國立交通大學

電子物理學系

碩士論文

利用繪圖處理器平行運算技術計算電子電洞交換能

GPU's parallel computing technology for the computation of electron-hole exchange interaction energy

研究生：饒家祥

指導教授：鄭舜仁教授

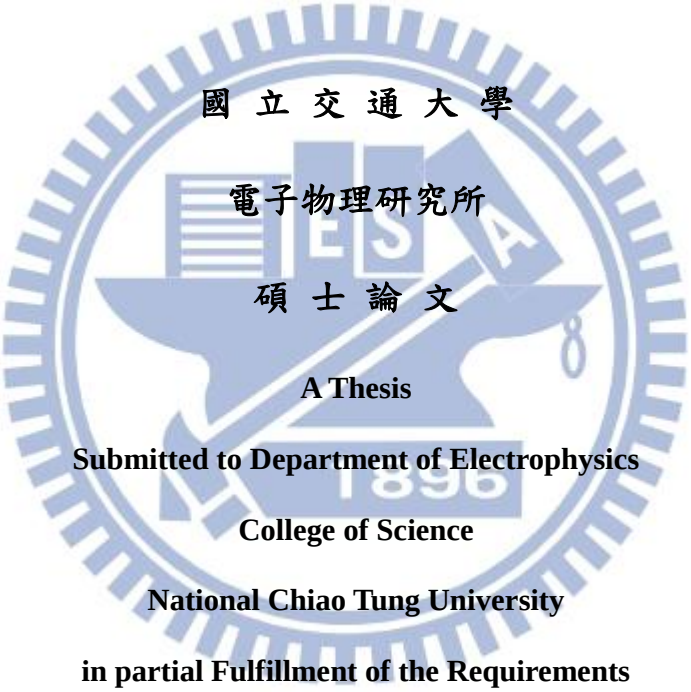
中華民國一百零一　年　九　月

利用繪圖處理器平行運算技術計算電子電洞交換能

GPU's parallel computing technology for the computation of electron-hole exchange interaction energy

研 究 生：饒家祥　　Student：Chia-Hsiang Jao

指導教授：鄭舜仁　教授　　Advisor：Shun-Jen Cheng



國 立 交 通 大 學

電子物理研究所

碩 士 論 文

A Thesis

Submitted to Department of Electrophysics

College of Science

National Chiao Tung University

in partial Fulfillment of the Requirements

for the Degree of

Master

in

Electrophysics

2012

HsinChu, Taiwan, Republic of China

中華民國一百零一年九月

利用繪圖處理器平行運算技術計算電子電洞交換能

學生：饒家祥　　指導教授：鄭舜仁　博士

國立交通大學電子物理研究所碩士班

摘要

本論文主要探討繪圖處理器平行運算技術計算電子電洞交換能，將庫侖交互作用積分式子離散化後程式中實質上是六重迴圈，因此我們將研究繪圖處理器對迴圈增快的速率。而後會介紹一些有關繪圖處理器的每個核心是如何運行的，以及記憶體的分配與傳遞方式，進而了解每個細節對迴圈增快的效益，使我們能更進一步了解程式碼在程式中執行的方式，讓程式的撰寫更順利。

程式中所使用的理論則是利用量子點 $\vec{k}\cdot\vec{p}$ 單能帶模型和有效質量近似法的激子系統計算庫侖交互作用，而庫侖交互作用分為直接庫侖作用與交換能。直接庫侖作用的計算則是引用建智學長論文[1]中的程式，將程式修改成使用繪圖處理器作程式平行運算即可；對於交換能而言，先前使用中央處理器單核心計算需耗費大量的時間，一直無法驗證程式的正確性，而後使用繪圖處理器計算時，相對於單核中央處理器時間減少 10 倍以上，因此我們將能驗證此程式的正確性並討論數值解與解析解的誤差，對於將來要計算庫侖矩陣以及交換能矩陣時，能計算的更迅速。

GPU's parallel computing technology for the computation of electron-hole exchange interaction energy

Student：Chia-Hsiang Jao **Advisor：Shun-Jen Cheng**

Department of Electrophysics

National Chiao Tung University

Abstract

This thesis theoretically investigates GPU's parallel computing technology for the computation of electron-hole exchange interaction energy, the Coulomb's integration in the program is essentially the six loops, we will study the rate is accelerated by the GPU loop. Accelerated rate in order to study, will find some relevant information to understand, and thus to understand every detail of the loop to increase the amount of fast effective help when we write code, to further understand the code in the Fortran program is how to perform.

When it calculates the Coulomb's interaction, we are based on $\vec{k}\cdot\vec{p}$ theory's single-band model and effective mass approximation method of exciton's system in the quantum dot to calculate, the Coulomb's interaction divide into direct coulomb and exchange energy. Direct coulomb's part have been calculated via Chien-Chih senior's thesis[1], so long as the original program have to modify the GPU who can run the program；exchange energy's part have been unable to verify the correctness of the program, the reason is that the program's execution time is very long. When using the GPU run the program, it will speed up to 10 times than single-core CPU, so we will be able to verify the correctness of this program in short time(error less than 5%), calculating Coulomb's matrix and exchange energy's matrix in second quantization in the future can calculate more quickly.

誌謝

首先要感謝鄭舜仁老師在我碩士生涯兩年來的指導與諄諄教誨，讓我能在固態物理與程式理論運算能有很大的進步與了解，而在此領域中也不時的指點我正確的方向與研究上應具備的態度，使我在這些年中獲益匪淺。感謝各位口試委員，陳煜璋老師、張文豪老師、林炯源老師在口試時提出寶貴的意見使我受益良多。

在這兩年碩士生涯中，感謝盧書楷學長與廖禹淮學長的教導與照顧，遇到困難時適時的指引我一盞明燈，讓我能繼續向前走。感謝研究室學長姐鄭丞偉、張語宸、林以理、陳力瑋、張書瑜，在我剛進來時能適時的教導我一些研究上所需的技術與基礎觀念，更要特別感謝帶領我的學長趙虔震、廖建智，教導我與研究相關的技術和物理觀念而能有重大的突破，使論文能順利完成。感謝研究室同屆同學余書睿、張智偉、林佩儀，不管是在課業或研究上的討論都能給予幫助，使我能順利突破難關。也要感謝學弟妹的陪伴與幫忙，使我生活中添增了一些色彩。感謝與我一起前來交大的同學們源龍、威霖、光彥、凱勝、軒毫、維綸，有你們給予我鼓勵使我更有信心突破難關。

感謝我的父母與姐姐、弟弟與家人親戚能適時的鼓勵我，使我能無後顧之憂的專心唸書與學習。以及遇到挫折時能給予我一個溫暖又舒適的家，使我有繼續堅持下去的勇氣。由於要感謝的人太多無法在致謝一一答謝，因此最後要感謝曾經幫助過我的親朋好友們說聲謝謝。

目錄

表目錄

圖目錄

第一章、序論

1.1 研究背景與動機

隨著科技的進步，半導體產業將電晶體越做越小，以目前發展可做到奈米等級的大小，當電晶體尺寸小於或等於德布羅伊(de Broglie)波長時，電子將不再遵循古典物理規則，這時必須考慮量子效應(Quantum Effect)。目前量子侷限有 3 種，分別量子井(Quantum well)、量子線(Quantum wire)、量子點(Quantum dot)，而量子點三維都受到侷限近似零維[2]，因此量子效應相較其他量子侷限會比較大。

在量子資訊發展前期，D. P. DiVincenzo提出好的量子體系有5個要素[2]：

1. 要有定義明確，容易擴充(scalable)的量子位元系統。
2. 要能夠把量子位元初始化成如|000…>。
3. 量子位元要有夠長的去同調時間(decohernce time)。
4. 要能夠對量子位元實現普世量子邏輯閘。
5. 要能夠高效率的量測量子位元。

隨著量子資訊的發展，其它要素也相映著出現，但這 5 個要素仍是每個研究者追求的目標。1997 年 Daniel Loss 跟 David P. DiVincenzo 提出了利用半導體量子點來體現量子位元的構想[2]，此構想讓量子資訊跟半導體量子點做連結，使奈米尺度技術與量子侷限理論能被廣泛的運用。

量子點具有優良的發光性質，用於量子傳輸(quantum teleportation)、量子位元(quantum qubit)、量子密碼(quantum cryptography)應用上[3]。其中量子傳輸是利用量子點當作發射糾纏態光子對(entangled photon-pair)的光學元件，這裡糾纏表示量子糾纏，意旨複合的量子系統中有特定關聯，無法被分解成各自量子態的乘積[4]。而量子點發光必須要考慮庫侖交互作用，分成電子－電洞間直接庫侖作用與電子－電洞間交換能，而電子－電洞間交換能會使量子點內產生精細結構匹裂(fine structure splitting，FSS)，使單激子(single exciton)自旋態能階匹裂，讓自旋不同電子電洞結合的路徑變成可分辨，妨礙糾纏光子對發生[3]。

庫侖交互作用實質上要計算二次量子化後的CI矩陣，當我們選擇當基底的組態越多時，其矩陣元素會越多，而每個矩陣元素都代表是一個或多個六重迴圈的積分，對於此模擬需使用大量的數值運算，因此會耗費大量的時間。為了將耗費時間縮短，必需找尋變快的方法，而我們想利用或GPU硬體多核心功能，縮短數值運算的時間；電子－電洞間直接庫侖作用目前已經有相關的程式，因此只要將程式改成能讓GPU運算的程式碼，就能讓運算速率加快，而對於電子－電洞間交換能雖然已經有程式，但以之前技術要計算量子點以$\vec{k}\cdot\vec{p}$單能帶模型和有效質量近似法計算激子系統的交換能，想將計算誤差限制5%以內並且驗證其正確性，必須要花費極多時間，因此我們引入GPU相關技術，利用此技術驗證電子－電洞間交換能數值結果是否正確，而本論文要驗證的是電子－電洞間長程偶極－偶極交換能的部份。電子－電洞間交換能分成短程作用與長程作用，計算長程作用積分時為了避免計算到短程作用的部分，將會考慮短程與長程作用之間的界限，使我們能計算正確的結果。長程交換能又分成長程單極－單極交換能與長程偶極－偶極交換能，目前要驗證長程偶極－偶極交換能的部份。由於研究主軸包含GPU的使用與相關技術，因此下一小節將會介紹GPU歷史、CPU與GPU的差異及使用GPU的好處。

1.2 GPU歷史背景簡介

這幾年來，隨著科技蓬勃發展，電腦運算速度也日益增加，GPU 也隨著時間運算能力不斷提升，許多高階程式使用者發現 GPU 運算的潛力，在 2003 年 SIGGRAPH(美 國 計 算 機 協 會 計 算 機 圖 形 專 業 組) 大 會 提 出 GPGPU(General-purpose computing on graphics processing units)的想法[5]，之後引發各大廠商的關注，因此 GPU 開發方向以固定功能單元(Fixed Function Unit)轉成專用併行處理器為主[5]。

在硬體方面,在一般電腦處理資料都是由CPU(Central Processing Unit)執行。就目前來說，CPU 單核時脈提升已經到達一個瓶頸了，許多 CPU 廠商原以增進單核時脈提升速度轉成發展多核心，這時需要多核心來處理圖像的 GPU 崛起。GPU 原是為了影像處理而設計的，但近年來被運用到高速運算中，因 GPU 核心數多且擁有可發展性。對核心數而言，GPU 個數比 CPU 來的多，因此延伸出一個想法，將每個核心都運用到計算能力上，且提升核心時脈，可讓 GPU 運算效能大幅提升，使運算能力遠超過 CPU。

以目前單核心來說,CPU 時脈可以到達 3.5GHz 以上[6],GPU 就只有 1.5GHz 以下[7]；整體來說,CPU 核心最多 8 個核心,GPU 可以有 200 個以上甚至更多，因此GPU 浮點運算能力比 CPU 更為迅速。以實際例子來解釋,CPU 型號為 i7-960 與 GPU 型號為 Tesla C2050 來比較，利用以下浮點運算能力公式[6]：

浮點效能(Gflops)=
核心數×單核心頻率(GHz)×每秒運行最高浮點運算次數(flops) (1.2.1)

可計算出如表 1.2.1：

表 1.2.1 比較 CPU 與 GPU 核心數及計算速率。

	CPU	GPU
型號	I7-960	Tesla C2050
核心數	4	448
單核心頻率(GHz)	3.2	1.15
每秒運行最高浮點運算次數(flops)	4	1
浮點效能(Gflops)	51.2	515

因此可看出 GPU 計算能力比 CPU 還要快很多。

表 1.2.2 比較目前最新 CPU(i7 系列)與 GPU(Telsa 系列)核心數及時脈。

CPU 型號	處理器核心	處理器時脈(MHz)
i7-3960X	6	3300
i7-3930K	6	3200
i7-3820	4	3600
i7-2600	4	3400
vs		
GPU 型號	CUDA 處理器核心	處理器時脈(MHz)
Tesla C2050/C2070/C2075	448	1150
Tesla C1060	240	1296

表 1.2.2 為最新 CPU(i7 系列)與 GPU(Tesla 系列)的核心數與時脈比較，CPU 的核心數遠遠小於 GPU 核心數，雖然時脈比 GPU 快但考慮所有核心一起運作也無法跟 GPU 的運算效率相比。

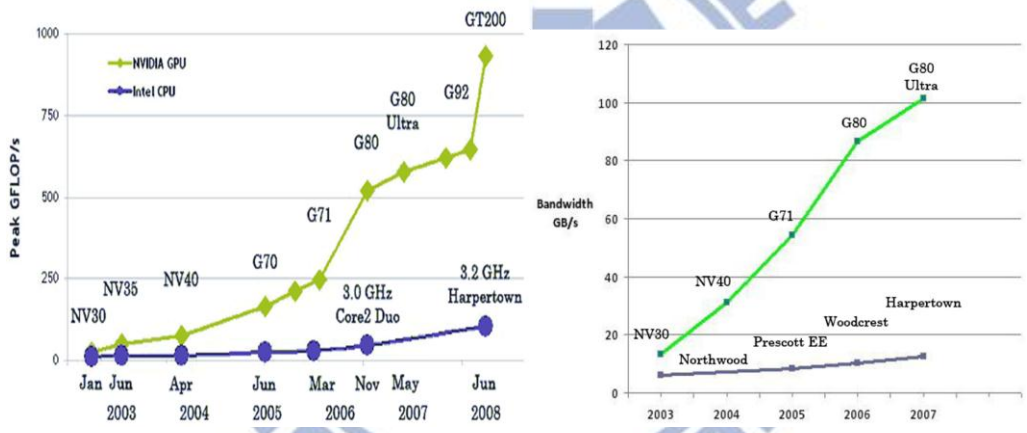


圖 1.2.1 CPU 和 GPU 比較圖：(左)年份對每秒浮點運算效率圖；(右)年份對記憶體寬頻圖。資料來源：文獻[5]與文獻[8]。

由圖 1.2.1(左)得知，CPU 和 GPU 浮點運算速度隨著時間漸漸拉大，因為 GPU 有高度平行運算能力，而 CPU 一開始是以單核時脈為研究方向，因此 GPU 比 CPU 擁用更多的電晶體來做數值運算。記憶體寬頻意思是在資料傳遞時的傳送速度，由圖 1.2.1(右)得知兩者差異隨著時間年年劇增，因為 GPU 可執行多顆核心，所有核心同時將資料傳遞到記憶體，因此在記憶體存取的過程中，可節省更多的時間。

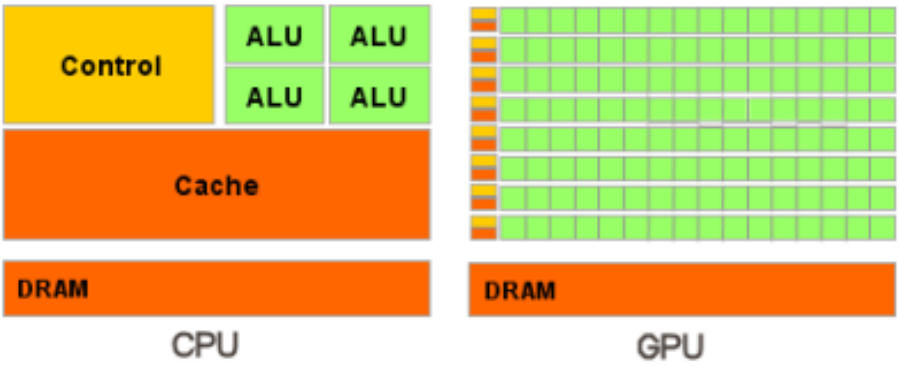


圖 1.2.2 CPU 和 GPU 硬體運算架構圖：一個核心擁有一個 ALU。

資料來源：文獻[8]。

GPU 相對於 CPU 在運算速度及記憶體寬頻都有明顯優勢，金錢成本與功率消耗不需花費太多，能以低成本且少量的時間完成任務。GPU 擁有高度平行性，通過控制處理單元與儲存器控制單元個數提高運算速度與記憶體寬頻，由圖 1.2.2 知 GPU 內有許多的 ALU(Arithmetic Logical Unit，中譯為算術邏輯單元)，雖然每個 ALU 不如 CPU 的運算速度快，但 GPU 可運用平行性來提高整體的執行效率，同時對每個核心運算。當我們做運算時，每個 ALU 都必須利用記憶體儲存與讀取資料，而 GPU 所花費記憶體大小為：

GPU 花費記憶體大小=(正在執行程式的核心數)x(單核花費記憶體容量)　　(1.2.2)

由上式得知 GPU 運行程式時會比 CPU 花費記憶體要多，因此在做程式撰寫有時必須做些取捨。

隨著平行化技術被受重視與 GPU 發展快速，原本 GPU 並不支援資料運算，只能用於 DirectX 或 OpenGL 等相關影像處理函式庫[9]。為了讓 GPU 能運用在廣泛的計算，Nvidia 公司在 2007 年開發了 CUDA 方案[10]，是一種在平行架構下的軟體，可控制 GPU 運算方式，使設計者能利用 C/C++函式庫撰寫平行化的程式；和 Nvidia 共同合作的公司 The Portland Group(PGI)，更是看重高效率運算專業人士所用的重要語言「Fortran」，發明 CUDA 可寫入 Fortran 的資料庫，讓使用者可利用 Fortran 撰寫平行化的程式[11]。

1.3 研究目的

考慮使用單核 CPU 數值計算電子－電洞間直接庫侖或交換能的六重積分，需耗費龐大的計算時間，才能使得誤差在可以接受的範圍內。藉由 GPU 對浮點運算能力的優點，將可加速數值模擬運算，讓時間縮短 10 倍以上[9]。透過此研究，未來計算直接庫侖或交換能時，可以更快速、更有效率的得到數值結果，讓數值分析更加容易。另一個重點則是關於電子－電洞間交換能的驗證，電子－電洞間交換能分成短程作用與長程作用，其中長程作用積分必須考慮不能計算到兩粒子間的短程作用，因此要考慮到兩者邊界的長度設定，並使其數值運算結果與解析解誤差在合理的範圍內。

1.4 論文架構

第一章探討研究背景與動機、GPU 相關背景、比較 CPU 與 GPU 的差異；第二章簡單敘述 GPU 和 CUDA 基本架構、GPU 平行運算技術相關理論、GPU 與 CUDA 的溝通方式；第三章介紹庫侖和交換能理論基礎、定義短程交換能和長程交換能的界線、可使用的數值積分方式以及選擇的積分方法分析、使用單核 CPU 與 GPU 的簡單積分計算比對及程式撰寫流程(庫侖、交換能)；第四章比較單核 CPU 和 GPU 平行運算庫侖交互作用的數值(不同格點間距)與時間結果、計算直接庫侖作用(不同格點、不同量子點大小)與長程偶極－偶極交換能數值解(不同格點、不同 a_0、不同量子點大小)與解析解比對以及結果與討論；第五章將所有章節總結。

第二章、CUDA 基礎架構

以下是這章節介紹的一些 CPU 或 GPU 的電腦專有名詞，文中在介紹專有名詞時可能會以英文型式來表示，因此為了方便對照中英文而做出表 2.1：

表 2.1 電腦專用術語中英文對照表。

中文	英文
中央處理器	*Central Processing Unit*(*CPU*)
繪圖處理器	Graphics Processing Unit(GPU)
統一計算架構	Compute Unified Device Architecture(CUDA)
主機	Host
設備	Device
線程處理器群集	Texture Processing Clusters(TPC)
流多處理器	Streaming Multiprocessor(SM)
流處理器	Streaming Processor(SP)
網格	grid
區塊	block
執行緒	thread
全域記憶體	global memory
常數記憶體	constant memory
共享記憶體	shared memory
暫存器	register
材質記憶體	texture memory

電腦執行程式時，必須考慮硬體及軟體的搭配，圖 2.1 表示執行流程圖。資訊 1 表計算需要由外部輸入的資料，資訊 2 表計算完輸出的資料，CUDA 為 Host 端與 Device 端互傳的資料庫。表 2.2 為執行程式的工具：

表 2.2 本論文計算程式使用的軟硬體名稱。

軟體(程式語言)	硬體(運算)	資料庫
PGI Fortran	GPU	CUDA

接下來要介紹 GPU 硬體與 CUDA 軟體相關資訊。

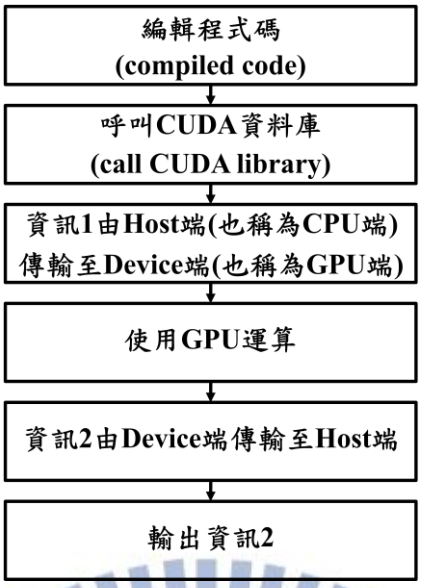


圖 2.1 執行程式流程示意圖。

2.1 平行運算的概念

一般 CPU 運算時，通常都是條列式的進行運算程式，而 GPU 運算時，是利用平行運算使計算時間縮短。如圖 2.1.1 所示，六個人同時做一件事，每個人負責一個步驟，完成的時間遠比一個人做一件事快很多，平行運算與上述概念差不多。一般來說，GPU 比 CPU 核心數多，表示可以使用較多的電晶體同時計算同一個方程式，因此 GPU 計算時間會很快。在硬體方面，GPU 是利用 SP 來平行運算，而軟體方面則是使用 thread，接下來會討論平行運算在軟硬體內部架構是如何區分的，以及要如何將軟硬體做連結。

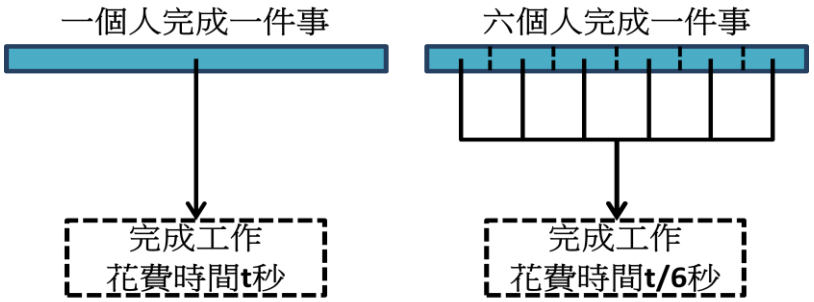


圖 2.1.1 一個人與六個人完成工作花費時間示意圖。

2.2 GPU 硬體

2.2.1 硬體架構

圖 2.2.1.1 表示 GPU 內部架構；GPU 內會先區分數個 TPC，由 GPU 型號決定 TPC 個數；在 TPC 內為數個 SM 加上一些其他單元組成，由 GPU 型號決定 SM 個數；SM 內又分成數個 SP，SP 數量通常是 8 個一組；SP 是最基本的處理單位，也稱為核心，所以 GPU 內部有許多 SP 在做平行運算。

圖 2.2.1.1 GT200 架構示意圖：GT200 共有 240 個 SP，每個 SM 有 8 個 SP，每個 TPC 有 3 個 SM。資料來源：文獻[12]。

對於不同型號 GPU，內部架構也有些許不同，表 2.2.1.1 是利用 PGI Fortran 附加程式(deviceQuery.cuf)測試 GPU 型號「Tesla C2050」的結果：

表 2.2.1.1 利用 deviceQuery.cuf 測試 GPU 結果。

型號	TPC	SM	SP
Tesla C2050/2070	14	56	448

與圖 2.2.1.1 中 GPU 型號「GT200」相比，會發現 SP 個數、TPC 組成 SM 個數會不一樣，可藉此比較兩者差異。

2.2.2 硬體間傳輸速率

當資料透過電腦做傳遞時，會發生三種情形如表 2.2.2.1：

表 2.2.2.1 程式運算 Host 和 Device 端傳遞可能的傳遞情況。

傳遞方式	示意
1. Host 端內傳遞	Host端(CPU) ↔ Host端(CPU)
2. Device 端內傳遞	Device端(GPU) ↔ Device端(GPU)
3. Host 端與 Device 端互傳	Host端(CPU) ↔ Device端(GPU)

以上三種情形記憶體寬頻皆不相同，因此我們必須考量將資料如何做傳遞會最恰當。當資料在 Host 端傳遞時，過程是使用 CPU 將資料讀取或計算出來，之後再將資料傳送到記憶體儲存，記憶體寬頻大約為 5～10GB/s；在 Device 端傳遞時，由於它是利用平行性來增快傳輸速率，因此會比 Host 端傳遞來的快，記憶體寬頻大約為 70～100GB/s；而考慮 Host 端與 Device 端互傳時，記憶體寬頻取決於主機板與 GPU 的插槽，例如 GPU 型號「Tesla C2050」使用的插槽為 PCI-E Gen.2 x16，記憶體寬頻大約 1～5GB/s。

利用 PGI Fortran 附加程式(bandwidthTest.cuf)，測試如表 2.2.2.2 提供的硬體配備：

表 2.2.2.2 執行 bandwidthTest.cuf 程式的電腦設備。

	主機板	CPU	GPU
型號	HP ML350 G6	IntelXeon E5520 2.26Ghz	Tesla C2050

測試結果如表 2.2.2.3：

表 2.2.2.3 執行 bandwidthTest.cuf 程式的結果。

Device端(GPU) ↔ Device端(GPU)	71.67432(GB/s)
Host端(CPU) → Device端(GPU)	1.381577(GB/s)
Host端(CPU) → Device端(GPU)	1.502922(GB/s)

在表 2.2.2.3 測試中，CPU 型號是比較舊型的，因此 GPU 記憶體寬頻會被影響而變慢。由上述得知，Host 端與 Device 端互傳記憶體寬頻是最慢的，而 Device 端記憶體寬頻是最快的。因此我們在做運算時，最理想的狀況就是把要計算的數值，在 Device 端一次計算完，再傳回 Host 端輸出我們要的資訊。

2.3 CUDA 程式

2.3.1 thread、block 與 grid

由 2.2.1 小節得知，硬體架構分為 TPC、SM 以及 SP，而軟體則是分為 grid、block 與 thread 三個部分。如圖 2.3.1.1 所示，grid 包含所有的 block，表示將要執行的任務(kernel)，程式會把它直接丟給 GPU 作運算，等任務完成後才會執行下一個任務，而 CUDA 並沒有 TPC 這層架構；block 相當於 SM，內部包含很多 thread，主要目的在於建構 block 內所有 thread 的溝通，數量可自行設定；thread 相當於 SP，是軟體內的最小單位，主要功能是將程式做平行運算，數量可自行設定。

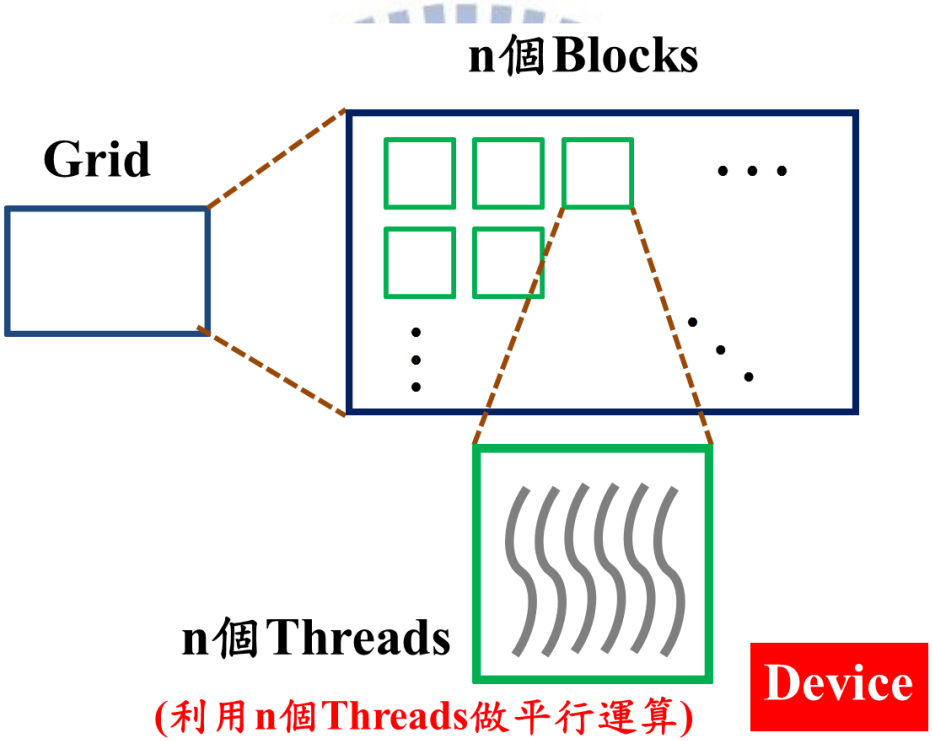


圖 2.3.1.1 CUDA 架構示意圖。

本節介紹的是寫程式必備的觀念，由於本論文所修改的程式皆為迴圈所構成的，這部分的說明都是以附錄 C 為主要範例，而此範例的計算式子如下：

$$SUM = \sum_{i=1}^{n_x} 1 \qquad (2.3.1.1)$$

，程式中定義每個 grid 內 block 個數為 3 個，而每個 block 內 thread 個數為 4 個因此整個 grid 內擁有 thread 總個數為 3x4=12 個，表示我們將(2.3.1.1)式切割成 12 等份來計算且令迴圈數(n_x)為 36，如圖 2.3.1.2 所示。

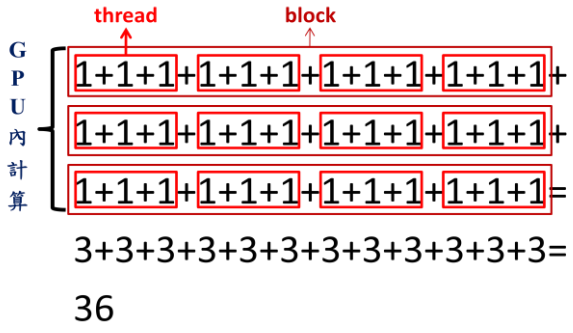
單核 CPU	GPU
1+1+1+1+1+1+1+1+1+1+1+1+ 1+1+1+1+1+1+1+1+1+1+1+1+ 1+1+1+1+1+1+1+1+1+1+1+1= 36	

圖 2.3.1.2 使用單核 CPU 與 GPU 的計算方式：迴圈數(n_x)為 36、每個 grid 內 block 個數為 3 個、每個 block 內 thread 個數為 4 個。

我們將附錄 C 的程式以軍隊來比喻，如圖 2.3.1.3。軍隊[相當於 CUDA]發放給連隊一個任務[相當於 grid]，而連隊中共有 3 個班[相當於 block]，每個班有 4 個小兵[相當於 thread]，所以整個連隊共有 3x4=12 個小兵可配任務，因此可將任務均分成 12 等分分配給每個小兵執行。

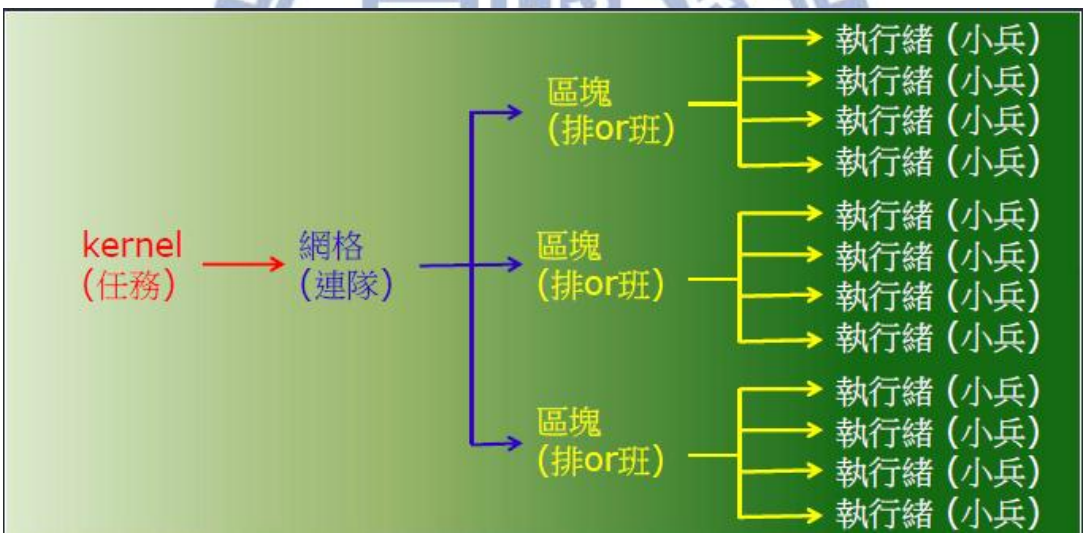


圖 2.3.1.3 CUDA 架構範例說明。

資料來源：文獻[12]。

在程式中定義參數 griddx 為每個 grid 內 block 個數、參數 tPBdx 為每個 block 內 thread 個數，程式中寫法如圖 2.3.1.4，此圖比較 CPU 與 GPU 的寫法，假設 griddx=3、tPBdx=4，所以整個 grid 內共擁有 thread 總個數為 3x4=12 個，而更詳細說明請參考附錄 B 與 C。

單核 CPU

```
  PROGRAM one_loop
    IMPLICIT NONE

    INTEGER,PARAMETER :: nx=2*10**5
    DOUBLE PRECISION  :: sums
    INTEGER           :: time_begin,time_end

    sums=0.0D0

    CALL SYSTEM_CLOCK(time_begin)
    CALL loop_CPU(sums,nx)
    CALL SYSTEM_CLOCK(time_end)

    WRITE(*,*) 'SUM=',sums
    WRITE(*,*) 'cpu_time=',
&   (time_end-time_begin)*1D-6,'sec'

  END PROGRAM one_loop
```

GPU[Host 端]

```
PROGRAM one_loop
  USE CUDAFOR
  USE loop_m
  IMPLICIT NONE

  INTEGER,PARAMETER        :: nx=2*10**5
  INTEGER,PARAMETER        :: griddx=3,tPBdx=4
  DOUBLE PRECISION         :: a(griddx*tPBdx)
  DOUBLE PRECISION,DEVICE  :: a_d(griddx*tPBdx)
  DOUBLE PRECISION         :: sums
  INTEGER                  :: i
  INTEGER                  :: istat
  INTEGER                  :: time_begin,time_end

  a=0.0D0

  CALL SYSTEM_CLOCK(time_begin)
  a_d=a
  CALL loop_GPU<<<griddx,tPBdx>>>(a_d,nx)
  a=a_d
  CALL SYSTEM_CLOCK(time_end)

  istat=CUDAFREE(a_d)

  sums=0.0D0

  DO i=1,griddx*tPBdx
  sums=sums+a(i)
  ENDDO

  WRITE(*,*) 'SUM=',sums
  WRITE(*,*) 'gpu_time=', &
  (time_end-time_begin)*10D-6,'sec'

END PROGRAM one_loop
```

定義每個grid內block個數、每個block內thread個數

1.宣告Host與Device端互相傳遞的參數

2.Host端與Device端參數互傳

3.呼叫kernel(CUDA副程式)

4.釋放device記憶體

5.所有thread計算數值總加起來

單核 CPU

```
SUBROUTINE loop_CPU(sums,nx)
  IMPLICIT NONE

  INTEGER           :: nx
  DOUBLE PRECISION  :: sums
  INTEGER           :: l

  DO l=1,nx
  sums=sums+1.0D0
  ENDDO

END SUBROUTINE loop_CPU
```

GPU[Device 端(kernel)]

1.使用MODULE指令及定義CUDA副程式

```
MODULE loop_m
CONTAINS
  ATTRIBUTES(GLOBAL) SUBROUTINE loop_GPU(a_d,nx)
    IMPLICIT NONE

    INTEGER,VALUE           :: nx
    DOUBLE PRECISION        :: a_d(:)
    INTEGER                 :: i
    INTEGER                 :: l

    i=(BLOCKIDX%X-1)*BLOCKDIM%X+THREADIDX%X

    DO l=i,nx,BLOCKDIM%X*GRIDDIM%X
    a_d(i)=a_d(i)+1.0D0
    ENDDO

  END SUBROUTINE loop_gpu
END MODULE loop_m
```

2.宣告接收Host端傳遞的Device端參數

3.將所有thread編號

4.命令每個thread運算的方式

圖 2.3.1.4 一般 Fortran 修改成 CUDA Fortran 迴圈範例：詳細說明請參考附錄 B 與 C。

Thread 與 Block 是屬於三維結構的，因此 Grid 內 block 總個數與每個 block 內 thread 總個數是所有維度乘積後的值，而 CUDA 設定 thread、block 總個數也有數目上的限制，這裡我們利用 PGI Fortran 附加的程式(bandwidthTest.cuf）來做測試，表 2.3.1.1 與表 2.3.1.2 為測試的結果：

(以型號「Tesla C2050」為範例)

表 2.3.1.1 利用 bandwidthTest.cuf 程式測試結果一。

	X 維度	Y 維度	Z 維度
最大 block 個數	65535	65535	65535
最大 thread 個數	1024	1024	64

表 2.3.1.2 利用 bandwidthTest.cuf 程式測試結果二。

所有維度乘積後最大 block 總個數	65535
所有維度乘積後最大 thread 總個數	1024

2.3.2 記憶體配置

在軟體內可分為 thread、block 與 grid，而每個部分所使用的記憶體又有所區分，因此可以將 GPU 的記憶體分成：

1. 全域記憶體(global memory)
2. 常數記憶體(constant memory)
3. 共享記憶體(shared memory)
4. 暫存器(register）
5. 材質記憶體(texture memory，目前沒使用到)

以上記憶體分佈如圖 2.3.2.1 表示 CUDA 內分別對應到的記憶體。

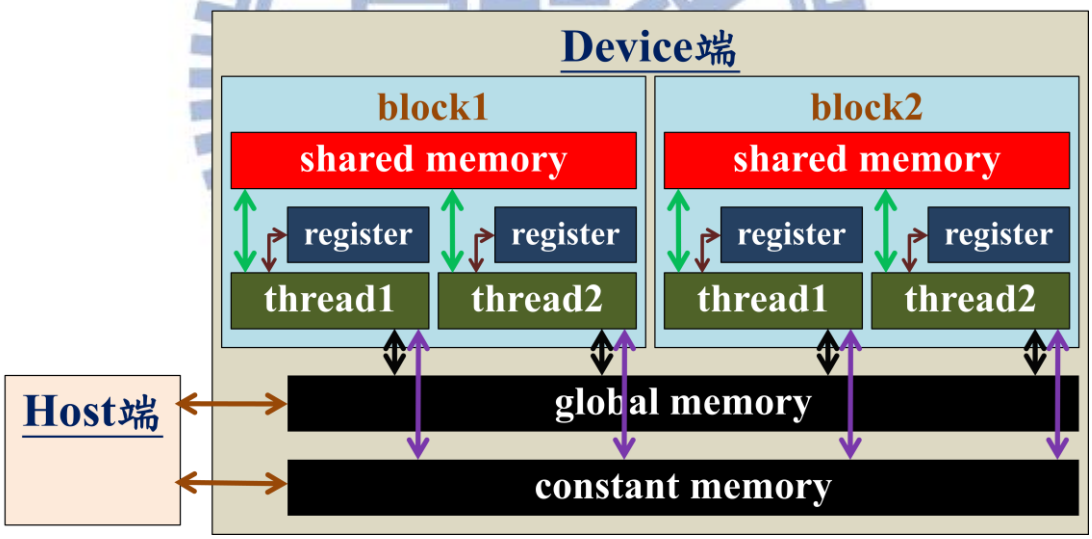


圖 2.3.2.1 CUDA 內對應到的記憶體。

global memory 是用來存取由 Host 端傳進來的所有資訊，當存取完 Host 端的資訊時，之後會分配到各個記憶體內，生命期為一個 grid 內所有 thread 都計算完的時間，global memory 記憶體寬頻是最慢的；constant memory 主要是存取由 Host 端傳遞到 Device 端的固定資訊，容量大約 64k，生命期與 global memory 一樣，記憶體寬頻比 shared memory 快；shared memory 用來存取共享資料，讓

