GPU accelerated sequence alignment /Zhao Kaiyong.

Kaiyong Zhao

Follow this and additional works at: https://repository.hkbu.edu.hk/etd_oa

Recommended Citation
https://repository.hkbu.edu.hk/etd_oa/378

This Thesis is brought to you for free and open access by the Electronic Theses and Dissertations at HKBU Institutional Repository. It has been accepted for inclusion in Open Access Theses and Dissertations by an authorized administrator of HKBU Institutional Repository. For more information, please contact repository@hkbu.edu.hk.
DATE: November 15, 2016

STUDENT'S NAME: ZHAO Kaiyong

THESIS TITLE: GPU Accelerated Sequence Alignment

This is to certify that the above student's thesis has been examined by the following panel members and has received full approval for acceptance in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Chairman: Prof. Wu Xiaonan
(Designated by Dean of Faculty of Science)

Internal Members: Prof. Ng Joseph K Y
Professor, Department of Computer Science, HKBU
(Designated by Head of Department of Computer Science)

Dr. Chen Li
Assistant Professor, Department of Computer Science, HKBU

External Members: Dr. He Bingsheng
Associate Professor
School of Computer Science and Engineering
Nanyang Technological University
Singapore

Dr. Wang Qixin
Associate Professor
Department of Computing
The Hong Kong Polytechnic University

Proxy: Dr. Cheung Kwok Wai
Associate Professor, Department of Computer Science, HKBU
(as proxy for Dr. He Bingsheng)

In-attendance: Dr. Chu Xiaowen
Associate Professor, Department of Computer Science, HKBU

Issued by Graduate School, HKBU
GPU Accelerated Sequence Alignment

Zhao Kaiyong

A thesis submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy

Principal Supervisor: Dr. Chu Xiaowen

Hong Kong Baptist University

November 2016
DECLARATION

I hereby declare that this thesis represents my own work which has been done after registration for the degree of MPhil (or PhD as appropriate) at Hong Kong Baptist University, and has not been previously included in a thesis or dissertation submitted to this or any other institution for a degree, diploma or other qualifications.

I have read the University’s current research ethics guidelines, and accept responsibility for the conduct of the procedures in accordance with the University’s Committee on the Use of Human & Animal Subjects in Teaching and Research (HASC). I have attempted to identify all the risks related to this research that may arise in conducting this research, obtained the relevant ethical and/or safety approval (where applicable), and acknowledged my obligations and the rights of the participants.

Signature: zhaokaijong
Date: November 2018
Abstract

DNA sequence alignment is a fundamental task in gene information processing, which is about searching the location of a string (usually based on newly collected DNA data) in the existing huge DNA sequence databases. Due to the huge amount of newly generated DNA data and the complexity of approximate string match, sequence alignment becomes a time-consuming process. Hence how to reduce the alignment time becomes a significant research problem. Some algorithms of string alignment based on HASH comparison, suffix array and BWT, which have been proposed for DNA sequence alignment. Although these algorithms have reached the speed of O(N), they still cannot meet the increasing demand if they are running on traditional CPUs.

Recently, GPUs have been widely accepted as an efficient accelerator for many scientific and commercial applications. A typical GPU has thousands of processing cores which can speed up repetitive computations significantly as compared to multi-core CPUs. However, sequence alignment is one kind of computation procedure with intensive data access, i.e., it is memory-bounded. The access to GPU memory and IO has more significant influence in performance when compared to the computing capabilities of GPU cores. By analyzing GPU memory and IO characteristics, this thesis produces novel parallel algorithms for DNA sequence alignment applications. This thesis consists of six parts. The first two parts explain some basic knowledge of DNA sequence alignment and GPU computing. The third part investigates the performance of data access on different types of GPU memory. The fourth part describes a parallel method to accelerate short-read sequence alignment based on BWT algorithm. The fifth part proposes the parallel algorithm for accelerating BLASTN, one of the most popular sequence alignment software. It shows how multi-threaded control and multiple GPU cards can accelerate the BLASTN algorithm significantly. The sixth part concludes the whole thesis.

To summarize, through analyzing the layout of GPU memory and comparing data under the mode of multithread access, this thesis analyzes and concludes a perfect optimization method to achieve sequence alignment on GPU. The outcomes can help practitioners in bioinformatics to improve their working efficiency by significantly reducing the sequence alignment time.
Acknowledgements

Seven years in Hong Kong are days of improvement and progress and the best time of my life. For these years, I have once lost, depressed and experienced all kinds of painfulness and happiness that all PhD students should have experienced. From Master to Doctor, I want to show my great gratitude to Dr. Chu for his encouragement and support. When I was confused, he didn't give me up and carefully taught me. When study met critical bottleneck, he always gave me enough time and space. Thanks to my companions for their tolerance and understanding. The process of studying is not only learning knowledge but also a spiritual journey, which is to feel science and life with your heart.
# Table of Contents

DECLARATION..............................................................................................................................i

Abstract ......................................................................................................................................ii

Acknowledgements .................................................................................................................iii

Table of Contents ......................................................................................................................iv

List of Tables ...............................................................................................................................vii

List of Figures .............................................................................................................................viii

Chapter 1  Introduction and Background................................................................................ 1

1.1  The scoring model ............................................................................................................. 1

1.2  Background of sequence alignment................................................................................... 2

1.3  Short read alignment algorithms....................................................................................... 4

1.4  GPU computing .................................................................................................................. 6

1.5  Motivation .......................................................................................................................... 7

Chapter 2  Research Problem and Literature Review ............................................................ 9

2.1  Sequence alignment in biology.......................................................................................... 9

2.2  Sequence alignment algorithms......................................................................................... 11

2.3  Dynamic programming algorithms..................................................................................... 12

2.4  Seed match and extend alignment algorithms.................................................................... 13

2.5  GPU based alignment algorithms....................................................................................... 16

Chapter 3  CUDA Memory Model ......................................................................................... 17

3.1  Global memory .................................................................................................................. 20

3.1.1  Shared memory............................................................................................................... 21

3.1.2  Constant memory............................................................................................................. 23

3.2  Test process design............................................................................................................ 24

3.2.1  Demand analysis............................................................................................................. 24

3.2.2  Experimental design....................................................................................................... 24

3.2.3  Testing code schema....................................................................................................... 27

3.3  Experimental results............................................................................................................ 31

3.3.1  Experimental environment ........................................................................................... 31

3.3.2  Experiment analysis........................................................................................................ 32
Chapter 6 Conclusion and future work ................................................................. 96
Publication List ........................................................................................................ 98
References .................................................................................................................. 101
CURRICULUM VITAE ............................................................................................ 107
List of Tables

Table 3-1 Two dimensional array ......................................................... 25
Table 3-2 Data Structure ........................................................................ 25
Table 3-3 Global variables ..................................................................... 27
Table 3-4 Enumerated variables ............................................................... 28
Table 3-5 Structures and foundation class ............................................... 29
Table 3-6 Functions ................................................................................ 29
Table 3-7 Cases ..................................................................................... 30
Table 3-8 Hardware environment ............................................................ 32
Table 3-9 Software environment ............................................................. 32
Table 4-1 Time of align 1 million reads of length 100 against the human genome in seconds ............................................................... 60
Table 5-1 High Level Pseudocode of BLASTN ........................................ 63
Table 5-2 The GPU scan kernel function using small lookup table .......... 75
Table 5-3 The GPU lookup kernel function using small lookup table ........ 75
Table 5-4 The GPU scan kernel function using megablast lookup table .... 77
Table 5-5 The GPU lookup kernel function using megablast lookup table .. 77
Table 5-6 SSE instructions used by s_SeqDBMapNcbiNA8ToBlastNA8() .... 78
Table 5-7 System configuration ............................................................... 80
Table 5-8 Average speedup of G-BLASTN under megablast mode ........ 92
Table 5-9 Speedup of G-BLASTN under blastn mode using Qbac query set 92
Table 5-10 Speedup of pipelined G-BLASTN (GTX780 vs. Intel i7-3820) 94
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-1</td>
<td>Genomes can now be sequenced around 50,000 times faster than in 2000</td>
</tr>
<tr>
<td>2-1</td>
<td>Calculating the value in bottom right-hand corner of each square cells from one of the other three values</td>
</tr>
<tr>
<td>3-1</td>
<td>Typical memory hierarchical structure</td>
</tr>
<tr>
<td>3-2</td>
<td>CUDA memory model</td>
</tr>
<tr>
<td>3-3</td>
<td>Memory access methods</td>
</tr>
<tr>
<td>3-4</td>
<td>One Dimension Access</td>
</tr>
<tr>
<td>3-5</td>
<td>Two Dimension Access method</td>
</tr>
<tr>
<td>3-6</td>
<td>Tree access method</td>
</tr>
<tr>
<td>3-7</td>
<td>One dimension access method</td>
</tr>
<tr>
<td>3-8</td>
<td>Two dimension access method</td>
</tr>
<tr>
<td>3-9</td>
<td>Two dimension access method</td>
</tr>
<tr>
<td>3-10</td>
<td>Tree Access method</td>
</tr>
<tr>
<td>3-11</td>
<td>Exponential data structure</td>
</tr>
<tr>
<td>3-12</td>
<td>Geometric data structure</td>
</tr>
<tr>
<td>3-13</td>
<td>Poisson data structure</td>
</tr>
<tr>
<td>3-14</td>
<td>Random data structure</td>
</tr>
<tr>
<td>3-15</td>
<td>Standard normal data structure</td>
</tr>
<tr>
<td>3-16</td>
<td>Uniform data structure</td>
</tr>
<tr>
<td>3-17</td>
<td>One dimensional array</td>
</tr>
<tr>
<td>3-18</td>
<td>Two dimensional array</td>
</tr>
<tr>
<td>3-19</td>
<td>Tree structure</td>
</tr>
<tr>
<td>3-20</td>
<td>Random access 1k memory</td>
</tr>
<tr>
<td>3-21</td>
<td>Random Access 4K memory</td>
</tr>
<tr>
<td>3-22</td>
<td>Random Access 10k Memory</td>
</tr>
<tr>
<td>3-23</td>
<td>Random access 40k Memory</td>
</tr>
<tr>
<td>3-24</td>
<td>Standard Normal access 1k Memory</td>
</tr>
<tr>
<td>3-25</td>
<td>Standard normal access 4k memory</td>
</tr>
<tr>
<td>3-26</td>
<td>Standard normal access 10k Memory</td>
</tr>
<tr>
<td>3-27</td>
<td>Standard normal access 40k Memory</td>
</tr>
<tr>
<td>3-28</td>
<td>Thread access clock</td>
</tr>
<tr>
<td>3-29</td>
<td>Occupancy of one SM</td>
</tr>
<tr>
<td>3-30</td>
<td>Warp computing and memory access pipeline</td>
</tr>
<tr>
<td>4-1</td>
<td>Suffix Tree</td>
</tr>
<tr>
<td>4-2</td>
<td>BWT</td>
</tr>
<tr>
<td>5-1</td>
<td>SCAN function data access pattern</td>
</tr>
<tr>
<td>5-2</td>
<td>Speed up with CUDA version</td>
</tr>
<tr>
<td>5-3</td>
<td>GPU memory Hierarchy</td>
</tr>
<tr>
<td>5-4</td>
<td>Profiling of BLASTN for 300 query sequences (ranging from 500 to 100,000 bases) against human build 36 genome database under megablast mode. The lengths of the query sequences can be found in Figure 5-12</td>
</tr>
<tr>
<td>5-5</td>
<td>The framework of G-BLASTN</td>
</tr>
</tbody>
</table>
Figure 5-6 Framework of scan sub-step on GPU ............................................................... 74
Figure 5-7 The pipeline mode of G-BLASTN ................................................................... 79
Figure 5-8 Lengths of database sequences ................................................................. 81
Figure 5-9 Lengths of query sequences ......................................................................... 83
Figure 5-10 Speedup of megablast mode (G780 vs. i7-3820; speedup-1 is for 1 CPU core, speedup-4 is for 4 CPU cores) ................................................................. 86
Figure 5-11 Speedup of megablast mode (G780 vs. E5620; speedup-1 is for 1 CPU core, speedup-4 is for 4 CPU cores, speedup-8 is for 8 CPU cores) .............................. 88
Figure 5-12 Speedups of blastn mode (G780 vs. i7-3820; speedup-1 is for 1 CPU core, speedup-4 is for 4 CPU cores) ................................................................................. 89
Figure 5-13 Speedups of blastn mode (G780 vs. E5620; speedup-1 is for 1 CPU core, speedup-4 is for 4 CPU cores, speedup-8 is for 8 CPU cores) .............................. 90
Figure 5-14 Speedup of G-BLASTN on human genome database (GTX780 vs. Two Xeon E5620) ........................................................................................................ 91
Figure 5-15 The time distribution of Qbac on Human genome database. Remark: In (b), the time of ungapped extension is too short to be noticeable ................................................. 93
Chapter 1 Introduction and Background

