ĐẠI HỌC BÁCH KHOA HÀ NỘI

TRƯỜNG ĐIỆN - ĐIỆN TỬ

-----□□□□□-----

BÁO CÁO BÀI TẬP LỚN
LẬP TRÌNH SONG SONG

ĐỀ TÀI: SONG SONG HÓA THUẬT TOÁN FAST FOURIER TRANSFORM

Giảng viên hướng dẫn:	TS Phạm Doãn Tĩnh	
Nhóm:	9	
Sinh viên thực hiện:	Võ Bá Thông	20203887
	Giáp Thành Trung	20200639
	Nguyễn Trọng Tuấn	20203892
	Đỗ Xuân Chiến	20203874
	Lê Hữu Toàn	20203889
Mã lớp:	144005	

Hà Nội, 1 – 2024

MỤC LỤC

CHƯƠNG 1. TỔNG QUAN THUẬT TOÁN

1.1 Thuật toán Fast Fourier Transform (FFT)

1.1.1 Khái niệm

Được đặt tên theo nhà toán học người Pháp Jean - Baptiste Joseph Fourier cuối thế kỷ 18, biến đổi Fourier là một phép toán biến đổi tín hiệu từ miền thời gian (không gian) sang miền tần số. Theo định lý Fourier, một tín hiệu là sự hợp thành của một số hàm điều hòa với biên độ, tần số và pha cho trước. Dạng chuỗi Fourier cho tín hiệu tuần hoàn, x(t):

$$X(t) = \frac{A_0}{2} + \sum_{n=1}^{N} (A_n \sin(\frac{2\pi nt}{P} + \phi_n) = \sum_{n=-N}^{N} (C_n e^{\frac{2\pi jnt}{P}})$$

Trong đó, P là chu kỳ của tín hiệu và

C_n được gọi là các hệ số Fourier của tín hiệu $x(t)$. Các hệ số này được tính bằng cách:

$$C_n = \frac{1}{P}\int_0^P e^{-\frac{2\pi nt}{P}} x(t)dt$$

Tuy nhiên, hầu hết các tín hiệu không tuần hoàn. Các tín hiệu không tuần hoàn không có chuỗi Fourier. Thực tế này làm Fourier thất vọng, ông phải mất gần 20 năm để phát triển một công thức chung có thể hoạt động cho bất kỳ hàm nào. Bây giờ, mọi tín hiệu đều có thể được chuyển từ miền thời gian sang miền tần số theo phương trình sau đây:

$$X(\Omega) = \sum_{n=-\infty}^{\infty} x[k]e^{-j\Omega nk}$$

[Lưu ý O là tần số rời rạc]

Phương trình này không chỉ đơn thuần là toán học. Nó mô tả các khối cấu thành các tín hiệu, và từng khối riêng biệt. Tức là, cho dù các thành phần của tín hiệu dù nhỏ hay lớn, chúng đều sẽ xuất hiện trong danh sách thành phần được cung cấp bởi phép biến đổi. Xét một tín hiệu phức tạp bao gồm một sóng sin và một hàm sinc như hình dưới:

Biểu diễn tín hiệu trên miền thời gian không có gì hữu ích. Phổ tần số biên độ (Hình 4) cho thấy rõ ràng thành phần của nó. Lưu ý rằng những thành phần mong muốn có thể được phân tích một cách riêng biệt. Đó là khả năng của phép biến đổi.

Công thức Fourier tìm được thực sự tuyệt vời. Nhưng sẽ phải thực hiện việc biến đổi với khối lượng tính toán lớn. Số lượng phép tính cần thiết để xử lý phương trình này tỉ lệ với bình phương hệ số N cần có.

$$X(k) = \sum_{n=0}^{N-1} x(n) e^{-j2\pi kn/N}$$

Rõ ràng, vấn đề trở nên phức tạp khi càng số lượng hệ số càng lớn vì số lượng các phép tính tăng lên nhanh chóng đến mức không thực tế. Các máy tính đầu tiên đã mất hàng trăm giờ để thực hiện một phép biến đổi đơn giản theo các tiêu chuẩn hiện nay. Do đó, kể từ năm 1805 đã có những nỗ lực để nâng cao hiệu quả của thuật toán. Cùng năm đó, Carl Friedrich Gauss đã phát minh ra một phương pháp biến đổi Fourier hiệu quả. Tuy nhiên, nó vẫn không rõ ràng trong 160 năm nữa.

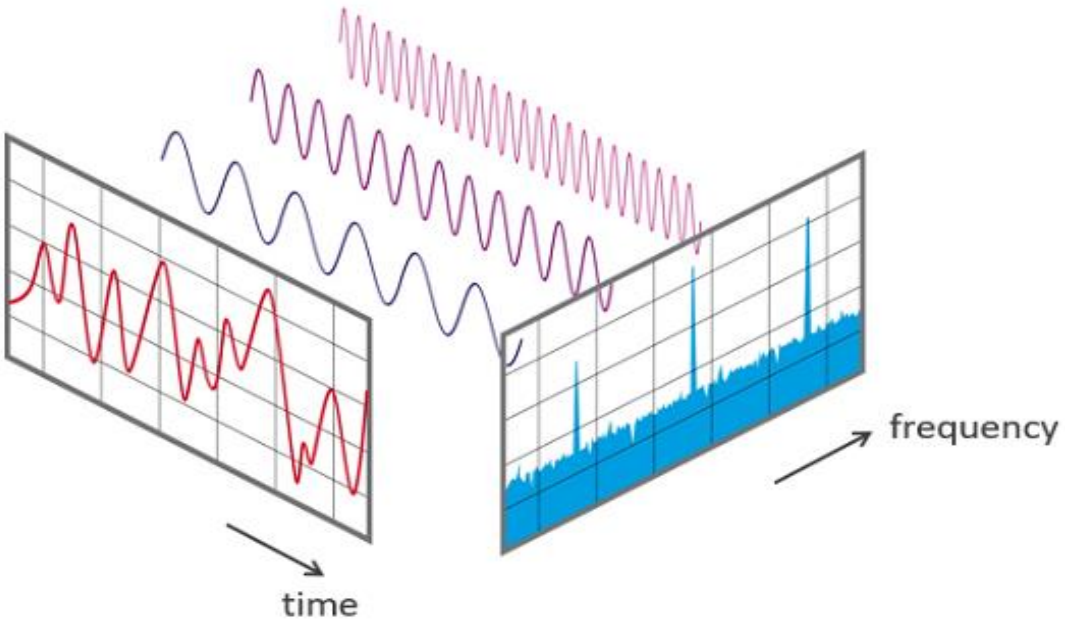
Năm 1965, James Cooley của IBM và John Tukey của Princeton đã khám phá ra thuật toán này và phổ biến nó. Việc làm của họ làm giảm đáng kể số lượng phép tính. Sự suy giảm đặc biệt đáng chú ý khi bằng lũy thừa của 2. Thuật toán được gọi là FFT ngay sau khi được giới thiệu. Nó lập tức tạo nên sự cách mạng trong biến đổi Fourier của các tín hiệu. Với 64000 điểm FFT, thuật toán này nhanh gấp 4000 lần so với phương pháp ban đầu. Nó đã có một số sửa đổi kể từ khi phát hiện ra. Và vào tháng 1 năm 2000, nó được đưa vào Top 10 Thuật toán của thế kỷ 20.

