View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by SHAREOK repository

UNIVERSITY OF OKLAHOMA

GRADUATE COLLEGE

EMPIRICAL TRANSITION PROBABILITY INDEXING GENOME SEQUENCE ALIGNMENT BASED ON CUDA

A THESIS

SUBMITTED TO THE GRADUATE FACULTY

in partial fulfillment of the requirements for the

Degree of

MASTER OF SCIENCE IN TELECOMMUNICATIONS ENGINEERING

By

DONG HAN
Norman, Oklahoma
2016

EMPIRICAL TRANSITION PROBABILITY INDEXING
GENOME SEQUENCE ALIGNMENT BASED ON CUDA

A THESIS APPROVED FOR THE
SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING

BY

____
Dr. Samuel Cheng, Chair

____
Dr. Pramode Verma

____
Dr. Kam Wai Clifford Chan

© Copyright by DONG HAN 2016
All Rights Reserved

Table of Contents

List of Tables vi

List of Figures vii

Abstract viii

Chapter 1: Introduction 1

1.1 Motivation 1

1.2 Objectives 2

1.3 Organization of the Thesis 2

Chapter 2: GPU Architecture 3

2.1 General GPU Architecture 4

2.2 GPU Hardware 5

2.2.1 Fermi Architecture 6

2.2.2 Kepler Architecture 6

2.3 GPU Programming Languages 10

2.4 CUDA 12

2.4.1 CUDA Architecture 12

2.4.2 CUDA Programming Model 13

2.4.3 CUDA Memory Model 15

Chapter 3: Sequence Alignment 19

3.1 Bioinformatics 19

3.2 DNA Sequencing 20

3.3 Sequence Alignment Algorithms 22

3.3.1 Global Alignment 23

3.3.2 Local Alignment 24
Chapter 4: Transition Probability Indexing Based on CUDA 25
4.1 Transition Probability Indexing 25
4.1.1 Indexing 25
4.1.2 Indexing matching 26
4.2 Transition Probability Indexing CUDA System 27
4.3 CUDA Implementation 29
4.3.1 Preprocessing 29
4.3.1 CUDA Kernels 30
Chapter 5: Results and Conclusions 33
5.1 Environment 33
5.2 Results 33
5.2 Conclusions 35
References 36

List of Tables

Table 1 Comparison of Fermi and Kepler architectures 7

Table 2 Comparison of Sanger and NGS 21

Table 3 Results for sequential and parallel.. 35

List of Figures

Figure 1 Basic GPU Architecture 4

Figure 2 Kepler Memory Hierarchy 7

Figure 3 Fermi streaming multiprocessor architecture[9] 9

Figure 4 Kepler GK104 streaming multiprocessor (SMX) architecture[10] 10

Figure 5 Typical GPU execution 11

Figure 6 CUDA Programming Architecture 14

Figure 7 Memory Hierarchy[19] 16

Figure 8 Global and Local alignment 23

Figure 9 The transition diagram between nucleotides 27

Figure 10 Diagram of Transition Probability Indexing CUDA system 28

Figure 11 Basic kernel structure 31

Figure 12 The cost time running in C 34

Figure 13 The cost of kernel time running on CUDA 34

Abstract

After Deoxyribonucleic Acid (DNA) was discovered, finding the similarities in proteins became a fundamental procedure. In recent years, there has been a rapid development in alignment technologies. Alignment is the basic operation used to compare biological sequences and to determine the similarities that eventually result for structural, functional, or biological process relationships. These new technologies produce data in the order of numerous gigabyte-pairs per day. With the use of a Graphics Processing Unit (GPU), these data can be solved. We can utilize a GPU in computation as a massive parallel processor because the GPU consists of multiple pips. This new hardware creates new opportunities to study and improve current algorithms that are used for research in DNA alignment. In this thesis, we proposed a new algorithm to tackle this problem. We matched blocks of reference and target sequences based on the similarities between their empirical transition probabilities matrixes. The computations were conducted on an NVIDIA GTX 760, equipped with 2GB RAM, running Microsoft Windows 8.1 Professional. Our experimental results show robustness in nucleotide sequence alignment, and the parallelized transition probability indexing on a GPU achieves faster results than a former study of a proposed sequential method on a CPU[1].

Chapter 1: Introduction

With the rapid development of sequencing technologies, sequence alignment is crucial in any analysis of biological process relationships, because it helps to extract practical and even tertiary structure data from an organic compound sequence. The simplest way to find the relationships between two sequences would be by counting the number of identical and similar amino acids. However, the traditional technologies are too tedious to do the sequence alignment, and it is important to develop more effective sequence alignment techniques. Based on dynamic programming, the empirical transition probability indexing algorithm is one of the methods used to search for all possible alignments between two sequences to find the optimal local alignments. However, the empirical transition probability indexing algorithm is usually implemented with sequential calculations, and the computational complexity is proportional to the result of the lengths of the two sequences. Researchers often use high performance devices to reduce the computational complexity, but the drawbacks of this method are obvious. Sequencing alignments are very computation-intensive tasks, and expensive hardware is required to run these programs. Therefore, GPUs have been found to be good alternatives for solving sequencing alignments.

1.1 Motivation

In recent years, new technologies have evolved dramatically, and enormous new databases containing gigabytes of data are created every day. However, the sequence analyses are slow and numerous, and using the traditional alignment tools, such as BWA, BFAST, and BLAST, have proven to be too time consuming. There are many new techniques available, such as transition probability indexing. However, this novel

technique is usually realized with a sequential calculation, whereas a GPU can be used to supply a powerful platform and reduce the computational time.

Studying and analyzing the genetic make-up of each living being can be a way to understand ourselves. Thus, it is vital that these tools be further developed and improved. Parallel algorithms running on GPUs can often achieve speeds up to 100 times faster than similar CPU algorithms, and there are many existing applications, including those relating to physics simulations, signal processing, financial modeling, neural networks, and countless other fields[2].

1.2 Objectives

This thesis aims to implement the sequence alignment method and to improve the method by using the large multiprocessing capabilities provided by current GPUs. We present a new alignment method that uses empirical transition probability indexing to solve the alignment problem. Meanwhile, we are planning to do an analysis to determine whether or not GPUs can be adjusted to construct a quicker tool for this, since GPUs are a great tool for speeding up algorithms through enormous parallelism.

1.3 Organization of the Thesis

The rest of this thesis is organized as follows. Chapter 2 starts with the GPU architecture, including the GPU structure, the GK104 architecture, and the CUDA language. Chapter 3 presents a brief introduction to DNA sequence alignment and dynamic programming. Chapter 4 introduces the implementation of the empirical transition probability indexing on the GK104 with the CUDA language. Chapter 5 discusses the experimental results and draws some conclusions for this work.

Chapter 2: GPU Architecture

A GPU (Graphics Processing Unit) is a specialized processor for accelerating graphics representations in real time. Displaying graphics is a computationally heavy job, and implementing this in real time sets high requirements for the hardware. NVIDIA presented the application of the graphics processor for general purpose calculations that are traditionally treated by personal computers or workstations. Many more scientific applications have been accelerated by GPU. Almost all computational sellers started receiving this new technology since it has been given high-end services and improved industry principles. A large set of problems in molecular dynamics, physics simulations, and scientific computing have been tackled by mapping them onto a GPU[3].

Graphics chips designed by NVIDIA strained for a specific set of practicality, it serves an extremely parallel programming and computational environment. This was begun in the 1999-2000 timeframe, and domain scientists and computer researchers have started using GPUs for accelerating scientific applications and achieving efficient performance gains. This timeframe saw the advent of the movement referred to as GPGPU—General-Purpose computation on a GPU. GPU computing is the use of a GPU in conjunction with a CPU to accelerate general scientific and engineering applications. Now, the floating point performance of the GPU is higher than the performance of the CPU because the architecture of the GPU has been changed dramatically via improvement of the chip design and manufacturing technology[4].

Pioneered five years ago by NVIDIA, GPU computing has quickly become an industry standard, enjoyed by millions of users worldwide and adopted by virtually all

computing vendors[5]. From a user's point of view, applications simply run significantly faster.

