國立政治大學

應用數學研究所

碩士論文

PyCUDA在熱傳導方程的應用

The application of heat equation with PyCUDA

研 究 生：施政丞
指導教授：曾正男博士

中華民國104年7月

PyCUDA在熱傳導方程的應用

The application of heat equation with PyCUDA

研 究 生：施政丞　　Student: Zheng-Cheng Shih
指導教授：曾正男博士　　Advisor: Dr. Jengnan Tzeng

A Thesis
Submitted to Department of Mathematical Sciences

中華民國104年7月

PyCUDA在熱傳導方程的應用

學生：施政丞　　指導教授：曾正男博士

國立政治大學應用數學研究所

摘　　要

在本篇論文中，我們將呈現解擴散方程的交替分組法(Alternating group method)，對於傳統前向差分法(Explicit Forward Method(、後向差分法(Implicit Backward Method)、克蘭克-尼科爾森法(Crank-Nicolson method)來說，交替分組法的精確度比較好，並且具有平行化的特性。當資料量放大時，傳統方法將會需要較長的計算時間，因此交替分組法在平行計算時間上可以明顯的縮短計算時間。本論文將透過PyCUDA平行化套件將此方法實現在GPU計算上，藉此取得在計算時間上的優勢。雖然GPU的單位計算精度較CPU的單位計算精度差，然而在最後的數值計算誤差比較上，交替分組法的CPU版本與GPU版本之間的誤差幾乎相同。若此問題擴展到二維或三維，其計算量更是龐大，因此交替分組法的GPU平行化經驗在數值計算上是必要的。

ABSTRACT

In this paper, we will present an alternating group method for solving diffusion equation. The alternating group method is more precise than the Explicit forward Euler method, the Implicit backward Euler method and Crank-Nicolson method. Moreover, the alternating group method can be easily implement to the parallel version. When the computational system become huge, the serial computing methods take more time than the parallel computing methods. Hence, the parallel alternating group method will take the adventage in computational time. We will demonstrate the GPU version of alternating group method by the PyCUDA packge in this thesis. Although the precision of the GPU hardware is worse than CPU, the numerical results between GPU and CPU have almost no difference. Because the computational cost of 2D or 3D problem is much higher than the 1D problem, the experience of GPU version of alternating group method is very important in this field.

誌　　謝

結束了，在政大應數系待了八年，一直都想趕快離開學校進入職場，但真的到了這個時刻，又充滿了不捨。這一路走來面對資格考、實變函數論的打擊，讓我幾度想放棄，真的很感謝曾正男老師不但沒有放棄不成材的我，而是給我鼓勵、希望，儘管有責罵，但這也讓我有更多機會學習獨力解決問題的能力，讓我的能力在業界能夠被接受。而家人對於我延畢不但沒有責罵，反而是信任我，讓我可以放手一搏。在這邊要向所有對我有期許的你們說聲：抱歉，讓你們久等了，八年抗戰，勝利！

目　　錄

論文口試委員審定書 ii

授權書 iii

中文摘要 iv

英文摘要 v

誌謝 vi

目錄 vii

表目錄 ix

圖目錄 x

第一章、簡介 1

1.1 工具的選擇-Python 1

1.2 平行化的需求 2

1.3 GPU的基本介紹 4

第二章、Python之平行計算 7

2.1 Python基本運算之套件與工具 7

2.2 PyCUDA平行計算之套件與相關介紹 9

第三章、熱方程的多種解法與其平行化 20

3.1 Explicit Forward Method , Implicit Backward Method and Crank-Nicholson Method及其穩定性分析 20

3.2 交替分組法Alternating Group Method及其穩定性分析 25

3.3 交替分組法的CPU演算法介紹 30

3.4 交替分組法的GPU演算法介紹 34

第四章、實驗結果 39

4.1 AGM, CNM, IBM的誤差比較 39

第五章、結論 45

表　目　錄

3.1 error compare between IMP and CN 24

4.1 speedup compare by PyCUDA 43

4.2 speedup compare by PyCUDA 44

圖　目　錄

1.1 編譯語言排行榜(http://blog.codeeval.com/codeevalblog/2015) 3

1.2 CPU和GPU的比較[1] 3

1.3 The excute process of CUDA 4

1.4 The architecture of threads 5

1.5 The memory architecture of GPU 6

2.1 vector index with threadIdx 13

2.2 3X3 size of 2D Block 16

2.3 convert 3X3 thread block to 1X9 thread block 16

2.4 index of matrix after loading in memory 17

2.5 index of matrix after loading in memory 18

3.1 The stability of Explicit Forward Method 22

3.2 The stability of Implicit Backward Method 23

3.3 The error of IMP by increasing k 24

3.4 The architecture of AGM-case1 30

3.5 The architecture of AGM-case2 31

3.6 index of matrix after loading in memory 34

3.7 The architecture of GM group 35

3.8 copy the vector of GM 35

3.9 The architecture of GL group 36

3.10 The architecture of GR group 36

3.11 copy the vector 37

3.12 The process of parallel AGM 38

4.1 error compare 40

4.2 error compare 40

4.3 error compare 41

4.4 error compare 42

4.5 execute time compare 42

4.6 execute time compare 43

第 一 章　簡介

近年來在很多領域都能看到平行化的字眼，平行處理的重點不外乎就是為了加速處理的效率，舉凡：影像及視訊處理、計算生化學、流體力學模擬、電腦斷層、地震分析、光線追蹤，當然本篇論文所探討的數值計算也是一個重要的應用。而平行運算發展至今，最為人知的便是由Nvidia在2008年推出的一種整合技術，CUDA（Compute Unified Device Architecture，統一計算架構），並以C語言為開發環境，讓使用者可以控制GPU做平行運算。演變至今在許多語言中也能使用CUDA，如Fortran,Matlab,Java,Python等，本篇論文將以Python為計算工具，向讀者介紹如何在Python中使用PyCUDA套件來控制GPU的執行緒，達到平行運算的目的，文中會提供一些入門範例讓讀者可以由淺入深的了解平行化的操作方法與實際效用。

1.1　工具的選擇-Python

Python對於數學系出身的我來說，與C語言相比，其語法相對的簡單、明瞭，而且使用上更為直覺，接下來介紹Python的幾個優點，也是本篇論文選擇使用Python作為計算工具的原因。 [2]

簡單、直覺 由於Python是直譯式語言，所以在操作時不用宣告變數就能使用，也不需要先透過編譯才能得到結果，甚至在GUI介面底下就能直接當成計算機來使用，在操作上有點類似Matlab這種套裝軟體，但重點是Python是完全免付費。

强大的擴充功能與社群 Python本身已有內建的函式庫，若要做一些陣列運算、數值線性代數的運算、最佳化問題等，皆有完整的套件可以使用，甚至是本篇論文要使用的PyCUDA 套件也是可以直接使用的。在網路上有許多社群可以使用，互相學習，其中最為知名的社群是PyCon APAC，這是由Python社群自發性為了亞太地區Python 愛好者所舉辦每年一次的研討會。

物件導向 Python是一款完全物件導向的語言。函式、模組、類別、數字和字串都是物件，並且完全支援繼承、重載、衍生與多繼承，有益於增强原始碼的重複使用性。因此當程式愈大，物件導向的特性也讓Python的開發成本下降。

跨平台 因為Python直譯的特性，不需要編譯就可以執行，因此在不同的作業系統上，只要有直譯器都可以執行Python，甚至在Linux、Mac OS這類的系統中就有內建，近期推出的Ipython notebook更是方便，在任何平台都能在網頁介面下直接使用，而且套件幾乎一應俱全。

廣泛的使用 舉凡現在許多知名的服務，例如Dropbox，Google，YouTube，Wikipedia 引擎 MoinMoin、强大的應用程式伺服器 Zope，以及最常用的 mailing list 軟體 Mailman 也是用 Python 所開發出來的。另外NASA甚至使用Python來計算衛星軌道！

容易擴充和嵌入 Python本身是很好擴充的，如果有非常大量的計算量並且需要速度夠快，這時就可以考慮將負載量大的部分用C語言來寫，本篇論文所使用的PyCUDA就是一種擴充的應用。

Python的衆多優點不僅讓使用者更方便外也迅速地將自己推進A級的語言排行榜裡，在2015年已成為最受歡迎的語言。如下頁圖1.1

1.2 平行化的需求

要解釋平行化的最基本意義，從中國俗諺就曾經告訴過我們：三個臭皮匠，勝過一個諸葛亮。 這就是平行化的精神。傳統的演算法中，在for迴圈中控制index做一步步的運算，但假若演算法中並不需要前一個index的資訊，為什麼還需要等待它算完才能算下一個呢？直接藉由GPU具有多執行緒的特性，將所有index分配給每一條執行緒，讓它們同時計算，如此一來便能節省許多時間，但其所需計算時間並不是成線性關係，由於資料在Host與Device兩端互相傳遞時會有一些延遲的時間，所以要真正達到最佳的計算效率必須將資料傳遞的時間考慮進去。藉由下頁圖1.2[1]可以清楚了解GPU的計算精神。

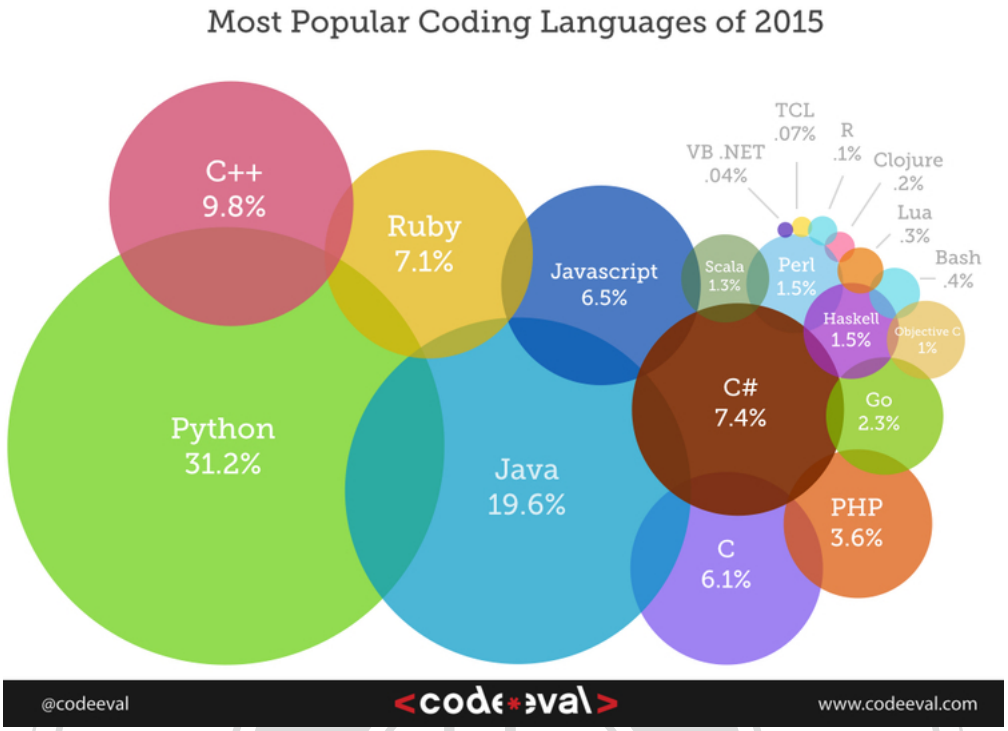
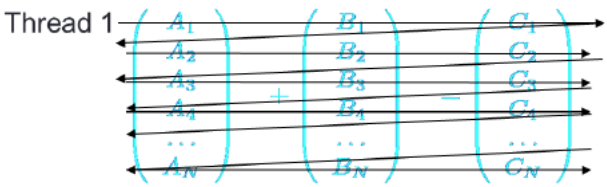
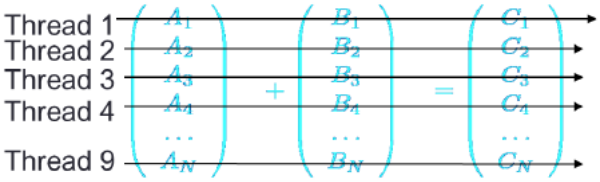


圖 1.1: 編譯語言排行榜(http://blog.codeeval.com/codeevalblog/2015)



(a) CPU version



(b) GPU version

圖 1.2: CPU和GPU的比較[1]

而本篇論文所討論的Alternating group method便具有index獨立的特性，在處理每一個time step時，格點間彼此是獨立的，因此用GPU來做平行運算是再適合

不過了。

1.3 GPU的基本介紹

CUDA運算環境 CUDA (Compute Unified Device Architecture) 是NVIDIA提出，可在NVIDIA圖形處理器進行平行運算的計算環境。程式設計者可以利用CUDA的C語言擴充 (extension) 直接用C語言寫程式，設計資料分配 (data distribution) 及程式流程將運算工作分配到上千個執行緒(threads)及圖形處理器中數以百計的計算核心 (cores)[7]。

在CUDA程式中，有兩個不同的運算環境：Host及Device。Host就是原本CPU的計算環境，可以讀寫檔案、配置記憶體、使用外部函式庫、呼叫和傳遞參數給GPU的副程式。Device是指GPU，它有獨立的記憶體和計算核心，計算用的資料需要從Host傳送到Device上的記憶體，才能在Device中處理。在Device上執行的副程式，稱為Kernel，通常有上百到上千個執行緒(thread)執行同一個Kernel。如圖1.3：

