東海大學

資訊工程研究所

碩士論文

指導教授:林祝興博士

快速傅立葉轉換在 CUDA GPU 的平行化加速運算之研究

On the Speedup of Parallel Fast Fourier Transform Using CUDA GPU

研究生: 楊博凱

中華民國 107 年 7 月

東海大學碩士學位論文考試審定書

東海大學資訊工程學系 研究所

研究生 楊 博 凱 所提之論文

快速傅立葉轉換在 CUDA GPU 的平行化加速運算之研究

經本委員會審查，符合碩士學位論文標準。

學位考試委員會

召 集 人 _____ 簽章

委 員 _____

_____

_____

_____

指 導 教 授 _____ 簽章

中華民國 107 年 7 月 6 日

摘 要

傅立葉變換是信號處理上對於時域與頻域互相轉換的機制，傅立葉變換的應用範圍跨越物理、數學、醫學、電信等領域，在生活中已是不可或缺的技術。快速傅立葉變換比一般的傅立葉變換有較低的時間複雜度，一般的快速傅立葉變換時間複雜度為 $O(N^2)$，而快速傅立葉變換的時間複雜度是 $O(N \cdot \log N)$。本實驗將快速傅立葉變換分別在一般電腦上與 GPU 中進行運算，比較快速傅立葉運算在 CPU 中未進行優化的運算時間與其在 GPU 上平行化的運算時間。本論文的實驗，我們使用 GPU 為 NVIDIA GTX 750 Ti，程式語言使用 CUDA 撰寫其 GPU 平行化的部分。實驗結果顯示，在資料量為 32678 的時候加速比最高只到達 48 倍；也就是 CPU 所花費的時間為 GPU 平行化花費時間的 48 倍。為了得到更佳的效率，我們將整體平行化運算進行記憶體配置最佳化，將ω事先計算出其數值並儲存在 GPU 中的唯讀記憶體，再將輸入資料放入共享記憶體中，將加速比提高到 114.7 倍。

關鍵字:快速傅立葉變換、GPU 平行化、GPU 最佳化、記憶體交錯與平行運算

Abstract

Applications of Fourier Transform are far-reaching, spanning fields such as physics, mathematics, medical science, and telecommunications; hence, its applications have become an indispensable part of our daily lives. This study compares performances of Fast Fourier Transforms on a host CPU, GPU parallel computing and GPU memory allocation optimizations. From the experimental results, GPU parallel computing is proven to be effective in enhancing computation speed of the FFT, the speedup rate is 48 times. In addition, by optimizing GPU memory allocation, the computation speed of the FFT was further enhanced with speedup rate 114.7.

Keywords: Fast Fourier Transform, GPU parallel computing, GPU optimization, memory interleaving and parallel processing

致謝

兩年的研究所生涯一晃眼就過了，這些日子以來有需多的人們需要感謝，感謝有你們的陪伴、幫忙與諒解。

首先要感謝在我大學專題就幫忙我需多，在研究所的指導老師 林祝興老師，願意讓我在研究所時常是那麼特殊的議題，真的非常感謝老師一路上的包容與幫助。

楊博凱 於東海大學研究所 資安實驗室 107 年 7 月

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF EQUATIONS

第一章　簡介

隨著時代的進步，現代的顯示晶片已經具備有高度的可程式化能力，由於顯示晶片通常有大量的執行單元，因此有許多用顯示晶片來幫助進行一些需要重複且大量計算的工作，即 GPGPU (General-Purpose computing on Graphics Processing Units) 。CUDA (Compute Unified Device Architecture) 即是 NVIDIA 所開發出 GPGPU 模型的一種整合技術 [1], [2]。

GPGPU 和 CPU 相比有以下幾個優點：

1. 有更大的記憶體。

2. 有更多的執行單元，雖然每個單元的運算效能並沒有 CPU 快，但更多的單元同時運算可發揮出比 CPU 更好的運算速度。

CUDA 是 NVIDIA 對於 GPGPU 的正式名稱，應用 CUDA 技術不只可在 GPU 上進圖形運算，它可以在 GPU 上執行速度較慢的大量執行緒 (threads) 的並行架構中進行平行化運算。CUDA 的出現大幅的減少了許多程式的運算時間，在某些計算上甚至可到達 100 倍左右的加速比 (speedup rate) 。在本論文裡我們使用 GPU 對於快速傅立葉變換進行平行化運算用以加速運算速度，再對其平行化進行記憶體的配置以達到最佳化的效果。

隨著資料量越來越大的各種應用環境，GPU 平行化運算也變得越來越重要，GPU 同時也帶動了 AI 的發展，在此同時如何讓 GPU 進行更加快速的平行化運算也成了一個值得研究的課題。傅立葉變換是許多領域中不可或缺的一項重要技術，在密碼學、信號處理、資料科學、物理學、醫學、商業等許多領域都擁有著極為廣泛的應用。在本文中，我們將快速傅立葉變換進行平行化運算加速其運算效率，並對其平行化過程中的記憶體做最佳化的配置，使得快速傅立葉變

換的運算速度能夠加速到更高的倍速。

本論文一共分為五章: 本章為簡介，說明整篇論文內容；第二章中討論背景相關知識；第三章主要是介紹實驗中所使用的平行化演算法；第四章為實驗中的效能探討；第五章則是結論以及未來展望。

第二章　背景知識

本章節詳細介紹快速傅立葉變換以及 CUDA 在 GPU 上的運作。

2.1　離散傅立葉變換

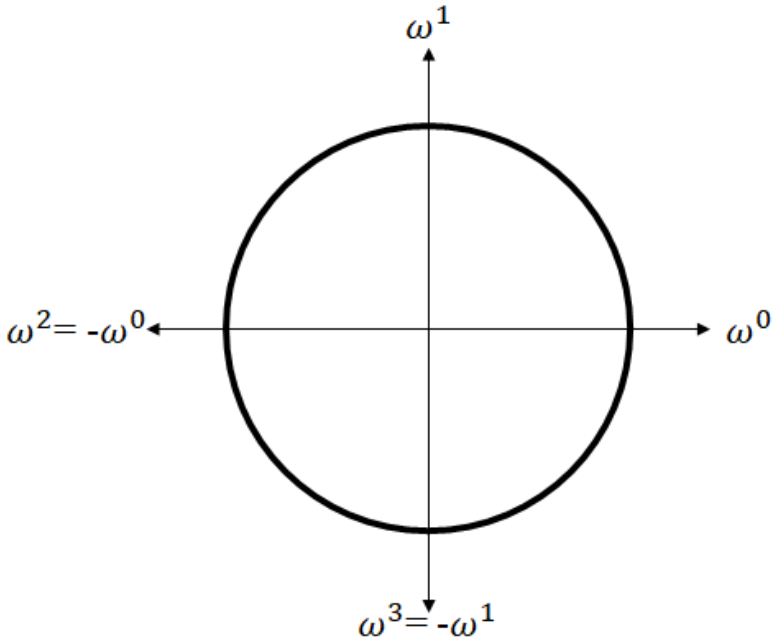
一維的離散傅立葉變換（Discrete Fourier Transform）公式如 Equation(2. 1)。

$$Y[n] = \sum_{k=0}^{N-1} X[k] \times \omega^{-nk}, n = 0, 1, \dots, N-1 \qquad (2.1)$$

其中 X 為輸入；Y 是輸出； $\omega = e^{i(2\pi/N)} = \cos\left(\frac{2\pi}{N}\right) + i \times \sin\left(\frac{2\pi}{N}\right)$；N 為資料總數，離散傅立葉變換的輸入和輸出都是一個複數數列，使用電腦作運算的都是離散數值的離散版本，本論文中的傅立葉變換都是使用離散傅立葉變換。

2.2　快速傅立葉變換

由於一般離散傅立葉變換公式的時間複雜度為 $O(N^2)$其中 N 為變數的數量，後續改進的快速傅立葉變換時間複雜度為 O(N・log N)。在離散傅立葉變換中ω具有以下特性，令 N 為資料總數；N=2a，則$-\omega^j = \omega^{j+a}, j = 0, 1, \dots, a-1$，ω的特性如 Figure 2.1。利用ω的特性能把奇數項與偶數項分開整理，整體過程在下方舉了一個 N = 4 的例子詳述整體運作的過程。



$$N = 4; \omega = \cos\left(\frac{\pi}{2}\right) + i \cdot \sin\left(\frac{\pi}{2}\right)$$