Nature is a tinkerer and not an inventor [1]. New biological sequences (e.g., DNA) are adapted from pre-existing sequences rather than invented de novo [2].

The DNA, RNA or protein sequences in computational bioinformatics are composed of characters like \{A, C, G, T \ldots\}, so a biological sequence is a string composed of finite characters. In bioinformatics, a sequence alignment is a way of arranging the sequences of DNA, RNA, or protein to identify regions of similarity that may be a consequence of functional, structural, or evolutionary relationships between the sequences [3]. Aligned sequences of nucleotide or amino acid residues are typically represented as finding the maximum likelihood of the substring of two strings in special comparison ways. In the same position of the two substrings, it can contain different characters or gaps inserted. We can quantify the similarities of two substrings by calculating a score to the two aligned substrings according to some predefined scoring models.

1.1 The scoring model

How to define the similarity of two sequences? First we should know that each character in a biological sequence has its own story. For example, there are two DNA sequences $x$ and $y$, with length $l$ and $k$ respectively. Sequence $x$ and $y$ are constructed by characters from set \{A, C, G, T\}. The character in sequences has the probability $p$.

But for the bioinformatics research, someone defined the probability of $x_i$ with the frequency of character \{A, C, G, T\}, \{$p_A, p_C, p_G, p_T$\} [1]. If one sequence is similar to another sequence, it means that they have exact match, or in some positions they have exact match but different characters may appear at other positions. A scoring matrix can help to measure how similar two sequences are.

Using the most popular sequence alignment tool BLAST as an example [50]. There are three steps to align two sequences by BLAST. First, find a seed (a short substring) which appears in both sequences. Second, extend the seed with no-gap, but scoring for extended strings. The last step is to extend the string with gap (insertion or
deletion). In each step, we can use the scoring matrix to give a score to a matched character.

The **BLOSUM** (**BLO**cks of Amino Acid **SU**bstitution **M**atrix) matrix is a substitution matrix used for sequence alignment of proteins. BLOSUM matrices are used to score alignments between evolutionarily divergent protein sequences and they are based on local alignments. BLOSUM matrices were first introduced in a paper by Henikoff and Henikoff [4]. Choose different scoring matrix will reflect biological or statistical observations which known sequences is important to produce better alignments.

In practice, DNA and RNA sequence alignments often simply utilize three scores: a positive match score, a negative mismatch score, and a negative gap deduction.

### 1.2 Background of sequence alignment

Sequence alignment is a big research area in bioinformatics. There are different methods for different alignment requirements, such as pairwise alignment, multiple sequence alignment, structural alignment and phylogenetic analysis. Pairwise sequence alignment methods are used for two query sequences. It is used to find the best-matching piecewise (local) or global alignments. Pairwise alignments can only be used between two sequences at one time. There are three primary methods of producing pairwise alignments: dot-matrix methods, dynamic programming, and word methods [3]. Multiple sequence alignment is an extension of pairwise alignment for many sequences. It can align more than two sequences at a time. Multiple sequence alignments are computationally challenging, and most formulations of the problem lead to NP-complete combinatorial optimization problems [11] [12]. Structural alignment, which is usually specific to protein or works sometimes on RNA sequences. It uses information about the secondary and tertiary structure of the protein or RNA molecule to aid in aligning the sequences. At last, Phylogenetic is closely related to sequence alignment due to the shared necessity of evaluating sequence relatedness [10].
In this research, we are focusing on pairwise alignment algorithms of biological sequences. As we mentioned previously, there are three major ways to process pairwise alignment: dot-matrix methods, dynamic algorithms and word methods. A dot matrix method can provide a global view of local similarities between two sequences. It implicitly produces a family of alignments for each of individual sequence regions. It is very useful for detection of repeats within protein sequences. Detection of shared domains between protein sequences is also performing well.

Dynamic sequence alignment algorithms can be classified as global alignments and local alignments. Global alignments align every residue in every sequence. When the sequences in the query set are similar and of roughly equal size, the global alignment is very useful. A general global alignment technique is the Needleman–Wunsch algorithm [5]. The Needleman-Wunsch algorithm is based on dynamic programming. In 1982, Gotoh improved this algorithm [6]. Local alignments are more useful for two sequences: one is the dissimilar sequences that are suspected to contain regions of similarity; one is the similar sequence motifs within their larger sequence context. The Smith–Waterman algorithm [49] is based on dynamic programming. It is a general local alignment method also. In fact, in 1972 David Sankoff [9] proposed a better dynamic programming algorithm with quadratic running time for the same problem (no gap deduction). In 1968, T. K. Vintsyuk for speech processing ("time warping") discovered a similar quadratic-time algorithm [16]. In 1974, Robert A. Wagner and Michael J. Fischer had new algorithms for string matching [47]. Needleman and Wunsch supposed their problem as in terms of maximizing similarity. There is another problem to minimize the edit distance between sequences. In 1974, Peter H. Sellers showed that the two problems are equivalent [48]. “With sufficiently similar sequences, there is no difference between local and global alignments” [8].

The above dynamic programming algorithms are too slow for aligning long sequences such as a human genome whose length is about 3 billion. To overcome the computational issues, word methods, also known as k-tuple methods, become the main stream of sequence alignment. These methods are heuristic algorithms that are not guaranteed to find the optimal alignment solution. They are significantly more
efficient than dynamic programming [8]. Typically, a set of \( k \)-tuple words are defined as seeds. The seeds’ positions in the reference sequence database are stored in some index structure such that it is very efficient to find matches of a seed. Then such matches can be extended to find the longest local alignment. The FASTA and the BLAST family tools are word methods that are best known for their implementation in the database search [3].

1.3 Short read alignment algorithms

In recent years, the second-generation DNA sequencing machines are able to generate millions of short DNA fragments (referred to as short reads hereafter) in a very short time. These short reads are generated at random locations from the tested genome, and the ordering and location information are totally lost. A critical task before making use of these short reads is to assemble these fragments into a full genome, which can be done by resequencing algorithms, i.e., by aligning these short reads to an existing reference genome. Those traditional sequence alignment software as BLAST and BLAT [64] are unable to cope efficiently with the huge amount of reads generated. At the same time, SSAHA [65] is optimized to find long alignments and fails in practice on most short queries [14]. Since 2007, there has been a plenty of research activities in developing efficient algorithms for short reads sequence alignment. Some of these tools, such as Eland (Cox, 2007, unpublished material), RMAP [66], MAQ [67], ZOOM [68], SeqMap [69], CloudBurst [70] and SHRIIMP (http://compbio.cs.toronto.edu/shrimp), WHAM (http://research.cs.wisc.edu/wham/) [71], work by hashing the read sequences and scanning through the reference sequence [15]. Index-based alignment algorithms were beginning to occupy the stage of history. The ungapped short reads sequence alignment counts the number of exact hits of a string of length \( n \) in \( O(n) \) time, independent of the size of the genome. But for the gapped sequence alignment, those algorithms are still too slow.

The DNA sequencing technique is evolving very fast, and the cost of sequencing a whole human genome drops significantly: from 3 billion dollars for the first human
genome, to a few thousand dollars today. The genomes can be sequenced around 50,000 times faster than 2000 in 2010 [17]. Figure 1-1 shows the evolvement of DNA sequencing technique.