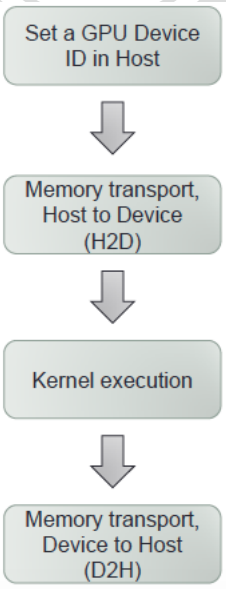


圖 1.3: The excute process of CUDA

每一個執行緒會屬於一個執行緒區塊 (thread block)，每一個block裡的thread總數有上限，不能超過1024個thread，可以指定成一維到三維的

排列方式。所有的block又會以一維或二維的方式排列在格子(grid)裡，每個thread及block依不同的排列順序會有不同的編號 (thread/block ID)，藉由執行緒編號，資料可以分配到不同執行緒上進行處理。如下圖：

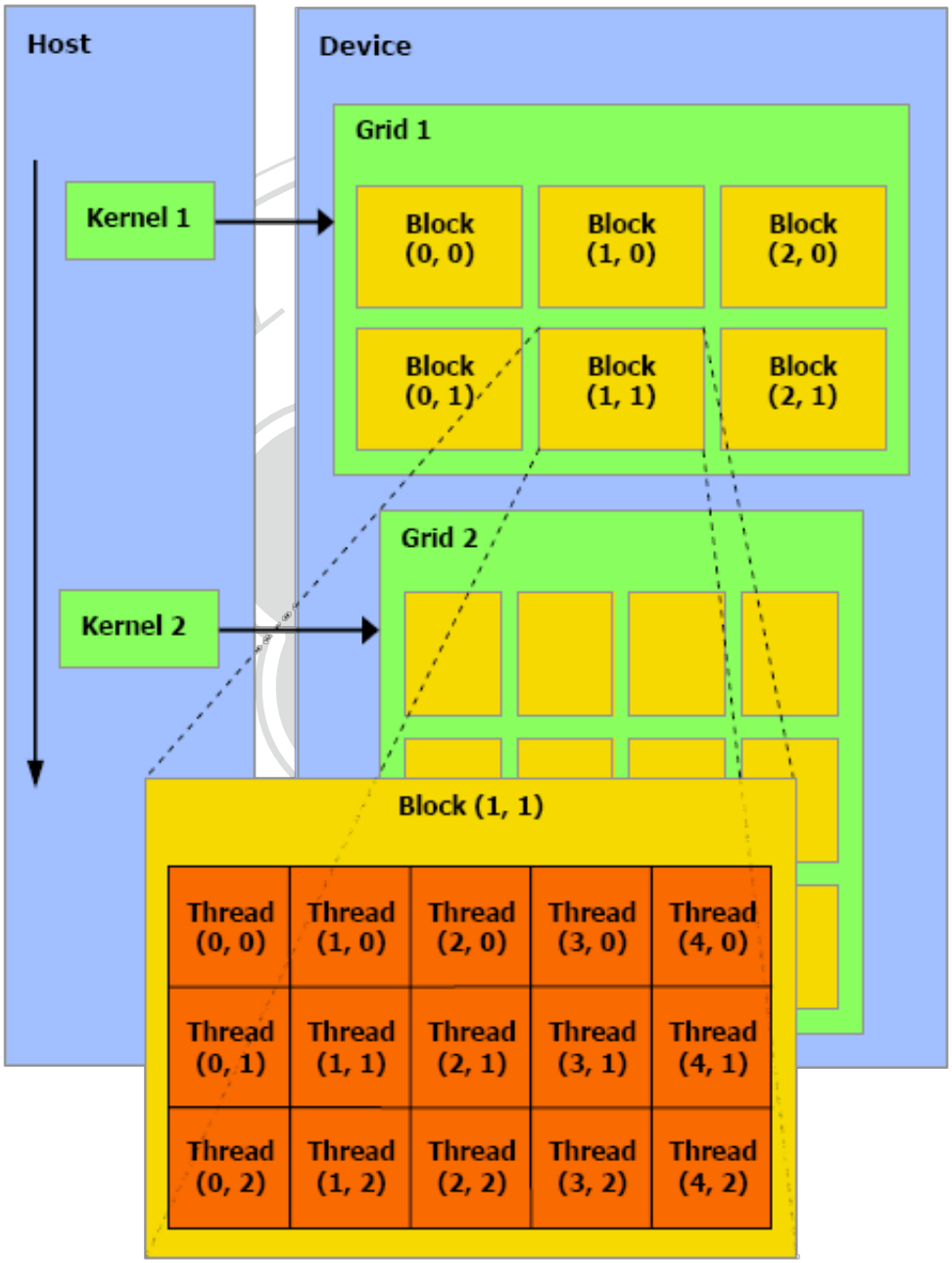


圖 1.4: The architecture of threads

圖形處理器硬體環境　　GPU的硬體是由許多Streaming Multiprocessors (SMs)及Global memory組成。每一個SM裡還有數個Scalar Processors (SPs)、Shared

memory、指令提取分派 (instruction fetch/dispatch)、雙倍精度浮點數 (double precision unit) 等控制器。程式中的一個Block會分配到一個SM上面執行，Block中的Thread會分配到這個SM的SP上執行，因此同一個Block中的所有Thread都可以看到共同的Share memory區段，也可以進行同步指令 (synchronize)。如下圖：

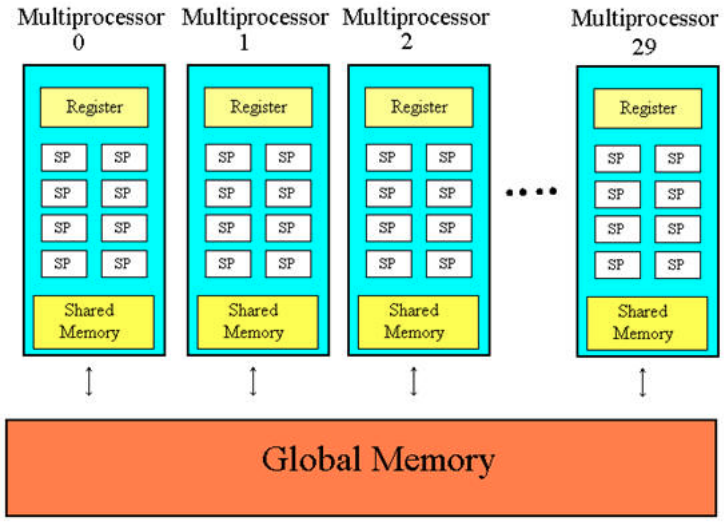


圖 1.5: The memory architecture of GPU

GPU的記憶體類型有很多種；Global memory是由全部的Block共用的，在Host的環境中也可以存取得到；每個SM上的Share memory是Block內部共用，還有Register讓每個Thread存放資料。存取的速度是Tegister > Share memory > Global memory。

本論文的架構為第一章做基本介紹、第二章為介紹PyCUDA平行計算的套件使用，並透過簡單的範例去定義kernel function，以及如何控制、分配GPU的主要計算單元-執行緒、第三章為介紹Alternating group method for solving diffusion equation，會詳細的推導其演算法流程以及對其穩定性的分析，第四章為實驗結果，討論Alternating group method在GPU的執行效率。第五章為本論文的結論。

第 二 章　Python之平行計算

2.1　Python基本運算之套件與工具

在網路上有許多Python入門的電子書[12]、以及書籍[5]在這裡提供給讀者做參考，由於篇幅的關係所以詳細的使用方法以及介紹就不在這裡多作說明了。這一章節介紹一些Python內常用的套件，有了這些套件後大部分的問題幾乎都可以處理了；而且使用這些套件來進行科學計算，還能獲得世界各地的開發者提供的套件資源。以下就是上述所提到的套件 [2]：

Numpy 是一種Python語言的延伸，其中包含了定義數值陣列和矩陣類型與一些在上面的基本操作。這裡會介紹到兩個例子，主要的目的是要表現出不用Numpy這個套件比起用了Numpy這個套件在處理一些簡單的基本操作中哪個速度會比較快，例子2.1-1會介紹到Python中不用Numpy這個套件的例子、例子2.1-2則會介紹到Python中用了Numpy這個套件的例子。

例子2.1-1：不用Numpy裡陣列的方式使得數值相加。

```
# add.py
# 以下的程式碼檔名為add.py
01| import time
  # 引入time套件
02| start = timeit.default_timer()
  # 起始時間。把此刻的執行時間記錄下來儲存到start中。
03| a = range(1000)
  # a是一個串列，其中第1個位置是0、第2個位置是1、以此類推到第1000個位置是999，因此串列的大小是1000。
04| b = range(1000)
  # b與上述的a一樣
05| c = []
  # c是一個空的串列，用來儲存a值與b值相加的結果。
  # 若將兩串列相加，是將b串列掛在a串列之後，並非是各別的元素相加。
```

所以要透過for迴圈把各別位置的值相加後並放入c串列。

```
06| for i in range(len(a)):
  # for迴圈跑的次數是a這個串列的大小，而i的值會根據迴圈的次數依序從0跑到999
07|     c.append(a[i] + b[i])
      # 每一次的迴圈中會把a[i]與b[i]的值作加總並放入c這個串列中
08| end = timeit.default_timer()
  # 結束時間。把此刻的執行時間記錄下來儲存到end。
09| print end-start
#將結束時間減去起始時間，得到其計算時間。
-----------------------------------------------------------------
# 以上為add.py的程式碼
```

執行上述程式碼，得到計算時間為0.00041389465332。

例子2.1-2：Numpy陣列相加。

```
# add_numpy.py
# 把以下為add_numpy.py的程式碼
01| import timeit
  # 引入time套件
02| import numpy as np
  # 把numpy更名為np
03| start = timeit.default_timer()
  # 起始時間。把此刻的執行時間記錄下來儲存到start。
04| a = np.arange(1000)
  # a是一個np中的陣列，其中第1個位置是0、第2個位置是1、以此類推到第1000個位置是999，因此陣列的大小是1000
05| b = np.arange(10000000)
  # b與上述的a一樣
06| c = a + b
  # 由於np中的陣列是可以直接相加的，所以c為陣列a與陣列b的加總
07| end = timeit.default_timer()
  #結束時間。把此刻的執行時間記錄下來儲存到end。
```

```
08| print end-start
#將結束時間減去起始時間，得到其計算時間。
-----------------------------------------------------------------
# 以上為add1.py的程式碼
```

執行上述程式碼，得到計算時間為0.000334978103638。

以上便是簡單的透過Python裡面numpy套件所得到的程式優化，非常的直覺明瞭且方便。底下將介紹本篇論文最重要的工具PyCUDA。

2.2 PyCUDA平行計算之套件與相關介紹

在介紹了基本的科學計算套件之後，這一節會介紹到PyCUDA套件，以及詳細介紹如何將CPU版本的code轉成GPU版本的code，讓讀者更清楚如何做平行化[4][10]。

剛開始使用PyCUDA，需要匯入PyCUDA套件以及做初始化的動作：

```
01| import pycuda.driver as drv
02| import pycuda.autoinit
03| from pycuda.compiler import SourceModule
  # 以上三項是執行PyCUDA缺一不可的工具
04| import numpy as np
05| a = np.random.randn(4,4)
  # 在CPU上創造一個4x4 矩陣，其資料為亂數資料。
06| a = a.astype(np.float32)
  # 因為多數的Nvidia 硬體只支援到單精度的運算，所以這裏需要利用np.float32將其資料形態改為單精度的浮點數。
```

由於GPU 是一個計算晶片，在計算過程中需要將資料從CPU傳到GPU 上做計算，再將算完的結果傳回CPU，所以資料的傳遞是使用PyCUDA的重點。

(Method 1)

```
07| a_gpu = cuda.meme_alloc(a.nbytes)
```

我們需要在GPU上開啓一個空間存放要轉換上去的資料，我們利用下列指令來宣告這個空間。

```
08| cuda.memcpy_htod(a_gpu,a)
```

最後就是將CPU內的資料a 轉換到GPU上面的a_gpu。CPU這端稱為Host，GPU 那端稱為Device，所以傳上去是 htod 。相反地，如果要從GPU拿東西下來就是dtoh。

接著就可以執行核心程式，以1D的執行緒為例

```
09| mod = SourceModule("""
10| __global__ void double(float *a)
11| {
12|     int i = threadIdx.x;
13|     a[i] * = 2;
14| }
15| """)
```

在PyCUDA中，主函數必須寫在SourceModule之中，這一部分是為了讓CUDA C和Python串接的重點，因此這一部分的語法都是C語言。

"threadIdx"為CUDA裡的執行緒索引值。

將a矩陣裏的每個元素個別平方。

```
16| func = mod.get_function('double')
```

利用mod將函數拿到外部讓Python可以讀取。

```
17| func(a_gpu,block=(4,4,1))
```

將block設定成4x4的大小，也就是有4x4條的threads，分別做對應元素的平方。

```
18| a_double = np.zeros_like(a)
```

造出一個與a的資料形態相同的矩陣，其值全為0。

```
19| cuda.memcpy_dtoh(a_double,a_gpu)
```

將計算完的結果從GPU傳回CPU上的a_double。

(以上的說明是能表現完整的資料傳輸方式以及簡單的在GPU上做計算。而底下就介紹一種簡便的傳輸方法及計算方法)