2.1 General GPU Architecture

GPU architecture is built with a specialized circuit that can accelerate the output image in a frame buffer intended for output to the display. GPUs are very efficient at manipulating computer graphics and are generally more effective than general purpose CPUs for algorithms in which large blocks of data are processed in parallel. The basic GPU architecture is shown in Figure 1.

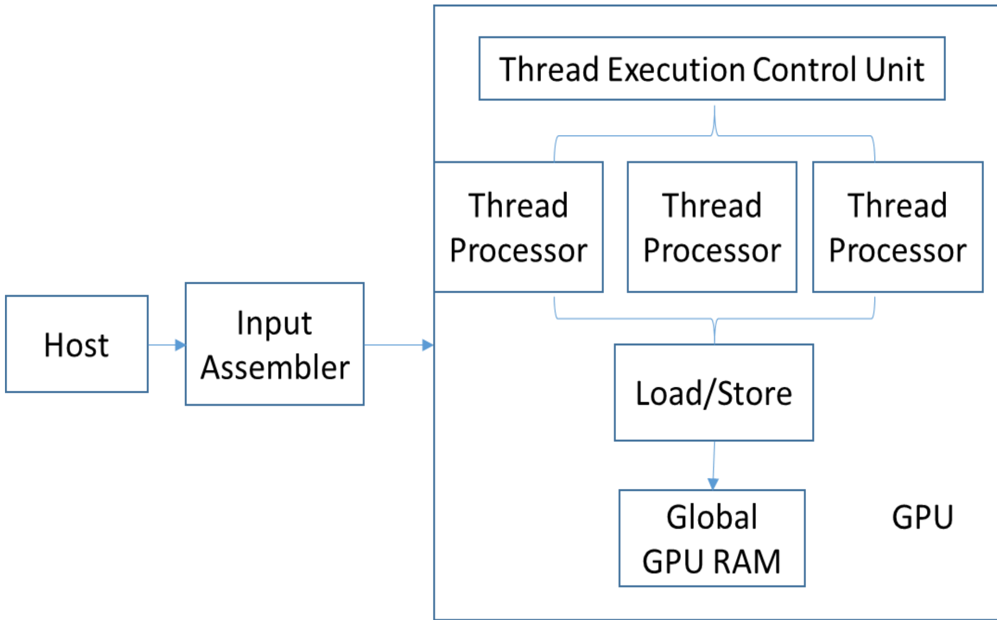

Figure 1 Basic GPU Architecture

The NVIDIA GPU architecture is constructed around a scalable array of multithreaded Streaming Multiprocessors (SMs). The increased flexibility and high computing capabilities of GPUs have led a new research field that explores the performance of GPUs for general purpose computation. GPU is mainly used for extremely parallel operations, whereas CPUs execute programs serially[6]. For this

reason, GPUs have several parallel execution units and better semiconductor device counts as compared to CPUs, which have few execution counts against greater clock speeds. The technology of the GPU has evolved into a multiple programming pipeline, and the graphics pipeline has been substituted by the user programmable vertex and pixel shaders. A programmer can now implement custom transformations, lighting, and texturing algorithms by writing programs called shaders[7]. This pipeline proceeds from host interface to memory interface. Host interface is followed by vertex processing later sets up the triangle finally pixel processing is finished before memory interface.

2.2 GPU Hardware

The NVIDIA GPU consists of Streaming Multiprocessors (SMs), each of which consists of many Streaming Processors (SPs). A multiprocessor device is designed to execute hundreds of thousands of threads at the same time. To manage such a large quantity of threads, it employs a singular design called SIMT (Single-Instruction, Multiple-Thread). The instructions are pipelined to leverage instruction-level correspondence within one thread, as well as thread-level parallelism extensively through simultaneous hardware multithreading. All instructions are issued in order and there is no branch prediction and no speculative execution. Current GPUs consist of a high number of section processors with high memory bandwidth. In some ways, the architecture of a current GPU is similar to a multiple processor that achieves higher parallel code performance for rasterization applications. This is in contrast with multi-core CPUs, which include best single-thread performing cores. GPUs are primarily optimized for 2D arrays. Below is a high-level abstraction for CPU and GPU memory hierarchies. The GPUs (on the right) write to a high-bandwidth, high-latency video

memory using small, write-through caches. Caches on the GPU are shared by a large number of fragment processors (FPs). Differences in the architecture between CPUs and GPUs indicate that the code must be optimized differently for the GPU to achieve higher performance[8].

2.2.1 Fermi Architecture

The first Fermi based GPU features up to 512 CUDA cores, which are organized in 16 SMs of 32 cores each. To build it, NVIDIA took all they learned from the two prior processors and all the applications that were written for them. Fermi’s 16 SMs are positioned around a common L2 cache. Each SM is a vertical rectangular strip that contains scheduler and dispatch, execution units, and register file and L1 cache. Fermi supports concurrent kernel execution, in which different kernels of the same application context can execute on the GPU at the same time, thus fully utilizing GPU capacity. Figure 3 shows a diagram of the general architecture[9].

2.2.2 Kepler Architecture

NVIDIA’s Kepler architecture is built on the foundation of NVIDIA’s Fermi GPU architecture, initially established in 2010. The biggest modification with Kepler is that there is no longer a shader frequency. There is simply GPU frequency. This is often a compromise to create more space for more CUDA cores in the Kepler architecture, which can be clocked even higher than before. Each Stream Multiprocessor contains 96 CUDA cores, in contrast to the 32-48 that Fermi had.

The change in layout of the CUDA cores and the clock frequency is most likely a way for NVIDIA to get more performance from the circuit. The 1536 CUDA core number has been quite interesting and it is basically three time more than what we had a

chance to see on the Fermi based GTX 680. This is without the shader frequency, which reduces the efficiency of each core. If you add every GPC up, you'll get a total of 4 Raster Units, 8 Geometry Units, 32 ROPs, and 128 Texture Units. This doesn't have to be bad, since the raster units in Kepler can be more efficient than those in Fermi. Figure 4 shows the general architecture of Kepler.

Kepler's memory hierarchy is organized similarly to Fermi's. The Kepler architecture supports a unified memory request path for loads and stores, with an L1 cache per SMX multiprocessor[10]. Kepler GK104 also enables compiler-directed use of an additional new cache for read-only data, as described in Figure 2.

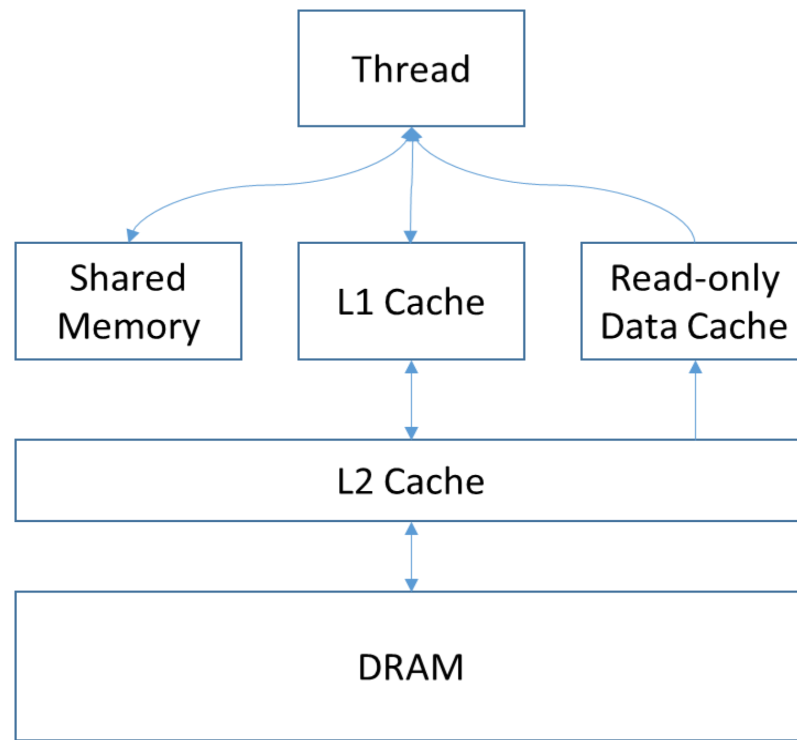

Figure 2 Kepler Memory Hierarchy

Table 1 Comparison of Fermi and Kepler architectures

Model	Fermi(GTX 580)	Kepler(GTX 680)
Node	40nm	28nm

CUDA cores	512	1536
Streaming Multiprocessors(SM)	16	8
Cores per SM	32	192
Warp Schedulers per SM	2	4
L1 Cache per SM	64KB	64KB
L2 Cache	768KB	512KB