GPU 在計算時可互相傳遞資訊，每個 block 都擁有 shared memory，容量大約 16k，使 block 內的 thread 互相溝通，而生命期為一個 block 內所有 thread 計算完的時間，記憶體寬頻比 global memory 快；register 專門存取正在做計算的暫存資料，每個 thread 擁有數個 register，不同型號 GPU 擁有數量都不一樣，當 thread 計算完後，儲存的資訊會自動被刪除，因此生命期只有在 thread 內計算完的時間，對於傳送速率而言，花費時間幾乎零秒就傳完了，而 thread 內的 register 記憶體空間不足以存取所有資訊時，thread 會將剩餘的資訊存在 global memory 內，因此計算速度會變慢，相當於我們使用 CPU 計算程式，當電腦記憶體不足以存取所有資訊時，會將剩餘的資訊存到虛擬記憶體內，此時計算速度會變得非常慢；texture memory 是 GPU 還沒用於計算時的產物，原本用於儲存圖形的資訊，但目前我們作運算並沒使用到，在此無相關討論。

表 2.3.2.1 Device 端記憶體相關資訊。

	生命期	存取範圍	記憶體寬頻	Device 端存取
register	區塊	thread	即時	讀/寫
shared memory	區塊	block	140~200GB/s	讀/寫
constant memory	程式	grid	200~300GB/s	讀
global memory	程式	grid	70~100GB/s	讀/寫

2.4 GPU 與 CUDA 的溝通

在 CUDA 執行程式時，電腦會把 grid 直接丟給 GPU，再將 grid 內 block 傳送至 GPU 內 SM 執行，如圖 2.4.1 示意圖；硬體把傳送過來的 block 當作基本單位，而 block 內 thread 又會以 warp 分組來執行程式，目前 CUDA 的 1 個 warp 大小等於 32 個 thread。

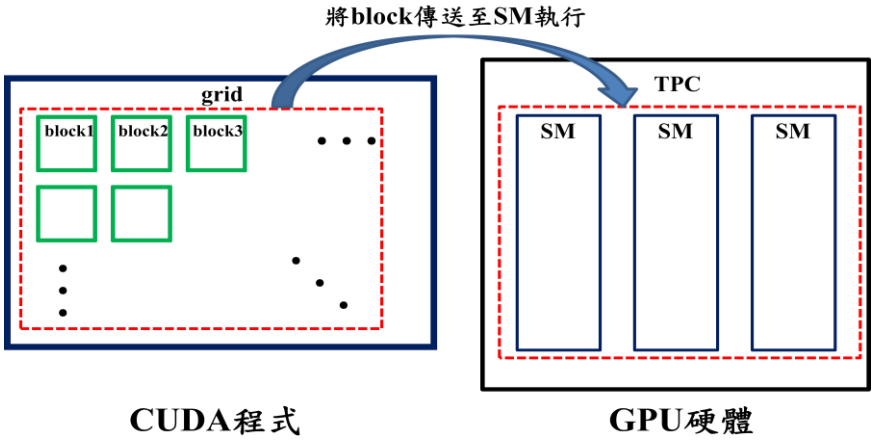


圖 2.4.1 CUDA 傳送 block 到 GPU 內的 SM 示意圖。

由上述得知 CUDA 將 block 傳送至 SM，當 block 數量大於 SM 時，GPU 會先執行與 SM 個數相符的 block 數量，其餘的 block 會閒置，等到這些 block 執行完成後或等待時[例如存取 global memory 需要花費許多時間]，才會將閒置的 block 傳送至 SM 做計算，至於先前等待中的 block 會變成閒置狀態繼續等下一輪執行；block 可分成很多個 warp，SM 會先執行一個 warp，其餘的 warp 閒置，等這一個 warp 執行完後才會執行下一個 warp，圖 2.4.2 表示觀察 1 個 SM 執行 block 與 warp 的順序；warp 有 32 個 thread，是利用 SM 內的 SP 運行程式，SM 通常有 8 個 SP，因此他會同時計算 warp 內 8 個 thread，計算完成後才會繼續計算其它 thread。

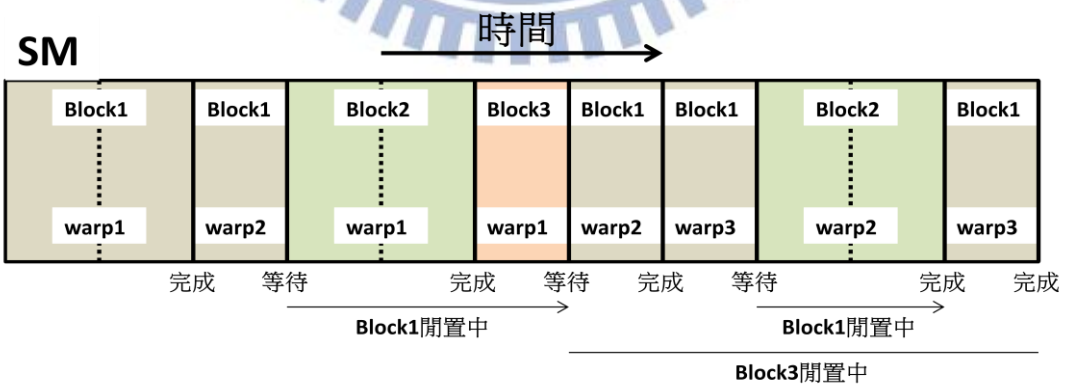


圖 2.4.2 warp 在 1 個 SM 運作情形：假設每個 warp 需要運作的時間都是 2 個週期。

當 1 個 warp 沒有滿 32 個 thread 時，SM 還是一次只會執行 1 個 warp，這會使某幾個 SP 閒置著，造成資源上的浪費。如圖 2.4.3 範例，Block 內有 99 個 thread，

此時 Thread 可分為 4 個 warp，分別是 32、32、32、3 個 thread，發現 warp4 只有 3 個 Thread，而利用 SM 執行這個 warp 時，會造成 5 個 SP 閒置，使程式無法最佳化，因此在定義 block 內 thread 個數時，最好設定能被 32 整除的數目。

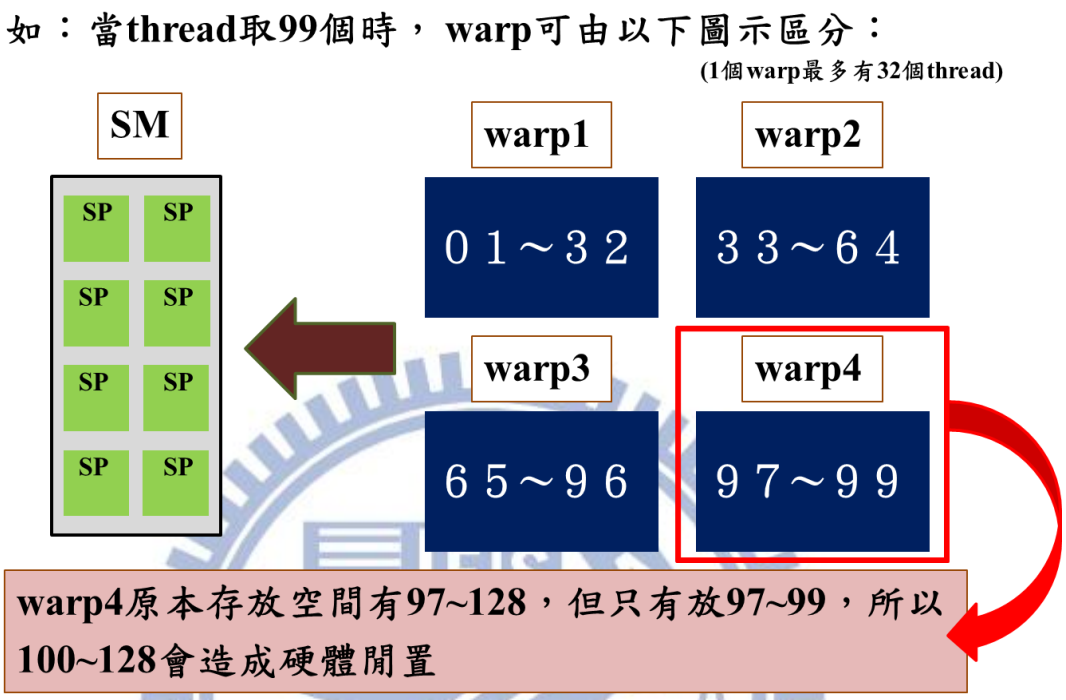


圖 2.4.3 warp 傳送至 SM 計算範例，當 warp4 不足 32 個 Thread 會造成資源浪費。

第三章、庫侖交互作用理論及其數值方法測試

3.1 積分方法

我們積分通常會將式子離散化，其方法有分很多種，本論文將要介紹三種方法，分別是矩形法、梯形法與辛普森法。

1.矩形法

如圖 3.1.1 所示，考慮區間 $x_a \le x \le x_b$，可將此切割 N 個寬度為 Δx_i 的子區間，則各子區間的座標為

$$x_i = x_a + (i-1)\times\Delta x_i \text{，} i = 1,2,3,\cdots,n \tag{3.1.1}$$

將定積分 $\int_{x_a}^{x_b} f(x)dx$ 近似為 $x = x_a \sim x = x_b$ 間的矩形總面積，以 $f(x_i)$ 當作矩形的高，可表示為

$$\int_{x_a}^{x_b} f(x)dx \approx \lim_{\Delta x_i \to 0} \sum_{i=1}^{N} f(x_a + (i-1)\Delta x_i)\Delta x_i \tag{3.1.2}$$

隨著格點數變多，矩形的寬度會越來越窄，而矩形總面積則越來越接近所求的值。

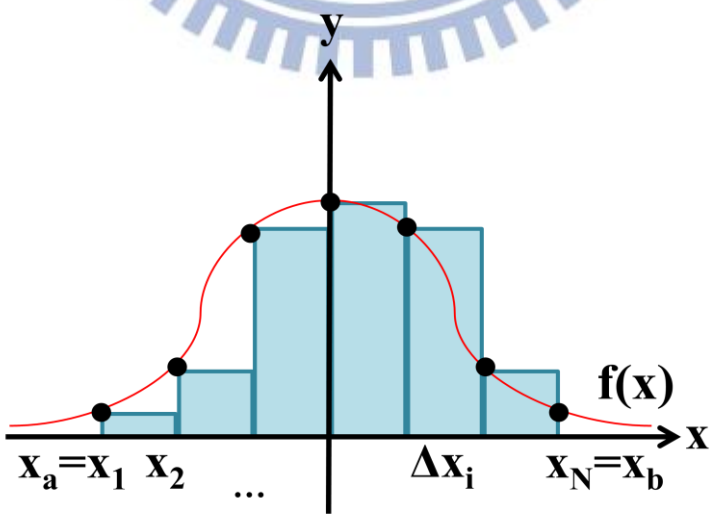


圖 3.1.1 矩形法求定積分示意圖。

考慮均勻格點 $\Delta x_i = \Delta x = (x_b - x_a)/N$，則(3.1.2)式可改寫為

$$\int_{x_a}^{x_b} f(x)dx \approx \lim_{\Delta x\to 0}\sum_{i=1}^{N} f\left(x_a+(i-1)\Delta x\right)\Delta x \tag{3.1.3}$$

將(3.1.3)式推廣到三維的狀況，則數學形式可表示

$$\int_{x_a}^{x_b}\int_{y_a}^{y_b}\int_{z_a}^{z_b} f(x,y,z)dxdydz \approx \lim_{\Delta x,\Delta y,\Delta z\to 0}\sum_{i}^{N_x}\sum_{j}^{N_y}\sum_{k}^{N_z} f\left(x_i,y_j,z_k\right)\Delta x\Delta y\Delta z \tag{3.1.4}$$

其中

$$x_i = x_a+(i-1)\times\Delta x \text{ 、 } y_j = y_a+(j-1)\times\Delta y \text{ 、 } z_k = z_a+(k-1)\times\Delta z$$

$$\Delta x=\frac{x_a-x_b}{N_x} \text{ 、 } \Delta y=\frac{y_a-y_b}{N_y} \text{ 、 } \Delta z=\frac{z_a-z_b}{N_z}$$

2.梯形法

利用(3.1.1)式將定積分 $\int_{x_a}^{x_b} f(x)dx$ 近似為 $x=x_a \sim x=x_b$ 間的梯形總面積，如圖3.1.2。而 $f(x_i)$ 與 $f(x_{i+1})$ 分別當作梯形的上底與下底，可表示為

$$\int_{x_a}^{x_b} f(x)dx \approx \lim_{\Delta x_i\to 0}\sum_{i=1}^{N}\frac{\left[f\left(x_a+(i-1)\Delta x_i\right)+f\left(x_a+i\Delta x_{i+1}\right)\right]}{2}\Delta x_i \tag{3.1.5}$$

格點數變多，寬度越來越窄，則梯形總面積則越來越接近所求的值。

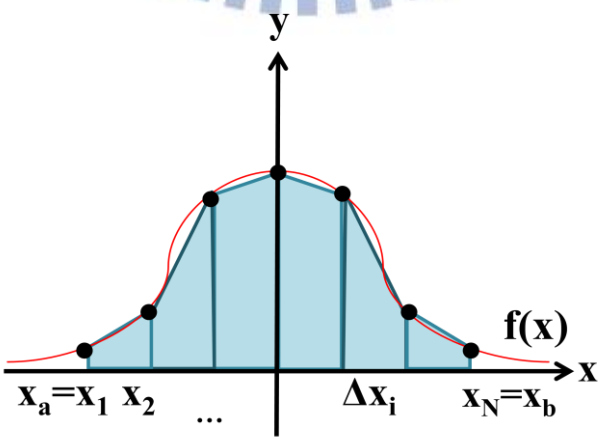


圖 3.1.2 梯形法求定積分示意圖。

考慮均勻格點 $\Delta x_i=\Delta x=(x_b-x_a)/N$，則(3.1.5)式可改寫為

$$\int_{x_a}^{x_b} f(x)\,dx \approx \lim_{\Delta x\to 0}\sum_{i=1}^{N}\frac{\left[f\left(x_a+(i-1)\Delta x\right)+f\left(x_a+i\Delta x\right)\right]}{2}\Delta x \tag{3.1.6}$$

將(3.1.6)式推廣到三維的狀況，則數學形式可表示

$$\int_{x_a}^{x_b}\int_{y_a}^{y_b}\int_{z_a}^{z_b} f(x,y,z)\,dxdydz \approx$$
$$\lim_{\Delta x,\Delta y,\Delta z\to 0}\sum_{i}^{N_x}\sum_{j}^{N_y}\sum_{k}^{N_z}\begin{bmatrix} f\left(x_i,y_j,z_k\right)+f\left(x_i,y_j,z_{k+1}\right)+\\ f\left(x_i,y_{j+1},z_k\right)+f\left(x_i,y_{j+1},z_{k+1}\right)+\\ f\left(x_{i+1},y_j,z_k\right)+f\left(x_{i+1},y_j,z_{k+1}\right)+\\ f\left(x_{i+1},y_{j+1},z_k\right)+f\left(x_{i+1},y_{j+1},z_{k+1}\right)\end{bmatrix}\frac{\Delta x\Delta y\Delta z}{8} \tag{3.1.7}$$

其中

$$x_i = x_a+(i-1)\times\Delta x \text{ 、 } y_j = y_a+(j-1)\times\Delta y \text{ 、 } z_k = z_a+(k-1)\times\Delta z$$

$$\Delta x=\frac{x_a-x_b}{N_x} \text{ 、 } \Delta y=\frac{y_a-y_b}{N_y} \text{ 、 } \Delta z=\frac{z_a-z_b}{N_z}$$

由上面介紹的積分方法可知：矩形法的運算式子最簡單，而梯形法的運算式子較為複雜。當運算式子越複雜時，電腦會耗費較多記憶體相對的時間也會比較久，使用 GPU 運算影響速率更為顯著(詳細 2.3.2 節)，因此矩形法運算速率會大於梯形法，而接下來的小節要比較矩形法與梯形法數值結果。

3.2 數值積分測試

由 3.1 小節可知數值運算方式，接下來將分析兩個部分，分別是

1.分析格點數：證明格點數越多時越數值越準確。
2.採取的積分方法：比較矩形法與梯形法的收斂範圍。

3.2.1 一維積分

以下式子為一維積分的形式

$$\int f(x)dx \tag{3.2.1.1}$$

我們將以一個積分式子當作範例來做分析，採取的積分式子如下

$$F=\int_{0}^{\frac{5}{2}\pi} x\cos(x)\,dx \tag{3.2.1.2}$$

而(3.2.1.2)式積分後的解析解為

$$F=\frac{5}{2}\pi-1 \tag{3.2.1.3}$$

利用矩形法與梯形法將積分離散化且取均勻格點(格點數取 N_x 個)後，其形式為

$$F\approx F'=\sum_{i=1}^{N_x} x_i\cos(x_i)\Delta x \tag{3.2.1.4}$$

$$F\approx F'=\sum_{i=1}^{N_x}\frac{\left[x_i\cos(x_i)+x_{i+1}\cos(x_{i+1})\right]}{2}\Delta x \tag{3.2.1.5}$$

其中(3.2.1.4)式為矩形法、(3.2.1.5)式為梯形法。將(3.2.1.4)式與(3.2.1.5)式利用 Fortran 程式做運算可畫出收斂結果，如圖 3.2.1.1 與圖 3.2.1.2。

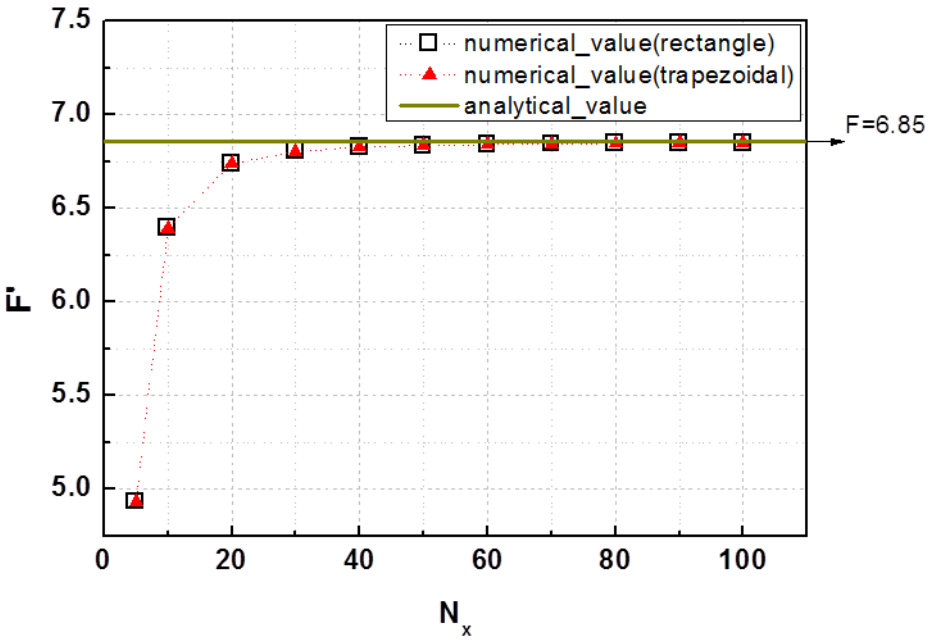


圖 3.2.1.1 利用矩形法(黑色方形中空)與梯形法(紅色三角形)算出的數值收斂結果。

由圖 3.2.1.1 可看出，不管是矩形法或梯形法，格點數取越多數值差異越小，我們以矩形法當作範例畫出圖 3.2.1.3 與圖 3.2.1.4 證明此結果。圖中格點數取 5 個時會發現積分面積有一部分沒被計算到，而格點數取 20 個時積分沒被計算到的面積相對於取 5 個格點就少很多了，因此格點數取越多，計算數值會越準確。

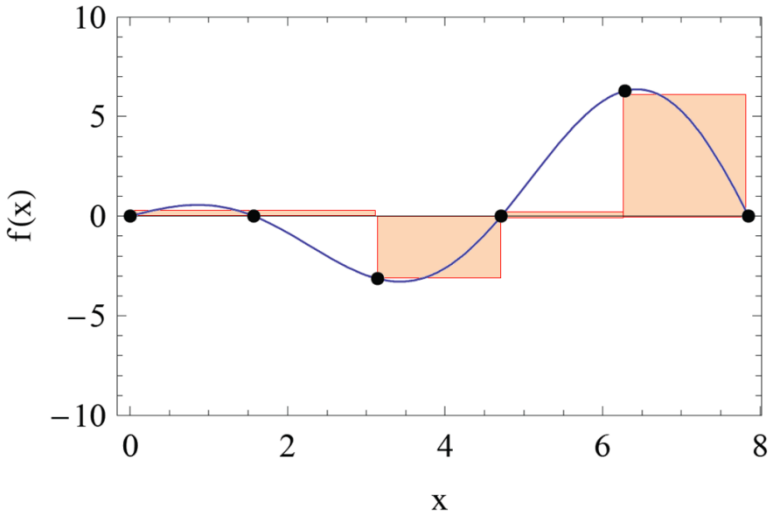


圖 3.2.1.2 利用矩形法將積分離散化的示意圖($N_x = 5$)。

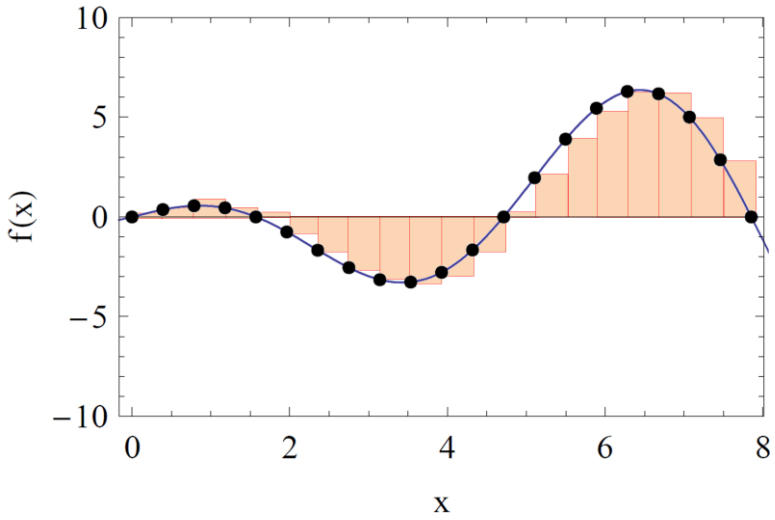


圖 3.2.1.3 利用矩形法將積分離散化的示意圖($N_x = 20$)。

由圖 3.2.1.1 比較矩形法與梯形法的數值差異，可由誤差公式 $error = \frac{|解析解 - 數值解|}{解析解} \times 100\%$ 計算出圖 3.2.1.4。

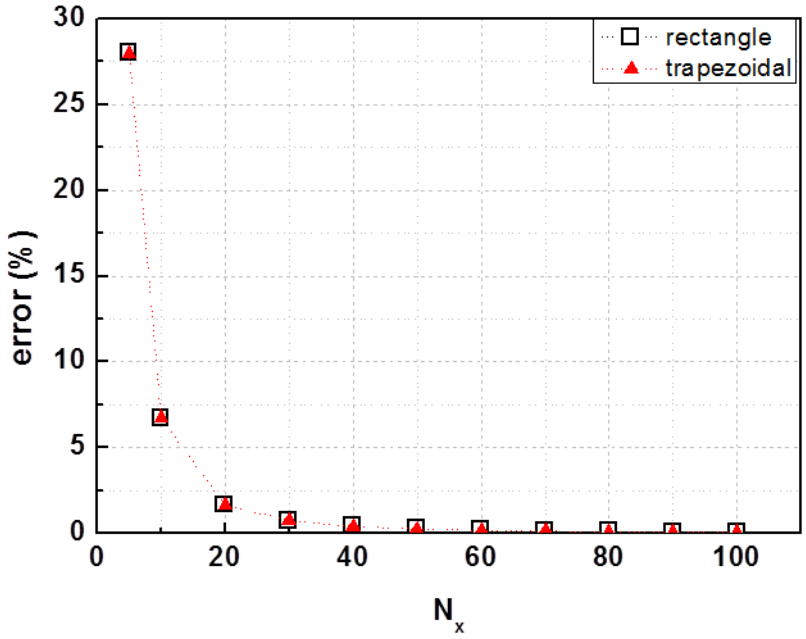


圖 3.2.1.4 利用矩形法和梯形法算出的結果與解析解比較後的誤差值。

由圖 3.2.1.4 可知矩形法與梯形法在一維積分的數值結果差異不大，且兩者數值結果很快就收斂(取 N_x =20)。

3.2.2 三維積分

由 3.2.1 小節可知，一維積分的數值收斂結果是可接受的，接著將要分析三維積分，以下式子為三維積分形式

$$\iiint f(x,y,z)dxdydz \tag{3.2.2.1}$$

而我們選擇三維積分範例式子如下

$$F=\int_0^{\frac{5}{2}\pi}\int_0^{\frac{5}{2}\pi}\int_0^{\frac{5}{2}\pi} xyz\cos(x)\cos(y)\cos(z)\,dxdydz \tag{3.2.2.2}$$

(3.2.1.2)式積分後的解析解為

$$F=\left(\frac{5}{2}\pi-1\right)^3 \tag{3.2.2.3}$$

利用矩形法與梯形法將積分離散化且取均勻格點(每個維度的格點數分別取 N_x、N_y、N_z 個)後，其形式為

$$F\approx F'=\sum_{k=1}^{N_z}\sum_{j=1}^{N_y}\sum_{i=1}^{N_x} x_i y_j z_k\cos(x_i)\cos(y_j)\cos(z_k)\Delta x\Delta y\Delta z \tag{3.2.2.4}$$

$$F\approx F'=\sum_{k=1}^{N_z}\sum_{j=1}^{N_y}\sum_{i=1}^{N_x}\begin{bmatrix} x_i y_j z_k\cos(x_i)\cos(y_j)\cos(z_k)+\\ x_i y_j z_{k+1}\cos(x_i)\cos(y_j)\cos(z_{k+1})\\ x_i y_{j+1} z_k\cos(x_i)\cos(y_{j+1})\cos(z_k)+\\ x_i y_{j+1} z_{k+1}\cos(x_i)\cos(y_{j+1})\cos(z_{k+1})+\\ x_{i+1} y_j z_k\cos(x_{i+1})\cos(y_j)\cos(z_k)+\\ x_{i+1} y_j z_{k+1}\cos(x_{i+1})\cos(y_j)\cos(z_{k+1})+\\ x_{i+1} y_{j+1} z_k\cos(x_{i+1})\cos(y_{j+1})\cos(z_k)+\\ x_{i+1} y_{j+1} z_{k+1}\cos(x_{i+1})\cos(y_{j+1})\cos(z_{k+1}) \end{bmatrix}\frac{\Delta x\Delta y\Delta z}{8} \tag{3.2.2.5}$$

其中(3.2.2.4)式為矩形法、(3.2.2.5)式為梯形法。將(3.2.2.4)式與(3.2.2.5)式利用 Fortran 程式做運算可畫出收斂結果，計算方式為 $N_x=N_y=N_z$ 如圖 3.2.2.1，其中 $N_{3D}=N_xN_yN_z$。

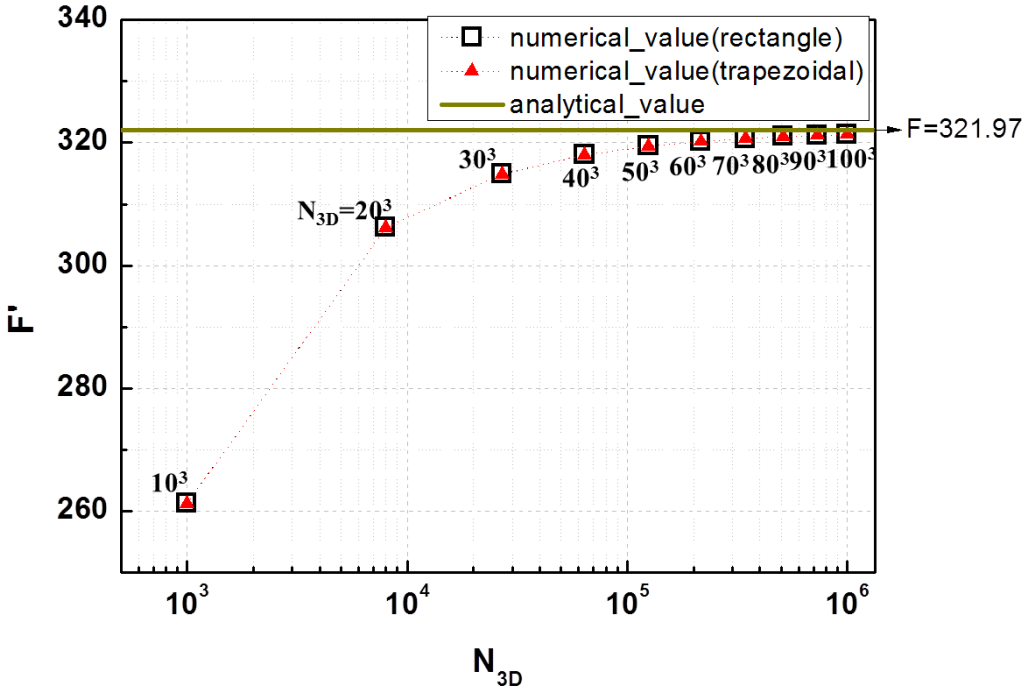


圖 3.2.2.1 利用矩形法(黑色方形中空)與梯形法(紅色三角形)算出的數值收斂結果。

比較(3.2.2.3)式和(3.2.2.4)式與(3.2.2.5)式的數值差異，而此差異可由誤差公式 $error = \frac{|\text{解析解} - \text{數值解}|}{\text{解析解}} \times 100\%$ 計算出圖 3.2.2.2。

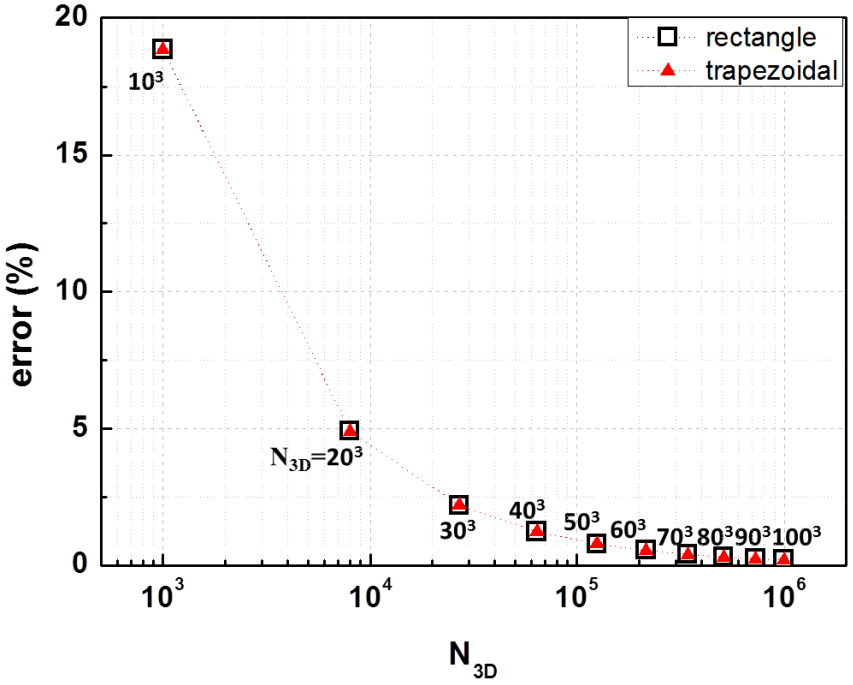
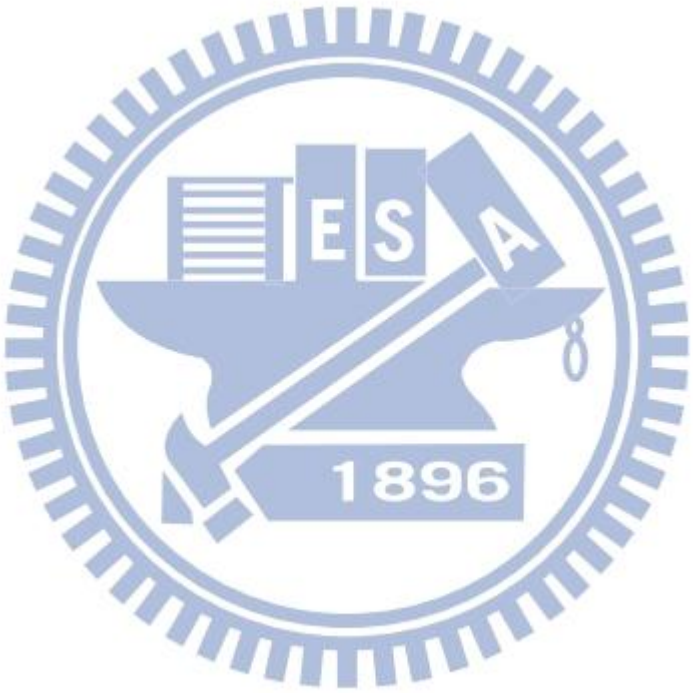


圖 3.2.2.2 利用矩形法和梯形法算出的結果與解析解比較後的誤差值。

由一維與三維積分比較矩形法與梯形法的收斂值，數值結果差異不大。矩形法計算式子比梯形法簡單，所以矩形法在程式的運算速率較快且式子可讀性較清楚，因此我們之後的積分將選擇矩形法來做計算。



3.3 量子點內單一激子理論基礎

3.3.1 單一能帶理論

首先以 single-band model 來描述電子與電洞的行為，再由波包近似法(envelope function approximation)將半導體內電子與電洞波函數寫成

$$\Psi_{i_e}^{e}\left(\vec{r_e}\right)=g_{i_e}^{e}\left(\vec{r_e}\right)u_{s_z}^{e}\left(\vec{r_e}\right) \tag{3.3.1.1}$$

$$\Psi_{j_h}^{h}\left(\vec{r_h}\right)=g_{j_h}^{h}\left(\vec{r_h}\right)u_{j_z}^{h}\left(\vec{r_h}\right) \tag{3.3.1.2}$$

$u_{s_z}^{e}\left(\vec{r_e}\right)$、$u_{j_z}^{h}\left(\vec{r_h}\right)$分別表示電子與電洞的 Bloch's function；$g_{i_e}^{e}\left(\vec{r_e}\right)$、$g_{j_h}^{h}\left(\vec{r_h}\right)$分別表示電子與電洞的波包函數(envelope function)；i_e、j_h分別表示電子與電洞所在的軌域。

由於要計算電子與電洞在半導體量子點的能階能量與波函數，將利用有效質量近似薛丁格方程(Schrödinger equation)及(3.3.1.1)式、(3.3.1.2)式可得

$$\left[\frac{p^2}{2m_0m_e^*}+V_{QD}^{e}\left(\vec{r}_e\right)\right]g_{i_e}^{e}\left(\vec{r}_e\right)=E_{i_e}^{e}g_{i_e}^{e}\left(\vec{r}_e\right) \tag{3.3.1.3}$$

$$\left[\frac{p^2}{2m_0m_{HH,\alpha}^*}+V_{QD}^{HH}\left(\vec{r}_h\right)\right]g_{j_h}^{h}\left(\vec{r}_h\right)=E_{j_h}^{HH}g_{j_h}^{h}\left(\vec{r}_h\right) \tag{3.3.1.4}$$

m_0為電子靜止的質量，m_e^*、$m_{HH,\alpha(=x,y,z)}^*$分別為電子與重電洞的有效質量且重電洞的有效質量與方向有關[1]，p為動量算符，$V_{QD}^{e}\left(\vec{r}_e\right)$為電子在導電帶受量子效應(結構與形狀)侷限的位能，而$V_{QD}^{HH}\left(\vec{r}_h\right)$為重電洞在價電帶受量子效應(結構與形狀)侷限的位能，$E_{i_e}^{e}$、$E_{j_h}^{HH}$為電子與重電洞動能。因此決定$V_{QD}^{e}\left(\vec{r}_e\right)$、$V_{QD}^{HH}\left(\vec{r}_h\right)$後，利用有限差分法(finite difference method)[13]或基底展開等方法對角化矩陣即可得到不同能態所對應的電子、重電洞動能與波包函數。

3.3.2 庫侖交互作用

電子－電洞間直接庫侖作用矩陣元素定義

$$V_{i_e,j_h,k_h,l_e}^{eh}=\iint d\vec{r}_1 d\vec{r}_2\,\Psi_{i_e}^{e*}\left(\vec{r}_1\right)\Psi_{j_h}^{h*}\left(\vec{r}_2\right)\frac{e^2}{4\pi\varepsilon_0\varepsilon\left|\vec{r}_1-\vec{r}_2\right|}\Psi_{k_h}^{h}\left(\vec{r}_2\right)\Psi_{l_e}^{e}\left(\vec{r}_1\right) \tag{3.3.2.1}$$

而電子－電洞間交換能矩陣元素定義

$$\delta_{i_e,j_h,k_h,l_e}^{s_z,j_z,j_z',s_z'}=\iint d\vec{r}_1 d\vec{r}_2\,\Psi_{i_e}^{e*}\left(\vec{r}_2\right)\Psi_{j_h}^{h}\left(\vec{r}_2\right)\frac{e^2}{4\pi\varepsilon_0\varepsilon\left|\vec{r}_1-\vec{r}_2\right|}\Psi_{k_h}^{h*}\left(\vec{r}_1\right)\Psi_{l_e}^{e}\left(\vec{r}_1\right) \tag{3.3.2.2}$$

e 表電子電量，ε 表材料介電常數(dielectric constant)，$\Psi_{i_e}^{e}\left(\vec{r}\right)$、$\Psi_{j_h}^{h}\left(\vec{r}\right)$ 表電子與電洞在空間中的波函數。而接下來將兩者分別做探討：

1. 電子－電洞間直接庫侖作用[1]：

首先定義電荷密度 $\left(\Psi_{i_e}^{e*}\Psi_{l_e}^{e}\right)$ (charge density)所建立的電位為

$$U_{i_e,l_e}\left(\vec{r}_2\right)\equiv\frac{e^2}{4\pi\varepsilon_0\varepsilon}\int_{-\infty}^{\infty}d\vec{r}_1\frac{\Psi_{i_e}^{e*}\left(\vec{r}_1\right)\times\Psi_{l_e}^{e}\left(\vec{r}_1\right)}{\left|\vec{r}_1-\vec{r}_2\right|} \tag{3.3.2.3}$$

因此(3.3.2.1)式可以寫成

$$V_{i_e,j_h,k_h,l_e}^{eh}\equiv\int_{-\infty}^{\infty}d\vec{r}_2\Psi_{j_h}^{h*}\left(\vec{r}_2\right)\Psi_{k_h}^{h}\left(\vec{r}_2\right)U_{i_e,l_e}\left(\vec{r}_2\right) \tag{3.3.2.4}$$

接下來要利用波包近似法(envelope function approximation)及拆解 $\vec{r}_i=\vec{R}_i+\vec{\tau}_i$ (如圖3.3.2.1)簡化積分式子。

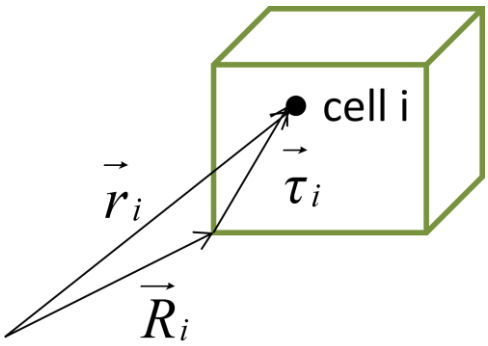


圖 3.3.2.1 cell i 的位置向量分解示意圖。

$\vec{R}_i$ 表示第 i 個 Wigner-Seitz(WS) cell 的位置向量，$\vec{\tau}_i$ 表示 WS cell 內的位置向量。由文獻[14]可將將位置向量拆解，把庫侖矩陣元素 $V^{eh}_{i_e,j_h,k_h,l_e}$ 分成長程部分(電子和電洞在不同的 WS cell 內)和短程部分(電子和電洞在相同的 WS cell 內)，而式子如下

$$\begin{aligned} V^{eh}_{i_e,j_h,k_h,l_e} = & \frac{e^2}{4\pi\varepsilon_0\varepsilon}\sum_{I=1}^{N_{cell}} \int_{\vec{r}_2\in WS(\vec{R}_I)} \Psi^{h*}_{j_h}(\vec{r}_2)\Psi^{h}_{k_h}(\vec{r}_2) \int_{\vec{r}_1\in WS(\vec{R}_I)} \Psi^{e*}_{i_e}(\vec{r}_1)\frac{1}{|\vec{r}_1-\vec{r}_2|}\Psi^{e}_{l_e}(\vec{r}_1)d\vec{r}_1 d\vec{r}_2 \\ & + \frac{e^2}{4\pi\varepsilon_0\varepsilon}\sum_{\substack{I,J=1 \\ I\neq J}}^{N_{cell}} \int_{\vec{r}_2\in WS(\vec{R}_J)} \Psi^{h*}_{j_h}(\vec{r}_2)\Psi^{h}_{k_h}(\vec{r}_2) \int_{\vec{r}_1\in WS(\vec{R}_I)} \Psi^{e*}_{i_e}(\vec{r}_1)\frac{1}{|\vec{r}_1-\vec{r}_2|}\Psi^{e}_{l_e}(\vec{r}_1)d\vec{r}_1 d\vec{r}_2 \end{aligned} \quad (3.3.2.5)$$