(Method 2)

記得在一開始import pycuda.driver as drv，此套件中的"In","Out","InOut"，可以直接將Python外部變數轉成gpuarray傳送到GPU上，並且將計算過後的結果傳回CPU上。使用方法如下：

```
20| func(drv.InOut(a),block=(4,4,1))
```

如此便能將CPU上的資料傳送到GPU上，經過計算後再傳回CPU上，是不是很簡單！

(Method 3)

除此之外，Pycuda還有一個更方便的套件：pycuda.gpuarray.GPUArray，這個型別具有物件導向的特性，透過這個套件，可以把上述範例縮減成更簡短的程式。

```
01| import pycuda.gpuarray as gpuarray
02| import pycuda.autoinit
03| import numpy as np
04| a_gpu = gpuarray.to_gpu(numpy.random.randn(4,4).astype(numpy.float32))
05| a_doubled = (2*a_gpu).get()
```

'gpuarray.to_gpu'可以直接把CPU裡面的資料轉換成GPU的資料，當一個變數a_gpu被宣告成 gpuarray 型別時，他的.get() 函數又可以把資料轉回CPU。

在GPU上做基本的平行計算就是如此簡單，藉由Method 1,2可以了解GPU的計算精神與邏輯，而Method 3就是屬於快速簡便的使用模式，建議讀者可以先熟悉Method 1,2，再使用Method 3做其他的計算。

接下來要透過一個簡單的範例來表現CPU版本與GPU版本的差異。此範例是要將兩個向量做內積。

(Method 1：inner_product_PyCUDA.py)

```
01| import pycuda.driver as drv
02| import pycuda.autoinit
03| from pycuda.compiler import SourceModule
04| import numpy as np
```

為了做陣列的運算，引進numpy套件，並命名為np。

```
05| import timeit
```
引入計算執行時間的套件。

```
06| a = np.random.randn(400).astype(np.float32)
```
a是隨機創造的一個1X400的陣列，因為多數的Nvidia 硬體只支援到單精度的運算，所以這裏需要利用numpy.float32將其資料形態改為單精度的浮點數。

```
07| b = np.random.randn(400).astype(np.float32)
```
與上述的a一樣。

```
08| dest = np.zeros_like(a)
```
將dest設為與a大小、形態相同的陣列，但其值全為0，用來儲存計算結果。

接下來是主程式的部分

```
09| mod = SourceModule("""
10| __global__ void inner_product(float *dest, float *a, float *b)
11| {
12|     int i = threadIdx.x;
13|     dest[i] = a[i]*b[i];
14| }
15| """)
```
__global__表示這個函數是在GPU上執行，inner_product是函數的名稱，dest為輸出的變數，a與b為輸入的變數。

i為執行緒的索引，此處使用1D的執行緒，其中"threadIdx" 為CUDA的函數，".x"為只取x方向的座標。

上述即為a,b兩向量間元素對元素相乘，並將結果儲存在dest變數中。

如圖2.1：

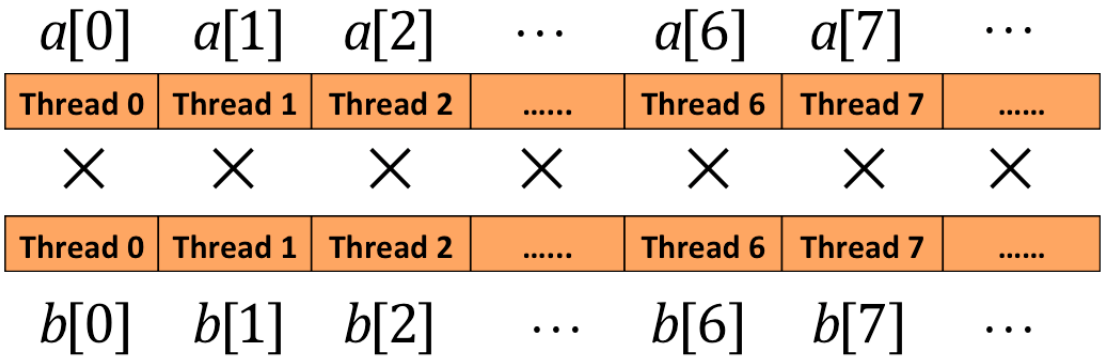


圖 2.1: vector index with threadIdx

```
16| inner_product = mod.get_function('inner_product')
  # 此處將C函數拿到外部給Python使用。
17| start = timeit.default_timer()
  # 起始時間。把此刻的執行時間記錄下來儲存到start中。
18| inner_product(drv.Out(dest),drv.In(a),drv.In(b),block=(400,1,1),grid=(1,1)
  # 這個範例使用1D的block，也就是一個1X400大小的thread，分別處理
個別元素的相乘。
19| np.sum(dest)
  # 要完成兩向量內積，最後將計算結果全部相加。
20| end = timeit.default_timer()
  # 結束時間。把此刻的時間記錄下來儲存到end中。
21| PyCUDA_time = end - start
  # 計算整個程式執行的時間。
-----------------------------------------------------
# 以上為inner_product.py的平行化版本

(Method 2：inner_product_Serial.py)

22| tmp = 0
  # 給定一個起始值，用來累加答案
23| ss = timeit.default_timer()
  # 起始時間。把此刻的執行時間記錄下來儲存到ss中。
```

```
22| for i in range(1024):
23|     tmp += a[i]*b[i]
24| ee = timeit.default_timer()
  # 結束時間。把此刻的執行時間記錄下來儲存到ee中。
25| Python_time = ee-ss
-------------------------------------------------
# 以上為inner_product.py的傳統版本
26| print "time of CPU",Python_time
27| print "time of GPU",PyCUDA_time
28| execfile('inner_product.py')
  # 執行inner_product.py
time of CPU = 0.00486898422241
time of GPU = 0.00254702568054
```

可以看到計算速度的提升，由於計算量不大，所以得到的speed up 有限，若是提升向量的大小，便可以發現計算速度明顯的提升。

下一個範例要展示兩個3X3方陣的相乘。

```
01| import pycuda.driver as drv
02| import pycuda.autoinit
03| from pycuda.compiler import SourceModule
# 以上為使用PyCUDA必要套件
04| import numpy as np
# 引進numpy做陣列運算
05| (n,m,p)=(3,3,3)
# 如果讀者想做其他大小的矩陣相乘，在此更改參數即可
06| a=np.random.rand(3,3)
07| a=a.astype(np.float32)
08| b=np.random.rand(3,3)
09| b=a.astype(np.float32)
 # a與b是隨機創造的一個3X3的矩陣，因為多數的Nvidia 硬體只支援到單
```

精度的運算，所以這裏需要利用numpy.float32將其資料形態改為單精度的

浮點數。

```
10| c=np.zeros_like(a)
```

將c設為與a大小、形態相同的矩陣，但其值全為0，用來儲存計算結果。

接下來是主程式的部分

```
10| mod = SourceModule("""
11| __global__ void matrixmulti(float *c, float *a, float *b, int n, int m,
      int p)
12| {
13|    int i =p*threadIdx.x + threadIdx.y;
14|    c[i] = 0;
15|    for(int k=0;k<m:k++)
16|    {
17|     c[i]+=a[m*threadIdx.x+k]*b[threadIdx.y+k*p];
18|    }
19| }
20| """)
```

在global底下宣告函數名稱:matrixmulti，c為輸出的矩陣，a與b為輸入的矩陣。

在這個範例中，我們只使用1X1的Grid，也就是在一個Grid裡面只有一個Block，而這個Block是由3X3的Thread所組成，而讀入的矩陣a,b以及輸出的矩陣c所在的位置皆為Global memory。我們用一張圖來說明2D thread block的指標如何用來控制矩陣的元素。如圖2.2：

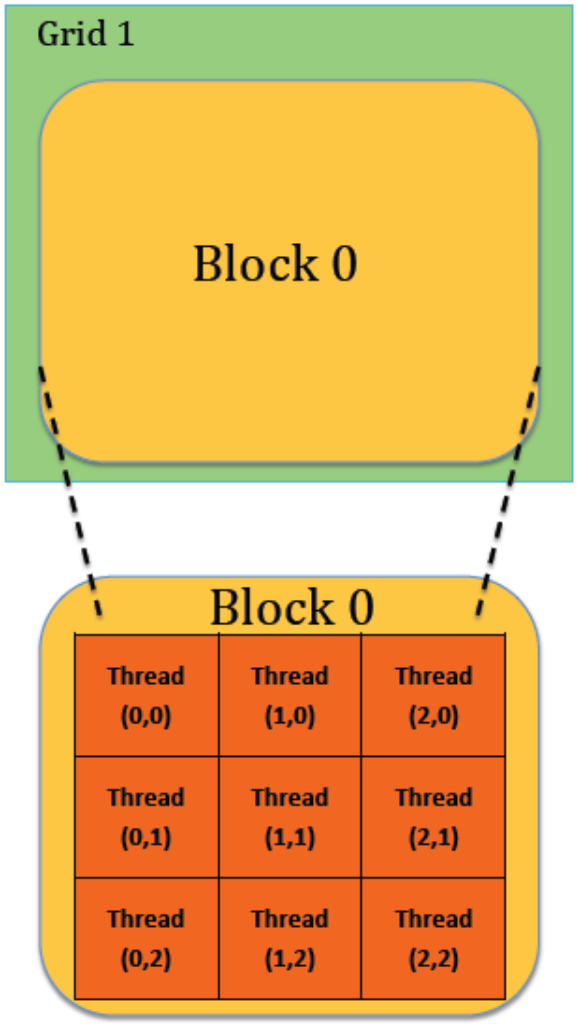


圖 2.2: 3X3 size of 2D Block

接著，利用int i =p*threadIdx.x + threadIdx.y，其中p=3，將3X3的矩陣做一個線性轉換，使其轉換成1D的向量，目標讓每一個index的Thread個別去讀取矩陣A的一條列向量以及矩陣B的一條行向量。如2.3圖：

圖 2.3: convert 3X3 thread block to 1X9 thread block

接下來的for loop是矩陣乘法的重點。 首先，在C語言中，利用” * ”指標的概念讀入欲相乘的矩陣a、b，以及儲存的矩陣c，讀入之後原本3X3的矩陣，其索引值將變成以下的形式：

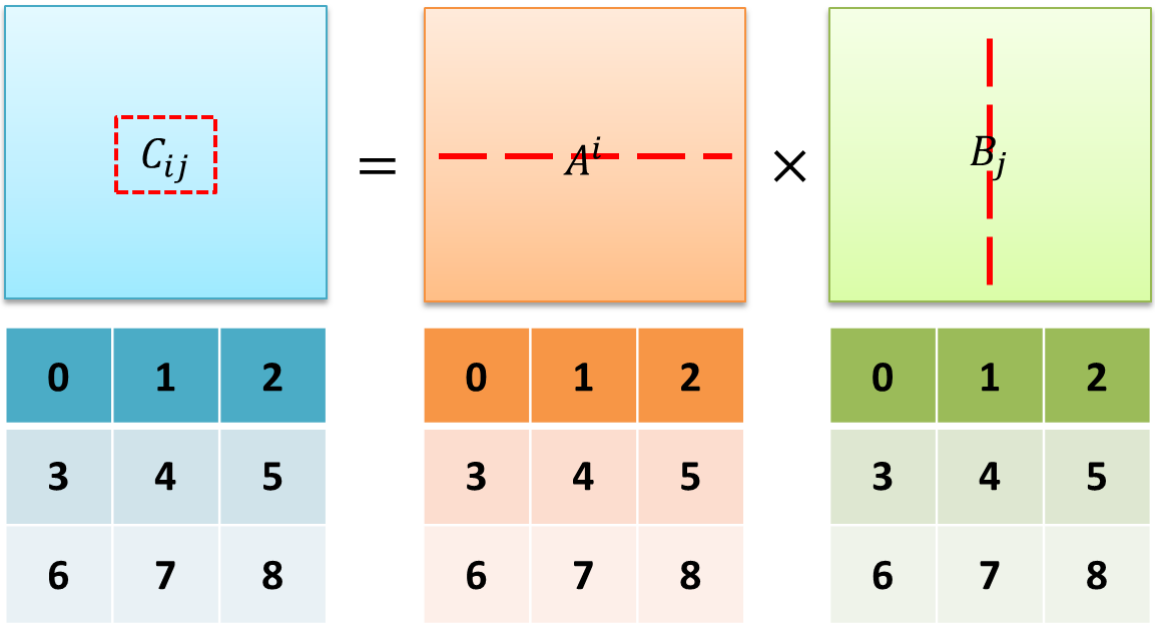


圖 2.4: index of matrix after loading in memory

接下來詳細說明for loop如何運作：

```
for(int k=0;k<m:k++)
{
 c[i]+=a[m*threadIdx.x+k]*b[threadIdx.y+k*p];
}
```