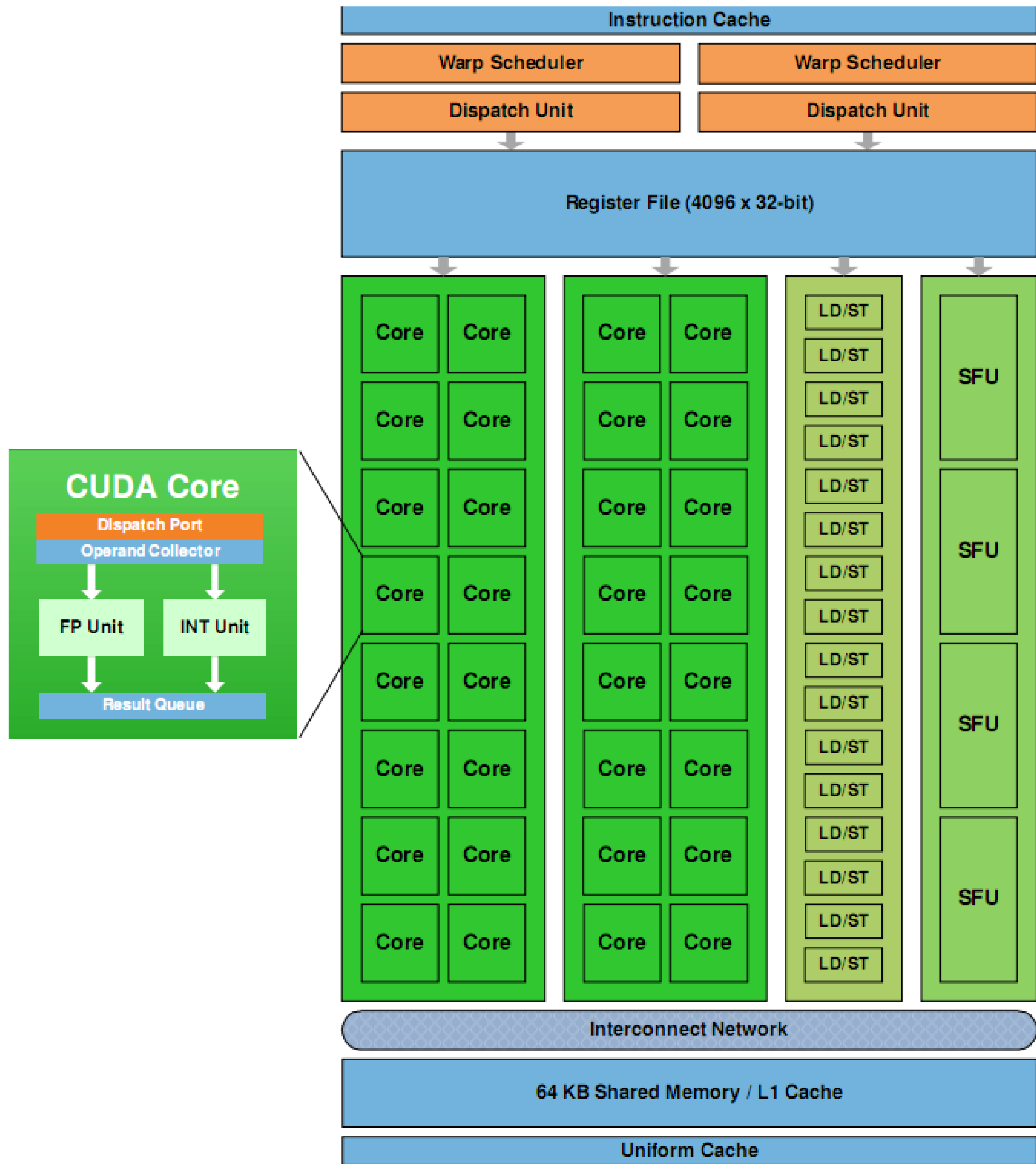

Figure 3 Fermi streaming multiprocessor architecture[9]

Figure 4 Kepler GK104 streaming multiprocessor (SMX) architecture[10]

2.3 GPU Programming Languages

A GPU application has two parts: the CPU code and the GPU code. The CPU code, which its name implies, runs on the CPU and is that part of the application answerable for initializing the device, allocating memory, copying data from and to the GPU memory and conjointly launching the GPU code on the device. The software engineer specifies through the CPU code what number of threads ought to be launched on the GPU and how they should be organized[11].

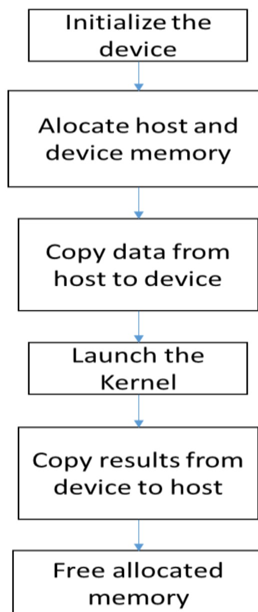

Figure 5 Typical GPU execution

Figure 5 illustrates the most common cycle in a GPU application. The entire process, in a GPU application, revolves around a main cycle that, when initializing the device: 1) copies data to be processed into the GPU, 2) executes the kernel to process the data, and 3) copies the results from the GPU memory back to the host's memory. Once there is no more data to process by the GPU, the memory is freed.

There are several programming languages suitable for GPU programming. CUDA is an extended C language environment that unlocks the processing power of GPUs to solve the most complex computational problems. OpenCL is a language for GPU based on the C language and proposed by Apple in cooperation with others[12]. BrookGPU is the Stanford University graphics group's stream-oriented language intended for stream processing that was developed for specialized high performance stream machines[13]. AMD Stream Computing SDK was their first production of a

GPU language, which was run on Windows XP. The SDK, which includes Brook+, is also open-sourced for stream computing[14].

2.4 CUDA

The most popular framework for programming NVIDIA GPUs is CUDA[15], which presents the programmer with the illusion of a virtually unlimited set of threads.

In 2006, NVIDIA broadcasted CUDA. CUDA is a general purpose parallel computing architecture that permits programmers to create enormously parallel code that runs on NVIDIA GPUs. CUDA is well-suited for programming on multiple threaded multi-core GPUs. Previously, this was only possible to realize through the use of graphic APIs. CUDA could be a giant step forward for making the programming of the GPU easier. The API and the general ideas behind it are described in the CUDA C Programming Guide[16].

2.4.1 CUDA Architecture

CUDA serves the efficient programming environment for a number of cores of graphics processors to run in parallel and provides a high number of computations. Applications that run on the CUDA architecture can take advantage of an installed base of over one hundred million CUDA-enabled GPUs in desktop and notebook computers, professional workstations, and supercomputer clusters. CUDA architecture splits the device into grids, blocks, and threads in a hierarchical structure. Since there are a number of threads in one block and a number of blocks in one grid and a number of grids in one GPU, the parallelism can be achieved by this hierarchical architecture[17].

2.4.2 CUDA Programming Model

NVIDIA CUDA is a general purpose, scalable, parallelized programming model for highly parallel processing applications. It enables a dramatic increase in computing performance by harnessing the power of the graphics processing unit (GPU). CUDA's hierarchy of threads maps to a hierarchy of processors on the GPU. A GPU executes one or more kernel grids. A GPU consists of multiprocessors that execute one or more thread blocks. CUDA cores, the processing elements within a multiprocessor, execute threads in groups of 32, called warps. The CUDA programming model enables the programmer to expose substantial fine-grained parallelism sufficient for utilizing massively multithreaded GPUs, while at the same time providing scalability across the broad spectrum of physical parallelism available in the range of GPU devices[18].

