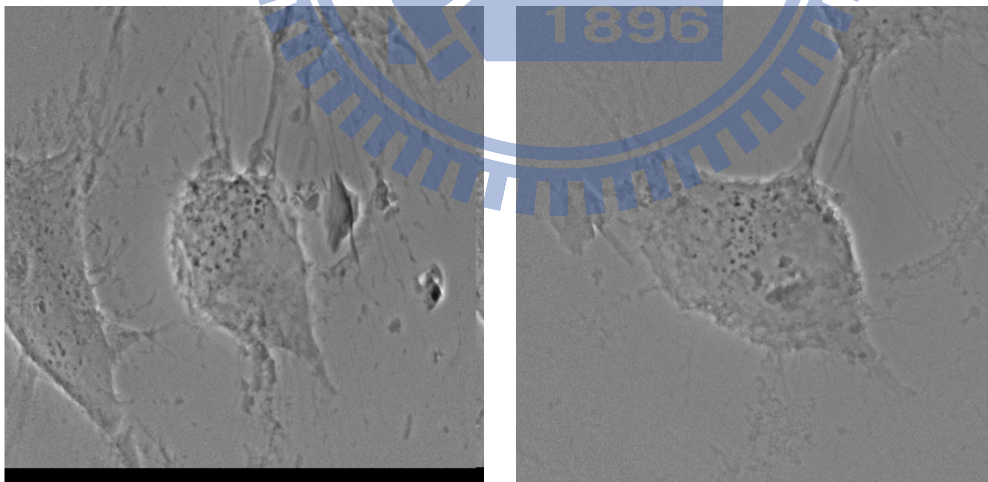


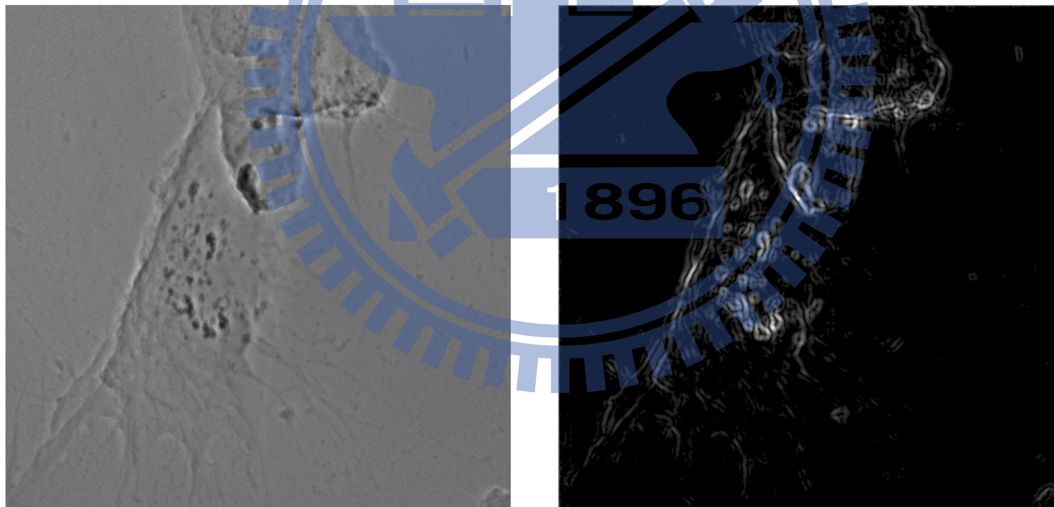
圖 5-3 (a) 細胞二第 10 張　　　　　　　　(b) 第 71 張

細胞二處理時間

GPU 9800GT	CPU
平均跑一個 slice 約 22 秒	平均跑一個 slice 約 209 秒

細胞三

原始的影像:　　　　　　　　　　　sobel filter 後的影像:



對位的結果:

