高效能 Cholesky 分解法使 GPU 與 CPU 平行處理應用於帶狀矩陣

國立交通大學土木工程學系

研究生：連江祥　　指導教授：洪士林 博士

摘要

線性系統求解隨著矩陣之變大，所需要之記憶體以及處理時間隨之增大與拉長，一直以來都有許多研究如何將平行運算理論應用在線性系統求解上。過去大部分的平行計算之研究都著重於疊代求解演算法，並使用分散式平行運算技術。疊代法雖然擁有平行成果較好，但演算法本身適用性較為狹小，且較適合應用於大型電腦上。

本研究之重點則著重於較難平行的直接演算法，及分別使用較新的多核心 CPU(Multi-core)及 GPU 平行運算技術，希望可以達到較好的計算效能，同時也可節省計算所需之記憶體與時間。本研究主要分為三個步驟。首先使用多核心(Multi-core)將基本的直接演算法平行化而後作精度與效率之比較，尋找最適合的線性求解演算法。接著以最適合的 Cholesky 演算法做為研究基礎，並且參考相關期刊的 Cholesky 磚塊化切割理論再加以進行優化與改良；最後將優化過後的演算法，透過 GPU 平行運算，以達到最佳的計算效果。在四核心系統下，當帶狀矩陣中之寬帶大於 100 則採用多核心技術可達到相較於單核心 2.3 倍以上效能，而寬帶大於 1000 可達 3.3 倍以上效能，而在 GPU 上使用 CUDA 技術則可達到 10 倍以上效能，隨著矩陣寬帶之增大在 GPU 上會有更佳的實數運算效果。

關鍵字：Cholesky 、CUDA、OpenMP、解算器、寬帶矩陣、平行

High-performance Cholesky Factorization using the GPU and CPU parallel processing for band matrix

Student：Jiang-Siang Lain　　Advisor：Dr.Shih-Lin Hung

Department of Civil Engineering, College of Engineering

National Chaio Tung University

Abstract

The required memory storage and processing time will be increased and elongated when solver linear system in larger matrices. Hence, the application of parallel computing technology on solving of linear system has received considerable interest in the last decade. Most of the parallel computing technologies of the previous studies have focused on iterative algorithm on the distributed parallel computing platforms。 However, the performance of iterative algorithms can realize only for matrices with larger-scaled linear system on super computers.

The aim of this study focuses on developing more complicated direct parallel algorithm, on the multi-core CPU (Multi-core) and GPU parallel computing platforms. There are three stages in this study. First, the direct linear system solving algorithms are parallelized and implemented on the multi-core platform. The computing time and precision of solution were investigated and compared to conclude the performance of these different algorithms. Following, the blocked-Cholesky algorithm was utilized and optimized to develop a novel parallel algorithm. Finally, the optimized novel blocked-Cholesky algorithm was implemented on multi-core CPU and GPU parallel computing platforms. The computing results revealed that a 2.3 speed-up achieved fir band-matrices of bandwidth greater than 100 on a four-core platform as compared with performance on a single-core platform. Moreover, the computing performance accomplished 3.3 when the bandwidth of matrices greater than the1000. Notable, a ten-time performance can be reached when the novel algorithm was implemented on a platform of GPU with CUDA technology. The results also revealed that the more the bandwidth of matrices, the higher the achieved performance for computing on GPU platforms.

Keyword；Cholesky 、CUDA、OpenMP、band matrix、parallel

致謝

首先誠摯的感謝指導教授洪士林老師，在研究的道路上老師讓我有所發揮的空間並不斷的替我修正與指引正確方向，使我在平行運算領域中充滿了熱誠與鬥志讓我鎖定目標勇往直前，真心地感謝老師的欣賞。再來感謝詹君治大學長，兩年來的嚴謹叮嚀，讓我在期刊閱讀與簡報能力有大幅度的成長。

花費了一年半的研究時光，論文從無到有歷經了不少的挑戰，從 c 語言未使用過一直到迅速熟悉、一直到踏入國家網路中心學習後，本論文才正式跨入研究門檻。感謝國家網路中心推廣 GPU 平行運算使得我減少大量的摸索時間。其中特別感謝吳自勝同學與我有相同的資訊嗜好，一起熬夜架設 Linux 伺服器，與我一起征戰了許多研討會與資訊展覽，不斷的吸取新知識，讓我研究不孤單且充滿了樂趣。

長時間編譯程式是枯燥乏味的，資訊與結構組雙重身分讓我的交友圈更加豐富，410 研究室是個甜蜜的負擔把它是我的老家，作業完成就是要來一場遊戲，研究室總是待到半夜才甘心，總是充滿了奮鬥與歡樂，感謝思伶、綸桓、孟軒與學弟：宣治、俊佐、晟佑，是我教學相伴的『好戰友』。401 研究室是我的新家，資訊組的學弟妹的很窩心很用功，文平、懿函、廣德、亦中、哲豪感謝你們這一年來的支持與愛戴。最後是 417 研究室，是我一起上山下海的好夥伴，鈞誠、宣妤、維莘、進順你們要一直嗨下去，有你們在我一點都不覺得無聊。另外感謝學長：家宇、凱平、宗輝、承哲帶領我去健身房運動，讓我在最後的研究生活有運動陪伴，我的肌肉就如同研究一樣漸漸長大茁壯。

最後，謹以此文獻給我摯愛的雙親，”媽!我是碩士了!”。

目錄

表目錄

圖目錄

第一章 緒論

1.1 研究動機

有限元素軟體計算隨著所解問題自由度的增加而所需計算時間亦隨之增長，若以提升硬體來縮短計算時間所需要的花費都相當地高昂。使用一般桌上型電腦，在軟體設定上切割元素將被限制在一千個元素以內，若需使用 3D 元素或不規則模型下，PC 將可能不敷使用。

近來，利用大量的叢集化電腦系統架構將可解決上述問題，主要方法一是經由個人架設昂貴的叢集電腦並使用分散式平行運算技術藉以達到省時的功效但費用卻要花費近百萬，或者申請學術機構或國家級電腦中心所建置的公用大型叢集電腦，但此方法卻需要經由相關單位排程等待耗費時間將更為漫長。

除了分散式運算外，近來多核心電腦架構所發展的程式語言拓展應用套件『OpenMP』編譯已漸漸成熟【1】，多核心 CPU 已經非常普遍且實用，且比起傳統的分散式運算所需要的叢集架構硬體成本降低許多。另一方面，GPU 更是擁有上百顆計算核心其單精度實數運算效能比起當下頂級 CPU 約有 10 倍的效能，目前多種 GPU 平行運算技術中由 NVIDIA 公司所發展的程式語言拓展應用套件『CUDA』，此套件可以省下大量的硬體建置成本達到高速運算的效果，實際效能往往可與大型電腦媲美，CUDA 雖然編譯難度較高限制較多且目前編譯器仍然在不斷發展更新，但所產生的效能在耗時的科學計算是據有相當吸引力。

1.2 研究目的

由於有限元素法之程式分析流程中，絕大部的計算時間都消耗在求解有限元素所建構之大型線性系統當中，直接求解法基本程式中其計算量將是該線性系統矩陣大小之三次方 $O(n^3)$的計算量而記憶體空間需求為矩陣大小之二次方 $O(n^2)$，所以需要花費大量的計算時間與記憶體空間。

所以本研究之目的，希望運用現今平行計算技術及硬體架構發展一套有效的演算法以解決上述問題，首先將選定效率較高且適用性較好的直接求解法做為基礎，以此來發展出適合於寬帶矩陣的平行演算法，並將利用現今平行處理技術 OpenMP 及 CUDA 分別用於多核心 CPU 架構與效率較高的 GPU 系統上發展出高效率的平行處理演算法，藉此達到節省時間、降低所需記憶體的目的。本研究將以主流的數學函數機構所制訂的壓縮格式來編寫程式，希望可以將本程式以物件函數方式嵌入軟體，應用在符合線性求解系統之各種軟體領域當中。

1.3 研究流程、範圍

本研究使用較先進的 OpenMP 及 CUDA 平行發展平台並採用較不易平化的直接求解法來進行研究，並將之應用於 100~1000 的帶狀矩陣上。

研究方法分為三個主要步驟：

第一步驟：經由幾種常用的消去法、矩陣分解法進行 OpenMP 的平行運算測試比較其中精度與效率，最後選定了精度與效率都很高 Cholesky 演算法做研究基礎。

第二步驟：並且利用矩陣帶狀且對稱的特性，可將計算量大幅減少可節省大量時間。進一步，將二維正定矩陣壓縮寬帶成為一維的向量，省下大量的記憶體，解決惱人的記憶體不足問題。參考期刊區塊化 Cholesky 演算法，改良將原本三階切割簡化為二階增加些許計算量可使得傳輸次數降低平行化程度更高，演算法中加入 Gauss 概念求出反矩陣，增加不可平行的效率，計算也多出平行效率很高的矩陣相乘，經 OpenMP 測試後達到良好的效果。

第三步驟：利用此演算法投入 CUDA 的 GPU 運算，加以改進致適合 GPU 運算的方式，以期望達到更佳的計算效率。

第二章 文獻

2.1 平行運算發展簡介

平行運算發展至今在已是個悠久的歷程，不同硬體架構延伸出不同的傳遞信息技術。事實上，對於傳遞信息技術來說，20 世紀 60 年代美國就已出現了最早的平行計算機、70 年代之後也曾經出現過平行向量機（Cray 公司為代表及中國銀河系列），90 年代初的 SIMD 平行計算機 (ThinkMachine 公司的 CM 系列)，通過大量體系由結構簡單、功能較弱的處理器用網路連接實現大規模的並行計算系統，但由於應用領域有限等原因，這類系統壽命很快就消亡了【2】。過去處理器一直以來仍依照摩爾定律之十八個月增加一倍的電晶體數不斷發展，處理器的時脈增加速度超越了建構平行電腦與編譯平行程式的時間，以至於平行電腦架構的使用壽命短、實際價值低落。

如今平行運算應用價值較高，叢集架構(Cluster)解決了傳輸速度較慢問題，解決了過去長久以來平行電腦效能低落的問題並解擁有良好的擴充性，叢集架構無疑地將平行運算開啟了實用價值的一面，可將多台普通的個人電腦透過高速網路連接組成叢集電腦，讓平行運算不在只是高端硬體所使用的工具，架設叢集電腦一直至今仍然是目前的主流。

另外一個重大發展是處理器往單晶片多核心發展，原因是電路設計的物理極限，單個處理器的線寬總有一天要達到物理極限，且散熱和漏電是兩個迫使我們不得不轉向多核處理的深層次的工業原因，工業界在 2005 年突然要面臨一個拐點，所以不得不轉向多核心發展，將多核心架構普及到一般大眾。 90 年代末就已經有人在做多核心處理器的研究，其思路是把功能簡單的處理器用網絡連接起來，互相協作來解決延遲的問題。比較

早的是 RAW 處理器，由美國 MIT 大學開發，是我們目前稱為 Tile 結構處理器的先驅。多核心處理器的出現實際上是一次平行計算方式的革命。一般大眾不得不面對平行操作這些通常是並行計算的專業人員和高端用戶才需要面對的問題。

如今平行運算應用價值提高廣泛應用於各種領域，在科學運算方面：奈米科學、工程計算、石油探勘、材料科學、核爆模擬、數值氣象預測、醫學影像、流體力學、分子生物蛋白質序列分析.....等領域都已大量使用平行運算為必要工具之一。在一般應用上：解壓縮、轉檔、影音播放軟體、繪圖渲染器、遊戲物理效果、瀏覽器.....等，現今平行運算已不在是高端用戶所使用的工具，已深入每一位電腦使用者當中。

在平行計算領域裡，最有名的兩個定律是 Amdahl 定律【3】和 Gustafson 定律【4】。Amdahl 定律指出，如果一個算法裡不能平行的部分所佔的比重是 10%的話，那麼並行化算法所能達到的最大加速比超不過 10。這個定理出來以後，對平行計算打擊很大。後來 Gustafson 發現，實際上 Amdahl 定理存在的問題是只假定並行系統處理一個固定規模的問題，在這種情況下，再增加處理器當然沒有意義。但如果把問題規模隨著機器規模一起變大，加速比仍然可以變大。Gustafson 定理出現以後，平行計算機的發展前途豁然開闊。

2.1.1 多核心計算的發展

平行是一個應用的概念，根據硬體架構與實現的層次不同，可分為幾種方式。如圖 2.1-1 所示，最下層的是單核指令級並行(ILP)，單顆核心可執行多條指令，可將單核心的運作效能更接近理論值；第二層是多核

(multi-core)平行，即在一個晶片上集合了多個處理器核心，實現線程級平行(TLP)，共享同一區塊之暫存記憶體：第三層是多處理器平行(multi-processor)平行，在一塊主機板上安裝多顆處理器，一般使用在伺服器或大型電腦；最後，可以借助網路實現大規模的集群或者分散式平行，每個節點(node)就是一台獨立的計算機，透過網路將其連接，實現更大規模的平行計算【5】。

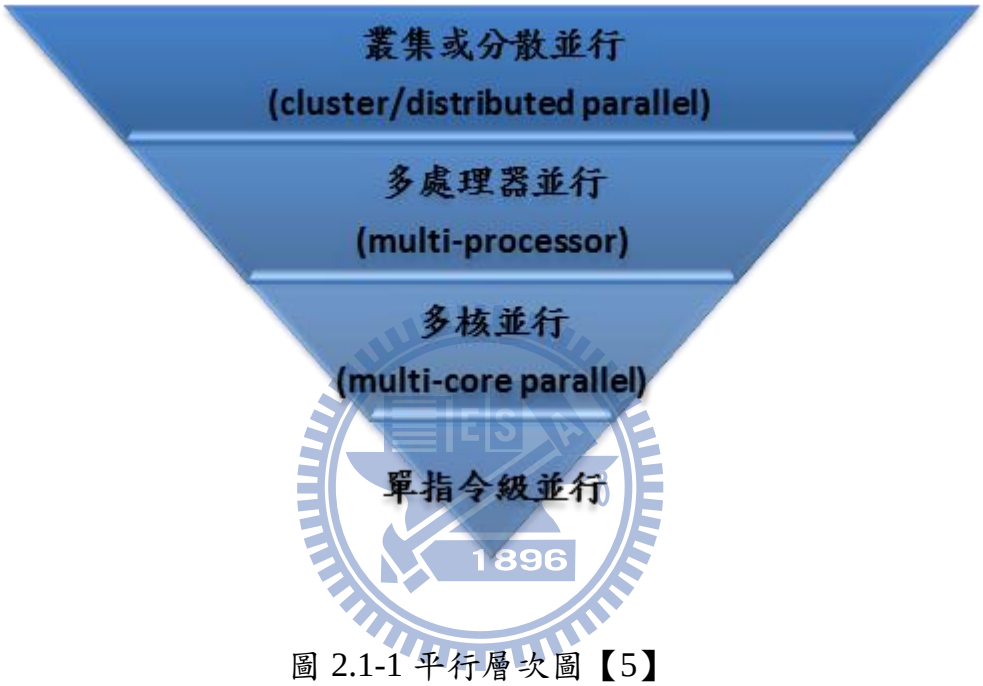


圖 2.1-1 平行層次圖【5】

隨著平行運算架構的發展，平行運算編譯方式也不斷的成熟與優化。從歷史上看，並行算法研究的高峰是 20 世紀七八十年代。此階段，出現了許多採用不同互連架構和存儲模型的 SIMD 機，出產許多優秀的平行演算法，奠定了平行運算基礎【5】。但由於處理器電晶體數量發展快速，實際應用上平行處理並不實際。到了 20 世紀末，平行處理的研究更加注重應用層面，不但研究平行演算法的設計與分析，也兼顧平行硬體結構與平行應用的程序設設計。

2.1.2 三大主流架構與對應編譯語言拓展

1.CPU 多核平行：

2003 年 Intel CPU 開始量生產多核心晶片，共享記憶體已經並非是超級電腦的專利，越來越多的程式設計師業意識到了多線程編譯的重要性及普遍性。多線程編譯既可以在多顆 CPU 核心間實現線程平行，也可以通過超線程技術更好地利用每一個核心內的資源，充分利用 CPU 的計算能力。目前最常用的多線程編譯語言擴展是 intel 所發展的『OpenMP』。

2.超級電腦、叢集與分散式計算：

超級電腦指在性能上居於世界領先地位的計算機，通常有成千上萬顆處理器，以及專門設計的內存與 I/O 系統。採用的架構與個人電腦有不小的區別，技術也隨改變。但在程式上超級電腦與普通 PC 關係仍然十分緊密，超級電腦的技術也都能普及到普通電腦中。計算機叢集是一種通過高傳輸速度網路，將許多台電腦主機並聯在一起。目前超級電腦與叢集計算中常用的工具是『MPI』(Message Passing Interdace)

3.CPU+GPU ：

CPU 由於處理器需要具有全面有較大的邏輯能力以及大容量的暫存空間，多核心的發展大約依照摩爾定律每 18 個月一倍速度發展。 GPU 是天生的多核心平行架構，目前 NVIDIA GPU 約有 512 顆 CUDA 核心，發展速度超越摩爾定律，單精度能力大約是主流 CPU 的 10 倍左右，如圖 2.1-2 所示【6】，因此我們可以將 GPU 用非傳統的 3D 圖形上，這種平行計算應用稱為 GPGPU。由於此技術難度高，程式有許多部分難以平行。目前 NVDIA 提供了一個關於 GPGPU 與 CPU 的平行運算架構整合技術，稱為『CUDA』

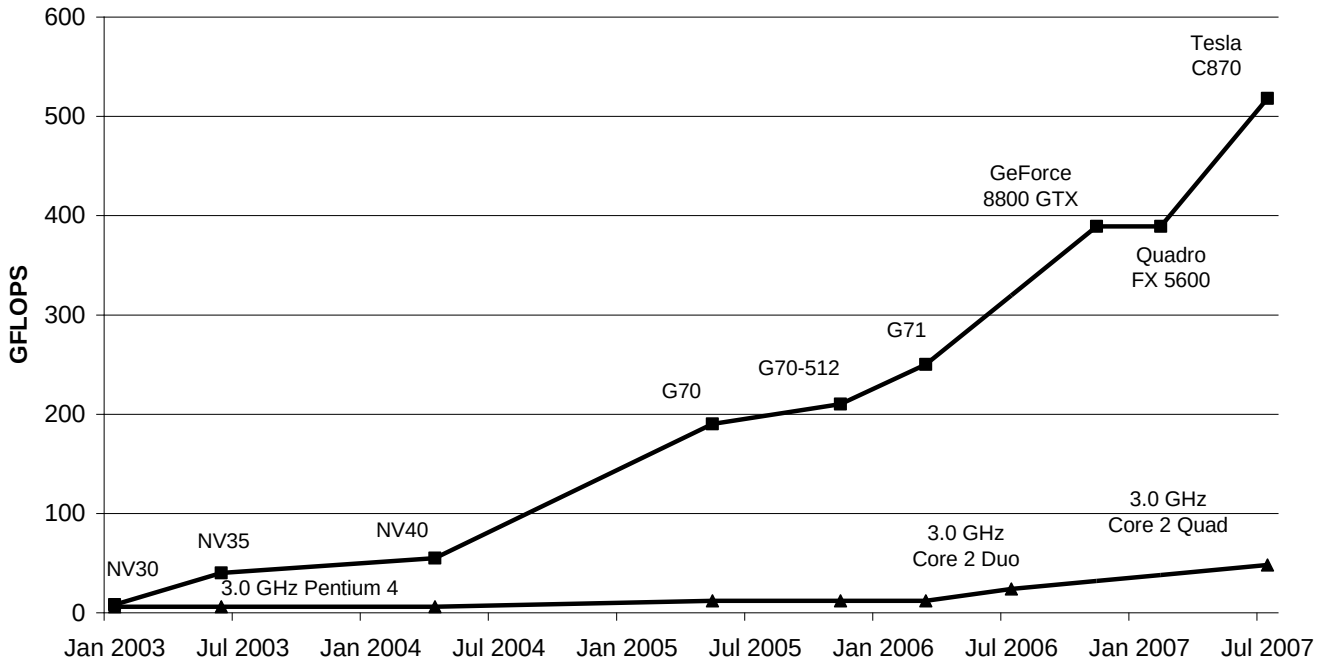


圖 2.1-2 GPU 與 CPU 之實數運算能力【7】

2.2 SPMD 架構

平行程式的計算過程中，根據程式的平行層次可分為兩類：一是SPMD，另一則是MPMD【8】。 SPMD (Single Program Multiple Data)，通常是應用於相同的多處理器所建構的『均質』平行計算環境下，在計算過程是執行相同程式碼但每個平行處理器運算不同組的資料。

SPMD 的平行程式著重迴圈平行(loop-parallel) 與資料平行(data-parallel)的建構，也是將資料與計算平均分散至整個計算環境下進行平行處理。MPMD (Multiple Program Multiple Data) 可應用於非均質多顆處理器環境下，在平行計算過程中不同的處理器可執行的不同程式碼、處理不同的資料，一每顆核心不同能力依『比例』分配工作，常應用於不同型號電腦所架構的叢集電腦，或使用在多不同等級之險式晶片卡上。本研究平行計算應用在SPMD架構上。

2.3 OpenMP 簡介

在 1990 年早期，共用式(shared)記憶體機器的製造供應商提供類似以指令為基礎的Fortran語言的擴展程式，使用者只要增加一系列的Fortran程式指令就可使程式中的迴圈平行化。於1994 年時，美國國家標準局ANSI(American National Standards Institute)X3H5 小組第一次企圖擬定OpenMP的標準化，但沒被採用，主要是因為當時分散式記憶體架構的電腦蔚為主流，直到1997 年時，OpenMP才開始標準化，其發展歷史如下【1】：

□ 1997 年10 月：Fortran 1.0 版□ 1998 年晚期 ：C/C++ 1.0 版

□ 2000 年6 月 ：Fortran 2.0 版□ 2002 年4 月 ：C/C++ 2.0 版

□ 2004 年11 月：Fortran、C/C++草稿版(由OpenMP ARB (Architecture ReviewBoard)宣布，並公開給大眾修正此版至2005 年1 月)OpenMP是一個應用程式介面(API)，適用於多執行緒(Multi-Threaded)、共用記憶體平行化架構上，主要包含三個主要的應用程式介面【1】：

1. Compiler Drectives

2. Runtime Library Routines

3. Environment Variables

雖然多核心電腦已成為目前電腦之主流， 但傳統循序程式在多核心電腦上並不能充分發揮處理器之效能；唯有將循序的單一執行緒程式改寫成多執行緒的平行程式，才可充分運用多核心之優勢，藉由多執行緒平行處理資料以加速程式執行之速度。不過，要撰寫多執行緒程式，對於執行緒工作之分派及控制要下很大的功夫。POSIX Threads 是選擇之一，然而，其執行緒之控制較為複雜；若僅是單純地將迴圈加以平行化，OpenMP 是

一個理想的選擇。實際撰寫程式時，程式設計師只要透過幾個OpenMP 專用的高階指令，就能將單一執行緒的循序程式轉化成多執行緒的平行程式；程式設計師比較需要注意的是程式本身之邏輯正確性，以及演算法的效率問題，至於如何分配執行緒之細節問題，就交由OpenMP 處理了。