N_{cell} 為 WS cell 的數量，表示第 $I(J)$個 WS cell，而我們忽略短程部份的貢獻，則(3.3.2.5)式可寫成

$$V^{eh}_{i_e,j_h,k_h,l_e} \approx \frac{e^2}{4\pi\varepsilon_0\varepsilon}\sum_{I=1}^{N_{cell}} \int_{\vec{r}_2\in WS(\vec{R}_I)} \Psi^{h*}_{j_h}(\vec{r}_2)\Psi^{h}_{k_h}(\vec{r}_2) \int_{\vec{r}_1\in WS(\vec{R}_I)} \Psi^{e*}_{i_e}(\vec{r}_1)\frac{1}{|\vec{r}_1-\vec{r}_2|}\Psi^{e}_{l_e}(\vec{r}_1)d\vec{r}_1 d\vec{r}_2 \quad (3.3.2.6)$$

由(3.3.2.3)式定義的電位可將(3.3.2.6)整理為

$$V^{eh}_{i_e,j_h,k_h,l_e} = \sum_{J=1}^{N_{cell}} \int_{\vec{r}_2\in WS(\vec{R}_J)} \Psi^{h*}_{j_h}(\vec{r}_2)\Psi^{h}_{k_h}(\vec{r}_2)U_{i_e,l_e}(\vec{r}_2)d\vec{r}_2 \quad (3.3.2.7)$$

波包函數具有緩慢變化的性質且 Bloch's function 為週期性函數，如下式

$$g(\vec{R}_i+\vec{\tau}_i) \approx g(\vec{R}_i) \quad (3.3.2.8)$$

$$u(\vec{R}_i+\vec{\tau}_i) = u(\vec{\tau}_i) \quad (3.3.2.9)$$

利用(3.3.2.8)式與(3.3.2.9)式最後可將(3.3.2.7)式化簡為

$$V^{eh}_{i_e,j_h,k_h,l_e} = \int_{-\infty}^{\infty} g^{h*}_{j_h}(\vec{R}_2)g^{h}_{k_h}(\vec{R}_2)U_{i_e,l_e}(\vec{R}_2)d\vec{R}_2 \quad (3.3.2.10)$$

$$U_{i_e,l_e}(\vec{R}_2) = \frac{e}{4\pi\varepsilon_0\varepsilon}\times\int_{-\infty}^{\infty}\frac{g^{e*}_{i_e}(\vec{R}_1)g^{e}_{l_e}(\vec{R}_1)}{|\vec{R}_1-\vec{R}_2|}d\vec{R}_1 \quad (3.3.2.11)$$

當 $\vec{R}_1 \neq \vec{R}_2$ 時，$U^{LR}_{i_e,l_e}(\vec{R}_2)$ 稱為長程作用(Long Range)，當 $\vec{R}_1 = \vec{R}_2$ 時，$U^{SR}_{i_e,l_e}(\vec{R}_2)$ 稱為

短程作用(Short Range)，如下式所示

$$U_{i_e,l_e}\left(\vec{R}_h\right)=U^{LR}_{i_e,l_e}\left(\vec{R}_h\right)+U^{SR}_{i_e,l_e}\left(\vec{R}_h\right) \tag{3.3.2.12}$$

令 $\vec{R}_1=\left(x_i,y_j,z_k\right)$、$\vec{R}_2=\left(x_m,y_n,z_l\right)$ 且取均勻格點及經過矩形法(如圖 3.3.2.3)離散化可近似為

$$V^{eh}_{i_e,j_h,k_h,l_e}=\sum_{m=1}^{N_x+1}\sum_{n=1}^{N_y+1}\sum_{l=1}^{N_z+1}\left[\begin{array}{l} g^{h*}_{j_h}\left(x_m,y_n,z_l\right)g^{h}_{k_h}\left(x_m,y_n,z_l\right) \\ \times U_{i_e,l_e}\left(x_m,y_n,z_l\right)\times\Delta x\Delta y\Delta z \end{array}\right] \tag{3.3.2.13}$$

$$U_{i_e,l_e}\left(x_m,y_n,z_l\right)=\frac{e}{4\pi\varepsilon_0\varepsilon}\times\left\{\begin{array}{l} \displaystyle\sum_{i\neq m}^{N_x+1}\sum_{j\neq n}^{N_y+1}\sum_{k\neq l}^{N_z+1}\frac{g^{e*}_{i_e}\left(x_i,y_j,z_k\right)g^{e}_{l_e}\left(x_i,y_j,z_k\right)\Delta V}{\sqrt{\left(x_i-x_m\right)^2+\left(y_j-y_n\right)^2+\left(z_k-z_l\right)^2}} \\ +F_s\times\displaystyle\sum_{m=1}^{N_x+1}\sum_{n=1}^{N_y+1}\sum_{l=1}^{N_z+1}\left[\begin{array}{l} g^{e*}_{i_e}\left(x_m,y_n,z_l\right) \\ \times g^{e}_{l_e}\left(x_m,y_n,z_l\right) \end{array}\right] \end{array}\right\} \tag{3.3.2.14}$$

$$F_s\equiv\int_{-\Delta x/2}^{\Delta x/2}\int_{-\Delta y/2}^{\Delta y/2}\int_{-\Delta z/2}^{\Delta z/2}\frac{dxdydz}{\sqrt{x^2+y^2+z^2}} \tag{3.3.2.15}$$

F_s 定義單位晶胞內的積分式。由於單位晶胞內的積分使用 mathemetica 程式在短時間內即可被計算出來，在 Fortran 程式計算當中，已經把 F_s 當作參數來輸入，因此將(3.3.2.13)式與(3.3.2.14)式使用 Fortran 程式計算及為所求。

2. 電子－電洞間交換能[15]：

將(3.3.1.1)式與(3.3.1.2)式代入(3.3.2.2)經過可得

$$\begin{aligned}\delta^{s_z,j_z,j_z',s_z'}_{i_e,j_h,k_h,l_e}=&\int\int dr_1^3dr_2^3\left[g^{e}_{i_e}\left(\vec{r}_2\right)u_{s_z}\left(\vec{r}_2\right)\right]^*\left[g^{h}_{j_h}\left(\vec{r}_2\right)u_{j_z}\left(\vec{r}_2\right)\right]\frac{e^2}{4\pi\varepsilon_0\varepsilon_b\left|\vec{r}_1-\vec{r}_2\right|} \\ &\times\left[g^{h}_{k_h}\left(\vec{r}_1\right)u_{j_z'}\left(\vec{r}_1\right)\right]^*\left[g^{e}_{l_e}\left(\vec{r}_1\right)u_{s_z'}\left(\vec{r}_1\right)\right]\end{aligned} \tag{3.3.2.16}$$

接下來要利用波包近似法(envelope function approximation)及拆解 $\vec{r}_1=\vec{R}_1+\vec{\tau}_1$、$\vec{r}_2=\vec{R}_2+\vec{\tau}_2$(如圖 3.3.2.2)來簡化積分式子。

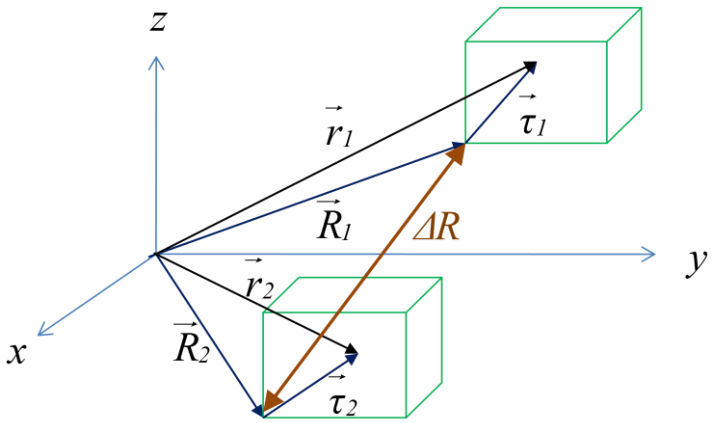


圖 3.3.2.2 兩個不同 WS cell 位置向量分解示意圖。

將(3.3.2.16)式簡化得到

$$\delta_{i_e j_h k_h l_e}^{s_z j_z j_z' s_z'} \approx \sum_{\vec{R}_1,\vec{R}_2} \left\{ g_{i_e}^{e*}\left(\vec{R}_2\right) g_{j_h}^{h}\left(\vec{R}_2\right) g_{k_h}^{h*}\left(\vec{R}_1\right) g_{l_e}^{e}\left(\vec{R}_1\right) \right.$$
$$\left. \times \int_{WS1}\int_{WS2} d\vec{\tau}_1^{\,3} d\vec{\tau}_2^{\,3} \left[u_{s_z}\left(\vec{\tau}_2\right)\right]^* \left[u_{j_z}\left(\vec{\tau}_2\right)\right] \frac{e^2}{4\pi\varepsilon_0\varepsilon_b} \frac{1}{\left|\vec{R}_1 - \vec{R}_2 + \vec{\tau}_1 - \vec{\tau}_2\right|} \times \left[u_{j_z'}\left(\vec{\tau}_1\right)\right]^* \left[u_{s_z'}\left(\vec{\tau}_1\right)\right] \right\} \tag{3.3.2.17}$$

$\vec{R}$ 表示 WS cell 的位置向量，$\vec{\tau}$ 表示 WS cell 內的位置向量。利用(3.3.2.18)式可把(3.3.2.17)式分兩項

$$\sum_{\vec{R}_1,\vec{R}_2} \int_{WS1}\int_{WS2} d\vec{\tau}_1^{\,3} d\vec{\tau}_2^{\,3} \to \sum_{\vec{R}_1=\vec{R}_2}^{N_{cell}} \int_{WS1}\int_{WS2} + \sum_{\vec{R}_1\neq\vec{R}_2}^{N_{cell}} \int_{WS1}\int_{WS2} \tag{3.3.2.18}$$

分別是短程作用與長程作用，N_{cell} 為 WS cell 個數。短程作用表示當 $\vec{R}_1 = \vec{R}_2$ 時的情況，而長程作用表示當 $\vec{R}_1 \neq \vec{R}_2$ 時的情況。而長程作用又可分成單極－單極作用(monopole- monopole interaction)與偶極－偶極作用(dipole- dipole interaction)。因此(3.3.2.17)式又可寫成

$$\delta_{i_e,j_h,k_h,l_e}^{s_z,j_z,j_z',s_z'} = \delta_{i_e,j_h,k_h,l_e}^{s_z,j_z,j_z',s_z',LR(D)} + \delta_{i_e,j_h,k_h,l_e}^{s_z,j_z,j_z',s_z',LR(M)} + \delta_{i_e,j_h,k_h,l_e}^{s_z,j_z,j_z',s_z',SR} \tag{3.3.2.19}$$

目前本論文要探討的是長程偶極－偶極交換能的部分，其式子為

$$\delta_{i_e j_h k_h l_e}^{s_z j_z j_z' s_z' LR(D)} = \sum_{\vec{R}_2} \Delta\Omega^2 \times g_{i_e;s_z}^{e\,*}\left(\vec{R}_2\right) g_{j_h}^{h}\left(\vec{R}_2\right) \sum_{\vec{R}_1\neq\vec{R}_2} W_{LR(D)}^{s_z,j_z,j_z',s_z'}\left(\Delta\vec{R}\right) \times g_{k_h}^{h*}\left(\vec{R}_1\right) g_{l_e}^{e}\left(\vec{R}_1\right) \tag{3.3.2.20}$$

$$W_{LR(D)}^{s_z,j_z,j_z',s_z'}\left(\Delta\vec{R}\right) \equiv \sum_{\alpha,\beta} C_{s_z,j_z,\alpha} C_{s_z',j_z',\beta}^{\;*} w_{\alpha,\beta}^{LR(D)}\left(\Delta\vec{R}\right) \tag{3.3.2.21}$$

$$w_{\alpha,\beta}^{LR(D)}\left(\Delta\vec{R}\right) = \frac{e^2}{4\pi\varepsilon_0\varepsilon_b}\left(\frac{\hbar^2}{E_g^2}\frac{E_p}{2m_0}\right) \times \left\{ \frac{\alpha\cdot\hat{\beta}}{\left|\Delta\vec{R}\right|^3} - \frac{3\Delta\alpha\cdot\Delta\beta}{\left|\Delta\vec{R}\right|^5} \right\} \tag{3.3.2.22}$$

$\Delta\Omega$ 為單位體積的大小，α 、β 表卡氏座標的{x、y、z}，E_g 為導電帶與價電帶之間的能隙(energy gap)，E_p 為導電帶與價電帶間交互作用能(conduction-valence band interaction energy)，$C_{s_z,j_z,\alpha}$ 、$C_{s_z',j_z',\beta}$ 係數如表 3.3.2.2：

表 3.3.2.2 每一組不同的自旋、自旋軌道角動量與方向對應的係數。

s_z, j_z, α	$C_{s_z,j_z,\alpha}$	s_z, j_z, α	$C_{s_z,j_z,\alpha}$
$+\frac{1}{2},+\frac{3}{2},x$	0	$-\frac{1}{2},+\frac{3}{2},x$	$\sqrt{1/2}$
$+\frac{1}{2},+\frac{3}{2},y$	0	$-\frac{1}{2},+\frac{3}{2},y$	$-i\sqrt{1/2}$
$+\frac{1}{2},+\frac{3}{2},z$	0	$-\frac{1}{2},+\frac{3}{2},z$	0
$+\frac{1}{2},-\frac{3}{2},x$	$-\sqrt{1/2}$	$-\frac{1}{2},-\frac{3}{2},x$	0
$+\frac{1}{2},-\frac{3}{2},y$	$-i\sqrt{1/2}$	$-\frac{1}{2},-\frac{3}{2},y$	0
$+\frac{1}{2},-\frac{3}{2},z$	0	$-\frac{1}{2},-\frac{3}{2},z$	0

令 $\vec{R}_1=(x_i,y_j,z_k)$ 、$\vec{R}_2=(x_m,y_n,z_l)$ 且取均勻格點及經過矩形法(如圖 3.3.2.3)離散化，(3.3.2.20)式、(3.3.2.21)式、(3.3.2.22)式中兩項可寫成

$$\delta_{i_e j_h k_h l_e}^{s_z,j_z,j_z',s_z',LR(D)}=\sum_{m=1}^{N_x+1}\sum_{n=1}^{N_y+1}\sum_{l=1}^{N_z+1}\left\{\begin{array}{l}\Delta\Omega^2\times g_{i_e}^{e*}(x_m,y_n,z_l)\,g_{j_h}^{h}(x_m,y_n,z_l)\\ \times\sum_{i\neq m}^{N_x+1}\sum_{j\neq n}^{N_y+1}\sum_{k\neq l}^{N_z+1}\left[\begin{array}{l}W_{LR(D)}^{s_z,j_z,j_z',s_z'}\left(\sqrt{(x_i-x_m)^2+(y_j-y_n)^2+(z_k-z_l)^2}\right)\\ \times g_{k_h}^{h*}(x_i,y_j,z_k)\,g_{l_e}^{e}(x_i,y_j,z_k)\end{array}\right]\end{array}\right\} \tag{3.3.2.23}$$

$$W_{LR(D)}^{s_z,j_z,j_z',s_z'}\left(\sqrt{(x_i-x_m)^2+(y_j-y_n)^2+(z_k-z_l)^2}\right)\equiv$$
$$\sum_{\alpha,\beta}C_{s_z,j_z,\alpha}C_{s_z',j_z',\beta}^{*}\,w_{\alpha,\beta}^{LR(D)}\left(\sqrt{(x_i-x_m)^2+(y_j-y_n)^2+(z_k-z_l)^2}\right) \tag{3.3.2.24}$$

$$w_{\alpha,\beta}^{LR(D)}\left(\sqrt{(x_i-x_m)^2+(y_j-y_n)^2+(z_k-z_l)^2}\right)=$$
$$\frac{e^2}{4\pi\varepsilon_0\varepsilon_b}\left(\frac{\hbar^2}{E_g^2}\frac{E_p}{2m_0}\right)\times\left\{\frac{\alpha\cdot\hat{\beta}}{\left|\sqrt{(x_i-x_m)^2+(y_j-y_n)^2+(z_k-z_l)^2}\right|^3}-\frac{3\Delta\alpha\cdot\Delta\beta}{\left|\sqrt{(x_i-x_m)^2+(y_j-y_n)^2+(z_k-z_l)^2}\right|^5}\right\} \tag{3.3.2.25}$$

利用(3.3.2.23)式、(3.3.2.24)式、(3.3.2.25)式並使用 Fortran 程式計算即為所求。

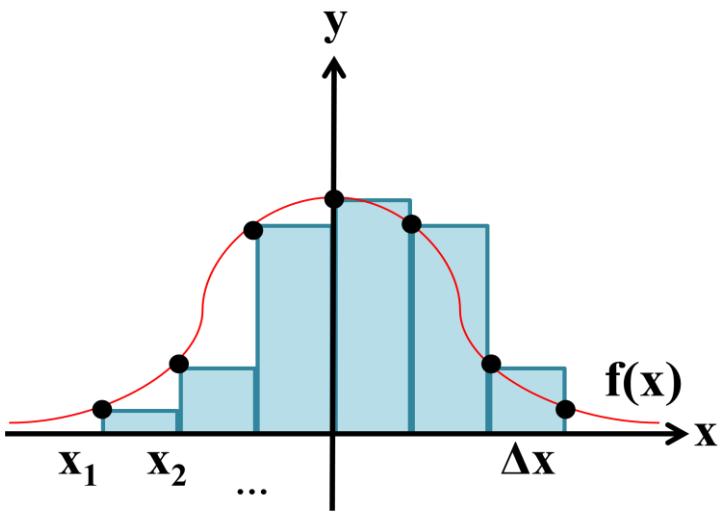


圖 3.3.2.3 取均勻格點且使用矩形法一維積分離散化示意圖。

3.3.3 三維拋物線模型的解析解

接下來要驗證數值方法與程式的正確性，考慮電子－電洞的庫侖交互作用，在三維非等向性拋物線位能模型下，假設電子與電洞波函數對稱，則基態波函數為

$$g^{e}\left(\vec{R_1}\right)=\frac{1}{\sqrt{\pi^{3/2}l_x l_y l_z}}\exp\left[-\frac{1}{2}\left(\left(\frac{x_1}{l_x}\right)^2+\left(\frac{y_1}{l_y}\right)^2+\left(\frac{z_1}{l_z}\right)^2\right)\right] \tag{3.3.3.1}$$

$$g^{h}\left(\vec{R_2}\right)=\frac{1}{\sqrt{\pi^{3/2}l_x l_y l_z}}\exp\left[-\frac{1}{2}\left(\left(\frac{x_2}{l_x}\right)^2+\left(\frac{y_2}{l_y}\right)^2+\left(\frac{z_2}{l_z}\right)^2\right)\right] \tag{3.3.3.2}$$

l_x、l_y和l_z代表電子與電洞x、y和z方向的特徵長度定義。將(3.3.3.1)式與(3.3.3.2)是代入庫侖交互作用矩陣元素(3.3.2.13)式與(3.3.2.23)式可推導出：

1.電子－電洞間直接庫侖作用：

若$l_x \cong l_y$且定義$l_{\parallel}=\sqrt{l_x l_y}$可推導出

$$V_{s,s,s,s}^{eh}=\frac{e^2}{4\pi\varepsilon_0\varepsilon}\frac{\sqrt{2}}{\sqrt{\pi}}\frac{1}{l_z}\times\frac{\tanh^{-1}\left(\sqrt{1-\left(l_{\|}/l_z\right)^2}\right)}{\sqrt{1-\left(l_{\|}/l_z\right)^2}}\ ,\ l_z/l_{\|}>1 \tag{3.3.3.3}$$

$$V_{s,s,s,s}^{eh}=\frac{e^2}{4\pi\varepsilon_0\varepsilon}\frac{\sqrt{2}}{\sqrt{\pi}}\frac{1}{l_{\|}}\ ,\ l_z/l_{\|}=1 \tag{3.3.3.4}$$

$$V_{s,s,s,s}^{eh}=\frac{e^2}{4\pi\varepsilon_0\varepsilon}\frac{\sqrt{2}}{\sqrt{\pi}}\frac{1}{l_{\|}}\times\frac{\sin^{-1}\left(\sqrt{1-\left(l_z/l_{\|}\right)^2}\right)}{\sqrt{1-\left(l_z/l_{\|}\right)^2}}\ ,\ l_z/l_{\|}<1 \tag{3.3.3.5}$$

參考文獻為陳彥廷學長發表的期刊[16]。

2.電子－電洞間長程偶極－偶極交換能：

推導出其中一項解析解為

$$\delta_{s,s,s,s}^{-\frac{1}{2},+\frac{3}{2},-\frac{3}{2},+\frac{1}{2},LR(D)}=K\times\xi\left(1-\xi\right)\frac{\gamma_z}{l_y^3} \tag{3.3.3.6}$$

$$K=\frac{3\sqrt{\pi}e^2\hbar^2E_p}{\left(4\pi\varepsilon_0\right)16\sqrt{2}\varepsilon m_0E_g^2} \tag{3.3.3.7}$$

$$\gamma_z=\exp\left[\left(\frac{3\sqrt{\pi}l_z}{4l_y}\right)^2\right]erfc\left(\frac{3\sqrt{\pi}l_z}{4l_y}\right) \tag{3.3.3.8}$$

參考文獻為實驗室博後 H. Y. Ramirez 發表的期刊[17]。此期刊所使用的 E_g 是塊材(bulk)的能隙 E_g^b，但在量子點中應做修正，如下式

$$E_g=E_g^{QD}=E_g^b+E_h+E_e \tag{3.3.3.9}$$

$$E_e=\frac{\hbar^2}{m_e^*}\left[\frac{1}{2l_x^2}+\frac{1}{2l_y^2}+\frac{1}{2l_z^2}\right] \tag{3.3.3.10}$$

$$E_h=\hbar^2\left[\frac{1}{2m_{HH,x}^*l_x^2}+\frac{1}{2m_{HH,y}^*l_y^2}+\frac{1}{2m_{HH,z}^*l_z^2}\right] \tag{3.3.3.11}$$

，而 E_g^b 、 E_h 、 E_e 、 E_g^{QD} 關係示意圖如圖 3.3.3.1。

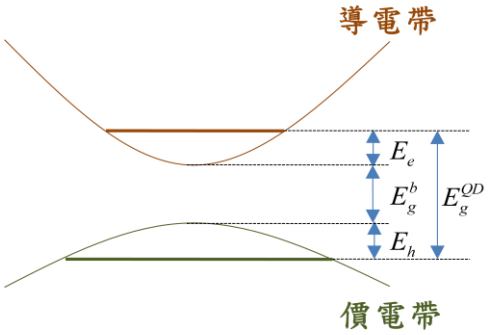


圖 3.3.3.1 半導體能帶 E_g^b 、E_h 、E_e 、E_g^{QD} 關係示意圖。

將塊材與量子點的能隙代入推導出來的解析解與數值解比較其結果。

3.3.4 高斯函數積分範圍選取

由於庫侖交互作用解析解與數值解要做結果比對，因此將要針對積分式子裡的波函數所選取的範圍做討論，我們選取的波函數為高斯函數。接下來要利用下式積分測試高斯函數的計算範圍：

$$F = \int_{-\infty}^{\infty} \exp\left[-\frac{1}{2}\left(\frac{x}{l_x}\right)^2\right] dx \qquad (3.3.4.1)$$

(3.3.4.1)式積分後的解析解為

$$F = \sqrt{2\pi} l_x \qquad (3.3.4.2)$$

利用矩形法將積分離散化且取均勻格點(格點數取 N_x 個)後，其形式為

$$F \approx F' = \sum_{i=1}^{N_x} \exp\left[-\frac{1}{2}\left(\frac{x_i}{l_x}\right)\right] \Delta x \qquad (3.3.4.3)$$

將(3.3.4.3)式利用 Fortran 程式做運算，而程式運算中計算範圍無法選擇無限大，因此需要選擇一個可以接受的誤差範圍，我們定義積分範圍為 L_x，如圖 3.3.4.1。

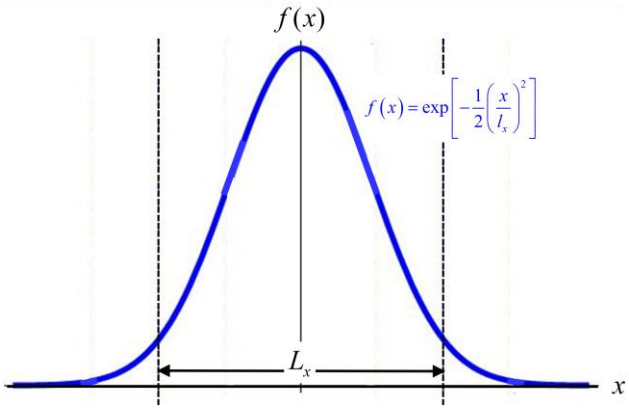


圖 3.3.4.1 高斯函數分布範圍示意圖，l_x 為在 x 方向的特徵長度、f(x)為高斯函數。

接下來令 l_x=3、l_x>>Δx(格點間距)=0.01，利用(3.3.4.3)式畫出 $L_x - F'$ 關係圖及 L_x －誤差關係圖，如圖 3.3.4.2。

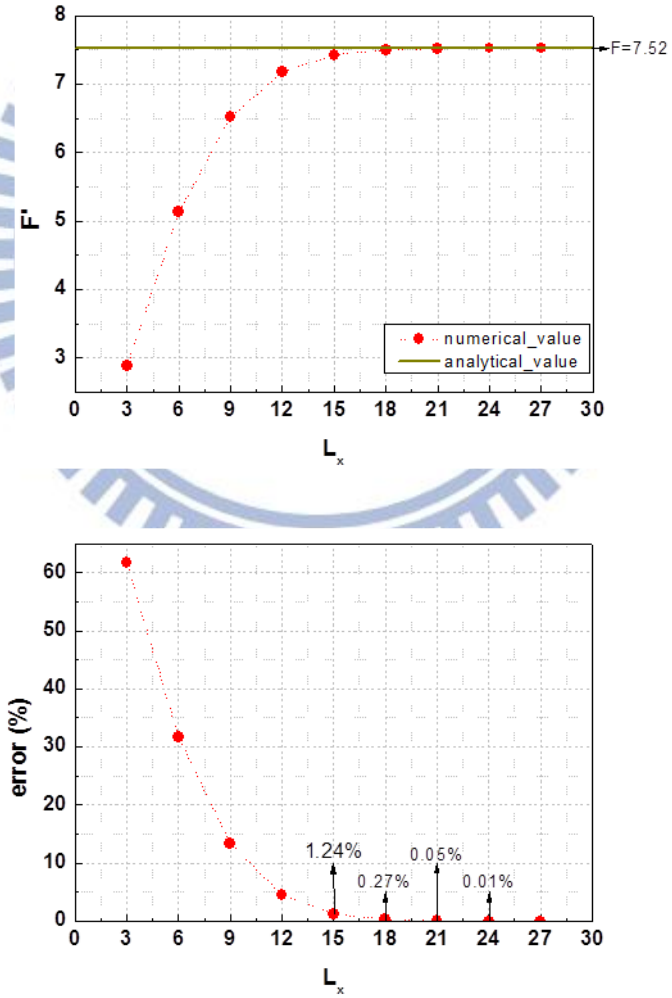
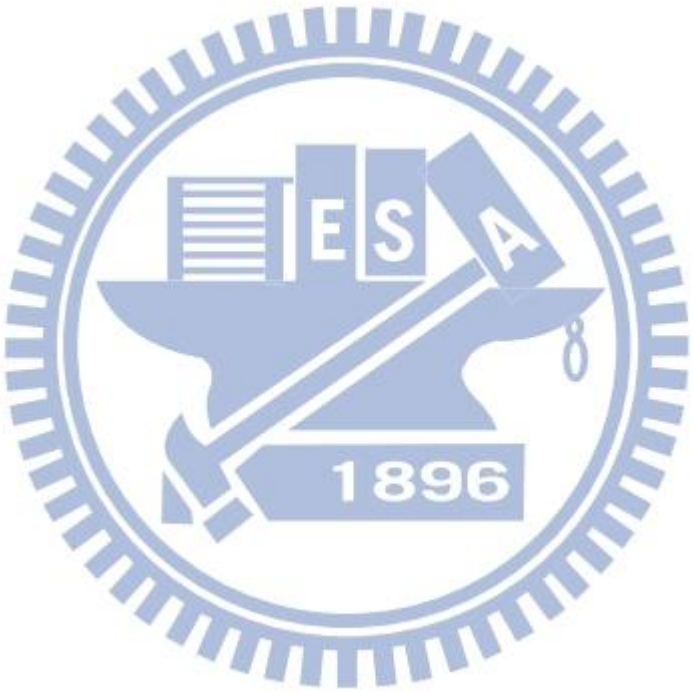


圖 3.3.4.2 高斯函數特徵長度 l_x=3，(上) $L_x - F'$ 關係圖及(下) L_x －誤差關係圖。

由圖 3.3.4.2 會發現選取積分範圍為 6 倍特徵長度時，其積分結果會與解析解誤

差為 0.27%，表示選取 6 倍特徵長度時幾乎包含整個波函數，即使積分範圍不考慮 6 倍特徵長度外的積分對整體結果影響不大，因此接下來有關高斯函數的積分將採取 6 倍特徵長度當作積分範圍。



3.4 計算長程偶極－偶極交換能

3.4.1 短程與長程交換能之間的界限

以圖 3.4.1.1 為例，威格納－塞茲晶胞定義為晶胞內只包含一個晶格點[18]，此結構裡每個晶格點具有兩個原子，黑線所框出來的大約為一個威格納－塞茲晶胞的體積大小，而選取威格納－塞茲晶胞的方式如圖 3.4.1.2 所示，附錄 G 為利用四個單位晶胞(unit cell)畫出圖 3.4.1.1 的威格納－塞茲晶胞的體積。

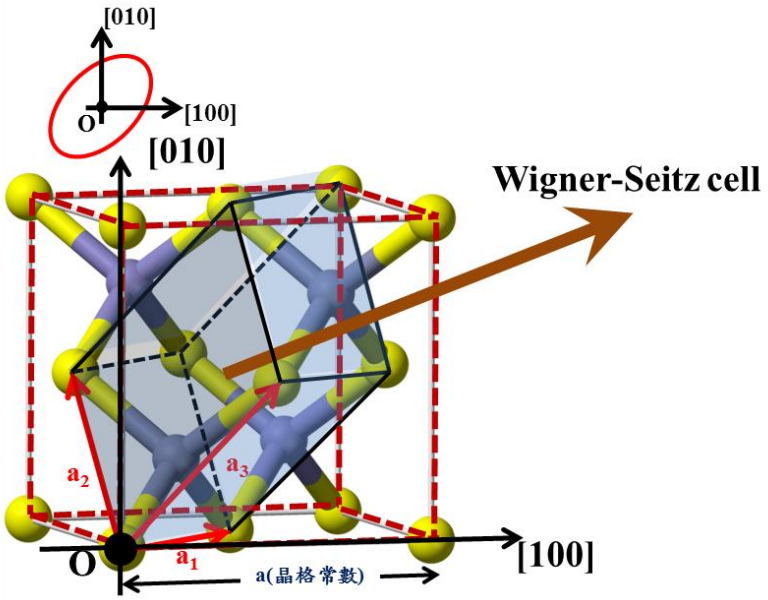


圖 3.4.1.1 閃鋅結構：一個單位晶胞佔有四個威格納－塞茲晶胞體積大小，在座標上的橢圓為[100]-[010]方向上的量子點。

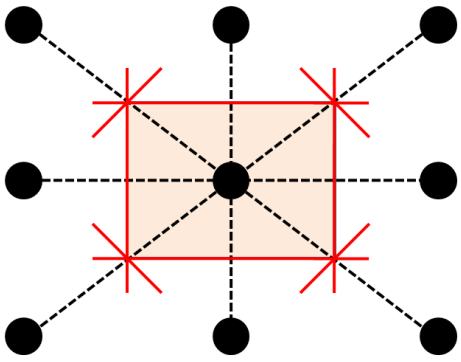


圖 3.4.1.2 選取威格納－塞茲晶胞方式：(1)選擇一個晶格點(黑點)用直線連接附近晶個點；(2)畫出這些直線的垂直平分線(和面)；用此方式得到最小封閉面積為威格納－塞茲晶胞。

閃鋅結構(Zincblende structure)的威格納－塞茲晶胞選取後的形狀為傾斜的六面體，其原始平移向量為

$$\begin{cases} \vec{a}_1 = (\frac{a}{2}, \frac{a}{2}, 0) \\ \vec{a}_2 = (0, \frac{a}{2}, \frac{a}{2}) \\ \vec{a}_3 = (\frac{a}{2}, 0, \frac{a}{2}) \end{cases} \tag{3.4.1.1}$$

因此可計算出體積為

$$V = \left|\vec{a}_1 \cdot \vec{a}_2 \times \vec{a}_3\right| = \frac{a^3}{4} \tag{3.4.1.2}$$

一個單位晶胞有八個原子，因此體積等於四個威格納－塞茲晶胞的體積大小。而我們再做數值運算時，將它近似為正立方體，如圖 3.4.1.3。

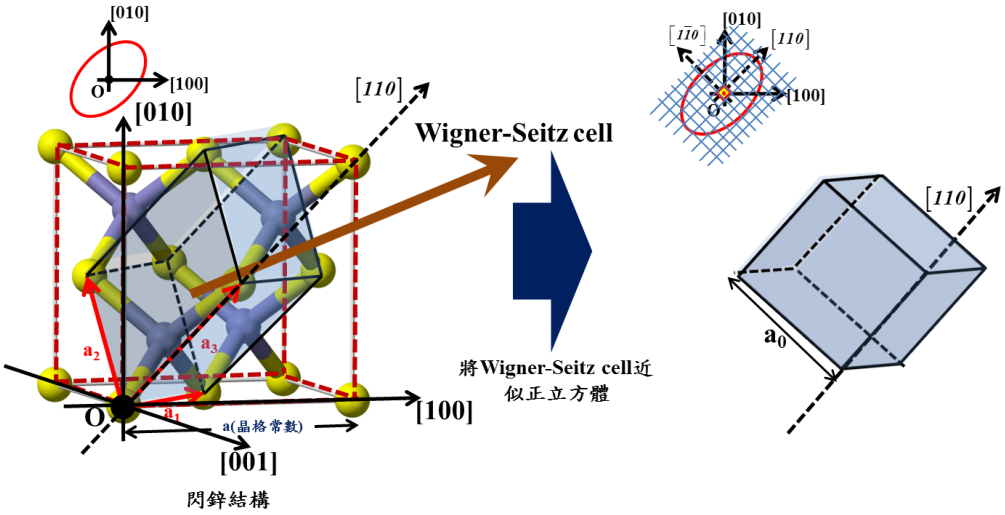


圖 3.4.1.3 將威格納－塞茲晶胞近似正立方體示意圖：由於量子點長軸與短軸的方向為$[110]$、$[1\bar{1}0]$，因此正立方體的面垂直於$[110]$,$[1\bar{1}0]$,$[001]$。

接著將計算正立方體的邊長，計算結果如下

$$a_0 = V^{\frac{1}{3}} = \left(\frac{a^3}{4}\right)^{\frac{1}{3}} = \sqrt{0.25}a \tag{3.4.1.3}$$

以 InAs 為例，閃鋅結構的一個晶格常數(lattice constant) $a = 0.605\,\text{nm}$，因此

$$a_0 = \sqrt{0.25} \times 0.605 \approx 0.38(nm) \tag{3.4.1.4}$$

3.3 小節可知電子－電洞間交換能分成短程交換能與長程交換能，而短程交換能表示兩個粒子處於同一個威格納－塞茲晶胞之間的交互作用，長程交換能表示兩個粒子處於不同威格納－塞茲晶胞的交互作用，此結構的每個威格納－塞茲

晶胞為正立方體，在此定義每個威格納－塞茲晶胞的邊長為 a_0，也就是短程交換能與長程交換能之間的界線長度，如圖 3.4.1.2 所示。在做數值計算時為了避免長程偶極－偶極交換能積分計算到部分短程交換能，會設定兩個粒子在同一個威格納－塞茲晶胞時，程式不會計算到，因此我們要驗證在 $a_0 \approx 0.38$ nm 時，數值解與解析解比對是否恰當。

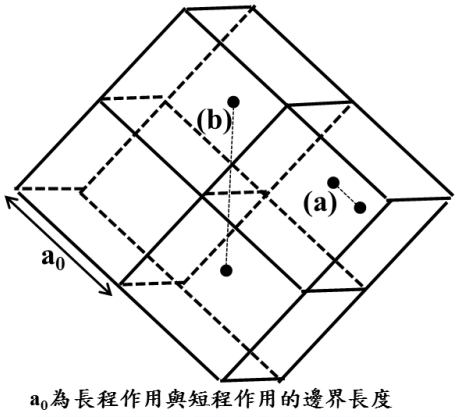


圖 3.4.1.2 (a)表示兩粒子交換能為短程交換能，(b)表示兩粒子交換能為長程交換能。

3.4.2 分析奇異點

在長程偶極－偶極交換能的計算中，必需考慮到不計算到短程交換能的部分，因此將(3.3.2.29)式寫成積分的式子做分析

$$\delta_{i_e,j_h,k_h,l_e}^{s_z,j_z,j_z',s_z',LR(D)} = \int\int d\vec{R}_1 d\vec{R}_2\, g_{i_e}^{e*}\left(\vec{R}_2\right) g_{j_h}^{h}\left(\vec{R}_2\right) \times W_{LR(D)}^{s_z,j_z,j_z',s_z'}\left(\Delta\vec{R}\right) \times g_{k_h}^{h*}\left(\vec{R}_1\right) g_{l_e}^{e}\left(\vec{R}_1\right) \qquad (3.4.2.1)$$

探討 $\delta_{i_e,j_h,k_h,l_e}^{-\frac{1}{2},+\frac{3}{2},-\frac{3}{2},+\frac{1}{2},LR(D)}$ 為例，因此將(3.4.2.1)式寫成

$$\delta_{i_e,j_h,k_h,l_e}^{-\frac{1}{2},+\frac{3}{2},-\frac{3}{2},+\frac{1}{2},LR(D)} = \int\int d\vec{R}_1 d\vec{R}_2\, g_{i_e}^{e*}\left(\vec{R}_2\right) g_{j_h}^{h}\left(\vec{R}_2\right) \times W_{LR(D)}^{-\frac{1}{2},+\frac{3}{2},-\frac{3}{2},+\frac{1}{2}}\left(\Delta\vec{R}\right) \times g_{k_h}^{h*}\left(\vec{R}_1\right) g_{l_e}^{e}\left(\vec{R}_1\right) \qquad (3.4.2.2)$$

我們將令 $f(\vec{R}_1,\vec{R}_2)$ 為(3.4.2.3)式

$$f(\vec{R}_1,\vec{R}_2) \equiv g_{i_e}^{e*}\left(\vec{R}_2\right) g_{j_h}^{h}\left(\vec{R}_2\right) \times W_{LR(D)}^{-\frac{1}{2},-\frac{3}{2},+\frac{3}{2},+\frac{1}{2}}\left(\Delta\vec{R}\right) \times g_{k_h}^{h*}\left(\vec{R}_1\right) g_{l_e}^{e}\left(\vec{R}_1\right) \qquad (3.4.2.3)$$