Figure 2.1: ω特性

經由傅立葉變換將向量 X={X_0, X_1, X_2, X_3}轉換成向量 Y={Y_0, Y_1, Y_2, Y_3}

根據 Equation 2.1：

$$Y_0 = X_0\omega^0 + X_1\omega^0 + X_2\omega^0 + X_3\omega^0$$

$$Y_0 = (X_0\omega^0 + X_2\omega^0) + (X_1\omega^0 + X_3\omega^0)$$

$$Y_0 = (X_0 + X_2) + \omega^0(X_1 + X_3)$$

$$Y_1 = X_0\omega^0 + X_1\omega^1 + X_2\omega^2 + X_3\omega^3$$

$$Y_1 = (X_0\omega^0 + X_2\omega^2) + (X_1\omega^1 + X_3\omega^3)$$

$$Y_1 = (X_0\omega^0 + X_2\omega^2) + \omega^1(X_1\omega^0 + X_3\omega^2)$$

$$Y_2 = X_0\omega^0 + X_1\omega^2 + X_2\omega^4 + X_3\omega^6$$

$$Y_2 = (X_0\omega^0 + X_2\omega^4) + (X_1\omega^2 + X_3\omega^6)$$

$$Y_2 = (X_0\omega^0 + X_2\omega^4) + \omega^2(X_1\omega^0 + X_3\omega^4)$$

$$Y_3 = X_0\omega^0 + X_1\omega^3 + X_2\omega^6 + X_3\omega^9$$

$$Y_3 = (X_0\omega^0 + X_2\omega^6) + (X_1\omega^3 + X_3\omega^9)$$

$$Y_3 = (X_0\omega^0 + X_2\omega^6) + \omega^3(X_1\omega^0 + X_3\omega^6)$$

由以上推算可以得到：

$$Y_0 = (X_0 + X_2) + \omega^0(X_1 + X_3)$$

$$Y_1 = (X_0\omega^0 + X_2\omega^2) + \omega^1(X_1\omega^0 + X_3\omega^2)$$

$$Y_2 = (X_0\omega^0 + X_2\omega^4) + \omega^2(X_1\omega^0 + X_3\omega^4) = (X_0 + X_2) - \omega^0(X_1 + X_3)$$

$$Y_3 = (X_0\omega^0 + X_2\omega^6) + \omega^3(X_1\omega^0 + X_3\omega^6) = (X_0\omega^0 + X_2\omega^2) - \omega^1(X_1\omega^0 + X_3\omega^2)$$

如此一來快速傅立葉變換會預先將輸入數列重新排列並且在每一層都能夠兩對一起計算，這樣一般的傅立葉變換橫向的時間複雜度 O(N) 如 Figure 2.2 就能變成 O(log N)如 Figure 2.3。整體的運算過程不僅僅減少了計算的時間，同時也因為兩兩配對而節省了記憶體的空間。

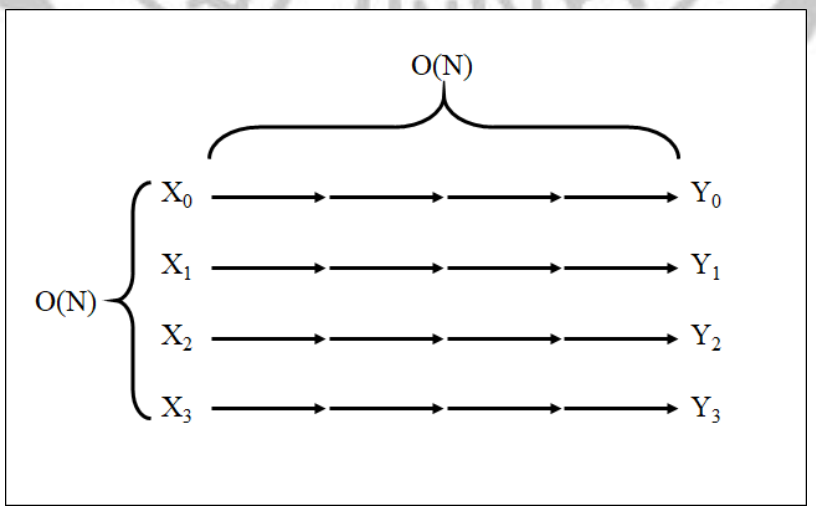


Figure 2.2: 一般傅立葉變換時間複雜度

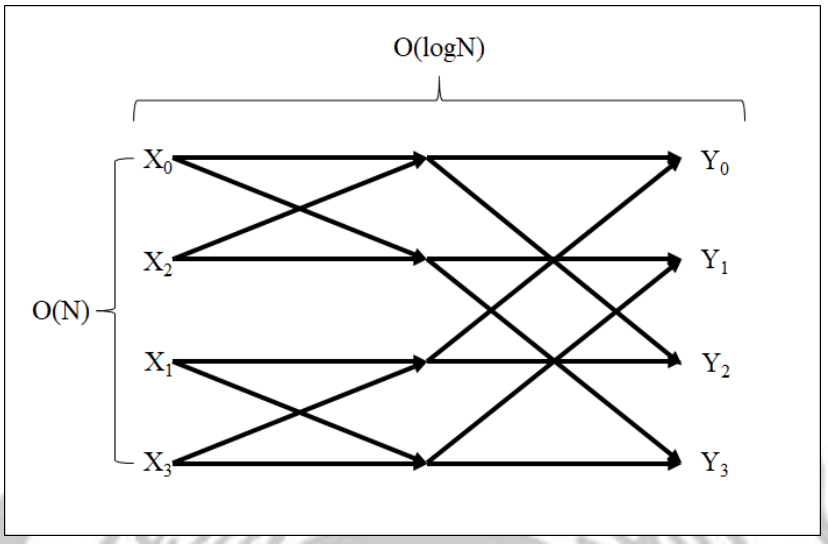


Figure 2.3: 當 N=4 時，FFT 時間複雜度

2.3 GPU 平行化運算

在最一開始 GPU 被製造出來的目的是為了進行繪圖運算的工作，然而在近年來因為 SIMD (Single Instruction Multiple Data)的應用使得 GPU 漸漸地發展出了一個新的名詞 GPGPU(General-Purpose computing on Graphics Processing Units) [3], [4]，這也讓我們開始可以在 GPU 上編寫程式。目前來說有兩種能在 GPU 上編寫程式的 API，分別是 CUDA 以及 OpenCL，其中 OpenCL 能在所有廠牌開發的 GPU 上進行編寫而 CUDA 只能在 NVIDIA 公司所開發的 GPU 上使用 [5]。而前面所說的 SIMD 是指每個執行緒會接受到同樣的指令，每個資料都進行相同的運算動作 [6]，透過 SIMD 的架構能達到對大量純數值運算的加速。在 SIMD 這種架構，欲計算 $A_i + B_i = C_i$, i = 1, 2, 3，SIMD 只會透過一個指令來命令 i 個執行緒，讓 i 個執行緒同時執行加法運算如 Figure 2.4。CPU 上則是 MIMD(Multiple Instruction Stream Multiple Data Stream)架構，MIMD 是對於各個執行緒分別下指令，每個執行緒可以執行不同的指令，欲計算 $A_i + B_i = C_i$, i = 1, 2, 3，MIMD 會分別對 i 個執行緒下達指令，每個執行緒的加法指令都是獨立的

如 Figure 2.5。雖然 GPU 無法達到 CPU 上的那種多工分配，但是僅僅是其執行緒數量就能達成大量的加速。此外，在 GPU 進行計算的時候並不會佔有 CPU 的資源，CPU 仍可進行其他運算。這樣一來，CPU 便可以在 GPU 進行運算的時候事先將下一步的配置計算並安排好，使得整體的運算效能又進一步的提升。

本論文中所使用在 GPU 上編寫程式的 API 為 NVIDIA 所開發的 CUDA，通過 CPU 發出指令讓 GPU 中的上千執行緒同時執行相同的運算指令。CPU 做出邏輯的判斷指令，由 GPU 來執行大量但單一的運算 [7]。CUDA 程式在進行編譯的時候會分成 host 與 device 兩個部分， host 的部分與 C 程式的編譯方式相同； device 的部分則是經由 NVIDIA 的編譯器。在 Figure 2.6 中紅色框中為 device 所執行的部分，而其餘則是 host 的部分。CUDA 所定義的記憶體架構如 Figure 2.7。

GPU 是由許多 SMs (Streaming Multiprocessors) 及 global memory 組成，每一個 SM 裡還有數個 SPs (Streaming Processors)、Shared memory 以及 register 等單元。程式中的一個 block 會對應到一個 SM 上，block 中的 thread 則會分配到該 SM 上的 SP。，一個 multiprocessor 只有八個 stream processor [6]，由於 stream processor 進行各種運算都有延遲，因此 CUDA 在執行程式的時候，是以 warp 為單位。目前的 CUDA 一個 warp 裡面有 32 個執行緒，若是一次沒有用滿會造成不必要的浪費，所以最佳化的程式中通常會以 32 個執行緒一組避免造成浪費 [8]。