上述的$i = 0, \cdots, 8$即為thread index，例如：當$i = 4$時，對照原始3X3的thread block，如圖2.5，

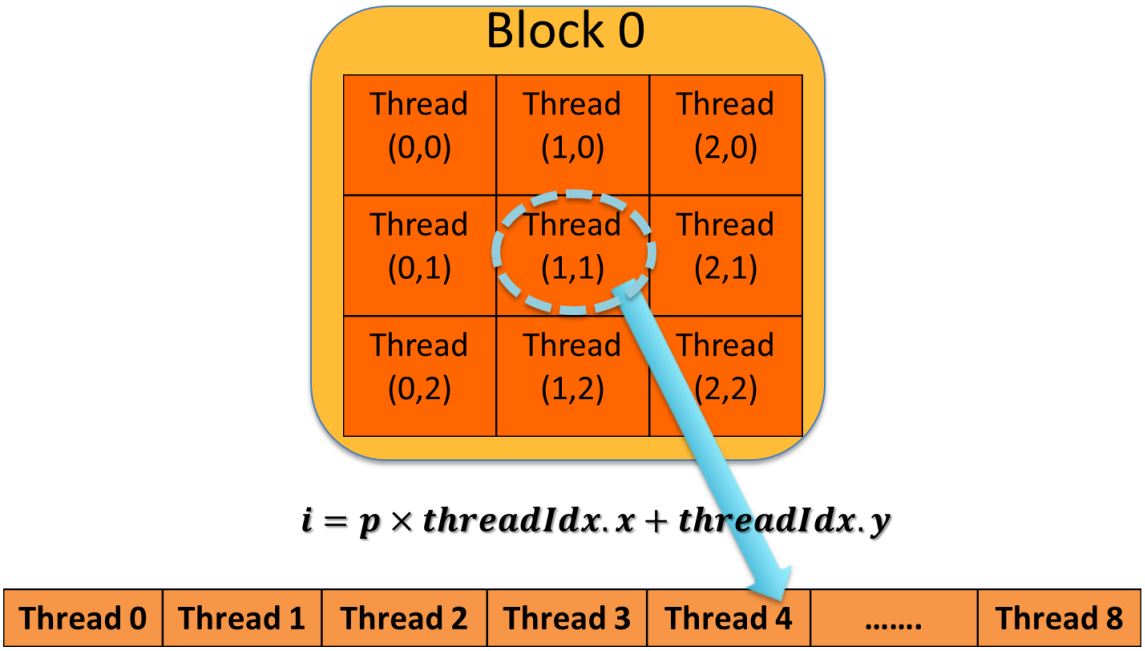


圖 2.5: index of matrix after loading in memory

得知其原本的thread index為$(1,1)$，因此$threadIdx.x = 1$，$threadIdx.y = 1$

根據矩陣乘法規則，我們知道$c[4] = a[3] \times b[1] + a[4] \times b[4] + a[5] \times b[7]$

因此，

當k=0，可以得到$a[3] \times b[1]$

當k=1，可以得到$a[4] \times b[4]$

當k=2，可以得到$a[5] \times b[7]$

將其餘index套用上述的方法，便能成功讓每一條Thread個別讀取矩陣A的一條列向量以及矩陣B的一條行向量，並將兩向量內積，完成矩陣乘法的平行化。

```
21| matrixmul=mod.get_function("matrixmul")
# 此處將C函數拿到外部給Python使用。
22| matrixmul(drv.Out(c),drv.In(a),drv.In(b),np.int32(n),np.int32(m),np.int32(p),
          block=(m,m,1),grid=(1,1))
# 將矩陣a,b讀入，以gpu array的形式存在Global memory，並將答案寫入同樣是gpu array型態的c矩陣。
# 此處使用3X3的thread block，每一條Thread個別負責一個內積的運算。到此就完成平行化的矩陣相乘，只要資料量夠大，在計算速度上就能得到明顯的提
```

升。

從以上範例我們得知，GPU計算的精神就在於如何運用index的分配來達到計算目的。

第 三 章　熱方程的多種解法與其平行化

這一章節我們會先介紹傳統方法，包括前向差分法(Explicit Forward Method) , 後向差分法(Implicit Backward Method) and 克蘭克-尼科爾森法(Crank-Nicholson Method)的原理及優劣，接著介紹交替分組法(Alternating group method)的原理，以及適合引入平行化的特性，並給出交替分組法的穩定性分析、誤差分析，最後給出傳統CPU版本與GPU版本的演算法。

3.1 Explicit Forward Method , Implicit Backward Method and Crank-Nicholson Method及其穩定性分析

Explicit Forward Method

考慮擴散方程

$$\begin{aligned} \frac{\partial u}{\partial t} &= \frac{\partial^2 u}{\partial x^2}, \quad (x,t) \in [0, x_f] \times [0, T] \\ u(x,0) &= f(x), \qquad x \in [0, x_f] \\ u(0,t) = g_1(t), u(1,t) &= g_2(t), \qquad t \in [0, T] \end{aligned} \tag{3.1}$$

我們使用finite difference method 把 $[0, x_f]$ 區間切割成 M 等分，把 $[0, T]$ 時間區間切割成 N 等分。因此，我們有 $\Delta x = \frac{x_f}{M} = h$ 以及 $\Delta t = \frac{T}{N} = k$。並且我們用central difference approximation 去取代 $u(x,t)$ 在空間上的二次偏微分，用forward difference approximation 取代對時間的偏微分。我們得到下列差分方程式[12]：

$$\frac{u_{i+1}^n - 2u_i^n + u_{i-1}^n}{h^2} = \frac{u_i^{n+1} - u_i^n}{k} \tag{3.2}$$

經過整理，可以得到下列差分方程式：

$$u_i^{n+1} = ru_{i-1}^n + (1-2r)u_i^n + ru_{i+1}^n \tag{3.3}$$

其中 $r = \frac{k}{h^2}$ ，$i = 1, 2, ..., M-1$，利用泰勒展開示可以得到其截尾誤差(truncation error)為$O(k) + O(h^2)$。

觀察此式，我們發現可以直接利用前一個時間點的值來判定下一個時間點的值，因此稱其為顯示(explicit)。 若我們希望這個疊代式穩定，(3.4)式會對應一個線性系統 $u^{n+1} = Au^n$。為了讓其收斂，
令w_n為滿足(3.5)式的真解，u_n為計算所得的數值解，計算兩者的誤差

$$\begin{aligned} e_n &= w_n - u_n \\ &= Aw_{n-1} - Au_{n-1} \\ &= A(w_{n-1} - u_{n-1}) \\ &= Ae_{n-1} \end{aligned}$$

為了要確保誤差e_n沒有被放大，我們必須要求A的譜半徑(spectral radius)$\rho(A) \leq 1$，但若真的去找特徵值(eigenvalue)絕對值的最大值是較為麻煩的，我們可以利用下列方法決定適合的 k和 h。
利用傅立葉基底將 u_i^n表示成$u_i^n = \lambda^n e^{j\frac{i\pi}{P}}$ ，其中 j 是 $\sqrt{-1}$ 的意思，P 是任意非零的整數，將其代入(3.4)式，我們得到

$$\lambda = 1 - 2r(1 - cos(\frac{\pi}{P}))$$

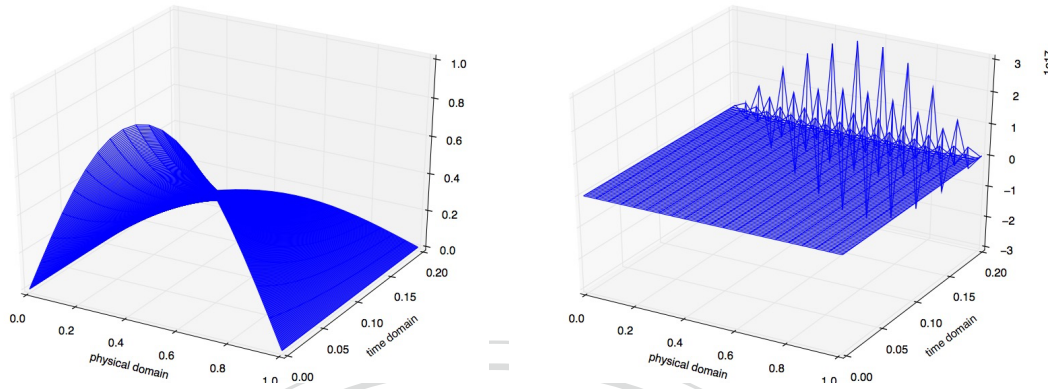
當我們希望 u_i^n不會隨著 k 增加而爆掉時，我們希望 $|\lambda| \leq 1$，可以得到

$$r = \frac{k}{h^2} \leq \frac{1}{2}$$

因此，以Explicit Forward Method求解熱方程，若$2k \leq h^2$，則此方法是穩定的。底下用一個範例來觀察r值對穩定性的影響。

$$\frac{\partial^2 u(x,t)}{\partial x^2} = \frac{\partial u(x,t)}{\partial t}, for \;\; 0 \leq x \leq 1, 0 \leq t \leq 0.1 \tag{3.4}$$
$$u(x,0) = sin(\pi x), u(0,t) = 0, u(1,t) = 1$$
$$u(x,t) = sin(\pi x)e^{-\pi^2 t}$$

下圖即為分析的結果：



(a) r=0.3 (b) r=0.8

圖 3.1: The stability of Explicit Forward Method

當$r > \frac{1}{2}$，可以看出數值解是爆掉的，因此我們需要其他的方法來避免這種情況發生。

Implicit Backward Method

為了避免誤差被放大的性質，我們可以使用Implicit Backward Method做比較。若把(3.2)式的右端改成backward difference，我們得到

$$\frac{u_{i+1}^n - 2u_i^n + u_{i-1}^n}{h^2} = \frac{u_i^n - u_i^{n-1}}{k} \tag{3.5}$$

經過整理後可得下列差分方程式：

$$-ru_{i-1}^{n+1} + (1+2r)u_i^{n+1} - ru_{i+1}^{n+1} = u_i^n \tag{3.6}$$

其中 $r = \frac{k}{h^2}$ ，$i = 1, 2, ..., M-1$，利用泰勒展開示可以得到其截尾誤差為$O(k) + O(h^2)$。

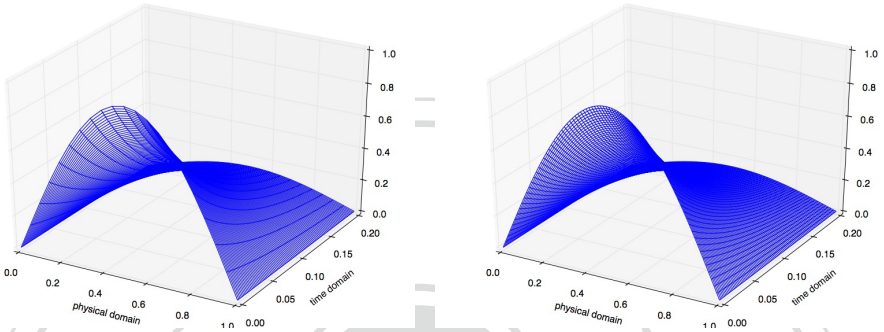
這時我們無法對每一個 i 位置的 u 進行單純地迭代得到答案，而是解一個三對角的矩陣。如底下形式：

$$\begin{bmatrix} 1+2r & -r & 0 & \cdots & 0 \\ -r & 1+2r & -r & \ddots & \vdots \\ 0 & -r & 1+2r & \ddots & 0 \\ \vdots & \ddots & \ddots & \ddots & -r \\ 0 & \cdots & 0 & -r & 1+2r \end{bmatrix}$$

雖然解矩陣是比較麻煩的方式，但若我們套用先前的傅立葉分析方法將$u_i^n = \lambda^n e^{j\frac{i\pi}{P}}$帶入(3.4)式，經過整理算式後我們發現$|\lambda|$ 自動小於 1，這表示

不用特別調整 h 和 k 的大小，我們用Implicit Backward Euler Method 可以得到比較穩定的解，且是無條件穩定。

同樣用(3.4)式的範例，底下的圖即為分析的結果：



(a) r=0.8　　(b) r=1.8

圖 3.2: The stability of Implicit Backward Method

可以看到，$r > \frac{1}{2}$時，其數值解還是收斂的。 既然Implicit Backward Euler Method屬於無條件收斂，那麼我們就可以討論因為時間和空間的離散化所造成的截尾誤差大小。由於時間離散化產生的誤差階數為$O(k)$，而空間離散化的誤差階數為$O(h^2)$，因此當空間步長h很小時，其所造成的截尾誤差相對於$O(k)$將顯得微不足道，甚至可以粗略的描述為$O(k) + O(h^2) \approx O(k)$，因此$O(k)$將會是誤差的主要來源[8]。

我們以固定的$h = 0.1$和逐步減少的k來呈現其關係，如圖3.3。

因此，k必須要愈小，才能避免產生不穩定性。相對的，k要愈小，時間格點要更細，要解更多次的三對角矩陣，計算時間也將被拉長，如何縮短計算時間便顯得重要。

Crank-Nicholson method

讓我們回顧一下implicit backward Euler method 的遞迴公式

$$\frac{u_{i+1}^n - 2u_i^n + u_{i-1}^n}{h^2} = \frac{u_i^n - u_i^{n-1}}{k}$$

可以發現左端點的差分公式是對 $u(x_i, t_n)$ 做展開，而右端的差分公式是對 $u(x_i, (t_n + t_{n-1}))$ 這一點展開，這裡存在時間上的不一致，就會增加更多的截尾誤差。Crank-Nicholson Method 就是為了改進時間上不一致而有的方