其中 $\Delta\vec{R} = \left|\vec{R}_1 - \vec{R}_2\right|$。令 $\vec{R}_1 = (x_1,y_1,z_1)$、$\vec{R}_2 = (x_2,y_2,z_2)$ 及將(3.4.2.3)式代入(3.4.2.2)式可將積分式子寫成

$$\delta_{i_e,j_h,k_h,l_e}^{-\frac{1}{2},+\frac{3}{2},-\frac{3}{2},+\frac{1}{2},LR(D)} = \int\int\int\int\int\int dx_1 dy_1 dz_1 dx_2 dy_2 dz_2 \; f\left((x_1,y_1,z_1),(x_2,y_2,z_2)\right) \qquad (3.4.2.4)$$

我們由 3.3.4 小節的結果將積分範圍 L_x 、 L_y 採取 6 倍的特徵長度，$\vec{R}_1$ 固定為 $(0,0,0)$ 與 $(4.5,0,0)$ 且令 $\vec{R}_2=(x,y,0)$，在此可想像空間中有電荷雲分布，而我們考慮 $\vec{R}_1$ 位置電偶極矩的電荷與 $\vec{R}_2$ 位置電偶極矩的電荷互相作用產生的交換能。將(3.4.2.3)式代入三維拋物線模型的波函數且利用 mathemetica 程式畫出 $f((0,0,0),(x,y$ 與 $f((4.5,0,0),(x,y,0))$ 對 x,y 平面關係圖，我們可以看出當兩電偶極矩的電荷距離越近時，所產生的交換能會越大；當兩電偶極矩的電荷距離很遠時，彼此幾乎不會作用，因此交換能幾乎為零，如圖 3.4.2.1 與圖 3.4.2.2。

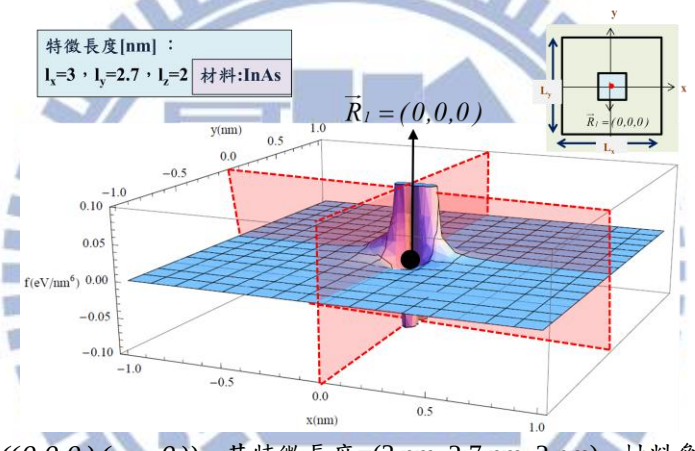


圖 3.4.2.1 分析 $f((0,0,0),(x,y,0))$，其特徵長度=(3 nm ,2.7 nm ,2 nm)、材料參數使用 InAs 畫出 $f((0,0,0),(x,y,0))$[eV/nm^6]對 x,y [nm]平面關係圖。

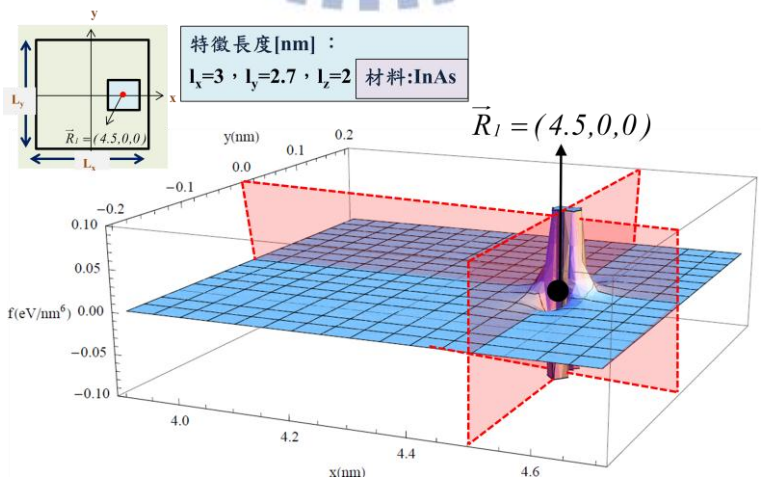


圖 3.4.2.2 分析 $f((4.5,0,0),(x,y,0))$，其特徵長度=(3 nm ,2.7 nm ,2 nm)、材料參數使用 InAs 畫出 $f((4.5,0,0),(x,y,0))$[eV/nm^6]對 x,y [nm]平面關係圖。

觀察圖 3.4.2.1 與圖 3.4.2.2 可知當兩電偶極矩的電荷距離很近時，所產生的交換能會很大；兩電偶極矩的電荷距離很遠時，所產生的交換能遠小於兩電偶極矩的電荷距離很近交換能。因此距離很遠不會對結果造成很大的影響，所以我們將座標平移到以 $\vec{R}_1$ 為中心，對 $\vec{R}_1$ 固定 $(4.5,0,$ 且令 $\vec{R}_2=(x+4.5,y,$，利用 mathemetica 程式畫出 $f((4.5,0,0),(x'+4.5,y',0))$ 對 x',y' 平面關係圖，如圖 3.4.2.3。

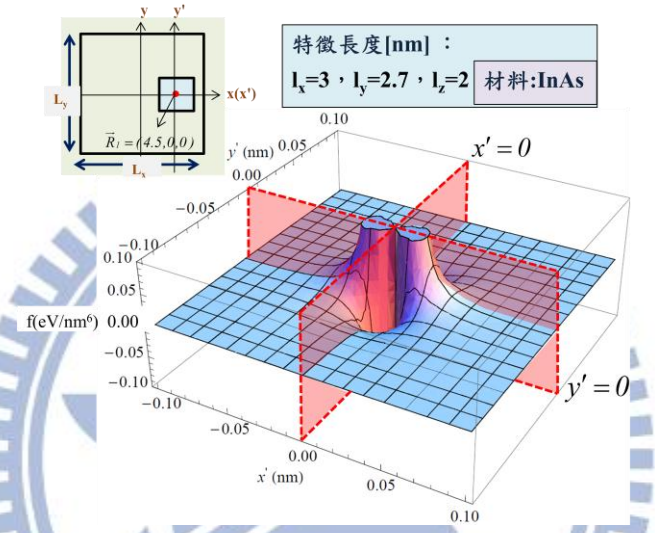


圖 3.4.2.3 分析 $f((4.5,0,0),(x'+4.5,y',0))$，其特徵長度=(3 nm ,2.7 nm ,2 nm)、材料參數使用 InAs 畫出 $f((4.5,0,0),(x'+4.5,y',0))$[eV/nm^6]對 x',y' [nm]平面關係圖。

考慮圖 3.4.2.3 的 $x'=0$ 與 $y'=0$ 截面可畫出 $f((4.5,0,0),(x'+4.5,0,0))$ 對 x' 關係圖與 $f((4.5,0,0),(4.5,y',0))$ 對 y' 關係圖，如圖 3.4.2.4。

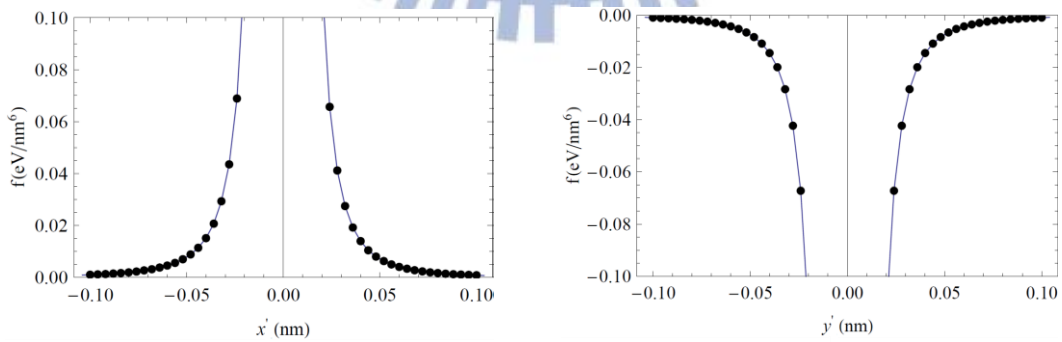


圖 3.4.2.4 分析 $f((4.5,0,0),(x'+4.5,0,0))$[左]與 $f((4.5,0,0),(4.5,y',0))$[右]，其特徵長度=(3 nm ,2.7 nm ,2 nm)、材料參數使用 InAs 畫出 $f((4.5,0,0),(x'+4.5,0,0))$[eV/nm^6]對 x' [nm]關係圖與 $f((4.5,0,0),(4.5,y',0))$[eV/nm^6]對 y' [nm]關係圖：x',y'座標在 -0.1~0.1 nm 範圍內取 50 個格點。

由圖 3.4.2.3 可發現此函數有 2 個正值的奇異點與 2 個負值的奇異點，以 InAs

而言一個威格納－塞茲晶胞平均長度約為 0.38(nm)，再由圖 3.4.2.4 可知數值變化較大的值都落在同一個威格納－塞茲晶胞內，此部分是屬於短程交換能，而我們計算長程交換能不需要計算此部分，因此必須要考量它所提供的積分貢獻，是否對於積分結果有差異，我們分析圖 3.4.2.3 且將圖分成四等份，分別旋轉到同一個象限，而旋轉的示意圖如圖 3.4.2.5。

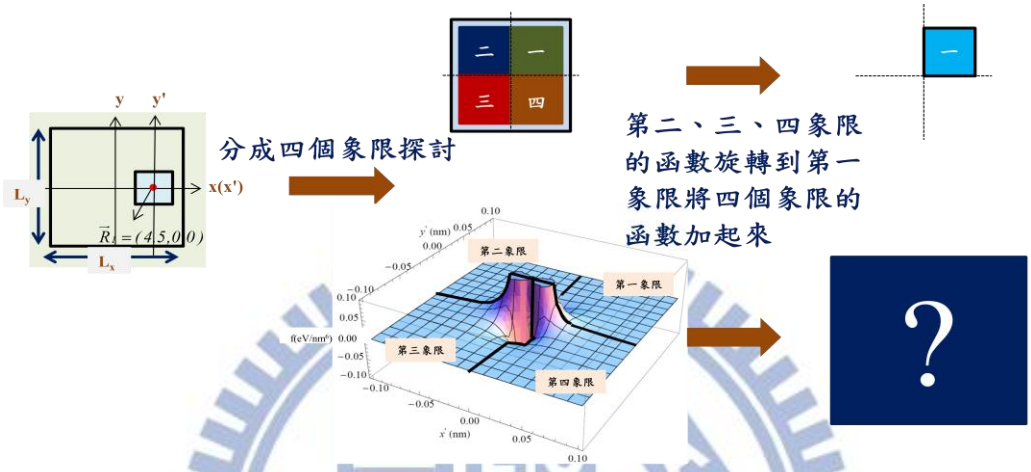


圖 3.4.2.5 讓圖 3.4.2.2 分割四等份及旋轉到同一個象限示意圖

旋轉後的函數定義 $f'(x',y')$，即

$$f'(x',y')\equiv\begin{bmatrix} f((4.5,0,0),(x'+4.5,y',0))+f((4.5,0,0),(-y'+4.5,x',0)) \\ +f((4.5,0,0),(-x'+4.5,-y',0))+f((4.5,0,0),(y'+4.5,-x',0)) \end{bmatrix} \quad (3.4.2.5)$$

其中 $x',y'\geq 0$。利用(3.4.2.5)式將四個象限的函數旋轉把正負奇異點附近的數值互相抵消，假如只考慮截面上的相加，會有如圖 3.4.2.6 所示。

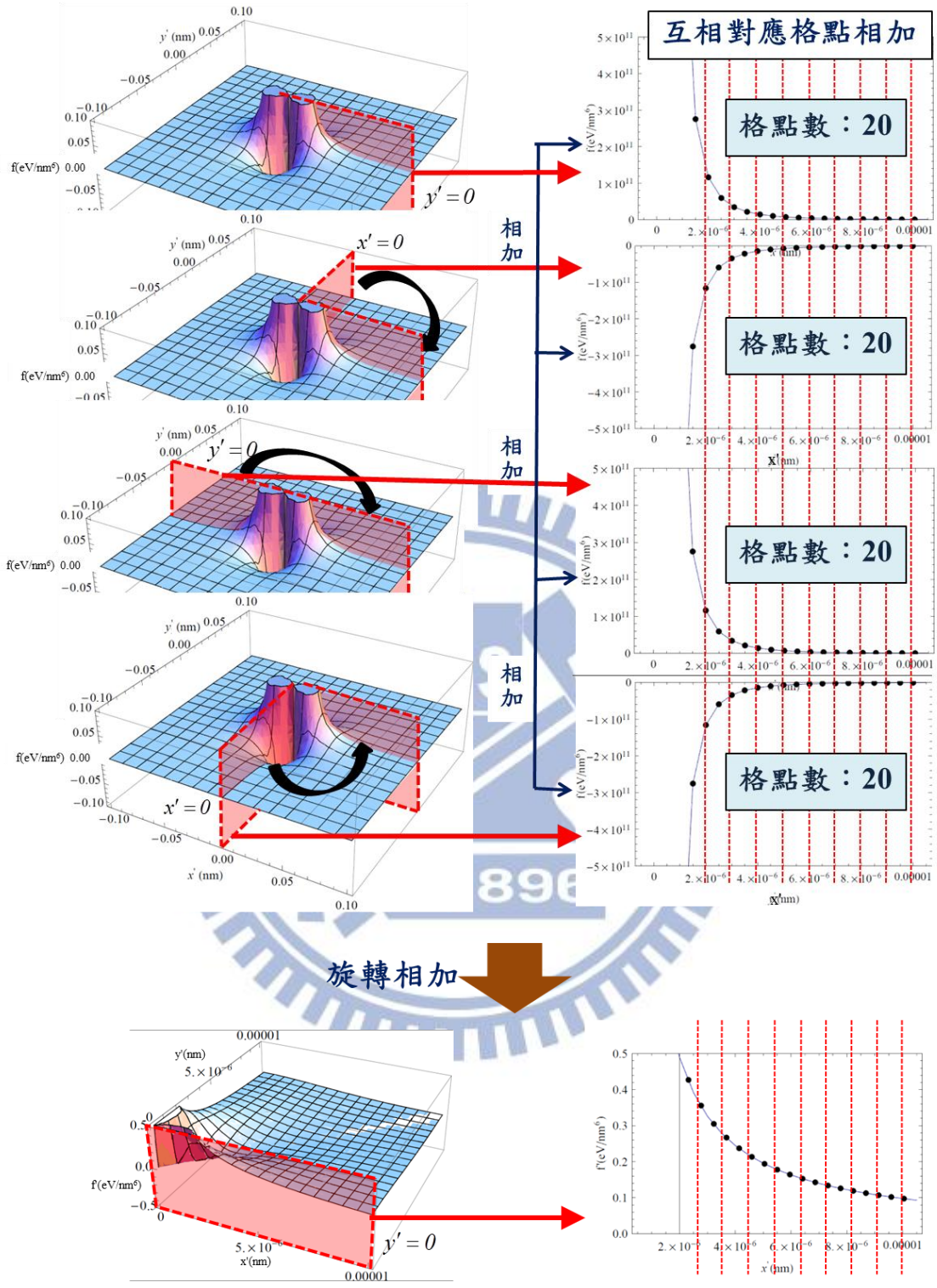


圖 3.4.2.6 利用一維格點描述數值上的限制：x',y'座標在-0.00001~0.00001 nm 範圍內取 40 個格點，再將二、三、四象限平面旋轉到第一象限。

圖 3.4.2.6 得知，在數值運算產生格點時會有限制條件，其條件為

以圖 3.4.2.3 的 $\vec{R}_I$ 為中心，產生左右對稱的格點；由於 $\vec{R}_I$ 會改變，因此產生每個維度的格點採取格點間距相同的方法，即產生均勻格點且三個維度的格點相同($\Delta x = \Delta y = \Delta z = \Delta r$)來做計算。

利用此方式產生格點原因為要使正負奇異點互相抵消，必須考慮旋轉後相消的格點要全部疊在一起，由於奇異點附近變化非常大，因此假如旋轉後格點沒對上，其數值會與真正的結果必定會不一樣，如圖 3.4.2.7。

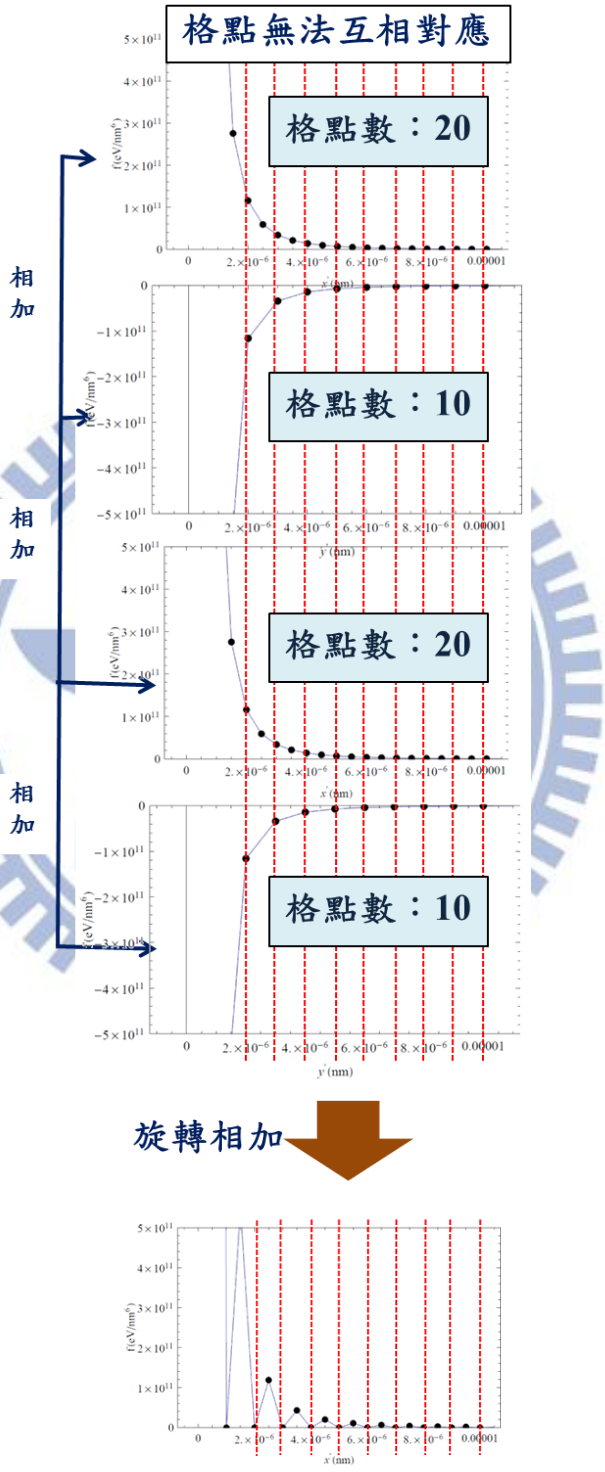


圖 3.4.2.7 格點無法相互對應相加的後果；曲線會劇烈震盪，積分後結果誤差會相當大：x'座標在-0.00001~0.00001 nm 範圍內取 40 個格點，y'座標在- 0.00001~0.00001 nm 範圍內取 20 個格點，再將二、三、四象限平面旋轉到第一象限。

將圖 3.4.2.6 轉完後的圖切 $y'=0$ 得截面放大，如圖 3.4.2.8。

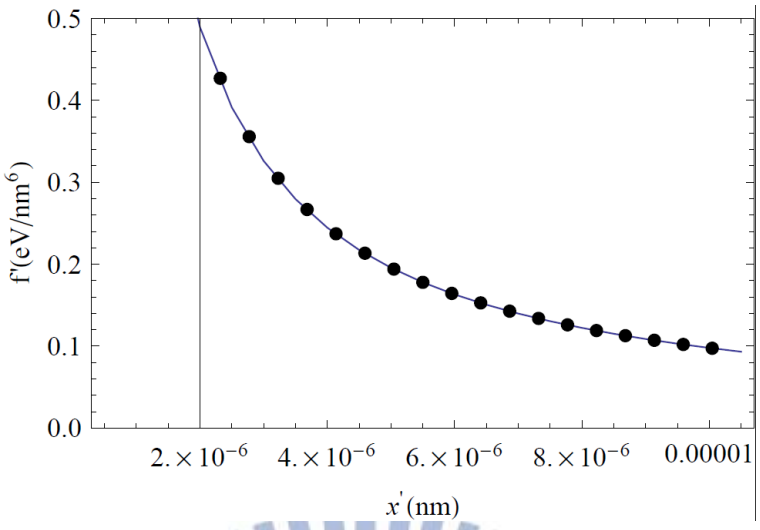


圖 3.4.2.8 分析 $f'(x',0)$ 且畫出 $f'(x',0)$ [eV/nm^6]對 x'[nm]的關係圖。

圖 3.4.2.8 發現還是有奇異點，但數量級已經很小了，因此對於積分貢獻不大。我們可以得知考慮奇異點積分時，正奇異點與負奇異點的數值會互相抵消，因此計算威格納－塞茲晶胞內的積分時，其數值結果會是一個微小的值，表示不考慮威格納－塞茲晶胞內的積分，對於整體積分效果影響不大。而第四章將要驗證 3.4.2 小節討論的理論是否正確，因此會以 InAs 當作例子來比較數值解與解析解對於 a_0=0.38(nm)的結果是否適當。

3.5 單核 CPU 與 GPU 時間差異測試(6 重迴圈)

由於庫侖交互作用積分是有奇異點的，計算式子較為複雜，所以首先考慮無奇異點的簡單積分例子來測試單核 CPU 與 GPU 的時間差異，而積分式子如下

$$S=\int_0^{\frac{5}{2}\pi}\int_0^{\frac{5}{2}\pi}\int_0^{\frac{5}{2}\pi}\left\{\int_0^{\frac{5}{2}\pi}\int_0^{\frac{5}{2}\pi}\int_0^{\frac{5}{2}\pi}\left[\begin{array}{l}x_1y_1z_1\cos(x_1)\cos(y_1)\cos(z_1)\times\\ x_2y_2z_2\cos(x_2)\cos(y_2)\cos(z_2)\end{array}\right]dx_1dy_1dz_1\right\}dx_2dy_2dz_2 \qquad (3.5.1)$$

而(3.4.2)式積分後的解析解為

$$S=\left(\frac{5}{2}\pi-1\right)^6 \qquad (3.5.2)$$

將此積分式子使用矩形法離散化且取均勻格點，其格點數在每個維度分別是 N_x、N_y、N_z，離散化後式子可寫成

$$S\approx S'=\sum_{m=1}^{N_x+1}\sum_{n=1}^{N_y+1}\sum_{l=1}^{N_z+1}\left\{\sum_{i=1}^{N_x+1}\sum_{j=1}^{N_y+1}\sum_{k=1}^{N_z+1}\left[\begin{array}{l}x_my_nz_l\cos(x_m)\cos(y_n)\cos(z_l)\times\\ x_iy_jz_k\cos(x_i)\cos(y_j)\cos(z_k)\times(\Delta x\Delta y\Delta z)^2\end{array}\right]\right\} \qquad (3.5.3)$$

利用 Fortran 程式可將此式子分別寫成單核 CPU 與 GPU 運行的程式碼(CUDA 撰寫方式在附錄 B、C)，而程式流程圖分別為圖 3.5.1 與圖 3.5.2。

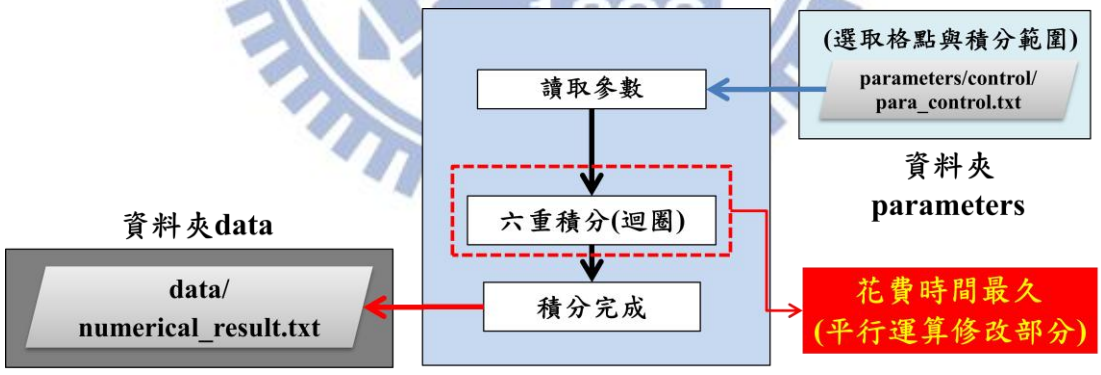


圖 3.5.1 簡單積分程式流程圖。

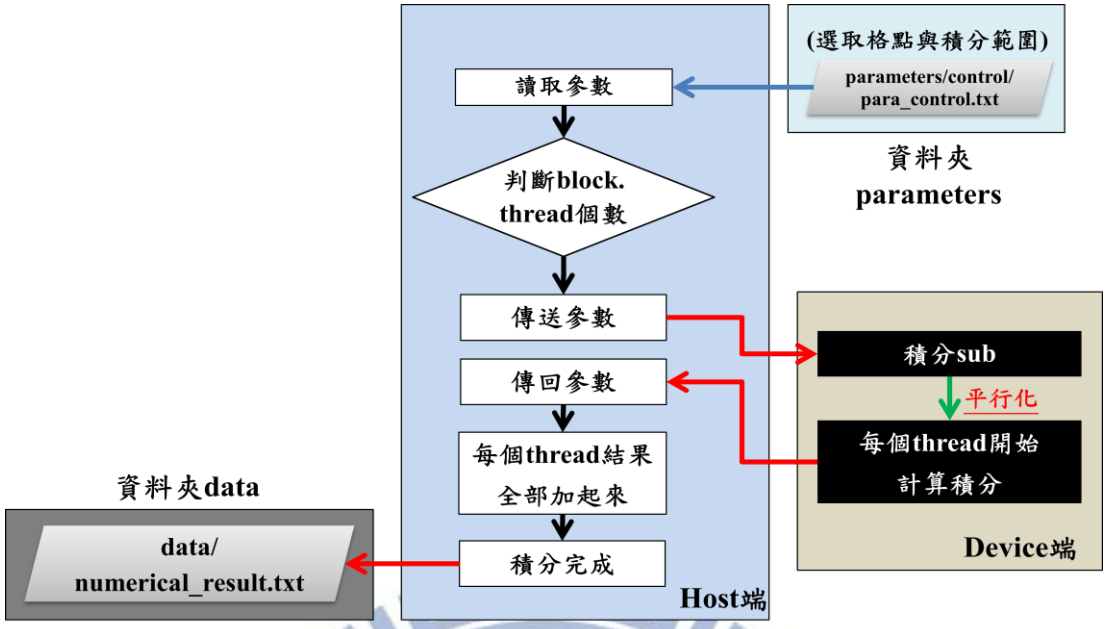


圖 3.5.2 簡單積分程式修改後流程圖。

利用圖 3.5.1 與圖 3.5.2 流程圖撰寫出 Fortran 程式碼後畫出收斂結果，計算方式為 $N_x = N_y = N_z = N$，結果如圖 3.5.3 所示，其中 $N_{3D} = N_x N_y N_z = N^3$ 。

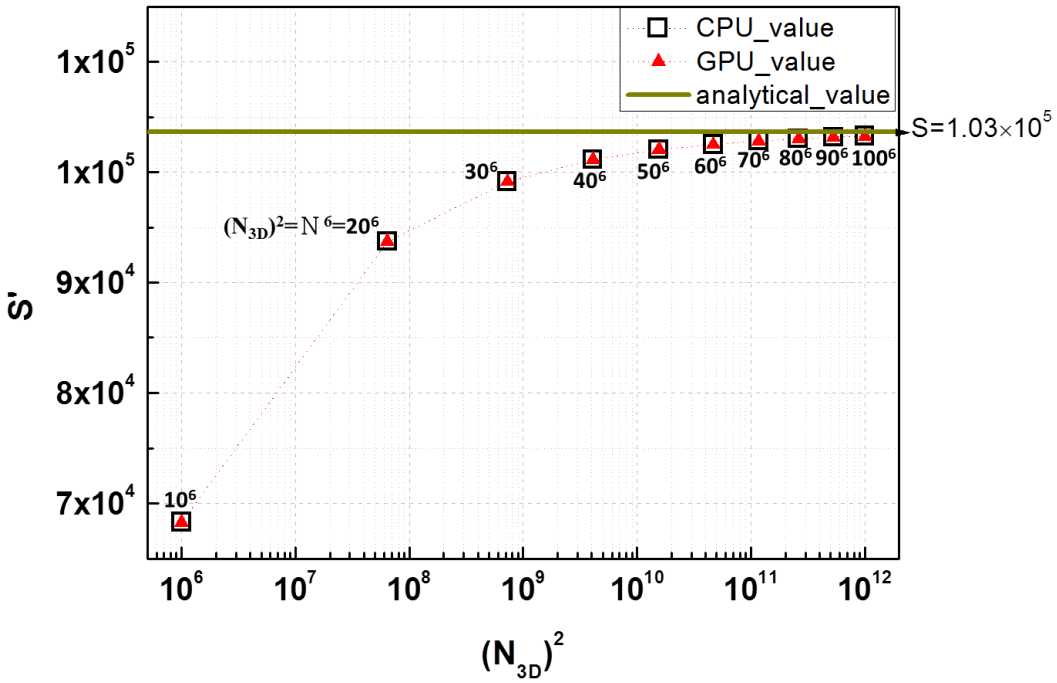


圖 3.5.3 數值收斂結果：$(N_{3D})^2$ 為程式所執行的迴圈個數。

而圖 3.5.3 的數值收斂圖對應到的時間與增快的倍率如圖 3.5.4 與圖 3.5.5 所示。

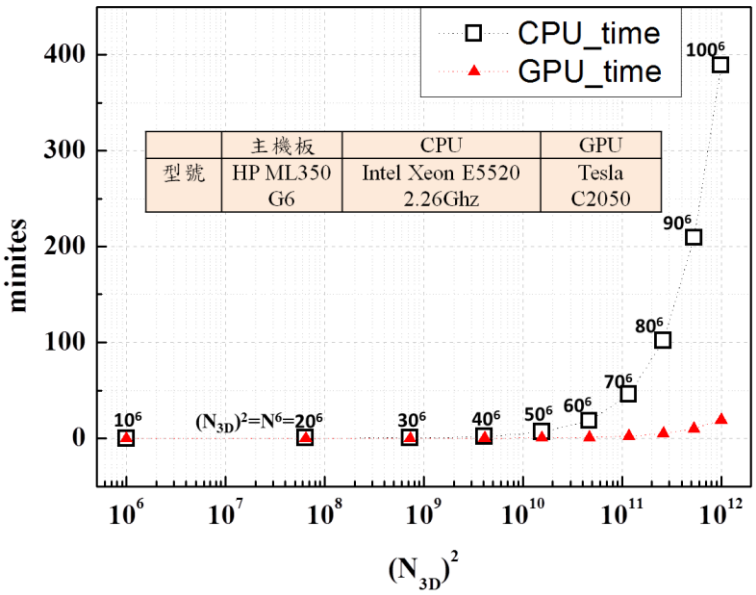


圖 3.5.4 圖 3.5.3 收斂圖對應到的時間。

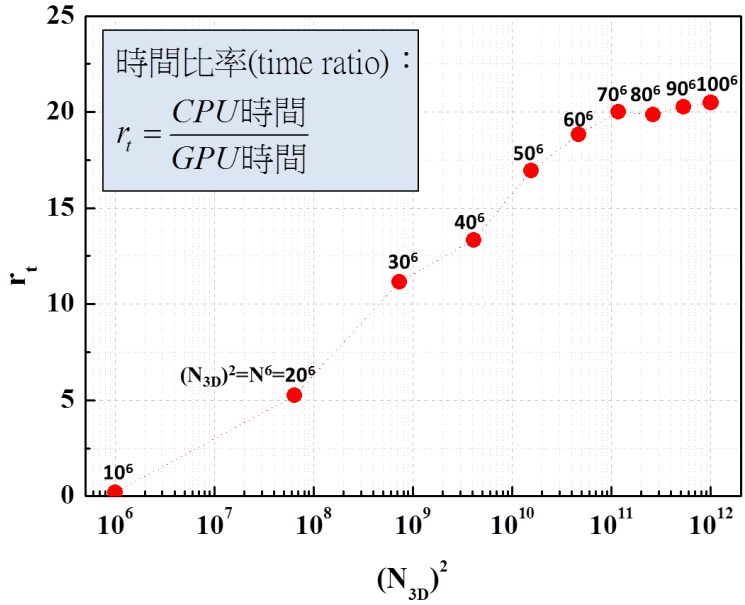


圖 3.5.5 CPU 時間相對應 GPU 時間的倍率。

由圖 3.5.5 可知，此積分式子修改成 GPU 運行的程式碼後，最多能讓時間比率增快到 20 倍。此積分相對於庫侖交互作用程式而言，運算式子較為簡單且需要使用的記憶體較少，因此增快的倍率相較於庫侖積分也較快。

3.6 庫侖交互作用程式寫法

根據 3.3.2 小節電子－電洞間直接庫侖作用矩陣元素與電子－電洞間長程偶極－偶極交換能矩陣元素推導結果，可利用 Fortran 程式計算兩種庫侖作用。觀察(3.3.2.17)式與(3.3.2.29)式，發現兩種庫侖作用都是以六重迴圈所組成的。

使用離散化後的式子可畫一張程式流程圖，如圖 3.6.1 與圖 3.6.2。

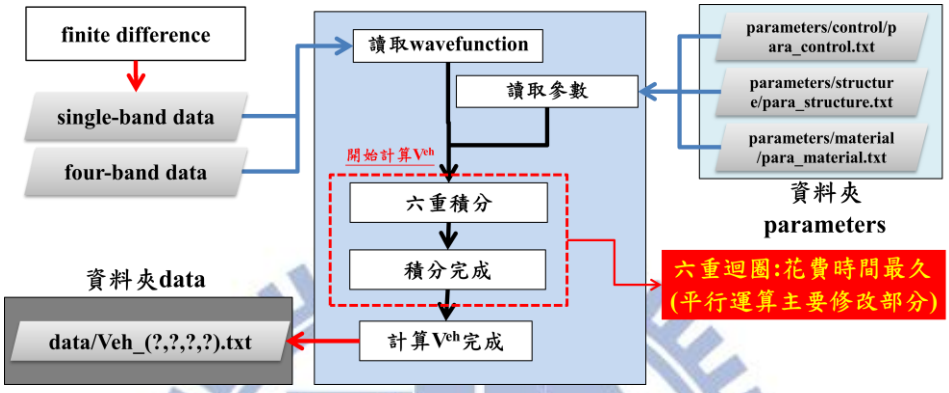


圖 3.6.1 電子－電洞間直接庫侖作用矩陣元素程式流程圖。

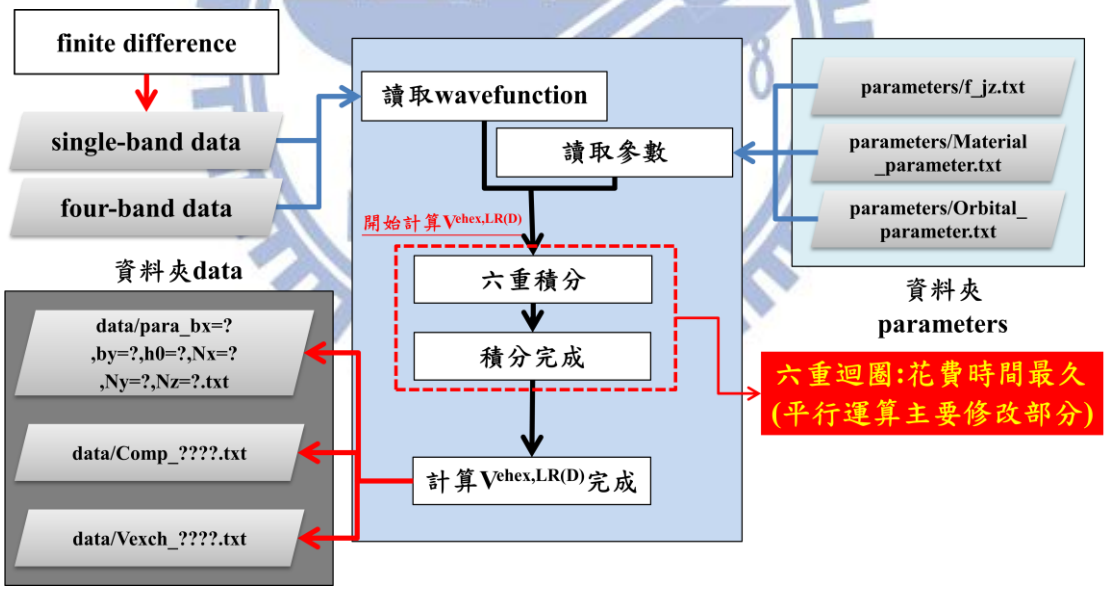


圖 3.6.2 電子－電洞間長程偶極－偶極交換能矩陣元素程式流程圖。

附錄 A、B、C 可知 CUDA 在程式中迴圈要如何撰寫，而庫侖交互作用的程式中利用迴圈做計算最耗費時間，因此我們可以利用迴圈平行運算的寫法來修改庫侖交互作用的程式，圖 3.6.3 與圖 3.6.4 為修改後的流程圖。

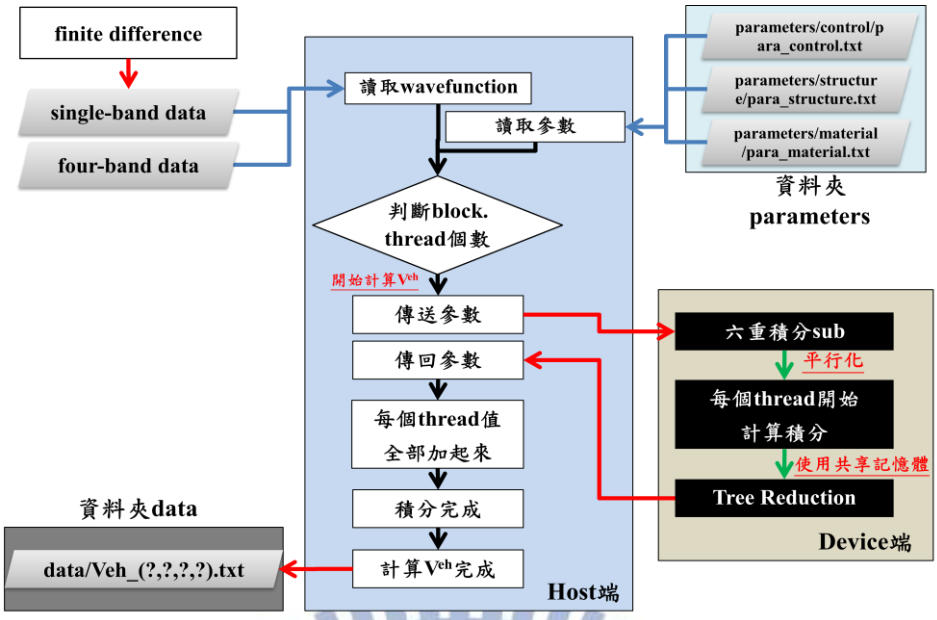


圖 3.6.3 電子－電洞間直接庫侖作用矩陣元素程式修改後流程圖。

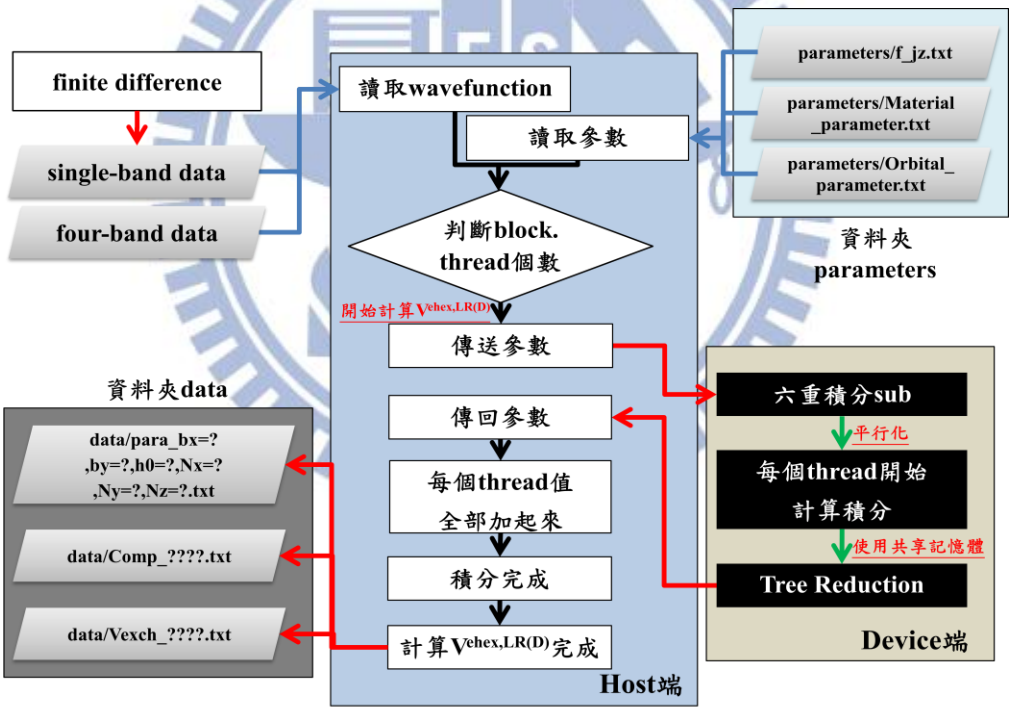


圖 3.6.4 電子－電洞間長程偶極－偶極交換能矩陣元素程式修改後流程圖。

在附錄 D 中有說明撰寫庫侖交互作用程式須注意的部分。而接下來第四章會驗證此程式的可信度，對於單核 CPU 與 GPU 比較結果以及兩者與解析解做比對。

第四章、結果與討論

以下的驗證均假設量子點結構位能為三維拋物線模型且電子、電洞波函數取高斯函數，材料為 InAs(材料參數在文獻[13])且假設三個維度的格點間距Δx=Δy=Δz=Δr 來做討論。