A CUDA program consists of at least 2 parts: a main program that runs on the CPU (host code) that supplies and retrieves data from the GPU, and a kernel (device code) that runs on the GPU. The GPU does not usually have access to the memory on the host computer, but it has its own device memory that the host code can transfer data to. Figure 6 shows the CUDA programming structure.

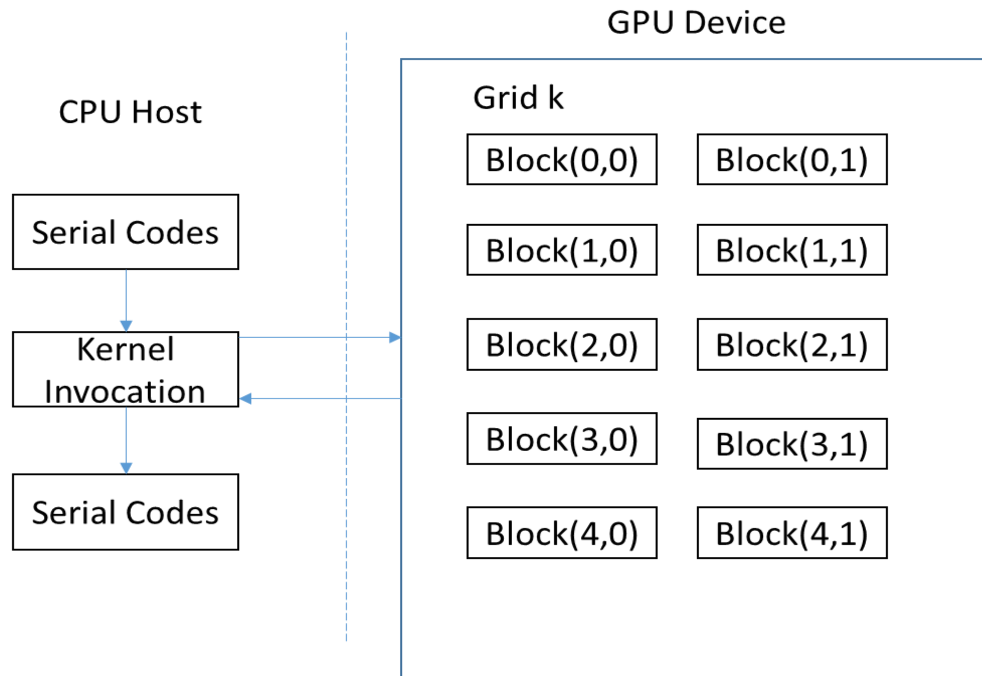

Figure 6 CUDA Programming Architecture

Kernels are similar to normal functions in C but they have a given prefix: "_global_". Executing a kernel program on the GPU is a relatively time-consuming operation. By default, kernel calls are asynchronous, which means that a CUDA program will proceed with consecutive instructions before the kernel really completes execution. However, most CUDA programs launch a kernel and then immediately transfer back the result.

When calling a kernel function, it will run multiple times in parallel. And the arrangement of the dimension for CUDA is determine the exact times running in kernel. Inside the kernel functions you have access to a few extra structs:

- gridDim: the x and y dimensions of the grid. Grids cannot be tridimensional, and therefore z is always 1.
- blockDim: the x, y and z dimensions of the block (3-dimensional vector).

- blockIdx: the index of the block within the gridDim (3-dimensional vector).
- threadIdx: the index of the thread within the block (3-dimensional vector).

2.4.3 CUDA Memory Model

In the CUDA parallel programming model various memory spaces exist[9]. The GPU has several different types of memory available, and they all have different access times and size limitations, as on a CPU. However, unlike when coding for the CPU one has more control of what memory type to use in the code. The 3 most important types of memory are register, shared memory, and global memory, as shown in Figure 7.

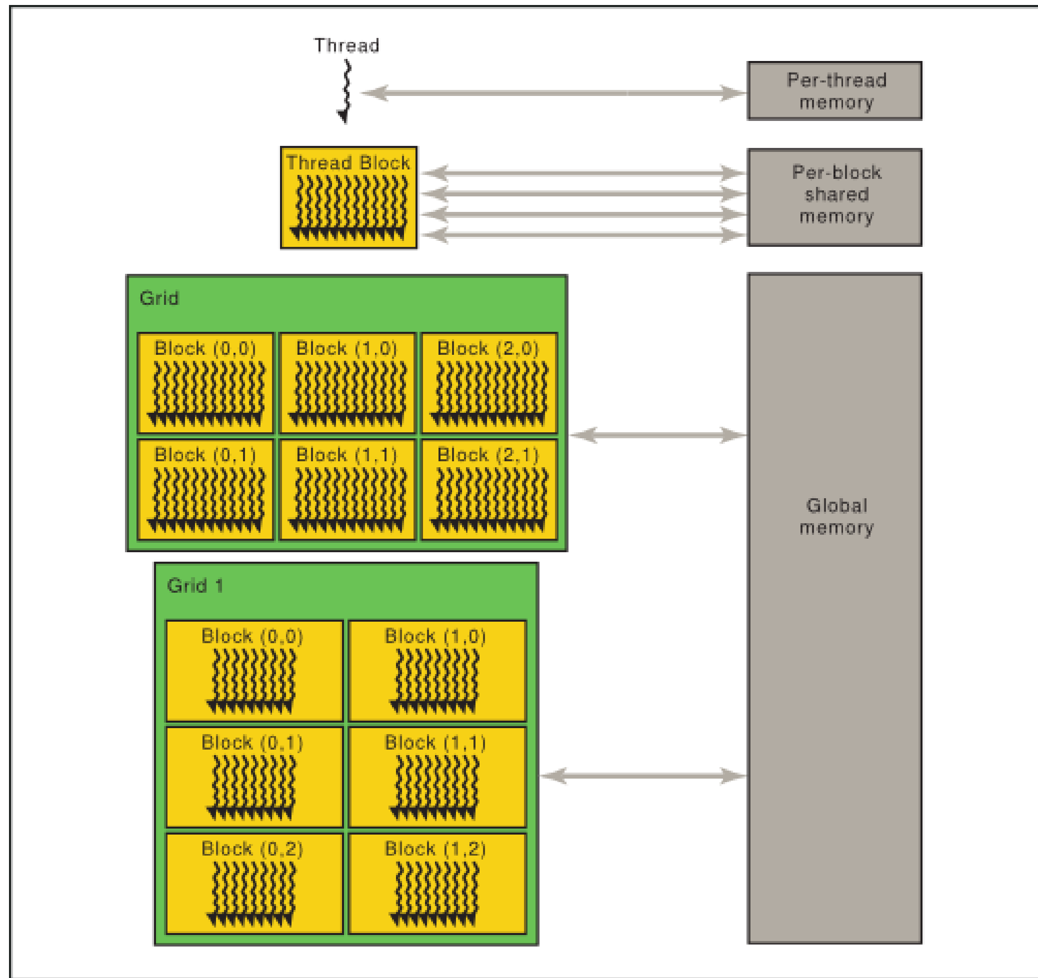

Figure 7 Memory Hierarchy[19]

Threads might access data from multiple memory areas throughout their execution. Every thread includes private memory. Each thread block has a shared memory visible to all or any threads of the block and with the equivalent lifetime as the block. Finally, all of the threads can access the same global memory. There are extra memory spaces accessible by all threads: the constant, texture, and surface memory spaces. Constant and texture memory are read-only; surface memory is readable and writable. The on-chip memories, such as registers and shared memory, can be accessed at very high speeds. The registers are allocated to the individual threads and have a lifetime of thread execution.

The register is built in each thread. Generally, accessing a register consumes zero extra clock cycles per instruction. However, delays might occur due to register memory bank conflicts and register read-after-write dependencies.