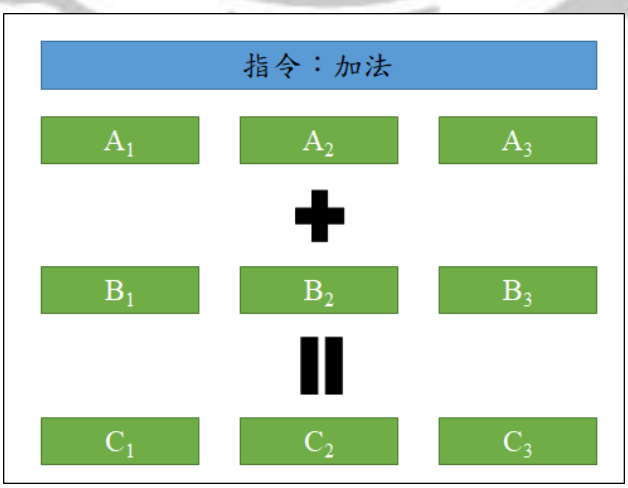


Figure 2.4: SIMD 架構

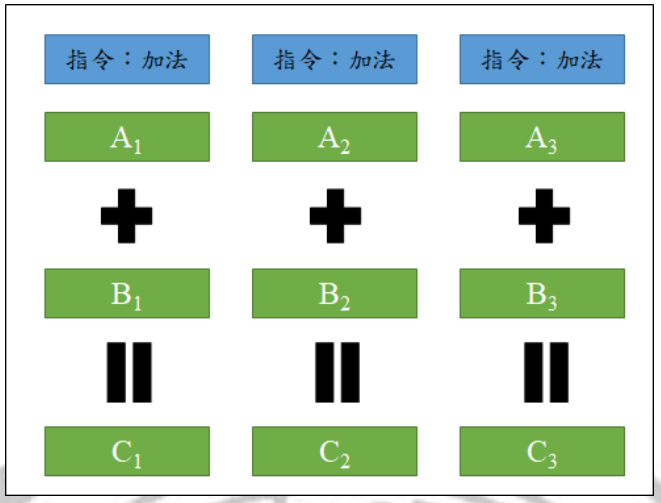


Figure 2.5: MIMD 架構

```
__global__ void addKernel(int *c, const int *a, const int *b)
{
    int i = threadIdx.x;
    c[i] = a[i] + b[i];
}

int main()
{
    const int arraySize = 5;
    const int a[arraySize] = { 1, 2, 3, 4, 5 };
    const int b[arraySize] = { 10, 20, 30, 40, 50 };
    int c[arraySize] = { 0 };

    // Add vectors in parallel.
    cudaError_t cudaStatus = addWithCuda(c, a, b, arraySize);
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "addWithCuda failed!");
        return 1;
    }

    printf("{1,2,3,4,5} + {10,20,30,40,50} = {%d,%d,%d,%d,%d}\n",
        c[0], c[1], c[2], c[3], c[4]);

    // cudaDeviceReset must be called before exiting in order for profiling and
    // tracing tools such as Nsight and Visual Profiler to show complete traces.
    cudaStatus = cudaDeviceReset();
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "cudaDeviceReset failed!");
        return 1;
    }
```

Figure 2.6: CUDA 程式範例

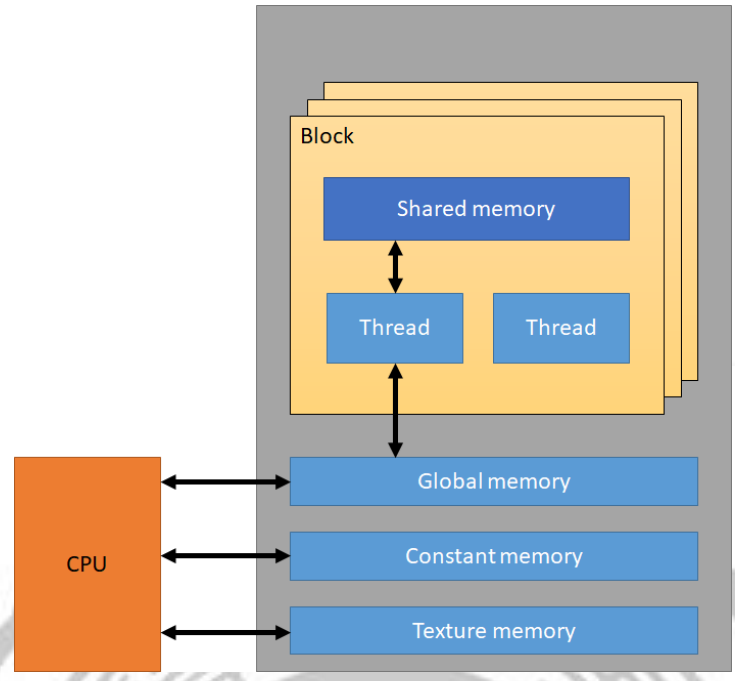


Figure 2.7: CUDA 架構

2.4 GPU 基本架構

在 NVIDIA 的 GPU 中 Streaming Processor(SP)是最基本的處理單元，多個 SP 會組成 Streaming Multiprocessor(SM)，多個 SM 會再組成 Texture Processing Clusters(TPC) [9]。而 CUDA 中並不包含 TPC 的部分，在編寫程式的過程中只需要調配 SP 與 SM 做資源調整即可。SP 與 SM 二者間的關係就像執行緒與 block 之間的關係，其架構如 Figure 2.8。

CUDA 在執行的時候會把執行緒以 warp 為單位執行，一個 warp 共包含 32 個執行緒，也就是 32 個執行緒會接受同一個指令。假設一個 block 有 512 個執行緒，那麼它會被分成 16 個 warp 執行，這樣充分的利用了每個 warp 中的執行緒。但如果 block 中的執行緒個數為 500 那麼最後一個 warp 將會有 12 個執行緒並沒有使用到，這造成了 12 個執行緒的計算浪費。另外 CUDA 會把 block 分配給 SM 進行運算，而 SM 每次只會執行 block 中的一個 warp，但是當正在執行的 warp 進入等待狀態的時候就會切換到另一個 warp 進行運算藉此隱藏執行緒的延遲時間 [10]，如 Figure 2.8。所以，如果一個 SM 裡的 warp 越多就能夠隱藏越多的延遲時間，但是相對的每個執行緒能夠分配到的資源也就越少 [11]。

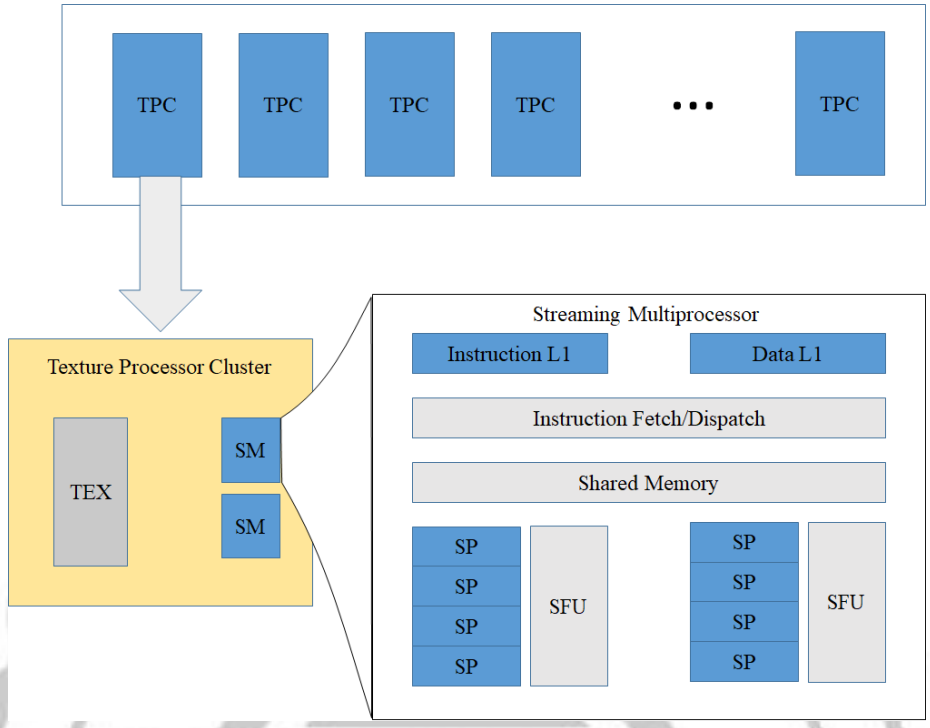


Figure 2.8: GPU 運算架構

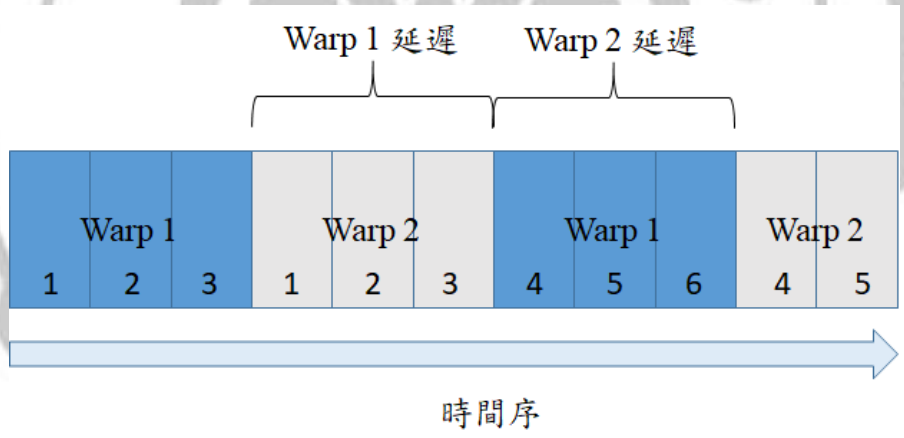


Figure 2.9: SM warp 執行

2.5 GPU 記憶體

GPU 中有許多的記憶體，在本節中只討論 GPU 中可被 CUDA 編寫的記憶體。CUDA 中的記憶體大約有以下幾種[12]：

(1) 暫存器（register)

(2) 全域記憶體（global memory)

(3) 共享記憶體（shared memory)

(4) 材質記憶體（texture memory)

(5) 常數記憶體（constant memory)

(6) 區域記憶體（local memory)