4.1 單核 CPU 與 GPU 的比較

當考慮要計算的庫侖、交換能矩陣元素時，我們將使用單核 CPU 計算會花費極多的時間才可以取得一組完整庫侖、交換能矩陣的數據，在這種情況下，我們引入 GPU 平行運算的技術提高運算的速率，將可以有效的縮短等待數據的時間，也進一步提高了工作的效率。在此節將比較單核 CPU 與 GPU 運算一個庫侖、交換能矩陣元素所花費的時間。

4.1.1 直接庫侖作用

每個矩陣元素計算的時間都差不多，因此我們先計算一個矩陣元素，進而估算創造整個矩陣需花費的時間，經過程式計算後的數值驗證與時間倍率如圖 4.1.1.1、圖 4.1.1.2、圖 4.1.1.3。

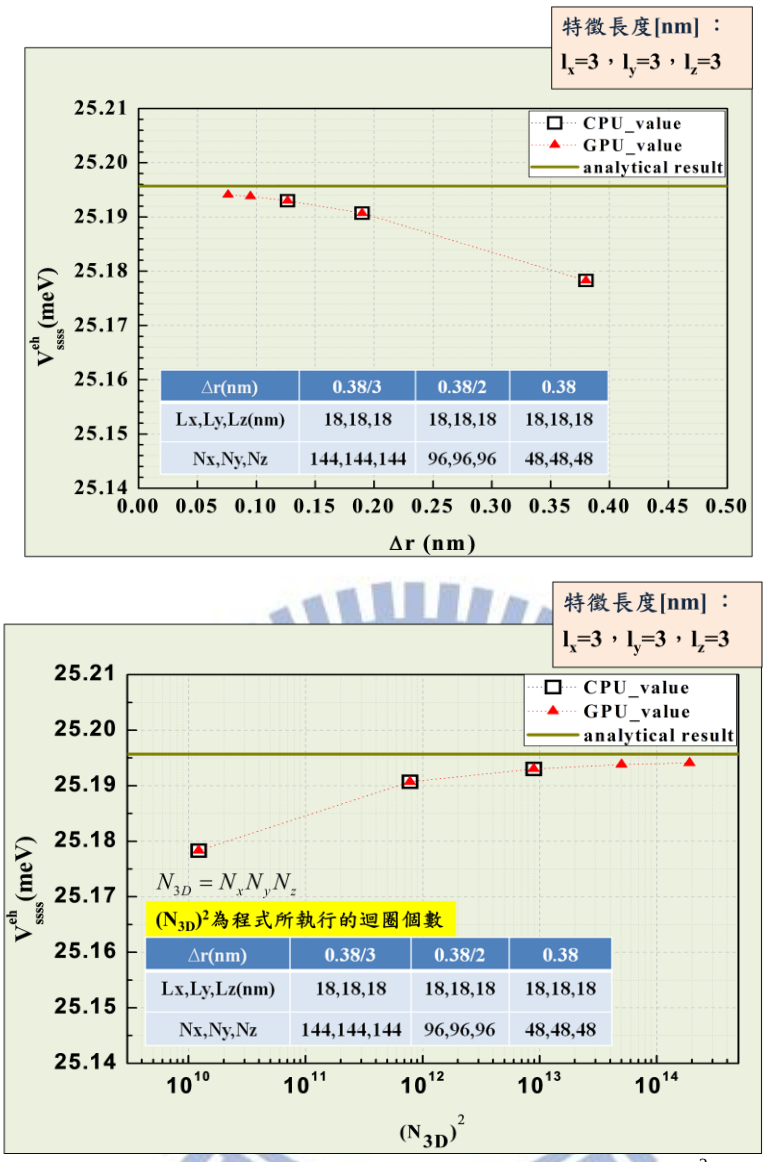
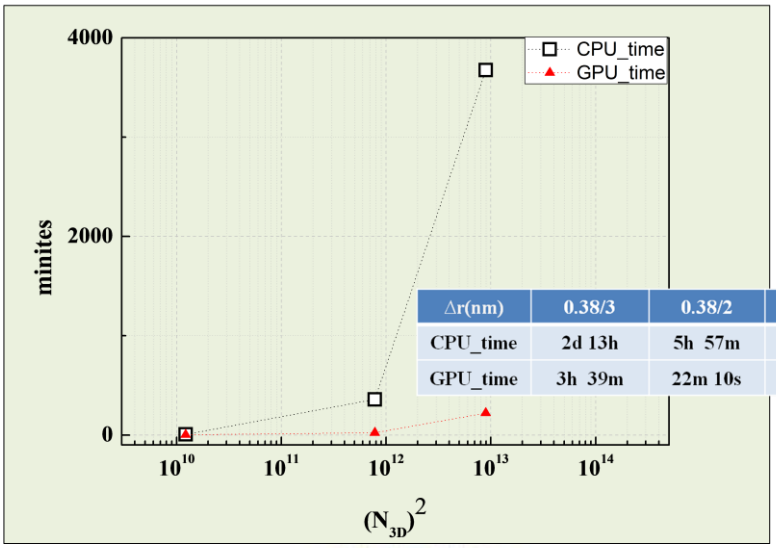


圖 4.1.1.1 單核 CPU 與 GPU 數值收斂圖，(上)Δr－$V^{eh}_{s,s,s,s}$ 關係圖(下) $(N_{3D})^2$－$V^{eh}_{s,s,s,s}$ 關係圖：L_x, L_y, L_z 為三個維度的積分範圍，N_x,N_y,N_z 為三個維度的格點間格數，而 $N_{3D}=N_x \times N_y \times N_z$，$(N_{3D})^2$ 為程式所執行的迴圈個數。



Δr(nm)	0.38/3	0.38/2	0.38
CPU_time	2d 13h	5h 57m	6m 6s
GPU_time	3h 39m	22m 10s	49s

圖 4.1.1.2 單核 CPU 與 GPU 的$(N_{3D})^2$－時間關係圖。

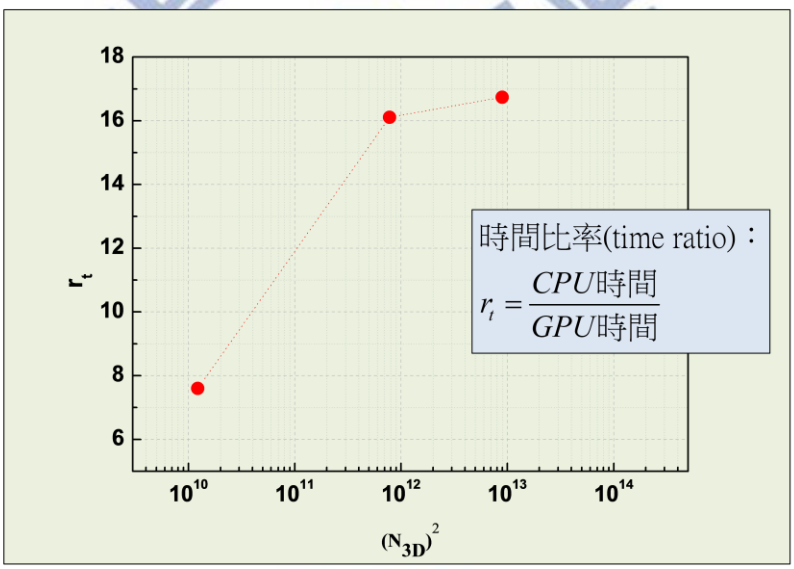


圖 4.1.1.3 比較單核 CPU 與 GPU 的$(N_{3D})^2$－倍率關係圖。

由圖 4.1.1.1 可看出單核 CPU 與 GPU 計算的數值結果是一致的，所以 GPU 運算庫侖矩陣元素的程式是有可信度的，其收斂結果亦趨近解析解；圖 4.1.1.3 中，知道當格點數很少時，Host 端傳遞到 device memory 與創造 CUDA 內 grid 時間大於計算時間，計算速度反而變慢，而考慮格點數越來越多的情形，因為計算量足夠多，使 GPU 裡的 SM 閒置時間越少，因此計算時間越快。

4.1.2 長程偶極－偶極交換能

假設格點間距Δx=Δy=Δz=Δr，且格點取法必須剛好取到長程偶極－偶極交換能與短程交換能之間的邊界，因此Δr=a_0/n，其中 $n \in N$ ，在此我們為了比較以相同的例子分析直接庫侖作用與長程偶極－偶極交換能的結果與時間，先畫出Δr－$V^{eh}_{s,s,s,s}$ 收斂圖，如圖 4.1.2.1。而對於塊材能隙修正成量子點能隙，再畫出a_0≈0.38(nm)能隙修正前後的Δr－$\delta^{-\frac{1}{2},+\frac{3}{2},-\frac{3}{2},+\frac{1}{2},LR(D)}_{s,s,s,s}$ 收斂圖，如圖 4.1.2.2。

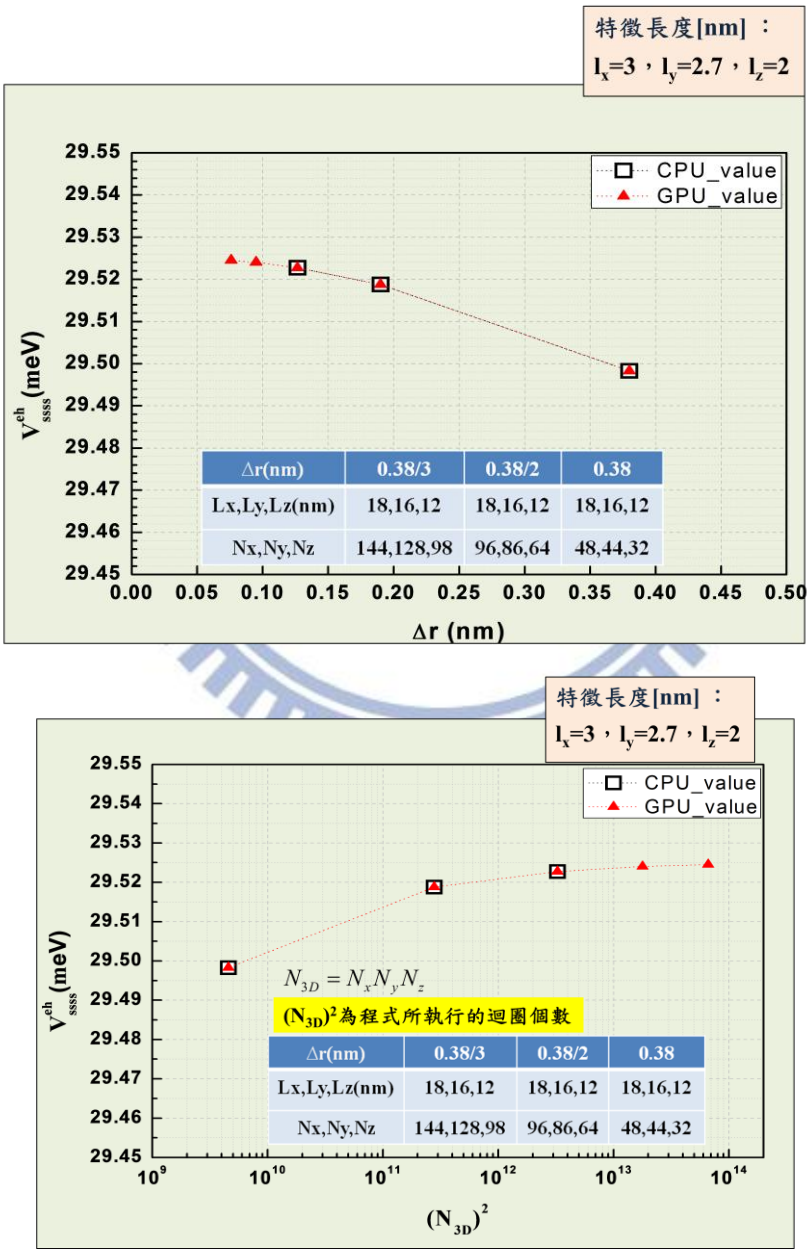


圖 4.1.2.1 單核 CPU 與 GPU 數值收斂圖，(上)Δr－$V^{eh}_{s,s,s,s}$ 關係圖(下) $(N_{3D})^2$－$V^{eh}_{s,s,s,s}$ 關係圖，$(N_{3D})^2$為程式所執行的迴圈個數。

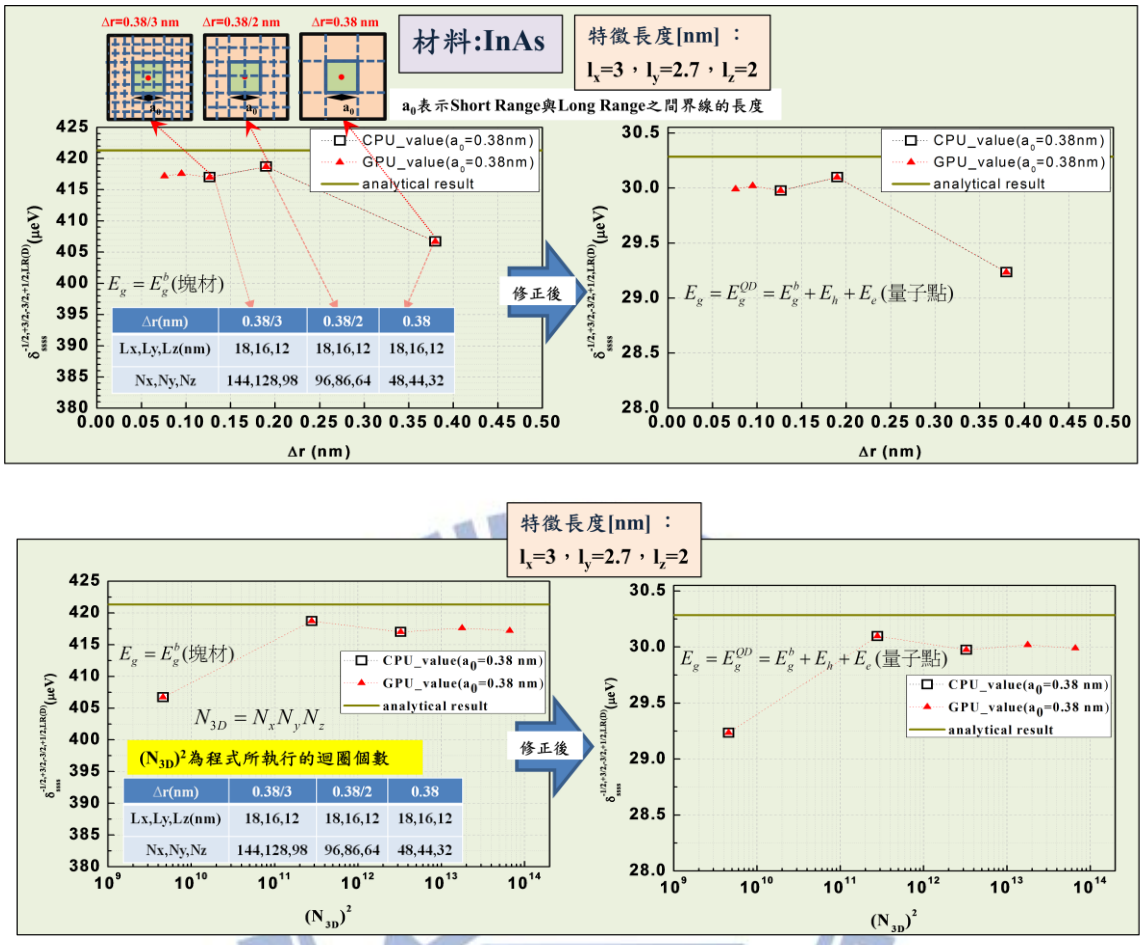


圖 4.1.2.2 單核 CPU 與 GPU 數值收斂圖，能隙修正前後的(上)Δr－$\delta^{-\frac{1}{2},+\frac{3}{2},-\frac{3}{2},+\frac{1}{2}LR(D)}_{s,s,s,s}$ 關係圖，(下) $(N_{3D})^2$ －$\delta^{-\frac{1}{2},+\frac{3}{2},-\frac{3}{2},+\frac{1}{2}LR(D)}_{s,s,s,s}$ 關係圖，$(N_{3D})^2$ 為程式所執行的迴圈個數。

兩者比較數值結果發現可看出單核 CPU 與 GPU 計算的結果是一樣的，而比較直接庫侖作用的收斂能量會高於長程偶極－偶極交換能的收斂能量兩個數量級，因此對於庫侖交互作用而言，能量大小主要來自直接庫侖作用。而接下來則是畫出比較單核 CPU 與 GPU 花費時間與倍率，圖 4.1.2.3 與圖 4.1.2.4 為直接庫侖作用結果所對應的時間與倍率；圖 4.1.2.5 與圖 4.1.2.6 為能隙修正前後的長程偶極－偶極交換能結果所對應的時間與倍率。

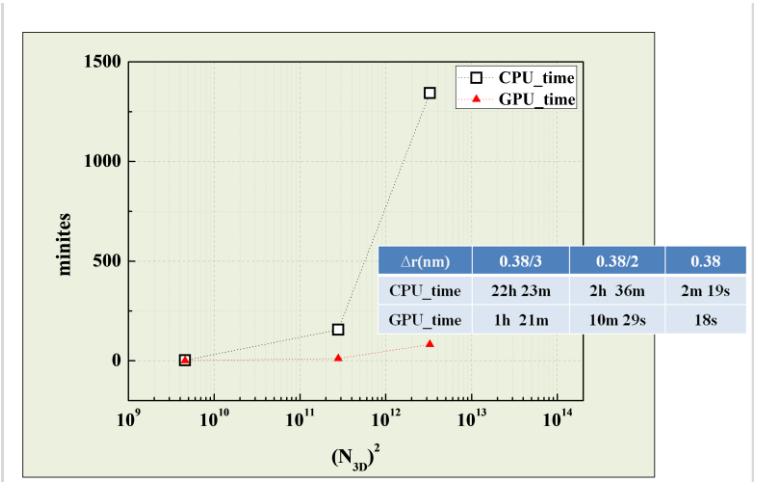


圖 4.1.2.3 比較單核 CPU 與 GPU 的$(N_{3D})^2$－時間關係圖(直接庫侖作用)。

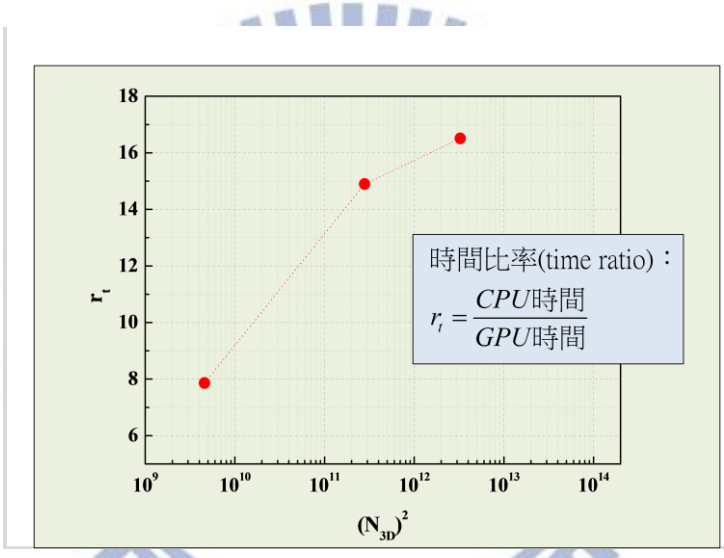


圖 4.1.2.4 比較單核 CPU 與 GPU 的$(N_{3D})^2$－倍率關係圖(直接庫侖作用)。

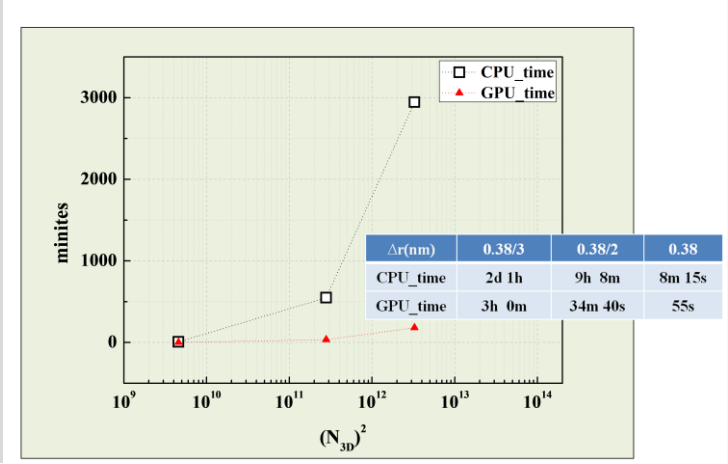


圖 4.1.2.5 比較單核 CPU 與 GPU 能隙修正前後$(N_{3D})^2$－時間關係圖(長程偶極－偶極交換能)。

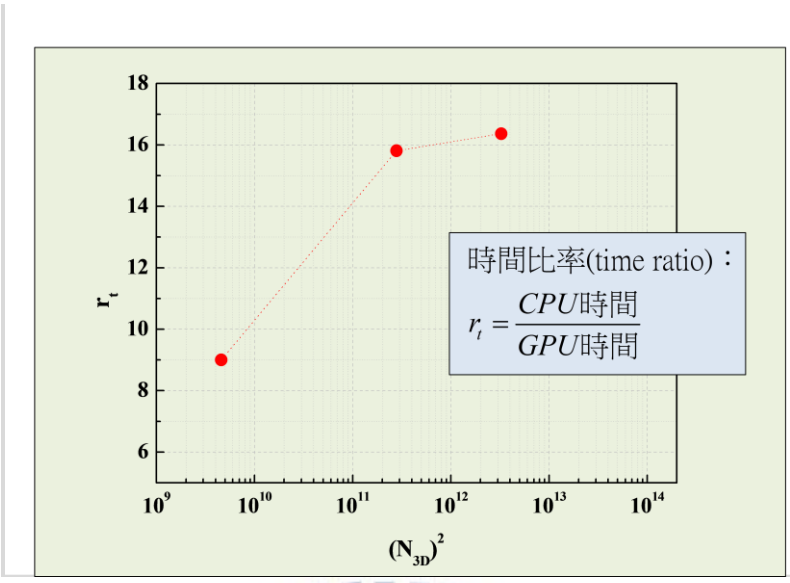


圖 4.1.2.6 比較單核 CPU 與 GPU 能隙修正前後$(N_{3D})^2$－倍率關係圖(長程偶極－偶極交換能)。

比較直接庫侖作用與長程偶極－偶極交換能時間可知，對於相同的例子，由於長程偶極－偶極交換能的積分離散化式子比較複雜，因此計算時間會比直接庫侖作用來的久，但兩者所增加的倍率差異不大。圖 4.1.2.4 與圖 4.1.2.6 表示格點數取越多，GPU 計算速度相對於單核 CPU 會變更快。因此我們確認程式沒問題後可利用此程式驗證長程偶極－偶極交換能對於 $a_0 \approx 0.38$(nm)結果是否合理。

4.2 數值測試

在上一個小節已經測試可以使用 GPU 做數值分析，接下來要使用此程式來做一些數值計算。

4.2.1 直接庫侖作用

此部分要測試對於不同量子點大小收斂性與其格點間距的誤差值，其結果如圖 4.2.1.1、圖 4.2.1.2 與圖 4.2.1.3。

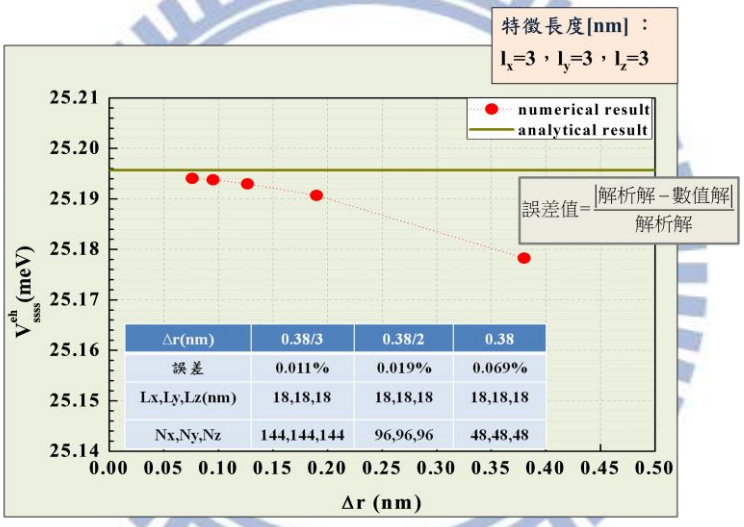


圖 4.2.1.1 l_x=3 nm , l_y=3 nm , l_z=3 nm 的收斂圖。

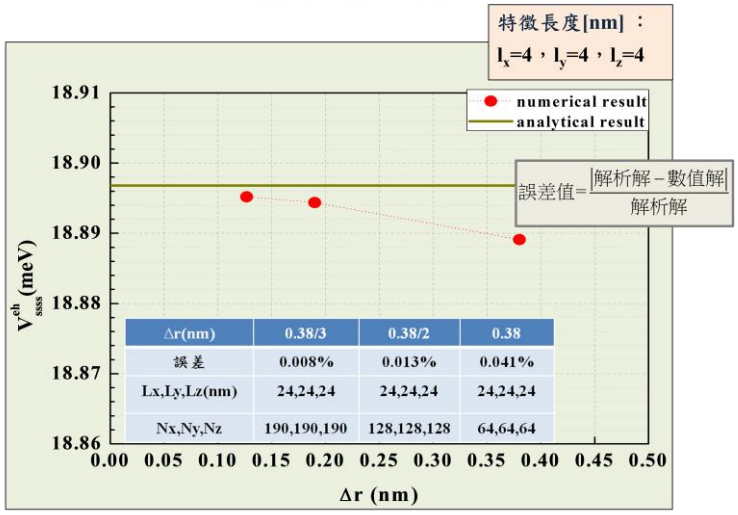


圖 4.2.1.2 l_x=4 nm , l_y=4 nm , l_z=4 nm 的收斂圖。

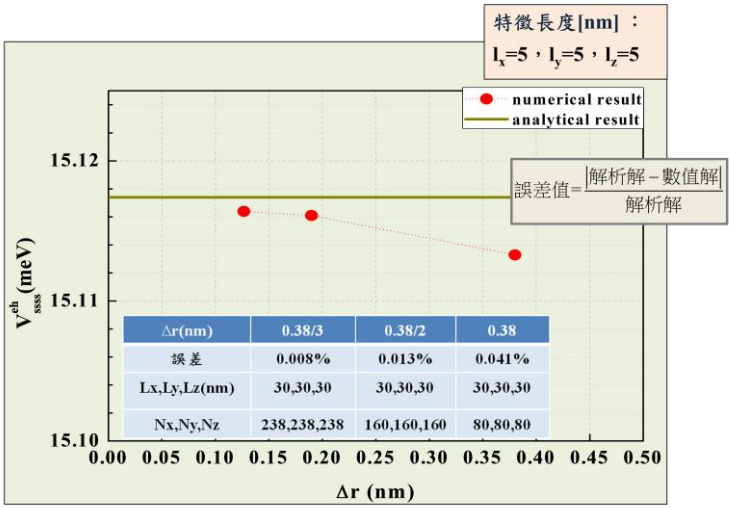


圖 4.2.1.3 l_x=5 nm , l_y=5 nm , l_z=5 nm 的收斂圖。

由以上可知，直接庫侖積分對於不同例子一樣漸漸趨近於收斂，因此證明此程式與積分方法是可信的。由於Δr=0.38 nm 的誤差值非常小，因此我們將測試 l_x=l_y 時，固定Δr=0.38 nm 且畫出 l_x 與直接庫侖積分矩陣元素的關係圖，如圖 4.2.1.4。

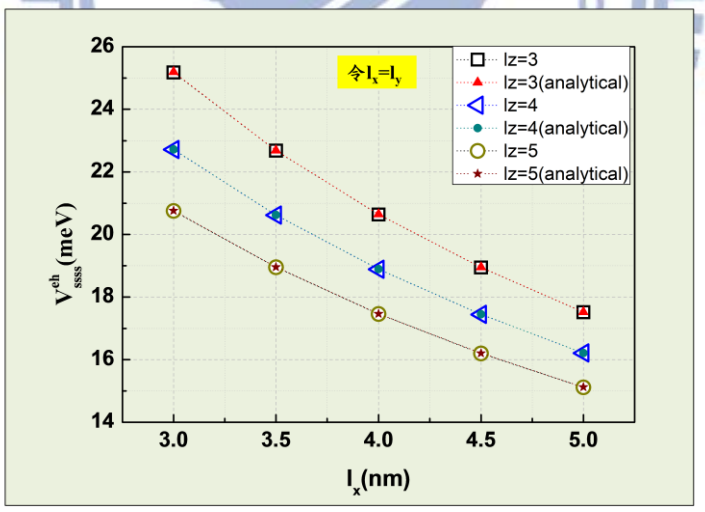


圖 4.2.1.4 l_x 與直接庫侖積分矩陣元素的關係圖。

由圖 4.2.1.4 可以計算的誤差值，如圖 4.2.1.5。

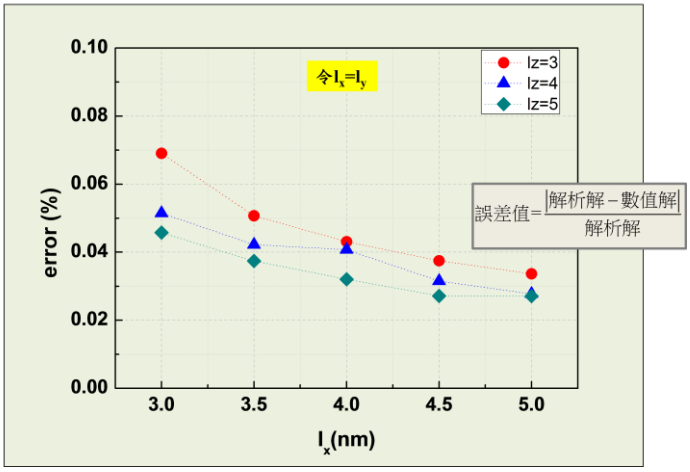


圖 4.2.1.5 計算數值與解析的誤差值。

由圖 4.2.1.5 可知，對於相同的格點間距而言，當量子點越大時格點數越多，數值解與解析解的誤差值結果越小。

4.2.2 長程偶極－偶極交換能

在 3.3.4 小節提到驗證短程交換能與長程交換能之間的界線為 a_0≈0.38 nm 是否合理，由數值結果和(3.3.3.6)式解析解做比對，如圖 4.2.2.1。

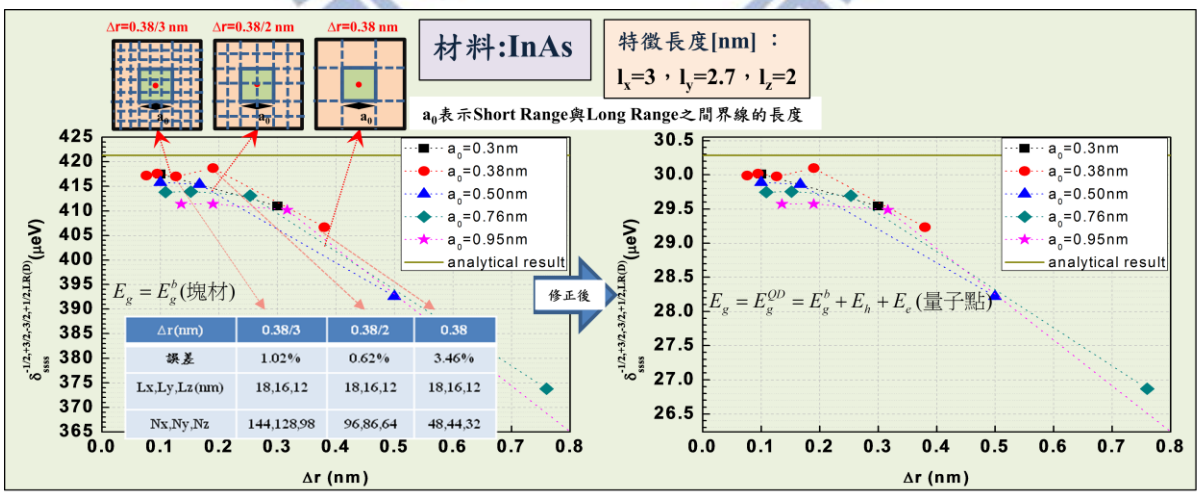


圖 4.2.2.1 不同 a_0 與解析解比對能隙修正前後的 Δr－$\delta_{s,s,s,s}^{-\frac{1}{2},+\frac{3}{2},-\frac{3}{2},+\frac{1}{2},LR(D)}$ 收斂圖。

將圖 4.2.2.1 轉換成迴圈個數($(N_{3D})^2=(N_xN_yN_z)^2$)與 $\delta_{s,s,s,s}^{-\frac{1}{2},+\frac{3}{2},-\frac{3}{2},+\frac{1}{2},LR(D)}$ 的收斂圖，

如圖 4.2.2.2。

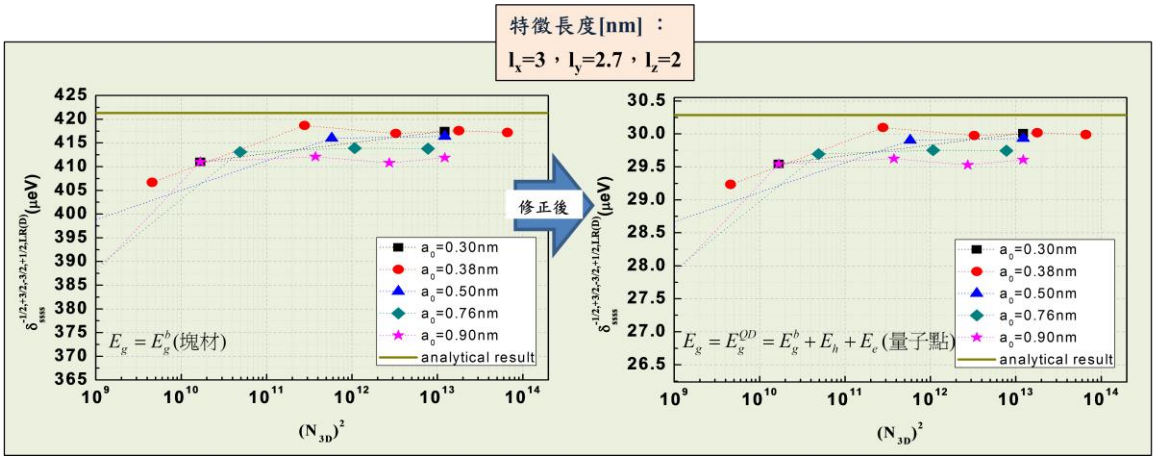


圖 4.2.2.2 不同 a_0 與解析解比對能隙修正前後的 $N_{3D} - \delta_{s,s,s,s}^{-\frac{1}{2},+\frac{3}{2},-\frac{3}{2},+\frac{1}{2},LR(D)}$ 收斂圖。

由圖 4.2.2.1 可知考慮 a_0=0.30 nm、a_0=0.38 nm 與 a_0=0.50 nm 三個例子對 Δr 的收斂值，會發現這三個例子其實數值差異不大且跟解析解比對誤差不大，因此對於 a_0=0.38 nm，數值結果是可接受的。接下來固定 a_0=0.38 nm，量子點不對稱性 $\left(\xi = \frac{l_x}{l_y}\right)$ 為 0.9。改變 l_x、l_z 畫出收斂圖，將數值解與解析解做比對，如圖 4.2.2.3、圖 4.2.2.4。

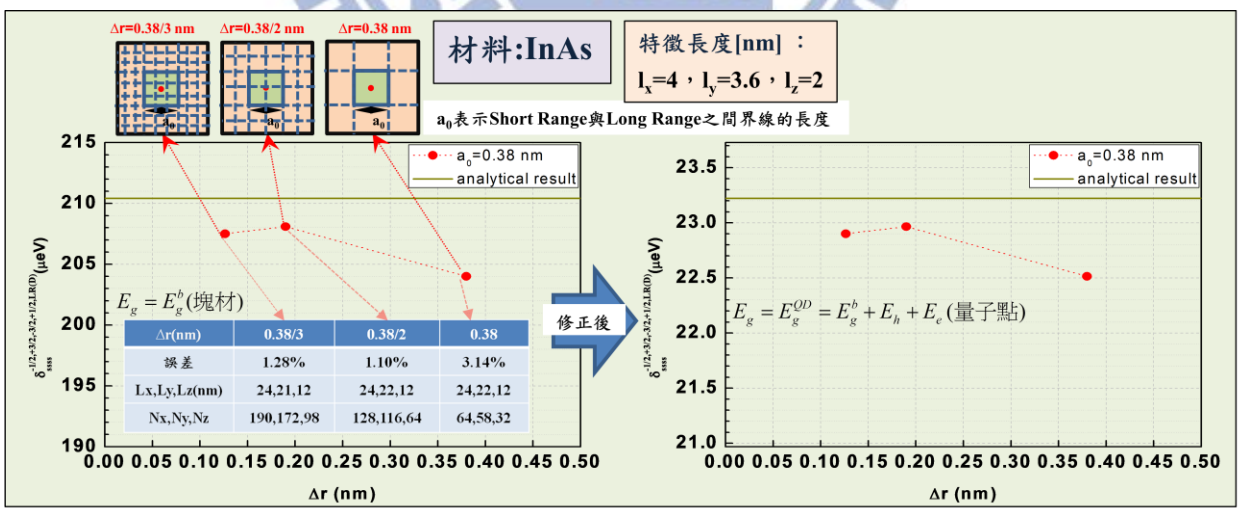


圖 4.2.2.3 l_x=4 nm , l_y=3.6 nm , l_z=2 nm 能隙修正前後的收斂圖。

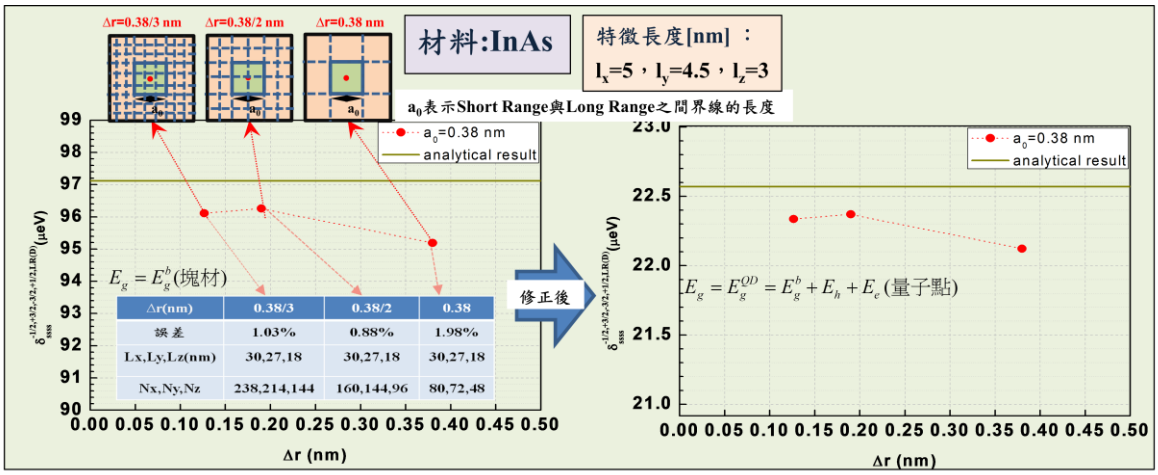


圖 4.2.2.4 l_x=5 nm , l_y=4.5 nm , l_z=3 nm 能隙修正前後的收斂圖。

由圖 4.2.2.1、圖 4.2.2.3 與圖 4.2.2.4 可看出當Δr=0.38 nm 的誤差都在 5%以內是收斂的，因此此程式計算與積分方法是可接受的，接下來以Δr=0.38 nm 來做測試，將固定 l_z=3.0 nm、2.5 nm、2.0 nm 與$\xi = 0.9$畫出 $l_x - \delta^{-\frac{1}{2},+\frac{3}{2},-\frac{3}{2},+\frac{1}{2},LR(D)}_{s,s,s,s}$ 關係圖，如圖 4.2.2.5。