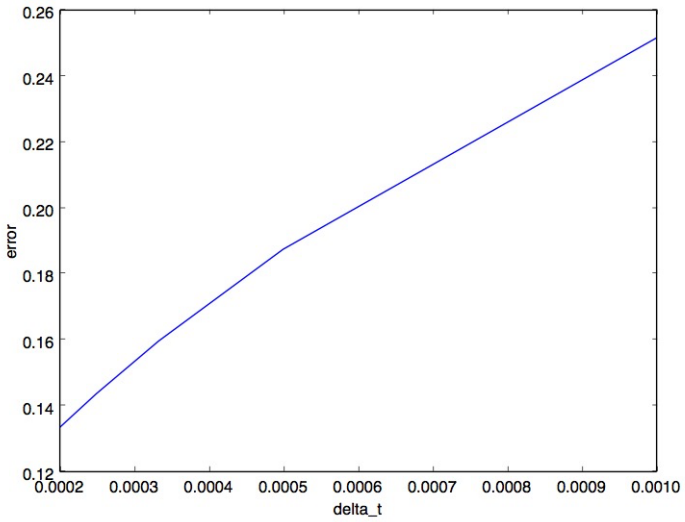


圖 3.3: The error of IMP by increasing k

法，他的差分方程式如下：

$$\frac{1}{2}\left(\frac{u_{i+1}^{n+1}-2u_i^{n+1}+u_{i-1}^{n+1}}{h^2}+\frac{u_{i+1}^{n}-2u_i^{n}+u_{i-1}^{n}}{h^2}\right)=\frac{u_i^{n+1}-u_i^{n}}{k}$$

經過這樣的調整，等號兩邊的位置與時間都保持一致，簡化後的差分式如下：

$$-ru_{i+1}^{n+1}+2(1+r)u_i^{n+1}-ru_{i-1}^{n+1}=ru_{i+1}^{n}+2(1-r)u_i^{n}+ru_{i-1}^{n}$$

其中 $r=\frac{k}{h^2}$，利用泰勒展開示可以得到其截尾誤差為$O(k^2)+O(h^2)$。
這時候同樣要去解三對角矩陣的形式，仿照同樣的分析穩定性的方法，我們得到對應的$|\lambda|$ 也是自動小於等於 1 。說明這個方法也是一個穩定的解法，並且預期它的誤差會比Implicit Backward Euler Method 來的好。
利用同樣(3.4)式的範例：

表 3.1: error compare between IMP and CN

h	k	error of IBM	error of CN
0.05	0.02	0.33982	0.00423
0.025	0.01	0.34518	0.00211
0.0125	0.005	0.34811	0.00105

可以看到，誤差明顯比Implicit Backward Euler Method來的好，而且，當h和k減半時，Crank-Nicolson Method的誤差減少為$\frac{1}{2}$。到目前為止，Crank-Nicholson Method的表現最好，但每經過一個time step皆需要解一個線性系統，並不具有平行化的特性，一旦資料量放大，其計算時間將會相當可觀，必須透過平行化才能有效的提高計算速度，因此我們將介紹具有平行化特性的交替分組法。

3.2 交替分組法Alternating Group Method及其穩定性分析

擴散方程的求解問題在工程計算中比較常見，且計算較為複雜，為了便於在GPU上實行平行運算以加快計算速度，本方法以第二類Saul'yev非對稱格式，建構出求解擴散方程的交替分組方法。其基本結構為四點組，並針對內點為偶數的情況，在節點兩端點處進行了處理，以提高精度。交替分組法在運算邏輯上具有很適合平行運算的部分，接下來就說明這個方法的原理[6]。

在3.1節介紹的Explicit forward method, Implicit backward method, Crank-Nicolson method。這些方法可以分成兩類，一類是顯式，不需要去解聯立方程組，而且具有平行運算的邏輯，適合在平行計算機上使用，但是這種類型是條件穩定的，對時間、格點有很大的限制；另一類是隱式，這種類型是無條件穩定，但需要解聯立方程組，此種計算邏輯不適合在平行計算機上使用。因此，建構一個適合在平行計算機上執行並且無條件穩定的方法就顯得重要，交替分組法剛好具備這樣的特性。

首先將區域$(0,1)\times(0,T)$進行分割，將空間格點記為Δx，其中$\Delta x=\frac{1}{M}=h,\ x_i=ih,\ i=0,1,...,M$，時間格點記為$\Delta t$，其中$\Delta t=\frac{T}{N}=k,\ t_n=jk,\ j=0,1,...,N$，並將節點$(x_i,t_n)$記為$(i,n)$，$u_i^n$代表解在節點$(x_i,t_n)$的數值解。

為了方便計算，定義：

$$\delta_x u_i^n=\frac{u_{i+1}^n-u_i^n}{h},\ \delta_{\bar{x}}u_i^n=\frac{u_i^n-u_{i-1}^n}{h},\ (\delta_x(\delta_{\bar{x}}u))_i^n=\frac{u_{i+1}^n-2u_i^n+u_{i-1}^n}{h^2},\ \delta_t u_i^n=\frac{u_i^{n+1}-u_i^n}{\Delta t}$$

為了建構交替分組法，我們需要用到第二類Saul'yev非對稱格式[9]去估計(3.1)式在節點$(x_i, t_{n+\frac{1}{2}})$的值：

$$\delta_t u_i^n = \frac{1}{2}((\delta_x(\delta_{\bar{x}}u))_i^n + \frac{\delta_x u_i^{n+1} - \delta_{\bar{x}} u_i^n}{h}) \tag{3.7}$$

$$\delta_t u_i^n = \frac{1}{2}((\delta_x(\delta_{\bar{x}}u))_i^n + \frac{\delta_x u_i^n - \delta_{\bar{x}} u_i^{n+1}}{h}) \tag{3.8}$$

$$\delta_t u_i^n = \frac{1}{2}((\delta_x(\delta_{\bar{x}}u))_i^{n+1} + \frac{\delta_x u_i^{n+1} - \delta_{\bar{x}} u_i^n}{h}) \tag{3.9}$$

$$\delta_t u_i^n = \frac{1}{2}((\delta_x(\delta_{\bar{x}}u))_i^{n+1} + \frac{\delta_x u_i^n - \delta_{\bar{x}} u_i^{n+1}}{h}) \tag{3.10}$$

將其用來逼近(3.1)式，可以得到四個對應的Saul'yev非對稱格式：

$$(1+\frac{r}{2})u_i^{n+1} - \frac{r}{2}u_{i+1}^{n+1} = ru_{i-1}^n + (1-\frac{3}{2}r)u_i^n + \frac{r}{2}u_{i+1}^n \tag{3.11}$$

$$-\frac{r}{2}u_{i-1}^{n+1} - (1+\frac{r}{2})u_i^{n+1} = \frac{r}{2}u_{i-1}^n + (1-\frac{3}{2}r)u_i^n + ru_{i+1}^n \tag{3.12}$$

$$-\frac{r}{2}u_{i-1}^{n+1} - (1+\frac{3r}{2})u_i^{n+1} - ru_{i+1}^{n+1} = \frac{r}{2}u_{i-1}^n + (1-r)u_i^n \tag{3.13}$$

$$-ru_{i-1}^{n+1} + (1+\frac{3}{2}r)u_i^{n+1} - \frac{r}{2}u_{i+1}^{n+1} = (1-\frac{r}{2})u_i^n + \frac{r}{2}u_{i+1}^n \tag{3.14}$$

其中 $r = \frac{k}{h^2}$

利用**Taylor expansion**容易證明其截尾誤差為$O(k + h^2 + \frac{k}{h})$[14]。在交替分組方法中，使用兩種分組模式：四點組、兩點組

四點組：

假設已知$u(x,t)$在第n層的值為u_i^n，為了求出第$n+1$層的值u_i^{n+1}，四點組含有四個內點，分別是(x_i, t_{n+1})、(x_{i+1}, t_{n+1})、(x_{i+2}, t_{n+1})、(x_{i+3}, t_{n+1})，將其應用在(3.11)-(3.14)式，可以得到差分格式為：

$$(1+\frac{r}{2})u_i^{n+1} - \frac{r}{2}u_{i+1}^{n+1} = ru_{i-1}^n + (1-\frac{3}{2}r)u_i^n + \frac{r}{2}u_{i+1}^n \tag{3.15}$$

$$-\frac{r}{2}u_i^{n+1} - (1+\frac{3r}{2})u_{i+1}^{n+1} - ru_{i+2}^{n+1} = \frac{r}{2}u_i^n + (1-r)u_{i+1}^n \tag{3.16}$$

$$-ru_{i+1}^{n+1} + (1+\frac{3}{2}r)u_{i+2}^{n+1} - \frac{r}{2}u_{i+3}^{n+1} = (1-\frac{r}{2})u_{i+2}^n + \frac{r}{2}u_{i+3}^n \tag{3.17}$$

$$-\frac{r}{2}u_{i+2}^{n+1} - (1+\frac{r}{2})u_{i+3}^{n+1} = \frac{r}{2}u_{i+2}^n + (1-\frac{3}{2}r)u_{i+3}^n + ru_{i+4}^n \tag{3.18}$$

兩點組：

為了避免影響精準度，在處理邊界的問題上，使用兩點組來逼近，分成左、右兩個邊界來處理

左邊界如下：

$$(1+\frac{3}{2}r)u_1^{n+1}-\frac{r}{2}u_2^{n+1}=(1-\frac{r}{2})u_1^n+\frac{r}{2}u_2^n+ru_0^{n+1} \tag{3.19}$$

$$-\frac{r}{2}u_1^{n+1}-(1+\frac{r}{2})u_2^{n+1}=\frac{r}{2}u_1^n+(1-\frac{3}{2}r)u_2^n+ru_3^n \tag{3.20}$$

右邊界如下：

$$(1+\frac{r}{2})u_{m-2}^{n+1}-\frac{r}{2}u_{m-1}^{n+1}=ru_{m-3}^n+(1-\frac{3}{2}r)u_{m-2}^n+\frac{r}{2}u_{m-1}^n \tag{3.21}$$

$$-\frac{r}{2}u_{m-2}^{n+1}-(1+\frac{3r}{2})u_{m-1}^{n+1}=\frac{r}{2}u_{m-2}^n+(1-\frac{r}{2})u_{m-1}^n+ru_m^{n+1} \tag{3.22}$$

由於邊界條件已知，因此以上兩種分組均可以用迭代式求出第$n+1$層的值。接下來考慮格點數，由於交替分組法是四點組與兩點組的組合，所以只考慮分點數m是奇數的情況，此時內點數$m-1$是偶數，依照上述的四點組劃分，則在邊界節點有可能剩餘0點或2點，詳細說明如下[13]：

case1: $m-1=4k+2$(k為整數)

在不失一般性的情況下，假設n為偶數，把u^{n+1}層上的內點劃分成$(k+1)$個獨立計算組。從左至右，第1組至第k組用四點組式，最右邊的第$(k+1)$組用兩點組式，在u^{n+2}層也劃分為$(k+1)$個獨立計算組，從左至右，最左邊的第1組用兩點組式，第2組至第$(k+1)$組用四點組式，兩層格式交替使用，得到差分格式的矩陣形式：

$$\begin{cases}(I+rG_1)U^{n+1}=(I-rG_2)U^n+b_1\\(I+rG_2)U^{n+2}=(I-rG_1)U^{n+1}+b_2\end{cases} \tag{3.23}$$

其中，

$$U^n=(u_1^n,u_2^n,...,u_{m-1}^n)^T,$$

$$b_1=(ru_0^n,0,...,0,ru_m^{n+1})^T,$$

$$b_2=(ru_0^{n+2},0,...,0,ru_m^{n+1})^T,$$

b_1和b_2是與邊界條件有關的$(m-1)$維向量，

$$G_1=\begin{bmatrix}A^1&&&&\\&A^2&&&\\&&\ddots&&\\&&&A^k&\\&&&&A'\end{bmatrix},$$

$$G_2 = \begin{bmatrix} B' & & & & \\ & A^1 & & & \\ & & A^2 & & \\ & & & \ddots & \\ & & & & A^k \end{bmatrix}$$

case2：$m-1=4k$

同樣假設n為偶數，如同$case1$，在兩個時間層上交替構造差分格式，在u^{n+1}層上劃分$(k+1)$個獨立計算組，最左邊的第1組用兩點組式，最右邊的第$(k+1)$組用兩點組式，從第2組至第k組用四點組式。在u^{n+2}層劃分k個獨立計算組，均使用四點組式，兩層交替使用得到差分格式的矩陣形式：

$$\begin{cases} (I+r\widetilde{G}_1)U^{n+1} = (I-r\widetilde{G}_2)U^n + b \\ (I+r\widetilde{G}_2)U^{n+2} = (I-r\widetilde{G}_1)U^{n+1} + b \end{cases} \tag{3.24}$$

其中，

$$U^n = (u_1^n, u_2^n, ..., u_{m-1}^n)^T,$$
$$b = (ru_0^{n+1}, 0, ..., 0, ru_m^{n+1})^T,$$

b是與邊界條件有關的$(m-1)$維向量，

$$\widetilde{G}_1 = \begin{bmatrix} B' & & & & \\ & A^1 & & & \\ & & \ddots & & \\ & & & A^{k-1} & \\ & & & & A' \end{bmatrix},$$

$$\widetilde{G_2} = \begin{bmatrix} B^1 & & & & \\ & B^2 & & & \\ & & \ddots & & \\ & & & B^{k-1} & \\ & & & & B^k \end{bmatrix}$$