2.5.1 暫存器

所有記憶體中最快的 [13]，執行緒中大部份的變數都預設使用暫存器，但是在佔滿的情況下會使用較慢的區域記憶體 (local memory) 取代。暫存器是每個執行緒私有的，只要執行緒一結束，暫存器就會失去作用。為了維持效能通常會避免將暫存器使用完的狀況。

2.5.2 全域記憶體

全域皆可存取的記憶體，在顯卡 DRAM 中所有 thread 皆可存取的記憶體。全域記憶體是儲存空間最大，同時也是延遲最高的記憶體[11]。

2.5.3 共享記憶體

只有同 block 的 thread 才能使用，存取前後需同步化，用以避免資料存取順序混亂 [14]。使用上有大小的限制，效能僅次於暫存器。

2.5.4 材質記憶體

在 CUDA 執行 程式前，都會把資料從 host 複製到 device 的記憶體中，一般都是使用 device 的 全域記憶體 (global memory) 來直接進行存取，不過除了全域記憶體 (global memory) 以外還可透過材質記憶體 (texture memory) 來存取。透過材質記憶體存取的資料是唯讀狀態。

2.5.5 常數記憶體

常數記憶體的範圍能夠被所有 kernel 讀取，除了第一次存取需要一點時間外，其他時間與共享記憶體速度差不多，它也是唯讀記憶體 [5]。

2.5.6 區域記憶體

暫存器如果不夠用了，就會把資料儲存在這裡，在區域記憶體中的資料本質上跟全域記憶體儲存在同一區塊 [5]。

整體的記憶體速度等級分佈如 Figure 2.10。越上層的記憶體，存取速度越快；亦即是，延遲時間越小。

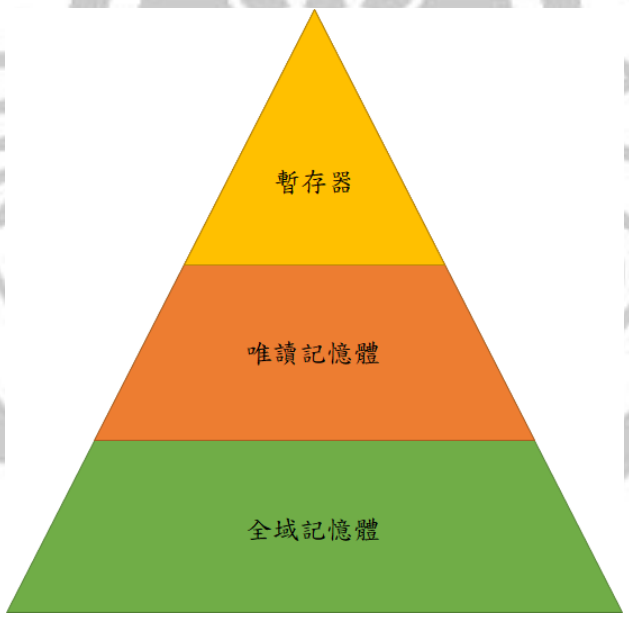


Figure 2.10: 記憶體延遲等級分佈圖

第三章　平行化演算法

本章透過在 CPU 與 CUDA GPU 上設計快速傅立葉變換演算法與執行運算。

3.1 CPU 上的 FFT

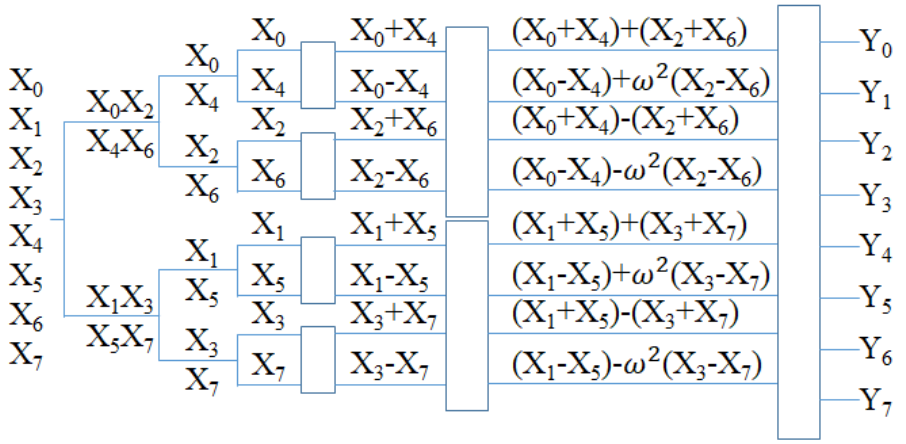
快速傅立葉變換遞迴的時間複雜度為 O(N・log N)，其演算法如 Figure 3.1。

```
FFT(X)
 //Input vector X={X0,X1,…,XN-1}
 N = length(X)
 if N == 1
     return X
 ωN = e^(2πi/N)
 ω = 1
 X^even= {X0,X2,…,XN-2}
 X^odd= {X1,X3,…,XN-1}
 // Output vector Y ={Y0,Y1,…,YN-1}
 Y^even = FFT(X^even)
 Y^odd = FFT(X^odd)
 for k = 0 ~ n/2 – 1
       Yk = Y_k^even + ωY_k^odd
       Yk+(N/2) = Y_k^even - ωY_k^odd
       ω = ω ・ ωN
 return Y
```

Figure 3.1: 快速傅立葉變換遞迴演算法

快速傅立葉變換遞迴演算法，會先由上而下分成奇數與偶數往下呼叫，直到 X 輸入陣列無法再分割為止，便會由下而上進行傅立葉變換運算，一層算完後就會再往上一層做運算直到最上層。以下舉的一個 N=8 的例子，如 Figure 3.2，先做三層 top-down 第一層先將輸入的八個 X 分成四個一組，再將四個一組分成

兩個一組，最後一層將兩個一組分成一個一組；接著做 bottom-up，第一層兩兩做 FFT，再來四個一組做 FFT，最後八個一起做 FFT。



$Y_0=(X_0+X_4)+(X_2+X_6)+((X_1+X_5)+(X_3+X_7))$
$Y_1=(X_0-X_4)+\omega^2(X_2-X_6)+\omega^2((X_1-X_5)+\omega^2(X_3-X_7))$
$Y_2=(X_0+X_4)-(X_2+X_6)+\omega^1((X_1+X_5)-(X_3+X_7))$
$Y_3=(X_0-X_4)-\omega^2(X_2-X_6)+\omega^3((X_1-X_5)-\omega^2(X_3-X_7))$
$Y_4=(X_0-X_4)+\omega^2(X_2-X_6)-((X_1+X_5)+(X_3+X_7)))$
$Y_5=(X_0-X_4)+\omega^2(X_2-X_6)-\omega^2((X_1-X_5)+\omega^2(X_3-X_7))$
$Y_6=(X_0+X_4)-(X_2+X_6)-\omega^1((X_1+X_5)-(X_3+X_7))$
$Y_7=(X_0-X_4)-\omega^2(X_2-X_6)-\omega^3((X_1-X_5)-\omega^2(X_3-X_7))$

Figure 3.2: 快速傅立葉變換遞迴演算法流程

3.2 GPU 上的平行化

快速傅立葉變換在運算上會分成奇數與偶數兩個部分來處理，其整體可以用一個蝴蝶圖來表示如 Figure 3.3。平行化就是將每個 Level 進行一次平行化運算，其疊代次數為 log N 次， N 為變數數量。快速傅立葉變換分成奇偶運算後出現了順序重排的問題，然而在 GPU 中使用 if 邏輯判斷可能會使同區塊的執行緒產生分歧效應 (branch effect) [15] 進而影響執行效能，所以在本論文裡先將輸入資料陣列 X 存入緩衝陣列 M，再使用 Equation 3.1 來對緩衝陣列 M 做尋址用以省去邏輯判斷，此處 ω 仍是以次方做運算。

$$M[id-\frac{2^v}{2}\times e]+(2\times o-1)\times M[id+\frac{2^v}{2}\times o]\times \omega^{id(\mathrm{mod}\ 2^v/2)\times N/2^v} \quad (3.1)$$

其中 id 是 threadId; v 是 level; N 是資料總數; $e=\frac{id(\mathrm{mod}2^v)}{(2^{v-1})}$; $o=1-even$;

整體平行化過程分為下列幾個步驟：

(1) 因為快速傅立葉變換資料長度必須為 2 的冪次方，所以在最一開始資料進來時會先在 CPU 中判斷資料量是否為 2 的冪次方，如果不是將會補 0 至 2 的冪次方。