$$X(k) = \sum_{n=0}^{N-1} x(n) W_N^{kn}, \qquad k = 0,1,\ldots, N-1$$
$$= \sum_{n \text{ even}} x(n) W_N^{kn} + \sum_{n \text{ odd}} x(n) W_N^{kn}$$
$$= \sum_{m=0}^{(N/2)-1} x(2m) W_N^{2mk} + \sum_{m=0}^{(N/2)-1} x(2m+1) W_N^{k(2m+1)}$$

1.1.2 Chức năng

Thuật toán FFT chủ yếu được sử dụng để phân tích tín hiệu và dữ liệu thời gian thành các thành phần tần số, giúp hiểu rõ về cấu trúc và đặc điểm của tín hiệu. Cụ thể, chức năng của FFT bao gồm:

- Chuyển đổi tín hiệu từ miền thời gian sang miền tần số để phân tích tần số của các thành phần.
- Tính toán phổ tần số để xác định các thành phần tần số quan trọng.
- Áp dụng rộng rãi trong các lĩnh vực như xử lý âm thanh, xử lý hình ảnh, và truyền thông.



1.2 Mã giả thuật toán

Trong phần này chúng ta sẽ xét một số thuật toán để tính nhanh các giá trị của DFT được gọi là thuật toán biến đổi Fourier nhanh (FFT). Thuật toán FFT phải tính tất cả N giá trị

của DFT sao cho có hiệu quả cao nhất. Nếu yêu cầu tính toán chỉ một phần của vùng tần số $0 \leq \omega \leq 2\pi$ thì các thuật toán khác có thể hiệu dụng và mềm dẻo hơn, ví dụ như các thuật toán Goertzel, hay như thuật toán biến đổi tiếng hót.

Tiếp theo chúng ta xét vấn đề tính nhanh các số F(n) được xác định bởi công thức (1.21). Ta có $F(n) = f(0) + f(1)W_N^{-n} + f(2)W_N^{-2n} + \ldots + f(N-1)W_N^{-(N-1)n}$, $n = 0, 1, \ldots, N-1$. Như vậy với mỗi n > 0 chúng ta cần phải tiến hành $N-1$ phép nhân phức (tương ứng với $4(N-1)$ phép nhân thực) và $N-1$ phép cộng phức (tương ứng với $2(N-1)$ phép cộng thực). Như vậy để tìm được tất cả các số $F(0), F(1), \ldots, F(N-1)$ cần phải thực hiện $(N-1)^2$ phép nhân phức và $N(N-1)$ phép cộng phức. Suy ra với N càng lớn thì số phép tính cần thực hiện càng tăng lên rất nhiều.

Thuật giải FFT chỉ áp dụng cho trường hợp $N = 2s, s \in N$. Vì N chẵn, nên tổng (1.21) có thể phân tích thành hai tổng.

$$\begin{aligned} F(k) &= \sum_{n=0}^{N-1} f(n)W^{-kn} = \sum_{n \text{ chẵn}} f(n)W^{-kn} + \sum_{n \text{ lẻ}} f(n)W^{-kn} \\ &= \sum_{m=0}^{N/2-1} f(2m)(W^2)^{-km} + W^{-k} \sum_{m=0}^{N/2-1} f(2m+1)(W^2)^{-km} \\ &:= F_e(k) + W^{-k}F_o(k). \end{aligned}$$

Vì $W^2 = e^{2.2\pi i/N} = e^{2\pi i/(N/2)}$, nên ta thấy $Fe(k)$ và $Fo(k)$ lần lượt là biến đổi Fourier của hai dãy $\{f(2m) | m = 0, 1, \ldots, N/2 - 1\}$ và $\{f(2m+1) | m = 0, 1, \ldots, N/2 - 1\}$. Có nghĩa là mỗi một $Fe(k)$ và $Fo(k)$ được phân tích thành tổng của hai phép biến đổi Fourier rời rạc của $N/2$ điểm. Tiếp tục quá trình trên cho đến khi cho đến khi ta được biến đổi Fourier rời rạc của 2 điểm. Ngoài ra, do tính tuần hoàn chu kỳ $N/2$ nên chỉ cần tính $Fe(k)$ và $Fo(k)$ với $N/2 \leq k \leq N-1$ hoặc với $0 \leq k \leq N/2 - 1$.

Lặp lại việc tách tổng như trên đối với Fe(k), Fo(k), ta có:

$$F_e(k) = \sum_{m=0}^{N/2-1} f(2m)(W^2)^{-km} = \sum_{p=0}^{N/4-1} f(4p)(W^4)^{-kp}$$

$$+ W^{-2k} \sum_{p=0}^{N/4-1} f(4p+2)(W^4)^{-kp} = F_{ee} + W^{-2k} F_{eo},$$

$$F_o(k) = \sum_{m=0}^{N/2-1} f(2m+1)(W^2)^{-km} = \sum_{p=0}^{N/4-1} f(4p+1)(W^4)^{-kp} +$$

$$W^{-2k} \sum_{p=0}^{N/4-1} f(4p+3)(W^4)^{-kp} = F_{oe} + W^{-2k} F_{oo}.$$

$$W^4 = e^{4.2\pi i/N} = e^{2\pi i/(N/4)} = W_{N/4}.$$

Thực hiện quá trình trên đối với $Fee,\ Feo$, ..., ta có:

$$F_{ee}(k) = \sum_{p=0}^{N/4-1} f(4p)(W^4)^{-kp} = \sum_{q=0}^{N/8-1} f(8q)(W^8)^{-kq}$$

$$+ W^{-4k} \sum_{q=0}^{N/8-1} f(8q+4)(W^8)^{-kq} = F_{eee} + W^{-4k} F_{eeo},$$

$$F_{eo} = \sum_{q=0}^{N/4-1} f(4p+2)(W^4)^{-kp} = \sum_{q=0}^{N/8-1} f(8q+2)(W^8)^{-kq}$$