Shared memory is memory shared by all of the threads in an exceedingly block and resides on-chip within the SM. Because it is on-chip, shared memory have much higher bandwidth and lower latency than global or local memory, provided there are no bank conflicts between the threads. Threads inside the same block can only access-shared memory. Shared memory is often used for exchanging information between threads and to store values needed by all the threads in a block. Access to shared memory is slower than the register.

Global memory is the device memory. All blocks can share data via GPU global memory. This memory type is the slowest type of GPU memory. Global memory is cached when running on hardware with compute capability 2.0 or higher. The data can be transferred directly from the host memory to the global memory. The size of the global memory is much larger than the size of the register and shared memory. The lifetime of global memory is from CPU using cudaMalloc until cudaFree.

The local memory is part of the global device memory. Local memory is ‘local’ to an individual thread, that is, it is accessible only by the thread that declares it. Variables which kernel function using, are often placed in the register. However, if the variable were too big to fit in, the compiler would place it in local memory. Local memory can reside in the registers in the SM on the hardware or in the global memory. The compiler decides where to place the local memory at compile time. The local

memory in the registers is the fastest possible memory on the GPU. Local memory is mostly used as working memory for different kind of threads.

To make the best use of the GPU, it is important to use the correct type of memory for every variable, and to reduce the use of global memory as much as possible. The most common way to avoid this problem is to place the variables that more than 1 thread will use into the shared memory early, and then sync all of the threads.

Chapter 3: Sequence Alignment

Sequence alignment is a way of arranging two sequences to identify regions of similarity[20]. Sequence alignment attempts to find similarities between two sequences in protein or nucleotide databases, which can help researchers find the functional, structural, or evolutionary relationships between the sequences. Sequence alignment algorithms are mostly used to align two sequences at one time. Aligning two sequences can be done with recursively replacing, inserting, or removing an element. The quality of the alignment is represented as an associated score. Sequence alignment algorithms find the optimal alignment that maximizes the score.

3.1 Bioinformatics

Bioinformatics has been defined as a means of analyzing, comparing, graphically displaying, modeling, storing, systemizing, searching, and ultimately distributing biological information, which includes sequences, structures and functions. Due to the rapidly increasing quantities of biological data, high computational resources are required in research to reduce the time of analysis. Bioinformatics is a serious attempt to understand what it means when we say that genes code for physiological traits, like intelligence, brown hair, or susceptibility to cancer. Bioinformatics strives to further our knowledge of biological systems and the capacity to interpret biological processes for utilization in different applications. This is evidenced by its development and the use of computationally intensive techniques. That said, common activities include:

- Mapping and analyzing DNA, RNA, Protein, Amino Acid, and Lipid sequences.
- Sequence Alignment and Analysis.
- Creation and Visualization of 3-D structure models of biological molecules of significance, e.g., proteins.
- Genome Annotations.

3.2 DNA Sequencing

DNA sequencing is the process of determining the precise order of the nucleotide bases within a DNA molecule. It includes any method or technology that is used to determine the order of the four bases: Adenine (A), Guanine (G), Cytosine (C), and Thymine (T). The advent of rapid DNA sequencing methods has greatly accelerated biological and medical research and discovery.

Knowledge of DNA sequences has become indispensable for basic biological research, and for numerous applied fields. The rapid speed of sequencing attained with modern DNA sequencing technology has been instrumental in the sequencing of complete DNA sequences, or genomes of numerous types and species of life, including the human genome and other complete DNA sequences of many animal, plant, and microbial species.

The first DNA sequences were obtained in the early 1970s by academic researchers using laborious methods based on two-dimensional chromatography. Following the development of fluorescence-based sequencing methods with automated analysis, several notable advances in DNA sequencing were made during the 1970s. Frederick Sanger developed rapid DNA sequencing methods at the MRC Centre,

Cambridge, UK, and published a method in 1977[21]. Walter Gilbert and Allan Maxam at Harvard also developed sequencing methods, including one for DNA sequencing by chemical degradation[22].

DNA sequencing may be used to determine the sequence of individual genes, larger genetic regions, full chromosomes, or entire genomes. Depending on the methods used, sequencing may provide the order of nucleotides in DNA or RNA isolated from the cells of animals, plants, bacteria, or virtually any other source of genetic information. The resulting sequences may be used by researchers in molecular biology or genetics to further scientific progress or may be used by medical personnel to make treatment decisions or aid in genetic counselling.

Sanger sequencing was the most popular technology before the 21st century[23]. Since the beginning of the 21st century, Next-generation sequencing (NGS) has become widely popular. Table 2 shows the comparison of these two techniques.

Table 2 Comparison of Sanger and NGS

	Sanger	NGS
Sequencing Samples	Clones, PCR	DNA Libraries
Sample Tracking	Many samples in 96, 384 well plates	Few
Preparation steps	Few, Sequencing reactions clean up	Many, Complex procedures
Data Collection	Samples in plates 96, 384	Samples on slides 1-16
Data	One read/sample	Thousands and Millions of reads/Samples

3.3 Sequence Alignment Algorithms

Sequence alignment is the procedure of comparing two (pair-wise alignment) or more sequences by searching for a series of individual characters or patterns that are in the same order in the sequences. This is useful because sequences with high similarity have a good chance to be related according to functional, structural, or evolutionary aspects.

The standard method of representing strings—for example, a consecutive chain of characters—is terribly slow for string matching issues. Solving the string matching issue using strings involves checking one of the strings (the reference string) for the locations where the first character of the second string (the target string) appears and in those locations apply the same method for the second character and so on until all characters of the second string have been used. But this type of method can be less efficient when the data size becomes bigger. Pairwise sequence alignment can be generally classified as global alignment and local alignment. Given two sequences, A = {ACTAGC} and B = {TATCTGCCGT}, it is possible to align them globally or locally, as shown in Figure 8.

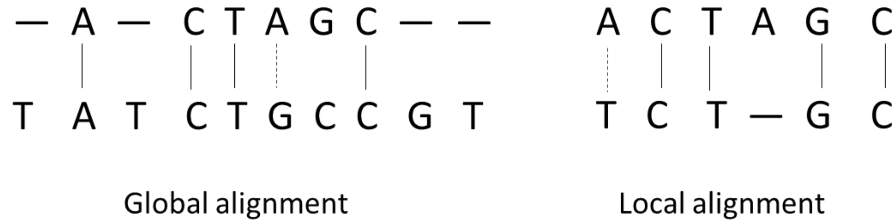

Figure 8 Global and Local alignment

3.3.1 Global Alignment

When aligning sequences globally, the optimal alignment is the alignment in which the overall number of matches is maximal. All of the characters in each sequence participate in the alignment. This method is beneficial when comparing closely related sequences. The Needleman-Wunsch algorithm, proposed in 1970 by Saul B. Needleman and Christian D. Wunsch, was one of the first applications of dynamic programming to biological sequence comparison[24]. This algorithm uses the sequence representation for DNA and builds a matrix of n by m dimensions, where n is the size of the reference sequence and m is the size of the target sequence. This algorithm fills the matrix with the following functions:

$$F_{i,0} = -iG, \qquad 0 \leq i \leq m$$

$$F_{0,j} = -jG, \qquad 0 \leq j \leq n$$

$$F_{i,j} = max\begin{cases} F_{i-1,j-1} + s(A_i, B_j) \\ F_{i-1,j} - G \\ F_{i,j-1} - G \end{cases} ,1 \leq m, 1 \leq n$$

Where F is the $m \times n$ matrix, A and B are the 2 sequences, m and n are the lengths of A and B, G is the gap penalty function, and S is the similarity score function, defined as:

$$s(A_i, B_j) = \begin{cases} > 0, A_i = B_j \; Match \\ < 0, A_i \neq B_j, Mismatch \end{cases}$$

The Time complexity of Needleman-Wunsch algorithm is $O_{(nm)}$.