2.3.1 OpenMP 平行模式

一般的共用式記憶體系統提供thread靜態(Static)與動態(Dynamic)的產生【9】，動態產生thread的方式如圖 2.3-1，master thread藉著fork的方式在欲平行計算的程式碼區域(也就是圖 2.3-1中的parallel region)最前面啟動其它的thread，在執行完平行計算的程式碼區域後，master thread會在欲平行計算的程式碼最後面以join的方式等待剛被呼叫的thread，等到全部的thread都執行完程式碼後，會剩下master thread繼續執行sequential程式，而其它的thread則會終止。而靜態(Static)的thread切割方式較為簡單，將迴圈平均切割成等份，每個thread擁有自己分配的工作量，當其中之一thread做完運算量後將不會協助其他thread，會等待所有thread完成工作才會執行下一步。

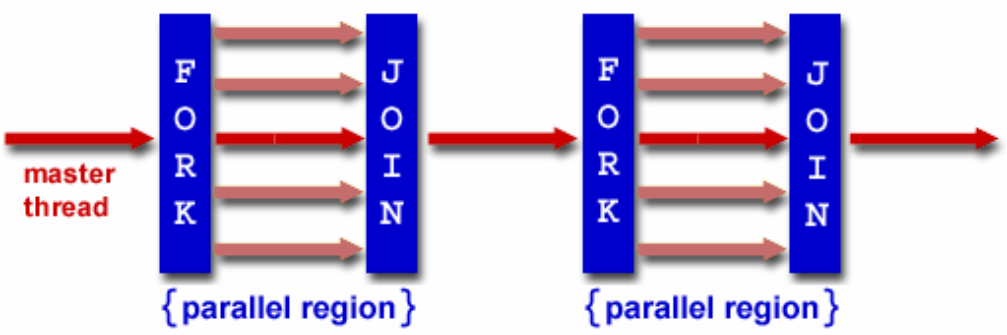


圖 2.3-1 OpenMP Fork - Join Model【1】

2.3.2 共用式記憶體多處理器系統

共用式記憶體以往通常應用在超級電腦，現經多核心處理器也都擁有共享記憶體，以程式設計來講，較容易撰寫且具有可攜性(portable)，但缺點為價格昂貴，且擴充不易、發展有限，目前最常使用的平行程式函式庫為OpenMP。圖 2.3-2為一般常用的共用式記憶體架構。

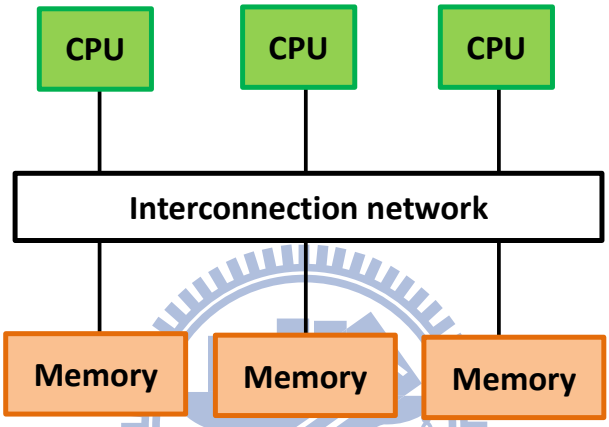


圖 2.3-2 一般常用的共用式記憶體架構(由【10】重新繪製)

此系統中記憶體為透過網路內部連接成共用式記憶體的方式，可分為以下兩種【9】：

1. Bus-Based 架構

此架構中每個CPU都有一個cache，且共用一個匯流排，如果每個CPU同時存取資料，則匯流排的頻寬會達到飽和，造成存取資料延遲。故因為匯流排頻寬的關係，此架構無法架設太多的CPU。一般的Bus-Based 架構如圖 2.3-3。

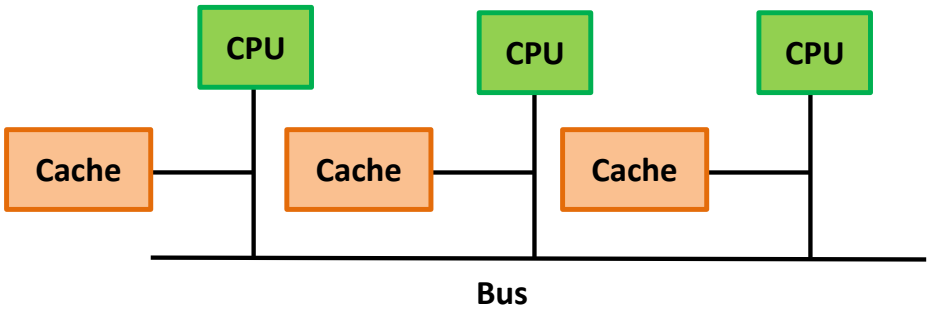


圖 2.3-3 Bus-Based 架構(由【9】重新繪製)

2. Switch-Based 架構

大多數的共用式記憶體多處理器系統使用此架構為主的不同型式。在此架構中，任何CPU都能同時存取任何記憶體的資料，且不會衝突或干涉，所以沒有Bus-Based架構的頻寬飽和問題。唯此架構的成本非常昂貴。一般的Switch-Based 架構如圖 2.3-3。

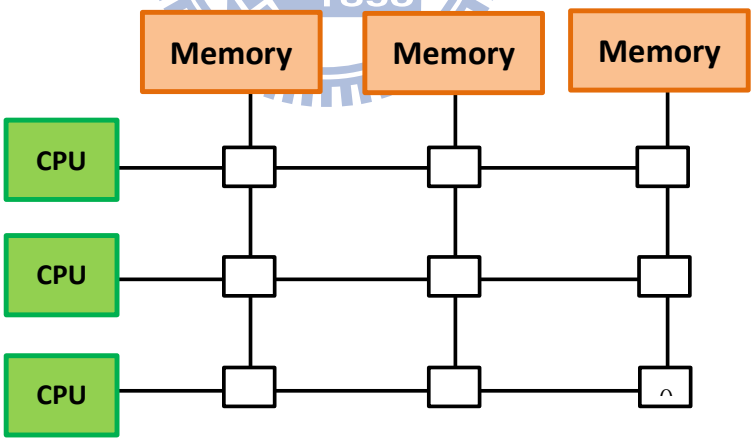


圖 2.3-4 Switch-Based 架構 (由【9】重新繪製)

2.4 CUDA 介紹

將顯示晶片GPU 用來做為傳統遊戲、繪圖、及3D顯示以外的計算用途，一般稱為GPGPU。GPU 原本最初設計的用途為特定的圖形運算處理，然而隨著這些晶片的可程式化程度不斷提升，GPU 進而可以作為CPU 的輔助處理器；也由於高速運算的能力使得許多的應用更適合用GPU來執行。

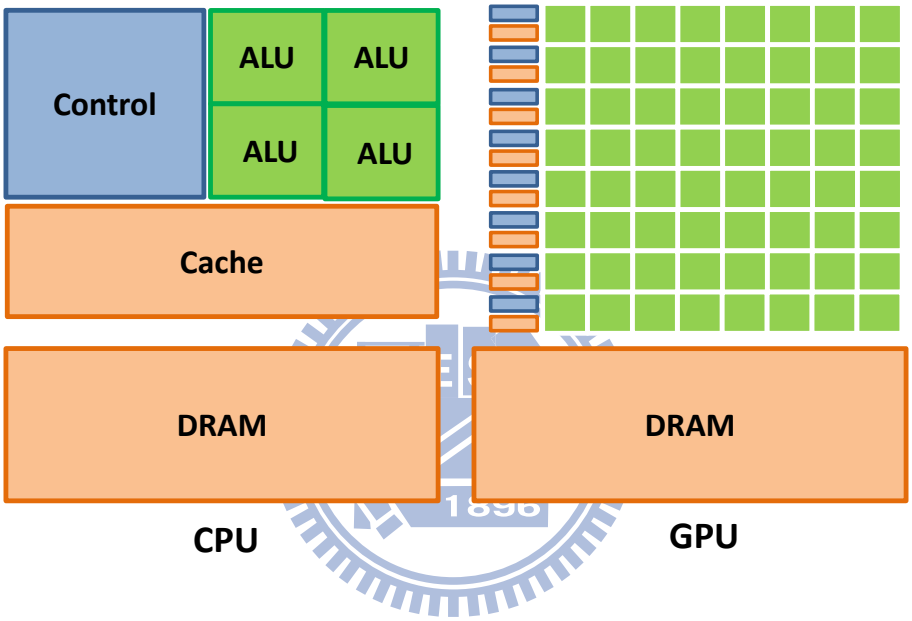


圖 2.4-1 CPU 與 GPU 的架構差異示意圖(由【11】重新繪製)

早期只有CPU 可以進行複雜的浮點數運算，而CPU 原本的設計就是為了處理通用的數學運算。而與此同時開發的GPU，並不是使用浮點運算進行2D 或3D 的圖形處理，GPU 是一種圖形處理專用的特定加速硬體，尚無法達成其他設計初衷之外的運算功能。然而Microsoft 在2001 年推出的DirectX 9 讓GPU 支援浮點數運算，讓繪圖晶片可程式化，開創了在通用運算上的發展性，GPU 上的著色器(shader)可程式化，讓程式開發者可以透過控制shader 來做需要的計算。透過OpenGL 或是DirectX 這一類現有的圖形函式庫，以編寫Shading Language的方式來控制shader 是最傳統的GPGPU 開發方式；然而此種方式有一定的限制，程式開發者必須對

OpenGL 或DirectX 這類的繪圖API 有一定程度的瞭解難度較高，因為開發者得透過計算機圖學的方式將資料傳給GPU 來處理，而事實上顯示晶片也是藉由這類的API 來達成圖形的計算，此種方式不僅提高了GPGPU 的開發門檻，對於GPGPU 的開發以及研究的普及率都有一定的影響【12、4】。

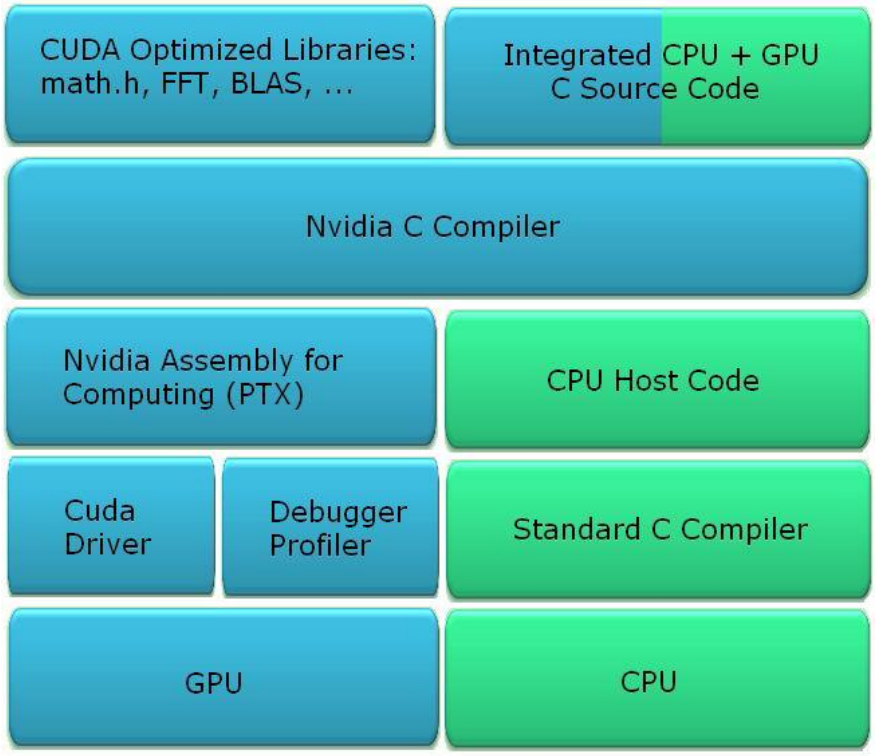


圖 2.4-2 程式在 Nvidia CUDA 平台上的 GPU 與 CPU 編譯架構【12、4】

傳統的中央處理器架構，以Intel 的Xeon 處理器為例，最多只有4~6個核心，然而新推出Nvidia GeForce GTX580 繪圖晶片，卻擁有512 個處理核心，多核心的架構比傳統的處理器架構更適合作為超級電腦的平行運算運用。GeForce GTX580 顯示卡的運算能力可達到1Tflops以上，利用可擴充PCI-E介面(ScalableLink Interface)系統，就可以達到更驚人的效果。因此目前包括ATi 與Nvidia 等繪圖晶片廠商都相繼推出相關的GPGPU 產品，如今Intel與AMD相繼投入CPU+GPU處理器，讓處理器的浮點運算能力也能接近teraflops等級。GPGPU 最大的優勢就是強大的運算能力，但缺

陷是應用程式必須改寫門檻與難度較高，由於繪圖處理器與中央處理器的架構具有一定程度的差異，除非是全新開發的程式，否則要利用超級電腦來發揮平行運算的功能，通常都要經過改寫的步驟。此外開發者為了將資料傳給GPU 處理，也必須對圖學相關的API 有一定的知識背景，以人力成本考量這並非是一種非常有效率的開發方式。

而CUDA 則是Nvidia 針對GPGPU 提出的一種通用型平行運算架構，是Nvidia公司對於GPGPU 的正式名稱，其中包含了CUDA 指令集架構和繪圖處理器中的平行運算引擎。而CUDA 的開發使用C 語言作為基礎，在GPU 上能夠讓程式開發者使用C 語言的環境，因此CUDA 降低了GPGPU 的開發門檻。此外因為不用使用圖形函式庫，所以不必遵循render pipeline 的固定流程來設計程式，對於軟體的開發更為方便。針對CPU 和GPU 之間的資料傳輸(如圖2.4-3)，CUDA 提供了一套專屬的驅動程式用以進行快速的資料傳送，而對於傳統的GPGPU 開發函式庫，例如OpenGL和DirectX 繪圖驅動程式也可以同時交互運作，無論是CUDA C 語言或是OpenCL，最終驅動程式都會將指令集轉為PTX(Parallel Thread Execution，一種pseudo-assembly language)代碼，交由顯示核心計算【11】。

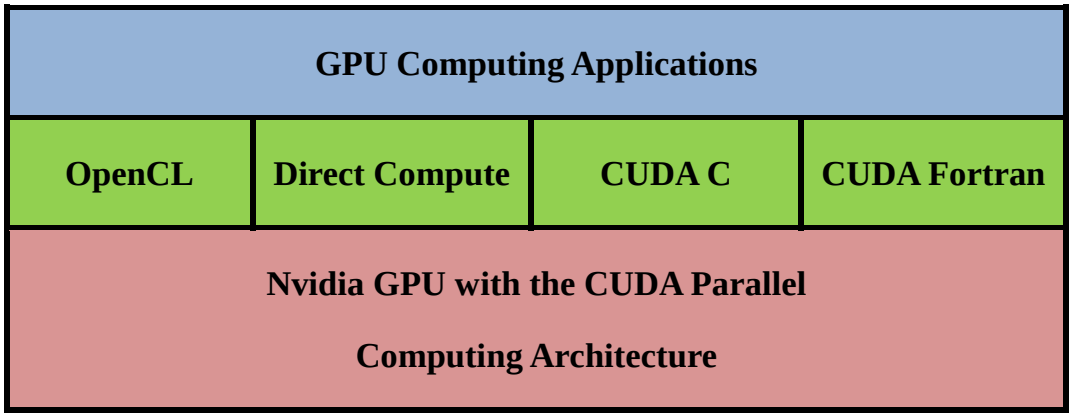


圖 2.4-3 CUDA 所提供 API 的支援度(由【11】重新繪製)

2.4.1 CUDA Kernel 編譯方式概述

在CUDA 的架構下將程式劃分為兩大主要部分，分別為Host 與Device。Host端指的是在CPU 上執行的程式部分，而Device 端則是經由圖形晶片上的GPU 處理的程式部分。Device 端裡面的程式，CUDA 將其定義名為kernel，通常會在函式之前用 __global__ 命名宣告，並且使用一個<<<...>>>符號 (稱為 execution configuration syntax) 用來指派CUDA threads 的數量，而這些數量即代表同時要產生多少threads 來平行執行此kernel 程式。每個執行kernel 程式的thread 都會藉由threadIdx 這個變數得到一個獨特的thread ID，而同一份程式碼就可以用不同的thread ID 來判斷，取得不同的資料加以平行運算(如圖2.4-4)【4】。

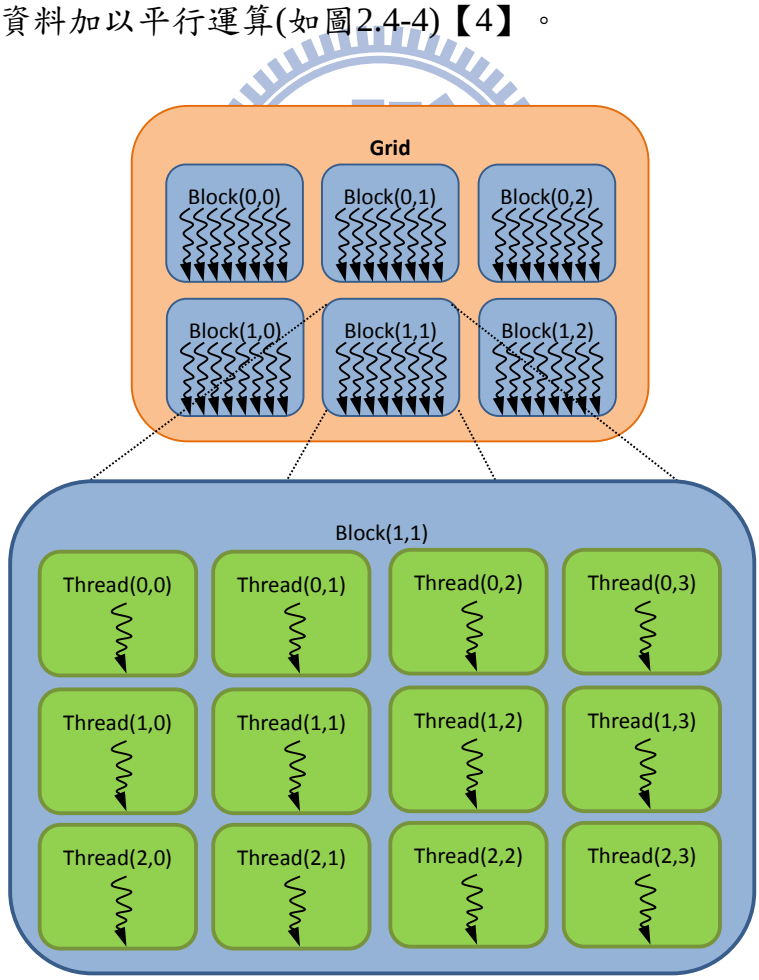


圖 2.4-4 Grid 中的 Thread Blocks (由【11】重新繪製)

2.4.2 Thread 階層

在CUDA 架構下，thread 是GPU 顯示晶片執行的最小單位，GPU的thread與CPU不同，CPU每多一個Thread需花上一定之工作時間延遲，而GPU每個thread 有自己的register 和local memory，而由thread 所組成的最小單位稱為block，同一個block 當中的thread 彼此擁有一份共享的shared memory。CUDA 將threadIdx 宣告為一個三維的向量變數，可以按照不同的需求建立一維、二維、及三維的thread index 和thread block，例如向量運算、矩陣運算或是三度空間中的任何運算。

每個block 中的thread 數量是受到限制的，在同一個block 當中的thread允許存取一塊共用的記憶體，所以同block 中的thread 彼此的同步計算速度較高，而不同block 當中的thread 因為無法共同存取一樣的共享記憶體，彼此的同步程度就沒那麼高，目前GPU 的技術可以允許一個 block 當中最多可包含512或768個thread。儘管block 當中的thread 數量受到限制，我們仍然可以把執行相同程式的block 集合起來組成一維或是二維空間的grid，所以在編寫程式時可以不用考慮繪圖晶片實際上可以同時執行的thread 數目限制。類似thread 組成block 的方式，CUDA 同樣運用block 組成grid 的概念，每個grid 中的block 會藉由blockIdx這個變數得到一個一維或是二維空間的index，而這個block 的維度(dimension)是由blickDim 此變數指派。不同的thread blocks 是可以個別獨立執行不受影響的，CUDA 允許這些blocks 具有任意的執行順序，不論是平行或是循序執行皆可，並且支援不同數量的GPU cores(如圖2.4-5)。

透過shared memory，thread 除了可以共享資料，也可以執行運算結果的同步化。對於同步化(synchronization)的能力，CUDA 允許程式開發者在kernel當中指定同步化的時間點。此動作是由呼叫 __syncthreads() 函式達成，此函式會確保所有執行此kernel 程式的thread 都能達成同步，其他

thread 會等待直到同步完成，同步化的使用會降低程式效率，為了確保數值正確此函數是必需的【4】。

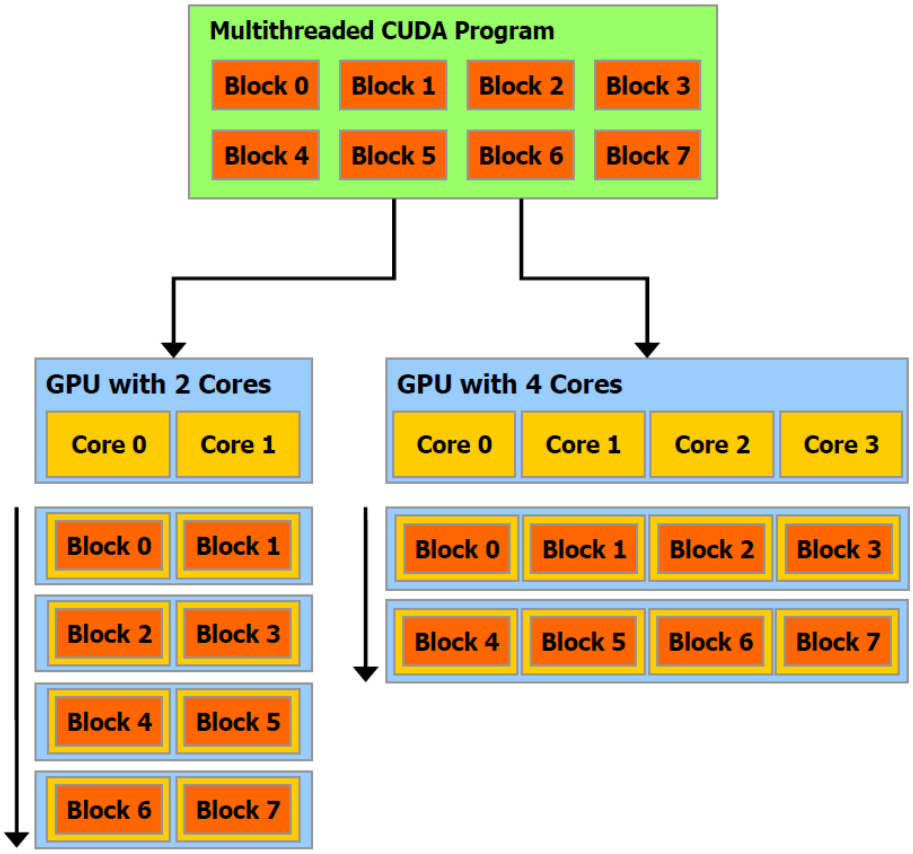


圖 2.4-5 不同 core 數目的 GPU 由 block 指派工作 【11】

2.4.3 異質編程

CUDA programming model 可以讓thread 程式以CPU 跟GPU 混合方式執行，Device 端的kernel 程式在GPU 上運作，而Host 端的C program則交給CPU 處理。所以CPU執行一般的C言語言與GPU記憶體的準備工作讓GPU 的程式處理可以並行，而Host 跟Device 可以各自去DRAM當中取出個別獨立的部分來使用，稱為Host Memory 及Device Memory(如圖2.4-6)。