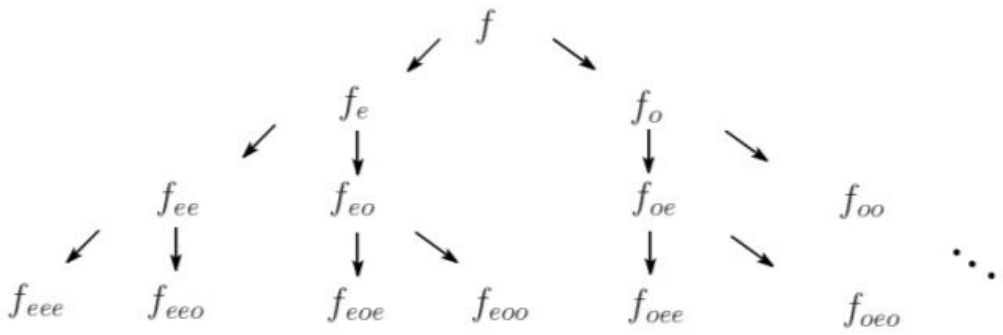
$$+ W^{-4k} \sum_{q=0}^{N/8-1} f(8q+6)(W^8)^{-kq} = F_{eoe} + W^{-4k} F_{eoo}.$$

Dưới đây chúng ta sẽ chứng minh rằng, nếu $N = 2s$ và $f(n) \in C$ N , thì để tính $F(m) = FN\,[f](m)$ bằng thuật toán FFT cần $\frac{N}{2} log2N$ phép nhân phức thay cho $(N-1)^2$ phép nhân phức và $Nlog2N$ phép cộng

phức thay vì $N(N\ -\ 1)$. Xét ví dụ sau đây để thấy số phép toán giảm đi đáng kể. Với $N\ =\ 2^6\ =\ 64$, thì

$$\frac{N}{2}log2N = 192,\ (N\ -\ 1)^2 = 63^2 =\ 3969,$$
$$Nlog2\ N =\ 384, N(N\ -\ 1)\ =\ 4032.$$

Như vậy, số phép nhân phức giảm đi khoảng 20 lần, còn số phép cộng phức giảm đi khoảng 10 lần.



Dưới đây là mã giả thuật toán FFT cơ bản:

```
def FFT(P):
    # P - [p_0, p_1, ..., p_{n-1}] coeff representation
    n = len(P) # n is a power of 2
    if n == 1:
        return P
    ω = e^{2πi/n}
    P_e, P_o = [p_0, p_2, ..., p_{n-2}], [p_1, p_3, ..., p_{n-1}]
    y_e, y_o = FFT(P_e), FFT(P_o)
    y = [0] * n
    for j in range(n/2):
        y[j] = y_e[j] + ω^j y_o[j]
        y[j + n/2] = y_e[j] - ω^j y_o[j]
    return y
```

CHƯƠNG 2. OPENMP VÀ CUDA

2.1 Giới thiệu về OpenMP

OpenMP là một tập hợp các chỉ thị của trình biên dịch cũng như API dành cho các chương trình được viết bằng C, C++ hoặc FORTRAN cung cấp hỗ trợ cho lập trình song song trong môi trường shared-memory OpenMP xác định các vùng song song dưới dạng các khối mã có thể chạy song song.

Các trình biên dịch hỗ trợ OpenMP:

- gcc: sử dụng –fopenmp
- Clang++: sử dụng –fopenmp
- SolarisStudio: sửdụng –xopenmp
- IntelC compiler: sử dụng –openmp
- MicrosoftVisual C++: sử dụng /openmp

2.1.1 OpenMP trong C/C++

OpenMP bao gồm một tập các chỉ dẫn biên dịch #pragma nhằm chỉ dẫn cho chương trình hoạt động Các pragma được thiết kế nhằm mục đích nếu các trình biên dịch không hỗ trợ thì chương trình vẫn có thể hoạt động bình thường, nhưng sẽ không có bất kỳ tác vụ song song nào được thực hiện như khi sử dụng OpenMP.

2.1.2 Cú pháp

Cú pháp OpenMP được sử dụng trong C/C++ thông qua #pragma omp, được áp dụng trực tiếp cho đoạn chương trình ngay sau nó. Cú pháp này giúp thực hiện song song các phần của chương trình.

Chỉ dẫn Parallel

Chỉ dẫn parallel bắt đầu một đoạn mã thực hiện song song. Nó tạo ra một nhóm gồm N luồng (N được xác định tại thời điểm chạy chương trình) và các lệnh sau #pragma hoặc block tiếp theo (nằm trong {}) sẽ được thực hiện song song.

```
#pragma omp parallel
{
   // Code trong vùng này chạy song song.
   printf("Hello!\n");
}
```

Chỉ dẫn For

Chỉ dẫn for chia vòng lặp for thành các phần, mỗi luồng trong nhóm thực hiện một phần của vòng lặp.

```
#pragma omp for
for (int n = 0; n < 10; ++n)
```

```
{
   printf(" %d", n);
}
printf(".\n");
```

Vòng lặp này có thể in ra màn hình không theo thứ tự.

2.2 CUDA (Compute Unified Device Architecture)

2.2.1 Giới thiệu tổng quan

CUDA (Compute Unified Device Architecture) là một nền tảng tính toán song song và mô hình lập trình, ra đời từ NVIDIA, dành cho việc tính toán tổng quát trên GPU (Graphics Processing Units). Đây là một công cụ mạnh mẽ giúp tận dụng sức mạnh của GPU cho phần tính toán có thể song song hóa, từ đó gia tăng hiệu suất ứng dụng. Điểm nổi bật của CUDA:

- Tính toán song song: CUDA cho phép lập trình viên tận dụng hàng nghìn lõi xử lý trên GPU để thực hiện các phép tính đồng thời, cải thiện hiệu suất tính toán đáng kể so với việc sử dụng CPU.
- Linh hoạt và hiệu quả: CUDA hỗ trợ việc lập trình bằng C/C++ và các ngôn ngữ khác, cho phép người dùng tận dụng sức mạnh của GPU trong nhiều lĩnh vực khác nhau như khoa học, máy học, kỹ thuật số, và các ứng dụng có yêu cầu tính toán lớn.
- Tối ưu hóa hiệu suất: CUDA cung cấp các công cụ để tối ưu hóa mã CUDA và tận dụng tối đa khả năng tính toán của GPU, bao gồm cấu trúc grid/block, quản lý bộ nhớ, và các công cụ phân tích hiệu suất.
- Phát triển cộng đồng: Cộng đồng lập trình CUDA ngày càng phát triển, có nhiều tài liệu, ví dụ và hỗ trợ từ NVIDIA và cộng đồng lập trình viên.