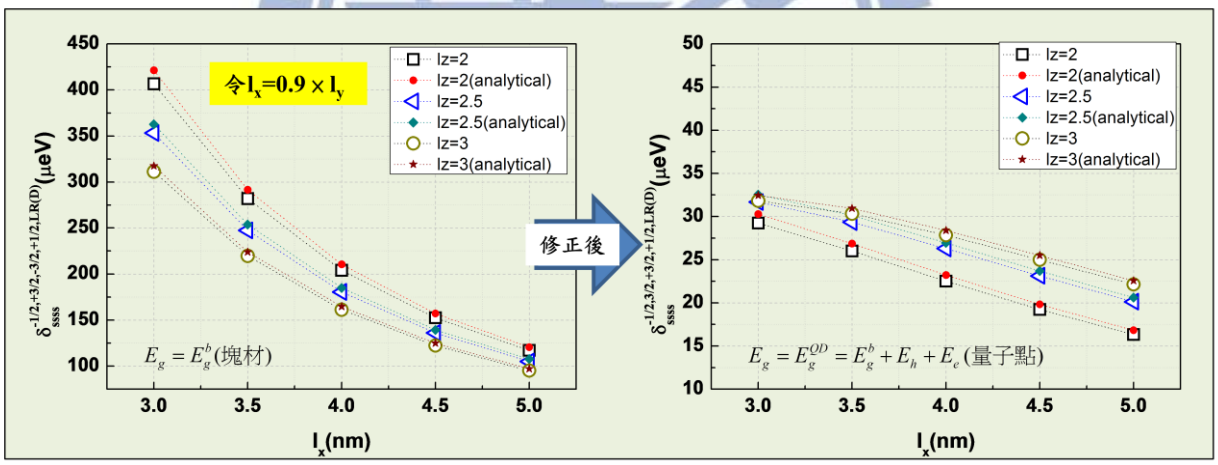


圖 4.2.2.5 能隙修正前後的 $l_x - \delta^{-\frac{1}{2},+\frac{3}{2},-\frac{3}{2},+\frac{1}{2},LR(D)}_{s,s,s,s}$ 關係圖。

而我們可以計算的誤差值，如圖 4.2.2.6。

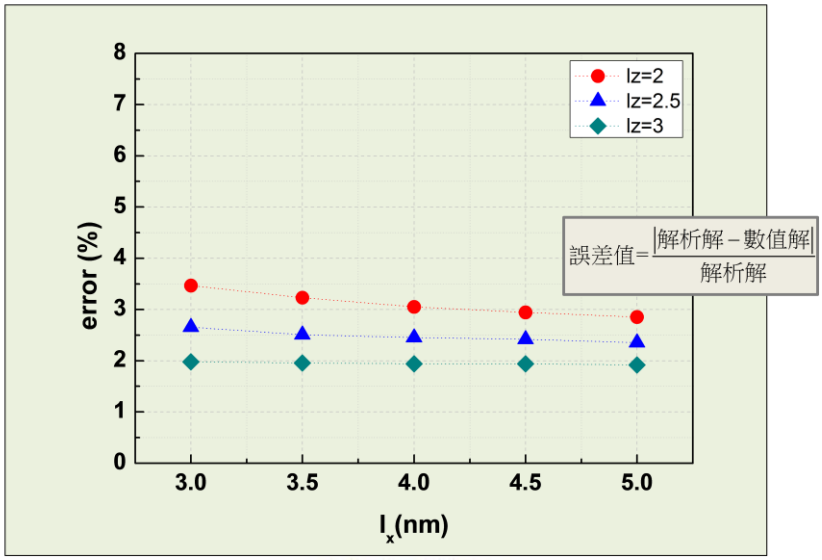


圖 4.2.2.6 計算能隙修正前後數值與解析的誤差值。

從圖 4.2.2.5 可看出取 Δr=0.38 nm 數值解與解析解做比對收斂結果都是可接受的範圍，相較直接庫侖積分雖然收斂性較差，但考慮量子點由小到大所得到的誤差值會越來越小，而收斂性較差的是因為交換能積分式子較直接庫侖積分式子複雜。最後由直接庫侖積分與交換能積分結果可看出：當固定 Δr=0.38 nm 時，庫侖交互作用計算出來的收斂結果是可以被接受的，由此可知將來做數值計算時可固定 Δr=0.38 nm 來做數值計算。

4.3 結果討論

經由上述的比較，可知道當計算庫侖或交換能時，要計算多個六重迴圈，因此計算時間會大幅增加。以格點數 N_x=100,N_y=100,N_z=50 為例，迴圈個數大約是 10^{11}~10^{12} 數量級，如果使用單核 CPU 總體計算一個六重迴圈所需要的運算大約需要 5 小時，要計算整個庫侖以及交換能的時間必定需要花好幾天，而我們使用 GPU 平行運算，計算一個六重迴圈卻只需要 14 分鐘的時間，相較於單核 CPU 運算可以快上 16 倍左右，因此未來在做計算時會使我們計算效率更加迅速。而在計算庫侖交互作用時，採用的量子點為 InAs 與假設波函數為高斯函數當例子，測試後發現 Δr=0.38 nm 時的誤差值是可以接受的範圍且量子點由小到大會漸漸趨近收斂，由於目前測試的量子點並不會太大，因此將來考慮其他大小量子點時，雖然波函數會較複雜，但對於計算後的誤差值與此誤差結果也不會相差太大，而且也符合原子排列方式；關於長程偶極－偶極交換能的計算，為了避免計算到短程交換能，我們將會考慮威格納－塞茲晶胞內部做積分，以 InAs 為例，它的威格納－塞茲晶胞邊長為 0.38 nm，經過測試後發現內部的積分對於整體積分而言，其積分的貢獻很小，因此即使不考慮威格納－塞茲晶胞內部積分其結果與解析解差異不大，因此這個計算方式是我們可以接受的。

第五章、多重能帶

5.1 單一激子多重能帶

以 single-band model 和 four-band model 來描述電子與電洞的行為，再由波包近似法將半導體內電子與電洞波函數寫成

$$\Psi_{i_e}^{e}\left(\vec{r_e}\right)=\sum_{s_z=\pm\frac{1}{2}} g_{s_z,i_e}^{e}\left(\vec{r_e}\right)u_{s_z}^{e}\left(\vec{r_e}\right) \tag{5.1.1}$$

$$\Psi_{j_h}^{h}\left(\vec{r_h}\right)=\sum_{j_z=\pm\frac{3}{2},\pm\frac{1}{2}} g_{j_z,j_h}^{h}\left(\vec{r_h}\right)u_{j_z}^{h}\left(\vec{r_h}\right) \tag{5.1.2}$$

$u_{s_z}^{e}\left(\vec{r_e}\right)$、$u_{j_z}^{h}\left(\vec{r_h}\right)$分別表示電子與電洞的 Bloch's function；$g_{s_z,i_e}^{e}\left(\vec{r_e}\right)$、$g_{j_z,i_h}^{h}\left(\vec{r_h}\right)$分別表示電子與電洞的波包函數(envelope function)；i_e、j_h分別表示電子與電洞所在的軌域。

由於要計算電子與電洞在半導體量子點的能階能量與波函數，因此分成電子與電洞討論：

1. 電子

根據有效質量近似薛丁格方程(Schrödinger equation)及(5.1.1)式可得

$$\left[\frac{p^2}{2m_e^*}+V_{QD}^{e}\left(\vec{r_e}\right)\right]g_{s_z,i_e}^{e}\left(\vec{r_e}\right)=E_{i_e}^{e}g_{s_z,i_e}^{e}\left(\vec{r_e}\right) \tag{5.1.3}$$

m_e^*為電子的有效質量，p為電子動量算符，$V_{QD}^{e}\left(\vec{r_e}\right)$為電子在導電帶受量子效應(結構與形狀)侷限的位能，$E_{i_e}^{e}$為電子動能。因此決定$V_{QD}^{e}\left(\vec{r_e}\right)$後，利用有限差分法(finite difference method)[13]或基底展開等方法對角化矩陣即可得到不同能態所對應的電子動能與波包函數。

2. 電洞

為了使式子不會看起來繁雜，利用表 5.1.1 表示電洞在價電帶的

Bloch'sfunction：

表 5.1.1 電子 Bloch function 的符號表示法。

符號	Bloch's function	軌道型式
$\left\|\frac{3}{2},\frac{3}{2}\right\rangle_h$	$u^h_{j_z=+3/2}\left(\vec{r_h}\right)$	$-\frac{1}{\sqrt{2}}\left\|P_x+iP_y;\uparrow\right\rangle$
$\left\|\frac{3}{2},\frac{1}{2}\right\rangle_h$	$u^h_{j_z=+1/2}\left(\vec{r_h}\right)$	$-\frac{1}{\sqrt{6}}\left\|P_x+iP_y;\downarrow\right\rangle+\sqrt{\frac{2}{3}}\left\|P_z;\uparrow\right\rangle$
$\left\|\frac{3}{2},\frac{-1}{2}\right\rangle_h$	$u^h_{j_z=-1/2}\left(\vec{r_h}\right)$	$\frac{1}{\sqrt{6}}\left\|P_x-iP_y;\uparrow\right\rangle+\sqrt{\frac{2}{3}}\left\|P_z;\downarrow\right\rangle$
$\left\|\frac{3}{2},\frac{-3}{2}\right\rangle_h$	$u^h_{j_z=-3/2}\left(\vec{r_h}\right)$	$\frac{1}{\sqrt{2}}\left\|P_x-iP_y;\downarrow\right\rangle$

資料來源：文獻[19]。

根據 four-band model，在 Luttinger-Kohnk·p model 裡的有效質量近似薛丁格方程可寫成

$$\left[H_h+V^h_{QD}\left(\vec{r}_h\right)\times \mathrm{I}_{4\times4}\right]\begin{pmatrix} g^h_{j_z=\frac{3}{2},i_h} \\ g^h_{j_z=\frac{1}{2},i_h} \\ g^h_{j_z=\frac{-1}{2},i_h} \\ g^h_{j_z=\frac{-3}{2},i_h} \end{pmatrix}=E^h_{i_h}\begin{pmatrix} g^h_{j_z=\frac{3}{2},i_h} \\ g^h_{j_z=\frac{1}{2},i_h} \\ g^h_{j_z=\frac{-1}{2},i_h} \\ g^h_{j_z=\frac{-3}{2},i_h} \end{pmatrix} \tag{5.1.4}$$

$V^h_{QD}\left(\vec{r}_h\right)$ 為電洞在價電帶受量子效應(結構與形狀)侷限的位能，$\mathrm{I}_{4\times4}$ 為 4×4 單位矩陣，$E^h_{i_h}$ 為電洞動能。而以 Bloch's function 當基底的 Luttinger-Kohn Hamiltonian 型式

$$H_h=\begin{array}{c} \begin{matrix} \left|\frac{3}{2},\frac{3}{2}\right\rangle_h & \left|\frac{3}{2},\frac{1}{2}\right\rangle_h & \left|\frac{3}{2},\frac{-1}{2}\right\rangle_h & \left|\frac{3}{2},\frac{-3}{2}\right\rangle_h \end{matrix} \\ \begin{bmatrix} P_k+Q_k & -S_k & R_k & 0 \\ -S_k^{\dagger} & P_k-Q_k & 0 & R_k \\ R_k^{\dagger} & 0 & P_k-Q_k & S_k \\ 0 & R_k^{\dagger} & S_k^{\dagger} & P_k+Q_k \end{bmatrix}_h \end{array} \tag{5.1.5}$$

其中

$$P_k = -\frac{\hbar^2\gamma_1}{2m_0}\left(\frac{\partial^2}{\partial x^2}+\frac{\partial^2}{\partial y^2}+\frac{\partial^2}{\partial z^2}\right)$$

$$Q_k = -\frac{\hbar^2\gamma_2}{2m_0}\left(\frac{\partial^2}{\partial x^2}+\frac{\partial^2}{\partial y^2}-2\frac{\partial^2}{\partial z^2}\right)$$

$$R_k = -\frac{\hbar^2}{2m_0}\left[-\sqrt{3}\gamma_2\left(\frac{\partial^2}{\partial x^2}-\frac{\partial^2}{\partial y^2}\right)+i2\sqrt{3}\gamma_3\frac{\partial^2}{\partial x\partial y}\right]$$

$$S_k = -\frac{\hbar^2\gamma_3}{m_0}\sqrt{3}\left(\frac{\partial}{\partial x}-i\frac{\partial}{\partial y}\right)\frac{\partial}{\partial z}$$

γ_1、γ_2、γ_3為材料參數，m_0為電子靜止的質量。當決定$V_{QD}^{h}(\vec{r}_h)$後，利用有限差分法(finite difference method)[13]或基底展開等方法創造矩陣後可將矩陣對角化得到不同能態所對應的電洞動能與波包函數。

5.2 庫侖交互作用

電子－電洞間直接庫侖作用矩陣元素定義

$$V_{i_e,j_h,k_h,l_e}^{eh} = \iint d\vec{r}_1 d\vec{r}_2 \Psi_{i_e}^{e*}(\vec{r}_1)\Psi_{j_h}^{h*}(\vec{r}_2)\frac{e^2}{4\pi\varepsilon_0\varepsilon|\vec{r}_1-\vec{r}_2|}\Psi_{k_h}^{h}(\vec{r}_2)\Psi_{l_e}^{e}(\vec{r}_1) \qquad (5.2.1)$$

而電子－電洞間交換能矩陣元素定義

$$V_{i_e,j_h,k_h,l_e}^{ehxc} = \iint d\vec{r}_1 d\vec{r}_2 \Psi_{i_e}^{e*}(\vec{r}_2)\Psi_{j_h}^{h}(\vec{r}_2)\frac{e^2}{4\pi\varepsilon_0\varepsilon|\vec{r}_1-\vec{r}_2|}\Psi_{k_h}^{h*}(\vec{r}_1)\Psi_{l_e}^{e}(\vec{r}_1) \qquad (5.2.2)$$

e表電子電量，ε表材料介電常數(dielectric constant)，$\Psi_{i_e}^{e}(\vec{r})$、$\Psi_{j_h}^{h}(\vec{r})$表電子與電洞在空間中的波函數。

由於在 3.3.2 小節有探討庫侖交互作用利用波包近似法與矩形法離散化方式化簡電子－電洞間直接庫侖作用與偶極－偶極交換能，在此直接寫下化簡後的式子。

電子－電洞間直接庫侖作用[1]：

$$V_{i_e,j_h,k_h,l_e}^{eh} = \sum_{j_z=\pm3/2,\pm1/2}\sum_{m=1}^{N_x+1}\sum_{n=1}^{N_y+1}\sum_{l=1}^{N_z+1}\left[\begin{array}{l} g_{j_h,j_z}^{h*}(x_m,y_n,z_l)\,g_{k_h,j_z}^{h}(x_m,y_n,z_l) \\ \times U_{i_e,l_e}(x_m,y_n,z_l)\times\Delta x\Delta y\Delta z \end{array}\right] \qquad (5.2.1)$$

$$U_{i_e,l_e}(x_m,y_n,z_l)=\frac{e}{4\pi\varepsilon_0\varepsilon}\sum_{s_z=\pm1/2}\left\{\begin{array}{l}\sum_{i\neq m}^{N_x+1}\sum_{j\neq n}^{N_y+1}\sum_{k\neq l}^{N_z+1}\frac{g^{e*}_{i_e,s_z}(x_i,y_j,z_k)\,g^{e}_{l_e,s_z}(x_i,y_j,z_k)\Delta V}{\sqrt{(x_i-x_m)^2+(y_j-y_n)^2+(z_k-z_l)^2}}\\ +F_s\times\sum_{m=1}^{N_x+1}\sum_{n=1}^{N_y+1}\sum_{l=1}^{N_z+1}\left[\begin{array}{l}g^{e*}_{i_e,s_z}(x_m,y_n,z_l)\\ \times g^{e}_{l_e,s_z}(x_m,y_n,z_l)\end{array}\right]\end{array}\right\} \tag{5.2.2}$$

$$F_s\equiv\int_{-\Delta x/2}^{\Delta x/2}\int_{-\Delta y/2}^{\Delta y/2}\int_{-\Delta z/2}^{\Delta z/2}\frac{dxdydz}{\sqrt{x^2+y^2+z^2}} \tag{5.2.3}$$

$U_{i_e,l_e}(x_m,y_n,z_l)$為電荷密度$\left(\Psi^{e*}_{i_e}\Psi^{e}_{l_e}\right)$所建立的電位，$F_s$ 定義單位晶胞內的積分式。

電子－電洞間偶極－偶極交換能[15]：

首先我們可以由廖禹淮學長博士生研究計畫構想書[20]第 3 章可知電子、電洞觀點轉換的概念，因此(5.2.2)式可寫成

$$V^{ehex}_{i_e,j_h,k_h,l_e}=\sum_{s_z,j_z,j_z',s_z'}f_{-j_z}f^{*}_{-j_z'}\delta^{s_z,-j_z,-j_z',s_z'}_{i_e,j_h,k_h,l_e} \tag{5.2.4}$$

$$\begin{aligned}\delta^{s_z,-j_z,-j_z',s_z'}_{i_e,j_h,k_h,l_e}=&\int\int dr_1^3dr_2^3\left[g^{e}_{i_e;s_z}(\vec{r}_2)u_{s_z}(\vec{r}_2)\right]^{*}\left[g^{v}_{j_h;-j_z}(\vec{r}_2)u_{-j_z}(\vec{r}_2)\right]\frac{e^2}{4\pi\varepsilon_0\varepsilon_b\left|\vec{r}_1-\vec{r}_2\right|}\\&\times\left[g^{v}_{k_h;-j_z'}(\vec{r}_1)u_{-j_z'}(\vec{r}_1)\right]^{*}\left[g^{e}_{l_e;s_z'}(\vec{r}_1)u_{s_z'}(\vec{r}_1)\right]\end{aligned} \tag{5.2.5}$$

$g^{v}_{j_v,j_z}(\vec{r})$為空軌域的波包函數，f_{-j_z}為選擇因子。接下來(5.2.5)式可根據 3.3.2 小節的推導可推導出結果，其結果為

$$V^{ehex,LR(D)}_{i_e,j_h,k_h,l_e}=\sum_{s_z,j_z,j_z',s_z'}f_{-j_z}f^{*}_{-j_z'}\delta^{s_z,-j_z,-j_z',s_z',LR(D)}_{i_e,j_h,k_h,l_e} \tag{5.2.6}$$

$$\delta^{s_z,-j_z,-j_z',s_z',LR(D)}_{i_e,j_h,k_h,l_e}=\sum_{m=1}^{N_x+1}\sum_{n=1}^{N_y+1}\sum_{l=1}^{N_z+1}\left\{\begin{array}{l}\Delta\Omega^2\times g^{e}_{i_e;s_z}{}^{*}(x_m,y_n,z_l)\,g^{v}_{j_h;-j_z}(x_m,y_n,z_l)\\ \times\sum_{i\neq m}^{N_x+1}\sum_{j\neq n}^{N_y+1}\sum_{k\neq l}^{N_z+1}\left[\begin{array}{l}g^{v}_{k_h;-j_z'}{}^{*}(x_i,y_j,z_k)\,g^{e}_{l_e;s_z'}(x_i,y_j,z_k)\\ \times W^{s_z,-j_z,-j_z',s_z'}_{LR(D)}\left(\sqrt{(x_i-x_m)^2+(y_j-y_n)^2+(z_k-z_l)^2}\right)\end{array}\right]\end{array}\right\} \tag{5.2.7}$$

$$\begin{aligned}&W^{s_z,-j_z,-j_z',s_z'}_{LR(D)}\left(\sqrt{(x_i-x_m)^2+(y_j-y_n)^2+(z_k-z_l)^2}\right)\equiv\\&\sum_{\alpha,\beta}C_{s_z,-j_z,\alpha}C_{s_z',-j_z',\beta}{}^{*}w^{LR(D)}_{\alpha,\beta}\left(\sqrt{(x_i-x_m)^2+(y_j-y_n)^2+(z_k-z_l)^2}\right)\end{aligned} \tag{5.2.8}$$

$$w_{\alpha,\beta}^{LR(D)}\left(\sqrt{(x_i-x_m)^2+(y_j-y_n)^2+(z_k-z_l)^2}\right)=$$

$$\frac{e^2}{4\pi\varepsilon_0\varepsilon_b}\left(\frac{\hbar^2}{E_g^2}\frac{E_p}{2m_0}\right)\times\left\{\frac{\alpha\cdot\hat{\beta}}{\left|\sqrt{(x_i-x_m)^2+(y_j-y_n)^2+(z_k-z_l)^2}\right|^3}-\frac{3\Delta\alpha\cdot\Delta\beta}{\left|\sqrt{(x_i-x_m)^2+(y_j-y_n)^2+(z_k-z_l)^2}\right|^5}\right\} \quad (5.2.9)$$

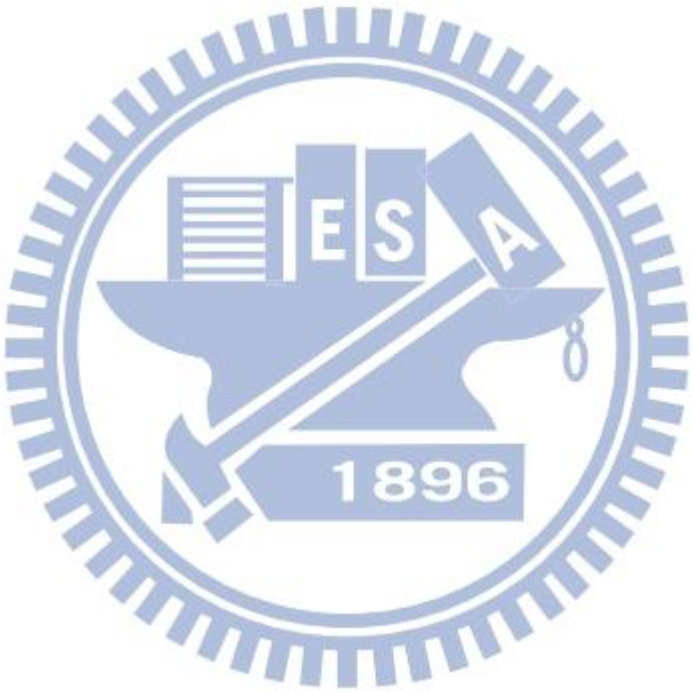
其中 $C_{s_z,-j_z,\alpha}$ 、 $C_{s_z',-j_z',\beta}$ 係數如表 5.2.1。

表 5.2.1 每一組不同的自旋、自旋軌道角動量與方向對應的係數。

$s_z,-j_z,\alpha$	$C_{s_z,-j_z,\alpha}$	$s_z,-j_z,\alpha$	$C_{s_z,-j_z,\alpha}$
$+\frac{1}{2},+\frac{3}{2},x$	$-\sqrt{1/2}$	$-\frac{1}{2},+\frac{3}{2},x$	0
$+\frac{1}{2},+\frac{3}{2},y$	$-i\sqrt{1/2}$	$-\frac{1}{2},+\frac{3}{2},y$	0
$+\frac{1}{2},+\frac{3}{2},z$	0	$-\frac{1}{2},+\frac{3}{2},z$	0
$+\frac{1}{2},+\frac{1}{2},x$	0	$-\frac{1}{2},+\frac{1}{2},x$	$-\sqrt{1/6}$
$+\frac{1}{2},+\frac{1}{2},y$	0	$-\frac{1}{2},+\frac{1}{2},y$	$-i\sqrt{1/6}$
$+\frac{1}{2},+\frac{1}{2},z$	$\sqrt{2/3}$	$-\frac{1}{2},+\frac{1}{2},z$	0
$+\frac{1}{2},-\frac{1}{2},x$	$\sqrt{1/6}$	$-\frac{1}{2},-\frac{1}{2},x$	0
$+\frac{1}{2},-\frac{1}{2},y$	$-i\sqrt{1/6}$	$-\frac{1}{2},-\frac{1}{2},y$	0
$+\frac{1}{2},-\frac{1}{2},z$	0	$-\frac{1}{2},-\frac{1}{2},z$	$\sqrt{2/3}$
$+\frac{1}{2},-\frac{3}{2},x$	0	$-\frac{1}{2},-\frac{3}{2},x$	$\sqrt{1/2}$
$+\frac{1}{2},-\frac{3}{2},y$	0	$-\frac{1}{2},-\frac{3}{2},y$	$-i\sqrt{1/2}$
$+\frac{1}{2},-\frac{3}{2},z$	0	$-\frac{1}{2},-\frac{3}{2},z$	0

在此會發現 3.3.2 小節所推出來的結果由於只考慮單一能帶，因此只需計算一次六重迴圈；四能帶需要計算六重迴圈的次數可由(5.2.1)式與(5.2.6)式得知，

因此考慮自旋軌道偶合角動量(j_z, j_z'),排列組合後需計算 16 次($j_z, j_z' = \pm\frac{1}{2}, \pm\frac{3}{2}$ 排列組合總個數=4×4)，將來假如考慮更多的能帶時，需要計算更多六重迴圈，因此研究使用 GPU 計算庫命交互作用，可以使我們將來能計算的極限更加的提升。

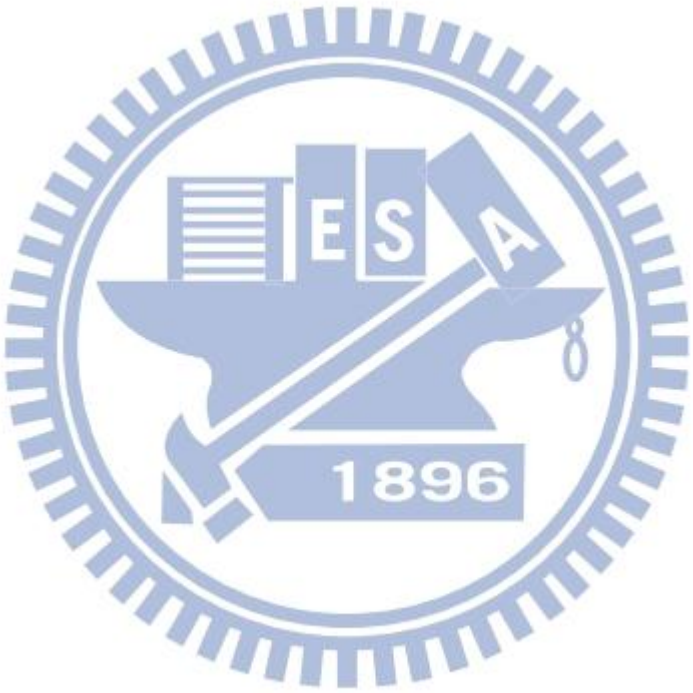


第六章、結論

本論文探討利用量子點的結構位能為三維拋物線位能配合$\vec{k}\cdot\vec{p}$單能帶和有效質量近似法計算激子的庫侖交互作用，經過二次量子化後的 Hamiltonian 且選擇一組無交互作用的激子系統組態為基底，可展開為庫侖、交換能矩陣。而原本利用單核 CPU 計算庫侖交互作用需花費許多時間，為了使效率提升必須要有相應的措施，我們選擇使用 GPU 來提升效率。而使用 GPU 又必須考慮相關的技術才能使程式達到最大效益，例如考慮 Host 端與 Device 端之間溝通問題、了解如何設定 block 與 thread 是最有效率以及編寫方式等…問題，經過測試 GPU 平行運算對迴圈具有較好的加速效果，庫侖積分本質上是數個六重迴圈的程式，因此將此程式修改可達到相當大的利益。當考慮將數值計算到收斂時，數值結果相對於單核 CPU 可使程式速率將會提升 10~20 倍，因此未來要計算一組完整的庫侖、交換能矩陣時，可使我們在同樣的時間內可以得到更多有用的資訊進而分析。

再來就是對於庫侖交互作用的計算，我們改變 Δr 來做數值收斂，採用的量子點為 InAs 與假設波函數為高斯函數當例子，而測試後結果曲線會漸漸逼近解析解且誤差在可接受範圍內，而我們在更進一步討論要如何做計算會更有效率，由於原子排列方式為 Δr=0.38 nm，因此選擇了 Δr=0.38 nm 來分析，而分析後結果發現量子點由小到大數值解與解析解誤差也在可接受範圍內，因此將來做數值運算我們可以選擇 Δr=0.38 nm 來做計算；有關長程偶極－偶極交換能的計算方法，交換能可以分成長程作用與短程作用，為了避免程式計算到短程作用的部分，因此要有一些因應措施，而程式中的的寫法是使用疊代的方式來做相加，當在做疊代時不能計算到兩粒子處於同一個威格納－塞茲晶胞內(定義一個威格納－塞茲晶胞的平均長度為 a_0)，可利用此方法來測試不考慮到威格納－塞茲晶胞內的積分，一樣符合我們預期的結果，而我們取 InAs 的平均長度(a_0=0.38 nm)來做測試，使用已推出的解析解與數值解做比對，而測試結果證明 a_0=0.3 nm、0.38 nm 與 0.5 nm 時，三者的收斂值差異不大且曲線漸漸逼近解析解以及誤差都在可接

受範圍內，此結果可知威格納－塞茲晶胞內的積分對於整體的積分貢獻很小，因此 a_0=0.38 nm 的結果是可信的。因此我們可選擇 Δr=0.38 nm 以及 a_0=0.38 nm 來計算直接庫侖矩陣與交換能矩陣，可使我們更快、更有效率的計算出結果。



附錄 A、CUDA compiler 方法

Fortran 程式碼 compiler 方式會根據不同開發者而有差異，表 3.3.1 是使用 CentOS 5.4 作業系統在終端機介面 compiler 的方法：

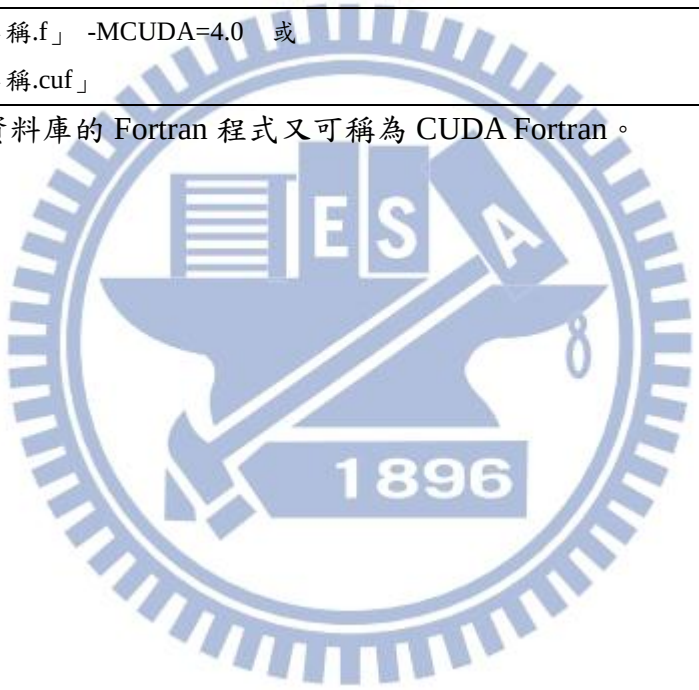
表 A.1 不同廠商在 CentOS 5.4 一般 compiler 方式。

開發廠商	一般 compiler 方法
GNU	gfortran 「檔案名稱.f」
Intel	ifort 「檔案名稱.f」
The Portland Group(PGI)	pgf95 「檔案名稱.f」

目前 Fortran 只有 PGI 公司開發的 PGI Fortran 支援 CUDA 資料庫，因此我們將使用 PGI Fortran 來撰寫程式碼，以下為 PGI Fortran 使用 CUDA 資料庫 compiler 程式碼的方法：

```
pgf95 「檔案名稱.f」 -MCUDA=4.0  或
pgf95 「檔案名稱.cuf」
```

加入 CUDA 資料庫的 Fortran 程式又可稱為 CUDA Fortran。



附錄 B-1、CUDA 基本程式編寫

由於將一般的寫法修改成 GPU 的寫法細節有點煩雜，因此先由最簡單的數值寫入參數中來分析，圖 B-1.1 為程式流程圖，而圖 B-1.2 為將陣列中的四個元素寫入數值的程式碼，我們將每個元素分別利用 GPU 的四個核心來分別寫入。

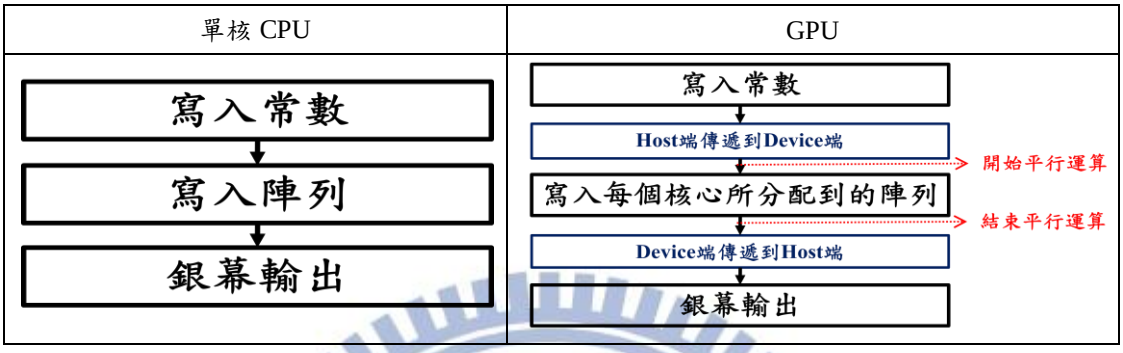


圖 B-1.1 使用 CPU 與 GPU 的程式流程圖。

單核 CPU

```
PROGRAM test
  IMPLICIT NONE

  DOUBLE PRECISION         :: numout(4)
  DOUBLE PRECISION         :: conin
  DOUBLE PRECISION         :: istat

  numout=0D0
  conin=1D0

  CALL test_CPU(numout,conin)

  WRITE(*,*) 'numout(1)=',numout(1)
  WRITE(*,*) 'numout(2)=',numout(2)
  WRITE(*,*) 'numout(3)=',numout(3)
  WRITE(*,*) 'numout(4)=',numout(4)
  WRITE(*,*) 'conin=     ',conin

END PROGRAM test
```

GPU[Host 端]

```
PROGRAM test
  USE CUDAFOR
  USE test_GPU_m
  IMPLICIT NONE
                                          1.宣告(常數、變數)
  DOUBLE PRECISION         :: numout(4)
  DOUBLE PRECISION,DEVICE  :: numout_d(4)
  DOUBLE PRECISION         :: conin
  DOUBLE PRECISION         :: istat

  numout=0D0
  conin=1D0
                                          2.Host端與Device端記憶體互傳
  numout_d=numout

  CALL test_GPU<<<5,6>>> &                3.呼叫kernel
  (numout_d,conin)                        (CUDA副程式)

  numout=numout_d

  istat=CUDAFREE(numout_d)                4.釋放Device記憶體

  WRITE(*,*) 'numout(1)=',numout(1)
  WRITE(*,*) 'numout(2)=',numout(2)
  WRITE(*,*) 'numout(3)=',numout(3)
  WRITE(*,*) 'numout(4)=',numout(4)
  WRITE(*,*) 'conin=     ',conin

END PROGRAM test
```

單核 CPU

```
SUBROUTINE test_CPU(numout,conin)
  IMPLICIT NONE

  DOUBLE PRECISION         :: numout(4)
  DOUBLE PRECISION         :: conin

  numout(1)=2D0

  numout(2)=3D0

  numout(3)=4D0

  numout(4)=5D0

END SUBROUTINE test_CPU
```

GPU[Device 端(kernel)]

```
                                          1.使用MODULE指令及定義CUDA副程式
MODULE test_GPU_m
CONTAINS
  ATTRIBUTES(GLOBAL) SUBROUTINE test_GPU(numout_d,conin)
    IMPLICIT NONE                         2.宣告(常數、變數)
    DOUBLE PRECISION         :: numout_d(4)
    DOUBLE PRECISION,VALUE   :: conin
    INTEGER                  :: i         3.使用新增指令將
                                          thread與block編號
    i=(BLOCKIDX%X-1)*BLOCKDIM%X+THREADIDX%X

    IF(i==1)THEN                          4.利用平行運算的概念做計算
    numout_d(i)=2D0
    END IF

    IF(i==2)THEN
    numout_d(i)=3D0
    END IF

    IF(i==3)THEN
    numout_d(i)=4D0
    END IF

    IF(i==4)THEN
    numout_d(i)=5D0
    END IF

  END SUBROUTINE test_GPU
END MODULE test_GPU_m
```

單核 CPU[執行結果]

```
[      @HPML350      ]$ pgf95 test_CPU.f
[      @HPML350      ]$ ./a.out
 numout(1)=    2.000000000000000
 numout(2)=    3.000000000000000
 numout(3)=    4.000000000000000
 numout(4)=    5.000000000000000
 conin=        1.000000000000000
[      @HPML350      ]$
```

GPU[執行結果]

```
[      @HPML350      ]$ pgf95 test_GPU.cuf
[      @HPML350      ]$ ./a.out
 numout(1)=    2.000000000000000
 numout(2)=    3.000000000000000
 numout(3)=    4.000000000000000
 numout(4)=    5.000000000000000
 conin=        1.000000000000000
[      @HPML350      ]$
```

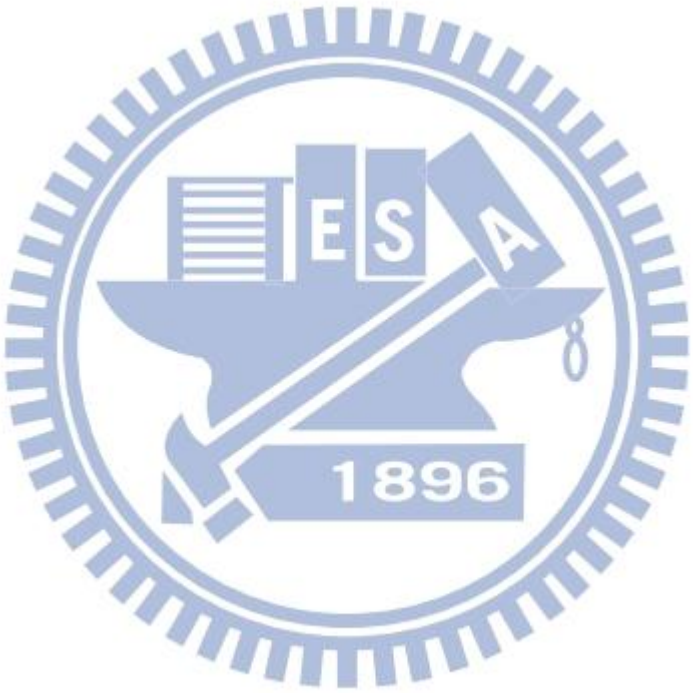
圖 B-1.2 一般 Fortran 修改成 CUDA Fortran 簡單範例：紅色框框表示需要修改或增加的部分。

由於 CUDA 是加入 Fortran 的資料庫，表示在程式中會新增幾個指令，而一般 Fortran 指令還是可以繼續沿用。表 B.1 是針對 Fortran 使用 CUDA 資料庫需要增加或修改程式碼：

表 B-1.1 Fortran 使用 CUDA 需要修改或增加的部分。

Host 端	Device 端
1. 宣告(常數、變數)	1. 使用 MODULE 指令及定義 CUDA 副程式
2. Host 端與 Device 端記憶體互傳	2. 宣告(常數、變數)
3. 呼叫 kernel(CUDA 副程式)	3. 使用新增指令將 thread 與 block 編號
4. 釋放 device 記憶體	4. 利用平行運算的概念做計算

接下來要解釋 CUDA Fortran 程式碼要如何編輯。



附錄 B-2、Host 端撰寫方式

在 Host 端執行程式流程大致上分成 5 個步驟：(如圖 B-1.1 所示)