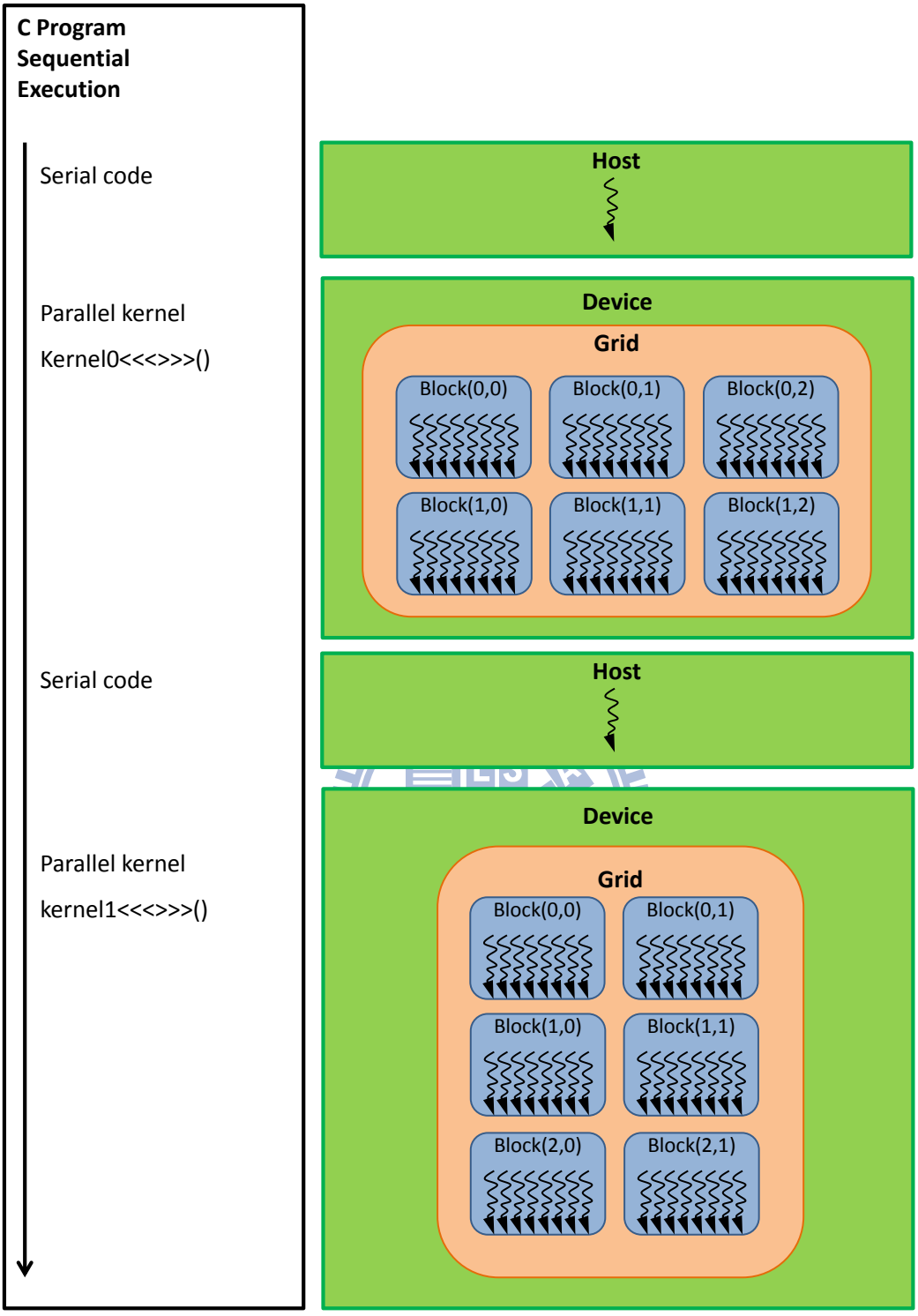


圖 2.4-6 Host 與 Device 在硬體與軟體上呼叫方式(由【11】重新繪製)

2.4.4 GPU 硬體架構

CUDA 圖形處理器的硬體架構是由一種可擴充陣列型的多工串流微處理器(Streaming Multiprocessor, 簡稱SM)所組成，而這些SM 是由GPU最基本的處理群組單元SP(Streaming Processor)所構成。我們可以按照下列基本流程來分工【11】：將Grid 對應到GPU 來執行，而程式中的一個block被分配到一個SM 上面，block 當中的許多thread 則被分配到這個SM 當中的每一個SP 上面分別執行。以Nvidia GeForce 8 為例每一個SM擁有16個SP(CUDA Core)，而這16個SP之間彼此共享一個傳輸極快的cache記憶體容量僅16KB(如圖2.4-7)。

程式編輯時會將block 當中的thread 分組，每32 個parallel threads 分為一組，稱為一個warp，同一個warp 裡面的thread 可以用不同的資料執行相同的指令，設計warp 的目的在於讓一個SM 一次執行一個block 裡面的一個warp，亦即32 個threads。一個warp 一次執行一個指令，所以可預期當warp 中的32 個thread正確同步時，會有最好的效能。當SM 遇到執行中的warp 等待其他資料時，SM具有切換到其他warp 做運算的能力，避免等待資源所浪費的時間，CUDA 透過warp的切換來隱藏thread 的延遲、等待，達到大量平行化的目的。

(圖2.3-7) 以實際 Nvidia GeForce 8 作為例子。GeForce 8 系列的GPU含有128 個thread processors(又稱為stream processors)，這個單位在圖形處理上通常被稱為 “programmable pixel shaders”。每個thread processor有一個single-precision FPU專門針對sin、cos…等常用函數透過硬體加速不避使用處理器計算與1024 個最快速的暫存器，大小為32bits。每八個thread processors 共同享有一個16KB 的shared local memory，可以用來作平行程式的資料處理跟交換；至於Thread-execution manager 會自動將thread 分散給處理器來執行，並不需要由程式設計師額外撰寫threaded code。最大平

行thread的數目是12288，至於Nvidia GPU 擁有128 個thread processors 的型號有：GeForrce 8800FTX, GeForce 8800 Ultra, GeForce 8800 GTS 512MB, Quadra FX5600 和Tesla C870, D870, S870 等 【11】。

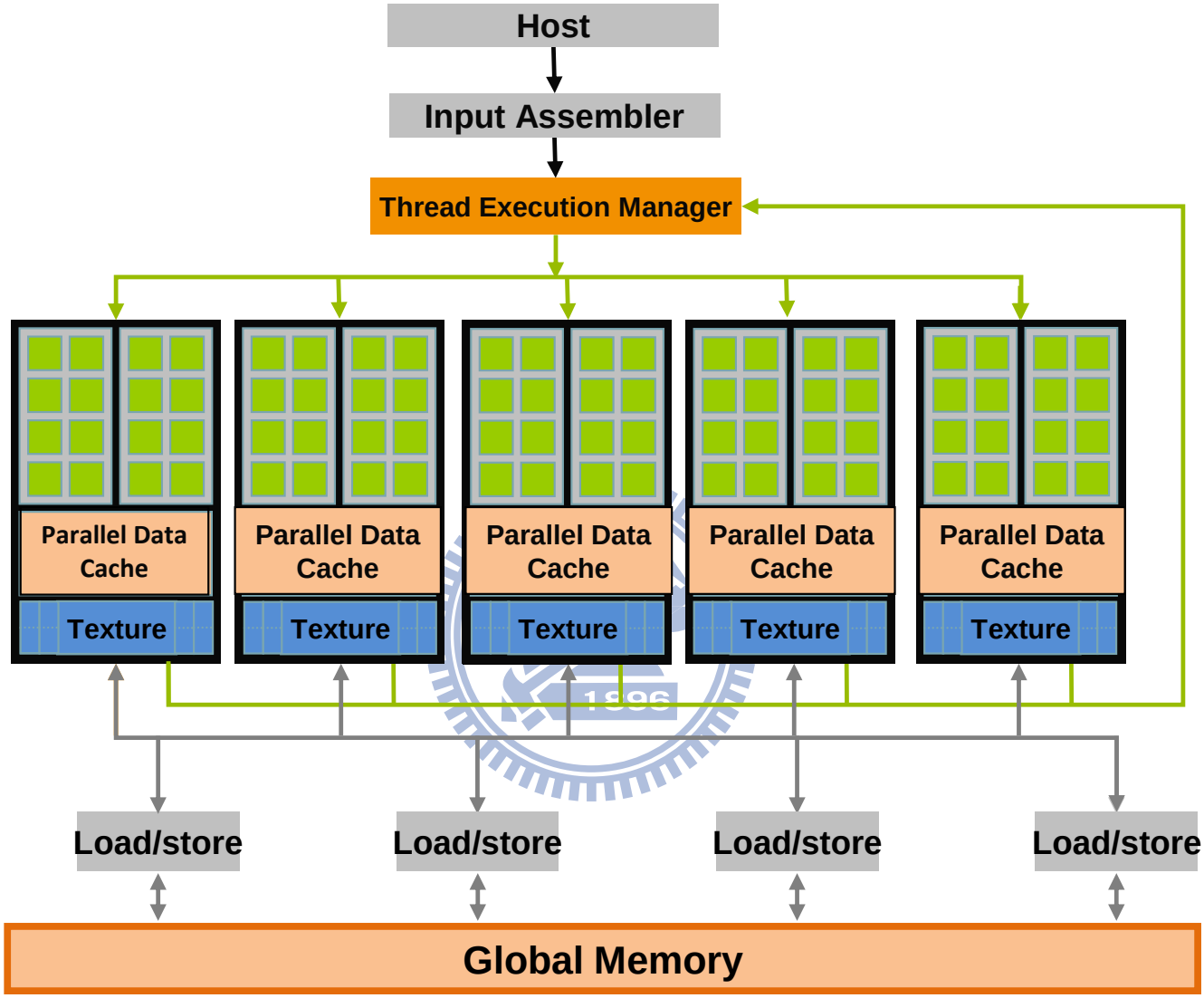


圖 2.4-7 簡化 Nvidia GeForce 8 圖形處理器架構 (由【11】簡化繪製)

2.4.5 記憶體與 Thread 關係

先前提到每個thread 有獨立的private local memory，而thread block彼此之

間共享一塊shared memory，同一block 之中的thread 彼此可見，並且有相同的生命週期；而對於全部的thread 而言，另外提供一個全域的global memory 空間。此外thread 尚可以讀取另外兩種唯讀記憶體空間：constant memory 和 texture memory，這三種記憶體配置分別有不同的用途。

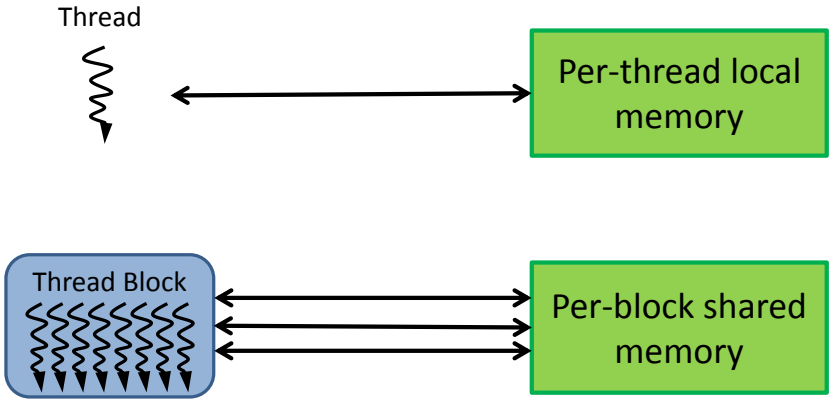


圖 2.4-8 Thread 之間的記憶體傳遞方式(由【11】重新繪製)

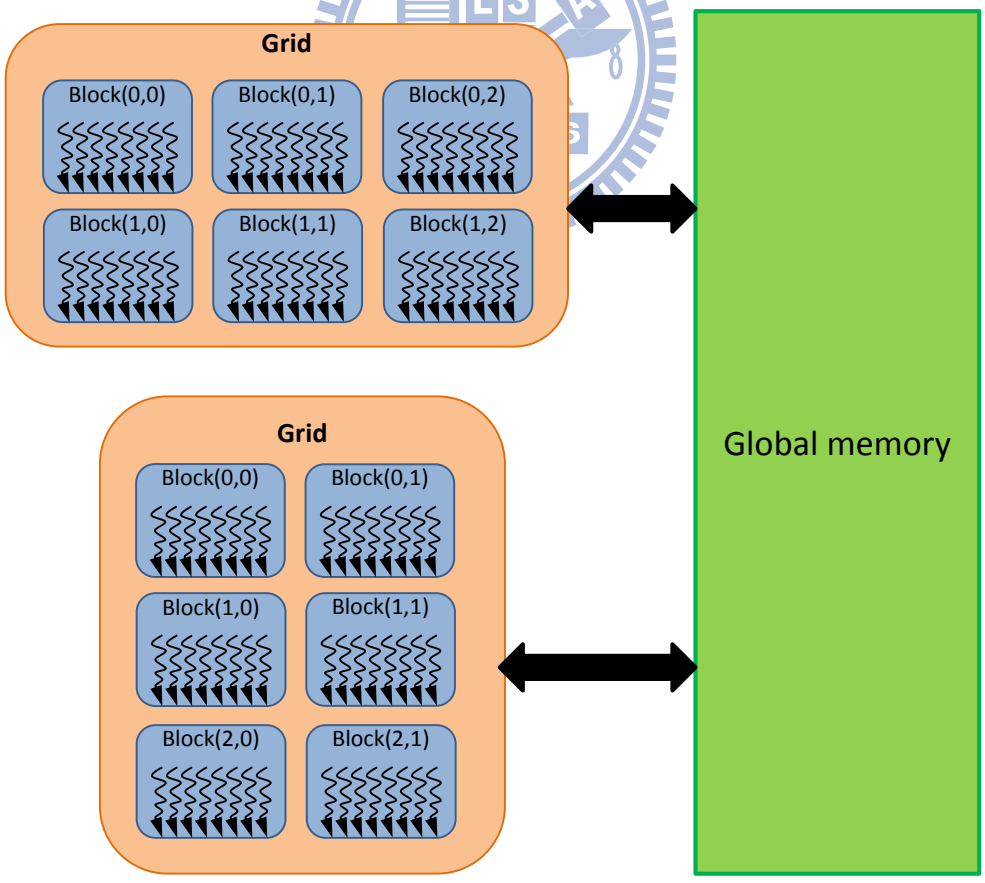


圖 2.4-9 Thread 之間的記憶體傳遞方式(由【11】重新繪製)

2.4.6 GPU 記憶體種類介紹

CUDA programming model 將系統劃分為host 與device 兩部份，各自有獨立配置的記憶體空間。而kernels 只對device memory 有存取控制及操作的權限，device memory 有兩種配置方式：線性記憶體配置(linear memory)或是陣列配置(CUDA arrays)。

Linear memory 的特點之一是可以用指標存取，在device 當中可以有32-bit或是40-bit 的存址方式，分別對應capability 1.x 及2.0。CUDA 提供了配置(cudaMalloc)、釋放(cudaFree)、以及傳輸(cudaMemcpy)等記憶體內容操作的函式，包括host memory 與device memory 之間的資料傳送也是由相關函式來運作【4】。

2.4.5.2 Register

CUDA 的記憶體架構有許多層級，其中 register 就存在 thread 當中。register 具有所有記憶體當中存取速度最快的特點，在 thread 當中大部分的局部變數會預設使用 register。然而 register 的數量最小最有限，每個 thread 約可以使用至多 7 個變數，所以必須搭配其他 memory 的輔助

2.4.5.3 Shared Memory

Shared memory 的共用範圍是整個區塊(block)，使用 __shared__ 標籤宣告，具有 block 之內 thread 間資料快速交換的特性，在使用上的用途很廣。可以將會多次重複使用的資料放入 shared memory，當作一種 cache；此外也可以存放共用變數，減少暫存器的使用量；亦可作為資料順序動態重整單元，避免區域記憶體的介入；此外也可作為 global memory 的合併讀取緩

衝。使用時必須配合__syncthreads()對 thread 進行同步化，避免 thread 不同步而讀取尚未寫入的記憶體位址，確保資料正確無誤。Shared memory 的效能僅次於 register，當我們要執行最佳化的改善時，shared memory 是一個可最佳化的重點項目。

2.4.5.4 Local Memory

Local memory 通常保留給某些automatic variables：例如會佔用到許多暫存器空間的大型陣列結構，或是某些變數讓kernels 使用了比可用暫存器還要多的額外空間時，這種情形通常稱為register spilling。當register 不夠時，編譯器會自動將資料切換到local memory，有點類似作業系統的page swap，這個動作會降低系統效能，然而為了加大區塊中thread 數目，我們必須透過限制每個thread 配置的最大register 使用量，需要local memory 的空間配置才可達成。由於local memory 在device memory 當中，所以存取上具有高延遲與低頻寬的特性，跟global memory 的存取速度一樣都較慢。

2.4.5.5 Global Memory

Global Memory 在 device memory 中，所以存取速度跟 local memory 一樣都是屬於較慢的，除了使用 __device__ 標籤宣告，透過 CUDA 相關 API 直接配置的記憶體空間也屬於 global memory。所有的 thread 都可以對 global memory 存取，定址方式分別為 32-, 64-, 或 128-byte，由於讀取寫入皆不經過快取，所以當一個 warp 執行指令存取 global memory 時，warp 可以按照每一個 thread 存取的 word 大小來分配 warp 當中的 memory address，藉此合併 memory access 來提升效能，這步驟稱為合併讀取(coalesced read)。

2.4.5.6 Constant Memory

Constant Memory 存放整個程式共用的常數，具有全域性質，只有在初次使用時需要載入，之後的存取效能就和shared memory 一樣快，對kernel 而言是唯讀的，只能在檔案中使用初始值的方式設定，可以由__constant__ 標籤宣告；或是在host 當中透過API 進行存取的動作。

2.4.5.7 Texture Memory

Texture memory 是獨立於其他記憶體的唯讀存取機制，使用上要透過API。雖然texture memory 本身是global memory 的一部份，因為本身有快取作為緩衝，用來加速和過濾(filter)。所以讀取時不需要做合併的動作，效能會比尚未合併讀取的global memory 快上許多。主要的應用在繪圖需求，CUDA 內建1D,2D,及3D 的texture memory，目前對2D 的空間區域性運用配置有最佳化的效能，所以可預期相同warp 當中的threads 以2D 方式存取texture memory 可以達到最好的效果。

2.5 Linpack 程式集

在評估電腦的浮點運算效益的效能評估程式集中，就以Jack Dongarra所寫的Linpack最為大家所熟悉【4】，此程式集原本是由Argonne 國家實驗室負責維護，現在則由田納西大學負責。

Linpack程式集是透過許多的副程式來求解稠密線性方程組（A Dense System of Linear Equations），再將全部的浮點運算數目除以實際計算所需的時間，得出的結果即為平均的MFLOPS ，所採用的方法則為高斯消去法。

Linpack效能評估報告是公開的， 任何人只要送個以電子信件方式， send performance from benchmark就可以收到由Linpack寄回的最新的報告。

根據問題尺度大小、數值精確度及某些規定，Linpack有幾個不同的版本，其中比較常被人引用的兩種規範為Linpack 100x100 及Linpack 1000x1000。一般而言，工作站廠商最喜歡引用Linpack 100x100的雙精度(Double Precision) 結果，而大的向量或平行電腦廠商則喜歡引用Linpack 1000x1000的結果。

在FORTRAN Linpack 100x100效能評估程式集中，主要的計算核心是一個叫daxpy的副程式。Daxpy是將一個向量乘上一個常數後，再將之與另一個向量相加，再將所得到的結果存回到原來的向量，其程式如下表：

表 2.5-1 Linpack 效能評估方式

Algorithm Daxpy(n,dy,dx)	
1：	do 30 i = 1,n
2：	dy(i) = dy(i) + da*dx(i)
3：	30 continue

從上面的程式可以看出，此迴圈 (Loop) 一共需與加二個浮點運算，以及包含二個取（Load）與一個存（Store）總共三個到記憶體存取資料的動作。基本上， Linpack 100x100效能評估程式集的主要目的，是在測試電腦系統的浮點運算效益及記憶系統的功能。

Linpack 100x100有一個非常嚴格的規定，即對原始程式不可有任何人為的變更， 否則得出的結果就不予採納。因此，Linpack 100×100的結果好壞，完全依賴系統的編譯器（Compiler）的功能。如果系統編譯器的最佳化做得好，如可自動做Loop Unrolling的動作，則所得出的結果往往可比不能自動做Loop Unrolling的系統快許多。

對許多大型向量或平行電腦而言，Linpack 100×100的問題尺度太小，無法發揮其計算能力，此時Linpack 1000×1000因其所處理的問題尺度和Linpack 100×100相較，大了許多，也就較能突顯出系統的計算能力，故大型向量或平行電腦的廠商喜歡引用Linpack 1000×1000的結果【37】。

2.6 ANSYS 求解器比較與介紹

ANSYS 在各種有限元素當中，擁有較完整的求解器，同時也擁有較為完整的解算器。每種解算器適用在各種不同的問題上，在計算效率方面AMSYS 擁有較佳的解算性能。在多種有限元素軟體當中 ANSYS 一直是大型的電腦所時常使用的軟體之一，具備了不錯的平行運算效能。表 2.6-1 以 ANSYS 官方所公佈的各種解算器適用性做整理。

表 2.6-1 ANSYS 求解器總整理與比較【38】

解算器	適合自由度	所需記憶體	硬碟空間
Sparse Direct Solver (直接法,支援多核心共享記憶體架構)	10^4 ~ $5*10^6$ DOFs	1 GB/MDOF)	10 GB/MDOF
	在非線性當中求解計算快速試必須的，而線性系統分析當中疊代法的收斂速度是緩慢的由其事求解病態矩陣，Sparse Direct Solver 為最常用且快速的求解器。		
PCG Solver (疊代法)	$5*10^4$to 10^7+ DOFs	0.3 GB/MDOF	0.5 GB/MDOF
	相對於 Sparse Direct 可減少大量的硬碟空間減少 I/O 傳輸。應用於大型模型與精密網格之最佳的求解器。在 ANSYS 中最強大的疊代求解器		
JCG Solver (疊代法)	$5*10^4$ to 10^7+ DOFs	0.5 GB/MDOF	0.5 GB/MDOF
	應用於較為單一領域問題(熱、磁學、聲波、多種物理)。使用較為快速、而簡單的疊代方法且記憶體需求量低。		