2.2.2 Cấu trúc và biên dịch của CUDA

CUDA yêu cầu một kiến trúc bao gồm bộ xử lý chủ (host processor), bộ nhớ chủ (host memory) và một GPU (card đồ họa với bộ xử lý Nvidia hỗ trợ CUDA). Các GPU hỗ trợ CUDA chủ yếu là các bộ xử lý đồ họa thực hiện các đường ống đồ họa thống nhất được đề xuất bởi tiêu chuẩn DirectX 10. CUDA có thể chạy trên nhiều dòng GPU của Nvidia như GeForce, Quadro và Tesla.

CUDA sử dụng một phiên bản mở rộng của ngôn ngữ lập trình C, giúp đơn giản hóa việc lập trình cho CPU và GPU.

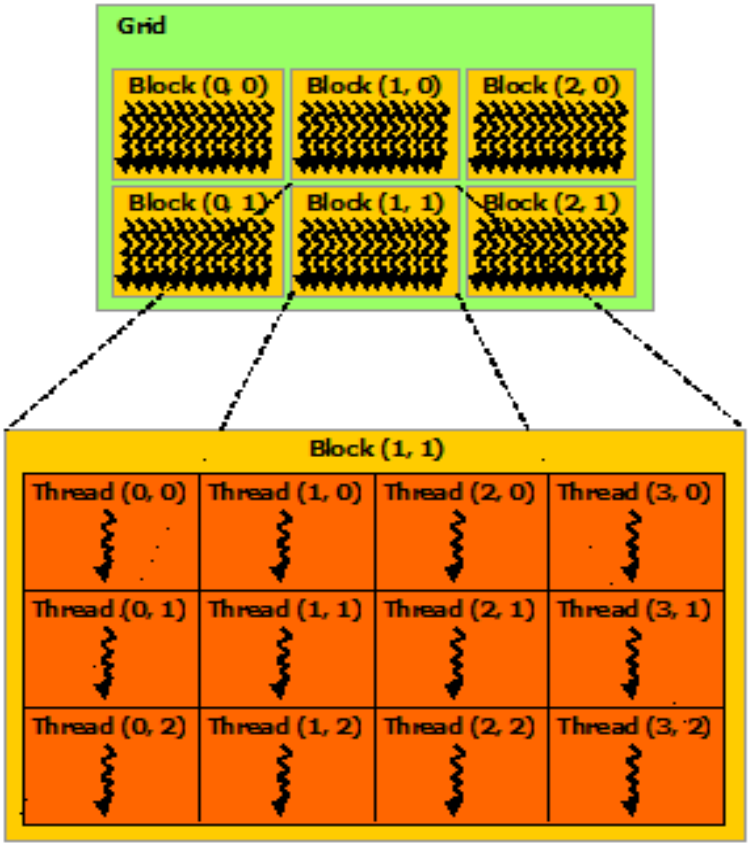
2.2.3 Cách thức hoạt động

CUDA tận dụng sức mạnh của GPU thông qua việc chia công việc thành các block và sử dụng khái niệm song song hóa để thực hiện tính toán trên các lõi xử lý của GPU. CUDA quy định mỗi 1 luồng của nó là 1 đơn vị xử lý và mỗi 1 luồng sẽ được thực thi

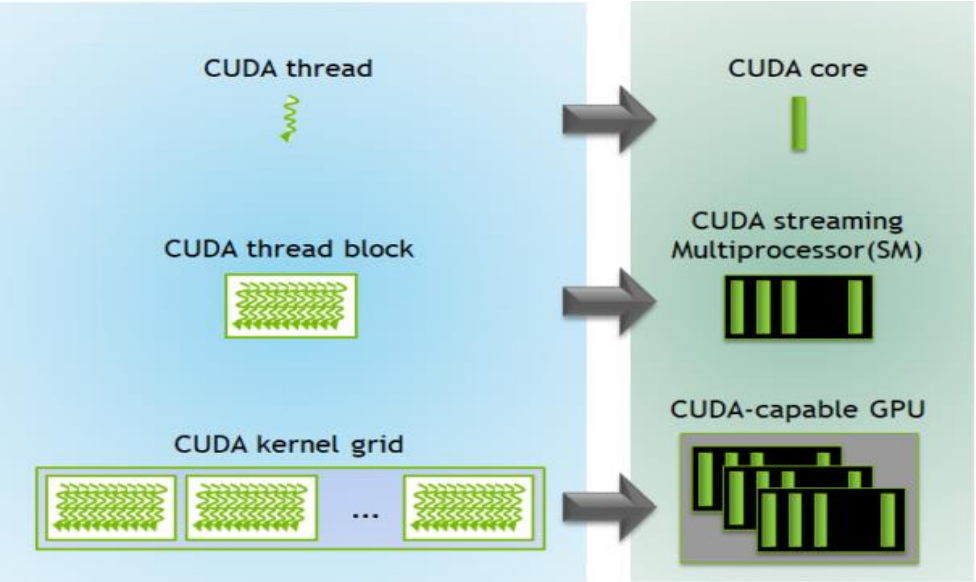
bởi 1 lõi CUDA (1 lõi xử lý). Kernel được định nghĩa là chương trình được thực thi bởi 1 luồng CUDA.

Block và Grid trong CUDA:

- Block: Là tập hợp các lõi xử lý CUDA trên một multiprocessor. Mỗi block chứa một số lượng lõi xử lý CUDA và các lõi này chạy song song trên cùng một multiprocessor.
- Grid: Là tập hợp các block. Grid chứa các block và các block này có thể thực thi độc lập với nhau. Grid cho phép lập trình viên tổ chức và điều phối các block để thực hiện tính toán trên GPU.