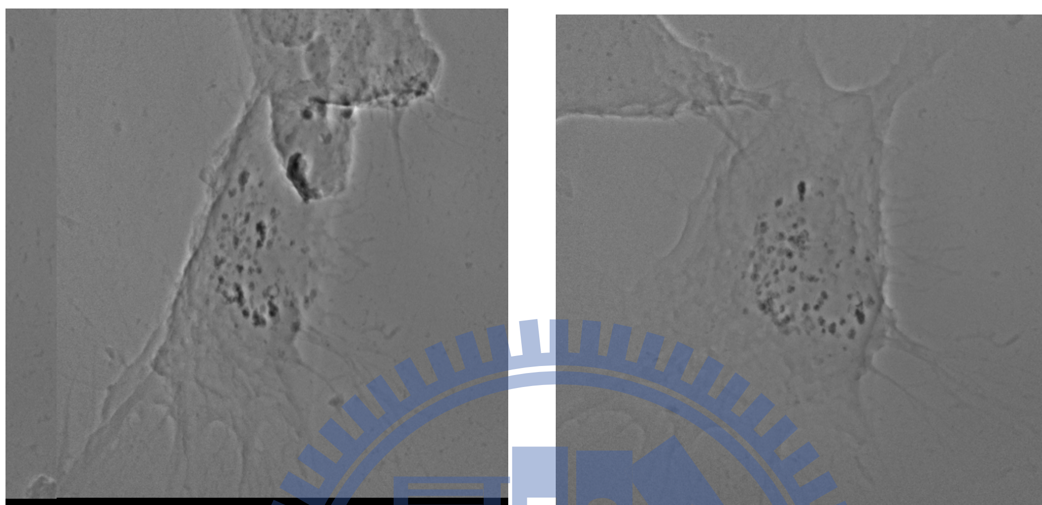


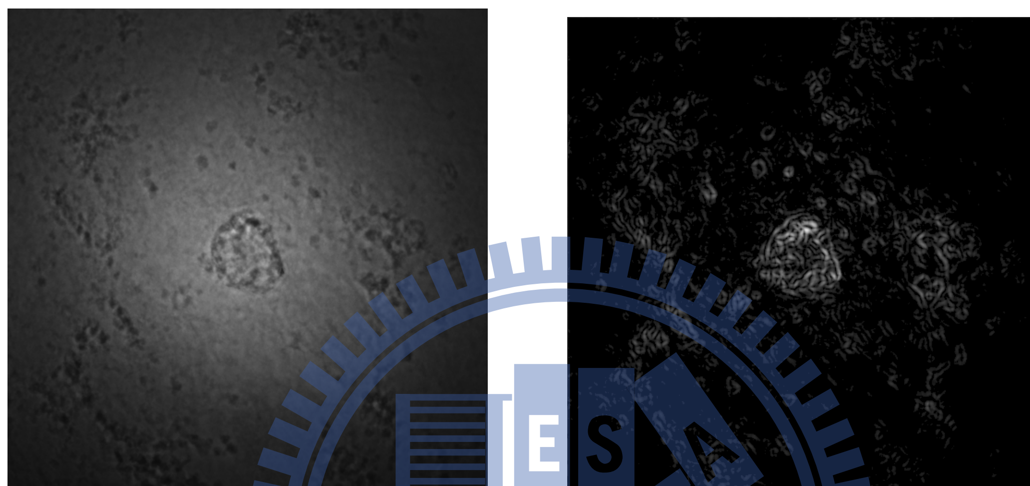
圖 5-4 (a) 細胞三第 10 張 (b) 第 71 張

細胞三處理時間

GPU 9800GT	CPU
平均跑一個 slice 約 15 秒	平均跑一個 slice 約 208 秒

細胞四

原始的影像:　　　　　　sobel filter 後的影像:



對位的結果:

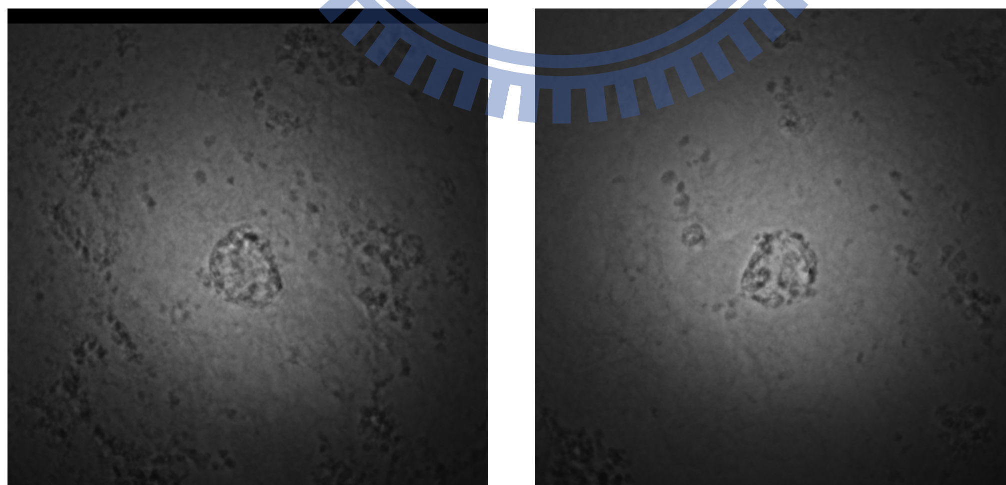


圖 5-5 (a) 細胞四第 10 張　　　　　(b) 第 71 張

細胞四處理時間

GPU 9800GT	CPU
平均跑一個 slice 約 15 秒	平均跑一個 slice 約 209 秒

原則上邊緣特徵愈少的影像，sobel filter 後像素值是 0 的部分會比較多，joint histogram 在 shared memory 存入的部分也會比較多，在 global memory 存入的部份就會比較少，所以 GPU 的執行效能會比較好。我們可以知道執行愈多次 global memory 的 atomic operation 的存入，GPU 的執行效能就會下降，我們的結果也就顯示了 shared memory 是 CUDA 加速的重要核心，shared memory 比 global memory 執行速度快很多。

第六章　未來展望

我們可以知道利用 MI 做影像對位的方法需要大量的時間。所以，如何運用特徵點，減少在座標系統上搜尋的範圍，是我們未來可以改善的目標。

計算 MI 所需的記憶體太大，是我們主要的困難，shared memory 是 GPU 的加快速度的核心，所以我們儘量將大部分的資料在 shared memory 內處理，以提升我們的執行效能，記憶體也是未來 GPU 改進的主要目標。

對於 joint histogram 找尋一個抽樣估計的方式也是一個改進的方向，雖然可能未必對於每個影像都會完全正確，但是對於某些影像或許可以有很好的結果，而且可以大幅減少執行時間，這也是我們未來可以改進的方向。

參考文獻

[1] L. G. Brown,” A Survey of Image Registration Techniques” ACM Computing Surveys ,vol 24, No. 4 , pp. 1–66, December1992

[2] Compute Unified Device Architecture (CUDA) Programming Guide. http://developer.nvidia.com/object/cuda.html: NVIDIA, 2007.

[3] M. Holden, D. L. G. Hill, E. R. E. Denton, J. M. Jarosz, T. C. S. Cox, T. Rohlfing, J. Goodey, and D. J. Hawkes, “Voxel similarity measures for 3-D serial MR brain image registration,” *IEEE Trans. Med. Imag.*, vol. 19, pp. 95–101, Feb. 2000.

[4] J. P. W. Pluim, J. B. A. Maintz, and M. A. Viergever,” Mutual-Information-Based Registration of Medical Images A Survey” *IEEE Trans. Med. Imag*, vol. 22, NO. 8, AUGUST 2003

[5] C. E. Shannon, “A mathematical theory of communication,” *Bell Syst. Tech. J.*, vol. 27, p. 379–423/623–656, 1948

[6] F. Maes, A. Collignon, D. Vandermeulen, G. Marchal, and P. Suetens, “Multimodality image registration by maximization of mutual information,” *IEEE Trans. Med. Imag.*, vol. 16, pp. 186–202, Apr. 1997