![Speed reading](image)

**Speed reading**

Genomes can now be sequenced around 50,000 times faster than in 2000

How to handle the exponential growth of sequencing data becomes a huge challenge. We identify three important issues for our research. First, the efficiency of BLAST, the most widely used bioinformatics tool, becomes unacceptable because of the ever-increasing size of sequence databases. In following sections we will analyze the BLAST algorithms and use parallel computing methods to accelerate it. Second, existing short read alignment algorithms cannot cope with the huge data generated from sequencing machines. The time to assemble the short reads is even longer than the time to generate the short reads. Third, the future sequencing machines will generate longer reads which will contain more errors. Current short read alignment tools can only accommodate up to three mismatches. We shall seek a novel method to support approximate sequence alignment that can tolerate more mismatches.
1.4 GPU computing

GPUs (Graphic Processing Units) are dedicated hardware for manipulating computer graphics. Due to the huge computing demand for real-time and high-definition 3D graphics, the GPU has evolved into a highly parallel, multithreaded, many core processor. Recent boosts in GPUs open a new era of GPU computing. For example, commodity GPUs like NVIDIA’s GK104 has 1536 processing cores and can achieve 3.01 TFLOPS of computational horsepower [23]. The advances of computing power in GPUs have driven the development of general-purpose computing on GPUs (GPGPU). The first generation of GPGPU requires that any non-graphics application must be mapped through graphics application programming interfaces (APIs). NVIDIA provided a general-purpose parallel programming model, namely Compute Unified Device Architecture (CUDA) [19] [20], extending the C programming language for general-purpose application development. In November 2006, NVIDIA introduced CUDA, a general purpose parallel computing architecture. CUDA is a new parallel programming model. It has a new instruction set architecture. CUDA leverages the parallel compute engine in NVIDIA GPUs to solve many complex computational problems in a more efficient way than on a CPU.

CUDA comes with a software environment that allows developers to use C/C++ as a high-level programming language. Other languages or application programming interfaces are supported, such as CUDA FORTRAN, CUDA Python, CUDA Java, OpenCL, and Direct Compute.

Meanwhile, another GPU vendor AMD also introduced Close To Metal (CTM) programming model which provides an assembly language for application development [21]. The new architecture of Next-Gen FirePro W9000 has 5TFLOPS single precision. Intel also exposed Larrabee, a many core GPU architecture specifically designed for the market of GPU computing this 2008 [24]. At the same time, the CPUs’ performance is only about 100GFLOPS [26] around.

So the GPU processors are more powerful than CPU processors. Since the release
of CUDA, GPU has been used for speeding up a large number of applications. More importantly, the NVIDIA CUDA programming model makes it easier for developers to develop non-graphic applications using GPU. In CUDA, the GPU becomes a dedicated coprocessor to the host CPU, which works in the principle of Single-Program Multiple Data (SPMD) where multiple threads based on the same code can run simultaneously. There are some researchers who use GPU to speed up different types of sequence alignment [28] [29] [30] [31] [32] [33] [34] [35]. Weiguo Liu & Bertil. speed up multiple sequence alignment with graphics hardware in [28]. Schatz, M. C., Trapnell, C. high-throughput sequence alignment with GPU in [29]. In [30], Manavski, Svetlin A. use CUDA to speed up Smith-Waterman sequence alignment algorithm, which is the basic sequence alignment algorithm. In [31], Ning Wan. accelerate BLASTN, but it’s only porting the CPU code into GPU. Liu, Y. in 2009, CUDASW++ also speeded up smith-waterman algorithm in [32]. In article [33], Panagiotis D. Vouzis and Nikolaos V. Sahinidis use GPU to speed up BLAST algorithm. In [34] and [35], C. Liu et al. speed up SOAP3 algorithm.

1.5 Motivation

Due to the continuous development of DNA sequencing technology, the existed CPU-based DNA sequence alignment algorithms can’t meet the growing demand for increasing DNA data. As GPU technology develops continuously, the now existed GPU computing performance has far exceeded the CPUs’. Especially for large-scale data parallel applications, it is more suitable to run it on GPU. Using GPU for DNA sequence alignment becomes a viable option.

Most DNA sequence alignment algorithms are designed based on CPU architecture. Hence, most of them are serial algorithm and not suitable for parallel computing architecture. In early 2009 I had tried to port the most time-consuming part in BLASTN (CPU based DNA alignment application) to GPU. However, the overall performance was not good. To get better performance on GPU, we need to have an in-depth study on DNA sequence alignment algorithms and GPU computing.
architecture. So, in the later chapters, we will introduce some existing DNA sequence alignment algorithms. Meanwhile, in order to optimize DNA sequence alignment algorithm, we will propose a new kind of GPU parallel computing model.

In Chapter 2 we will introduce various existing DNA sequence alignment algorithms. In Chapter 3, a new GPU parallel computing model based on memory operation will be introduced. Based on the theory mentioned in Chapter 3, SOAP3 algorithms in GPU will be implemented in Chapter 4. Chapter 5 describes optimization of BLASTN on GPU. Chapter 6 concludes the thesis.
Chapter 2  Research Problem and Literature Review

2.1  Sequence alignment in biology

For a biologist the sequences of RNA, DNA or protein strings are made of their respective alphabet set $\Sigma$ in different ways. There are some types of finite alphabet set for the string.

Though all DNA are built from nucleotides that always contain sugar, a phosphate group, but it is common to distinguish them by one of the four nucleic acids: Adenine, Cytosine, Guanine, and Thymine. The DNA alphabet set $\Sigma$ consists of their initials: DNA = {A, C, G, T}. Practically in the computer processing we can use {00, 01, 10, 11} binary code to represent the DNA characters {A, C, G, T} to save memory.

In RNA, Thymine is replaced by Uracil, so T is changed to U in the alphabet: RNA = {A, C, G, U}. But normally, the RNA sequence is usually re-transcribed into DNA for technical reasons.

There are 20 amino acids in protein. The set of characters for protein are Protein= {A, R, N, D, C, Q, E, G, H, I, L, K, M, F, P, S, T, W, Y, V}.

It is not always the case that all positions in a sequence are precisely known. In this scenario, an extended alphabet will be used [36].

A sequence alignment is not only a string matching method, but also an objective to find “similar” regions. It may provide additional information on the functional, mutational, structural, evolutionary and other interests between the sequences under study. Aligned sequences are typically represented in rows, one on top of the other.

As mentioned in Chapter 1, there are pairwise alignments for two sequences, and multiple alignments for many sequences, which more than two sequences. We will focus on pairwise alignment in the thesis. In practice, the pairwise alignment can be classified as global and local sequence alignments. The global sequence alignments are designed to find the best alignment of both sequences in their whole sequence. The local alignment algorithm finds the best approximate sub-sequence matched within two given sequences. The local sequence alignments are designed basically to
search for highly similar regions within the two given sequences [37].

In the following sections, we provide several fundamental definitions which are necessary to understand the alignment algorithms described in the later chapters.

**Definition 1**: Let $\Sigma$ be the set of characters, e.g., DNA = \{A, T, C, G\}, which is called the DNA alphabet set. A sequence $L$ is a string of characters from alphabet $\Sigma$. Those characters in string are all written from left to right contiguously and occupy a unique position in the sequence $L$. $|L|$ is denoted as the length of $L$ in characters. $L[i]$ is denoted as the $i$-th character of $L$.

There are two ways to quantify similarity of two sequences. First method is a similarity measure. It is a function that associates a numeric value with a pair of sequences, with the idea that a higher value indicates greater similarity. Second method is the notion of distance. This method is somewhat dual to similarity. It treats sequences as points in a metric space. Hamming Distance is always used to be the distance function. But sometimes similarity measures are a little more flexible [36].

For similarity measure method, it uses a scoring function. First, scoring function for the alignment of pairs of characters is defined. Then scores of aligned pairs of characters are added to get the sequence alignment score. According to the relationship between pair of characters in an alignment string, the alignment score function $F$ can be classified into three types.

**Definition 2**: The character alignment score $F$ is a real valued function on pairs of characters. The score function $F$ is called:

a) Match alignment score, in the same place the two characters are same.

b) Mismatch alignment score, in the same place if the characters are not indel but also different.

c) Gap alignment score, if one character is indel in the string and the other is not.

**Definition 3**: Let $L_1$ and $L_2$ be two aligned sequences of length $m$. The alignment score $S$ of $L_1$ and $L_2$ is defined as

$$S(L_1, L_2) = \sum_{k=0}^{m} S(L_1[k], L_2[k]).$$

The sequence alignment scoring function $S$ is called sum-of-pair. In Biological
sequence analysis: probabilistic models of proteins and nucleic acids [1] for the unrelated or random model R, the $S(L_1[k], L_2[k])$ function equal to $\log \frac{p_{ab}}{p_a p_b}$ where the $p_a$ and $p_b$ is each of the frequency of letter $a$ and $b$ occurs independently. In the alternative match model M, aligned pairs of residues occur with a joint probability $p_{ab}$ [39]. For example one character change operations that change $x$ into $y$. We introduce a gap character - and define that:

Given pair $(x, y), x \neq y$,

$\sigma(x, x) = a, a \in R^+:$ denotes a match $(x = y)$,

$\sigma(x, y) = b, b \in R^-:$ denotes replacement of $x$ to $y$, where $x \neq y$,

$\sigma(x, -) = c, c \in R^-:$ denotes deletion of character $x$,

$\sigma(-, y) = d, d \in R^-:$ denotes insertion of character $y$.

The constant values ‘$a’$, ‘$b’$, ‘$c’$ and ‘$d’$ are usually obtained from a scoring matrix. A scoring matrix is an $m \times m$ matrix. In the matrix, each cell contains a score for the corresponding pair of bases. The BLOSUM (BLOcks of Amino Acid Substitution Matrix) matrix is a substitution matrix used for sequence alignment of proteins [4] [37] [39].

The gap penalty is the standard cost associated with a gap of length $l$, and usually it is given by a linear score function [1]. DNA and RNA alignments may use a different scoring matrix. But in practice, we just give a positive match score, a negative mismatch score, and a negative gap deduction, for simplicity.