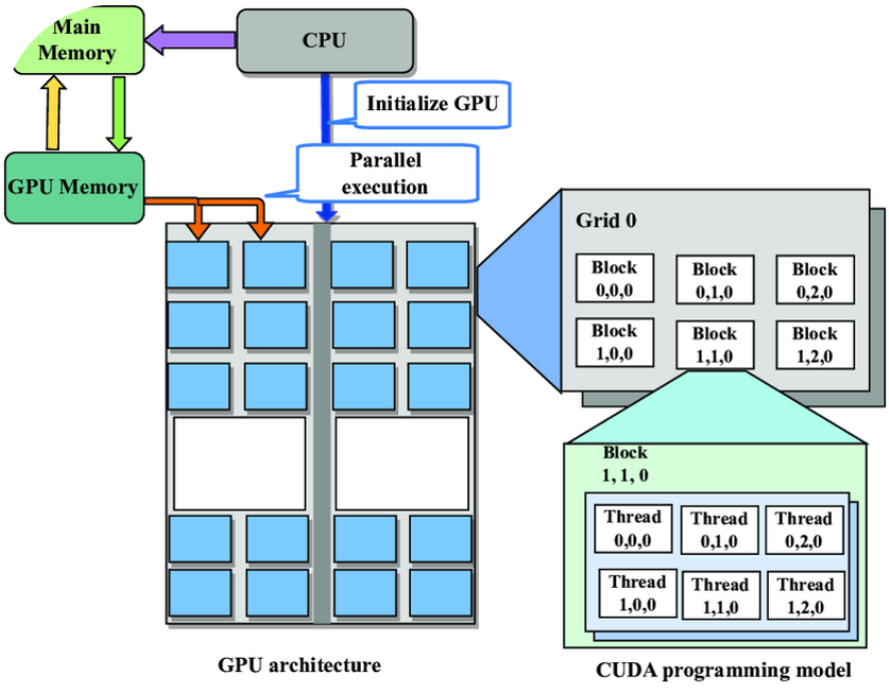


Kiến trúc của CUDA khi ánh xạ sang phần cứng GPU là mỗi Thread sẽ được thực thi bởi 1 lõi, mỗi Block sẽ được xử lý bởi 1 Streaming Multiprocesser và mỗi Grid sẽ được thực thi bởi 1 đơn vị GPU hoàn chỉnh.



Cách thức làm việc cơ bản của CUDA:

- Trong một chương trình CUDA tiêu biểu, dữ liệu được truyền từ bộ nhớ chính (main memory) đến bộ nhớ GPU, sau đó CPU gửi các chỉ thị đến GPU. GPU sau đó sắp xếp và thực thi kernel trên phần cứng song song có sẵn, cuối cùng kết quả được sao chép từ bộ nhớ GPU trở lại bộ nhớ CPU.



Hình 2.1 Sơ đồ cách thức hoạt động của CUDA

Cách thức hoạt động chi tiết của CUDA:

- Chuẩn bị dữ liệu: Dữ liệu cần được chuẩn bị trên CPU và sao chép sang bộ nhớ của GPU.

- Khởi tạo cấu hình grid và block: Người lập trình cần xác định cấu trúc grid và block cho việc thực thi hàm kernel trên GPU. Điều này bao gồm số lượng block và số lượng lõi xử lý CUDA trong mỗi block.
- Thực thi hàm kernel: Hàm kernel được gọi từ CPU và thực thi trên GPU, chạy song song trên các lõi xử lý CUDA trong các block của grid.
- Sao chép kết quả về CPU: Kết quả tính toán từ GPU cần được sao chép từ bộ nhớ của GPU sang CPU để xử lý hoặc hiển thị.
- Giải phóng bộ nhớ: Khi việc tính toán hoàn thành, bộ nhớ trên GPU cần được giải phóng để tránh lãng phí tài nguyên.
-

2.2.4 Cấu trúc chương trình CUDA

	Executed on	Only callable from
`__device__ float DeviceFunc()`	device	device
`__global__ void KernelFunc()`	device	host
`__host__ float HostFunc()`	host	host

Cấu trúc cơ bản của CUDA:

- Hàm Kernel: Đây là các hàm được thực thi trên GPU. Chúng được đánh dấu bằng từ khóa __global__ và chạy song song trên nhiều lõi xử lý CUDA.
- Hàm Host: Đây là các hàm được thực thi trên CPU, được sử dụng để chuẩn bị dữ liệu, gọi các hàm kernel và quản lý bộ nhớ trên GPU.
- Grid và Block: Grid là tập hợp các block, mỗi block chứa một nhóm các lõi xử lý CUDA. Cấu trúc grid và block quyết định cách các hàm kernel được phân phối và thực thi trên GPU.

2.2.5 Các hạn chế của CUDA

Truyền Dữ Liệu:

- Việc quản lý và truyền dữ liệu giữa bộ nhớ chủ và GPU đòi hỏi sự cẩn thận để tránh tình trạng chậm trễ.

Quản Lý Bộ Nhớ:

- CUDA áp dụng mô hình bộ nhớ có cấp độ, từ bộ nhớ toàn cục đến bộ nhớ chia sẻ và bộ nhớ địa phương.

Đồng Bộ Hóa và Hiệu Suất:

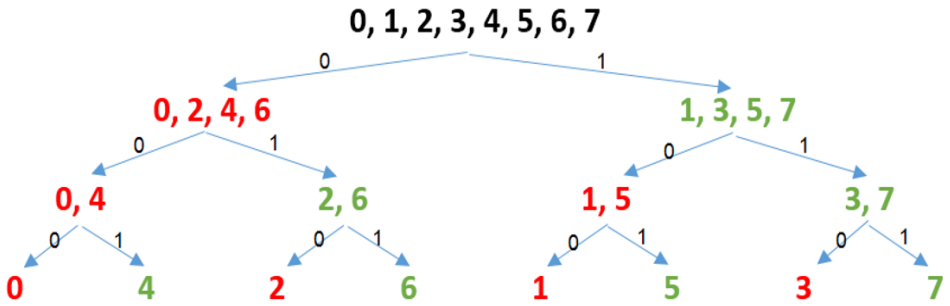
- Việc quản lý đồng bộ có thể làm giảm hiệu suất, vì vậy cần cân nhắc kỹ lưỡng khi sử dụng các yếu tố đồng bộ.

CHƯƠNG 3. SONG SONG HÓA

3.1 Ý tưởng

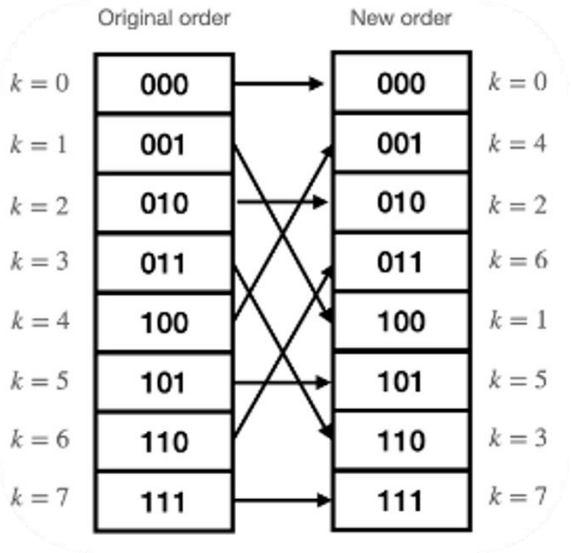
Thuật toán FFT nguyên bản là một phương pháp tối ưu hóa để thực hiện biến đổi Fourier trên các tín hiệu có độ dài là lũy thừa của 2 và hoạt động theo nguyên tắc “chia để trị". Thuật toán chia dãy tín hiệu đầu vào thành các dãy con liên tiếp theo thứ tự chẵn lẻ của từng phần tử đến khi nào dãy còn chỉ còn 1 phần tử. Sau đó thuật toán tiến hành biến đổi Fourier trên từng 1 phần tử cuối và tính ngược lại lên các dãy tín hiệu mẹ bên trên theo quy tắc cánh bướm của công thức FFT theo hình thức đệ quy.