(2) 將補完 0 後的資料傳送到 GPU 的記憶體中。

(3) 進行快速傅立葉變換的蝶形運算如 Figuer 3.3。在下方會以 N=8 的例子做一次完整運算。

(4) 將 GPU 中計算完畢的值送回 CPU。

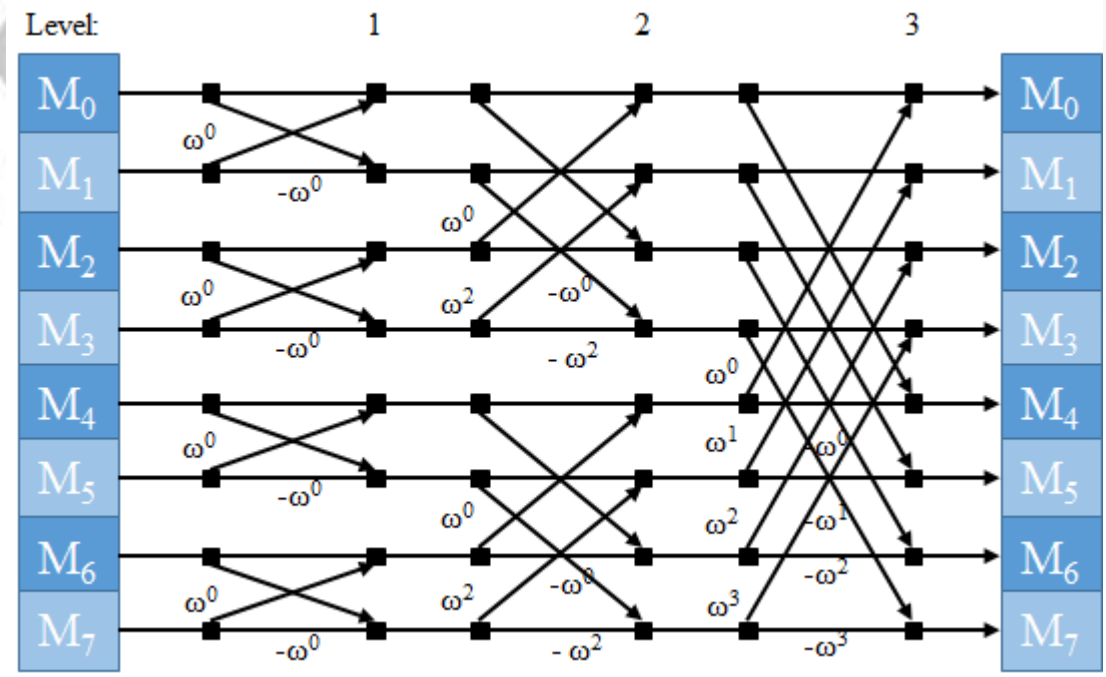


Figure 3.3: 快速傅立葉變換蝶形運算

Level = 1

M[0] = X[0]	id = 0	o = 1 e = 0	$M[0] = M[0] + M[1]\omega^0$
M[1] = X[4]	id = 1	o = 0 e = 1	$M[1] = M[0] - M[1]\omega^0$
M[2] = X[2]	id = 2	o = 1 e = 0	$M[2] = M[2] + M[3]\omega^0$
M[3] = X[6]	id = 3	o = 0 e = 1	$M[3] = M[2] - M[3]\omega^0$
M[4] = X[1]	id = 4	o = 1 e = 0	$M[4] = M[4] + M[5]\omega^0$
M[5] = X[5]	id = 5	o = 0 e = 1	$M[5] = M[4] - M[5]\omega^0$
M[6] = X[3]	id = 6	o = 1 e = 0	$M[6] = M[6] + M[7]\omega^0$
M[7] = X[7]	id = 7	o = 0 e = 1	$M[7] = M[6] - M[7]\omega^0$

Level = 2

M[0]	id = 0	o = 1 e = 0	$M[0] = M[0] + M[2]\omega^0$
M[1]	id = 1	o = 1 e = 0	$M[1] = M[1] + M[3]\omega^2$
M[2]	id = 2	o = 0 e = 1	$M[2] = M[0] - M[2]\omega^0$
M[3]	id = 3	o = 0 e = 1	$M[3] = M[1] - M[3]\omega^2$
M[4]	id = 4	o = 1 e = 0	$M[4] = M[4] + M[6]\omega^0$
M[5]	id = 5	o = 1 e = 0	$M[5] = M[5] + M[7]\omega^2$
M[6]	id = 6	o = 0 e = 1	$M[6] = M[4] - M[6]\omega^0$
M[7]	id = 7	o = 0 e = 1	$M[7] = M[5] - M[7]\omega^2$

Level = 3

M[0]	id = 0	o = 1 e = 0	$M[0] = M[0] + M[4]\omega^0$
M[1]	id = 1	o = 1 e = 0	$M[1] = M[1] + M[5]\omega^1$
M[2]	id = 2	o = 1 e = 0	$M[2] = M[2] + M[6]\omega^2$
M[3]	id = 3	o = 1 e = 0	$M[3] = M[3] + M[7]\omega^3$
M[4]	id = 4	o = 0 e = 1	$M[4] = M[0] - M[4]\omega^0$
M[5]	id = 5	o = 0 e = 1	$M[5] = M[1] - M[5]\omega^1$
M[6]	id = 6	o = 0 e = 1	$M[6] = M[2] - M[6]\omega^2$
M[7]	id = 7	o = 0 e = 1	$M[7] = M[3] - M[7]\omega^3$

由於 GPU 是 SIMD，所以執行緒每次只會執行一種指令，當 GPU 遇到 if-else 的邏輯判斷時，因為 if 與 else 有可能都需要被執行，這時便會產生分歧，而需要進行兩次記憶體存取，如 Figure 3.4。當分歧越多的時候會執行越多次，所以在 GPU 中會盡量避免使用 if 邏輯判斷。

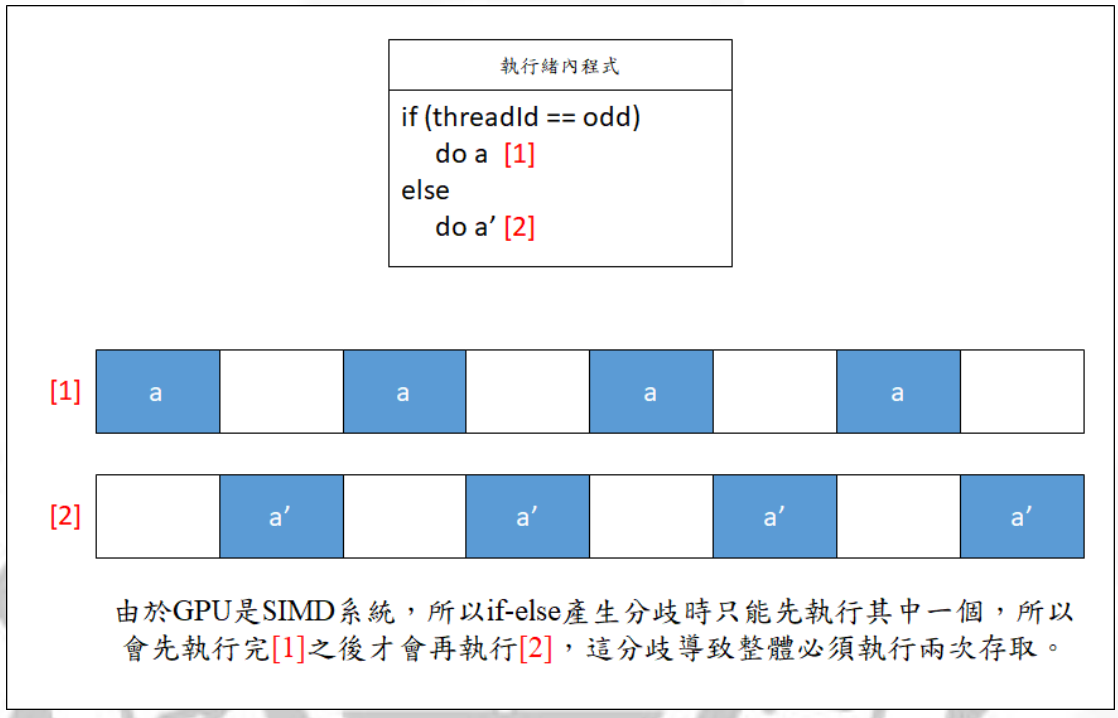


Figure 3.4: 分歧效應

3.3 最佳化

快速傅立葉變換中，ω 的計算是一個可以用更有效方式進行處理的部分。最初將 ω 的計算放在 GPU 中執行，但是這樣隨著編號越來越高每個執行緒中的 ω 計算所需時間也就越來越多。由於 ω 除了受到資料個數的影響外其值是固定的，於是我們決定將 ω 先在 CPU 中進行建表並放入 GPU 中存取較快速的唯讀記憶體中，讓 GPU 中的執行緒可以省去 ω 的計算直接從記憶體中進行讀取。這樣便能加速整體運算時間，ω 建表如 Figure 3.5，同時也將使用尋址公式 Equation 3.2 做 ω 的尋址。

$$M[id-\frac{2^v}{2}\times e]+(2\times o-1)\times M[id+\frac{2^v}{2}\times o]\times \omega[id(\mathrm{mod}\ 2^v/2)\times N/2^v] \quad (3.2)$$

CUDA 中的記憶體存包含暫存器、共享記憶體、全域記憶體、唯讀記憶體(材質記憶體以及常數記憶體),其中的存取速度最快的為暫存器,其次是共享記憶體,再來是唯讀記憶體,最後是全域記憶體。這邊將根據資料的情況,將各種資料放入延遲較小的記憶體當中來達到運行速度的最佳化。