2.2 Sequence alignment algorithms

In Section I, we introduced three ways to do sequence alignment. Over the past several years, there are many algorithms have been proposed for sequence alignment. These algorithms fall under two distinct categories: optimal and heuristic. Optimal algorithms focus to produce the best alignment. The heuristic algorithms produce near-best alignment. Dot-matrix method can get the whole result for pairwise sequence alignment. However, in most cases, it is not necessary, and it is very slow to process the alignment. The dynamic programming method is guaranteed to find an
optimal alignment in a given particular scoring function. At the same time, identifying a good scoring function is often an empirical rather than a theoretical matter. Although dynamic programming can be extended to more than two sequences, it is prohibitively much slow for large numbers of or extremely long sequences [8]. The third method is word method, which is heuristic method that is not guaranteed to find an optimal alignment solution. Those word methods are significantly more efficient than dynamic programming in speed. In practical, the third and second methods are always used together. Most of fast alignment algorithms usually construct auxiliary data structures, such as lookup table or indexed dictionary, for the read sequences or the reference data base, in some algorithms with both of them. Depending on the different types of the index, alignment algorithms can be largely classified into four categories: algorithms based on hash tables, algorithms based on suffix trees, algorithms based on merge sorting and search engine. All those algorithms are essentially as the same seed-and-extend exemplification [40].

2.3 Dynamic programming algorithms

There are many algorithms have been proposed for optimal alignment algorithms. Those algorithms are all based on dynamic programming. Needleman-Wunsch Algorithm is an optimal algorithm, and it is for global sequence alignment. The algorithm uses the same score function as the local sequence alignment. In 1981, the Smith-Waterman proposed an algorithm based on dynamic programming algorithm, named Smith-Waterman algorithm.

The global sequence alignment is looking for the best match between sequences from one end to the other. A much more common situation is where we are looking for the best alignment between subsequences of \( x \) and \( y \). The highest scoring alignment of subsequences of \( x \) and \( y \) is called the best local alignment. For local alignment algorithms Smith-Waterman Algorithm, the scoring matrix \( F \) shows as below:
\[ F(i, j) = \max \begin{cases} 0, \\ F(i - 1, j - 1) + s(x_i, y_j), \\ F(i - 1, j) - d, \\ F(i, j - 1) - d, \end{cases} \]

\( F(i, j) \) is the score of the best alignment between the initial segment \( x_1 \ldots i \) of \( x \) up to \( x_j \) and the initial segment \( y_1 \ldots j \) of \( y \) up to \( y_j \). We can build \( F(i, j) \) recursively. First, \( F(0,0) = 0 \). We then proceed to fill the matrix from top left to bottom right. If \( F(i - 1, j - 1), F(i - 1, j) \) and \( F(i, j - 1) \) are known, it is possible to calculate \( F(i, j) \). There are four possible ways that the best score, \( F(i, j) \), takes the value 0 if all other options have less than 0. \( F(i, j) \) of an alignment up to \( x_i, y_j \) could be obtained: \( x_i \) could be aligned to \( y_j \), in which case \( F(i, j) = F(i - 1, j - 1) + s(x_i, y_j) \); or \( x_i \) is aligned to a gap, in which case \( F(i - 1, j) - d \); or \( y_j \) is aligned to a gap, in which case \( F(i, j - 1) - d \). The best score up to \( (i, j) \) will be the largest of these four options.

![Figure 2-1](image.png)

Figure 2-1. Calculating the value in bottom right-hand corner of each square cells from one of the other three values.

When step by step to the end of matrix, a procedure traces back from the highest score cell to the cell containing zeros \((0, 0)\). The alignment string is actually built during this trace back stage.

### 2.4 Seed match and extend alignment algorithms

The idea of seed match and extending alignment algorithms can be traced back to BLAST in 1990 [50], which is one of the most popular sequence alignment tools. The paper [5] has received more than 40,000 citations since its publication. BLAST includes three major steps, collection a list of high scoring words, scanning the database to find those hits, and then extending those hits, each of which varies
somewhat depending on whether the database contains proteins or DNA sequences.

There are several popular seed searching techniques for heuristic algorithms, such as look-up table in hash model, suffix tree structure, suffix array, and Burrows–Wheeler Transform (BWT). In this following, we are going to briefly explain the two methods.

**Look-up Table:** The look-up table is a data structure. It is usually implemented as an array, used widely in heuristic local sequence alignment problems. The main idea of the look-up table is to have seeds occupy unique positions in an array. So, the position of seed in the array is easily obtained from the value computed using a hashing function. In practical, we can make a hash table for query sequence, reference sequence or both of them. Hashing provides an easy method to avoid a quadratic number of character comparisons in most practical situations. Instead of in check if there is a pattern text of each position, this seems to be more effective check only on the contents of the window similar pattern. In order to check the similarities between those two words a hashing function is used. Once make a look-up table for \( S_1 \), then one can linearly move through \( S_2 \) and find all seeds of length ‘\( l \)’ in \( S_1 \) using the look-up table. The exact positions of seeds in sequence \( S_2 \) is known when moving through it. With this method, all the seeds can be found in time \( O(m) \), where \( m \) is the length of \( S_2 \).

**Suffix tree:** Suffix tree is a data structure that represents the internal structure of a string in a comprehensive manner [56]. The exact matching problem for sequences can be solved in linear time \( O(m) \), where \( m \) is the length of the string. Weiner developed the first linear time suffix tree back in 1973 [57], which was later improved by [58] [59]. A string \( S \) of \( m \) words of the suffix tree \( T \), is the one that contains a root node to the tree, the tree with exactly \( m \) leaves, these leaves are given the label from 1 to \( m \). Each internal node, in addition to other than the root node, at least has two child nodes, and each edge with \( S \) is a non-empty string to identify. Any two edges from the same node identity will not be the same words. The key feature of the suffix tree: For any leaf \( i \), \( S \) happens to spell out all the logos of the experience from the root node to the leaf edge series start from the position \( i \) of the suffix, i.e., \( S [ i, ..., m ] \). The identity
of the tree node is defined as a series from the root to the node identity of all the edges. Similarly, the suffix tree of several strings, called an extension of the suffix tree: n string $S_n$, where the string is the length of $mn$ by these strings to form an extension of the suffix tree $T$, it is a the root node to the tree, the tree has $mn$ leaves, each leaf with a two-digit coordinate tuple $(k, l)$ to identify, where $k$ ranges from 1 to $n$, $l$, range from 1 to $mk$, each internal node, in addition to the root node, has two child nodes and each edge with a non-empty $S$ word constitute a substring to identify. And the first word of the logo side of any two from the same node cannot be the same. Any leaves $(i, j)$, the series from the root to the leaves experienced the identity of all the edges exactly spell the suffix of $S_i$, the suffix from the position $j$ the beginning, that they spell of $S_i[j .. mi]$.

A generalized suffix tree is that combines the suffixes of the string $S_1, S_2, \ldots, S_m$. An improvement in terms of speed and memory efficiency was proposed by [59], it also known as online construction suffix tree algorithm.

**Suffix Array:** Given a text $T[1...m]$, then we define the suffix array of $T$, denoted by $SA[1...m]$, as $SA[i] = j$ if the suffix $T[j...m]$ is lexicographically the $i$-th smallest suffix among all suffices of $T$ (and we say that the rank of the suffix $T[j...m]$ is $i$). That is, $SA$ stores the starting positions of all suffices of $T$ in lexicographical order. Assuming that a given pattern $P$ in $T$. We define the $P$ relative to the scope of the SA with respect to $T$ as $[s; e]$, such that $s$ and $e$ are respectively the rank of the lexicographically-minimum and longest suffix of $T$ that contains $P$ as a prefix. To find all occurrences of a pattern $P$ in $T$, we can first compute the SA range of $P$ and retrieve the occurrences of $P$ from the suffix array directly one by one.

**Burrows–Wheeler Transform (BWT):** The Burrows–Wheeler transform always named as BWT. It also called block-sorting compression. Basic, it is an algorithm used in data compression techniques. In 1994, Michael Burrows and David Wheeler invented it, while working at DEC Systems Research Center in Palo Alto, California. This algorithm is based on a previously unpublished transformation discovered by Wheeler in 1983 [46]. Many state-of-art sequence alignment algorithms such as BWA [15], SOAP2[61] are implemented in BWT based index algorithms.
2.5 GPU based alignment algorithms

There are some applications have been implemented based on GPU, as CUDASW++[32], GPU-BLAST[33], SOAP3[35] and so on. In CUDASW++, they had implemented the optimizing Smith-Waterman sequence database searches. The GPU-BLAST implemented the BLASTP algorithms in GPU version. They have implemented the index method for query sequence and the ungapped extension process for GPU, but the overall performance is not good: only 3-4 times of speedup can be achieved. To the best of our knowledge, we are the first group who investigated the acceleration of BLASTN on GPU [30], and we are currently working on the fully implementation of GPU-BLASTN. SOAP3 is also the first GPU-based tool for short read alignment.
Chapter 3  CUDA Memory Model

Memory management and accessibility are very crucial parts of programming languages. In High performance computing, with help of the hardware accelerators, memory management might have more significant influence on performance.

Many applications contain a large amount of data reading and writing operations, which tend to generate lots of latency. Fast data manipulation and improving the efficiency of data read-write are crucial for decreasing memory access rate, improving memory bandwidth and gaining higher performance.

As for hardware, a memory with both low latency but large capacity tends to be technically and economically impossible. Therefore, memory hierarchical structure model is widely used in today’s computer to achieve the goal of low latency and high capacity. The design principle of memory hierarchical structure model depends on temporal locality and spatial locality. Temporal locality assumes that if a certain data space is accessed, within a short time window, the data space will be visited again, while spatial locality assumes that if a certain data space is accessed, then the adjacent data space is likely to be visited at the same time. Memory hierarchical structure is composed of different levels of memory, and as we can see from the Figure 3-1, current typical model contains four levels, that are registers, cache, main memory and disk memory. The higher memorizer level has lower latency as well as lower consumption and higher bandwidth, however, the price for that is expensive and its capacity is smaller when compared to the lower memorizer level. Memory hierarchical structure enables the data with high temporal locality and spatial locality to be stored in the high level of memorizers, thereby to achieve the purpose of the large capacity and low latency.
Figure 3-1 Typical memory hierarchical structure

Memory hierarchy structure can improve the performance of the hardware; while for software, developers can optimize the application and reduce the unnecessary data read-write operations in the application and thus more performance improvements are gained through the combination of these two parts. To meet the requirements, the developer must fully understand the memory model of programming language, using the principles of locality to manage the memory efficiently, in order to obtain the minimum latency, the largest bandwidth, and the highest performance.