解算器	適合自由度	所需記憶體	硬碟空間
ICCG Solver (疊代法)	$5*10^4$ to 10^7+ DOFs	1.5 GB/MDOF	0.5 GB/MDOF
	比起 JCG 有梗複雜的收斂方式。在困難的問題求解上比起 JCG 更不容易失敗。		
QMR Solver (直接法)	$5*10^4$ to 10^7+ DOFs	1.5 GB/MDOF	0.5 GB/MDOF
	適用在高頻電磁學上		
Frontal Solver (直接法)	Under 50,000 DOFs	Less than 0.5 GB/MDOF	10 GB/MDOF
	為 Sparse Direct Solver 之前身。需要記憶體量較少，但計算速度較為緩慢，在較小問題上有良好的速度表現，適合用於小型模組。		
DPCG Solver (疊代法,支援叢集電腦架構)	$5*10^4$to 10^8+ DOFs	1.5-2.0 GB/MDOF in total*	0.5 GB/MDOF
	與 PCG 相同，支援分散式運算(使用上需而付費額外認證)		
DJCG Solver (疊代法,支援叢集架構)	$5*10^4$ to 10^7+ DOFs	0.5 GB/MDOF	0.5 GB/MDOF
	與 JCG 相同，支援分散式運算(使用上需而付費額外認證		
AMG Solver (疊代法,支援多核心共享架構)	$5*10^4$ to 10^6+ DOFs	1.5-3.0 GB/MDOF	0.5 GB/MDOF
	有良好的共享記憶體性能。對於病態矩陣有較佳的解算性能，但計算較微軟慢，支援用多核心共享架構。(使用上需而付費額外認證)		
DSPARSE Solver (直接法,支援叢集電腦架構)	10^4 to $5*10^5$ DOFs.	1.5 GB/MDOF on master machine, 1.0 GB/MDOF on slave machines	10 GB/MDOF
	類似 sparse solver 但在執行上使用分散式運算 (使用上需而付費額外認證) 可使用到 16 顆處理器。		

第三章 研究方法

3.1 常用線性求解演算法種類

一般聯立方程組為形式為：

$$\begin{cases} a_{00}x_0 + a_{01}x_1 + \cdots + a_{0,n-1}x_{n-1} = b_0 \\ a_{10}x_0 + a_{11}x_1 + \cdots + a_{1,n-1}x_{n-1} = b_{01} \\ \vdots \quad \vdots \quad \vdots \quad \vdots \\ a_{m-1,0}x_0 + a_{m-1,1}x_1 + \cdots + a_{m-1,n-1}x_{n-1} = b_{m-1} \end{cases}$$

方程組可寫成矩陣形式為$[a][x] = [b]$。其中

$$[A] = \begin{bmatrix} a_{00} & \cdots & a_{0,n-1} \\ \vdots & \ddots & \vdots \\ a_{m-1,0} & \cdots & a_{m-1,n-1} \end{bmatrix}, [X] = \begin{bmatrix} x_0 \\ \vdots \\ x_{n-1} \end{bmatrix}, [B] = \begin{bmatrix} b_0 \\ \vdots \\ b_{m-1} \end{bmatrix}$$

如果$[A]$非奇異，即$[A]$的行列式$\det(A) \neq 0$，則方程組有唯一解。求解線性代數演算法主要有兩種種類：直接法與疊代法。直接法的優點可預先知道計算量推估出所需要的時間，而直接演算法又可分為直接消去法與矩陣分解法兩種。疊代法求解法以逼近的方式逐步收斂至符合精度之真實解，疊代分解法中最重要的是疊代方法是否收斂以及收斂的快慢這兩個問題。

3.1.1 直接消去法：Gauss , Gauss –Jordan

消去法為直接法的一種，基本模式是透過消元方式減少方程中的未知數目，最終使每個方程只含有一個求和數，而得到所需要的解。但由於實際計算過程中總存在的誤差，誤差累積使得消去法得到的結果往往並不精

確，直接消去法存在數值穩定性問題。

Gauss 消去法為直接法中最常見的一種線性求解演算法，主要步驟包括下三角消元和後向帶入兩個主要過程。而 Gauss –Jordan 將聯立方程組[A]矩陣物轉變為對角矩陣，消元同時[B]必須跟著變動，可省略了後向帶入的過程，效率會比 Gauss 消去法來的差，適合用於[B]同時擁有多組省下大量後向代入時間【13】。

3.1.2 矩陣分解法 ：LU, Cholesky, QR, SVD

矩陣分解也是直接法的一種。這種方法先將係數矩陣依造某種形式進行分解，然而就可以很方便的快速的求出解向量。比較特別的是 QR 方法，透過簡單的數學演算，可將它應用到求解最小二成問題上來，這是一般消去法所不能夠做到的【14】。

分解法中最常見的為 LU 分解，將[*A*]分解[*L*][*U*]，然而能進行遞推求出解向量，Cholesky 為 LU 中的一種特例，由 LU 推倒而來針對正定矩陣的一種快速求解方法，只需分解出[L]就可利用對稱性質進行求解，較為節省計算量。奇異值分解 SVD 是另一種正交矩陣分解法；SVD 是最可靠的分解法，但是它比 QR 分解法要花上近十倍的計算時間。將[A]分解為[U,S,V]，其中 U 和 V 代表二個相互正交矩陣，而 S 代表一對角矩陣。 和 QR 分解法相同者， 原矩陣 A 不必為正方矩陣。 使用 SVD 分解法的用途是解最小平方誤差法和數據壓縮。

3.1.3 疊代求解法：Gauss-Seidel, Jacobi , SOR, PCG

在超大型線性矩陣當中會消耗大量的求解時間，這時選擇疊代求解法就是個良好的選擇，求解初步需猜測良好的初始值，透過不斷的修正希望能收斂並且快速得到正確的答案，因此衍生出許多的優化方法，以及最重要的何時停止收斂【13】。疊代法在求解病態矩陣上，有著比直接法更好的精度求解，可以透過時間換取病態矩陣令人詬病的精度問題。

Jacobi 是簡單而固定的方法。但它不一定是實際好用的方法，但對於疊代法是方法基礎是值得去學習的，Jacobi 有許多不錯的功能，不需額外使用多於記憶體做分解，對於平行處理相當適合，但收斂過程是相當緩慢的【15】。Gauss-Seidel 收斂比起 Jacobi 有著更快速的收斂方法，也是最多疊代法使用上的選擇，有著擁有 Jacobi 容易收斂的優點。SOR 是 Gauss-Seidel 的一種演進，增加了一個鬆弛係數加快了收斂效果，也因鬆弛係數可能導致無法收斂。 Jacobi 與 SOR 的雖然較快速但平行度皆不如 Jacobi。

3.2 直接法平行概念

直接法的迴圈方式彼此都有些類似，由表 3.2-1 可知，直接法演算過程中外層迴圈，每一次之迴圈計算是利用上一次迴圈之數值，彼此有關聯性為相依非獨立，所以無法在最外層迴圈進行平行，而必須在第二層迴圈進行平行，這也就是直接法的平行程度低的問題所在，每次迴圈必須不斷的打開與關閉平行造成延遲，假若在較小型計算使用叢集架構更是可能早造成網路延遲導致效率倒退，以往的研究大部分集中於疊代法，但由於直接法的效能與精度都是非常吸引人值得研究，也更適合一般的桌上型電腦使用。

OpenMP 的共享記憶體架構反應傳輸速率快，上述此種問題將較不明顯，隨著案例增大就能有明顯的效能產生。而 CUDA 是一般是透過北橋傳輸給 GPU，CUP 與 GPU 記憶體各自獨立，傳輸上必須有一定的耗損時間，傳輸延遲時間介於 MPI 與 OpenMP 之間，雖然比起 MPI 傳輸快了許多，但仍然會有延遲的問題，適用於大型案例才能帶來較好的加速加算效果。

表 3.2-1 直接法平行方式

Algorithm
1：　　**for** i = 1 to SIZE **do**
2：　　(相依)
3：　　**for** k = 1 to i **do**
4：　　(平行價值低)
5：　　**end for**
6：　　**#pragma omp for**
7：　　**for** j = i to SIZE **do**
8：　　**for** k = 0 to i **do**
9：　　(可平行)
10：　　**end for**
11：　　**end for**
12：　　**end for**

3.3 直接求解演算法比較

直接求解演算法相較於疊代求解法，在一般情況下直接法求解所需之計算時間都是較少的，但是在超大型矩陣使用疊代法會得到較好的效率，往往大型矩陣都已不敷一般桌上型電腦所使用之記憶體，所以本研究只討論適用性較好而較難以平行的的直接求解演算法。將以矩陣分解法與直接求解分別進行平行化做平行效能測試，將會以誤差精度與運算效率挑選一組較適合做為平行演算法的優化的基礎。其中排除執行效率的低且可解算病態矩陣的 SVD 演算法

3.3.1 矩陣分解演算法進行平行化比較

在 2008 年 V. Volkov 與 J.W. Demmel 就已經將矩陣分解法 Cholesky, LU, QR 分別進行平行化【16】，以 level 3 basic linear algebra subprograms (BLAS3)【17】塊狀切割的方式進行 GPU 平行運算利用此一動態負載平衡方法可得到較有效率的 BLAS 3 程式庫，對於矩陣分解法平行化有系統性的整理。目前已 LAPCK 函數庫開放程式碼提供測試，[A]為 n*n 大小之矩陣，表 3.3-1 使用 GPU：NVIDIA GTX260 做效能測試：

表 3.3-1 Cholesky, LU, QR 演算法進行單精度 GPU 平行【16】

Algorithm	**Cholesky**		**LU**		**QR**	
n	Gflops	Error	Gflops	error	Gflops	Error
1000	14.8	0.8	4.9	37.4	54.3	8.7
2000	101.1	1.0	97.6	60.9	123.0	12.6
4000	111.1	0.9	101.2	106.7	168.9	16.8
6000	172.1	1.4	173.1	146.3	196.9	20.6
8000	190.2	1.6	180.6	193.3	207.5	22.2
10000	199.4	1.6	194.2	225.6	215.9	27.7

由表 3.3-1 可得到下列結論，矩陣分解法在 n=10000 的矩陣大小實數運算量效率上皆大致相同，而在較小的 n=1000 的矩陣大小效能皆相當低落尤其在 LU 分解演算法，幾乎與一般 CPU 效能接近。而在精度方面，Cholesky 分解法誤差最小，擁有最佳的精度求解。如今本研究為正定矩陣，選定以精度最佳與效能中等的 Choleshy 分解法利用進行與直接消法做效能及平行化比較，

3.3.2 Cholesky 與直接消去法進行平行化比較

Gauss-Jordan 與 Gauss 消去法本質並無太大區別最大的不同在於，Gauss 是將係數矩陣轉換為上三角矩陣，透過後向而代入得到方程組未知數之數值。而 Gauss-Jordan 是將係數矩陣消元成為對角線矩陣其餘上、下三角皆為 0，則方程組解不需要進行向後代入。

將矩陣變換成對角矩陣要比起 Gauss 換成上三角矩陣多約 1/6 的工作量，不過 Gauss-Jordan 本身不需透過後向代入得到未知數之數值，消元上、下三角讓 Gauss-Jordan 整體花費時間比 Gauss 來的多一些，但 Gauss-Jordan 本身每次迴圈計算量皆相同，對於平行切割處理用於 thread 分配工作量是相對有利的。

所以我們將 Gauss-Jordan 與 Cholesky 使用單核心系統在各種矩陣大小下測試需要時間與產生之誤差如下表 3.4-1：

表 3.3-2 Gauss-Jordan 與 Cholesky 效能比較

		Gauss-Jordan		**Cholesky**	
n	memory	time	error	Time	error
20000	1600MB	8.45hour	0.28%	2.25hour	0.28%
10000	400MB	1.06hour	0.09%	0.28hour	0.08%
5000	100MB	0.13hour	0.04%	0.035hour	0.03%
2500	25MB	0.017hour	0.01%	0.004hour	0.01%

上由表之資訊可知 Cholesky 比 Gauss-Jordan 少了一倍實數運算量，但顯現出來的效能大約節省時間約是 3.5 倍(如表：3.3-2)，Cholesky 在相同矩陣大小下所需要之求解算時間優於 Gauss-Jordan，且兩種演算法精度皆差異不大。

接著將 Cholesky 與 Gauss-Jordan 在多核心共享記憶體架構下，使用 OpenMP 平行化做效能測試，矩陣大小為 n=5000 使用硬體設備 CPU：X4-915 2.6Hz。

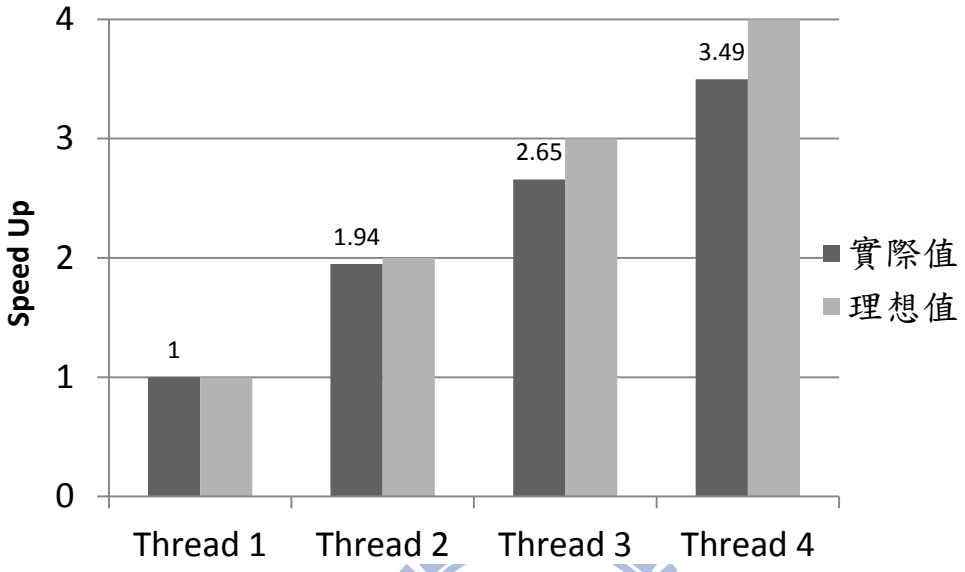


圖 3.3-2Gauss-Jordan 使用 OpenMP 平行成果

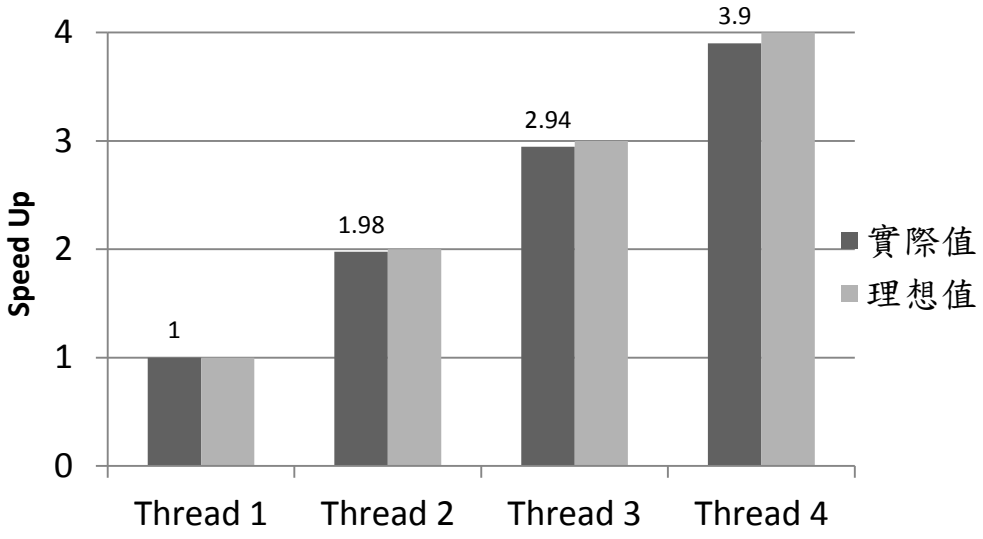


圖 3.4-3Cholesky 使用 OpenMP 平行成果

由圖 3.4-2 與 3.4-3 可以看出 Cholesky 與 Gauss-Jordan 分別在四核心 OpenMP 平行處理下，Cholesky 可達到 3.9 倍效能趨近於理想值，Gauss 約能達到 3.49 倍，Cholesky 效能表現明顯比較好，Cholesky 只有斜對角開

根號不可平行，所以效率較高。而 Gauss-Jordan 主要造成效率較低的原因有兩個，第一個原因是本身迴圈內多了『if』的邏輯判別，邏輯判別在平行運算會造成效能的拖累，在程式上多重迴圈內部是需要盡量避免使用的，另一個原因是不可平行的部分也隨著矩陣大小跟著變龐大，造成計算時間在不可平行部分將拖累整體程式。

由表 3.3-1 與圖 3.3.3 的結果可以得出結論， Cholesky 不論在演算法及平行運算的效能皆優於 Gauss-Jordan，而在有限元素所建構之線性系統矩陣是具有對稱性質符合套用在 Cholesky 演算法之限制條件。基於上述這些理由，選定以 Cholesky 作為演算法改良的基礎。

3.4 Cholesky 分解法介紹

Cholesky 演算法是矩陣分解當中的一種特例，與 LU 分解演算法相似，限制使用於對稱且正定的方程組，只需求出下三角分解矩陣並解利用對稱性質方可達到 LU 分解的效果，可省下比起 LU 分解一倍的實數運算量。

對於方程組來說$[A][X]=[B]$來說，則分解為$[r][r]^T[x]=[B]$。即$[A]=[r][r]^T$其中$[r]$為下三角，$[r]^T$為上三角

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ a_{31} & a_{32} & \cdots & a_{3n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix} = \begin{pmatrix} r_{11} & 0 & \cdots & 0 \\ r_{21} & r_{22} & \cdots & 0 \\ r_{31} & r_{32} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ r_{n1} & r_{n2} & \cdots & 0 \end{pmatrix} \begin{pmatrix} r_{11} & r_{12} & \cdots & r_{13} \\ 0 & r_{22} & \cdots & r_{23} \\ 0 & 0 & \cdots & r_{33} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & r_{nn} \end{pmatrix}$$

Cholesky 的好處是只需要求出上三角或下三角其中之一，方可以求解，

因對稱矩陣中分解出來的$[r]$會與$[\mathrm{r}]^T$成轉制，之後只需要將$[b]$將$[r]$與$[\mathrm{r}]^T$進行簡單的遞推就可以快速的求出解向量。

遞推方式：

首先令$[y]=[a]^T[x]$，解方程組$[a][y]=[b]$就可得到$[y]$，然後再求解$[\mathrm{a}]^T[x]=[y]$，就可以得到解$[x]$。遞推為雙層迴圈並不耗費時間，所以並非本研究之重點，主要著重於下列運算量大的分解演算法。

$[A]=[a][a]^T$分解演算法：

第一步：$a_{ii} \coloneqq \sqrt{a_{ii}}$

第二步：$a_{ji} \coloneqq a_{ji} - \sum_{k=1}^{i-1} a_{ki} * a_{kj}$

第三步：$a_{ji} \coloneqq a_{ji}/a_{ii}$

求解出下三角$[r]$是 Choleshy 最耗時的一個部分，與一般直接法相同都為三層迴圈，因此本研究重心針對這最耗時的這部分著手進行平行化。

3.5 帶狀矩陣壓縮記憶體方式

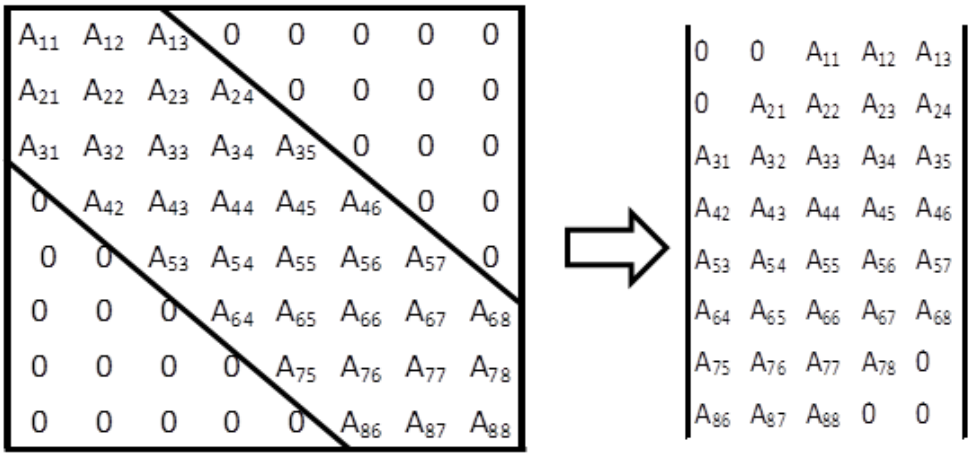


圖 3.5-1 寬帶矩陣壓縮方式

由上圖 3.5-1 左為帶狀矩陣;寬帶(k)取最大值為：2，右為壓縮後之矩陣模式。稀疏矩陣所需的迴圈次數為：(n^3)，考慮帶狀矩陣後將可降低迴圈次數為：$(2*k+1)^2 \times n$，可省下大量迴圈次數以至於大幅的節省計算時間，以有線元素方面為例子，將物件模型自由度編號編輯的適當，方可產生出帶狀矩陣，在 3D 結構物體或不規則物體下，編輯出具有最佳的寬帶在程式寫作上有一定的難度，寬帶縮小計算量亦隨之減少方可達到線性聯立方程求解目的，自由度編輯則是另一方面許多人研究的課題。

而在記憶體方面，稀疏矩陣所需記憶體為：$(n)^2$，透過矩陣壓縮技術將寬帶內之帶狀形矩陣保留，剃除其於上、下三角之空矩陣，可將記憶體壓縮成(如圖 3.5-1 右方)為一個狹窄長方形矩陣，而壓縮矩陣所需記憶體降低為：$n \times (2*k+1)$，經由壓縮矩陣後可將原本稀疏龐大的矩陣量，大幅度的壓縮解決記憶經常不足的問題，當電腦記憶體不足時會電腦將自動把不足的記憶體，轉交給硬碟暫存，由於硬碟傳輸極度緩慢，容易會造成速度延遲。令不解的是目前大部分軟體，在直接法並做壓縮矩陣較少，只考慮帶狀矩陣，大部分系線分析軟體做矩陣壓縮主要針對疊代法。

3.5.1 Cholesky 矩陣壓縮方式(PBTRF 格式)

如圖 3.5-2 左下採用 LAPCK 常規 PBRT 壓縮格式，讓本論文程式編譯格式，可以方便的與其它線性系統並使用矩陣壓縮之演算法彼此資料格式相容，方便本程式以物件方式嵌入軟體，應用在符合線性求解系統之各種軟體領域當中。

考慮帶狀矩陣使用 Cholesky 所需的迴圈次數降為：$(k+1)^2 \times n$，Cholesky 壓縮矩陣所需記憶體為：$(k+1) \times n$ ，利用矩陣對稱性質，可將帶狀矩陣迴圈次數再度減少，同時所需的記憶體也可減半，比起一般壓縮方式更省時更省硬體成本。