以上各式中，

$$A_i = \begin{bmatrix} \frac{1}{2} & -\frac{1}{2} & & & \\ -\frac{1}{2} & \frac{3}{2} & -1 & & \\ & -1 & \frac{3}{2} & -\frac{1}{2} & \\ & & -\frac{1}{2} & \frac{1}{2} & \end{bmatrix}$$

$$A^{'}=\begin{bmatrix}\frac{1}{2} & -\frac{1}{2}\\ -\frac{1}{2} & \frac{3}{2}\end{bmatrix}, B^{'}=\begin{bmatrix}\frac{3}{2} & -\frac{1}{2}\\ -\frac{1}{2} & \frac{1}{2}\end{bmatrix}$$

從矩陣$G_1, G_2, \widetilde{G}_1, \widetilde{G}_2$的構造可以看出其運算邏輯，在$u^{n+1}$或$u^{n+2}$層建立了若干個可以獨立計算的子系統，因此很適合在平行計算機上直接使用。

穩定性分析

Lemma.(Kellogg)[3] 設$r>0$，若矩陣$G+G^*$是半正定矩陣，則對任意參數$r>0$，有估計式：

$$\left\|(I+rG)^{-1}\right\|_2 \le 1, \left\|(I-rG)(I+rG)^{-1}\right\|_2 \le 1$$

Proof

首先看到G_1與G_2的子矩陣

$$A_i+A_i^T=\begin{bmatrix}1 & -1 & 0 & 0\\ -1 & 3 & -2 & 0\\ 0 & -2 & 3 & -1\\ 0 & 0 & -1 & 1\end{bmatrix}$$

$$A'+A'^T=\begin{bmatrix}1 & -1\\ -1 & 3\end{bmatrix} B'+B'^T=\begin{bmatrix}3 & -1\\ -1 & 1\end{bmatrix}$$

分別計算其特徵值，可以得到

$$A_i+A_i^T\text{的特徵值是：}3+\sqrt{5}\ ,\ 2\ ,\ 3-\sqrt{5}\ ,\ 0$$
$$A'+A'^T\text{的特徵值是：}2+\sqrt{2}\ ,\ 2-\sqrt{2}$$
$$B'+B'^T\text{的特徵值是：}2+\sqrt{2}\ ,\ 2-\sqrt{2}$$

$A_i+A_i^T, A'+A'^T, B'+B'^T$的特徵值皆$\ge 0$

因此我們知道$A_i+A_i^T, A'+A'^T, B'+B'^T$皆為半正定矩陣，所以$G_1+G_1^T$和$G_2+G_2^T$是半正定矩陣，符合引理的條件。同樣地，$A_1^*+(A_1^*)^T, A_2^*+(A_2^*)^T$也是半正定矩陣。

由式(3.23)可知

$$U^{n+2}=\mathbf{G}U^{n+1}=\mathbf{G}\mathbf{G}^{'}U^n$$

其中，

$$\mathbf{G}=(I+rG_2)^{-1}(I-rG_1)$$

$$\mathbf{G}' = (I + rG_1)^{-1}(I - rG_2)$$

也就是說$\mathbf{G}\mathbf{G}' = (I + rG_2)^{-1}(I - rG_1)(I + rG_1)^{-1}(I - rG_2)$

取$\widehat{\mathbf{G}\mathbf{G}'} = (I + rG_2)\mathbf{G}\mathbf{G}'(I + rG_2)^{-1}$

$= (I + rG_2)(I + rG_2)^{-1}(I - rG_1)(I + rG_1)^{-1}(I - rG_2)(I + rG_2)^{-1}$

$= (I - rG_1)(I + rG_1)^{-1}(I - rG_2)(I + rG_2)^{-1}$

根據上述***Lemma***，

$$\left\|(I - rG_i)(I + rG_i)^{-1}\right\|_2 \leq 1\ ,\ i = 1, 2$$

以及spectral decomposition[11]，可得

$$\rho(\mathbf{G}\mathbf{G}') = \rho(\widehat{\mathbf{G}\mathbf{G}'}) \leq \left\|\widehat{\mathbf{G}\mathbf{G}'}\right\|_2 \leq \left\|(I - rG_1)(I + rG_1)^{-1}\right\|_2 \cdot \left\|(I - rG_2)(I + rG_2)^{-1}\right\|_2 \leq 1$$

因此式(3.23)是絕對穩定的，同理可證式(3.25)也是絕對穩定。

3.3 交替分組法的CPU演算法介紹

在這節中，將會提出交替分組法的演算法，並與Explicit Forward Method, Implicit Backward Method, Crank-Nicolson Method比較其計算速度以及數值解與真解的誤差。

根據(3.24)式以及(3.25)式，建構了Alternating Group Method的演算法，圖3.4、3.5可以大致呈現其計算規則。

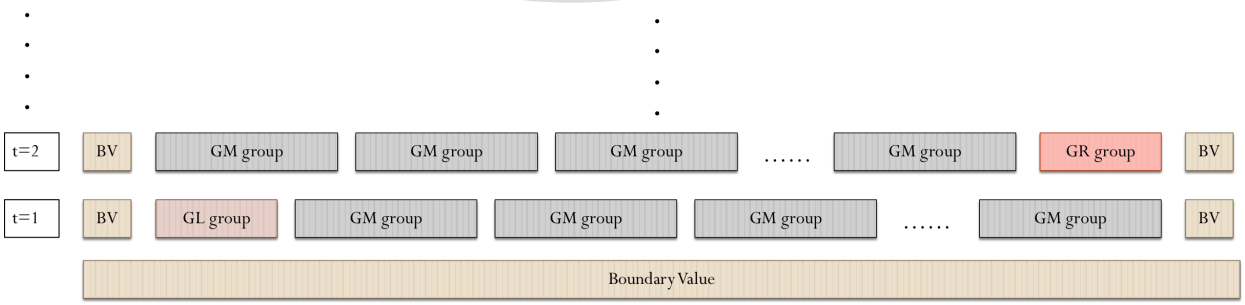


圖 3.4: The architecture of AGM-case1

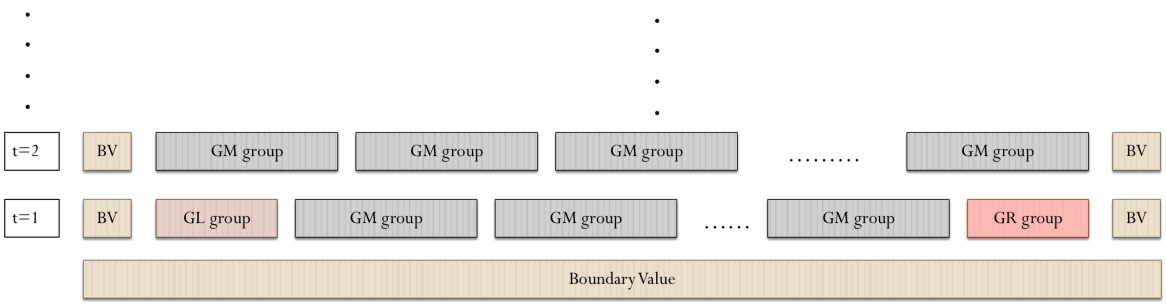


圖 3.5: The architecture of AGM-case2

我們的演算法將分成兩類做處理。

Input: meshgrid point of physical domain : **M** , meshgrid point of time domain : **N** , heat equation : $\mathbf{u_t} = \alpha \mathbf{u_{xx}}$

Output: numerical solution : **u**

GM : eq(3.15)-(3.18), **GL** : eq(3.19)-(3.20), **GR** : eq(3.21)-(3.22)

if $(M-3)\%4 = 0$ **then**
  for $j = 1; j \leq N$ **do**
    if $j\%2 = 0$ **then**
      solve GL system for left boundary : $u_{1,2}^{n+1} = GL - solution$
      for $i = 1; \leq \frac{M-1}{4}$ **do**
        solve GM system and $u_{4i-1}^{n+1}, u_{4i}^{n+1}, u_{4i+1}^{n+1}, u_{4i+2}^{n+1} = GM - solution$;
      end
    end
    if $j\%2 = 1$ **then**
      for $i = 0; il\frac{M-1}{4}$ **do**
        solve GM system and $u_{4i+1}^{n+1}, u_{4i+2}^{n+1}, u_{4i+3}^{n+1}, u_{4i+4}^{n+1} = GM - solution$
      end
      solve GR system for right boundary : $u_{M-1,M}^{n+1} = GR - solution$;
    end
  end
end

Algorithm 1: The algorithm of AGM, part 1

在第一部份的演算法，如Algorithm 1：基本的輸入包含physical domain、time domain的大小以及欲解的熱方程，而輸出就是熱方程的數值解。另外，依照(3.15)-(3.22)式中的四點組、二點組將其分別命名為GM,GL,GR。 首先，我們處理$M-3=4k$的類型，其中$k\in\mathbb{N}$，M為包含邊界條件的分點數。在這個類型底下，每一個time step的內點數為$4k+2$。

在奇數組的部分，最左邊界使用兩點組GL解一個2×2的線性系統，其餘的部分，以四個點為一組GM，個別解一個4×4的線性系統。

在偶數組的部分，最右邊界使用兩點組GR解一個2×2的線性系統，其餘的部分，以四個點為一組GM，個別解一個4×4的線性系統。

並隨著time step演進做交替分組的計算，直到time step的最後一步為止。

```
if (M − 1)%4 = 0 then
    for j = 1; j ≤ N do
        if j%2 = 0 then
            solve GL system for left boundary : u^{n+1}_{1,2} = GL − solution
            solve GR system for right boundary : u^{n+1}_{M−1,M} = GR − solution;
            for i = 1; i < (M−1)/4 do
                solve GM system and u^{n+1}_{4i−1}, u^{n+1}_{4i}, u^{n+1}_{4i+1}, u^{n+1}_{4i+2} = GM − solution
            end
        end
        if j%2 = 1 then
            for i = 0; i ≤ (M−1)/4 do
                solve GM system and u^{n+1}_{4i+1}, u^{n+1}_{4i+2}, u^{n+1}_{4i+3}, u^{n+1}_{4i+4} = GM − solution
            end
        end
    end
end
```

Algorithm 2: The algorithm of AGM, part 2

最後處理$M-1=4k$的類型，如Algorithm 2。其中$k \in \mathbb{N}$，M為包含邊界條件的分點數。在這個case底下，每一個time step的內點數為$4k$。

在奇數組的部分，最左邊界使用兩點組GL解一個2×2的線性系統，最右邊界使用兩點組GR解一個2×2的線性系統，其餘的部分，以四個點為一組GM，個別解一個4×4的線性系統。

在偶數組的部分，全部皆使用四個點為一組GM，個別解一個$4X4$的線性系統。

3.4 交替分組法的GPU演算法介紹

記得在第二章介紹PyCUDA的範例時，兩向量的內積，利用threadIdx將相同index的兩向量元素做個別相乘的動作。

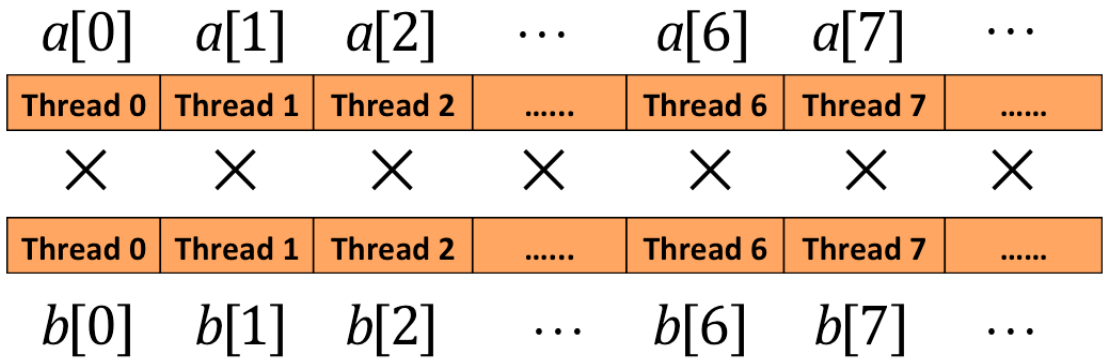


圖 3.6: index of matrix after loading in memory

如此簡單直覺的想法便是我們成功將Alternating Group Method平行化實作出來的關鍵因素。

在Alternating Group Method的兩種類別中，不管是GM,GL,GR，皆需要解一個獨立的線性系統，正因為是獨立的， 在每一個time step底下，GM,GL,GR並不會互相影響，因此這個特性很適合導入平行運算的精神，讓GPU的多重執行緒去做分工的計算。但GPU的每一條執行緒並不是如同CPU擁有強大的計算能力，也就是說儘管在numpy.linalg套件中就有解線性系統的solve函數，但並不適用在GPU身上。為了解決這個問題，在解線性系統的步驟中，我們採用克拉瑪公式來求解，底下分成四點組以及二點組來做說明：