As a currently popular high-performance platform, CUDA has a set of specific memory model, just as shown in Figure 3-2.
There are six types of memory in CUDA memory model, including registers, shared memory, local memory, constant memory, texture memory and global memory. These six kinds of memories can be divided into two categories, on-chip memory and off-chip memory. On-chip memory, which is located in each Stream-Multiprocessor, includes register, shared memory and local memory, while off-chip memory can be visited by the threads in all Stream-Multiprocessors, includes constant memory, texture memory and global memory. In recent generations of GPU architecture (Fermi, Kepler), different memories have different properties. We can take the comparison of Fermi with Kepler as an example. Fermi architecture has 16 SM, while Kepler’s has 15 SM; the size of register grows from 32KB(Fermi) to 64KB; the configuration options of L1 cache and shared memory increase from 2(16KB-48KB,48KB-16KB) in the Fermi to 3(16KB-48KB,48KB-16KB,32KB-32KB); lastly, the size of L2 cache grow from 768KB to 1536KB and global memory grow from 6GB to 12 GB.

Every memory has a different scope of access, life cycle and function. Each
thread in Kernel function has a private local memory and it will be put on the local memory when the local variables and arrays in the thread are larger than the capacity of the data, such as dynamic array and big structure or array. The local memory and the thread have the same life cycle; each thread owns a shared memory itself. The data of the shared memory can be accessed by all the threads in the same thread block; The shared memory and the thread block also have the same life cycle. Constant memory and texture memory is read-only memories that can be accessed by all thread and they are respectively optimized according to different application scenarios. Texture memory has the functions of hardware interpolation and compulsory conversion of data types, while constant memory is used to store constant and has the features of low latency of access and broadcast. Global memory is available for all threads to read and write. The data of global memory, constant memory, and the texture memory will be continued after the completed execution in a kernel function and can be called by other kernel function in the same program, having the same life cycle as the application.

In this section, the commonly used three kinds of memory (global memory, shared memory, constant memory) are introduced in details.

### 3.1 Global memory

Global Memory is one of the biggest and the most commonly used GPU Memory; also, it has the maximum access latency. “Global” represents its scope and life cycle. Global memory is read and written by all threads in SM on the device, that is to say, its life cycle is same as the entire life cycle of the application.

Global memory includes two application modes named static state and dynamic state. Developers can use __device__ static state to apply for global variables; They can also use cudaMalloc() dynamic to allocate global memory and release those memories by cudaFree().

Global memory pointer can be used as the parameters of the kernel function to be
passed to the device. Threads from different block can’t be synchronized. When multiple threads were writing the data in the same position of the global memory at the same time, it is likely to cause conflict and will lead to uncertainty of program behavior. In this case, atomic operation must be used to maintain the correctness of the data writing operation.

The access bandwidth of global memory is 32 bytes, 64 bytes or 128 bytes. When access the global memory, data size will be automatically aligned according to the access of bandwidth, namely, the first data address must be a multiples of 32, 64 or 128.

The optimizing of global memory access behavior will greatly optimize the performance of the application. When a warp in the thread read the data from the global memory, the size of loaded data is decided by two factors: first, the memory size covered by the data involved; second, the alignment of the memory address, for example, the size of the data accessed by all threads in a warp is 60 bytes, according to the principle of memory alignment, 64 bytes of data will be read to the cache by LDST unit (Load/Store).

### 3.1.1 Shared memory

In the kernel function, the variables modified by `__shared__` are stored in shared memory. Shared memory also includes two kinds of application forms, namely static and dynamic.

Developers use “`__shared__ int sharedData[256]`” to statically allocate a shared memory that size is 256 *(int) for each thread block. When Kernel function is in call, there are four configurable parameters in total, the format is as follows:

```
kernel_name<<<grid_size, block_size, shared_memory_size, stream>>>(argument list);
```

Among them, grid size is the number of block in kernel function; block size is the number of threads in a block; the unit of the size of shared memory applied by `shared_memory_size` is byte; stream represents the flow in which kernel function is
executed; the initial value is zero and carried out in the default stream. Using dynamic way to allocate shared memory, the size of shared memory used in the kernel function is not specified. Sentence is “extern __shared__ int sharedData[]” and the size of shared memory available in the thread block is dynamically set by the third parameter of kernel function.

Shared memory is on-chip memory, and compared to global memory, shared memory has higher bandwidth and lower latency. However, each SM has limited size of shared memory. The default size of shared memory in each SM of Kepler architecture is 48 k. As is known to all, whether the number of warp in SM can be run at the same time is decided by two factors: the shared memory resource and register resources. Therefore, developer must be careful to use shared memory, excessive using of shared memory in the thread block will reduce the amount of warp SM occupancy and GPU core processing occupancy, thus will lead to performance degradation. Shared memory is allocated within the kernel function, its life cycle is the same as the thread, and when a thread blocks is performed, the shared memory it held will be released and assigned to other threads. Shared memory is basic means of communication between threads in thread block. The thread in the thread block can communicate through shared memory. “syncthreads()” function can be used to synchronize data in one thread block. This function creates barrier, it is not allowed to continue until all threads in the same thread block arrive at the barrier. The role of the barrier is to prevent potential data risks, such as dirty data, reading and writing conflict, etc. In each SM, shared memory and L1 cache share the same piece of on-chip memory and it is statically allocated, but it can be used in the following function to be configured dynamically at run time.

```c
CUDAError_tCUDAFuncSetCacheConfig(const void* func, enumCUDAFuncCacheCacheConfig);
```

The function configuration base on the partition of the memory in every kernel based piece, setting configuration which is used for configuration of kernel function specified by “func”. Take Kepler architecture as an example, it supports the cache configurations as follows:
CUDAFuncCachePreferNone: No preference (default)

CUDAFuncCachePreferShared: 48 KB shared memory and 16 KB L1 cache

CUDAFuncCachePreferL1: 48 KB L1 cache and 16 KB shared memory

CUDAFuncCachePreferEqual: the size of the Shared memory and L1 cache are both equal to 32 KB

### 3.1.2 Constant memory

Constant memory is a unique area of memory and every SM has a dedicated constant cache. The variables modified with “__constant__” are stored in constant memory. Constant memory must be stated in the global scope outside of the kernel function. The size of constant memory is limited, which can be set up to 64 KB of constant memory. Constant memory is read-only memory and can be read by threads of all kernel functions on the same GPU. Constant memory must be stated statically, when the data type and size must be stated specifically, such as “__constant__ int constant_data[256]”. For constant memory, there are two ways to get the data assigned: the first is initially assigned in statement, the second is to generate the data on the host, that is to call function cudaMemcpyToSymbol() to copy the data on the host to the constant memory.

The scenario of using constant memory is more special, when all threads of a warp read data from the same constant memory address, it will broadcast the data to threads in the same warp, the properties of constant memory is optimized in this usage mode. For example, for a mathematical formula, the coefficients get the best performance when it is stored in constant memory.

The maximum throughput in every clock cycle of constant memory is 4B and if the size of data needed by all threads in a warp is greater than 4B, then it need to be read many times. If each thread in a warp read different data in the constant memory, in this case, the constant memory is the best choice.
3.2 Test process design

3.2.1 Demand analysis

The content of this test is to test the performance of GPU under different memory access mode. The major factors to be investigated are as follows:

- Different organization forms of data, i.e., data structure
- Different ways of access
- Different memories
- Different number of threads

Through analysis, other factors to consider in the experiment include: Data storage modes (one dimensional array and two-dimensional array, etc.), data copy mode (direct copy, alignment copy, array copy, etc.), data storage location (global memory, Shared memory, constant memory), data access mode (sequential access, step access, frequency of access) and so on. Under different scenarios, the best way of access may be completely different. In order to design this experiment meaningfully, experiment designers need to know more about the mode commonly used in CUDA.

3.2.2 Experimental design

According to requirement analysis, the testing experiment is designed based on four factors, including data structures, data internal distribution, data access mode and data storage location.

3.2.2.1 Data structure

This study selects three common data structures. They are one dimensional array, two-dimensional array and tree structure. Detailed data structures are as follows.

One dimensional array

Data structure: DATA_TYPE *data;

Dyadic isometric array

Because of the particularity of GPU’s memory copy form, one dimensional array
is here used to simulate two-dimensional array.

Table 3-1 Two dimensional array

```c
struct {
    size_t width;
    size_t height;
    DATA_TYPE *imgData;
    size_t pitchBytes;
}
```

**Tree structure**

There are two representations of tree structure, linked storage and sequential storage. Obviously, it is not advisable to select linked storage in CPU because it would lead to a lot of improper Read-Write operation to data. This design chooses sequential storage as the representation of tree structure.

There are three sorts of sequential storage including Parents-representation, Child-representation and Child-sibling-representation.

Out of comprehensive consideration, the tree structure in this document unified use complete binary tree and Child-representation is chosen as store mode.

Table 3-2 Data Structure

```c
typedef struct Node_st{
    DATA_TYPE data;
    Node* left;
    Node* right;
}Node;

typedef struct
{
    Node nodes[MAXSIZE];
    int r,n;  //location of root node and number of nodes
}Tree;
```

**3.2.2.2 Data Internal Distribution**

Data Internal distribution is the characteristics of data distribution within the array.

This test selects six common distributions as follows:

1. Random distribution
2. Normal distribution
\[ f(x) = \frac{1}{\sqrt{2\pi\sigma}} \exp\left(-\frac{(x - \mu)^2}{2\sigma^2}\right) \]

(3) Poisson distribution

\[ P(x) = \frac{m^x}{x!} \times e^{-m} \]

(4) Uniform distribution

\[ Y = x \]

(5) Geometrical distribution

\[ P(X = k) = p^x(k) = (1 - p)^{k-1}p \]

(6) Exponent distribution

\[ f(x) = \begin{cases} \lambda e^{-\lambda x} & x > 0 \\ 0 & x \leq 0 \end{cases} \]

3.2.2.3 Data access mode

In the core function, thread access data in accordance with subscript. Data access mode is the law in which data access subscript. This test selects seven forms of distribution as data access mode. They are order access, step access, random access, standard normal distributed access, Poisson distributed access, geometrical distributed access and exponent distributed access. The example of sequential access and Step-2 access shows as in Figure 3-3.
3.2.2.4 Data storage location

Three common kinds of memory are selected including global memory, constant memory and shared memory.