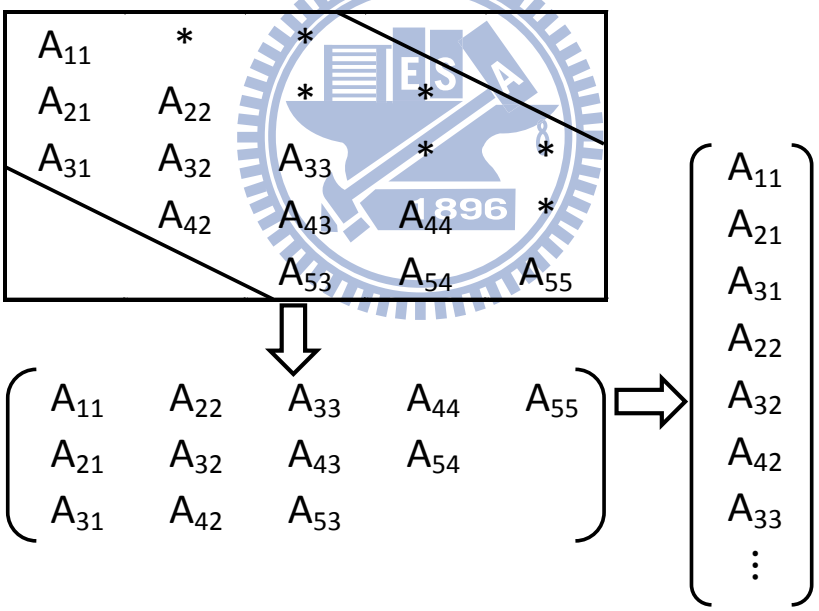


圖 3.5-2 Cholesky 矩陣壓縮模式【18】

在程式編輯上，本研究將一律採用一維方式做宣告，將壓縮記憶體宣告為一維記憶體是必需的，其中優點有：

1. 採用一維方式可減少矩陣尋找記憶體位置的次數，程式方面可將尋找位置之特定相同變數提出至迴圈外圍，方可減少計算量。

2. 編寫成一維記憶體方式程式編輯者也可以較清楚感受到記憶體是否有 JUMP，在 CPU 上的記憶體對於 JUMP 會產生延遲現象，需要盡量減少。

3. 在記憶體傳輸時使用一維矩陣的連續性高，減少延遲的次數所消耗的時間將比多維矩陣來的減少許多。

3.5.2 變異 Cholesky 演算法應用於帶狀矩陣

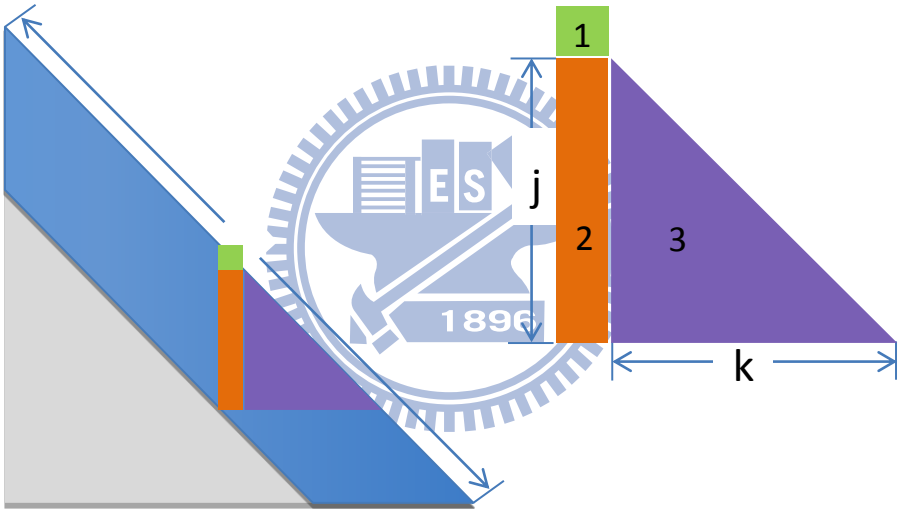


圖 3.5-3 變異 Cholesky 帶狀演算方式

步驟一：$A_{ii} \coloneqq \sqrt{A_{ii}}$

步驟二：$A_{ji} \coloneqq A_{ji}/A_{ii}$

步驟三：$A_{jk} \coloneqq A_{jk} - A_{ij} \times A_{kj}$

此演算法與 3.3 節所介紹的方式相同，但程式先後順序不同比起標準的程序，此方法擁有更好求解效率，可減少記憶體讀取次數，不僅適合在

帶狀矩陣，許多文獻在正定矩陣上也都使用這種演算方式。

3.5.3 帶狀矩陣假設原理與誤差計算

許多研究上在產生線性帶狀矩陣，是使用 The FLAME Tools【19】，The FLAME Tools 是一種方法推倒與實施稠密的線性代數，應用在各種線性求解計算。

由於 FLAME 團隊未回應且不知其他管道如何取得此 API 應用程式，本論文並未做較公正的誤差檢測，本論文自行設定一個簡易的線性代數矩陣，平行處理編寫程式除錯難度較高必須假設一組簡單線性聯立方程並且方便程式設計者的判讀錯誤。下列矩陣假設矩陣大小 n=6，寬帶 k=2，所產生的$[A][x] = [B]$如下圖：

$$\begin{pmatrix} 3 & 2 & 1 & & & \\ 2 & 3 & 2 & 1 & & \\ 1 & 2 & 3 & 2 & 1 & \\ & 1 & 2 & 3 & 2 & 1 \\ & & 1 & 2 & 3 & 2 \\ & & & 1 & 2 & 3 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{pmatrix} = \begin{pmatrix} 6 \\ 8 \\ 9 \\ 9 \\ 8 \\ 6 \end{pmatrix}$$

經由線性求解後[B]答案必須皆為 1，假使不為 1 其餘值將視為誤差，假使誤差過大程式將無法統計平均誤差則視為發散。

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{pmatrix}$$

計算平均誤差方式：

$$error = \frac{\sqrt{\sum_{i=0}^{n-1}(x_i - 1)^2}}{n} \tag{3.1}$$

3.5.4 寬帶壓縮後實際測試比較

以模型長寬比十比一的寬帶去做測試，在不同矩陣大小下測試與誤差，使用演算法是 3.5.2 節中的 Cholesky 用於帶狀演算法。下列表 3.5-1 與 3.5-2 使用相同演算法，表 3.5-1 為考慮帶狀矩陣測試，表 3.5-2 為不考慮帶狀矩陣做下三角全域計算。

表 3.5-1 考慮寬帶且壓縮矩陣情況下

矩陣寬度	寬帶	容量	時間	誤差
500000	223	446MB	129s	error
100000	100	40MB	5s	error
40000	63	10MB	0.8s	1.3%
20000	44	4MB	0.3s	0.28%

表 3.5-2 一般正定矩陣

矩陣寬度	容量	時間	誤差
20000	1600MB	2.25hour	0.28%
10000	400MB	0.28hour	0.08%
5000	100MB	0.035hour	0.03%
2500	25MB	0.004hour	0.01%

由表 3.5-1 與 3.5-1 分別在矩陣寬度 20000 時，所需要記憶體相差了 400 倍，而所需計算時間更是從 2.25 小時降至一秒內，可見考慮帶狀矩陣與記憶體壓縮，是非常有效果的，雖然壓縮矩陣在編譯上難度大幅提升，但所提高的效率遠超過平行運算，所以必須考慮進程式編輯當中。

3.5.5 精度與寬帶平行化問題

經過矩陣壓縮後，在一般桌上型電腦就可執行出矩陣寬度十萬以上的矩陣，隨著可運行的案例變大精度問題開始浮現出來，由表 3.5.2 可以看出，在矩陣寬度為 100000 時，解算精度開始發散，以迴圈次數$(k+1)^2 \times n$計算，大約在迴圈次數 10^9 次時精度發散，誤差累積造成單精度 32 位元運算已不符合使用。下表將在相同案例下比較單精度與雙精度與在相同條件下分別計算比較產生之精度差異：

表 3.5-3 單精度下的誤差表現

矩陣寬度	寬帶	容量	時間	誤差
500000	223	446MB	129s	**error**
100000	100	40MB	5s	**error**
40000	63	10MB	0.8s	1.3%
20000	**44**	**3.5MB**	**0.3s**	**0.28%**

表 3.5-4 雙精度下的誤差表現

矩陣寬度	寬帶	容量	時間	誤差
500000	223	892MB	147s	**0.00%**
100000	100	80MB	5.5s	**0.00%**
40000	63	20MB	0.9s	0.00%
20000	**44**	**7MB**	**0.3s**	**0.00%**

由表 3.5-3 與 3.5-4 可知單精度的問題在雙精度下執行可以完全解決，實際測試幾乎是零誤差值，雖然 64 位元處理器單精度與雙精度處理速度在理想上應該會相同，而實際測試雙精度對執行效率大約降低 15%，此降低之效率可被接受的。

而在平行化方面，稀疏矩陣對於平行化之影響效率變數是矩陣大小 n，矩陣大小會隨著矩陣增大而較具有平行價值，當矩陣較大時每秒所能夠產生之實數運算量隨之增加也可較接近於理想運算值。

考慮帶狀矩陣後，除了矩陣大小與寬帶大小接近，否則矩陣大小幾乎不會影響效能，影響效率的因素變為寬帶 k，寬帶較大所產的效能較好，矩陣大小必定會大於寬帶，每一次平行之運算量考慮帶狀後計算量會大幅減少，所以帶狀矩陣對平行效率來說反而是不利於平行的，在 2.5.4 節中考慮寬帶所節省的計算時間遠大於平行化所節省之時間，所以必須在考慮寬帶且壓縮矩陣底下做平行，方可達到最佳的省時效果。

3.6 Cholesky 寬帶壓縮進行 OpenMP 平行

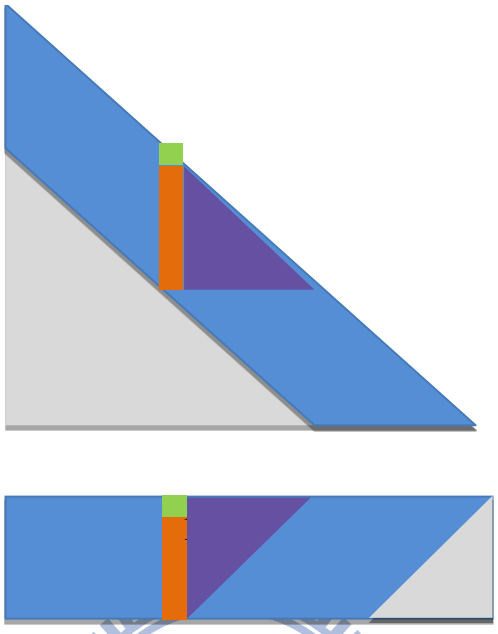


圖 3.6-1Cholesky 壓縮運作方式

由於大量計算時，三角區塊記憶體原先的每一列方向在壓縮矩陣當中皆為斜直方向，尋找記憶體位置幾乎都是以 JUMP 方式讀取記憶體連續性低，傳輸上造成延遲對於程式執行速度將會較為緩慢，在執行運算前可將大量計算部分另外宣告矩陣提出重新排列再做運算，記憶體連續性提高執行效率可與一般稀疏矩陣未經壓縮效率相同，只增加些許傳輸上短暫時間。

在 Cholesky 帶狀演算法三個步驟中可平行性：

步驟一：$A_{ii} \coloneqq \sqrt{A_{ii}}$ (不可平行)

步驟二：$A_{ji} \coloneqq A_{ji}/A_{ii}$ (平行價值低)

步驟三：$A_{jk} \coloneqq A_{jk} - A_{ij} \times A_{kj}$ (可平行)

步驟一：由於迴圈相依性質屬於不可平行部分所以程式碼不做更動。

步驟二：經由測試發現由於運算量太小，使用 OpenMP 平行去並沒有產生計算時間縮短，而在小案例底下執行運算反而會更加耗時，所以與步驟一相同不做更動。

步驟三：此步驟運算量最大，由於此矩陣是下三角形式使用 OpenMP 在迴圈當中使用均勻切割(static)效果並不好，會造分工不均勻的現象，許多 thread 會有閒置等待最後一個 thread 完成。使用 OpenMP 動態切割方式(dynamic)，將不會有彼此等待的現象，thread 會自動去幫助未完成計算，但在實際測試上效果並不理想，此切割方式所造成延遲較大，比起(static)所產生的效能還低。測試結果使用遞減切割(guided)方式效果佳，如圖 3.6-2，依造不同需矩陣大小給予一個適當的遞增值 a，使得到下三角工作分工均勻，達到「low balance」。

為了使每一個 thread 都能平均分配工作量，必須經由下三角面積公式推倒，設 k 為寬帶、x 為初始值、a 為遞增值、t 為 thread 總數(如圖 3.6-2)。以下三角形平均面積來推倒可得到 x。

$$x = n - n\sqrt{1 - \frac{1}{t}} \tag{3.7}$$

已知初始值 x 以每個 thread 面積等份可推導出遞增值 a

$$a = \frac{-1+\sqrt{1-2(x^2)+2kx}}{3} \tag{3.8}$$

此公式參數可在執行迴圈計算前先行得知，並不影響計算效能。

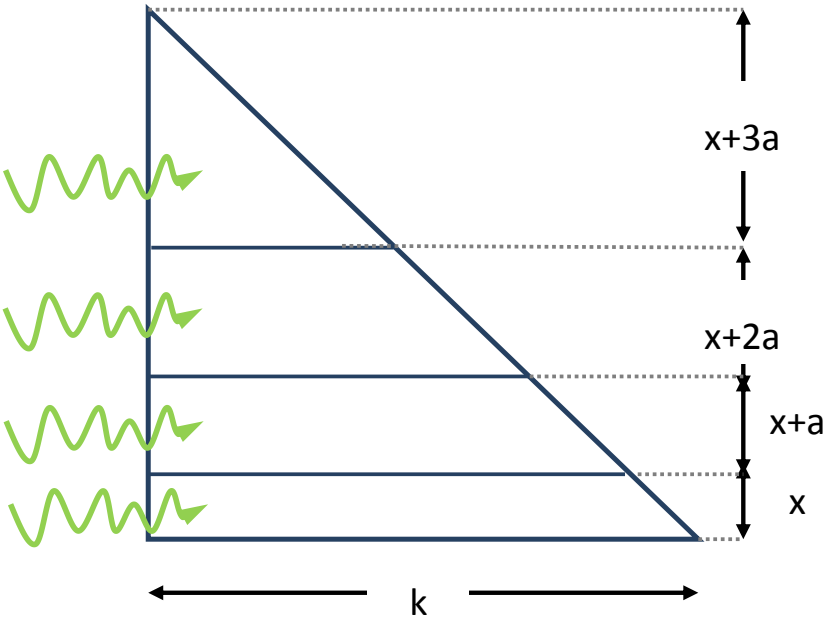


圖 3.6-2 遞減切割平行概念應用於下三角矩陣

下圖為 Cholesky 分解法考慮寬帶且壓縮矩陣進行 OpenMP 平行，利用遞減切割(guided)方式，使用硬體 i5-750 四核心 2.66GHz。

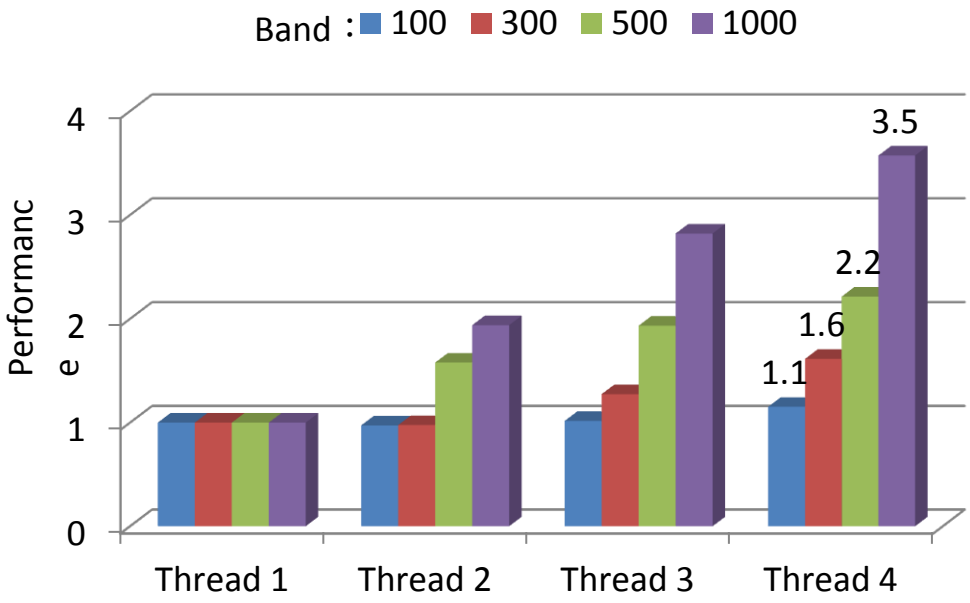


圖 3.6-3 Cholesky 寬帶使用 OpenMP 平行

由圖 3.6-3 可得到資訊，寬帶大小 k=100 時進行 OpenMP 平行與單核

心 CPU 所產生出效能僅有 1.1 倍差距，寬帶大小在 500 時效率上的差距也只有 2.2 倍的效能，唯一效果最佳只有在寬帶 1000 時效能接近 3.5 倍。

可證明考慮帶狀對於平行運算來說是較為不利的，必須在寬帶 500 以上才有明顯的效能變化，此平行程式在實際上應用的範圍也因此變得較為狹小，程式必須使要大型的案例才有平行價值。

在 OpenMP 上出現此問題，從 3.2 節的平行概念可知，多核心運算會有如此問題，假使套用在 GPU 或叢集電腦上，此問題會被更加放大，甚至變導致執行變緩慢的形情，實際應用會更狹窄，所以推測此演算法只適合在 OpenMP 上使用。

3.7 寬帶 Cholesky 塊狀分解演算法

早從 1990 年起, 不同的研究人員【20-23】提出應用儲存矩陣分塊方式相對於更習慣用於列方向存儲主要用於在 Fortran 與行方向存儲在主要用於 C 語言。原來的理由是，透過儲存矩陣的連續性讓性能提高. 最近的期刊,建議塊狀應被視為單位與操作的塊狀為單位計算【25,26】。在 2008 年有研究將帶狀 Cholesky 磚塊化，此演算法似乎適合平行化【18】。

在圖 3.7-1 中　整體$[A]$大小為$n \times n$，將矩陣$[A]$分解成 3 階 9 個區塊，A_{TL}與A_{ML}為 Cholesky 已求出的下三角值，A_{MM}矩陣大小為$k_d \times k_d$，A_{11}矩陣大小為$n_b \times n_b$會與A_{33}矩陣大小相同。n_b值必備需與 n 成倍數關係，必定$n \gg n_b$，A_{22}矩陣大小為 k。

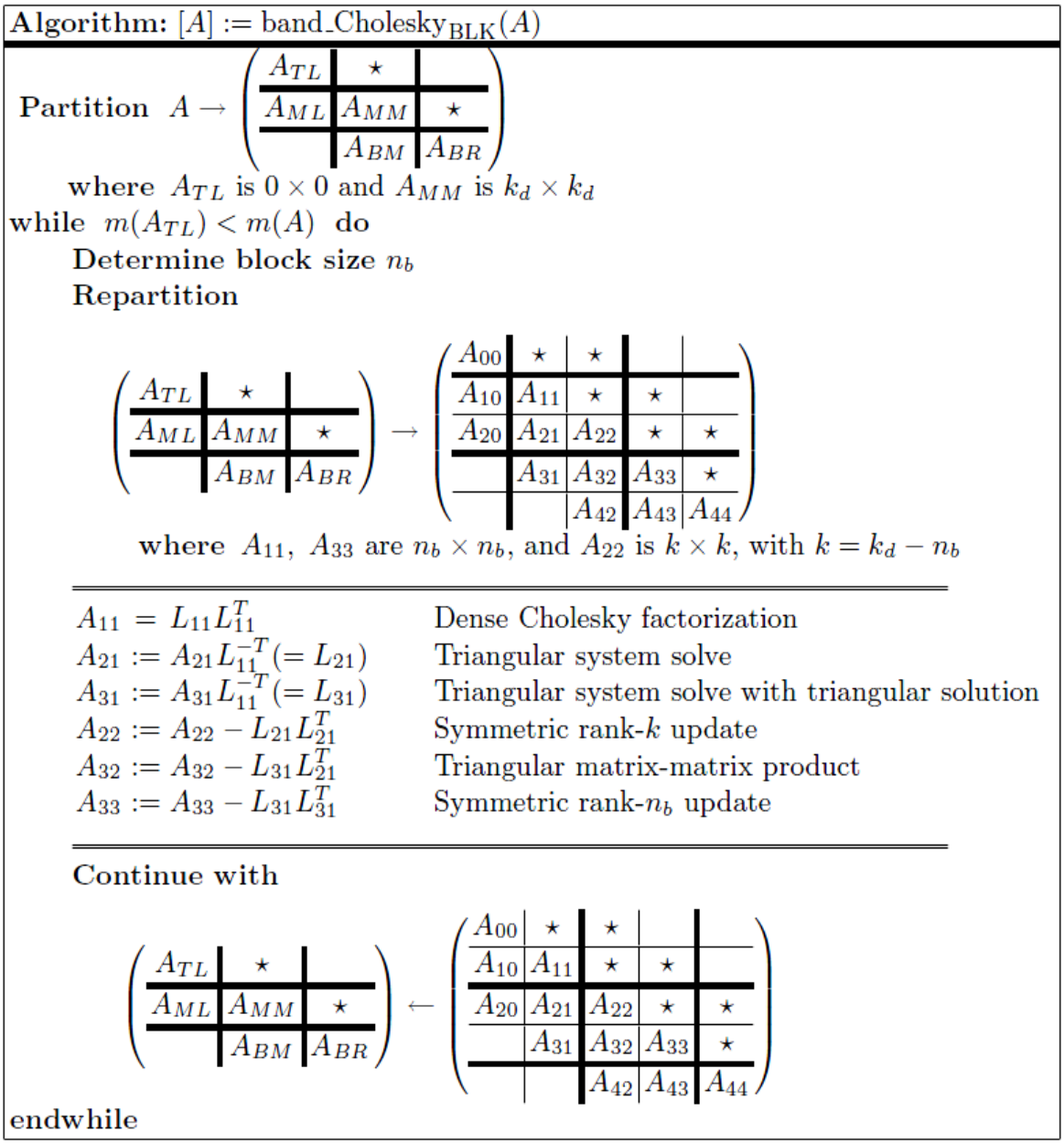


圖 3.7-1 Cholesky 塊狀分解演算法在帶狀矩陣【31】

下面將展示如何將詳細解說 Cholesky 塊狀分解的算法，更新成為帶狀 Cholesky 分解方式，使演算法分塊結果執行操作"微小的"$n_b \times n_b$塊。

如圖 3.7-1 當中的A_{11}至A_{33}之演算過程詳細方式。

$$\left(\begin{array}{c|c|c} A_{11} & \star & \star \\ \hline A_{21} & A_{22} & \star \\ \hline A_{31} & A_{32} & A_{33} \end{array}\right) \rightarrow \left(\begin{array}{c|cccc|c} A_{11} & \star & \star & \star & \star & \star \\ \hline A_{21}^{0} & A_{22}^{00} & \star & \star & \star & \star \\ A_{21}^{1} & A_{22}^{10} & A_{22}^{11} & \star & \star & \star \\ \vdots & \vdots & \vdots & \ddots & \star & \star \\ A_{21}^{p-1} & A_{22}^{p-1,0} & A_{22}^{p-1,1} & \cdots & A_{22}^{p-1,p-1} & \star \\ \hline A_{31} & A_{32}^{0} & A_{32}^{1} & \cdots & A_{31}^{p-1} & A_{33} \end{array}\right)$$

A11 經 Cholesky 分解後呈 $L_{11}L^{T}_{11}$，而後執行更新$A_{11} = A_{21}L_{11}^{-T}$詳細方式如下：

$$\begin{pmatrix} A_{21}^{0} \\ A_{21}^{1} \\ \vdots \\ A_{21}^{p-1} \end{pmatrix} := \begin{pmatrix} A_{21}^{0} \\ A_{21}^{1} \\ \vdots \\ A_{21}^{p-1} \end{pmatrix} L_{11}^{-T},$$