3.3.2 Local Alignment

Local alignment is used to seek out related regions in two sequences. This method is much more flexible than the global alignment method. Related regions that appear in different orders can still be identified as being related, whereas this is often not possible with the global alignment method. One of the most well-known dynamic programming algorithms for local DNA alignment is the Smith-Waterman algorithm[25]. This algorithm works by filling the sequence alignment matrix with scores. One sequence is placed at the top of the sequence alignment matrix, and another is place at the left side of the matrix. The cells in the first row and the first column are filled with 0 for initialization of the sequence alignment matrix. The Smith-Waterman algorithm fills the matrix with following functions:

$$F_{i,j} = 0, \begin{cases} 0 \leq i \leq m \\ 0 \leq j \leq n \end{cases}$$

$$F_{i,j} = max \begin{cases} F_{i-1,j-1} + s(A_i, B_j) \\ F_{i-1,j} - G \\ F_{i,j-1} - G \\ 0 \end{cases} , 1 \leq m, 1 \leq n$$

Where F is the $m \times n$ matrix, A and B are the 2 sequences, m and n are the lengths of A and B, G is the gap penalty function, and S is the score matrix described above.

The Time complexity of Smith-Waterman algorithm is $O_{(nm)}$.

Chapter 4: Transition Probability Indexing Based on CUDA

In this chapter, we describe the Transition Probability Indexing Algorithm for sequence alignment, and we focus on the implementation of the parallelized algorithm adapted to the GPU.

4.1 Transition Probability Indexing

In this section, we present our genome indexing and alignment framework in detail. We will introduce the indexing: index matching. In this report, we refer to “reference sequence” as the base-line sequence and try to align a “target sequence” against the base-line sequence. After introducing the idea of our algorithms, we will describe the different between a C-implemented program and a CUDA-implemented program.

4.1.1 Indexing

Compared with current genome indexing methods, our indexing process provides a faster and light-weight alternative for index generation, which is similar to the big data retrieval systems. These indices can reduce the search space and provide an estimation of the target sequence locations in the reference sequence. Our implemented genome indexing technique models a nucleotide sequence as a graph by counting the transitions between each pair of nucleotides. To be more specific, as shown in Figure 9, we take a graph with four states according to the different types of nucleotides and sixteen vertices according to all possible transitions between nucleotides. We read the first nucleotide of the sequence and treat it as the initial state. Then, we move from one state to the other state by scanning the next nucleotide repeatedly until the end of the sequence. Afterwards, we calculate the number of nucleotide transitions (we count how

many times we pass one vertex in the graph) and store them in a 4×4 matrix. Finally, we normalize the resulting matrix as follows: k_{sw} is the number that has the S-type nucleotide immediately before the W-type nucleotide.

$$I = \begin{array}{c} \\ A \\ C \\ G \\ T \end{array} \begin{array}{c} \begin{array}{cccc} A & C & G & T \end{array} \\ \begin{pmatrix} k_{aa} & k_{ac} & k_{ag} & k_{at} \\ k_{ca} & k_{cc} & k_{cg} & k_{ct} \\ k_{ga} & k_{gc} & k_{gg} & k_{gt} \\ k_{ta} & k_{tc} & k_{tg} & k_{tt} \end{pmatrix} \end{array} \times \frac{1}{\sum_{s,w \in \{a,c,g,t\}} k_{sw}}$$

4.1.2 Indexing matching

The goal of this step is to find similar indices based on the information of the sequence. We define a symmetric distance function between two index matrices I and J as follows: $D_{MSE}(I,J) = \|I - J\|_f$, where $\|\cdot\|_f$ is the Frobenius norm of the matrix.

After generating the indices of the reference sequence and the target sequence, the D_{MSE} distances to all of the reference sequence indices are calculated, where the best similar indices in terms of D_{MSE} is chosen as our location.

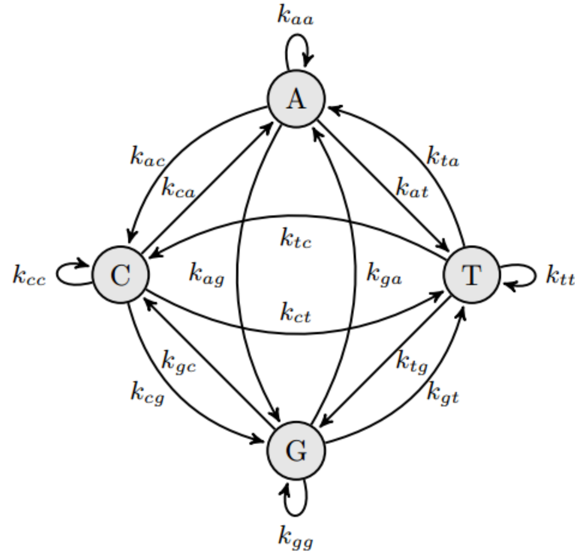

Figure 9 The transition diagram between nucleotides

A summary of the main procedure for our proposed alignment method is shown in Algorithm 1.

Algorithm 1 Proposed nucleotide sequence alignment algorithm for finding the location of the input sequence.

Inputs: a reference sequence $x \in \mathbb{R}^M$, a target sequence $y \in \mathbb{R}^N$.

Initialize: a 4×4 state matrix I storing the numbers of nucleotide states.

Fill the reference state matrix

Fill the target state matrix

Find the best similar subsequence from the reference sequence

Output: the estimated location of the target sequence in the reference sequence.

4.2 Transition Probability Indexing CUDA System

The Transition Probability Indexing implementation consists of 5 modules: sequence reading module, pre-processing module, CPU Transition Probability Indexing

processing module, GPU Transition Probability Indexing processing module, and output display module. Figure 10 shows the CUDA system below.