3.2.2.5 Others

Summary of data access modes: take step two and step 4 during Step access

Size of thread block: take 256, 512, 1024

Number of data elements:

Global memory: take 1024, 4096, 10240, 40960, 102400

Shared memory: take 512, 1024, 4096, 10240

Constant memory: 512, 1024, 4096, 10240

Data type: take unsigned char

Note: Because the size of Node is equal to 24, Number of data elements of constant memory is taken 128, 512, 1024, 2048 under the condition of tree structure.

3.2.3 Testing code schema

3.2.3.1 Code directory structure

This project has eight documents in total. They are three parts (Case, Common, Distribution), a main file and a Make file.

Common.h/cu: This file contains definitions of global variables, enumerated variables, structures, and foundation class.

Distribution.h/cu: This file includes declarations and definitions of a variety of distributed data generating function

Case.h/cu: The core logic class

Main.cu: Main function

3.2.3.2 Common.h/cu

(1) Global variables

Global variables involve types of test data, range of test data (MIN, MAX) PI and e

Among them, range of test data is related to type of test data.

Table 3-3 Global variables

<table>
<thead>
<tr>
<th>Macro</th>
<th>Meaning</th>
<th>Value</th>
</tr>
</thead>
</table>

27
Definition

<table>
<thead>
<tr>
<th>DATA_TYPE</th>
<th>type of test data</th>
<th>unsigned char</th>
</tr>
</thead>
<tbody>
<tr>
<td>MIN</td>
<td>The maximum and minimum of test data are related to the type of DATATYPE</td>
<td>0</td>
</tr>
<tr>
<td>MAX</td>
<td></td>
<td>256</td>
</tr>
<tr>
<td>PI</td>
<td>circumference ratio</td>
<td>3.141592654</td>
</tr>
<tr>
<td>e</td>
<td>mathematical symbol</td>
<td>2.718281828459</td>
</tr>
</tbody>
</table>

Table 3-4 Enumerated variables

(2) Enumerated variables

Enumeration defines the type of data content distribution, the type of data access and forms of data organization.

There are six types of data content distribution. Data access includes seven types. Data organization is consist of three forms.

<table>
<thead>
<tr>
<th>Types of Enumeration</th>
<th>Meaning of types</th>
<th>Enumerated variables</th>
<th>Meaning of variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>data_content</td>
<td>Data internal distribution</td>
<td>dc_random</td>
<td>random distribution</td>
</tr>
<tr>
<td></td>
<td></td>
<td>dc_standard_normal</td>
<td>standard normal distribution</td>
</tr>
<tr>
<td></td>
<td></td>
<td>dc_poisson</td>
<td>Poisson distribution</td>
</tr>
<tr>
<td></td>
<td></td>
<td>dc_uniform</td>
<td>uniform distribution</td>
</tr>
<tr>
<td></td>
<td></td>
<td>dc_geometric</td>
<td>geometrical distribution</td>
</tr>
<tr>
<td></td>
<td></td>
<td>dc_exponential</td>
<td>exponent distribution</td>
</tr>
<tr>
<td>access_mode</td>
<td>Data access mode</td>
<td>am_sequential</td>
<td>order access</td>
</tr>
<tr>
<td></td>
<td></td>
<td>am_step</td>
<td>step access</td>
</tr>
<tr>
<td></td>
<td></td>
<td>am_random</td>
<td>random access</td>
</tr>
<tr>
<td></td>
<td></td>
<td>am_standard_normal</td>
<td>standard normal distributed access</td>
</tr>
<tr>
<td></td>
<td></td>
<td>am_poisson</td>
<td>Poisson distributed access</td>
</tr>
<tr>
<td></td>
<td></td>
<td>am_geometric</td>
<td>geometrical distributed access</td>
</tr>
<tr>
<td></td>
<td></td>
<td>am_exponential</td>
<td>exponent distributed access</td>
</tr>
<tr>
<td></td>
<td></td>
<td>am_sequential</td>
<td>order access</td>
</tr>
</tbody>
</table>
(3) **Structures and foundation class**

There are simulation of 2D structure, tree node and tree class.

Tree structure makes use of complete binary tree and store mode selects child-representation

<table>
<thead>
<tr>
<th>Name of structures</th>
<th>meaning of structures</th>
<th>Name of variables</th>
<th>meaning of variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node</td>
<td>Tree Node</td>
<td>left</td>
<td>Left child node, Node pointer type</td>
</tr>
<tr>
<td></td>
<td></td>
<td>right</td>
<td>Right child node, Node pointer type</td>
</tr>
<tr>
<td></td>
<td></td>
<td>data</td>
<td>Data in node, DATA_TYPE type</td>
</tr>
<tr>
<td>data2D_st</td>
<td>1D stimulate 2D</td>
<td>rows</td>
<td>Rows of planar array</td>
</tr>
<tr>
<td></td>
<td></td>
<td>cols</td>
<td>cols of planar array</td>
</tr>
<tr>
<td></td>
<td></td>
<td>data</td>
<td>Array pointer</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Pitch Bytes</td>
<td>After copying GPU, the number of bytes of data in each row of memory occupied</td>
</tr>
<tr>
<td>Tree</td>
<td>Tree structure</td>
<td>nodes</td>
<td>Node array</td>
</tr>
<tr>
<td></td>
<td></td>
<td>num</td>
<td>Node number</td>
</tr>
</tbody>
</table>

**3.2.3.3 Distribution.h/cu**

The distributed data and basis are generated in accordance with the type of distribution. Subscript access data is generated according to the type of data access.

In Distribution.h file, every distribution is reload three time to generate one dimensional (or two-dimensional simulation) data (DATA_TYPE* data, Subscript access data (int* data) and tree node data (Node *nodes. Realization mode is to conduct min, max, distributed parameter, size ,data pointer and generate data and valuation within function.

<table>
<thead>
<tr>
<th>Name of function</th>
<th>Meaning of function</th>
</tr>
</thead>
<tbody>
<tr>
<td>random</td>
<td>random distribution</td>
</tr>
<tr>
<td>-----------------</td>
<td>--------------------------------------</td>
</tr>
<tr>
<td>standard_normal</td>
<td>Standard normal distribution</td>
</tr>
<tr>
<td>Poisson</td>
<td>Poisson distribution</td>
</tr>
<tr>
<td>uniform</td>
<td>uniform distribution</td>
</tr>
<tr>
<td>geometric</td>
<td>geometrical distribution</td>
</tr>
<tr>
<td>exponential</td>
<td>exponent distribution</td>
</tr>
</tbody>
</table>

### 3.2.3.4 Case.h/cu

This is core organizational class. Within Case.h is Case class. Member variable and member functions are as follows.

Table 3-7 Cases

<table>
<thead>
<tr>
<th>Name of Member variable</th>
<th>Meaning of Member variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>data_form df;</td>
<td>data organization form, one dimension, two dimension, tree structure</td>
</tr>
<tr>
<td>size</td>
<td>When it is one dimension array and tree structure, member function represents the size of testing data.</td>
</tr>
<tr>
<td>r,c</td>
<td>When it is two dimension array, r represents line number, c represents row number. Size is equal to r*c</td>
</tr>
<tr>
<td>data_content dc;</td>
<td>Data content distribution</td>
</tr>
<tr>
<td>access_mode am;</td>
<td>access mode of different distribution</td>
</tr>
<tr>
<td>thread_num</td>
<td>Thread number, it is equal to data size</td>
</tr>
<tr>
<td>block_size</td>
<td>block_size</td>
</tr>
<tr>
<td>step</td>
<td>needed Variable during step access</td>
</tr>
<tr>
<td>am_num</td>
<td>Data number of every thread access</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Name of member functions</th>
<th>Meaning of member functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Case()</td>
<td>constructor, initiating all kinds of default parameters</td>
</tr>
<tr>
<td>initData();</td>
<td>According to df (data organization form) and dc (data content form), calling corresponding distribution function to generate distribution data</td>
</tr>
<tr>
<td>global_run();</td>
<td>Three operated core function, respectively getting Access Performance in different memory, it is the same internal process of function.</td>
</tr>
<tr>
<td>shared_run();</td>
<td></td>
</tr>
<tr>
<td>constant_run();</td>
<td></td>
</tr>
</tbody>
</table>

(1) In accordance with data form, applying space of device and copy data;
(2) In accordance with different access distribution mode (sequential access, step access, other distribution access), executing different core function.
During other distribution access, outside the core function, generating access subscript in host port is needed according to different access method and then copy into device port. (Although it is ok to generate some distribution by curand outside the core function, it cannot not generate am_geometric distribution and am_exponential distribution. Taking function testing conformance into consideration, curand is not used)

Inside Case.cu, it is definition of constant memory. Core function and member function of Case class are achieved.

Size of Constant memory should be determined before Compilation, so testing of different data size of constant memory should be manually changed every time.

There are 27 kinds of core function in total of three kinds of memory (global、shared、constant), three sorts of data form (One-dimensional, two-dimensional, tree structure) and three kinds of access distribution (sequential、step、Common). Form of function ‘name is data form-kind of memory-access distribution.

Global memory Internal process of core function is (1) calculate Subscript (2) Subscript bound judgment (3) data reading with the number of am_num and then add all the data together.

Shared memory Internal process of core function is (1) calculate Subscript (2) Subscript bound judgment (3) copy data into shared memory according to copy_num_per_thread. copy_num_per_thread is parameter which has been calculated in advance. (3) data reading with the number of am_num and then add all the data together. Constant memory internal process of core function is the same as Global memory.