A21 更新後已為下三角真實值 A_{21}=L_{21}， A_{22}:= $A_{22} - L_{21}L^{T}_{21}$ 此 A_{22} 為下三角矩陣，L_{21} 與 L^{T}_{21} 矩陣相成時只需要求下三角求即可。

$$\begin{pmatrix} A_{22}^{00} & \star & \star & \star \\ A_{22}^{10} & A_{22}^{11} & \star & \star \\ \vdots & \vdots & \ddots & \star \\ A_{22}^{p-1,0} & A_{22}^{p-1,1} & \cdots & A_{22}^{p-1,p-1} \end{pmatrix} := \begin{pmatrix} A_{22}^{00} & \star & \star & \star \\ A_{22}^{10} & A_{22}^{11} & \star & \star \\ \vdots & \vdots & \ddots & \star \\ A_{22}^{p-1,0} & A_{22}^{p-1,1} & \cdots & A_{22}^{p-1,p-1} \end{pmatrix} - \begin{pmatrix} A_{21}^{0} \\ A_{21}^{1} \\ \vdots \\ A_{21}^{p-1} \end{pmatrix} \begin{pmatrix} A_{21}^{0} \\ A_{21}^{1} \\ \vdots \\ A_{21}^{p-1} \end{pmatrix}^{T},$$

$$\left(A_{32}^{0}\ A_{32}^{1}\ \ldots\ A_{31}^{p-1}\right) := \left(A_{32}^{0}\ A_{32}^{1}\ \ldots\ A_{31}^{p-1}\right) - A_{31}\begin{pmatrix} A_{21}^{0} \\ A_{21}^{1} \\ \vdots \\ A_{21}^{p-1} \end{pmatrix}^{T}$$

此演算法之所以切割成為九個區塊，理由就是 A_{31} 上三角矩陣，為了節省計算量使 A_{31} 切割進行特別處理，達到不浪費多餘的計算為目的。最後期刊【31】將此演算法進行 OpenMP 平行化，在 16 核心 CPU L 之伺服器等級電腦上平行處理，矩陣大小 n=5000，寬帶 0 至 1200。

由圖 3.7-2 可明顯看出此演算法在較小的寬帶下平行效率仍然很差，與 3.6 節有些許改善仍然有相同的問題，就算使用伺服器等級電腦在寬帶 200 以內所產生的效能仍然不佳 CPU 核心多效能仍然相同，每次平行的案例過小導致小能低落，因此認為此演算還不足以使用 GPU。

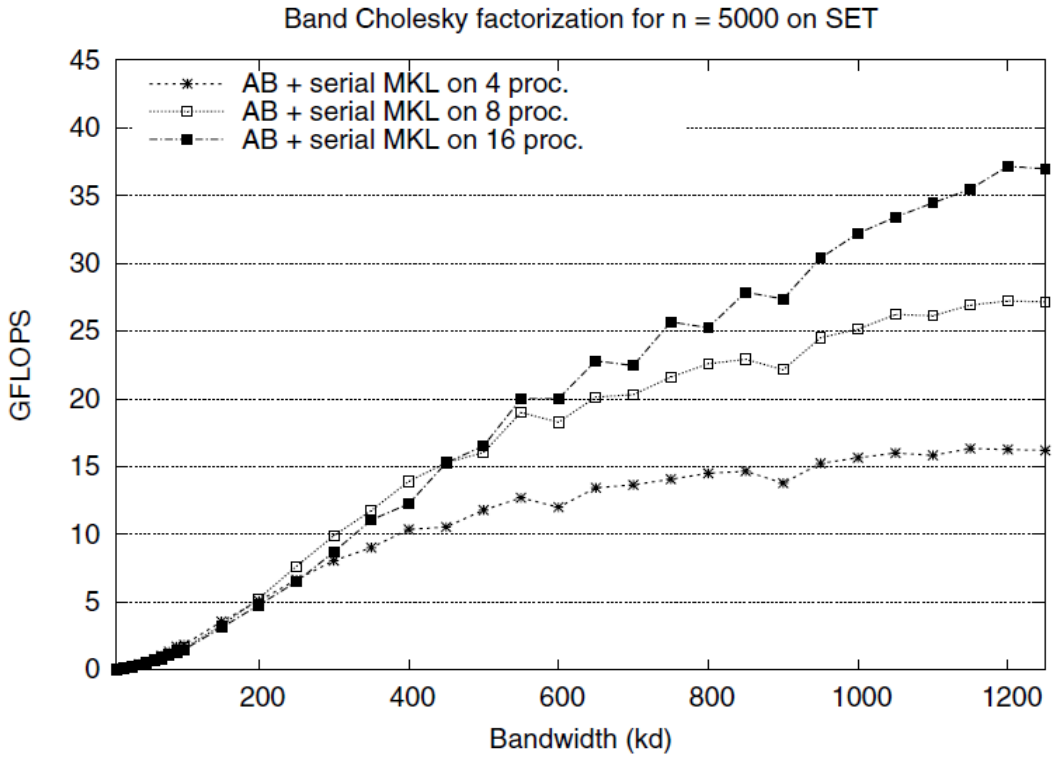


圖 3.7-2 寬帶 Cholesky 塊狀分解演算法之效能【31】

3.8 優化塊狀 Cholesky 分解演算法

由於 3.7 節當中的演算法，雖然比 3.6 節中適用性較好一些，問題寬帶適用性狹窄的問題仍然存在。

在 3.7 節中演算法針對可能的問題與缺點提出建議：

1. A_{11} 的矩陣大小並沒有適中，文獻中並沒有探討 A_{11} 在不同寬帶下所需要的大小。A_{11} 大小應該為一個隨著寬帶而變動的變數而非定值。

2. A_{21} 似乎沒做平行化，可使用圖 3.6-2 的平行概念，可縮短計算所需的時間、增加效率。

3. 3.7 節中的演算法為節省 A_{31} 計算量，而將下三角計算矩陣，切割成九份，雖然省下了計算時間，而傳輸次數增加，每次可平行的案例卻因此變小，導致 A_{32} 導致平行價值降低。

優化塊狀 Cholesky 演算法改良方式：

1. 將矩陣從原先三階矩陣切割的九個區塊，簡化為二階四個區塊，雖然浪費了 A_{31} 計算量但可以使得傳輸次數減少，A_{22} 與 A_{32} 合併，增加可平行效率。

2. 為了使得 A_{21} 可平行化，加入 GAUSS 演算法求得反矩陣，使得 A_{21} 可以在矩陣相乘中提高平行效率。

3. 將 A_{11} 大小設定為變數，增加經驗函數使得 A_{11} 大小在不同的寬帶下透過經驗函數決定一個較佳的 A_{11} 大小，使得效能提高。

3.8.1 簡化切割方式

以本研究最終的 CUDA 技術為例，GPU 與 CPU 的傳遞依靠北橋透過 PECe1.1×16 介面，CPU 與 GPU 傳遞理論上頻寬約 4GB/s，但實際上傳遞別以 1KB~100MB 測試後可以發現實際傳遞速度不符合預期，傳遞速度呈獻一種非線性的增加，在較小記憶體的對於傳輸速度上是非常不利的。LAPCK 在使用 CUDA 測試進行傳輸上測試推估出一組概估公式【16】:

$$(Time) = 15us + \frac{bytes\ transferred}{3.3GB/s} \tag{3.2}$$

在 3.7 節演算法當中為了節省 A_{31} 的計算量，將演算法切割為九個區塊，A_{31} 的計算量對整體來說比例低於 5%，為了比例低的計算量而將整體矩陣切割為九個區塊，導致每次平行的計算量變小傳輸次數較多，在圖 3.7-2 也證明此演算法在寬帶較小的時後效能並不佳。由公式 3.4 可知傳遞每次必定有延遲與其小檔案傳輸不如將矩陣合併傳輸，將原先的九個區塊降階為四區塊進行優化，A_{21} 與 A_{31} 合併，A_{22}、A_{32} 與 A_{33} 合併為一個下三角。

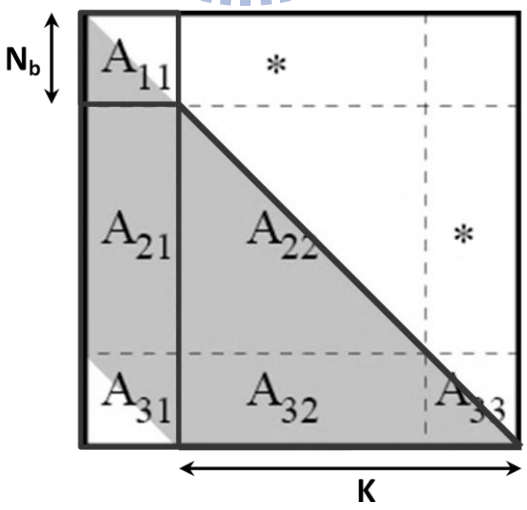


圖 3.8-1 優化後 Cholesky 切割方式【12】

合併後的公式流程為：

$$A_{11} = L_{11} * {L_{11}}^{T} \tag{3.3}$$

$$A_{11} \coloneqq {L_{11}}^{T}$$

$$\begin{bmatrix} A_{21} \\ A_{31} \end{bmatrix} \coloneqq \begin{bmatrix} A_{21} \\ A_{31} \end{bmatrix} * \left[{L_{11}}^{T}\right]^{-1} \tag{3.4}$$

$$\begin{bmatrix} A_{22} & 0 \\ A_{32} & A_{33} \end{bmatrix} \coloneqq \begin{bmatrix} A_{22} & 0 \\ A_{32} & A_{33} \end{bmatrix} - \begin{bmatrix} A_{21} \\ A_{31} \end{bmatrix} * \begin{bmatrix} A_{21} \\ A_{31} \end{bmatrix}^{T} \tag{3.5}$$

如此將能減少傳輸次數，方可將外圍迴圈次數減少且每次迴圈之計算量增大，期望透過傳輸的次數的減少與平行價值增加，減少了許多傳輸上與開啟平行化所帶來的延遲，方可將 A_{31} 之下三角所浪費掉的計算忽略。

3.8.2 演算法加入 GUASS 消去

在 3.8.1 公式 3.2 中 $\begin{bmatrix} A_{21} \\ A_{31} \end{bmatrix} \coloneqq \begin{bmatrix} A_{21} \\ A_{31} \end{bmatrix} * \left[{L_{11}}^{T}\right]^{-1}$ 希望將此平行化，所以不將 $\begin{bmatrix} A_{21} \\ A_{22} \end{bmatrix}$ 直接求解，由於 L_{11} 矩陣大小遠小於 A_{21} 與 A_{31} 首先將 L^{T}_{11} 先求反矩陣，再進行平行程度高的矩陣相乘，多做了反矩陣折損一些時間，而讓計算量較大的 $\begin{bmatrix} A_{21} \\ A_{31} \end{bmatrix}$ 透過矩陣相乘，此步驟原因是矩陣相乘擁有極佳的平行效果。

常用的反矩陣演算法為 Gauss-Jordan 求得反矩陣，L_{11} 為下三角矩陣，可以利用下三角性質使用 GUASS 求得反矩陣，不僅僅減少一半的運算數量且少了 IF 使用效能將會更佳。此步驟 A_{11} 為平行案例較小部分，較無平行價值必須盡量縮小計算量，對於整體程式效率提升有極大的幫助。

3.8.3 優化寬帶 Cholesky 分解演算流程

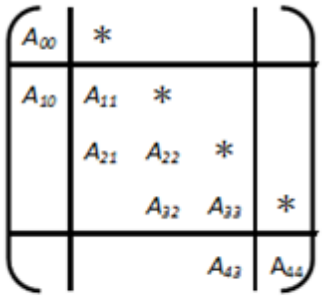
Algoritthm: $[A]$:=Band_Block_Choleskv(A)

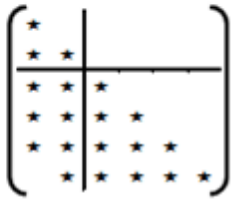
Partition [A] → $\begin{pmatrix} A_{TL} & * & \square \\ A_{ML} & A_{MM} & * \\ \square & A_{BM} & A_{BR} \end{pmatrix}$

where[A] is *nxn*，bandwidth is *k*

n_b=A11function(n,*k*) **Experience function determine A_{11} size**

for 0 **to** *n/nb* **do**

$\begin{pmatrix} A_{TL} & * & \square \\ A_{ML} & A_{MM} & * \\ \square & A_{BM} & A_{BR} \end{pmatrix} \rightarrow$ 

$[A_{MM}] = \left(\begin{array}{c|cc} A_{11} & & \\ \hline A_{21} & A_{22} & \\ A_{31} & A_{32} & A_{33} \end{array}\right) \rightarrow$ 

where$[A_{11}]$ is $n_b \times n_b$，$[A_{MM}]$ is $(k+n_b)\times(k+n_b)$

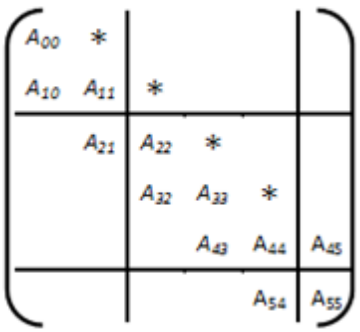
$A_{11}=L_{11}L^T_{11}$ use Cholesky factorization

$\begin{bmatrix} A_{21} \\ A_{31} \end{bmatrix} := \begin{bmatrix} A_{21} \\ A_{31} \end{bmatrix} * L_{11}^{-T} \left(= \begin{bmatrix} L_{21} \\ L_{31} \end{bmatrix}\right)$ L_{11}^{-T} use Guass inverse with Triangular matrix-matrix

$\begin{bmatrix} A_{22} & \square \\ A_{32} & A_{33} \end{bmatrix} = \begin{bmatrix} A_{22} & \square \\ A_{32} & A_{33} \end{bmatrix} - \begin{bmatrix} L_{21} \\ L_{31} \end{bmatrix} * \begin{bmatrix} L_{21} \\ L_{31} \end{bmatrix}^T$ Triangular matrix-matrix

---

Continue with

$\begin{pmatrix} A_{TL} & * & \square \\ A_{ML} & A_{MM} & * \\ \square & A_{BM} & A_{BR} \end{pmatrix} \leftarrow$ 

end for

3.8.4 矩陣壓縮方式

此演算法多了 A_{33} 的下三角，原本的壓縮方式已不符合使用，壓縮後矩陣大小$(k+n_b)\times n$ 如圖 3.8.3，此種壓縮方式並不符合 PBTRF 格式，如圖 3.8.3 A_{31} 只有上三角具有真實數值下三角為 0，所以黃色區域是可以忽略的，程式本身必須增加 if 做切割犧牲些許的效率，方可達到 PBTRF 所相容之壓縮格式。

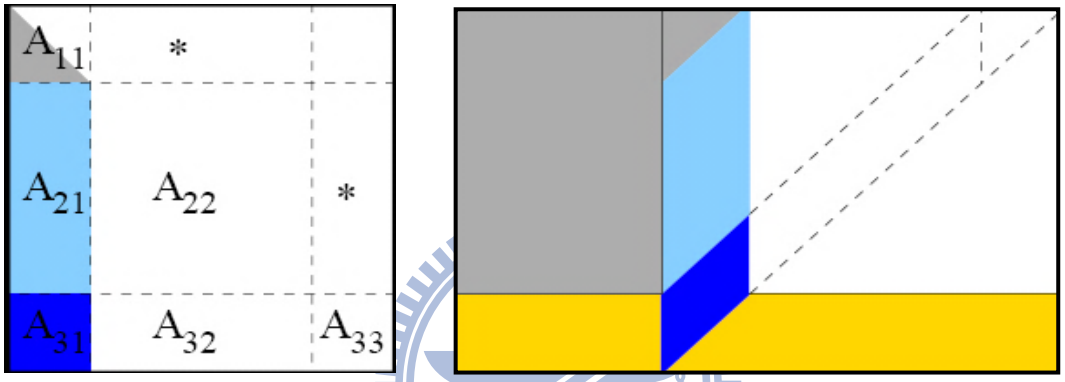


圖 3.8-3 左圖為稀疏矩陣下帶狀擷取、右圖為壓縮後存取方式【12】

3.8.5 一般 Cholesky 與塊狀 Cholesky 結果比較

由 3.8.3 節中公式得知，塊狀演算法比起 3.5.2 當中的演算法，從公式上來看，增加 A_{11} 的計算量從原先的單一開根號計算轉變為塊狀矩陣分解，以及多出了 A_{33} 下三角多餘運算，兩者皆會造成計算量的增加。

經過實際測試由表 3.8-1 可看出一般 Cholesky 與 塊狀 Cholesky，大約降低了 5%效能，此演算法有較佳的平行效能，所以多出了 5%的計算時間是可被接受的。

表 3.8-1 一般 Cholesky 與 塊狀 Cholesky 執行成果

		一般 Cholesky		塊狀 Cholesky	
矩陣大小	寬帶	時間	誤差	時間	誤差
500000	223	**119s**	0.0%	**125s**	0.0%
100000	100	**5.6s**	0.0%	**5.5s**	0.0%
40000	63	**1s**	0.0%	**0.9s**	0.0%
20000	44	**0.4s**	0.0%	**0.3s**	0.0%

3.8.6 塊狀 Cholesky 平行 OpenMP 策略

在公式 3.1 當中將 A_{11} 做 Cholesky 分解，A_{11} 矩陣在一般情況下，矩陣大小並不大，平行價值較低避免平行後在小案例計算中速度遭拖累，在此不做平行化。

$$A_{11} = L_{11} * {L_{11}}^{T} \tag{3.3}$$

公式 3.2 當中的 L^{T}_{11} 以高斯概念求得反矩陣，此方法不做平行理由與上述相同。而 A_{21}、A_{31} 與 $\left[{L_{11}}^{T}\right]^{-1}$ 計算量大矩陣相乘平行價值較高。

$$\begin{bmatrix} A_{21} \\ A_{31} \end{bmatrix} := \begin{bmatrix} A_{21} \\ A_{31} \end{bmatrix} * \left[{L_{11}}^{T}\right]^{-1} \tag{3.4}$$

在公式 3.3 當中，此計算量是程式當中比例最高也最為耗時，平行效能與整體計算速度提升也決定於此，此平行切割方式如圖 3.6-2，轉制與相乘在 CPU 當中會有特殊的解算方法，在 3.8.6 節當中會詳細說明。

$$\begin{bmatrix} A_{22} & 0 \\ A_{32} & A_{33} \end{bmatrix} := \begin{bmatrix} A_{22} & 0 \\ A_{32} & A_{33} \end{bmatrix} - \begin{bmatrix} A_{21} \\ A_{31} \end{bmatrix} * \begin{bmatrix} A_{21} \\ A_{31} \end{bmatrix}^{T} \tag{3.5}$$

整體上優化過後演算法只有在 3.2 式與 3.3 式的矩陣相乘需要平行，其餘效小之運算量程式將不做更動。

3.8.7 高效能矩陣轉置與相乘

$$[A] \times [B] = [N]$$

在矩陣相乘當中，高效率的矩陣相乘矩陣先行轉置，再做同方向的矩陣相乘，目的是提高記憶體讀取的連續性，效果將會比傳統方式快上許多倍，在 C 語言當中須將$[B]$先轉置再以列方向做迴圈，而在 Fortran 語言中則反之。在公式(3.1)當中，$\begin{bmatrix} A_{21} \\ A_{31} \end{bmatrix} * \begin{bmatrix} A_{21} \\ A_{31} \end{bmatrix}^T$ 此公式須先做轉置在相乘，剛好符合上述的方式，不必轉置直接以同方向進行相乘，可得到正確又快速的答案(如圖 3.8-2)。

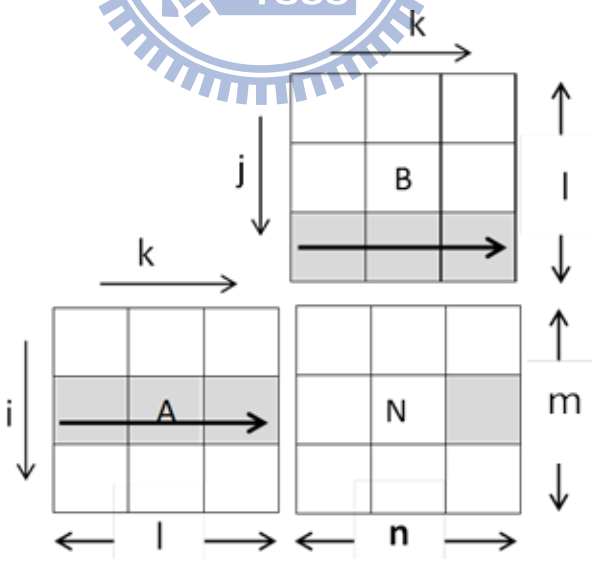


圖 3.8-2 高效率矩陣相乘方式

在平行化方面，矩陣相乘中$[N]$矩陣運算時是完全獨立的，所以平行化效率高，只需將最外層迴圈平行即可得到不錯的效能。

表 3.8-2 矩陣相乘演算法

Algorithm: $[N]$=Matrix_Mul(A,B,n,m,l)	
1:	**transpose**(D, B, l, n); $[D] := [B]^T$ **where** $[A]$ is $(m \times l)$, $[B]$ & $[D]$ is$(l \times n)$
2:	**#pragma omp for** OpenMP Language
3:	**for** i=1 **to** m **do**
4:	**for** j=1 **to** n **do**
5:	**for** k=1 **to** l **do**
6:	N[i*n + j]+= A[i*l + k] * D[j*l + k]
7:	**end for**
8:	**end for**
9:	**end for**

3.8.8 A11 之大小探討與優化

A_{11} 的在面積比例上與 A_{33} 相同是所有區塊中最小的，但扮演的角色卻非常關鍵，並沒有相關期刊在探討這問題，A_{11} 的大小決定於，最外圍的迴圈次數(3.8.3 節)，影響每一次的計算量，但並非 A_{11} 越大越好，A_{11} 是一個平行價值低，須經由 Gauss 與 Cholesky 演算法兩者計算量皆為 $O(n^3)$的演算法，所以 A_{11} 矩陣過大會造成計算量太大，效率降低，A_{11} 矩陣過小則會造成公式 3.2 與 3.3 的計算量過小，造成平行價值降低與傳輸次數過多導致延遲增加效率降低，A_{11} 大小必須適中每一種不同案例會有不同的最佳尺寸。