- Tín hiệu đầu vào được phân tách thành hai tín hiệu con: một chứa các thành phần ở vị trí chẵn và một chứa các thành phần ở vị trí lẻ.
- Các phần tử ở vị trí chẵn sẽ được xử lý riêng lẻ và các phần tử ở vị trí lẻ sẽ được xử lý riêng lẻ.



Hình 3.1 Thuật toán phân tách chẵn lẻ

Từ hình ảnh 3.1 cho thuật toán FFT ta có thể thấy thuật toán có một đặc điểm; đó là dãy con 1 phần từ cuối cùng khi gom lại sẽ được một dãy số có quan hệ đảo trật tự bit với dãy con đầu vào. Cụ thể, phần tử thứ i của tín hiệu đầu vào sẽ là phần thứ j của dãy con đã gom cuối cùng, với j là đảo trật tự bit của i.



Hình 3.2 Thuật toán đảo bit

Hình 3.2 trình bày chi tiết hơn về ý nghĩa của phép đảo trật tự bit hình thành mối quan hệ giữa đầu vào và dãy con 1 phần tử cuối cùng. Từ đây ta có thể khử đệ quy quá trình tính toán bằng cách bắt đầu ngay từ các dãy con cuối cùng mà không cần chia nhỏ dãy tín hiệu ban đầu nữa.

Thuật toán FFT sử dụng đảo trật tự bit này còn có tên gọi khác là thuật toán Cooley–Tukey giúp giảm độ phức tạp tính toán so với FFT thông thường, đặc biệt là khi xử lý các tín hiệu có kích thước lớn. Điều này làm cho nó trở thành một lựa chọn hiệu quả trong nhiều ứng dụng thực tế.

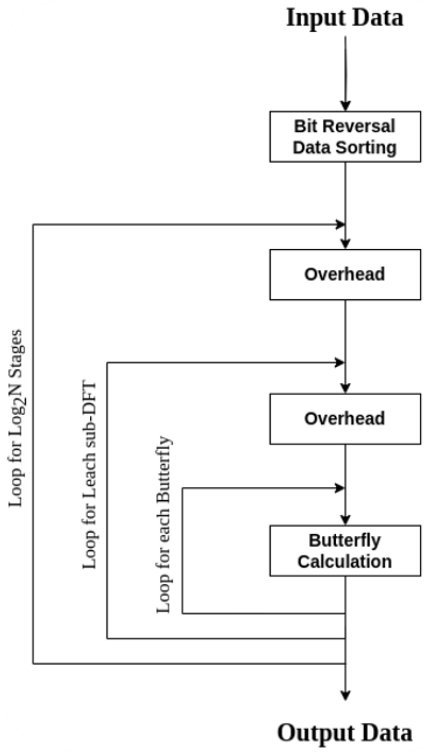
3.2 Triển khai thuật toán

3.2.1 Lưu đồ thuật toán

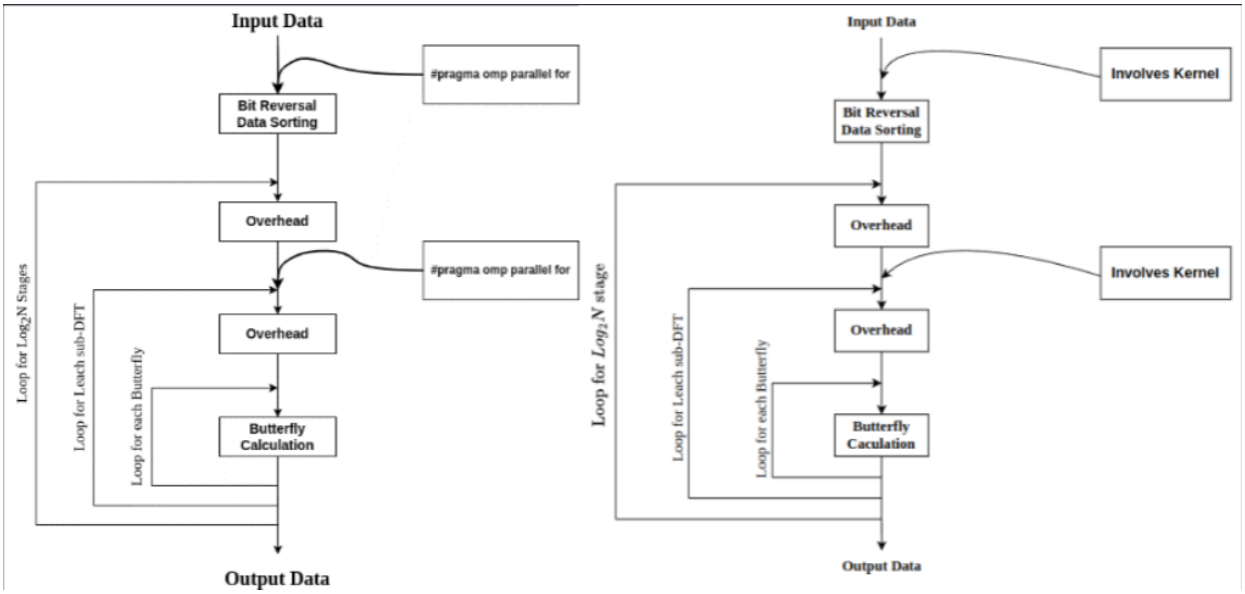
Từ ý tưởng đã trình bày bên trên, ta có thể trình bày thuật toán theo 4 bước sau:

- Bước 1: Sắp xếp dữ liệu đầu vào theo phép đảo trật tự bit.
- Bước 2: Kiểm tra xem đã đi hết các tầng của thuật toán FFT theo chiều đi lên chưa, chưa thì đi lên, đúng thì kết thúc.
- Bước 3: Kiểm tra đã biến đổi hết cặp cánh bướm của từng tầng chưa, chưa thì biến đổi tiếp, đúng thì quay lên bước 2.
- Bước 4: Biến đổi từng cặp dữ liệu thành phần theo công thức cánh bướm của từng cặp sau đó quay lên bước 3.

Hình 3.3, 3.4 thể hiện lưu đồ thuật toán tuần tự và lưu đồ thuật toán khi áp dụng thêm song song hóa bằng OpenMP và CUDA.