四點組：

根據式(3.15)-(3.19)，在解線性系統時，每一次都需要用到上一個time step所對應的6個值，並且每組GM所需要的6個值並不會互相影響，如下圖：

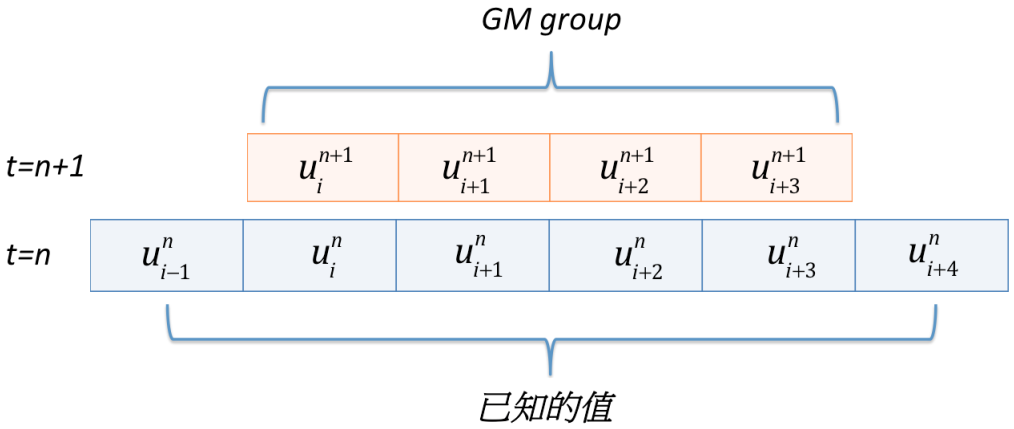


圖 3.7: The architecture of GM group

而為了配合克拉瑪公式的特性，四點組的每一個element都需要用到已知的6個值。因此，第一個步驟，我們將這6個已知的值當做一條行向量，並且將其複製出四條相同的行向量以供四點組的每一個element使用，如下圖：

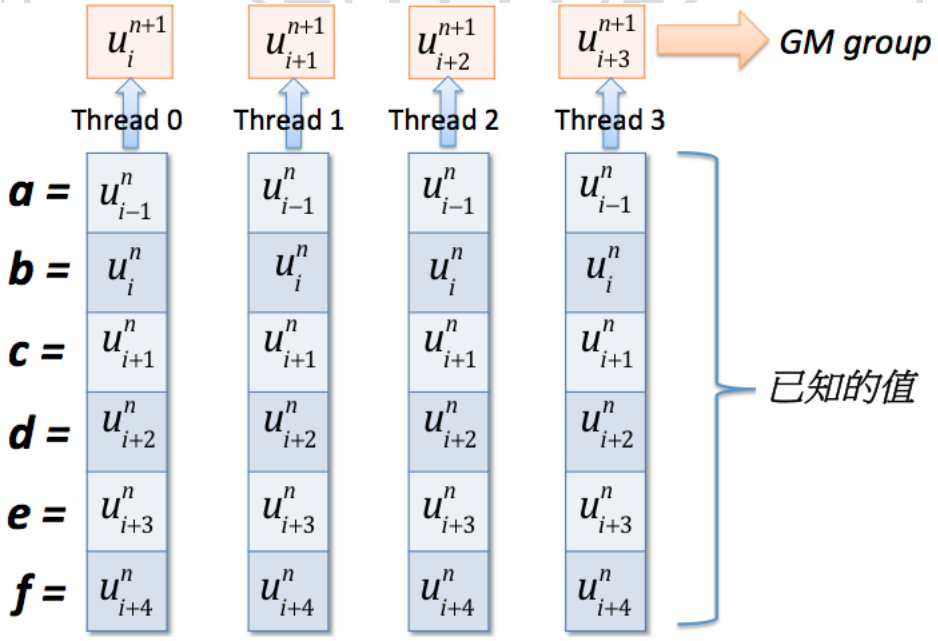


圖 3.8: copy the vector of GM

仿照第二章的範例，每一條thread依照對應的index去計算四點組的解，因此若有K組GM，則需要用到$4K$條threads，每一次time step便能同時間一起update下一個time step的數值解，達到平行運算的效果。

二點組：GL與GR

在GL中，已知的值需要用到給定的左邊界條件，如下圖：

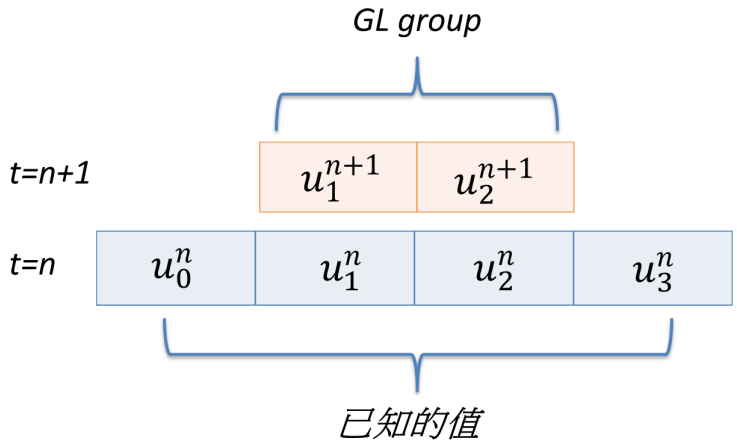


圖 3.9: The architecture of GL group

在GR中，已知的值需要用到給定的右邊界條件，如下圖：

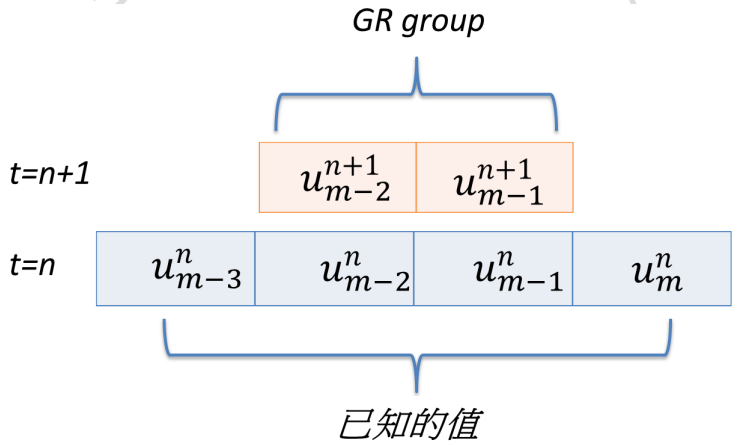
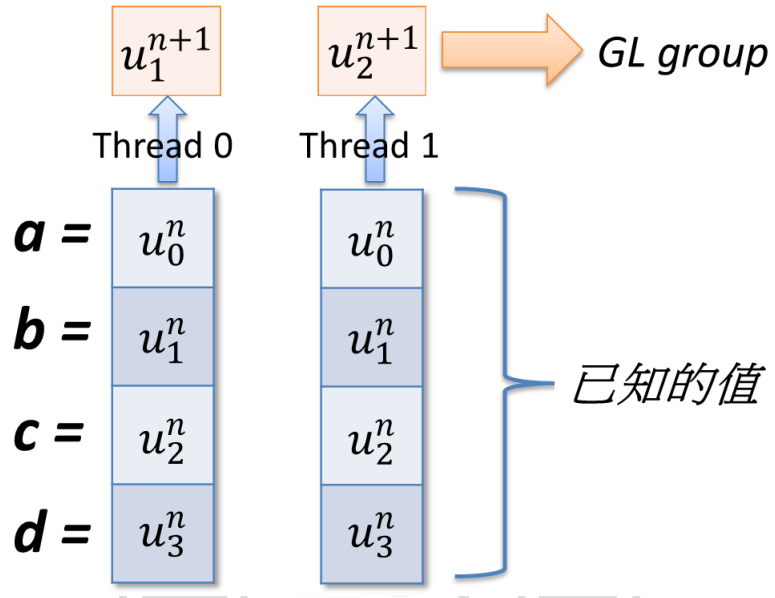
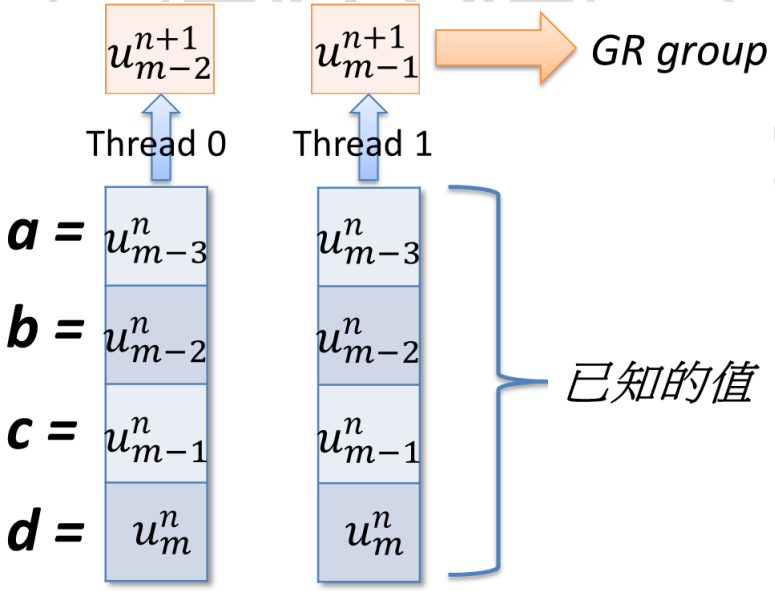


圖 3.10: The architecture of GR group

同樣的，為了配合克拉瑪公式的特性，二點組的每一個element都需要用到已知的4個值。因此，第一個步驟，我們將這4個已知的值當做一條行向量，並且將其複製出二條相同的行向量以供二點組的每一個element使用，如下圖：



(a) GL group



(b) GR group

圖 3.11: copy the vector

計算的流程如下：

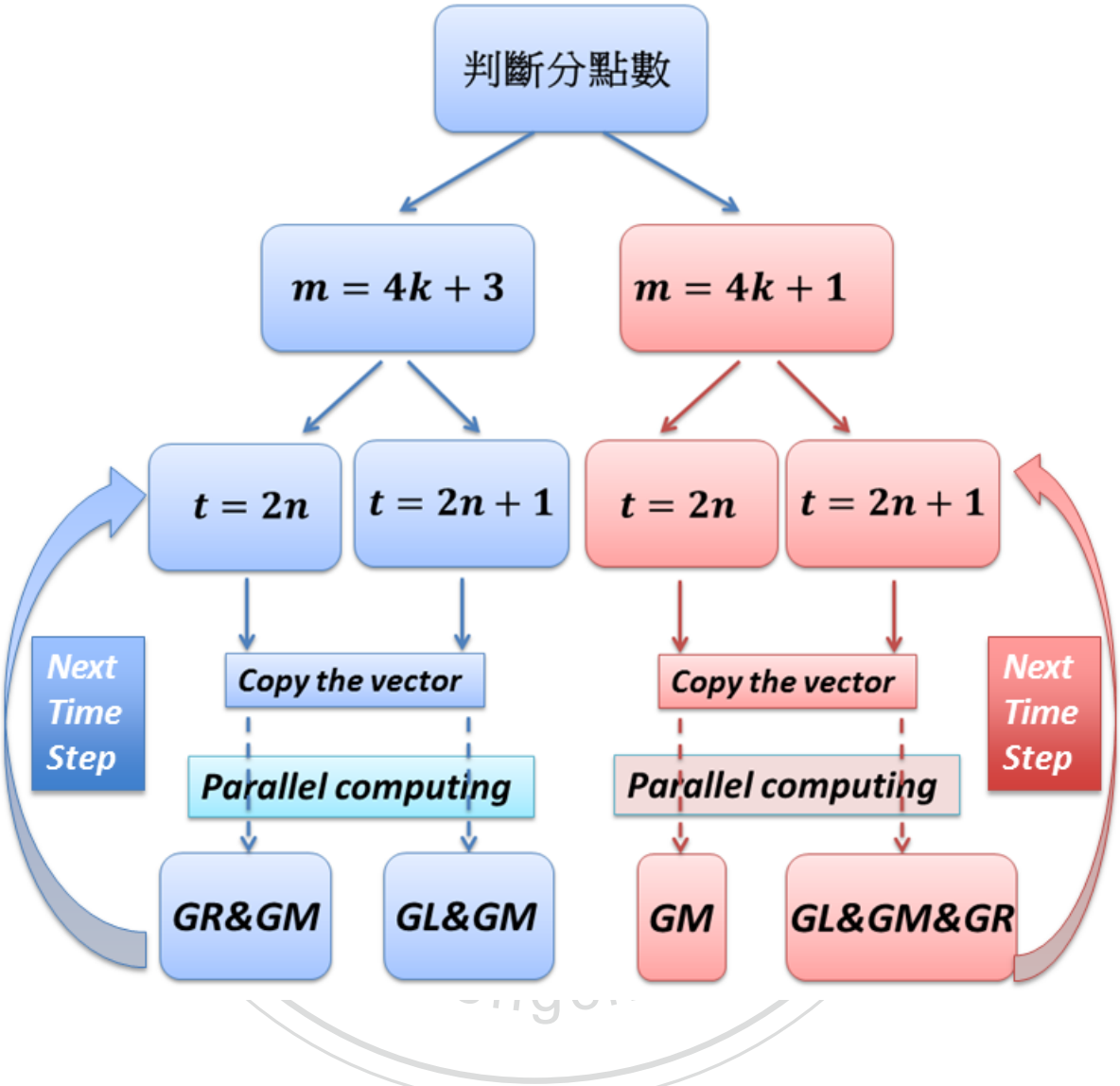


圖 3.12: The process of parallel AGM

第 四 章　實驗結果

首先，本篇先提供所使用的硬體當參考；處理器：AMD Opteron(tm) Processor 6128 2GHz、記憶體：64GB、作業系統Ubuntu 64位元12.04版、Python：2.7.1版以及實現PyCUDA最重要的顯示卡：nVIDIA Tesla M2070，其compute capability為2.0，每個block最多可以容納1024條threads，而每個grid可以容納2^{32}個左右的blocks。本章以Alternating Group Method(AGM)、Crank-Nicholson (CN)、Implicit Backward Method(IMP)為比較對象。由於Explicit Forward Method為條件穩定，因此在實驗上我們直接排除此方法。接著把PyCUDA平行運算的控制帶入Alternating Group Method來比較在Thread多寡時對計算時間的差異，證實此方法是適合平行化，並且與傳統版本的誤差相近。