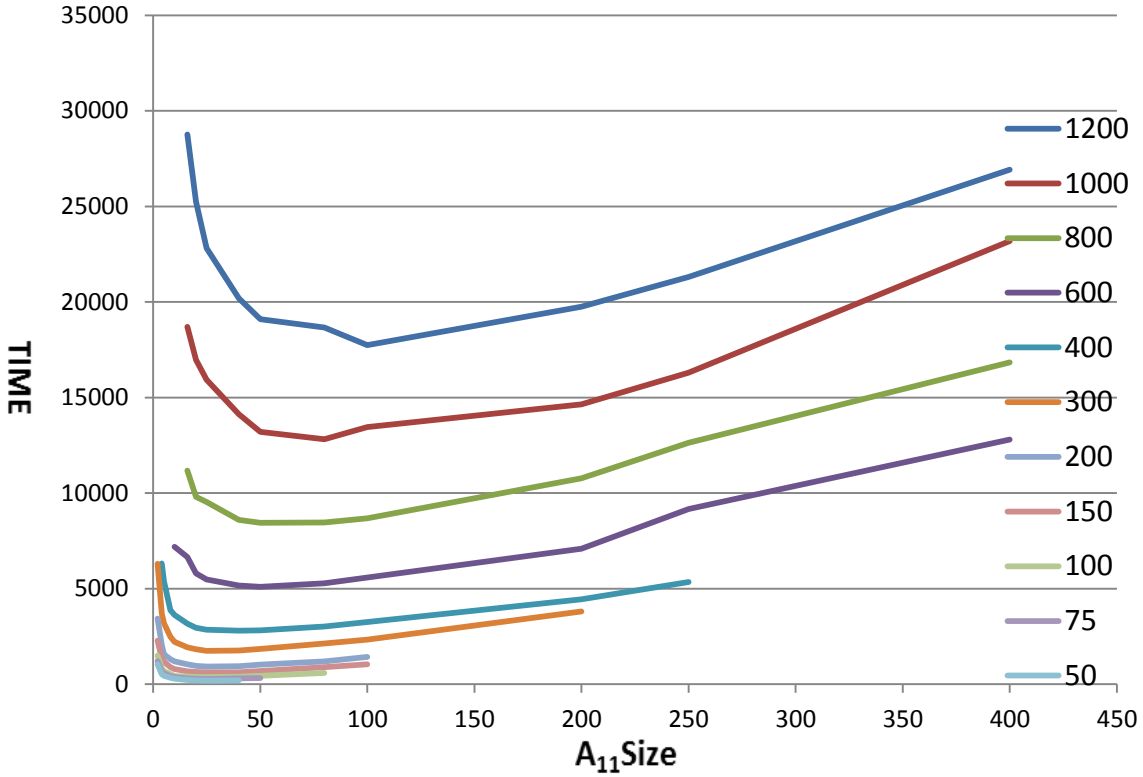


圖 3.8-3 A_{11} 矩陣大小對整體程式時間的影響

圖 3.8-3 為實際案例測試寬帶大小，使用硬體 CPU：AMDX4-2.6GHz、記憶體：8GB，A_{11} 大小由 1 到 400 分別做實際測試，似乎可以找出 A_{11} 的最佳大小，實測時發現在相同案例不同的電腦平台下會有不相同的最佳解，每一台電腦之記憶體頻寬、CPU 暫存的大小頻寬、CPU 內部的時脈，都會影響極值的位置，要分析歸類出每台電腦的可能的 A_{11} 極值，由於太過於複雜難以準確得之每台電腦所適合的最佳大小，所以使用了簡單的經驗公式概估：

$$\text{經驗公式：} \begin{cases} n_b \approx n^{0.5} * 2.3 + 5 \\ n/n_b \; is \; integer \end{cases} \tag{3.6}$$

推估 A_{11} 較適合的大小，測試發現極值會出現在寬帶大小的 1/10 至 1/20 內，另外極值的條件是 A_{11} 尺吋必須是 n 的因數，符合兩者條件計算結果才會正確又較為快速。

3.8.9 OpenMP 平行成果比較結論

硬體設備 CPU：i5-2.66GHz，圖 3.8-4 左圖為 3.6 節中一般 Cholesky 平行結果，右圖為 3.8 節中優化後 Cholesky 的進行 OpenMP 平行結果：

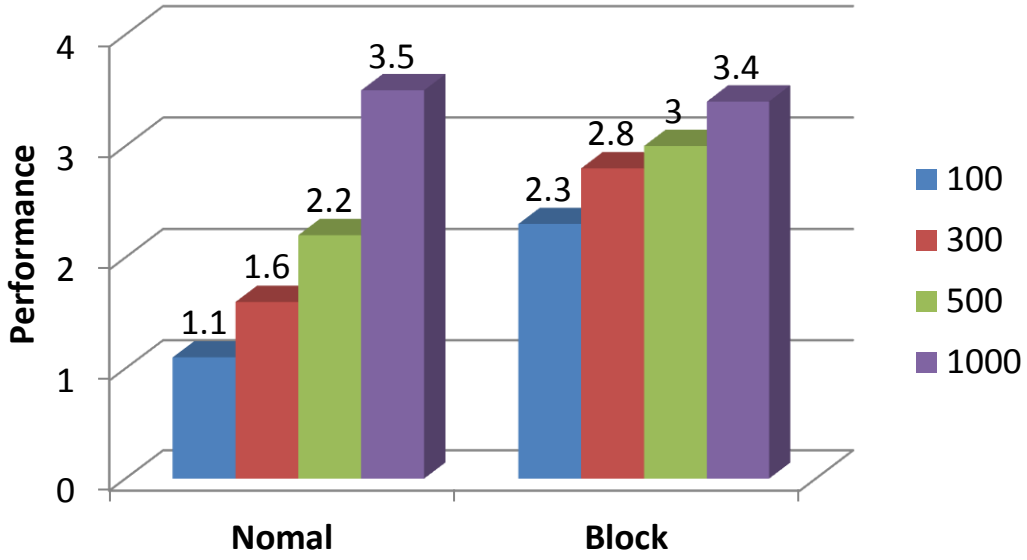


圖 3.8-4 一般 Cholesky 與 優化塊狀 Cholesky 成果

由圖 3.8-4 可以明顯的看出塊狀化在較小的寬帶也有良好的效果，在寬帶 k=100 時有了 2.3 倍的效率，寬帶在 100 以上皆適用於 OpenMP 平行化，不會有適用狹小的問題，適用性明顯比一般 Cholesky 演算法好。

為了與期刊接軌，Linpack 公司在演算法比較的規定單位皆為 Flops 每秒所能計算的實數數量，而此考慮寬帶演算法實數運算計算公式參考【12】。

帶狀實數計算量公式：$n(k^3+3k)$ (3.7)

在時數運算量上方面，圖 3.8-6 採用公式 3.6 以$4*n(k^3+3k)$的計算量做評估方式只是初略估計並不精確，真實實數運算值需由 LINPACK 來做公正評估。

圖 3.8-5 紅色現為 3.7 節中的演算法公式、藍色為改良之演算法兩者 n_b=20、綠色為增加 3.8.8 節中經驗公式優化，硬體為 AMDX4 2.6GHz 進行測試。

由圖 3.8-5 可得到資訊，在寬帶為 600 成長速度就開始減緩了，可以看出此演算法在較小寬帶具有較佳的效能。也證明了將演算法切割成二階，浪費 A_{33} 計算量反而可以將傳輸時間減少平行價值增加，讓效能提升，優化後將再度可提升效能。

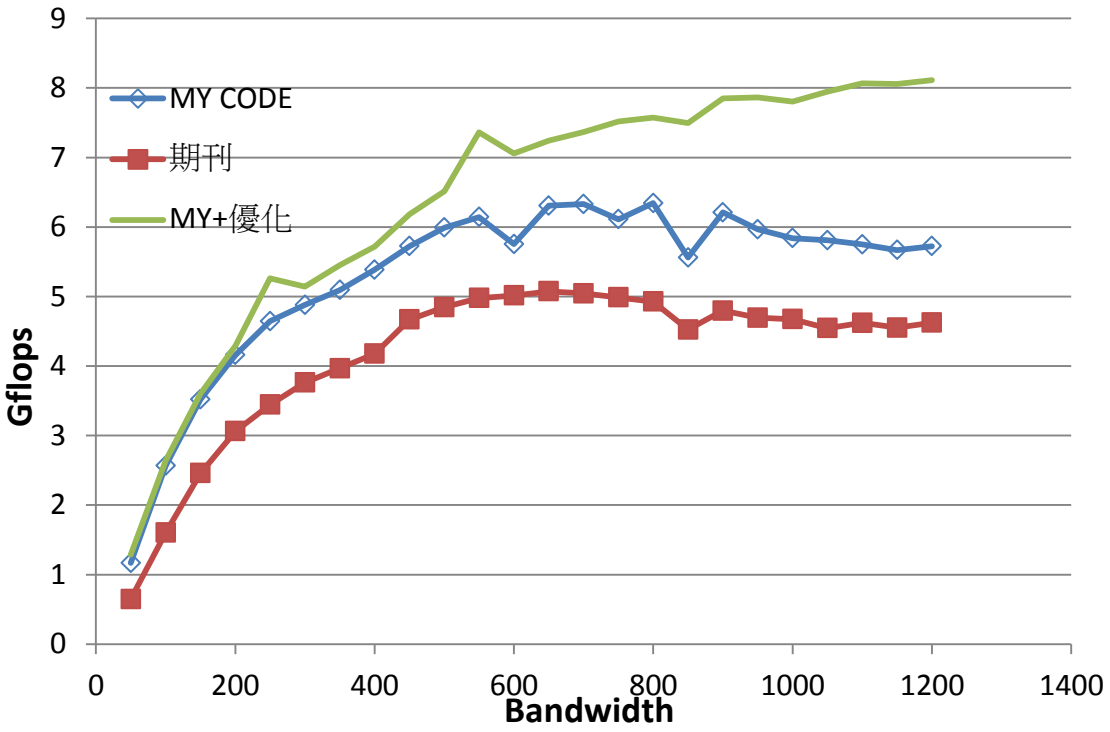


圖 3.8-5 改良塊狀 Cholesky 的實數運算效能

3.8.10 CUDA 平行運算

由 3.8.8 結論可以得知，優化過後 Cholesky 較適合平行計算，如今我們認為此演算法也較適合 CUDA，導入 GPU 使用平行運算，希望可以帶

來更好的效果。本論文 GPU 平行運算所使用的硬體設備皆為 CPU：Q6600 RAM：2GB 、GPU：GTX285，2011 年來看整台電腦價值不超過三萬元，希望可以帶來節省成本的目的。

由於 CUDA 是 GPU 與 CPU 協同運算，GPU 的效能並不能代表整體，在不平行時都是交由 CPU 所運算，此演算法傳輸記憶體非常頻繁，不僅僅是 CPU 重要，北橋晶片的傳輸能力也更為突顯重要。

3.8.11 CUDA 平行方式

演算法需要平行的部分與 3.8.5 雷同(如圖 3.8-7)，Cholesky 與 Gauss 反矩陣不做平行，公式 3.2 的計算量 CUDA 來說仍然是相當小的計算量，不採用下三角矩陣相乘進行平行運算，使用全域矩陣相乘反而較有效率。

公式 3.3 為一個下三角矩陣相乘，由於 CUDA 的 BLOCK 限制，每一個 BLOCK 最佳為 16*16 矩陣(如圖 3.8.8)，會呈現一個類似下三角鋸齒矩陣如圖：3.8-7，回傳 Host 之後須由 CPU 將鋸齒狀切割為下三角，增加一個轉置的平行計算，在 3.8.6 節中的 CPU 計算矩陣相乘演算法，在 GPU 上已不符合使用，由於顯示卡為定址記憶體，產生 JUNP 的問題聘不容易發生，較不必考慮記憶體連續性問題。

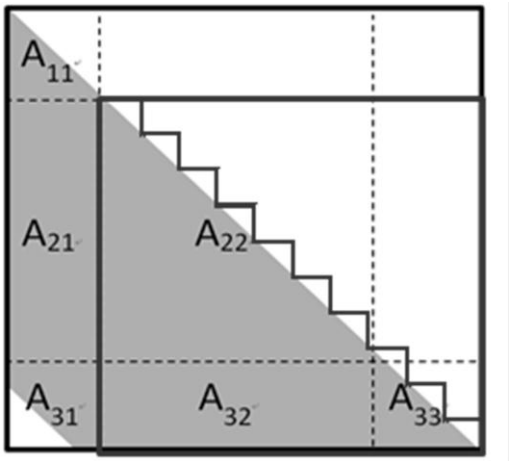


圖 3.8-6 下三角磚塊化

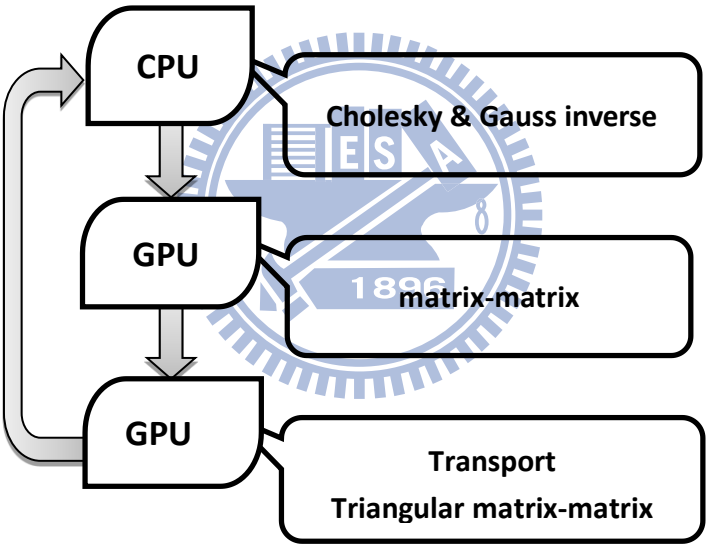


圖 3.8-7 GPU 與 CPU 分配計算方式

3.8.12 CUDA 矩陣相乘

矩陣相乘是 CUDA 演算法中重點研究之一，Nvidia 每年都會公佈效能

較佳的 SDK 程式範例，矩陣相乘也擁有了極佳時數運算效能，以變於提供使用者學習使用。在 3.8.6 節當中，利用轉置提高記憶體連續性，在 CUDA 當中並沒有這種記憶體連續性的問題，因 GPU 的記憶架構為定址記憶體不會發在 CPU 上的 JUMP 所造成的延遲問題，反而 CUDA 在矩陣相乘中著重於 Shear memory 的用法，3.8.6 的演算法會造成 Global memory 提取 Shear memory 造成重複提取同一區塊使得效能反而變降低。

GPU 當中每個 SM 可接受 512 或 768 條 thread，在矩陣相乘磚塊化中都習慣使用 16x16 的矩陣大小為 BLOCK 的單位，每個 SM 就能同時處理兩個 BLOCK，拿到較佳的效能，也因此矩陣大小必須是 16 的倍數才能夠計算，當矩陣不足 16 的倍數須自行補足空矩陣。

在矩陣相乘中每一次 BLOCK 的讀取利用 Shear memory 計算次數多，節省傳輸時間比 CPU 還多出許多，往往可以超越理論值達到與 CPU 比較的 100~1000 倍效能，矩陣相乘的效能一直是平行運算當中公認效能出色的演算法之一。

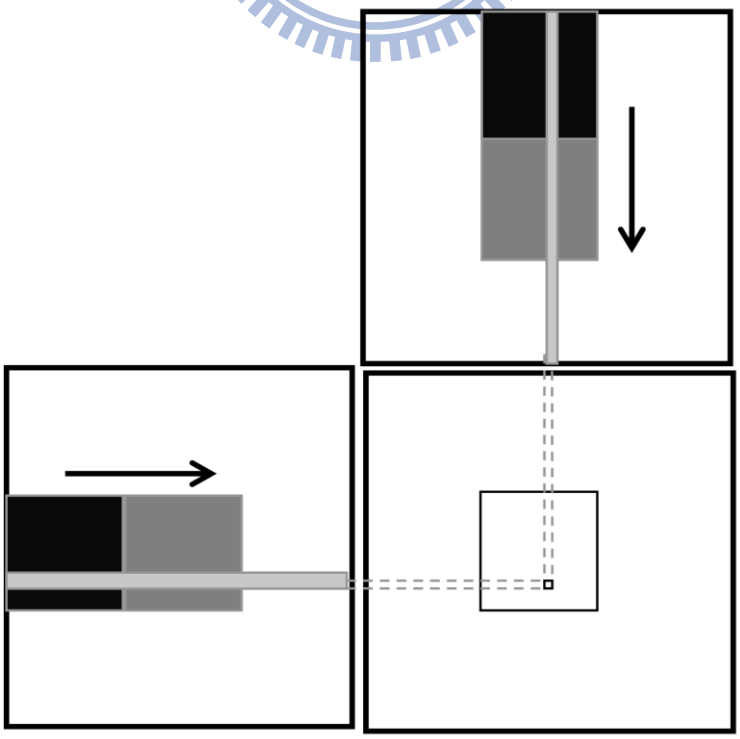


圖 3.8.8 CUDA 矩陣相乘模式【11】

3.8.13 CUDA 平行成果

在公式 3.2 中所使用的全域矩陣相乘使用 CUDA 技術，寬帶與 L_{11} 依造 3.8.7 節中的經驗公式給予 10 比 1 做測試，成果如圖 3.8-9，寬帶 1000 時可產生 378 倍的速度，與 CUDA 比較的演算法是 3.8.6 的高效矩陣乘法，在最小矩陣 100 當中就能比 CPU 快上 9 倍的效能非常快速、適用性也極佳。

$$\begin{bmatrix} A_{21} \\ A_{31} \end{bmatrix} := \begin{bmatrix} A_{21} \\ A_{31} \end{bmatrix} * \left[{L_{11}}^{T} \right]^{-1} \qquad (3.2)$$

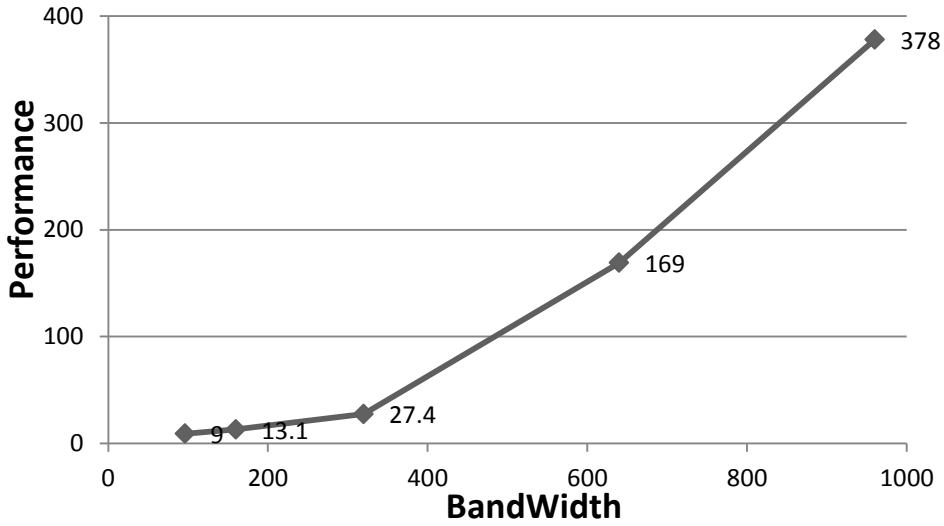


圖 3.8-9 CUDA 在 3.2 公式中矩陣相乘可產生效能與 CPU 比較

在公式 3.3 節當中必須先將轉置再做相乘會有比較好的效果，成果如圖 3.8-10，再轉置矩陣方面採用 Nvidia 官方公布效能較好的演算方法，同樣利用 Shear memory 將矩陣磚塊化轉置提高效能，但轉置計算不大，Shear memory 利用效率很低，寬帶 1000 時最高只能達到 14 倍速度相較於 CPU，

但轉置為 Order 二次方占整體計算時間比例相當少，並不會造成時間上的影響。

$$\begin{bmatrix} A_{22} & 0 \\ A_{32} & A_{33} \end{bmatrix} := \begin{bmatrix} A_{22} & 0 \\ A_{32} & A_{33} \end{bmatrix} - \begin{bmatrix} A_{21} \\ A_{31} \end{bmatrix} * \begin{bmatrix} A_{21} \\ A_{31} \end{bmatrix}^T \qquad (3.3)$$

在公式3.3節當中的矩陣相乘，只計算下三角矩陣此部分計算量龐大，在寬帶 1000 時可達到 790 倍速度，最低寬帶為 100 時也有 45 倍速度。

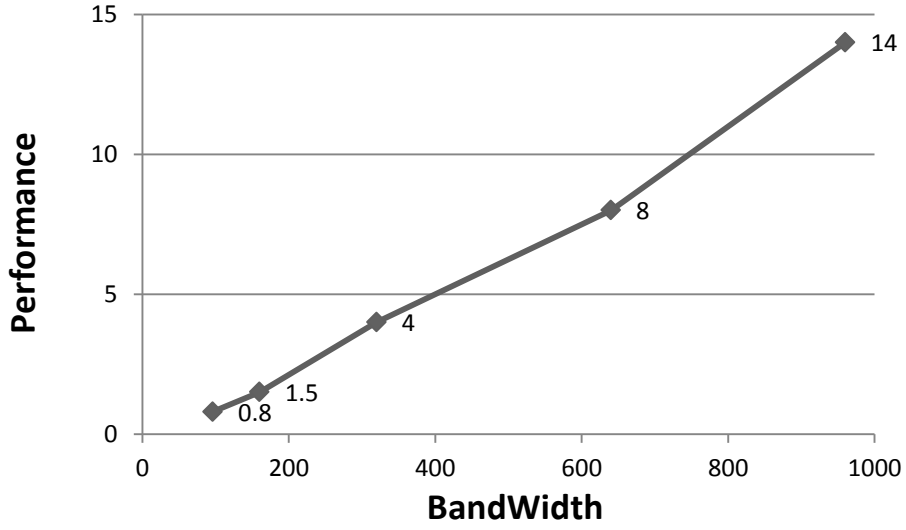


圖 3.8-10 CUDA 在 3.3 公式中轉置矩陣可產生效能與 CPU 比較

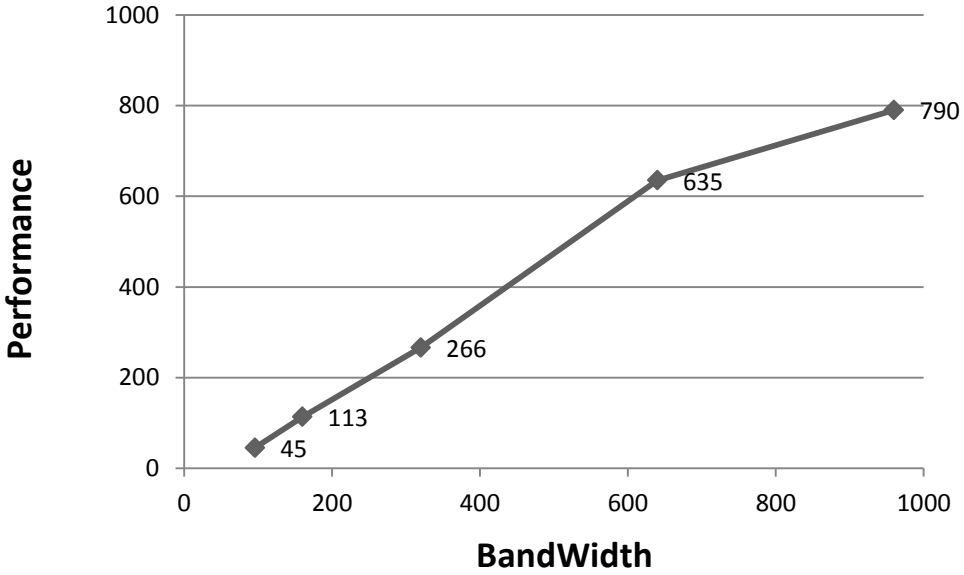


圖 3.8-11 CUDA 在 3.3 公式中矩陣相乘可產生效能與 CPU 比較

圖 3.8-12 為 CUDA 整體程式效能，CUDA 在單精度下可達 15 Gflops，但精度問題如同 CPU 一樣單精度下會產生發散問題，但 GPU 雙精度效能比起單精度慢了一倍，整體效能大約可達到 10Gflops。