在 GPU 中的記憶體存取,若是沒有事先進行宣告會自動將資料放入存取速度最為快速暫存器,直到暫存器的空間被使用完為止,之後的資料便會被自動放入全域記憶體。然而全域記憶體的存取延遲約為暫存器的 100 倍左右,因此我們需要將記憶體進行妥適的分配。我們將每個 block 中使用的執行緒限制在一定數量來確定不會因為使用超過暫存器的空間而使用到全域記憶體,並且確保了多數的變數存取能使用到存取最快速的暫存器。CUDA 在執行的時候執行緒會以 warp 為單位,每一個 warp 有 32 個執行緒,也就是一個 warp 會讓他擁有的 32 個執行緒進行相同的指令所以在每個 block 中使用的執行緒數量會儘量為 32 的倍數情況如 Figure 3.6 所示。接著把需要經過快速傅立葉變換計算的資料放入共享記憶體中,由於共享記憶體能夠在同一區塊內進行讀取與寫入所以將需要計算的資料放在其中。再來因為已經計算完的ω不需要再做任何的改變,所以把它放進了唯讀記憶體中以確保它能以穩定且較快的速度進行讀取。整體的記憶體配置如 Figure 3.7。

整體的配置過程分為下列幾個步驟:

(1) 因為快速傅立葉變換資料長度必須為 2 的冪次方,所以在最一開始資料進來時會先在 CPU 中判斷資料量是否為 2 的冪次方,如果不是將會補 0 至 2 的冪次方。

(2) 將補完 0 後的資料傳送到 GPU 的分享記憶體中。

(3) 劃分好每個 block 中所使用的執行緒數量以確保不使用超過既有暫存器的數量。

(4) 建立 ω 表並放入唯讀記憶體。

(5) 進行快速傅立葉變換的蝶形運算。

(6) 將 GPU 中計算完畢的值送回 CPU。

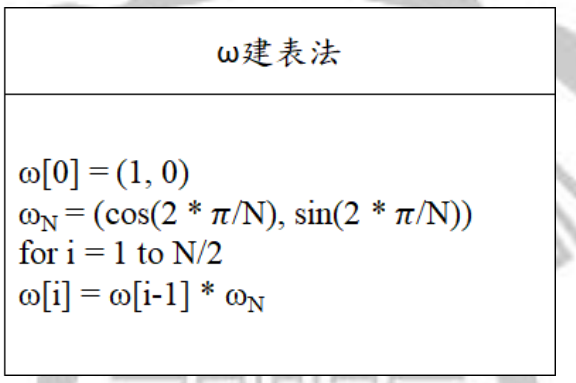


Figure 3.5: ω 建表演算法

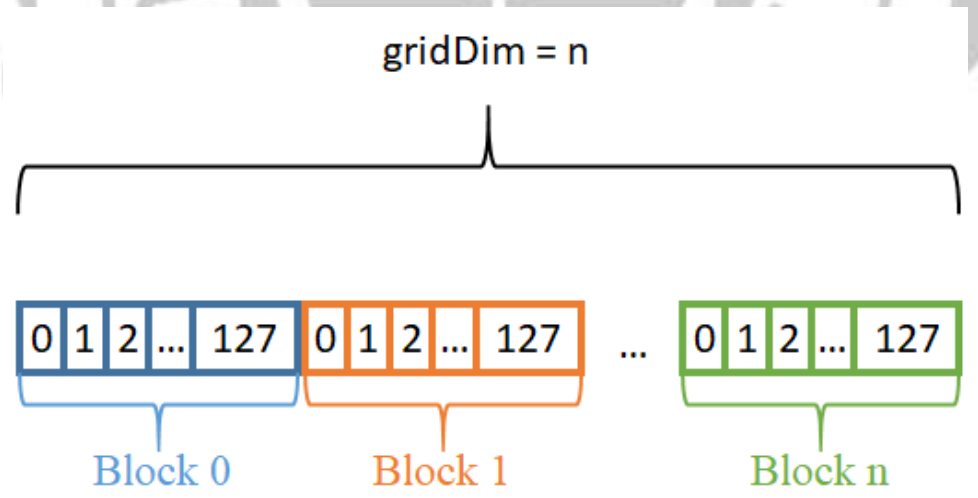


Figure 3.6: block 中的執行緒使用情形

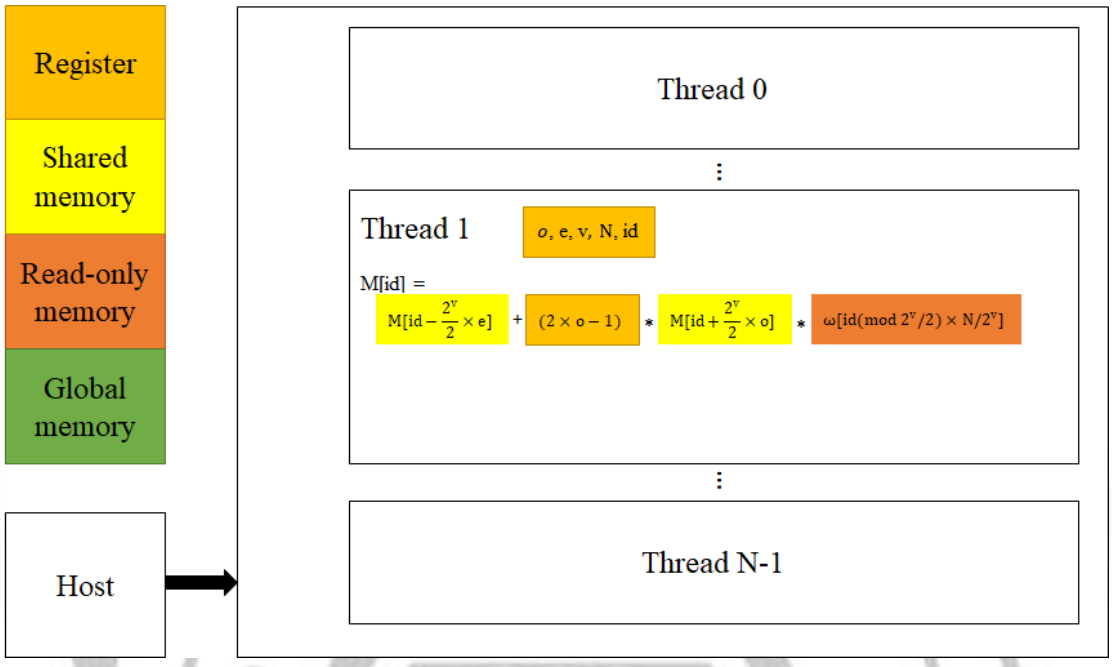


Figure 3.7: 記憶體使用情形

第四章　實驗結果

在本章將分別在 CPU 與 GPU 進行快速傅立葉變換程式設計，比較它們之間所需要的執行時間；此外，也針對配置與記憶體優化的平行化運算做效能比較。

4.1 實驗環境

本實驗的 CPU 運算環境是使用 Intel® Core™ i5-4460 @ 3.20 GHz，記憶體(RAM)共 8GB。

負責平行化部分的 GPU 運算環境則是使用 NVIDIA GeForce GTX 750 Ti，其中包含 5 個 Multiprocessors，每個 Multiprocessor 包含 128 個 CUDA Cores 共 640 個 CUDA Cores，而 CUDA Cores 也就是 SP (Streaming Processors)，每個 block 有 1024 個執行緒以及暫存器共 65536 個，同時還有 48 KB 的共享記憶體，每個 warp 有 32 個執行緒。GPU 實驗環境如 Table 4.1。

Table 4.1: GPU 實驗環境

GPU	Nvidia GTX 750Ti
Threads/Block	1024 個
Registers/Block	65536 個
Warp Size	32 threads
Shared Memory/Block	48 KB

4.2 GPU 平行化與 CPU 之比較

本次實驗中對多種資料數量進行快速傅立葉的平行化與非平行化版本並計算其執行時間，資料數量從 128 個開始每次測試將資料數量翻倍再次進行測試，最大數量測試到 32768 個。

未平行化的版本是在 CPU 上執行的單機版，是以遞迴版本的快速傅立葉變換其時間複雜度為 O(N • log N)，此版本為未經過優化版本，在 CPU 上以 C++ 進行程式編寫。

平行化的版本是以 CUDA 進行撰寫，並將ω的計算以及複數的基本加法、減法以及乘法皆寫成 function 放在 CUDA 的 device 中，最後再從 kernel 中呼叫進行運算，而所有輸入值 X 陣列以及 v 值皆是直接使用 cudaMemcpy 將資料從 Host 送到 Device 中。

4.2.1 平行化運算速度研究

實驗中平行化時間的計算是以 CUDA 計算時間為主，CPU 單機版的計算則是以傅立葉計算過程為主。

本實驗將比較平行化與單機版的速度，根據 Table 4.2 所示，可以得到以下幾點結論：

(1) 在資料量為 128 時，單機版的速度會略優於平行化版本，推測是因為 GPU 的運算單元其計算能力本身就弱於 CPU，所以在資料量沒有很多並沒有動用到很多執行緒的時候，平行化就沒有達到很好的加速效果[9]。

(2) 隨著資料量的增加，擁有數千執行緒的平行化版本的執行速度在資料量為

512 的時候超過了單機版本，並且隨著數量增加越來越高。

Table 4.2: 單機與平行化執行時間

Data size(個)	Host(ms)	GPU parallel(ms)
128	0.16	0.191552
256	0.54	0.218624
512	1.14	0.259520
1024	2.57	0.324352
2048	5.71	0.499840
4096	12.61	0.557312
8192	27.54	0.617184
16384	58.05	1.256512
32768	125.37	2.610016