4.1　AGM, CNM, IBM的誤差比較

我們用下列的熱方程來驗證四種數值分法的優劣：

$$\frac{\partial^2 u(x,t)}{\partial x^2} = \frac{\partial u(x,t)}{\partial t}, for \;\; 0 \le x \le 1, 0 \le t \le 0.1$$

且邊界條件為

$$u(x,0) = sin(\pi x), u(0,t) = 0, u(1,t) = 1$$

此方程的真解為

$$u(x,t) = sin(\pi x)e^{-\pi^2 t}$$

實驗結果如下：

首先，我們固定$h = 0.04$，而時間格點從1增加至10，也就是說Δt從0.01逐漸減少至0.001

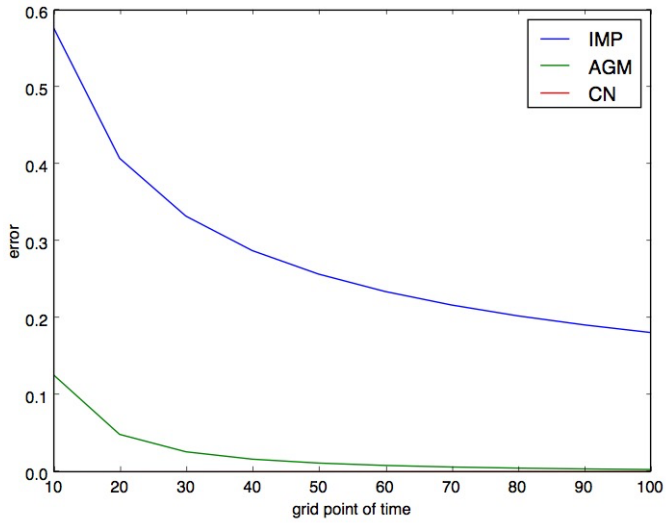


圖 4.1: error compare

可以發現，由於Implicit backward method的截尾誤差為$O(h) + O(k^2)$，因次與真解的誤差較大。Alternating group method的誤差隨著時間格點放大，已漸漸與Crank-Nicolson靠近。而由於scale的問題，底下將省略IMP的方法。

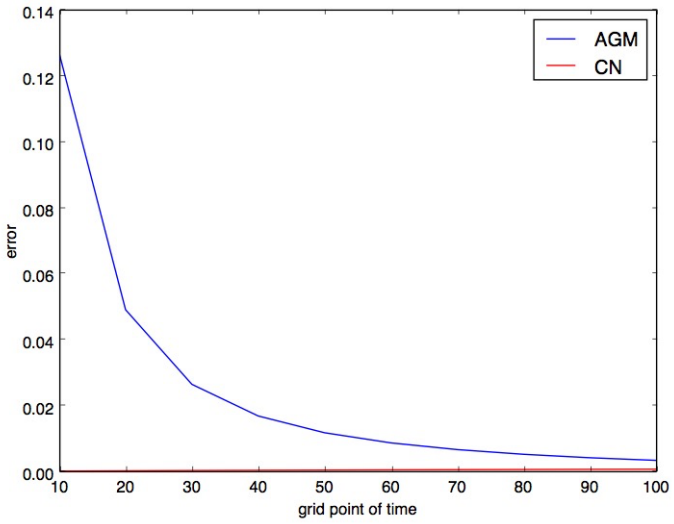


圖 4.2: error compare

可以發現Alternating group method的誤差逐漸下降，而Crank-Nicolson的誤差漸漸的在上升，可預期的是，若將Δt持續縮小，Alternating group method的誤差應該會比Crank-Nicolson要來得更好。

將時間格點持續放大，並省略前段造成scale問題的區間，可以得到以下的結果，Alternating group method的誤差已比Crank-Nicolson的誤差要來的好。

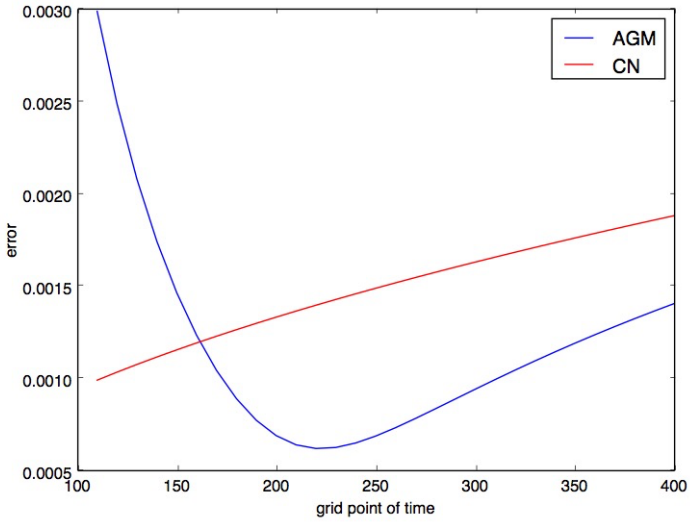


圖 4.3: error compare

持續放大時間格點，可以看到雖然alternating group method的誤差不再往下降，但誤差依舊比crank nicolson要來得好。

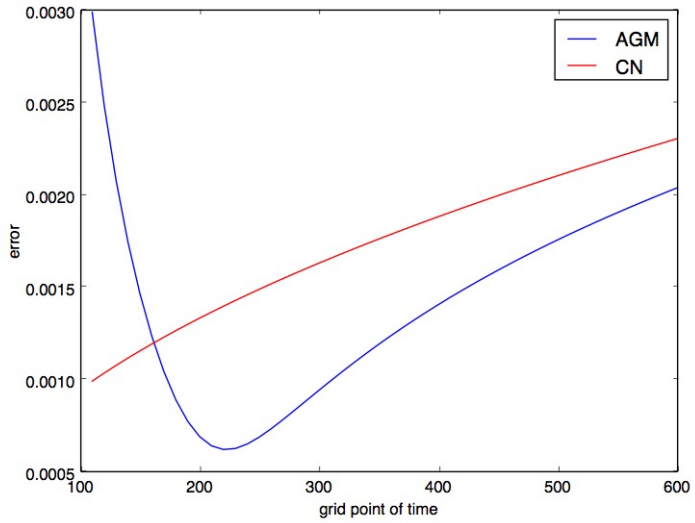


圖 4.4: error compare

接下來觀察計算時間的關係，一個很直覺的想法，要表現平行化的特性，勢必在於空間格點數的多寡，空間格點數愈多，也就是h愈小，將愈能表現平行運算的優勢。我們將h固定為0.001，時間格點從100逐漸增加至2000：

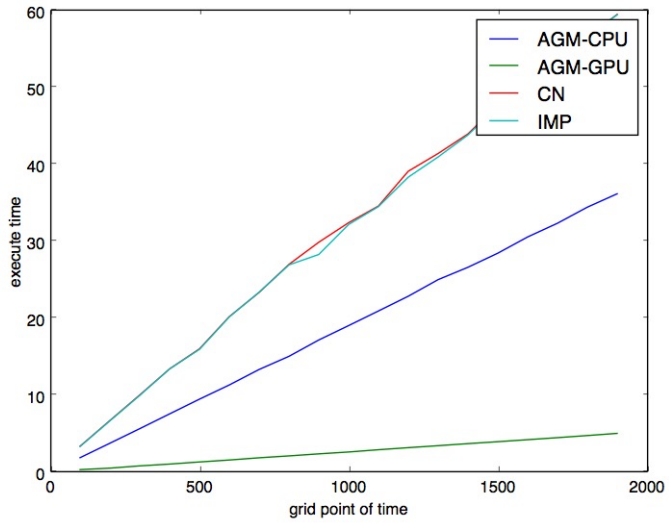


圖 4.5: execute time compare

可以看到，AGM在計算速度上比CN及IMP要來得更快，而且AGM-GPU版本更是發揮GPU平行運算的優勢，大幅縮短了計算時間。
另外可以看到平行化後帶來的speed-up：

表 4.1: speedup compare by PyCUDA

$AGM-CPU$	CN	IMP
5.977	12.495	11.862
6.829	12.380	12.350
6.840	12.449	12.521

每一個time step解很多個小線性系統的AGM-GPU版本對於每一個time step解大線性系統的CN以及IMP來說，大致可以得到12倍的speed-up，若將h再縮小，將可以得到更高的speedup，可以預想若將其應用在2D、3D的heat equation將能得到更明顯的計算優勢。
另一方面，固定$\Delta t = 0.0001$，空間格點從200增加至600：

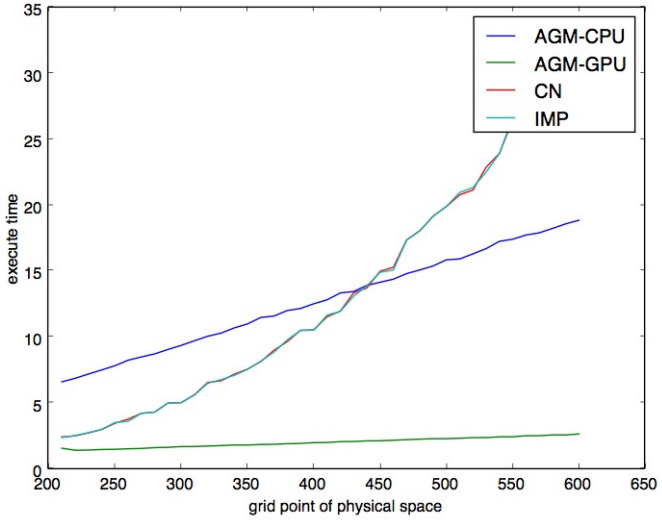


圖 4.6: execute time compare

這張圖可以明顯看出GPU的計算特性，由於GPU是每個時間點一起計算GM,GR,GL，因此在執行緒數足夠容納這些小獨立線性系統的情況下，隨著空

間格點數的增加，與其他方法相比，其計算時間只有些微的上升。
而為了避免GPU的單精度計算帶來的誤差，實際測試了AGM-CPU版本與AGM-GPU版本，與真解的誤差幾乎一樣，如下表：

表 4.2: speedup compare by PyCUDA

r	$AGM-CPU$	$AGM-GPU$
0.4205	0.0077754	0.0078453
0.5445	0.00340	0.00337
0.8405	0.00460	0.00470
1.0125	0.009857	0.009879

兩者的誤差幾乎一樣，利用T-test得到$t-value=-0.002462, p-value=0.99804$，顯示差異不顯著。因此用GPU來執行Alternating group method，不僅在穩定性、精準度以及計算速度均得到良好的結果。

第 五 章　結論

由本論文的實驗結果可得到以下結論：

1. Alternating group method交替分組法在精準度上得到提升，並且具有可平行化的特性。
2. 此方法在穩定性上屬於無條件穩定。
3. 透過在PyCUDA上實現平行化的運算，不僅在計算速度得到提升，在精準度上與傳統CPU版本近乎相同。

成功在一維熱方程完成平行化的Alternating group method交替分組法，未來可以拓展到二維、三維。至於在運算速度的提升，若加入shared memory的概念，相信會有更進一步的提升。

Bibliography

[1] Fang an Kuo. Parallel algorithm and cuda programming.

[2] Pearu Peterson Eric Jones, Travis Oliphant et al. Open source scientific tools for python, 2001.

[3] R.B. Kellogg. An alternating direction method for operations. *J. SIAM*, 12(4):848–854, 1964.

[4] Andreas Klöckner. Andreas klöckner's web page.

[5] Mark Lutz. *Learning Python*. O'reilly, 3 edition, 12 2008.

[6] Su Zhixun Nashun Bu-he. An alternating group method of parallel computing for the heat equations. *International Journal of Pure and Applied Mathematics*, 11(3):291–299, 2004.

[7] Chou Ping I. Gpu高效能運算環境—cuda與gpu cluster介紹.

[8] Timothy Sauer. *Numerical Analysis*. Pearson, 2 edition, 11 2011.

[9] V.K Saul'yev. *Integration of Techniques for Fluid Dynamics*. 1. Verlag, Berlin, spring 1988.

[10] Matthew Smith. Hands-on gpu tutorrial. 2012.

[11] Lawrence E. Spence Stephen H. Friedberg, Arnold J. Insel. *Linear Algebra*. Pearson, 4 edition, 2003.

[12] Jeng-Nan Tzeng. Glophy-數值偏微分方程.

[13] XU An-nong WANG Chen, HU Xiao-li. 求解擴散方程的一類交替分組法. *Journal of Guilin University of Electronic Technology*, 27(5), 10 2007.

[14] Wang Wenqia. Modified alternating group methods of four points for the convection-diffusion equation. 高等學校計算數學學報, 27(1), 2 2005.