1. Host 端配置 device 記憶體
2. 上傳資料到 device 記憶體
3. 呼叫 kernel 運算
4. 下載資料回 host 記憶體
5. 釋放 device 記憶體

GPU[Host 端]

```
PROGRAM test
  USE CUDAFOR
  USE test_GPU_m
  IMPLICIT NONE

  DOUBLE PRECISION         :: numout(4)
  DOUBLE PRECISION,DEVICE  :: numout_d(4)
  DOUBLE PRECISION         :: conin
  DOUBLE PRECISION         :: istat

  numout=0D0
  conin=1D0

  numout_d=numout

  CALL test_GPU<<<5,6>>> &
  (numout_d,conin)

  numout=numout_d

  istat=CUDAFREE(numout_d)

  WRITE(*,*) 'numout(1)=',numout(1)
  WRITE(*,*) 'numout(2)=',numout(2)
  WRITE(*,*) 'numout(3)=',numout(3)
  WRITE(*,*) 'numout(4)=',numout(4)
  WRITE(*,*) 'conin=    ',conin

END PROGRAM test
```

1.Host端配置device記憶體

2.上傳資料到device記憶體

3.呼叫kernel

4.下載資料回Host端

5.釋放device記憶體

圖 B-2.1 在圖 B-1.1 中 Host 端程式碼進行分析。

將圖 B-2.1 程式碼分成 4 個部分來說明，分別是宣告方式、傳送方式、呼叫 kernel 與釋放 device 記憶體。

一、宣告方式

在 CUDA Fortran 宣告有分 2 個部分，一個是由 Host 端配置 device 記憶體(如圖 B-2.2 的 Host 端變數「numout_d(4)」與常數「conin」)，另一個是在 Device

端宣告接收 Host 端的參數(此部分下一個小節會介紹)，表 B-2.1 要先介紹 CUDA 在 Host 端宣告方式：

表 B-2.1 CUDA 在 Host 端宣告方式。

	變數	常數
整數	INTEGER,DEVICE	INTEGER
浮點數(單精準度)	REAL, DEVICE	REAL
浮點數(雙精準度)	DOUBLE PRECISION, DEVICE	DOUBLE PRECISION
複數(單精準度)	COMPLEX, DEVICE	COMPLEX
複數(雙精準度)	DOUBLE COMPLEX, DEVICE	DOUBLE COMPLEX
邏輯	LOGICAL, DEVICE	LOGICAL
字串	CHARACTER*1, DEVICE	CHARACTER *1, DEVICE

這裡要注意的是字串宣告，在 CUDA 裡最多只能定義 1 個文字的字串，而它的常數宣告方式也與其他常數宣告方式不同。

GPU[Host 端]

```
DOUBLE PRECISION,DEVICE :: numout_d(4)
DOUBLE PRECISION        :: conin
```

→ 1.Host端配置device記憶體

圖 B-2.2 在圖 B-2.1 中 CUDA 在 Host 端的宣告方式。

```
[      @HPML350              ]$ pgf95 transfer.cuf
PGF90-S-0155-CUDA device routines do not support character strings with length >
 1  (transfer.cuf: 32)
  0 inform,   0 warnings,   1 severes, 0 fatal for transfer_gpu
[      @HPML350              ]$
```

圖 B-2.3 字串長度設定錯誤導致 compiler 沒過。

二、傳送方式[*Host*端(*CPU*) ↔ *Device*端(*GPU*)]

GPU 運算必須先由 Host 端上傳資料到 device 記憶體，之後會由 Device 端平行運算，等運算完後下載資料回 Host 端記憶體，如寫法圖 B-2.4。「numout_d=numout」表示由 CPU 讀取或計算完的資料「numout」上傳到 Device 記憶體「numout_d」，「numout=numout_d」則表示由 GPU 計算完的資料「numout_d」下載回 Host 端記憶體「numout」。

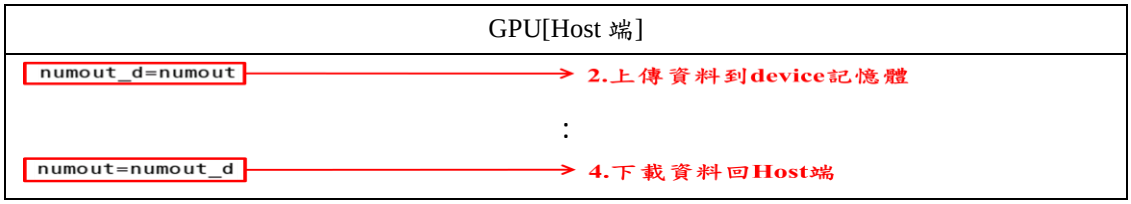


圖 B-2.4 在圖 B-2.1 中 Host 端與 Device 端參數傳送方式。

三、呼叫 kernel

由 2.3.1 小節可得知，CUDA 可分為 grid、block、thread，呼叫 kernel 意思就是創造 grid 使用 GPU 做計算，以下是在程式的編寫方式：

CALL 「副程式檔名」<<<「block 個數」,「thread 個數」>>>(「輸入輸出參數」)

範例如圖 B-2.5 所示，因此我們可以設定 block 個數與 thread 個數。

圖 B-2.5 在圖 B-2.1 中呼叫 kernel。

由於 CUDA 是三維結構，所以呼叫三維 CUDA 可寫成以下形式：

CALL「副程式檔名」<<<DIM3(「block_x 個數」,「block_y 個數」,「block_z 個數」),DIM3(「thread_x 個數」,「thread_y 個數」,「thread_z 個數」)>>>(「輸入輸出的參數」)

呼叫三維 CUDA 要利用 DIM3 函式，必須使用 CUDAFOR 資料庫，如圖 B-2.6。

```
USE CUDAFOR
                    :
CALL test_GPU<<<DIM3(1,1,1),DIM3(4,4,4)>>> &
(numout_d,conin)
```

圖 B-2.6 呼叫三維 CUDA 寫法：「USA CUDAFOR」表示使用 CUDAFOR 資料庫。

四、釋放 device 記憶體

釋放 device 記憶體必須利用 CUDAFREE 函數來釋放，因此要使用 CUDAFOR 資料庫，如圖 B-2.7 所示。「USE CUDAFOR」表示使用 CUDAFOR 資料庫，而釋放 device 記憶體撰寫方式為：

「Host 端宣告的參數」=CUDAFREE(「Device 端要釋放記憶體的參數」)

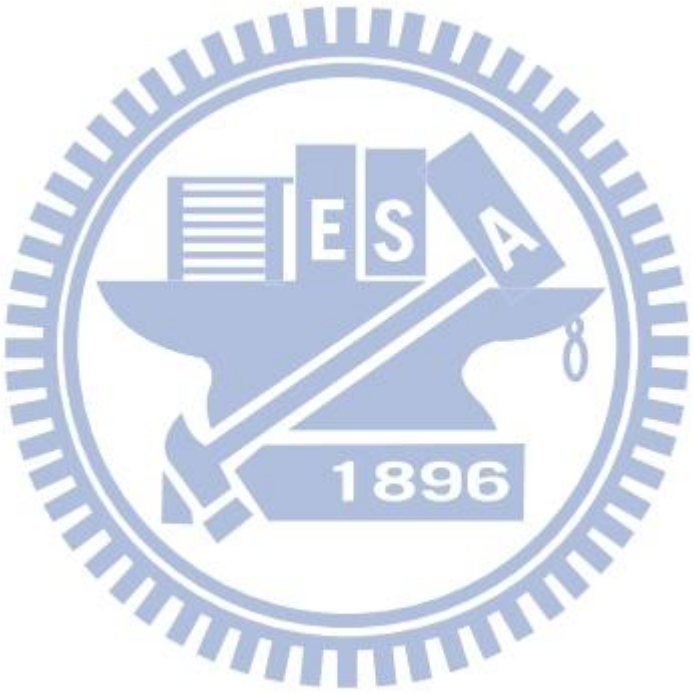
圖 B-2.7 中的「istat」為 Host 端宣告的任意參數即可，「numout_d」為需要釋放

的 device 記憶體參數。

GPU[Host 端]

```
USE CUDAFOR
                                                    :
istat=CUDAFREE(numout_d)  ──────────→  5.釋放device記憶體
```

圖 B-2.7 在圖 B-2.1 中宣告 device 記憶體利用 CUDAFREE 函數釋放。



附錄 B-3、Device 端撰寫方式

Device 端執行程式流程大致上分成 4 個步驟：(如圖 B-3.1 所示)

1. 利用 MODULE 將整個 kernel 包裝成 GPU 可運行的格式
2. Device 端宣告接收 Host 端的參數
3. 將 block 與 thread 編號
4. 平行運算

GPU[Device 端(kernel)]

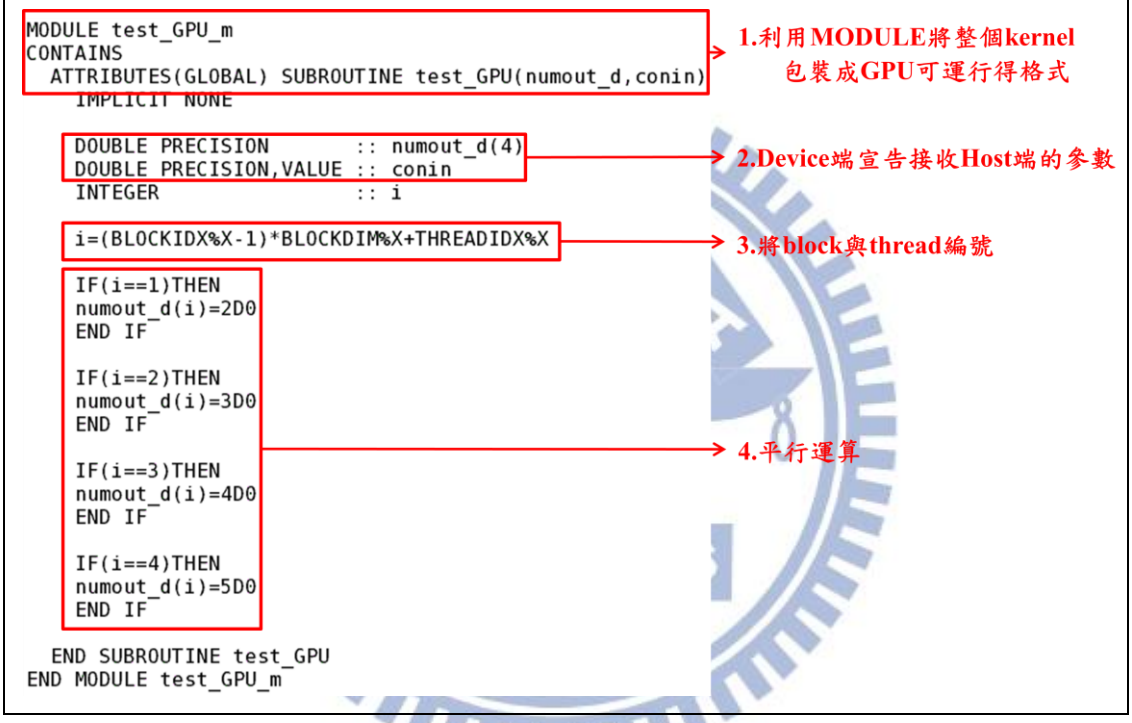


圖 B-3.1 在圖 B-1.1 中 Device 端程式碼進行分析。

將圖 B-3.1 分成 4 個部分說明，分別是包裝方式、宣告方式、CUDA 編號、計算想法。

一、包裝方式

使用 GPU 計算必須將 Device 端的程式碼利用 MODULE 包裝起來，compiler 後會產生 

檔案，之後在 Host 端利用「USE test_gpu_m」來使用此 mod，如圖 B-3.3；定義 CUDA 副程式與一般 Fortran 副程式不同的部分是需要在「SUBROUTINE」前面加入「ATTRIBUTES(GLOBAL)」指令，才能使 CUDA 副程式供 GPU 運算，如圖 B-3.2。

GPU[Device 端(kernel)]

```
MODULE test_GPU_m
CONTAINS
  ATTRIBUTES(GLOBAL) SUBROUTINE test_GPU(numout_d,conin)
                                                          → 1.利用MODULE將整個kernel
                                                            包裝成GPU可運行得格式
  :

  END SUBROUTINE test_GPU
END MODULE test_GPU_m
```

圖 B-3.2 在圖 B-3.1 中利用 MODULE 包裝 Device 端的程式碼。

GPU[Host 端]

```
PROGRAM test
  USE CUDAFOR
  USE test_GPU_m  → 使用mod
  IMPLICIT NONE

  :

END PROGRAM test
```

圖 B-3.3 在圖 B-3.1 中 Host 端使用 Device 端創造的 mod。

二、宣告方式

3.4.1 小節介紹過 Host 端的宣告方式，接下來要介紹在 Device 端的宣告方式。Device 端要宣告接收 Host 端的參數 (如圖 B-3.4 的 Device 端變數「numout_d(4)」與常數「conin」)，表 B-3.1 是 CUDA 在 Device 端宣告方式：

表 B-3.1 CUDA 在 Device 端宣告方式。

	變數	常數
整數	INTEGER	INTEGER,VALUE
浮點數(單精準度)	REAL	REAL,VALUE
浮點數(雙精準度)	DOUBLE PRECISION	DOUBLE PRECISION,VALUE
複數(單精準度)	COMPLEX	COMPLEX,VALUE
複數(雙精準度)	DOUBLE COMPLEX	DOUBLE COMPLEX,VALUE
邏輯	LOGICAL	LOGICAL,VALUE
字串	CHARACTER *1	CHARACTER *1

字串宣告在 Device 端的常數宣告方式與其他常數宣告方式不同。

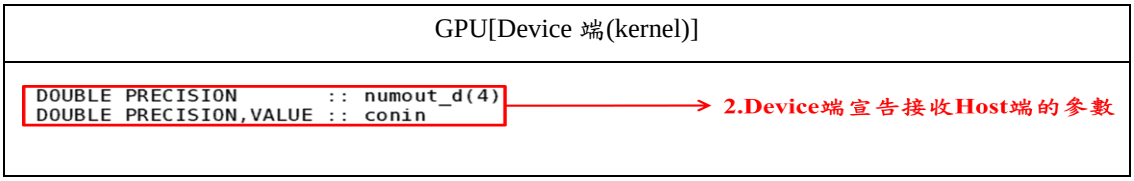


圖 B-3.4 在圖 B-3.1 中 CUDA 在 Device 端的宣告方式。

三、CUDA 編號

由 2.3.1 小節得知，CUDA 是由 thread 在做平行運算，因此想命令這些 thread 做運算的話，必須要將他們編號；在 grid 內區分很多 block，所以 block 也必須要一併編號。表 B-3.2 為 CUDA 新增的每個指令所代表的意思：

表 B-3.2 CUDA 自動編號指令表。

CUDA 編號指令	解釋
GRIDDIM%X	在 x 方向 grid 內 block 總個數
GRIDDIM%Y	在 y 方向 grid 內 block 總個數
GRIDDIM%Z	在 z 方向 grid 內 block 總個數
BLOCKDIM%X	在 x 方向每個 block 內 thread 總個數
BLOCKDIM%Y	在 y 方向每個 block 內 thread 總個數
BLOCKDIM%Z	在 z 方向每個 block 內 thread 總個數
BLOCKIDX%X	在 x 方向 block 索引(block 的 ID)
BLOCKIDX%Y	在 y 方向 block 索引(block 的 ID)
BLOCKIDX%Z	在 z 方向 block 索引(block 的 ID)
THREADIDX%X	在 x 方向 thread 索引(thread 的 ID)
THREADIDX%Y	在 y 方向 thread 索引(thread 的 ID)
THREADIDX%Z	在 z 方向 thread 索引(thread 的 ID)

呼叫 CUDA 副程式時，block 與 thread 會自動編號，如圖 B-3.5。

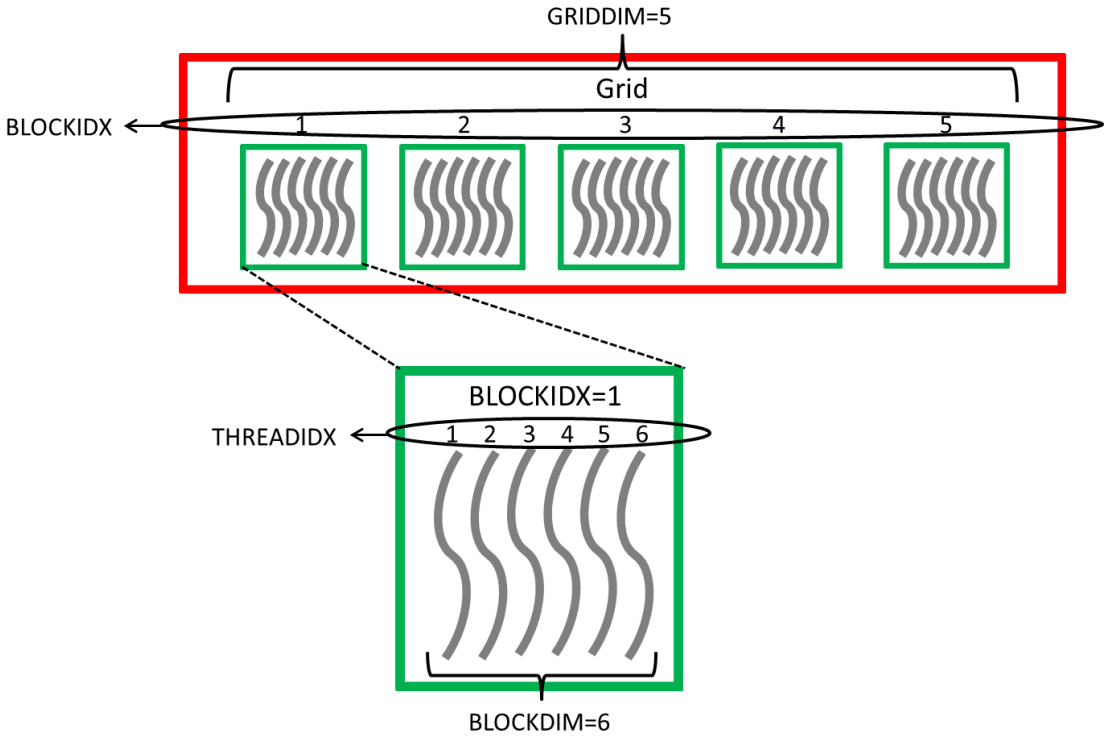


圖 B-3.5 此圖說明 CUDA 編號指令範例。我們選擇 Grid 內有 5 個 block，每個 block 有 6 個 thread，當呼叫 CUDA 副程式時，block「BLOCKIDX」會自動編號 1~5，thread「THREADIDX」會自動編號 1~6；grid 維度「GRIDDIM」=5，block 維度「BLOCKDIM」=6。

我們可自定編號將 Grid 內所有 thread 編成不同的號碼，這樣可以更明確決定每個 thread 要做的計算，如圖 B-3.6。

GPU[Device 端(kernel)]

```
i=(BLOCKIDX%X-1)*BLOCKDIM%X+THREADIDX%X
```

→ 3.將block與thread編號

圖 B-3.6 在圖 B-3.1 中自定的廣義編號：三維的廣義編號方式只是多加 2 個變數來編號，例如多加一個維度可多加一行「j=(BLOCKIDX%Y-1)*BLOCKDIM%Y+THREADIDX%Y」。

以圖 B-3.6 範例比較 CUDA 編號指令定義的編號與我們自定的廣義編號的差別，如表 B-3.3：

表 B-3.3 CUDA 編號與自定編號差異。

CUDA 編號指令定義的編號		自定的廣義編號
BLOCKIDX	THREADIDX	i=(BLOCKIDX-1)* BLOCKDIM+ THREADIDX
1	1	(1-1)*6+1=1
	2	(1-1)*6+2=2
	3	(1-1)*6+3=3
	4	(1-1)*6+4=4
	5	(1-1)*6+5=5
	6	(1-1)*6+6=6
2	1	(2-1)*6+1=7
	2	(2-1)*6+2=8
	3	(2-1)*6+3=9
	4	(2-1)*6+4=10
	5	(2-1)*6+5=11
	6	(2-1)*6+6=12
3	1	(3-1)*6+1=13
	2	(3-1)*6+2=14
	3	(3-1)*6+3=15
	4	(3-1)*6+4=16
	5	(3-1)*6+5=17
	6	(3-1)*6+6=18
4	1	(4-1)*6+1=19
	2	(4-1)*6+2=20
	3	(4-1)*6+3=21
	4	(4-1)*6+4=22
	5	(4-1)*6+5=23
	6	(4-1)*6+6=24
5	1	(5-1)*6+1=25
	2	(5-1)*6+2=26
	3	(5-1)*6+3=27
	4	(5-1)*6+4=28
	5	(5-1)*6+5=29
	6	(5-1)*6+6=30

表 B-3.3 可知自定的廣義編號會比 CUDA 編號指令定義的編號還來的明確。

四、計算想法

由「三、CUDA 編號」可知「i」為我們自定的編號，圖 B-3.7 是利用自定編號命令 4 個 thread 執行計算，其他 thread 閒置，如圖 B-3.8 示意圖。

GPU[Host 端]

```
IF(i==1)THEN
numout_d(i)=2D0
END IF

IF(i==2)THEN
numout_d(i)=3D0
END IF

IF(i==3)THEN
numout_d(i)=4D0
END IF

IF(i==4)THEN
numout_d(i)=5D0
END IF
```

→ 4.平行運算

圖 B-3.7 在圖 B-3.1 中 Device 端命令 4 個 thread 做運算。

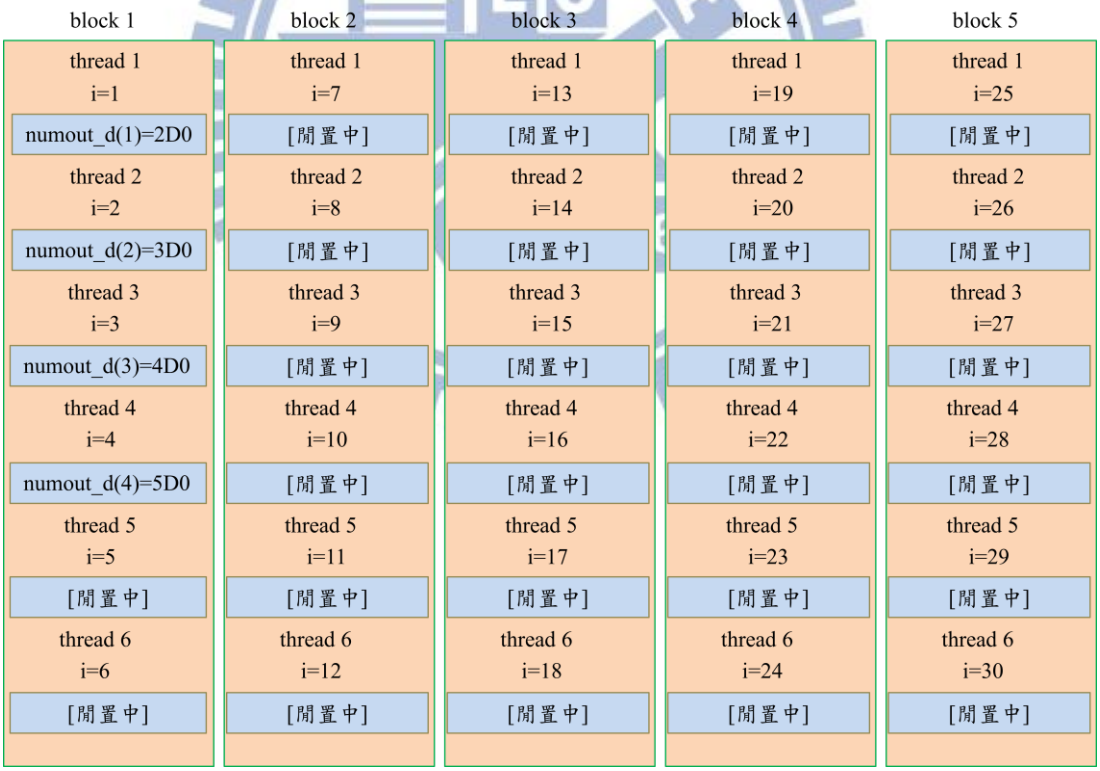


圖 B-3.8 在圖 B-3.6 中自定編號對應到圖 3.4.2.7 的平行運算。

由圖 B-3.7 看出可利用 IF 指令命令哪些 thread 做哪些事，在附錄 C 要討論如何使用 DO 指令將程式做平行運算。

附錄 C、使用平行運算來計算簡單迴圈的範例

要了解此部分寫法請先參考附錄 B 與 2.3.1 小節，此附錄探討迴圈平行化相對於附錄 B 又新增了哪些觀念。我們以一個簡單例子來做探討，計算式子如下：

$$SUM = \sum_{i=1}^{n_x} 1 \qquad \text{(C.1)}$$

，圖 C.1 為使用 CPU 與 GPU 程式流程圖。

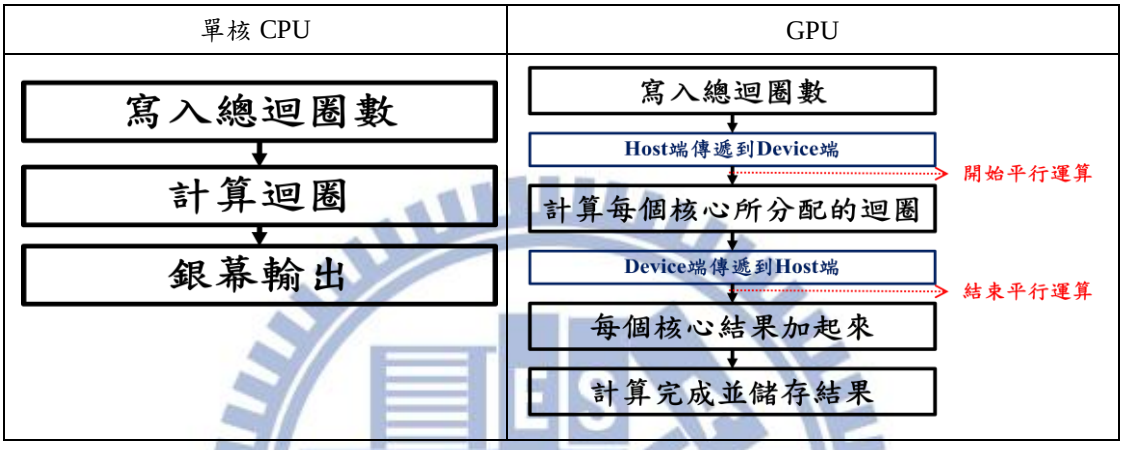


圖 C.1 使用單核 CPU 與 GPU 的程式流程圖。

原本使用單核 CPU 執行程式，而使用 GPU 執行程式就必須使用平行概念，2.3.1 小節有以附錄 C 說明 CUDA 內部架構，設定 block 與 thread 個數達到核心的分工計算，如圖 C.2。

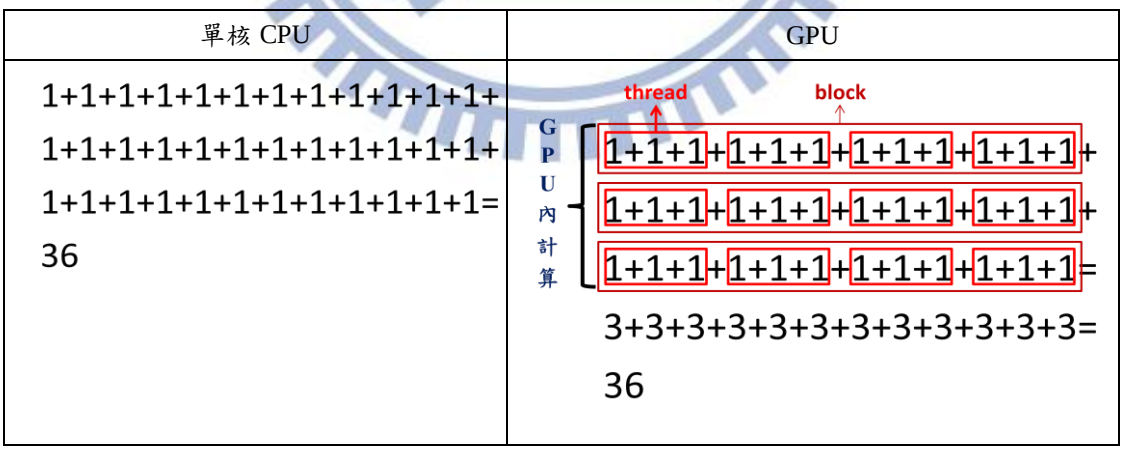


圖 C.2 使用 CPU 與 GPU 的計算方式：迴圈數(n_x)為 36、每個 grid 內 block 個數(griddx)為 3、每個 block 內 thread 個數(tPBdx)為 4。

由圖 C.2 可知我們利用 GPU 的核心將程式平分成 12 等份，讓每個核心分別做運算，最後再將所有核心計算結果傳回 Host 端且利用 CPU 將每個核心計算的結果加起來，此部分迴圈數極少且計算速度非常快，圖 C.3 為使用單核 CPU 與

GPU 計算所撰寫出的程式碼。

單核 CPU

```
  PROGRAM one_loop
    IMPLICIT NONE

    INTEGER,PARAMETER :: nx=2*10**5
    DOUBLE PRECISION  :: sums
    INTEGER           :: time_begin,time_end

    sums=0.0D0

    CALL SYSTEM_CLOCK(time_begin)
    CALL loop_CPU(sums,nx)
    CALL SYSTEM_CLOCK(time_end)

    WRITE(*,*) 'SUM=',sums
    WRITE(*,*) 'cpu_time=',
&   (time_end-time_begin)*1D-6,'sec'

  END PROGRAM one_loop
```

GPU[Host 端]

```
PROGRAM one_loop
  USE CUDAFOR
  USE loop_m
  IMPLICIT NONE

  INTEGER,PARAMETER        :: nx=2*10**5
  INTEGER,PARAMETER        :: griddx=3,tPBdx=4
  DOUBLE PRECISION         :: a(griddx*tPBdx)
  DOUBLE PRECISION,DEVICE  :: a_d(griddx*tPBdx)
  DOUBLE PRECISION         :: sums
  INTEGER                  :: i
  INTEGER                  :: istat
  INTEGER                  :: time_begin,time_end

  a=0.0D0

  CALL SYSTEM_CLOCK(time_begin)
  a_d=a
  CALL loop_GPU<<<griddx,tPBdx>>>(a_d,nx)
  a=a_d
  CALL SYSTEM_CLOCK(time_end)

  istat=CUDAFREE(a_d)

  sums=0.0D0

  DO i=1,griddx*tPBdx
  sums=sums+a(i)
  ENDDO

  WRITE(*,*) 'SUM=',sums
  WRITE(*,*) 'gpu_time=', &
  (time_end-time_begin)*1D-6,'sec'

END PROGRAM one_loop
```

1.宣告陣列大小=(thread總個數)的陣列

3.所有thread計算數值總加起來

單核 CPU

```
SUBROUTINE loop_CPU(sums,nx)
  IMPLICIT NONE

  INTEGER          :: nx
  DOUBLE PRECISION :: sums
  INTEGER          :: l

  DO l=1,nx
  sums=sums+1.0D0
  ENDDO

END SUBROUTINE loop_CPU
```

GPU[Device 端(kernel)]

```
MODULE loop_m
CONTAINS
  ATTRIBUTES(GLOBAL) SUBROUTINE loop_GPU(a_d,nx)
    IMPLICIT NONE

    INTEGER,VALUE          :: nx
    DOUBLE PRECISION       :: a_d(:)
    INTEGER                :: i
    INTEGER                :: l

    i=(BLOCKIDX%X-1)*BLOCKDIM%X+THREADIDX%X

    DO l=i,nx,BLOCKDIM%X*GRIDDIM%X
    a_d(i)=a_d(i)+1.0D0
    ENDDO

  END SUBROUTINE loop_gpu
END MODULE loop_m
```

2.迴圈分成(thread總個數)等份

單核 CPU[執行結果]

```
[      @HPML350         ]$ pgf95 one_loop_CPU.f
[      @HPML350         ]$ ./a.out
 SUM=    200000.0000000000
 cpu_time=   1.2620000000000001E-003 sec
[      @HPML350         ]$
```

GPU[執行結果]

```
[      @HPML350         ]$ pgf95 one_loop_GPU.cuf
[      @HPML350         ]$ ./a.out
 SUM=    200000.0000000000
 gpu_time=   9.9700000000000014E-004 sec
[      @HPML350         ]$
```

圖 C.3 一般 Fortran 修改成 CUDA Fortran 迴圈範例：紅色框框表示相較圖 B-1.1 迴圈新增寫法。

由附錄 B 得知要如何宣告 CUDA Fortran 程式碼、Host 端與 Device 端如何互傳、如何呼叫 kernel、釋放 device 記憶體方法以及撰寫 Device 端的基本格式與想法，因此在此不作探討。表 C.1 為除了附錄 B 所修改的部分，對於迴圈 CUDA Fortran 寫法和想法又要修改哪些部分：

表 C.1 CUDA Fortran 與一般 Fortran 不同的部分。

寫法	用途(想法)
1. 宣告陣列大小=(thread 總個數) 的陣列	儲存所有 thread 計算出來的數值。
2. 迴圈分成(thread 總個數)等份	利用 GPU 平行運算加快計算時間。
3. 所有 thread 計算數值總加起來	計算的結果。

，而我們針對表 C.1 分成三個部份說明。

1. 宣告陣列大小=(thread 總個數)的陣列

由於平行運算是利用 thread 運算，每個 thread 計算出來的結果都必須儲存，因此需要宣告一個專門供 thread 計算數值結果存放的陣列，此陣列大小=(thread 總個數)，如圖 C.4。

GPU[Host 端]

```
DOUBLE PRECISION          :: a(griddx*tPBdx)
DOUBLE PRECISION,DEVICE :: a_d(griddx*tPBdx)
```

1.宣告陣列大小=
(thread總個數)的陣列

圖 C.4 在圖 C.3 中 CUDA 在 Host 端宣告專門儲存 thread 計算結果的陣列。

一般 Fortran 經常將陣列宣告寫成可變陣列，因此圖 C.5 說明如何在 Host 端宣告 device 記憶體的可變陣列及釋放可變陣列 device 記憶體方式。

單核 CPU

```
DOUBLE PRECISION        :: a
ALLOCATABLE             :: a(:)
                  :
ALLOCATE(a(griddx*tPBdx))
                  :
DEALLOCATE(a)
```

GPU[Host 端]

```
DOUBLE PRECISION,DEVICE :: a_d
ALLOCATABLE             :: a_d(:)
                  :
ALLOCATE(a_d(griddx*tPBdx))
                  :
DEALLOCATE(a_d)
```

圖 C.5 一般 Fortran 宣告可變陣列及釋放記憶體與在 Host 端宣告可變陣列及釋放 device 記憶體兩者比較：與一般 Fortran 比較只有在一開始宣告有差異其他都一樣，而可變矩陣釋放 device 記憶體不能使用之前 CUDAFREE 函數的方法，直接使用 DEALLOCATE 即可。

2. 迴圈分成(thread 總個數)等份

圖 C.6 表示將迴圈分成(thread 總個數)等份，迴圈採取跳躍式的疊加，其跳躍間距等於「BLOCKDIM*GRIDDIM」也就是(thread 總個數)；計算的部份使用在 Host 端宣告陣列大小=(thread 總個數)的陣列，利用此陣列元素儲存每個 thread 計算後的結果，這樣才不會產生資料碰撞。圖 C.7 為每個 thread 運行的狀況，使

用陣列「a_d」將每個 thread 計算結果儲存起來。

GPU[Device 端(kernel)]

```
DO l=i,nx,BLOCKDIM%X*GRIDDIM%X
a_d(i)=a_d(i)+1.0D0
ENDDO
```

2.迴圈分成(thread總個數)等份

圖 C.6 在圖 C.3 中將迴圈分成(thread 總個數)「BLOCKDIM*GRIDDIM」等份，利用每個 thread 做計算。

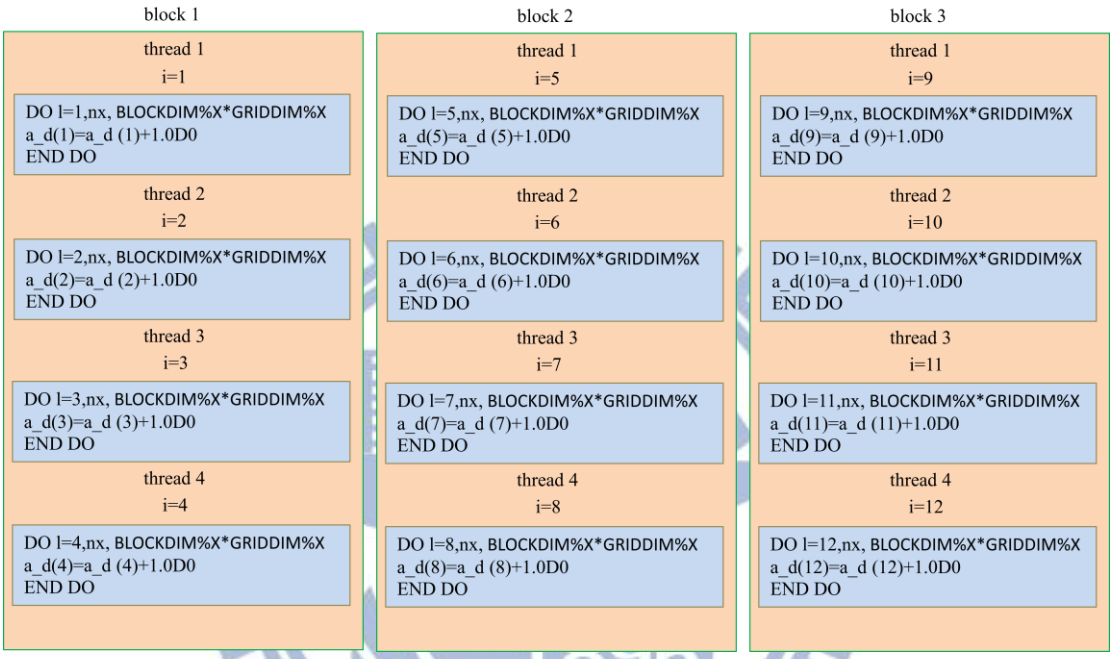


圖 C.7 每個 thread 運行的狀況：此程式的 grid 內 block 個數「griddx」=3，每個 block 內 thread 個數「tPBdx」=4，因此「BLOCKDIM*GRIDDIM」=12，每個 thread 迴圈跳躍間距為 12，這樣就不會重複計算到了。

3. 所有 thread 計算數值總加起來

GPU[Host 端]

```
sums=0.0D0

DO i=1,griddx*tPBdx
sums=sums+a(i)
ENDDO
```

3.所有thread計算數值總加起來

圖 C.8 在圖 C.3 中每個 thread 總加起來的結果。

在 GPU 內做平行計算，為了不發生資料碰撞，必須將每個 thread 計算出來的結果儲存起來，最後再傳回 Host 端利用如圖 C.8 方法做總加的步驟。

實際測試結果：

Fortran 本身就有一個參數能設定的數值極限，為了防止出錯，因此在此程式碼迴圈內又加入一個迴圈，如圖 C.9 程式碼所示。

單核 CPU

```
SUBROUTINE loop_CPU(sums,nx)
  IMPLICIT NONE

  INTEGER           :: nx
  DOUBLE PRECISION  :: sums
  INTEGER           :: l
  INTEGER           :: m

  DO l=1,nx
    DO m=1,1000
     sums=sums+1.0D0
    ENDDO
  ENDDO

END SUBROUTINE loop_CPU
```

防止出錯所加的內迴圈

GPU[Device 端(kernel)]

```
MODULE loop_m
CONTAINS
  ATTRIBUTES(GLOBAL) SUBROUTINE loop_GPU(a_d,nx)
    IMPLICIT NONE

    INTEGER,VALUE          :: nx
    DOUBLE PRECISION       :: a_d(:)
    INTEGER                :: i
    INTEGER                :: l
    INTEGER                :: m

    i=(BLOCKIDX%X-1)*BLOCKDIM%X+THREADIDX%X

    DO l=i,nx,BLOCKDIM%X*GRIDDIM%X
      DO m=1,1000
       a_d(i)=a_d(i)+1.0D0
      ENDDO
    ENDDO

  END SUBROUTINE loop_gpu
END MODULE loop_m
```

防止出錯所加的內迴圈

圖 C.9 在迴圈內加入一個迴圈，使它不會因為數值的極限而計算錯誤。

利用圖 C.9 修改過的程式碼，改變迴圈數比較 CPU(編輯器：Intel Fortran)與 GPU(編輯器：PGI Fortran)的數值結果、時間和倍率(CPU 時間/GPU 時間)，執行後的結果如圖 C.10 所示。(選取 grid 內 block 個數「griddx」=50，每個 block 內 thread 個數「tPBdx」=512。)