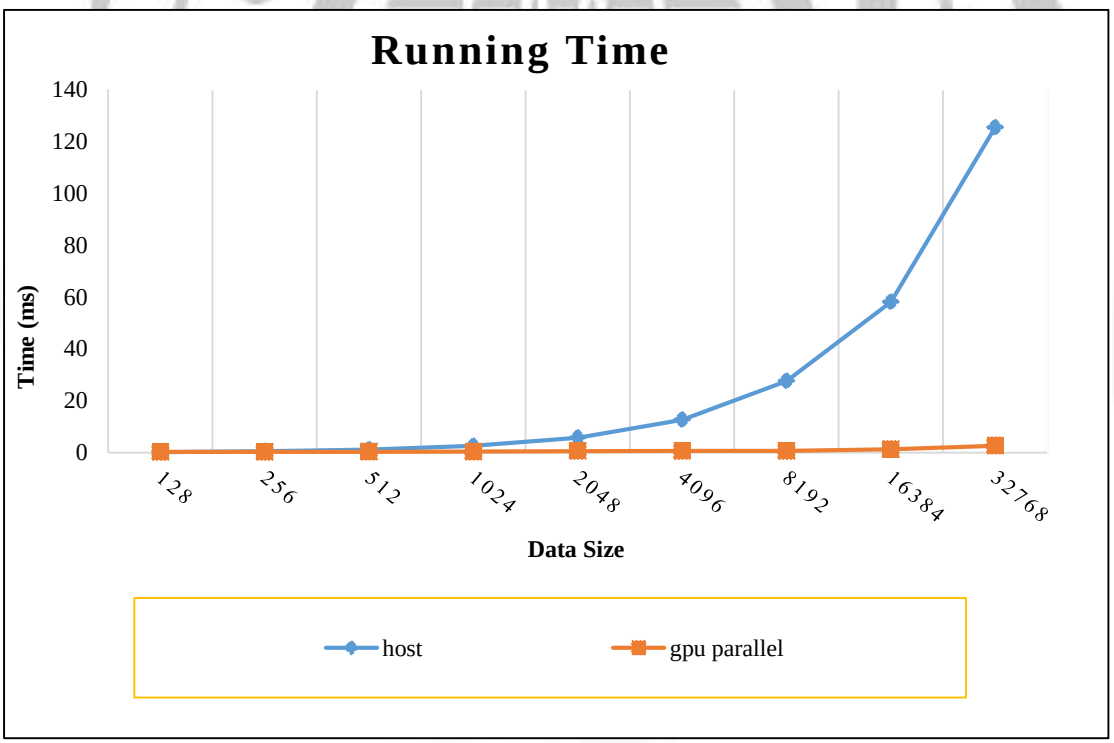


Figure 4.1: 單機版與平行化執行時間

4.2.2 平行化運算加速比研究

本節計算出平行化運算之加速比，加速比公式為 CPU 執行時間 (T_{cpu}) 與 GPU 執行時間 (T_{gpu}) 的比值如 Equation 4.1 [9]。

從 Table 4.3 與 Figure 4.2 中可得知下列幾點：

(1) 加速比理論上會呈現 2 的冪次方曲線，但因為 GPU 的處理單元運算效能較 CPU 來的差，以及各種記憶體的存取延遲導致加速比的曲線低於理論值[9]。

(2) 在資料量來到了 16384 的時候加速比又進一步的偏離了理想值，可以推測當數據量越來越多時，額外消耗的資源會抵消平行運算所帶來效能提升 [9]。

$$S = \frac{T_{cpu}}{T_{gpu}} \tag{4.1}$$

Table 4.3: 一般平行化加速比

Data size(個)	GPU parallel speedup
128	0.835
256	2.469
512	4.392
1024	7.923
2048	11.423
4096	22.626
8192	44.622
16384	46.199
32768	48.034

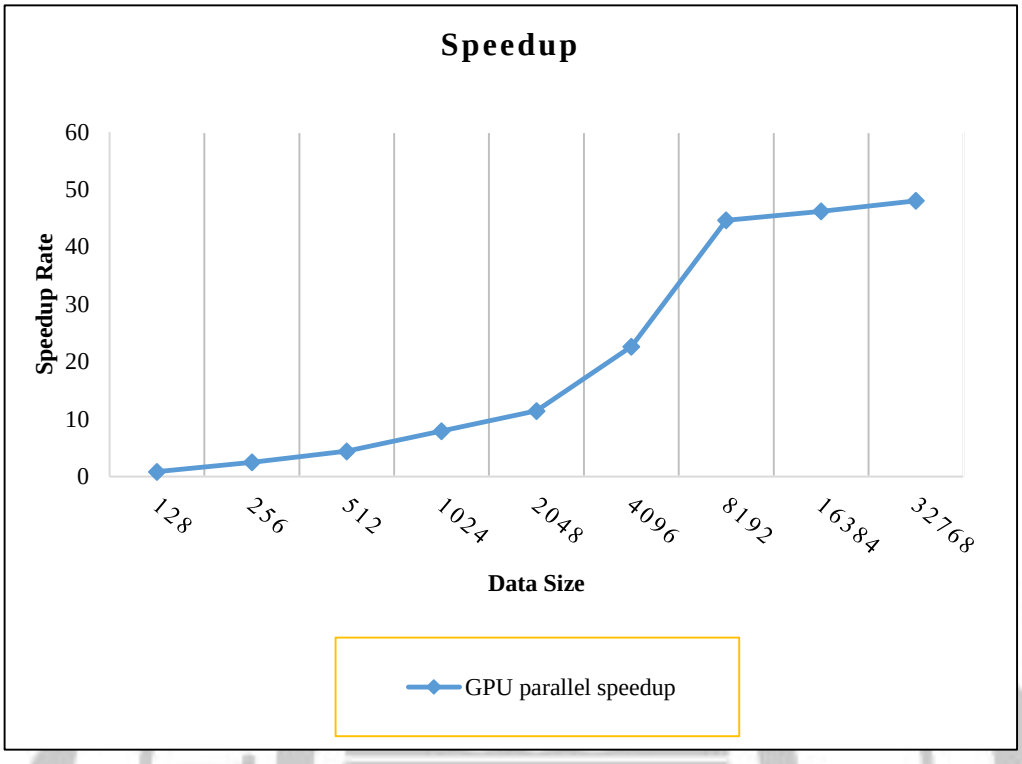


Figure 4.2: 平行化加速比

4.3 CUDA 記憶體配置後的平行化

在本節將對進行記憶體配置優化過後的平行化運算做測試，同時與尚未進行配置優化的平行化相互比較。

在本節中平行化的部分，由於事先將 ω 計算完畢，整體的計算變的非常簡單，所以便將原本 device 中的 function 直接在 kernel 中計算。輸入值 X 陣列也放入 shared memory 中的緩衝陣列 M。

本實驗將原先的平行化過程中的 ω 事先進行建表，並且放入低延遲的唯

讀記憶體，以及降低每個 block 中的執行緒數量以免佔用過多資源，並將其數量安排成 32 的倍數以達成不浪費每個 warp 的計算資源，最後再把要計算的資料放入共享記憶體中。在此實驗，優化過後的方法將與單機版進行比較。

4.3.1 進行優化過後的平行化運算速度研究

實驗中平行化時間的計算是以 CUDA 計算時間為主，CPU 單機版的計算則是以傅立葉計算過程為主。

實驗結果如 Table 4.4，可以看見在經過記憶體配置優化之後的運算效能有了顯著的提升，當資料量為 128 時的執行速度已經優於 CPU 的執行時間。雖然使用的設備不同，我們的方法與參考文獻 [16] 中的數據相比，一直到資料個數為 4096 個時效能都差不多。

Table 4.4: 單機與記憶體配置後的平行化執行時間

Data size(個)	Host(ms)	GPU optimization(ms)
128	0.16	0.091264
256	0.54	0.103840
512	1.14	0.116064
1024	2.57	0.130784
2048	5.71	0.173056
4096	12.61	0.204160
8192	27.54	0.309056
16384	58.05	0.570976
32768	125.37	1.092800