3.3 Experimental results

3.3.1 Experimental environment

Hardware environment
Table 3-8 Hardware environment

<table>
<thead>
<tr>
<th>Project</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>Intel Core i7 2600</td>
</tr>
<tr>
<td>Memory</td>
<td>12GB DDR3 1333MHz</td>
</tr>
<tr>
<td>Hard disk</td>
<td>WD 1TB</td>
</tr>
<tr>
<td>display card</td>
<td>Tesla K40c</td>
</tr>
</tbody>
</table>

Software environment

Table 3-9 Software environment

<table>
<thead>
<tr>
<th>Project</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>operating system</td>
<td>Ubuntu 14.04 Desktop x64</td>
</tr>
<tr>
<td>CUDA Version</td>
<td>7.0</td>
</tr>
</tbody>
</table>

3.3.2 Experiment analysis

3.3.2.1 Best memory

In order to explore how memory storage location affects data access performance, we choose rational thread block size and data size, select default setting as data access mode- that is sequential access and then observe which kind of memory can get the best performance on the condition of different data internal distribution and data form. The results are shown in the following picture Figure 3-4 Figure 3-5 Figure 3-6, in which abscissas is data internal distribution and ordinate is data operation hour.
Figure 3-4 One Dimension Access

Figure 3-5 Two Dimension Access method
It can be seen from the figure that the worst performance is shared memory under the condition of sequential access mode. The reason is that during each thread, data need be copied into shared memory first before thread reads data from the shared memory, which takes a lot of time. Constant memory gets the best performance while size of memory is limited. When it comes to a large amount of data, global memory is the best choice.

3.3.2.2 Best access mode

In order to explore how data access mode affects data access performance, we choose rational thread block size and data size, select global memory as memory storage location, and then observe which kind of access mode can get the best performance on the condition of different data internal distribution and data form. The results are shown in the following picture Figure 3-7 Figure 3-8 Figure 3-9 Figure 3-10, in which abscissas is data internal distribution and ordinate is operation hour.
Figure 3-7 One dimension access method

Figure 3-8 Two dimension access method
From the above figure, it can be concluded that for performance, sequential access and step access is basically the same. Cache row of a L1 cache is 128B, sequential access mode is greatest to cache followed by step access. Thus the performance of sequential access mode is the best. Performance is roughly the same among random...
access, Poisson distribution access, geometric distribution access and exponential distribution access. While the access sequences generated by standard normal distribution is in large-span, which is on the bad side of cache. So its performance is the worst.

3.3.2.3 Best data structure

In order to explore how data structures affect data access performance, we choose rational thread block size and data size, select global memory as memory storage location, and then observe which kind of memory can get the best performance on the condition of different data internal distribution and data access mode. The results are shown in the following picture Figure 3-11 Figure 3-12 Figure 3-13 Figure 3-14 Figure 3-15 Figure 3-16, in which abscissas is data internal distribution and ordinate is operation hours.

![Exponential data structure](image)

Figure 3-11 Exponential data structure
Figure 3-12 Geometric data structure

Figure 3-13 Poisson data structure
Figure 3-14 Random data structure

Figure 3-15 Standard normal data structure
3.3.2.4 Best data content

In order to explore how data inter distribution affects data access performance, we choose rational thread block size and data size, select global memory as memory storage location, and then observe which kind of memory can get the best performance on the condition of different data access mode and data form. The results are shown in the following picture Figure 3-17 Figure 3-18 Figure 3-19, in which abscissas is data access mode and ordinate is operation hours.
Figure 3-17 One dimensional array
From the above figure, it can be concluded that for different data content, generated data is almost no impact on access mode and data organization has almost no influence on performance as well.
### 3.3.3 Multi-access with global and shared memory I/O

In this section, we will discuss multi-access with global memory and shared memory in one thread. We should know the latency of the I/O with different read options.

**Test options:**
- **Summary of data access modes:** take step two and step 4 during Step access
- **Size of thread block:** take 128, 256, 512, 1024
- **Data structure:** random distribution one-dimensional array
- **Access method:** random, normal distribution
- **Data size:** 1k, 4k, 10k, 40k
- **Data type:** take unsigned char

**Experimental result**

Explore the time spent on copying data from the global memory to the Shared memory, pseudo code is as follows

**Algorithm premise:** assuming that all the parameters meet the conditions for the convenience of description

**Input:** input data, the number of data each thread access, the size of input data, access index data, the number of data copied by each thread from global memory to shared memory

**Output:** output data, start time and end time of copying data to the Shared memory

```c
index <- blockIdx.x * blockDim.x + threadIdx.x;  // Get the thread index

extern __shared__ DATA_TYPE sharedData[];   // Declare the shared memory

clock_t start_time <- clock();                  // Record the start time

for copt_index <- index to copy_num_per_thread+index
    sharedData[copt_index] <- data1D[copt_index];

clock_t end_time = clock();                    // Record the end time

d_start_end_time[index * 2 + 1] <- end_time;   // Synchronization of the
```
thread in the thread block

```c
    d_start_end_time[index * 2] <- start_time   // Store the start time
    __syncthreads();                           // Store the end time
```

Figure 3-20 Random access 1k memory

Figure 3-21 Random Access 4K memory

Figure 3-22 Random Access 10k Memory
Figure 3-23 Random access 40k Memory

Figure 3-24 Standard Normal access 1k Memory

Figure 3-25 Standard normal access 4k memory
Explore the time spent in accessing data of the shared memory, pseudo code is as follows

Algorithm premise: assuming that all the parameters meet the conditions for the convenience of description

Input: input data, the number of data each thread access, the size of input data, access index data, the number of data copied by each thread from global memory to shared memory

Output: the output data, the start time and end time of accessing to the data of Shared memory

```c
index<-blockIdx.x*blockDim.x+threadIdx.x; //Get the thread index
extern __shared__ DATA_TYPE sharedData[]; // Declare the shared memory
```
```cpp
__syncthreads(); // Synchronization of the thread in the thread block
clock_t start_time = clock(); // Record the start time
for (i <- 0 to am_num)
    clock_end_time = clock(); // Record the end time
dev_out[index] += sharedData[am_data[(index + i)%size] % size];
d_start_end_time[index * 2] <- start_time; // Store the start time
d_start_end_time[index * 2 + 1] <- end_time; // Store the end time
```

![Figure 3-28 Thread access clock](image)

From the Figure 3-28, it shows inside each warp, threads execute a set of commands at the same time according to the amount of hardware above the SM, so that you can perform data at the same time. So when designing parallel algorithms, we need to perform a number of commands to design the structure of the algorithm. Threads in one warp start and end at almost the same clock.

### 3.3.4 GPU memory model

For the study of GPU operation under the above sections of the memory layout of the different threads and different ways to access the situation, because the memory access often become IO bottleneck, so the computation and memory access unify design new memory access model.
Each warp as shown in the form of pipeline execution is performed on the SM, when accessing the memory becomes a bottleneck, will be defined according to the execution memory SM on the implementation of memory performance.

In parallel computing, usually considered respectively to calculate the theoretical performance, on the hand is to consider the IO access bandwidth, treats IO and computing from separate namespace independently. Based on the study, if all the execution unit work independent, we can unify the IO and computing command as same type of command. Two new conceptions:

1. IO and computing unify as one type of command
2. For Memory intensive algorithm, optimization Bandwidth as the target.

Define a new model based on memory access

1. Define the \( S = f_s(command) \) as the memory size of one execution unit operation.
2. Define \( T = f_t(command) \) as the time of one execution unit, which
*command* is the commands as IO or computing operations.

3. Parallel computing, there are many warp execution at the same time in CUDA platform.
   a. Define the number of execution unit as \( N_w \)

4. The Bandwidth of parallel computing is
   \[
   B = N_w \times \frac{S}{T} = N_w \times \frac{f_s(command)}{f_t(command)}
   \]

Because of current operations have computing operation and memory operation. So continue to use those commands, define the \( command_m \) as memory operation command, \( command_c \) as computing operation command

\[
T = f_t(command) = f(command_m) + f(command_c)
\]

For different memory have different operation time:

\[
f_t(command) = \begin{cases} 
  f_{t,\text{register}}(command) & \approx 1 \text{ cycle} \\
  f_{t,\text{shared}}(command) & \approx 1 \ldots 32 \text{ cycle} \\
  f_{t,\text{global}}(command) & \approx 500 \ldots 32 \times 500 \text{ cycle}
\end{cases}
\]

Because of current operations have computing operation and memory operation. So continue to use those commands, define the \( command_m \) as memory operation command, \( command_c \) as computing operation command

\[
T = f_t(command) = f(command_m) + f(command_c)
\]

For different memory have different operation time:

\[
f_t(command) = \begin{cases} 
  f_{t,\text{register}}(command) & \approx 1 \text{ cycle} \\
  f_{t,\text{shared}}(command) & \approx 1 \ldots 32 \text{ cycle} \\
  f_{t,\text{global}}(command) & \approx 500 \ldots 32 \times 500 \text{ cycle}
\end{cases}
\]

According to the memory model, parallel algorithm design avoids respectively to computing and IO, we can unify the computing and IO.
Chapter 4  GPU Accelerated SOAP3

Among the many CPU based short read alignment algorithms, SOAP2 has been shown to be one of the fastest. However, to resequence a human genome with 20X coverage, it still takes more than 13 days to just complete the alignment step. To obtain a drastic improvement in speed, we leverage the huge number of multiprocessors in GPU and develop a GPU version of SOAP2 together with our collaborators. The new alignment engine is named SOAP3 and it can achieve a speedup of 10, 22, and 40 times over SOAP2 when aligning with one, two, and three mismatches, respectively. In particular, aligning one million reads of length 100 onto the human genome with three mismatches can be done in only 46 seconds (and 15 seconds for 2 mismatches). In other words, only 7 hours are needed to align the 600 million reads in the process of resequencing a human genome. In the following, we introduce the major designs and techniques we deployed in SOAP3.

4.1 Background of SOAP3

4.1.1 Suffix string

Given an alphabet $\Sigma$, of with $x$, $y$, $z$ is the string, and $x = yz$, we call $y$ is the prefix of $x$, $z$ is the suffix of $x$. If we regard a text as a long string, then a certain suffix is the string contained from a particular location to the end of the text. If you want to find the pattern $P$, we can find the corresponding text all begin with suffix $P$.