Hình 3.3 Lưu đồ thuật toán tuần tự



Hình 3.4 Lưu đồ thuật toán trên OpenMP vs CUDA

3.2.2 Các chiến lược bổ sung khác:

- Sử dụng một struct số ảo gồm 2 phần tử double tượng trưng cho số ảo.
- Xây dựng lại các hàm cộng, đảo dấu, nhân là các phương thức để tính toán thông qua struct số ảo.
- Tính toán số block tỉ lệ theo kích thước đầu vào và số core được cài đặt.

CHƯƠNG 4. KIỂM THỬ VÀ ĐÁNH GIÁ

4.1 Thông số phần cứng

Dưới đây là thông số phần cứng của CPU Intel Core i5-6200U mà bọn em sử dụng để thực hiện và đánh giá thuật toán trên OpenMP:

```
~ lscpu
Architecture:            x86_64
  CPU op-mode(s):        32-bit, 64-bit
  Address sizes:         39 bits physical, 48 bits virtual
  Byte Order:            Little Endian
CPU(s):                  4
  On-line CPU(s) list:   0-3
Vendor ID:               GenuineIntel
  Model name:            Intel(R) Core(TM) i5-6200U CPU @ 2.30GHz
    CPU family:          6
    Model:               78
    Thread(s) per core:  2
    Core(s) per socket:  2
    Socket(s):           1
    Stepping:            3
    CPU max MHz:         2800,0000
    CPU min MHz:         400,0000
    BogoMIPS:            4800.00
```

Số nhân/ luồng	2 nhân 4 luồng
Tiến trình sản xuất	Intel 5
Xung nhịp cơ bản	2.3 GHz
Xung nhịp tối đa	2.8 GHz
Loại bộ nhớ	DDR4-2133,LPDDR3-1866, DDR3L1600
Bộ nhớ đệm	3 MB Intel Smart Cache
TDP	15 W
Xung nhịp đồ họa tối đa	300 MHz

Dưới đây là thông số phần cứng của GPU NVIDIA GeForce 930M mà bọn em sử dụng để thực hiện và đánh giá thuật toán trên CUDA:

```
Detected 1 CUDA Capable device(s)

Device 0: "NVIDIA GeForce 930M"
  CUDA Driver Version / Runtime Version          12.0 / 12.2
  CUDA Capability Major/Minor version number:    5.0
  Total amount of global memory:                 2003 MBytes (2099970048 bytes)
  (003) Multiprocessors, (128) CUDA Cores/MP:    384 CUDA Cores
  GPU Max Clock rate:                            941 MHz (0.94 GHz)
  Memory Clock rate:                             900 Mhz
  Memory Bus Width:                              64-bit
  L2 Cache Size:                                 1048576 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(65536), 2D=(65536, 65536), 3D=(4096, 4096, 4096)
  Maximum Layered 1D Texture Size, (num) layers  1D=(16384), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(16384, 16384), 2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total shared memory per multiprocessor:        65536 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  2048
  Maximum number of threads per block:           1024
```

CUDA Driver Version	12.0
CUDA Runtime Version	12.2
Kiến trúc	Maxwell
Số lượng nhân CUDA	384
Tần số cơ bản	928 MHz
Tần số boost	941 Mhz
Loại bộ nhớ	GDDR5
Kích thước bộ nhớ	2 GB
Kết nối	PCIe 3.0x8

4.2 Kết quả

4.2.1 Phương pháp đánh giá

Bọn em đánh giá thuật toán của mình thông qua 3 phương thức: Correct, Speedup và Efficiency. Với Correct(Độ chính xác) bọn em tính toán tỉ lệ sai số trên dữ liệu đầu ra giữa thuật toán song song và thuật toán tuần tự, nếu sai số xấp sỉ 0 thì kết quả là chính xác. Còn Speedup và Efficiency được bọn em tính theo công thức bên dưới:

$$speed\_up = \frac{time\_for\_serial}{time\_for\_parallel}$$

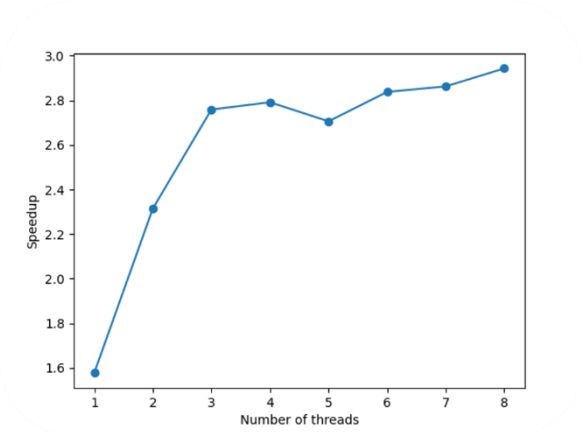
$$efficiency = \frac{speed\_up}{number\_of\_threads}$$

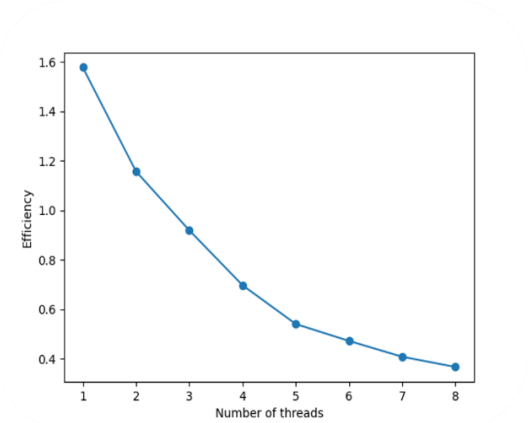
4.2.2 Kết quả trên OpenMP

Dưới đây là kết quả hiện thị của thuật toán khi thực hiện trên OpenMP. Bọn em sử dụng đầu vào là dãy tín hiệu ngẫu nhiên có độ dài 2^20 phần tử. Số luồng được bọn em cho chạy từ 1 đến 8 để đánh giá.

```
~/Desktop/Trung/AITI/LTSS/Code/openmp ./fftomp
N: 20
Number of inputs (2^N): 1.04858e+06
Range of threads: 8
(Recursive) Elapsed time in microseconds: 1973565 µs
(No recursive) Elapsed time in microseconds: 1302709 µs
Thread          Time          Speedup(vs non-recursive)    Speedup(vs recursive)    Correct
   1      1.248508e+06(µs)            1.043413                 1.580739              True
   2      8.119280e+05(µs)            1.604464                 2.430714              True
   3      7.592090e+05(µs)            1.715877                 2.599502              True
   4      7.282500e+05(µs)            1.788821                 2.710010              True
   5      7.528880e+05(µs)            1.730283                 2.621326              True
   6      7.215160e+05(µs)            1.805516                 2.735303              True
   7      7.179150e+05(µs)            1.814573                 2.749023              True
   8      6.824380e+05(µs)            1.908905                 2.891933              True
```