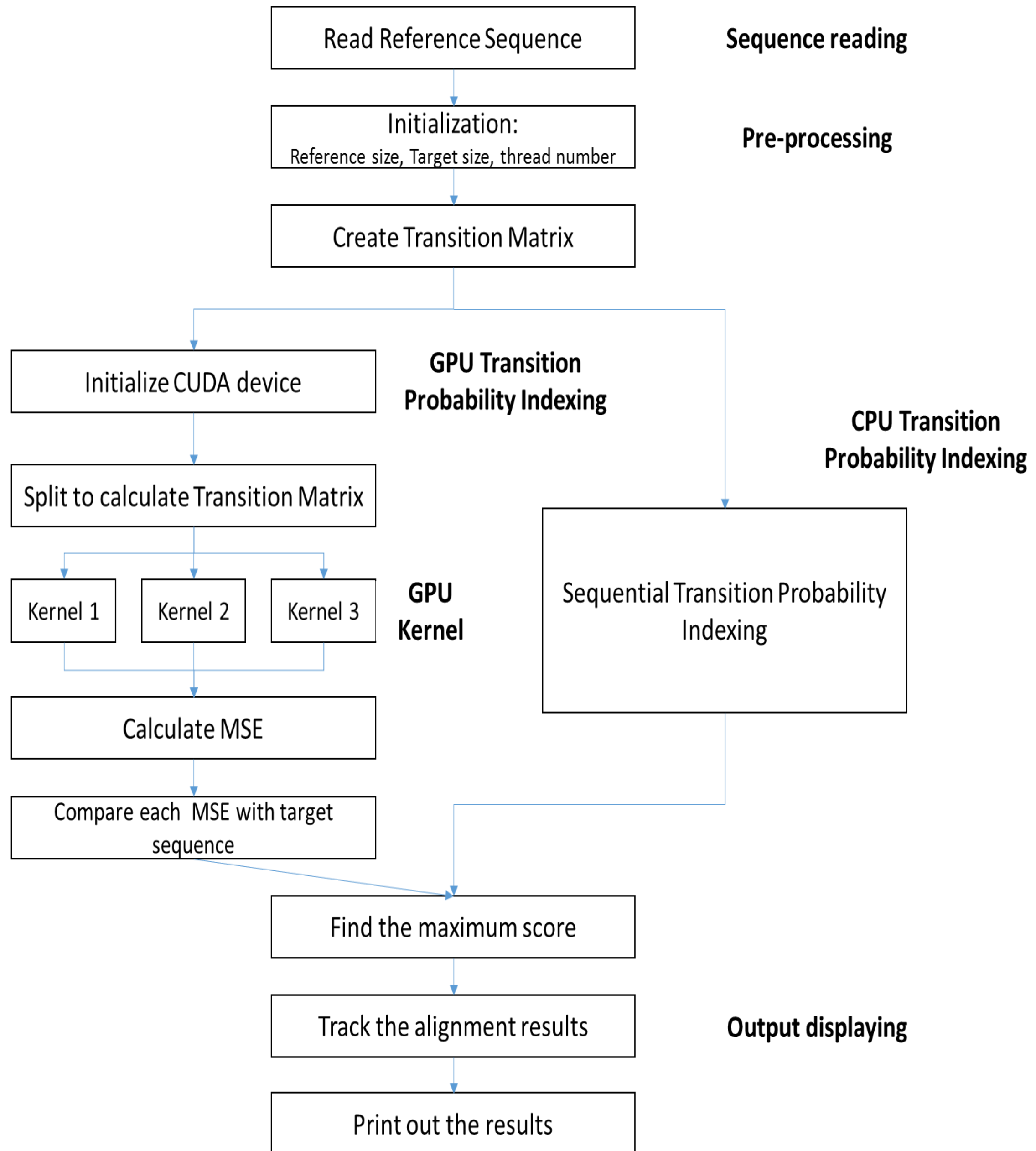

Figure 10 Diagram of Transition Probability Indexing CUDA system

The sequence reading module loads the reference sequence file from a database. It checks to see whether this file is a correctly loadable DNA sequence. It loads the protein sequences from the file and stores the sequences in an array.

The second module is the pre-processing module. This module prepares all of the required variables. It initializes the sequence transition matrix, and changes the sequence character to a number, which makes it easier to use in computations.

The CPU Transition Probability Indexing is the third module. It uses the classic serial method in the implementation.

The GPU Transition Probability Indexing is the fourth module. The code is written in CUDA, using multiple threads to calculate the transition matrix. Single Programming Multiple Data is allowed in CUDA and GK104.

The last module is the output display. It shows the results of the alignment: the number of matched or mismatched sequences, the total running time, and the kernel running time.

4.3 CUDA Implementation

The proposed algorithm for DNA alignment is based on exact and approximate string matching. One of the advantages of using CUDA is that GK104 supports many simultaneous threads. The basic concept of our implementation is to process more transition matrixes at the same time using threads.

4.3.1 Preprocessing

Before the GPU kernel function performs the alignment, some preparation is needed. Considering that the GPU kernel cannot access the host memory, the device memory should be allocated and the data should be copied explicitly from the host to the device before running the kernel.

GPU memory allocation and regular memory allocation work in a very similar way. A vital aspect to take into consideration is that the kernel will require memory

space to save the data results. Considering that each query is run by one thread and will return a single result, each thread will require a single memory position to save its result. These memory positions must be allocated prior to running the kernel. Otherwise, an error will occur when attempting to write the data into device memory that has not been allocated.

The actual memory copy operation is very simple and requires a single command to copy one block of data from the host to the device memory. The kernel cannot begin until all of the data have been copied to the device. It is sometimes helpful to use asynchronous memory copying, so the CPU will continue the processing while the copying takes place.

After the memory has been successfully allocated and all of the necessary data have been copied from the host to the device, it is necessary to call the exact matching kernel function. The detail of the implementation of the algorithm is described below.

4.3.1 CUDA Kernels

The kernel does a simplified transition probability indexing algorithm on all threads. The separate threads in a block work on separate positions for the same read simultaneously. We use the same CUDA kernel function for all calculations. To evaluate the robustness of the proposed method, we generate indices for long human genome sequences (i.e., 5×10^8 nucleotides). For an ideal rate, we divided the reference sequence into subsequences with a length of 10^4, so we picked 10^5 subsequences. And we randomly picked one subsequence as our target sequence. We applied the proposed algorithm to all of the subsequences into the CUDA kernel. We calculated 10^5 subsequences in the CUDA kernel at each time, which means that we

counted ten subsequences in a kernel at one time. Each subsequence has a transition matrix, and we compared each transition matrix with the target sequence transition matrix.

In filling the transition matrix, we used shared memory to update the matrix.

In matching indexes, we paralyzed the input data and used scatter to match the target sequence with each potential query subsequence.

Finally, the result with the most similar indices was the location of the target sequence in the reference sequence. Figure 11 shows the basic kernel structure.

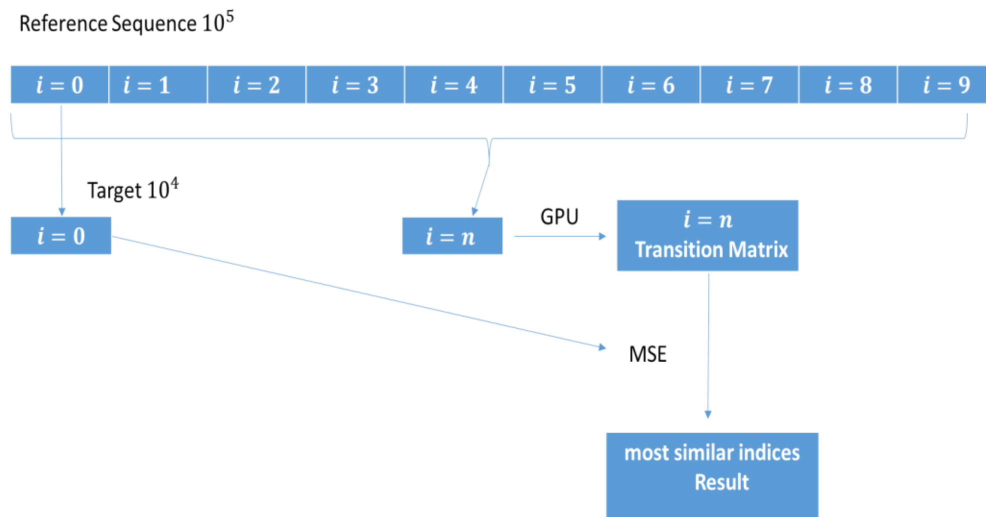

Figure 11 Basic kernel structure

Here is a simplified pseudo-code style version of the GPU kernel code.

```
__device__ int index(index for matrix)
__device__ void Normal_Matrix()
__device__ float MSE(two Matrix) # calculate MSE between target and subreference
sequence

__global__ FUNCTION (alignment kernel)
{
        <copy read into shared memory>
        <Initialize memory and get parameters>
        __syncthreads();
        <For loop to get target transition matrix>
        //constructing transition matrix for target sequence
        for (position)
        {
                row = ref_seq[index];
                column = ref_seq[index];

            atomicAdd(&(ref_Tran_Mat1D[index(i, row, column)]), 1.0);
        }
        //calculate the transition matrix for each subsequence
        for (postion)
        {
                <change dimension from 1D to 2D>
        }
        __syncthreads();
        Normal_Matrix(ref_Tran_Mat2D);
        Normal_Matrix(tar_Tran_Mat);
        measure = sqrt(MSE(ref_Tran_Mat2D, tar_Tran_Mat));
        //compare two transition matrix and get MSE of each pair
        <return result sequence back to host>
}
```

Chapter 5: Results and Conclusions

The performance of the GPU implementation of the transition probability indexing is affected by the number of simultaneous threads and the time used for data transfers between the host and the device. Moreover, the implementation is constrained by several hardware specific properties of the GK104 and the CUDA language. In the current work, we attempted to reduce the runtime by using parallel calculations. In this thesis, we also implemented the sequential code for the proposed algorithm based on the C program.