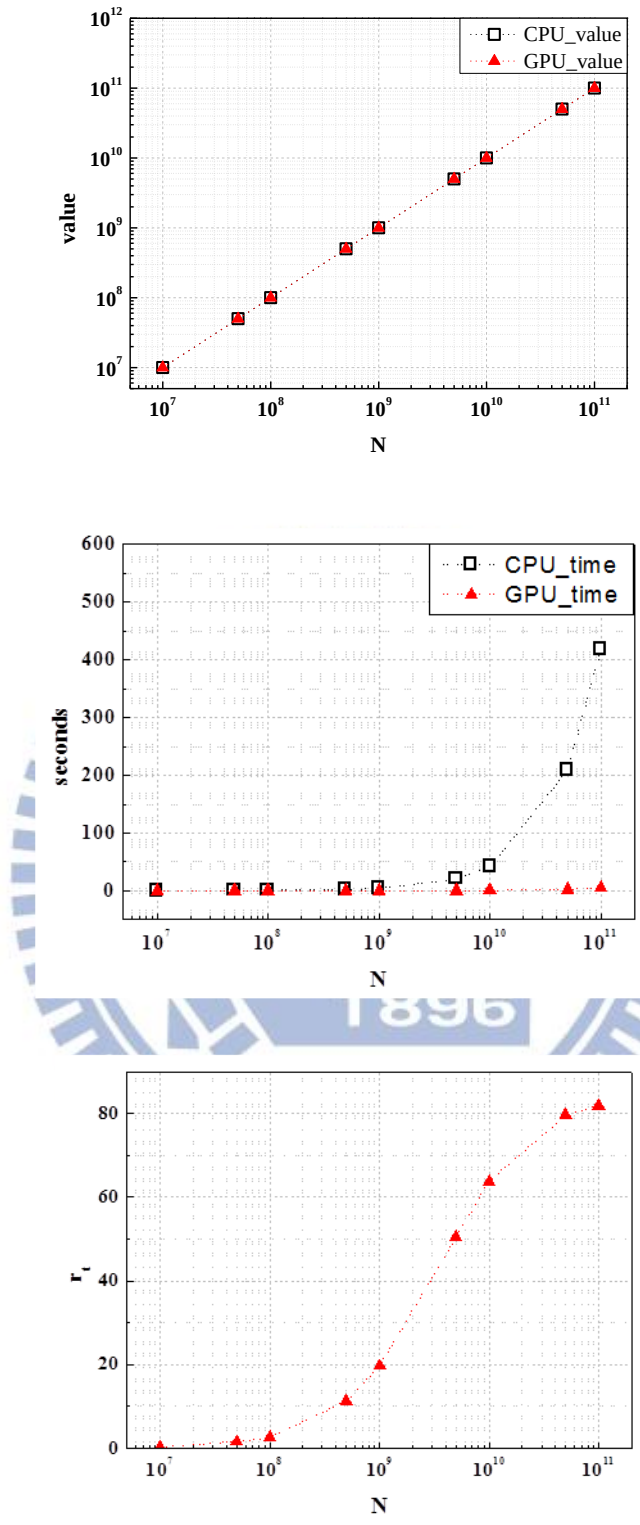
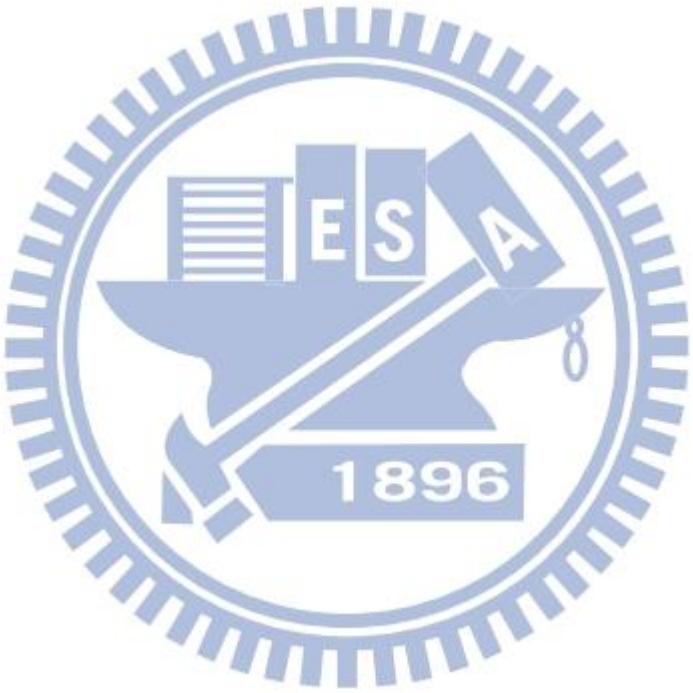


圖 C.10 比較單核 CPU 與 GPU 數值、時間、倍率關係：r_t 為增快倍率，N 為迴圈總個數。

由於結果都使用到 GPU 的優點(核心數多、Device 端內記憶體寬頻快)，因此增快效應會非常大。假如考慮 device 記憶體很大及 GPU 記憶體不足需分批計算(詳細 2.2.2 節)、register 不足需使用到 global memory (詳細 2.3.2 節)時，增快

效應就不會這麼大了，而庫侖交互作用程式需使用較多的 device 記憶體運算(每個核心都會消耗記憶體)，必須將程式耗時較久的部分分批計算，由於耗費 device 記憶體多，register 不足要使用到 global memory，因此增快效應沒有此範例結果大。



附錄 D、修改庫侖交互作用程式方法

對於庫侖交互作用的程式在撰寫程式時須注意的部分(附錄 C 程式不同的地方)有三個部分：

1. 需要較多的參數傳遞到 device 記憶體，因此需要花費較多的時間。
(相關內容在 2.2.2 小節有提到。)
2. 擁有較複雜的計算，需要較多 register 與使用 global memory 儲存暫存檔，不能將六重迴圈全部平行運算，因此要考慮讓 GPU 處於最大效益。
(相關內容在 2.3.2 小節與 2.4 小節有提到。)
3. 因為資料龐大，需要創造較多的 grid 來讓 GPU 計算，又為了不讓資料碰撞，每個 thread 的資訊必須儲存，在做整合時會消耗許多時間，因此必須利用 Tree Reduction 演算法使時間變快。
(此演算法需使用到 shared memory 與同步概念。)

接下來分成三項探討，分別是傳送方式、平行方式、Tree Reduction 演算法。

一、傳送方式

在附錄 B 有討論到傳送方式，而這裡主要探討的是考慮一個六重迴圈的程式，由於計算量太過龐大，因此需要分批利用 GPU 做計算。根據 2.2.2 小節探討的結果，盡量讓 GPU 內需要的參數一次傳遞完，圖 D.1 示意圖為一個六重迴圈的程式，假如寫程式時將傳遞寫在迴圈內會發生參數一直重複傳遞，會使程式變慢，因此這個部份需特別注意。

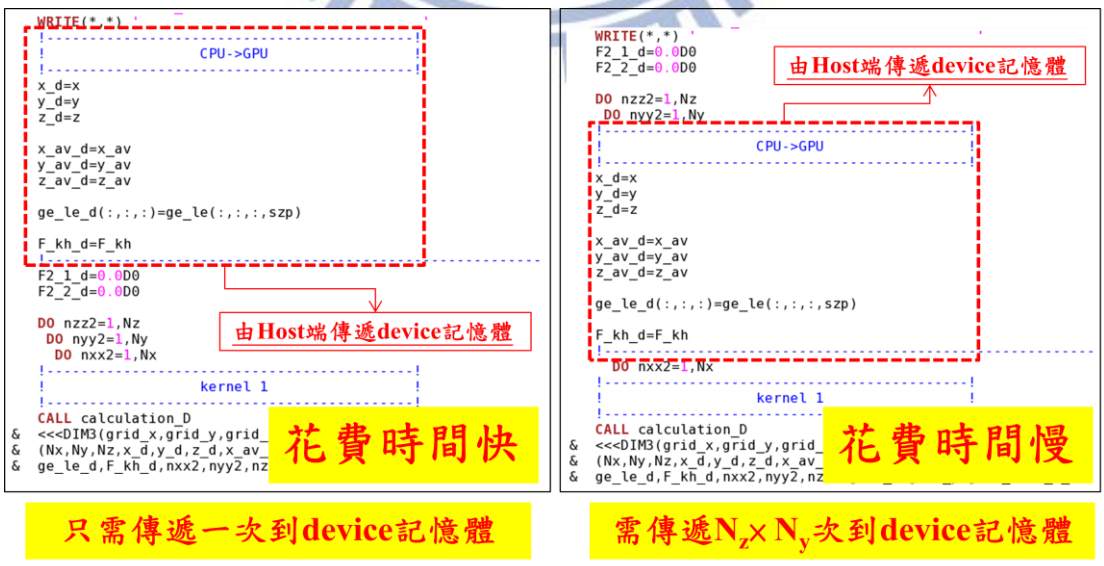


圖 D.1 左邊表示只傳遞一次，右邊表示傳遞 $N_z \times N_y$ 次。

利用電子－電洞間長程偶極－偶極交換能矩陣元素的程式當作例子，將傳遞次數變多會使程式變慢，表 D.1 為參數選擇與時間結果：

表 D.1 測試 Host 端傳遞到 Device 端次數的結果。

參數與時間	選取與結果
(l_x, l_y, l_z)[nm]	(3.0 , 2.7 , 2.0)
a_0[nm]	0.38
dr[nm]	0.38
$V^{ehex,LR(D)}$數值結果[μeV]	406.7
GPU 時間(Host 端傳遞 device 記憶體 1 次)	27 s
GPU 時間(Host 端傳遞 device 記憶體 1408 次)	45 s

傳遞越多次程式需要消耗的時間越多，因此盡量讓程式減少需要傳遞的次數。

二、平行方式

對於庫侖交互作用程式而言，是屬於計算量龐大的程式，因此可平行的迴圈數有限。在 2.4 小節有寫到平行運算是將 warp 傳到 SM 內做運算的，當有 warp 閒置或計算完成時，下一個 warp 填補到將要閒置的 SM 繼續做計算，所以對於大型運算來說，warp 越多計算速度越快，也就是 thread 越多越好，但 GPU 擁有記憶體有限制，因此 thread 總個數也不能太多。目前測試結果將三個迴圈平行運算是最恰當的結果，register 與 global memory 的存取及釋放暫存檔剛好能使 GPU 處於最大效益。而 2.3.2 小節所下的結論是盡量不要在 global memory 存取暫存檔，但庫侖交互作用程式計算量龐大，必須使用到 global memory 才能做運算，減少使用 global memory 的方法就是減少需要存取的暫存檔，例如將「A=B+C+D」在 Host 端先計算完「E=C+D」後，再利用 GPU 計算 Device 端的「A=B+E」，這樣可以減少存取的暫存檔，但壞處就是會破壞原有程式碼的可讀性，因此目前不採用此法。

三、Tree Reduction 演算法[12]

此部分必須先介紹 shared memory 的宣告與同步的概念。由附錄 B 可知宣告

分為 Host 端與 Device 端的宣告，而 shared memory 的宣告只適用於 Device 端的宣告。Device 端宣告分為變數與常數，shared memory 的宣告是屬於變數的宣告，宣告方式如表 D.2：

表 D.2 shared memory 在 Device 端宣告方式。

	變數
整數	INTEGER,SHARED
浮點數(單精準度)	REAL,SHARED
浮點數(雙精準度)	DOUBLE PRECISION,SHARED
複數(單精準度)	COMPLEX,SHARED
複數(雙精準度)	DOUBLE COMPLEX,SHARED

在 2.3.2 小節有提到 shared memory 有容量上的限制，因此宣告陣列的陣列大小也有限制，以 GPU 型號為「TESLA C2050」測試的結果陣列大小最大能設定 512。

接下來要介紹同步的概念，顧名思義就是讓每個 thread 同時進行與結束，而同步只適用於 Device 端，以下為同步的指令：

CALL SYNCTHREADS()

圖 D.2 中假設 thread1.2.3 同時開始進行程式，在程式碼加入同步指令後，當 thread1.2.3 計算到加入的那一行時，thread1.2.3 會先閒置，此時先將結果 A、B、C 存在 shared memory 裡以備接下來的計算會使用到，等結果 A、B、C 都計算完後 thread1.2.3 才會繼續執行程式。

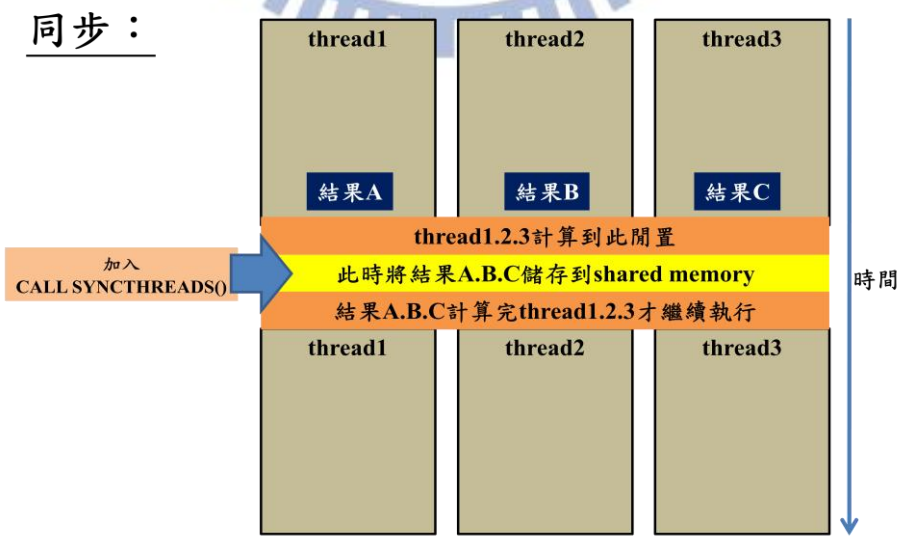


圖 D.2 程式同步的過程：2.3.2 小節可知 shared memory 用於同一個 block 內 thread 之間的溝通。

接下來要探討 Tree Reduction 演算法，此演算法只能再 Device 端進行，由於需使用到 thread 的溝通，因此將會考慮 shared memory 的使用及同步的概念。由「二、平行方式」得知程式中需產生大量的 grid，表示每個 thread 計算的結果都需要由 Device 端輸出，在做整合時也會耗費許多時間，因此利用 GPU 計算快速的優點，運用 Tree Reduction 演算法來做運算，如圖 D.3 所示，每個圓圈表示每個 thread 計算出來的結果，做整合時利用 shared memory 在同一個 block 內 thread 之間的溝通的特性與同步，再次使用 GPU 運算，每次計算都會讓需要整合的資料減少兩倍，計算效率會因此大大提升。

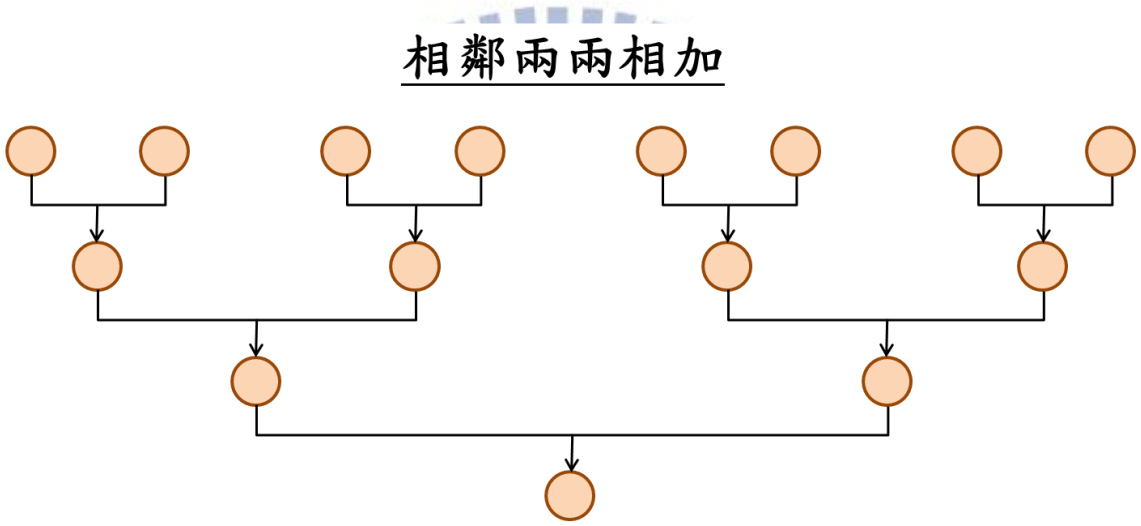


圖 D.3 Tree Reduction 演算法。

附錄 E、辛普森法(補充)

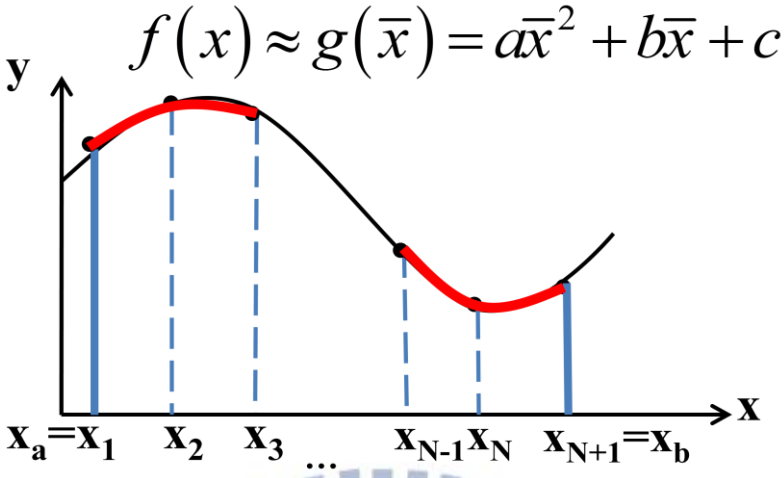


圖 E.1 將函數 $f(x)$、利用拋物線線求定積分示意圖。

如圖 E.3 所示，考慮區間 $x_1 \le x \le x_2$ 的函數 $f(x)$，利用拋物線做近似並且令 $\bar{x} = x - x_2$ 可將此區間寫成

$$f(x) \approx g(\bar{x}) = a\bar{x}^2 + b\bar{x} + c \text{，} \quad x_1 \le x \le x_3 \tag{E.1}$$

將(E.1)式代入定積分 $\int_{x_a}^{x_b} f(x)dx$ 可得

$$\int_{x_1}^{x_3} f(x)dx \approx \frac{2}{3}a\Delta x^3 + 2c\Delta x \tag{E.2}$$

其中 $\Delta x = x_3 - x_2 = x_2 - x_1$，利用以下聯立方程式求 a

$$g(\Delta x) = a\Delta x^2 + b\Delta x + c \tag{E.3}$$

$$g(0) = c \tag{E.4}$$

$$g(-\Delta x) = a\Delta x^2 - b\Delta x + c \tag{E.5}$$

可求得

$$a = \frac{g(\Delta x) - 2g(0) + g(-\Delta x)}{2\Delta x^2} \tag{E.6}$$

因此(E.2)式可整理成

$$\int_{x_1}^{x_3} f(x)dx = \frac{\Delta x}{3}\left[f(x_3) + 4f(x_2) + f(x_1)\right] \tag{E.7}$$

假如考慮區間 $x_a \le x \le x_b$ 可將積分式子寫成

$$\int_{x_a}^{x_b} f(x)dx = \int_{x_a}^{x_3} f(x)dx + \int_{x_3}^{x_5} f(x)dx + \cdots + \int_{x_{N-1}}^{x_b} f(x)dx \tag{E.8}$$

將(E.7)式代入(E.8)式整理後可得

$$\int_{x_a}^{x_b} f(x)dx = \frac{\Delta x}{3}\sum_{i\in odd}^{N-1}\left[f(x_i) + 4f(x_{i+1}) + f(x_{i+2})\right] \tag{E.9}$$

將(E.9)式推廣到三維的狀況，則數學形式可表示

$$\begin{aligned}\int_{x_a}^{x_b}\int_{y_a}^{y_b}\int_{z_a}^{z_b} f(x,y,z)\,dxdydz \approx \\ \approx \frac{\Delta x\Delta y\Delta z}{27}\sum_{i\in odd}^{N_x-1}\sum_{j\in odd}^{N_y-1}\sum_{k\in odd}^{N_z-1}\left\{\begin{array}{l} f(x_i,y_n,z_k) + 4f(x_m,y_n,z_{k+1}) + \\ f(x_i,y_n,z_{k+2}) + 4f(x_m,y_{n+1},z_k) + \\ 16f(x_i,y_{n+1},z_{k+1}) + 4f(x_m,y_{n+1},z_{k+2}) + \\ f(x_i,y_{n+2},z_k) + 4f(x_m,y_{n+2},z_{k+1}) + \\ f(x_i,y_{n+2},z_{k+2}) + 4f(x_{m+1},y_n,z_k) + \\ 16f(x_{i+1},y_n,z_{k+1}) + 4f(x_{m+1},y_n,z_{k+2}) + \\ 16f(x_{i+1},y_{n+1},z_k) + 64f(x_{m+1},y_{n+1},z_{k+1}) + \\ 16f(x_{i+1},y_{j+1},z_{k+2}) + 4f(x_{i+1},y_{j+2},z_k) + \\ 16f(x_{i+1},y_{j+2},z_{k+1}) + 4f(x_{i+1},y_{j+2},z_{k+2}) + \\ f(x_{i+2},y_j,z_k) + 4f(x_{i+2},y_j,z_{k+1}) + \\ f(x_{i+2},y_j,z_{k+2}) + 4f(x_{i+2},y_{j+1},z_k) + \\ 16f(x_{i+2},y_{j+1},z_{k+1}) + 4f(x_{i+2},y_{j+1},z_{k+2}) + \\ f(x_{i+2},y_{j+2},z_k) + 4f(x_{i+2},y_{j+2},z_{k+1}) + \\ f(x_{i+2},y_{j+2},z_{k+2}) \end{array}\right\}\end{aligned} \tag{E.10}$$

附錄 F、GPU 相關測試

提供電腦的硬體配備如表 F.1：

表 F.1 執行 bandwidthTest.cuf 程式的電腦設備。

	主機板	CPU	GPU
型號	HP ML350 G6	IntelXeon E5520 2.26Ghz	Tesla C2050

一、傳送次數測試(2.2.2 小節)

以傳送的陣列元素個數相同為例子，分別測試 CPU↔CPU、GPU↔GPU、CPU↔GPU 所花費的時間。

1. CPU↔CPU

```
Subroutine transmit_file()
IMPLICIT NONE

INTEGER :: i,j
DOUBLE PRECISION :: x(100,1000000)
DOUBLE PRECISION :: y(100,1000000)
INTEGER :: istat

x=1.0D0              !!input parameters

DO i=1,1
y=x                  !! CPU->CPU
x=y                  !! CPU->CPU
END DO

RETURN
END Subroutine
```

```
[    @HPML350 GPU_test]# make
pgf95 GPU_test.f -fastsse -Mcuda -o GPU_test
[    @HPML350 GPU_test]# time ./GPU_test
CODE END

real    0m1.041s
user    0m0.596s
sys     0m0.445s
```

圖 F.1 (上)表示 CPU↔CPU 傳送 1 次的程式碼與(下)實際時間。

2. GPU↔GPU

```
Subroutine transmit_file()
USE CUDAFOR
IMPLICIT NONE

INTEGER :: i,j
DOUBLE PRECISION,DEVICE :: x_d(100,1000000)
DOUBLE PRECISION,DEVICE :: y_d(100,1000000)
INTEGER :: istat

x_d=1.0D0              !!input parameters

DO i=1,1
y_d=x_d  |         !! GPU->GPU
x_d=y_d           !! GPU->GPU
END DO

istat=CUDAFREE(x_d) !! free memory
istat=CUDAFREE(y_d) !! free memory

RETURN
END Subroutine
```

```
[    @HPML350 GPU_test]# make
pgf95 GPU_test.f -fastsse -Mcuda -o GPU_test
[    @HPML350 GPU_test]# time ./GPU_test
Warning: ieee_inexact is signaling
CODE END

real    0m0.371s
user    0m0.054s
sys     0m0.315s
```

圖 F.2 (上)表示 GPU↔GPU 傳送 1 次的程式碼與(下)實際時間。

3. CPU↔GPU

當傳送次數越多時，花費時間越多，因此比較傳送次數 1 次與 10^6 次花費時間如圖 F.3。

```
Subroutine transmit_file()
USE CUDAFOR
IMPLICIT NONE

INTEGER :: i,j
DOUBLE PRECISION :: x(100,1000000)
DOUBLE PRECISION,DEVICE :: x_d(100,1000000)
INTEGER :: istat

x=1.0D0              !!input parameters

DO i=1,1
x_d=x                !! CPU->GPU
x=x_d                !! GPU->CPU
END DO

istat=CUDAFREE(x_d) !! free memory

RETURN
END Subroutine
```

傳送1次

```
Subroutine transmit_file()
USE CUDAFOR
IMPLICIT NONE

INTEGER :: i,j
DOUBLE PRECISION :: x(100,1000000)
DOUBLE PRECISION,DEVICE :: x_d(100)
INTEGER :: istat

x=1.0D0              !!input parameters

DO i=1,1000000
x_d=x(:,i)           !! CPU->GPU
x(:,i)=x_d           !! GPU->CPU
END DO

istat=CUDAFREE(x_d) !! free memory

RETURN
END Subroutine
```

傳送10^6次

```
[    @HPML350 GPU_test]# make
pgf95 GPU_test.f -fastsse -Mcuda -o GPU_test
[    @HPML350 GPU_test]# time ./GPU_test
Warning: ieee_inexact is signaling
CODE END

real    0m1.624s
user    0m1.131s
sys     0m0.491s
```

```
[    @HPML350 GPU_test]# make
pgf95 GPU_test.f -fastsse -Mcuda -o GPU_test
[    @HPML350 GPU_test]# time ./GPU_test
Warning: ieee_inexact is signaling
CODE END

real    0m36.649s
user    0m20.069s
sys     0m16.571s
```

圖 F.3 (左上)表示 CPU↔GPU 傳送 1 次的程式碼與(左下)實際時間，(右上)表示 CPU↔GPU 傳送 10^6 次的程式碼與(右下)實際時間。

二、記憶體測試(2.3.2 小節)

在我們編寫程式時，會碰到資料碰撞的問題，而如何解決此問題將成為使用記憶體多寡的關鍵。首先先介紹資料碰撞的意思：在使用 GPU 運算時，Host 端記憶體會先傳遞到 Device 端記憶體再利用 GPU 內的核心運算，當運算完後將每個核心運算後的結果傳到 Host 端記憶體時，假設程式設定不當會發生所有核心

都要傳到同一個資料裡，此時資料會重複紀錄而使運算結果出錯，而傳送錯誤的過程稱為資料碰撞，圖 F.4 為示意圖。

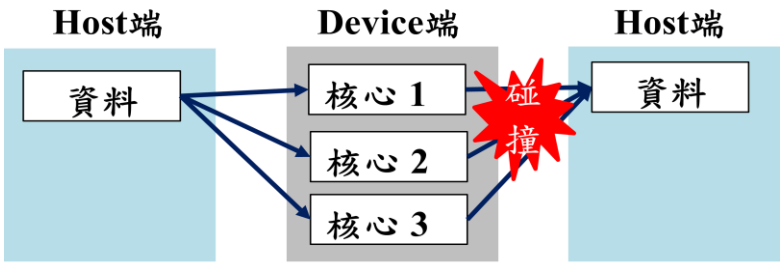


圖 F.4 資料碰撞示意圖。

解決資料碰撞問題有兩種方法：一種是由 Host 端先定義與核心數等量的資料，利用 Device 端的核心將結果計算出來直接存回此資料內；另一種是將每個核心的計算結果存於由 Device 端定義與核心數等量的資料，再將資料全部傳回 Host 端，圖 F.5 為示意圖。

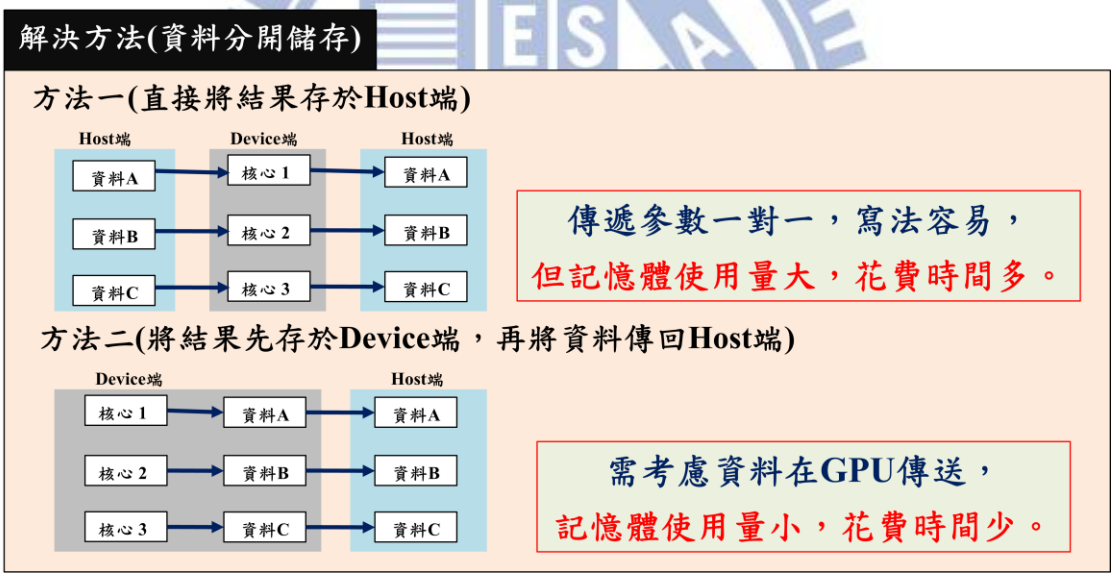


圖 F.5 解決資料碰撞示意圖。

前者雖然寫法簡單，但會遇到記憶體的使用量過多的情形，相較於後者記憶體使用量比前者少很多，而當暫存檔的存取使用到全域記憶體時，會花費許多時間，使增快的效益變差。因此接下來將探討記憶體使用量與時間的比較，測試式子如下

$$Sol = \sum_{i=1}^{80}\sum_{j=1}^{80}\sum_{k=1}^{80}\sum_{l=1}^{80}\sum_{m=1}^{80}\sum_{n=1}^{80} 10^{-3} \tag{F.1}$$

其結果如圖 F.6 所示。

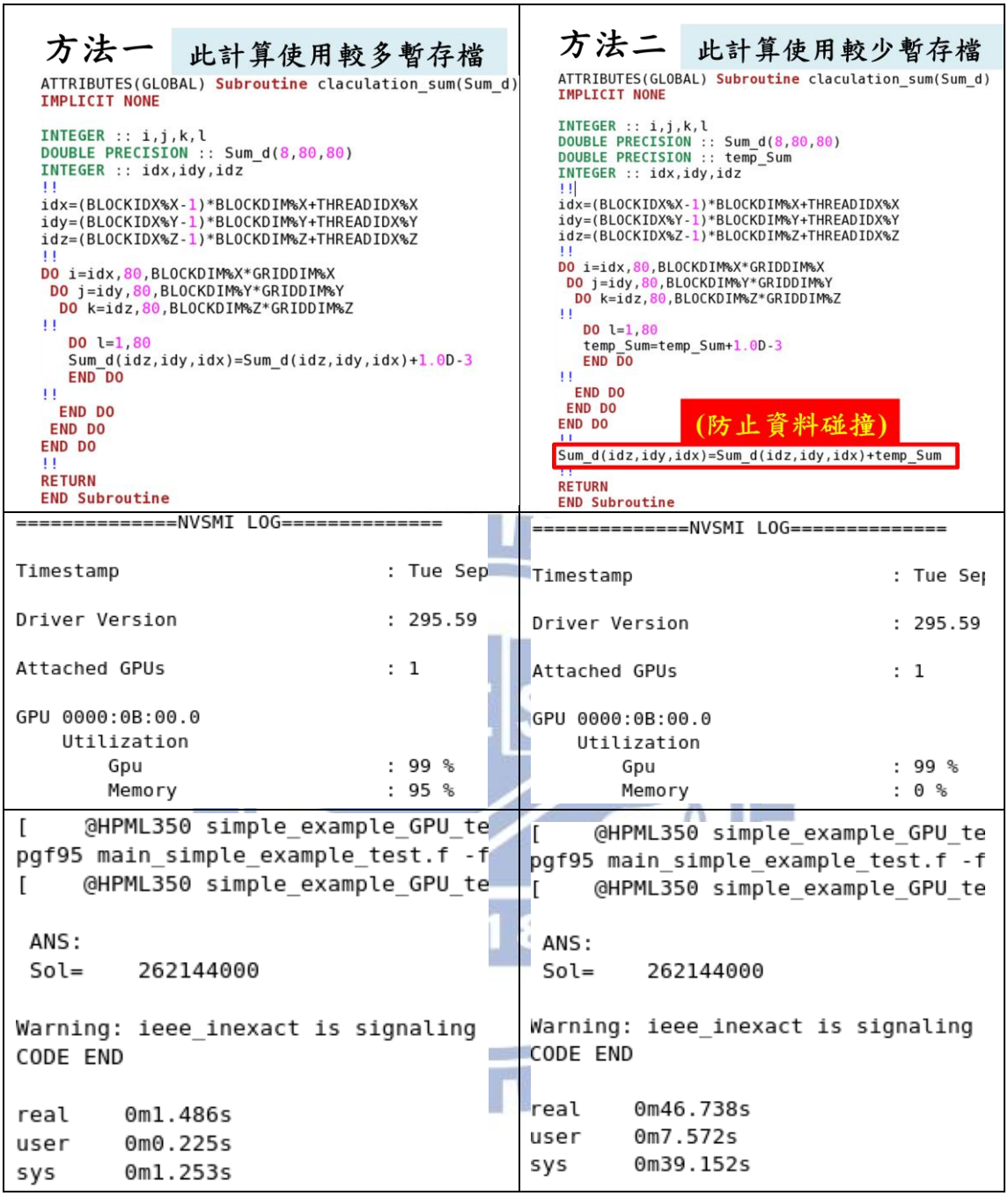


圖 F.6 (上)表示 GPU 內運算的程式碼與(中)計算記憶體使用量與(下)結果和實際時間。

附錄 G、閃鋅結構威格納－塞茲晶胞的形狀

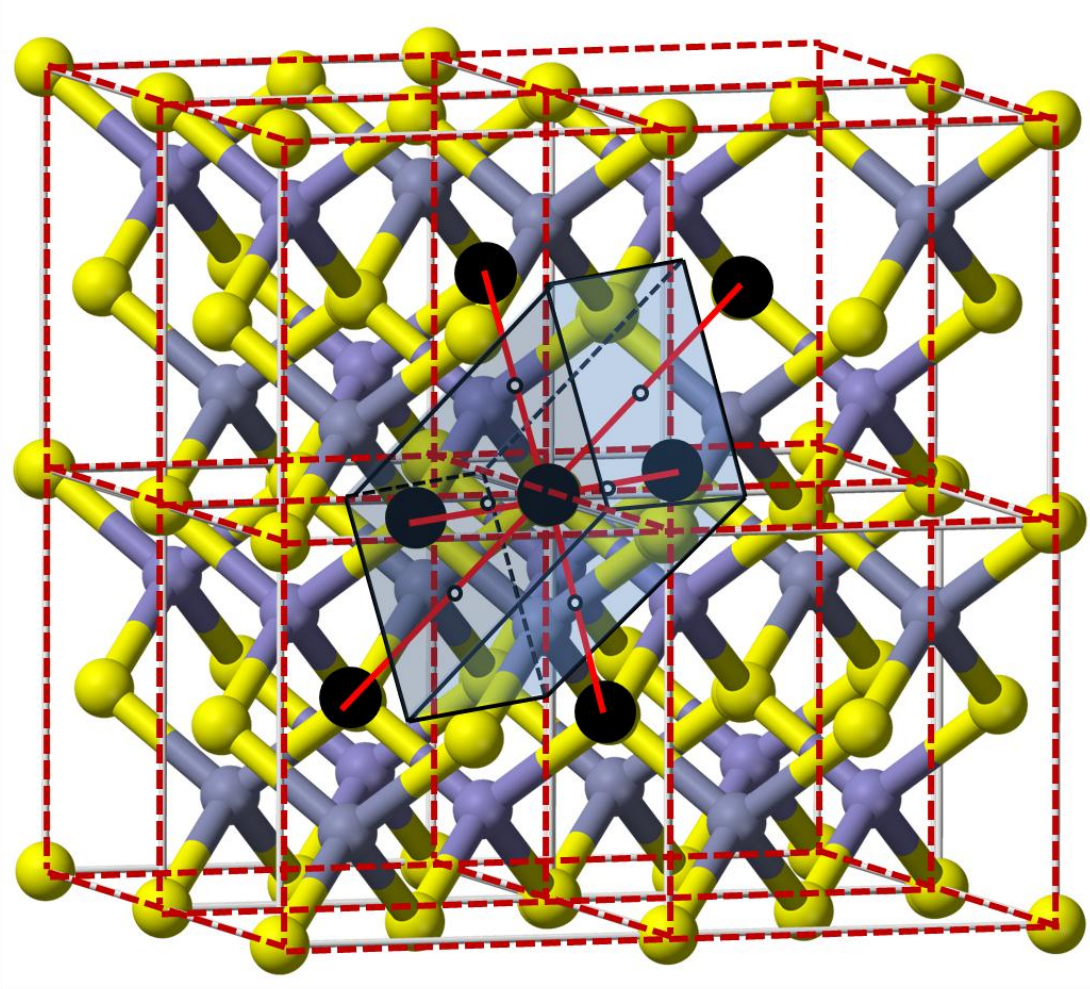


圖 G.1 閃鋅結構威格納－塞茲晶胞選取方式：形狀為傾斜的平行六面體。

參考文獻

[1] 廖建智，無應力 GaAs/AlGaAs 量子點的光學異相性，碩士論文，交通大學電子物理研究所 (2012)。

[2] 陳柏中，*半導體量子點與量子資訊*，物理雙月刊(卅卷五期)，2008 年十月。

[3] Hanz Y. Ramirez, Chia-Hsien Lin, Wen-Ting You, Shan-Yu Huang, Wen-Hao Chang, Sheng-Di Lin and Shun-Jen Cheng. *Size and Ga-diffusion effects on optical fine structure splitting of* $In_{1-x}Ga_xAs/GaAs$ *self-assembled quantum dots*. [unpublished]

[4] 蘇正耀，*量子計算淺談*，物理雙月刊(廿五卷四期)，2003 年 8 月。

[5] 東南大學楊春山 revised by 同濟大學程微宏，*GPU 通用計算調研報告*。

[6] Intel® Core™，http://www.intel.com.tw/content/www/tw/zh/processors/core/core-i5-processor.html?cid=sem103p7301。

[7] *NVIDIA* TESLA，http://www.nvidia.com.tw/object/tesla-supercomputing-solutions-tw.html。

[8] iXBT Labs-NVDIA CUDA，http://ixbtlabs.com/articles3/video/cuda-1-p1.html。

[9] *NVIDIA CUDA Programming Guide*。

[10] *張舒，GPU 高性能運算之 CUDA*。

[11] 電子工程專輯，*Portland 發佈 PGI 2011 高性能運算編譯器與開發工具*，http://www.eettaiwan.com/ART_8800637922_676964_NP_91f5dfe2.HTM。

[12] *碩研 CUDA Fortran 程式設計課程講義*。

[13] 古智豪，*利用有限差分法計算半導體量子點電子結構*，碩士論文，交通大學電子物理研究所 (2011)。

[14] Eugene Kadantsev and Pawel Hawrylak, Phys. Rev. B**81**, 045311(2010).

[15] H.Y.Ramirez's Note:*e-h Exchange*:General Formulation.

[16] Y.T. Chen, C.C. Chao, S.Y. Huang, C.S. Tang, S.J. Cheng*, *"Singlet–triplet transitions in highly correlated nanowire quantum dots"* , Physica E: Low-dimensional Systems and Nanostructures, 42, 837 (2010).

[17] H. Y. Ramirez, C. H. Lin, C. C. Chao, Y. Hsu, W. T. You, S. Y. Huang, Y. T. Chen, H. C. Tseng, W. H. Chang, S. D. Lin and S. J. Cheng*, "Optical fine structures of highly quantized InGaAs/GaAs self-assembled quantum dots", Phys. Rev. B 81, 245324 (2010).

[18] kittel,"Introduction to Solid State Physics 8/E"JOHN WILEY&Sons

[19] 趙虔震，*自組式量子點中單一激子的庫侖交互作用*，碩士論文，交通大學電子物理研究所 (2010)。

[20] 廖禹淮，量子點光源中激子自旋動態,精細結構與光學偏振的研究，博士生

研究計畫構想書，交通大學電子物理研究所 (2012)。