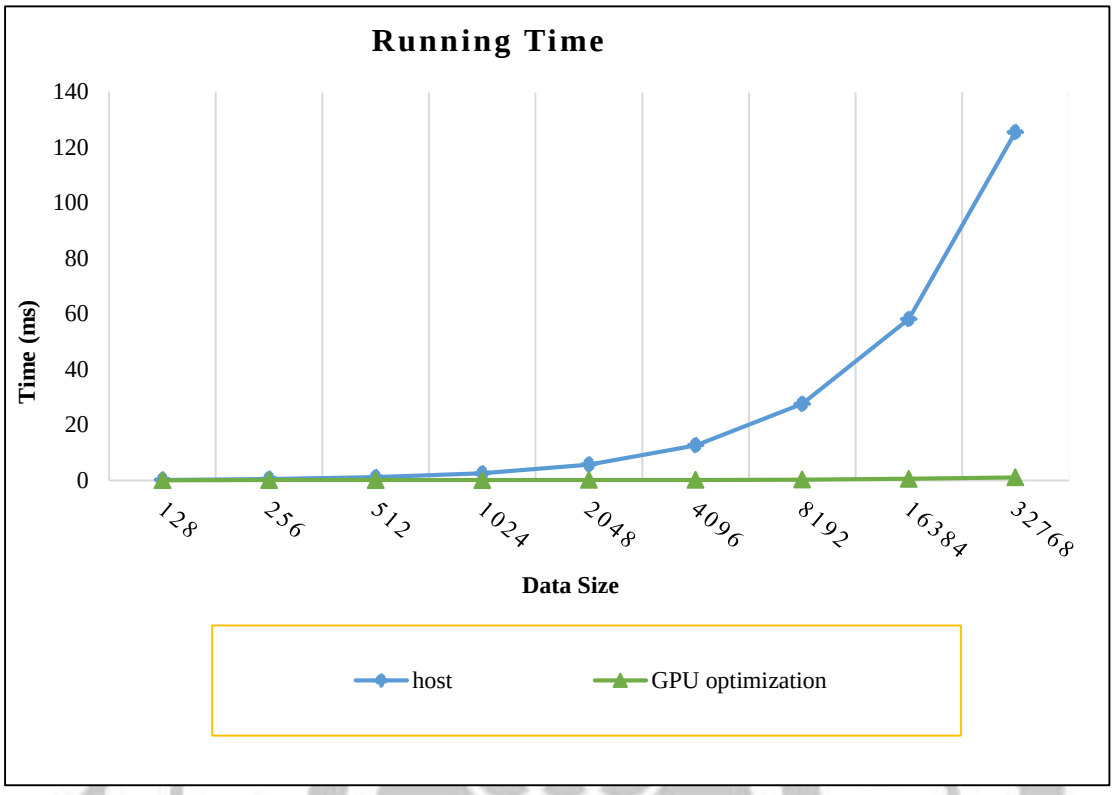


Figure 4.3: 記憶體配置後平行化與單機版執行時間

4.3.2 進行優化過後的平行化運算加速比研究

優化過後與優化前加速比的比較如 Table 4.5 所示，同樣的得到了下列幾點的結論：

(1) 加速比仍舊低於理論上會呈現 2 的冪次方曲線[9]，可以得知就算延遲較低仍然還是有一點延遲存在。

(2) 當資料量來到了 16384 的時候仍舊偏離了理想曲線，可見當數據量到達一定程度都會使得運算開始產生無法隱藏的效能下降。

Table 4.5: 記憶體配置前後之平行化加速比

Data size	GPU parallel speedup	GPU optimization speedup
128	0.835	1.753
256	2.469	5.200
512	4.392	9.822
1024	7.923	19650
2048	11.423	32.995
4096	22.626	61.765
8192	44.622	89.110
16384	46.199	101.668
32768	48.034	114.723

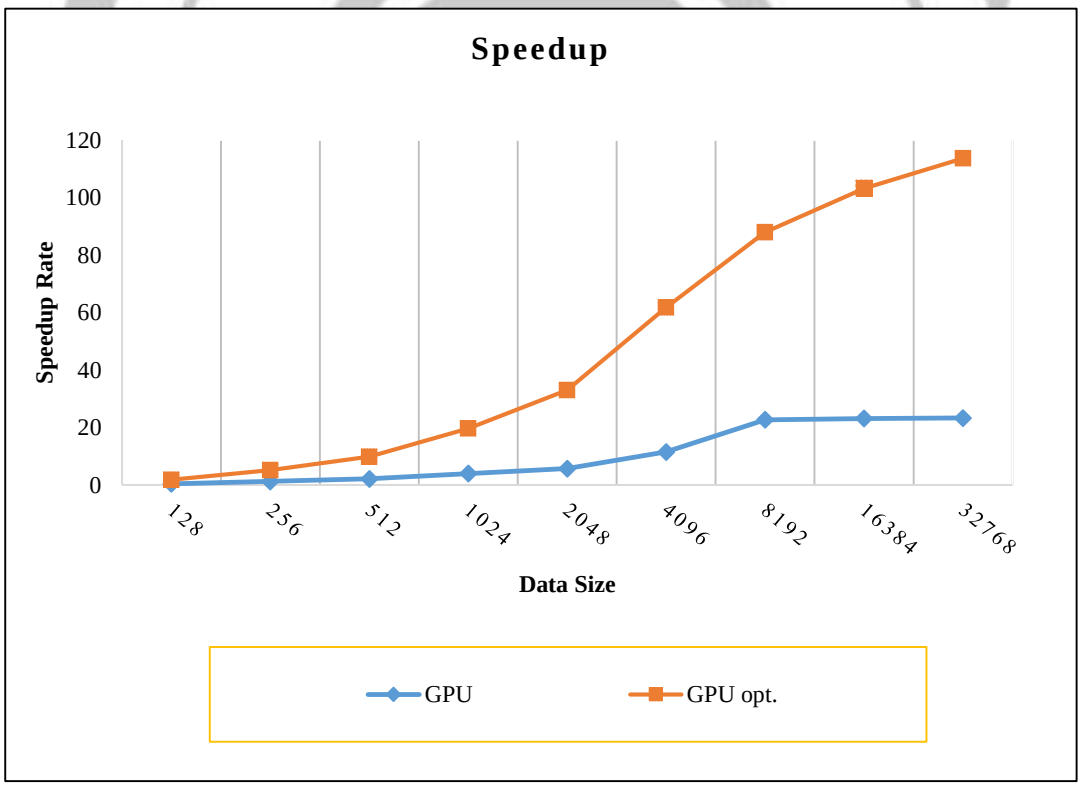


Figure 4.3: 記憶體配置前後之平行化加速比

第五章　結論

傅立葉變換在現代許多的領域之中，受到許多廣泛的應用，在本實驗中基於 CUDA 的平行化運算這項技術，透過 C-編譯器在 GPU 上對快速傅立葉變換進行平行化運算以及優化記憶體配置的平行化，加速其運算速度。

CUDA 是隨著近幾年來顯示晶片的高度可編程化衍生而來的 GPGPU 的模型，這項技術使得許多需要一次進行大量相同計算的工作能夠得到相當大量的速度提升，並降低 CPU 運算的負擔。本實驗也將現代許多領域中非常重要的傅立葉變換帶入 CUDA 中進行平行，並優化其記憶體配置，縮短運算所需的時間。比較優化記憶體前以及優化記憶體後的加速比，從實驗中得知 GPU 的平行化有效的提升了快速傅立葉運算的運算速度，經過優化後的記憶體配置更是大幅提升了平行化後的運算速度，由此可知平行化的最佳化是能夠帶來重大效益的一項動作。實驗中 GPU 大量的運算單元確實起到了不錯的加速效果，但是當資料量越大的時候執行緒所分配到的資源也就越少，得到的加速效果也並沒有預期中來的好，在後續或許可以對於 block 中的 thread 數量以及共享記憶體以及暫存器的分配再做更好的調整，讓資料量就算提高也能更好的分配資源。

參考文獻

[1] NVIDIA CUDA. http://www.nvidia.com./

[2] John Nickolls, William J. Dally, “The GPU Computing Era,” IEEE Micro Year 2010, Volume 30, Issue 2, pp. 56 – 69.

[3] K. Fatahalian and M. Houston, “A closer look at GPUs,” Communications of the ACM, Vol. 51, No. 10, October 2008.

[4] Dimitrov, Martin, Mike Mantor, and Huiyang Zhou, "Understanding Software Approaches for GPGPU Reliability," Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units. ACM, 2009.

[5] Su, Ching-Lung, et al, "Overview and Comparison of OpenCL and CUDA Technology for GPGPU," 2012 IEEE Asia Pacific Conference on Circuits and Systems (APCCAS).

[6] John Nickolls, Ian Buck, and Michael Garland, Kevin Skadron, “Scalable Parallel Programming with CUDA,” Queue, March/April 2008, Volume 6, Issue 2, pp.40-53.

[7] Michael Garland “CUDA Parallel Programming Model,” 2008 IEEE Hot Chips 20 Symposium (HCS), pp.1 – 29.

[8] Garland, Michael, et al, "Parallel Computing Experiences with CUDA," IEEE Micro 2008, Volume 28, Issue 4.

[9] Xueqin Zhang, Kai Shen, Chengguang Xu, Kaifang Wang, “Design and Implementation of Parallel FFT on CUDA,” 2013 IEEE 11th International Conference on Dependable, Autonomic and Secure Computing, pp. 583 – 589.

[10] Erik Lindholm, John Nickolls, Stuart Oberman, John Montrym, “NVIDIA Tesla: A Unified Graphics and Computing Architecture” IEEE Micro, March-April 2008, Volume 28, Issue 2.

[11] Harris, Mark, "Optimizing cuda," SC07: High Performance Computing with CUDA 2007.

[12] NVIDIA Corporation, “NVIDIA CUDA Programming Guide”.

[13] Ryoo, Shane, et al, "Program Optimization Space Pruning for a Multithreaded GPU," Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization. ACM, 2008.

[14] Yang, Zhiyi, Yating Zhu, and Yong Pu, "Parallel Image Processing Based on CUDA,"IEEE 2008 International Conference on Computer Science and Software Engineering, pp.198-201.

[15] Vaidya, Aniruddha S., et al, "SIMD Divergence Optimization through Intra-warp Compaction," ACM SIGARCH Computer Architecture News. Vol. 41. No. 3. ACM, 2013.

[16] Zhang, Fan, Chen Hu, Qiang Yin and Wei Hu, “A GPU Based Memory Optimized Parallel Method For FFT Implementation,” Computing Research Repository (CoRR) abs/1707.07263 (2017)