比起 CPU 達到 10 倍以上的效能是可以辦到的，本研究使用的是較舊型 GPU 顯示卡，由於市場成本考量對於雙精度的支援較低，在程式編寫上不能直接以雙精度編寫，使用雙精度以 GTX285 的 GPU 將會損失剩 1/12 的效能，這是目前使用較舊的硬體架構上令人麻煩的問題，遊戲顯示卡雙精度使用需求少，為了節省成本支援雙精度的核心，目前較新的遊戲顯示卡已將雙精度的支援程度提升，將來在硬體上很有可能獲得解決，目前要達到較好的效能需要在程式上單精度與雙精度程式個別編寫，在矩陣未解算時就必須估計是否需要雙精度，如此一來就可較有效率的發揮硬體效能。

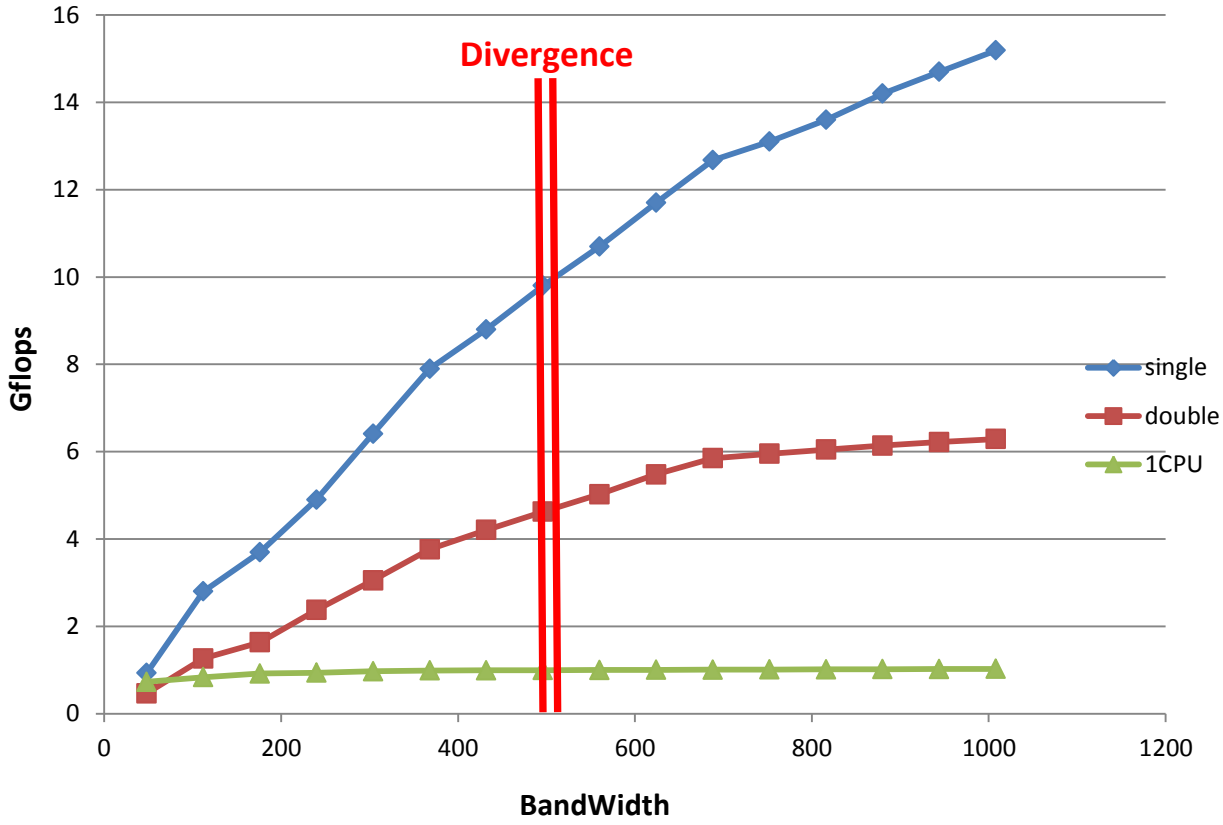


圖 3.8-12 塊狀化 Cholesky 使用 CUDA 在單精度與雙精度效能

3.9 精度探討與解決方案

GPU 的雙精度會造成效率降低，因此希望可以增加精度將 GPU 單精度時的應用範圍增加，在較大的寬帶下延後發散的可能性。

策略 1：

在 Gauss 常用增加精度時方法是樞紐轉換,將最大值調列至斜對角可增加精度，但在帶狀矩陣下卻無法使用調列會造成帶狀破壞

，因此許多人都會認為斜對角值對於整體解算精度來說非常關鍵。

將 CUDA 計算為單精度，A_{11} 交由 CPU 計算採用雙精度減少誤差，

測試結果： 精度能提高 10%

如此說來 Cholesky 斜對角在優良矩陣下對於精度的影響並不大。

策略 2：

在 Cholesky 完成後分解後進行遞推解算，此步驟並不平行運算量也不大，因此不做討論，而將矩陣轉為雙精度再進行遞推。

測試結果： 精度能提高 30 倍精度。

30 倍精度看似很多，但與雙精度的 2^{32} 倍差距仍然極大，提升的效果非常有限，將全矩陣轉為雙精度需消耗時間與一倍的記憶體，策略 1、2 所帶來精度效果皆不好，目前仍無法有效解決單精度發散問題。

3.10 綜合結果與討論

由圖 3.10-1 可得到資訊，在寬帶 n=1000 時 OpenMP 可達到 3.5 倍以上效能而 CUDA 可達到 10 倍以上效能，CUDA 與 OpenMP 在效能上有一段不小的差距且 CUDA 在效率的趨勢上似乎仍然沒趨緩的現象，這也是為何 GPU 在平行處理上計算效能會如此吸引本論文採用，相同的效能下，所發費的成本、許消耗的能源都可降低許多，所以許多大量計算軟體在未來都希望將 GPU 技術納入自家軟體。

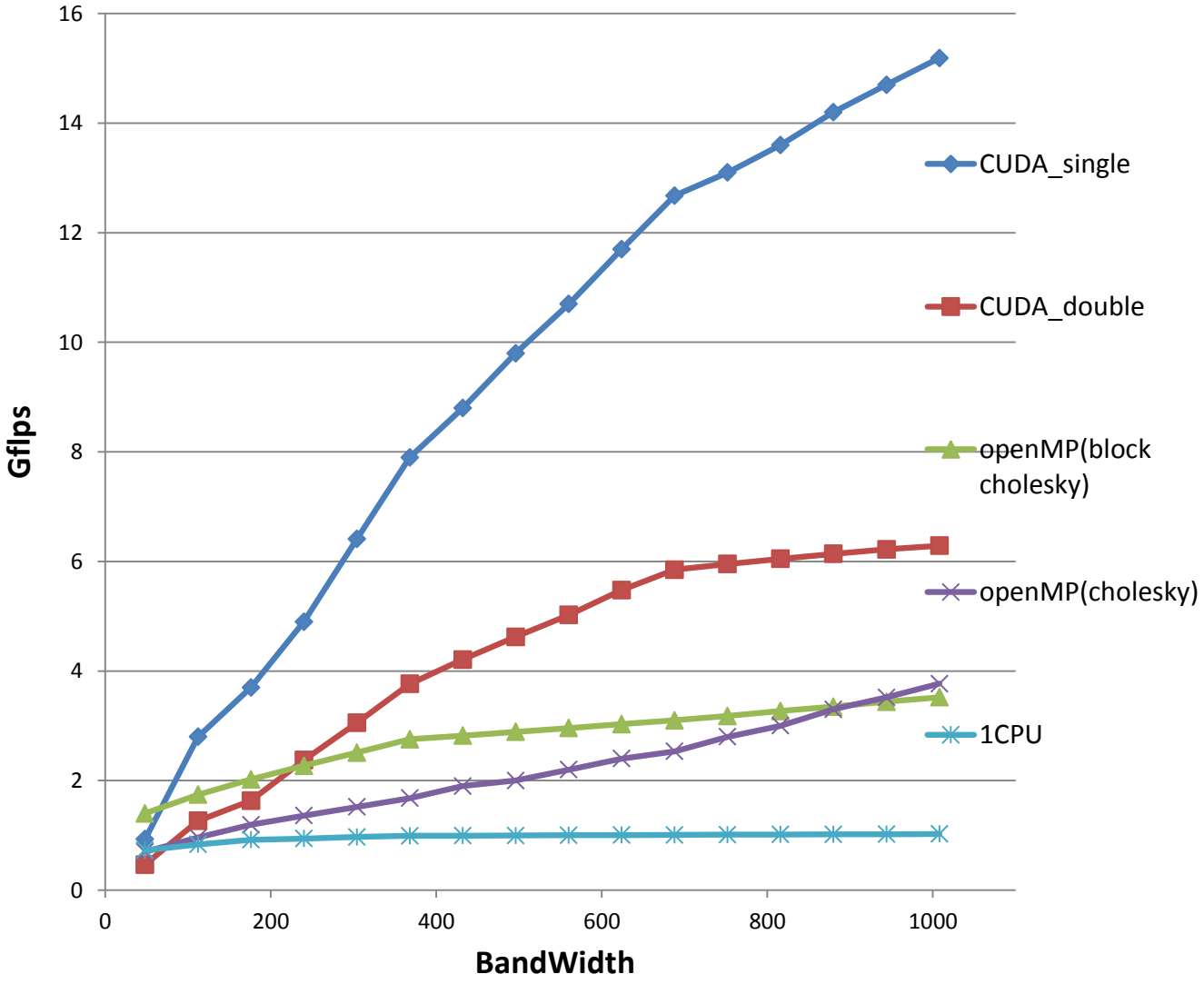


圖 3.10-1 綜合結論比較

第四章 結論與未來建議

運用現今平行計算技術及硬體架構發展一套有效的演算法以解決花費大量時間、記憶體需求量大的問題，選定效率較高且適用性較好的直接求解法做為基礎，以此來發展出適合於寬待矩陣的平行演算法，並將利用現今平行處理技術 OpenMP 及 CUDA 分別用於多核心 CPU 架構與效率較高的 GPU 系統上發展出高效率的平行處理演算法，藉此達到節省時間、降低所需記憶體的目的。本研究將以主流的數學函數機構所制訂的壓縮格式來編寫程式，希望可以將本程式以物件函數方式嵌入軟體，應用在符合線性求解系統之各種軟體領域當中。

4.1 結論

1. Cholesky 優勢與可應用範圍

Cholesky 分解法相較於其他演算法是個計算量需求效小求解較為快速限制於對稱矩陣的演算法。不僅僅能應用於有限有元素法、只要是求解對稱連立方程在任何領域皆能應用。在考慮寬帶後，可大幅度減少實數運算量節省大量計算時間，經由壓縮過後採用 PBRT 壓縮格式雖然記憶體連續性降低犧牲一些效能傳輸且程式編寫難度提高，但可大量解省記憶體，降低硬體需求避免矩陣數值轉入硬碟當中。

2.優化 Cholesky 演算法經 OpenMP 測試

將 Cholesky 演算法進行優化，分為二階區塊增加了一些計算量，減少了記憶體傳輸次數，利用 A_{11} 下三角特性加入了 Gauss 消去求反矩陣，增加了矩陣相乘增加了可平行性，整體來說增加 10%以內的運算時間。優化後 Cholesky 使用 OpenMP 平行，在 1000 寬帶下可達 3.4 倍，實數運算達

3.8Gflops 而平行的適用性大幅增加寬帶 100 以上就能有不錯的平行效果。

3.優化 Cholesky 演算法使用 GPU 計算

以同樣演算法代入CUDA，將兩種矩陣相乘、與一個轉置分別進行GPU平行處理，最終結果單精度在 1000 寬帶下可達 14Gflops，雙精度在 1000 寬帶下可達 10Gflops，由於 GPU 在雙精度下效能降低一半，精度問題在 CPU 計算時都使用雙精度，GPU 使用單精度此方法效果並不如預期，目前難以完全解決此問題，雙精度在大型矩陣下是必須使用的，而在 GPU 發展上雙精度效能一直提升在未來硬體上可望可以與單精度效能接近。

4.2 建議

1. mCUDA

並非每一台電腦皆有 NVIDIA 8 系列以上的獨立顯示卡，許多電腦仍然是以低階的顯示晶片為主，因此無法使 CUDA 做 GPU 計算。在 CUDA 程式開發上的推廣與適用性就會因此而降低。大部分電腦都是多核心 CPU，OpenMP 的適用性會較為廣泛，因此開發出 mCUDA 格式【24】，把 CUDA 的程式，透過 OpenMP 來做平行化處理，套用在多核心 CPU 上。而由於 thread 的建立在 CPU 上相對比較浪費時間，因此 MCUDA 是把 CUDA 中的一整個 thread block 用一個 CPU 的 thread 來執行，以減少建立過多 thread 的負擔。

如此一來就可將 OpenMP 與 CUDA 相容，當程式偵測硬體設備 CPU 的計算效能超越 GPU，就使用多核心 CPU 做計算，反之 GPU 效能偵測比 CPU 好時就交由 CUDA 來計算如圖 4.2-1。

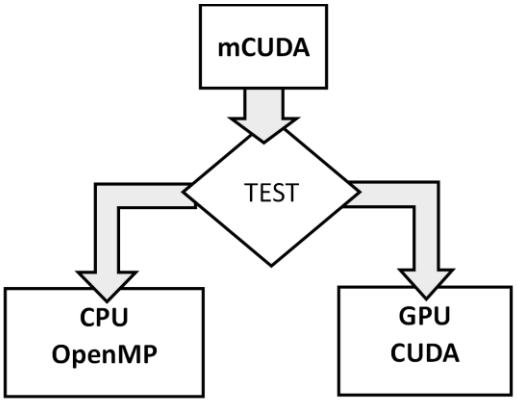


圖 4.2-1 mCUDA 格式

2.Cholesky 解病態矩陣

本論文研究方法皆以優良矩陣假設問題，一般演算法並不能解算出斜對角值趨近為 0 的聯立方程組【26】，可依造 ANSYS 納入解算器方式將程式另外編輯專屬於 Cholesky 解病態矩陣之解算器。

$$A = LDL^T$$

$$L = \begin{pmatrix} 1 & & & \\ l_{21} & 1 & & \\ l_{31} & l_{32} & \ddots & \\ l_{41} & l_{42} & l_{43} & 1 \end{pmatrix} \qquad D = \begin{pmatrix} d_1 & & & \\ & d_2 & & \\ & & \ddots & \\ & & & d_N \end{pmatrix}$$

將[A]矩陣分解三部分，下三角、對角、上三角矩陣，可解算出斜對角值趨近於 0 的解。可預期的是解算所花費計算時間必定會較多，精度與平行化部分是值得期待的。

3.OpenMP + CUDA 應用於 Cholesky

CPU 實數運算雖然效能不及 GPU，但傳輸速率快、適用性廣、程式編輯容易、硬體門檻低等優點。而 GPU 目前架構都是以北橋傳輸，需要一定的計算量才會有 GPU 平行價值的存在。在 3.8.5 節當中唯一不去平行的就是 A_{11} 的 Cholesky 與 Gauss inverse 的矩陣過小並不適合平行，OpenMP 不適合 CUDA 就更沒理由如此了。

將 Cholesky 與 Gauss 利用四核心 CPU 進行 OpenMP，在下三角矩陣下對於矩陣大小所產生的效能如，Cholesky 因被函數 sqrt()所拖累，造成需要矩陣大小 150 以上才能夠有加速效果，而高斯在小矩陣下反而會有較好的效果，整體大約估計來說將 A_{11} 平行 Cholesky 與 Gauss 至少要矩陣大小 100 以上，才有真正的加速效果，由 3.8.7 節中的經驗公式寬帶與 A_{11} 尺寸 10:1 的結論來說，A_{11} 矩陣大小到 100 寬帶至少要達到 1000。在寬帶達到 1000 而 A_{11} 生的平行價值後，進行 OpenMP 僅僅是縮短 A_{11} 計算時間，A_{11} 時間縮短也意味著在相同時間內 A_{11} 可容納更大的矩陣，如此將可以造就後方計算量增大，增加平行價值減少傳輸次數。3 A_{11} 進行 OpenMP 對於程式整體可將極值再度縮短時間，A_{11} 與寬帶的經驗公式將再度改變。

在 CUDA NVCC 編譯器當中不僅僅可以編譯 CUDA 也可同時編譯 OpenMP API 語法，如此我們可以將運算量最大的矩陣相乘交給 CUDA 做運算。利用 A_{11} 利用 OpenMP 小計算量適用性較佳的特性，與 CUDA 在矩陣相乘的高效能，在程式編輯中進行 OpenMP 與 CUDA 共同編譯。

此演算法 OpenMP + CUDA 適用寬帶 1000 以上之矩陣，較適合記憶體龐大的伺服器，本章節並未程式編譯完成，但最終成果是可以進行推估的預期。在這種超大計算量上，疊代求解法的優勢會展現出來，同時也需要進行疊代法測試與比較才會較為完整。

參考文獻

[1]High Performance Computing Training www.llnl.gov/computing/tutorials/workshops/workshop/openMP/MAIN.html.

[2] J. L. Gustafson, Reevaluating Amdahl's law, Comm. ACM 31(5), 532 (1988).

[3] Y. Shi. Reevaluating Amdahl's Law and Gustafson's Law.Computer Sciences Department, Temple University (MS:38-24),Oct. 1996.

[4]J.Dongarra,"The LINPACK Benchmark: An Explanation. Supercomputing, 10-14, Spring 1988.

[5]張舒、袾絕利、趙開勇, GPU 高性能運算之 CUDA,松崗出版社,2010

[6] 吳政倫,Implementation of Bzip2 Data Compression Algorithm with Parallel Program Based on GPU, Tsinghua University,2010

[7] Based on slide 7 of S. Green, "GPU Physics," SIGGRAPH 2007 GPGPU Course.

[8] Quintana-Ort´ı, G., Quintana-Ort´ı, E.S., Rem´on, A., van de Geijn, R.: SuperMatrix for the factorization of band matrices. FLAMEWorking Note #27 TR-07-51, The University of Texas at Austin, Department of Computer Sciences (September 2007)

[9]Peter Pacheco, 1997, Parallel Programming with MPI , Morgan Kaufmann Publishers,San Francisco, California

[10] Microsoft 高效能運算 平行程式設計軟體 http://www.microsoft.com/taiwan/windows2000/hpc/devtools.htm

[11] NVIDIA CUDA C Programming Guide version 4.0,October 2010 http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf

[12] Alfredo Rem´on, Enrique S. Quintana-Ort´ı, Gregorio Quintana-Ort´I, Cholesky Factorization of Band MatricesUsing Multithreaded BLAS, June 2006

[13] 程曉旭、耿魯靜、張海、王勇,數值分析-使用C語言,佳魁出版社,2009

[14] Robert Bridson,CS 542G: Solving Sparse Linear Systems, November 26, 2008

[15] by Yanhui Dong1 and Guomin Li2,A Parallel PCG Solver for MODFLOW, Vol. 47, No. 6–GROUND WATER–November-December 2009

[16] V. Volkov and J.W. Demmel, ‘‘LU, QR and Cholesky Factorizations Using Vector Capabilities of GPUs,’’ tech. report UCB/ EECS-2008-49, EECS Dept., Univ. of Calif., Berkeley, 2008.

[17] J. Dongarra, J. Du Croz, S. Hammarling, and I. Duff, A set of level 3 basic linear algebra subprograms. ACM Transactions on Mathematical Software, pages 1–17, March 1990

[18] Henry, G.: BLAS based on block data structures. Theory Center Technical Report CTC92TR89, Cornell University (February 1992)

[19] Bientinesi, P., Gunnels, J.A., Myers, M.E., Quintana-Ort´ı, E.S., van de Geijn, R.A.:The science of deriving dense linear algebra algo- rithms. ACM Transactions onMathematical Software 31(1), 1–26 (2005)

[20] Chatterjee, S., Lebeck, A.R., Patnala, P.K., Thottethodi, M.: Recursive array layouts and fast matrix multiplication. IEEE Trans. on Parallel and Distributed Systems 13(11), 1105–1123 (2002)

[21] Elmroth, E., Gustavson, F., Gustavson, F., Jonsson, I., Kagstrom, B.: Recursive blocked algorithms and hybrid data structures for dense matrix library software. SIAM Review 46(1), 3–45 (2004)

[22] Henry, G.: BLAS based on block data structures. Theory Center Technical Report CTC92TR89, Cornell University (February 1992)

[23] Chan, E., Quintana-Ort'ı, E., Quintana-Ort'ı, G., van de Geijn, R.: SuperMatrix out-of-order scheduling of matrix operations for SMP and multi-core architectures.In: SPAA 2007: Proceedings of the Nineteenth ACM Sym- posium on Parallelism in Algorithms and Architectures, pp. 116–126 (2007)

[24] Chan, E., Van Zee, F., van de Geijn, R., Quintana-Ort´ı, E.S., Quintana-Ort´ı, G.:Satisfying your dependencies with SuperMatrix. In: IEEE Cluster 2007, pp. 92–99 (2007)

[25] John A. Stratton, Sam S. Stone, and Wen-mei W. Hwu MCUDA: An Ef_cient Implementation of CUDAKernels on Multi-cores ,March 12, 2008

[26] 桂田祐史,Cholesky 分解ノート,2008 年 6 月 9 日 http://www.math.meiji.ac.jp/~mk/labo/text/cholesky.pdf

[27] 芮雍能,整合平行計算與生物演算法的技術應用於整體最佳化問題,國立暨南大學,2008

[28] Yanhui Dong1 and Guomin Li2,A Parallel PCG Solver for MODFLOW, Vol. 47, No. 6–GROUND WATER–November-December 2009

[29] E. Anderson, Z. Bai, C. Bischof, L. S. Blackford, J. Demmel,J. Dongarra, J. D. Croz, A. Greenbaum, S. Hammarling,A. McKenney, et al., *LAPACK Users' Guide* SIAM,Philadelphia, 1999

[30] Dongarra, J.J., Duff, I.S., Sorensen, D.C., van der Vorst, H.A.: Solving Linear Systems on Vector and Shared Memory Computers. SIAM, Philadelphia ,1991

[31] Gregorio Quintana-Ort´ı1, Enrique S. Quintana-Ort´ı1, Alfredo Rem´on1,and Robert A. van de Geijn2. An Algorithm-by-Blocks for SuperMatrix Band Cholesky Factorization,2008

[32] Pacheco, P.S. Parallel Programming with MPI, Morgan Kaufmann Publishers,1996

[33] Wallcraft, A. J. SPMD OpenMP vs. MPI for Ocean Models, Proceedings of FirstEuropean Workshops on OpenMP (EWOMP'99), Lund, Sweden, 1999.

[34] OpenMP Architecture Review Board. Fortran 2.0 and C/C++ 2.0 Specifications, At www.openmp.org

[35] Zhang, Z. and Zhang, X. Fast Bit-Reversals on Uniprocessors and Shared-MemoryMulti-processors, SIAM Journal on Scientific Computing, vol.22, no.6, pp.2113-2134,2000.

[36] Chuan Chen ,The Parallel Molecular Dynamics Simulation Performance and System Analysis on PC Cluster, October 2010

[37] 沈澄宇, High Performance Computing: An Introduction, 國家高速網路與計算中心, 5th Revised ,2003,http//www2.nchc.gov.tw/~c00cys00/

[38] Release 11.0 Documentation for ANSYS, Chapter 3. Solution http://www.kxcad.net/ansys/ANSYS/ansyshelp/Hlp_G_BAS3_2.html