Dưới đây là kết quả về Speedup và Efficiency:





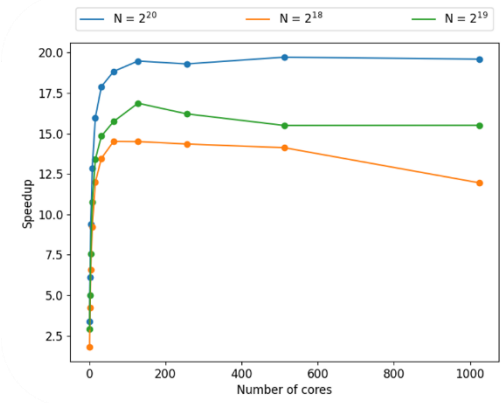
Từ biểu đồ Speedup và Efficiency, khi số luồng tăng lên thì Speedup cũng tăng lên nhưng không tăng theo cấp số với số luồng, còn Efficiency thì giảm dần

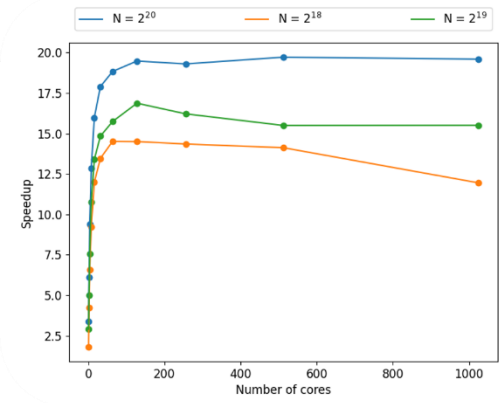
4.2.3 Kết quả trên CUDA

Dưới đây là kết quả hiện thị của thuật toán khi thực hiện trên CUDA. Bọn em sử dụng đầu vào là dãy tín hiệu ngẫu nhiên có độ dài 2^20 phần tử. Số core được bọn em cho chạy từ 1 đến 1024 để đánh giá.

```
~/Desktop/Trung/AITI/LTSS/Code/cuda ./FFT_forward
Enter min-max power of 2 (min -> max): 20 20
Write to file (y/n): n
Save to csv (y/n): n
Runing with N = 1048576
# of Cores    Parallel            Sequential       Speedup    Correct
       1     1.210047e+06(µs)   3.740216e+06(µs)   3.090968    True
       2     6.046290e+05(µs)   3.740216e+06(µs)   6.185969    True
       4     3.939030e+05(µs)   3.740216e+06(µs)   9.495272    True
       8     3.128810e+05(µs)   3.740216e+06(µs)   11.954117    True
      16     2.421170e+05(µs)   3.740216e+06(µs)   15.447969    True
      32     2.226540e+05(µs)   3.740216e+06(µs)   16.798333    True
      64     1.928100e+05(µs)   3.740216e+06(µs)   19.398454    True
     128     1.853400e+05(µs)   3.740216e+06(µs)   20.180296    True
     256     1.822530e+05(µs)   3.740216e+06(µs)   20.522109    True
     512     1.894680e+05(µs)   3.740216e+06(µs)   19.740621    True
    1024     1.978420e+05(µs)   3.740216e+06(µs)   18.905066    True
```

Dưới đây là kết quả về Speedup và Efficiency. Ở đây bọn em tiến hành đánh giá với số lượng phần tử đầu vào là 2^18, 2^19, 2^20. Số core cũng được cho chạy từ 1 đến 1024 :





So sánh biểu đồ Speedup và Effeiciency khi sử dụng CUDA và OpenMP, ta có thể thấy CUDA đã tăng tốc độ thực hiện thuật toán lên gấp rất nhiều lần. Với Speedup, khi số lượng dữ liệu càng lớn thì Speedup cũng lớn theo, bên cạnh đó khi số core tăng lên thì tỉ lệ tốc độ cũng tặng mạnh. Với Efficiency, số lượng dữ liệu dường như không ảnh hưởng quá nhiều đến Efficiency khi cả 3 trường hợp kích thước dữ liệu khác nhau thì Efficiency đều giảm nhanh về xấp xỉ 0.

CHƯƠNG 5. TỔNG KẾT

Trong dự án này, nhóm em đã thực hành triển khai thuật toán FFT (Fast Fourier Transform) trên OpenMP và CUDA, một công nghệ tính toán song song sử dụng CPU và GPU (Graphics Processing Unit). FFT là một thuật toán quan trọng trong xử lý tín hiệu và đại số tuyến tính, được sử dụng rộng rãi trong nhiều lĩnh vực như xử lý ảnh, âm thanh, và truyền thông.

Việc triển khai FFT trên OpenMP và CUDA giúp nhóm em tận dụng được khả năng xử lý song song của CPU và GPU, giảm thời gian tính toán và tăng hiệu suất so với việc thực hiện trên CPU. Qua quá trình thử nghiệm và đánh giá, nhóm em đã quan sát được sự gia tăng đáng kể về tốc độ xử lý so với phiên bản thực hiện trên CPU.

Mục tiêu của dự án là cung cấp một giải pháp hiệu quả cho việc thực hiện FFT, đặc biệt là đối với các đầu vào bài toán có kích thước lớn. Việc áp dụng CUDA đã mang lại những lợi ích rõ ràng về hiệu suất, đồng thời mở ra nhiều cơ hội ứng dụng trong các lĩnh vực như xử lý hình ảnh, xử lý âm thanh, và nhiều lĩnh vực khác.

Cuối cùng, nhóm em xin cảm ơn sự hướng dẫn và hỗ trợ từ thầy Phạm Doãn Tĩnh đã tận tâm hướng dẫn nhóm chúng em trong việc hoàn thiện bài tập lớn của môn học lập trình song song cũng như trong suốt quá trình học tập.

TÀI LIỆU THAM KHẢO

1. CUDA C Programming Guide (Hướng Dẫn Lập Trình CUDA C) - NVIDIA Corporation (2010)
2. CUDA by Example: An Introduction to General-Purpose GPU Programming - Jason Sanders & Edward Kandrot (2010)
3. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/
4. URL:https://www.cs.emory.edu/~cheung/Courses/355/Syllabus/94-CUDA/CUDA-thread.html
5. URL: https://viblo.asia/p/lap-trinh-song-song-bai-3-hello-world-cuda-c-0gdJzxbkVz5.
6. URL: https://viblo.asia/p/gioi-thieu-openmp-trong-c-phan-1-GrLZDAEBlk0.
7. Parallel FFT in Julia | Parallel Computing | Mathematics | MIT OpenCourseWare
8. URL: https://www.nersc.gov/assets/Uploads/IXPUG-2016-HelenHe.pdf