5.1 Environment

We implemented the proposed algorithm with both the C program and the CUDA program. The details of the environment are as follows:

C-program	CUDA-program
IDE: Code::Blocks	Visual Studio 2013
Compiler: mingw32-gcc	CUDA 6.5
CPU/GPU: intel i7	intel i7 + GeForce GTX 760
System: windows 8 64bit	windows 8 64bit

5.2 Results

We designed our experiments based on the assumption we discussed above. We used the digits 0, 1, 2, 3 to represent the bases A, C, G, T. We found that the CUDA-based program ran faster than the sequential C program. Also, we optimized the algorithm by using shared memory allocation. Our optimized algorithm performs better than our original algorithm. We have drawn Figure 12 and Figure13 below.

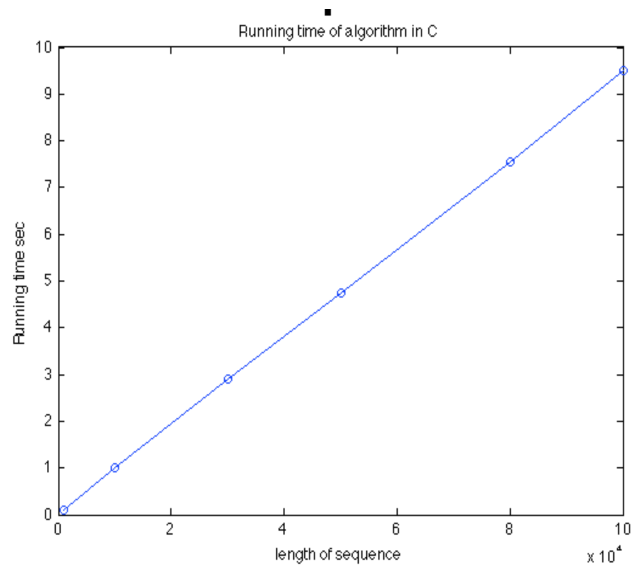

Figure 12 The cost time running in C

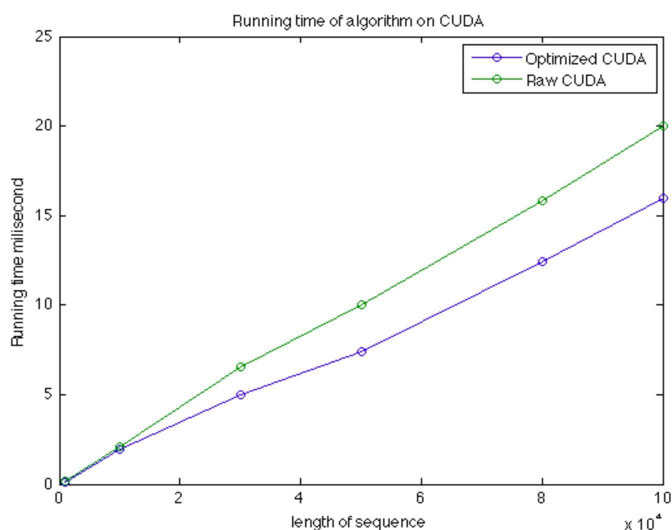

Figure 13 The cost of kernel time running on CUDA

The results from the runtime analysis for the long human genome sequences (5×10^8 nucleotides), as listed in Table 3, show the results of the CUDA based algorithm compared to sequential results achieved by former researchers[1]. Due to the I/O operations of the sequential algorithm, the sequential algorithm is much slower than the parallel algorithm based on CUDA.

Table 3 Results for sequential and parallel

Length of sequence	Sequential	Parallel based on CUDA
5×10^6	21 seconds	1.45 second
5×10^7	24 minutes	21 seconds
5×10^8	5 hours	223 seconds

5.2 Conclusions

In this thesis, we focused on parallelizing the transition probability indexing algorithm with the following results. First, we presented a simple approach to parallelize the proposed method. Our algorithm builds a transition matrix based on the neighboring nucleotides of a reference sequence. Second, we compared short sequences (10^5) for both the serial and parallel methods. Third, we conducted preliminary experiments for long human genome sequences (10^8) and found that the parallelized transition probability indexing algorithm on the GPU is at least 10 times faster than the sequential algorithm. Finally, the proposed method, based on GPUs, can process much longer sequences than the previous approach.

References

[1] A. Roozgard, N. Barzigar, S. Wang, X. Jiang, and S. Cheng, “Empirical Transition Probability Indexing Sparse-Coding Belief Propagation (ETPI-SCoBeP) Genome Sequence Alignment,” *Cancer Inform.*, vol. 13, no. Suppl 1, p. 159, 2014.

[2] E. Holk, M. Pathirage, A. Chauhan, A. Lumsdaine, and N. D. Matsakis, “Gpu programming in rust: Implementing high-level abstractions in a systems-level language,” in *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International*, 2013, pp. 315–324.

[3] N. Corporation, *CUDA C Programming Guide Version 4.0*. 2011.

[4] P. Trancoso and M. Charalambous, “Exploring graphics processor performance for general purpose applications,” in *Digital System Design, 2005. Proceedings. 8th Euromicro Conference on*, 2005, pp. 306–313.

[5] “Genetics home reference.” [Online]. Available: https://ghr.nlm.nih.gov/handbook/basics/dna.

[6] M. K. Hanif, “Mapping dynamic programming algorithms on graphics processing units,” 2014.

[7] N. Goodnight, R. Wang, and G. Humphreys, “Computation on programmable graphics hardware,” *Comput. Graph. Appl. IEEE*, vol. 25, no. 5, pp. 12–15, 2005.

[8] D. Luebke, “CUDA: Scalable parallel programming for high-performance scientific computing,” in *Biomedical Imaging: From Nano to Macro, 2008. ISBI 2008. 5th IEEE International Symposium on*, 2008, pp. 836–838.

[9] W. paper N. Corporation, “Fermi™"NVIDIA’s Next Generation CUDA™Compute Architecture,” 2009.

[10] N. Corporation, “NVIDIA’s Next Generation CUDA Compute Architecture: Kepler GK104,” 2012.

[11] S. P. Adey, “GPU Accelerated Pattern Matching Algorithm for DNA Sequences to Detect Cancer using CUDA Dissertation,” College of Engineering, Pune, 2013.

[12] T. Krazit, “Industry group to evaluate Apple’s OpenCL as standard,” 2008.

[13] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan, “Brook for GPUs: stream computing on graphics hardware,” in *ACM Transactions on Graphics (TOG)*, 2004, vol. 23, no. 3, pp. 777–786.

[14] AMD, “AMD STREAM SDK http://ati.amd.com/technology/streamcomputing/index.html.”

[15] NVIDIA, *CUDA C Programming Guide*. 2012.

[16] NVIDIA, “CUDA C Programming Guide,” 2010.

[17] G. Encarnao, “Parallelization of DNA alignment algorithms using GPUs.”

[18] S. Shah, “APPLIED MATHEMATICS CORNER ‘DNA Computation and Algorithm Design’ Cambridge,” no. MA 02138, 2009.

[19] N. Corporation, “CUDA TOOLKIT DOCUMENTATION,” 2015.

[20] J. Kleinberg and É. Tardos, *Algorithm design*. Pearson Education India, 2006.

[21] F. Sanger, S. Nicklen, and A. R. Coulson, “DNA sequencing with chain-terminating inhibitors,” *Proc. Natl. Acad. Sci.*, vol. 74, no. 12, pp. 5463–5467, 1977.

[22] O. Ohara, R. L. Dorit, and W. Gilbert, “Direct genomic sequencing of bacterial DNA: the pyruvate kinase I gene of Escherichia coli,” *Proc. Natl. Acad. Sci.*, vol. 86, no. 18, pp. 6883–6887, 1989.

[23] F. Sanger and A. R. Coulson, “A rapid method for determining sequences in DNA by primed synthesis with DNA polymerase,” *J. Mol. Biol.*, vol. 94, no. 3, pp. 441–448, 1975.

[24] S. B. Needleman and C. D. Wunsch, “A general method applicable to the search for similarities in the amino acid sequence of two proteins,” *J. Mol. Biol.*, vol. 48, no. 3, pp. 443–453, 1970.

[25] T. F. Smith and M. S. Waterman, “Identification of common molecular subsequences,” *J. Mol. Biol.*, vol. 147, no. 1, pp. 195–197, 1981.