4.1.2 The Trie structure

Trie is a tree structure and it can save a set of strings. If we want to find the pattern $P$, we just need the time complexity of $O(|P|)$, and has nothing to do with how much strings in Trie. In the Trie structure, each leaf node represents a string; an internal node corresponding to one or more string prefix, there is a line tagged as character $C$ point from the node represented by $s$ to the node represented by $Sc$. When
finding the Trie structure, we start from the root node and screen each character to find according to the character of the pattern \( P \), if the lookup finally reach a leaf node, then we find the string; If the lookup cannot continue in a node, then there is no model to find the string the collection; if the lookup end in a certain node, then the search mode is the prefix of one or some of the string. We can easily determine all string which use mode \( P \) as prefix in the collection, and according to the above process, we finally reached an internal node, then the string sets represented by the subtree which use that node as root all use pattern \( p \) as suffix. If there is no branch in the path from one node to the leaf nodes, we can suppress all node to leaf node to save space. We can use all the suffixes of the text to make up string collection to build the Trie structure, and add a character $ at the end of the each suffix, which makes no suffix is the prefix of another suffix, thus to establish the Trie structure that each leaf node corresponding to an only suffix string. This Trie structure is called suffix Trie. Suffix tree, varying from the suffix Trie, compress the path of the single branch into a node, so as to achieve the purpose of saving space, like Figure 4-1. Compare with suffix Trie, suffix tree reduces the space occupied, but the space usage is still too large for practical application, and its characteristics of random access is not suitable for preservation in auxiliary storage. Therefore, in general, the suffix tree only has theoretical significance for large-scale text queries.

![Figure 4-1 Suffix Tree](image-url)
4.1.3 The suffix array

Suffix array made up for the deficiency of suffix tree structure and it is an effective data structure used to find large text. Suffix array contains only orderly suffix pointers; each suffix pointers points to a textual suffix and all of the suffixes are arranged according to the dictionary.

The basic concepts of suffix array

The suffix array SA, a one dimensional array, holds a certain rank that $SA[1]$, $SA[2]$, ..., $SA[n]$, and ensure that $Suffix(SA[i]) < Suffix(SA[i+1])$, $1 \leq i < n$. That is to arrange $n$ suffixes of $S$ which from small to large and put the beginning of the sorted suffixes into $SA$ in sequence.

Among them, the suffix $(I)$ represent the string $s[i,i+1...n-1]$, that is the string $S$ began in the suffix of the $i$th character.

What array rank $[i]$ saves are the ranks expanding from small to large in all suffixes. It is easy to see, suffix array and rank array are inter-reverse operation for each other.

Height array: Define the longest common prefix of the $height[i] = suffix(SA[i-1])$ and $suffix(SA[i])$, which is also the longest common prefix of two suffix ranking of adjacent.

$h[i]=height[rank[i]]$ is the longest common prefix of suffix $(i)$ and the suffix of its former.

$LCP(i,j)$: For positive integer $i, j$, $LCP(i,j) = lcp(Suffix(SA[i]), Suffix(SA[j]))$, of which $i, j$ are integers among 1 to $n$. $LCP(i,j)$is the length of the longest common prefix of the $i$th a and the $j$th suffix e sin the suffix array. There into, function $lcp(u,v) = max\{|i|u=v\}$, that is to compare the corresponding characters of $u$ and $v$ in sequence from the beginning, and the largest position where corresponding characters continue to be equal is known as the longest common prefix of these two strings.

Some properties

$(1)LCP(i,j) = min\{height[k]|i+1 \leq k \leq j\}$, that is, calculate the $LCP(i,j)$ is equivalent to enquiry the minimum value of all elements in which subscripts are in the scope of
from $i+1$ to $j$.

(2) For $i>1$ and $Rank[i]>1$, there must be $h[i] \geq h[i-1]-1$.

### 4.1.4 Introduction to BWT algorithm

It is usual to compress storage of index in full-text search. If text were reversibly transformed before compressing, it will be easier to be suppressed. BWT is such a kind of transformation. In 1994, Michael Burrows and David Wheeler put forward a kind of brand-new universal data compression algorithm called Burrows-Wheeler Transformation in the article A Block-sorting Lossless Data Compression Algorithm.

The BWT algorithm designed by Burrows and Wheeler is completely different from the design idea of all previous general compression algorithms. The relatively famous compression algorithms now available process data flow model and it reads one or more bytes at a time. BWT makes it possible to process clumpy data. The core idea of this algorithm is to sort and transform the character matrix resulting from character rotation. Considering in common text applications such as in English text string ‘the’ is frequently used and after a BW transformation, all t will be moved to the last and together. Better compression ratio will be received if the transformed string is compressed by general statistical compression model, Huffman coding, LZ algorithm, PPM algorithm). Arrange these migratory strings according to the dictionary sequence through the cyclic shift of the original string. The last character in the string is printed in sequence. At the same time, the position of the original sequence in the collating sequence is printed. For example, string $GATATA$, as it is shown below:
Figure 4-2 BWT

The algorithm description is as follows:

<table>
<thead>
<tr>
<th>function construct_BWT (string l)</th>
</tr>
</thead>
<tbody>
<tr>
<td>create a table by moving the string l</td>
</tr>
<tr>
<td>rows are all possible rotations of l</td>
</tr>
<tr>
<td>sort rows by alphabetically</td>
</tr>
<tr>
<td>return (last column of the table)</td>
</tr>
</tbody>
</table>

4.1.5 FM-index

FM-Index (Full—text index which occupies Minute start Pace) is a compressive query index designed by Paolo Ferragina and Giovanni Manzini. It is generated by an algorithm combining compression with indexing. This algorithm can be used as a compressed tool like common compression software. Compressed document generated by it can be used as index for information retrieval. To count and locate the pattern of source document, users only need to check a small part of compressed document. It just takes seconds to inquire a few megabytes of document.

4.1.5.1 The application of FM-index

FM-Index combine BWA (Burrows-Wheeler) compression algorithm with suffix array. Compressive data generated by the combination is a suffix array in certain order as well. It supports the following two basic operations.

Counting: calculate the number of occurrences of given pattern in the source
document. The time complexity of counting does not change with the change of source document’s size.

Locating: Locating operation return to the location where source document is located in. The time required for the operation is related to the size of the source document and was influenced by factor. Factor is a definite constant when building index. Its size is decided by user. It allows to trade space for time. The larger the factor is, the larger the compressed document is, the shorter query time is.

The above two operations only need to release a small part of compressed document. (generally a few of kilobytes), which takes seconds. But the time required for locating operation is also related to with the occurrence frequency of pattern in the source document. The more patterns appear, the longer the time required for locating operation.

4.2 How FM-index build index

Sometime, full-text indexing structure is called opportunistic indexing structure. That is because its size of space occupation depends on the compressibility of text used for building index. Index structure of text which is easily compressed occupies small space while ones difficultly compressed occupies large space. Besides, the query performance of index structure does not decline significantly after compression. During the BWT transformation process, the probability matrix, $M$, can be regard that it is generated by all suffix of text in accordance with the order of the dictionary. FM-index implements rapid query about text by making use of this feature. The index structure of FM-index is consisted of two parts. One part is BWT transform text, $T$, after compression. The other one is assisted text information. One part of assisted information saved is used to implement the calculation of $OCC(c, 1, k)$ function within constant time. $OCC(c, 1, k)$ function means that the number of character $C$ appears in BWT transform text $T[1..k]$. The other part is used to mark specific line of matrix $M$ to implement locating operation in the process of searching. Ferragina and Manzini give a theoretic marking method which is specifically referred in their related thesis.

There are two typical operation of String pattern matching including counting and
locating. Counting is to determine the occurrence frequency of substring in the text. Locating is to determine the location of substrings that all found in the text. These two operations can be implements in BWT transform, L. To search substring p in the text, user only need to match the head of all suffix in the text T. that is because a specific location of text T exclusively determines a suffix beginning from this location. From the process of BW transformation, transformation matrix can be regard that it is generated by all suffix of text in accordance with the order of the dictionary. In matrix M, all lines beginning with the same substring are located in continuous location. We can use startP and endP respectively showing the starting and the end location of continuous zone. endP-startP+1 means the number of lines beginning with that substring and then user can determine the occurrence frequency of substring in the text. Specific algorithm is showed as follows:

<table>
<thead>
<tr>
<th>Algorithm count(L[1..p])</th>
</tr>
</thead>
<tbody>
<tr>
<td>c = L[p], i=p;</td>
</tr>
<tr>
<td>startP = C[c]+1, endP=C[c+1];</td>
</tr>
<tr>
<td>while((startP≤endP)and(i≥2)) do</td>
</tr>
<tr>
<td>C = L[i-1];</td>
</tr>
<tr>
<td>startP = C[c]+OCC(c, 1, startP-1)+1;</td>
</tr>
<tr>
<td>endP = C[c]+OCC(c, 1, endP);</td>
</tr>
<tr>
<td>i=i-1;</td>
</tr>
<tr>
<td>if(endP &lt; startP) then return “not found”</td>
</tr>
<tr>
<td>else return “found (endP-startP+1) occurrences”</td>
</tr>
</tbody>
</table>

On above algorithm, array c means the location of character in F. Function OCC(c, 1, k) shows the occurrence frequency of character C in L[1..k]. startP and endP respectively show the first and the last location beginning with L[i..p] in M. At the beginning of the algorithm, first of all, make the first evaluate toward startP and endP. Then at every step of circulation, update the value of startP and endP to make them point to new location. At the first step, startP and endP respectively point to the first line and the last line beginning with L[i..p] in M. For startP, OCC(c, 1, startP-1) shows
the number of lines which are above \textit{startP} and end with \( L[i-1] \) as well. In other words, it shows the number of line beginning with \( L[i-1] \) but without \( L[i, p] \) after rotation. Characters in \( L \) are ordered correspondence with Characters in \( F \). At the step of \( i-1 \), we find the start line of charter \( L[i-1] \) through \( C[c]+1 \), then skip lines without \( L[i, p] \) and finally get the location of first line beginning with \( L[i, p] \). The value of \( \text{endP} \) can be calculated and the final value of \( \text{startP} \) and \( \text{endP} \) can be got as well by iterating for p-1 times.

For locating operation, first step, mark original text \( T \) with certain rules. If the location searched has been marked, then return to position value. Otherwise, search a former charter by recursion until location has been marked. If the number of step is R, the true location is \((t+r)\). Specific algorithm is showed as follows:

<table>
<thead>
<tr>
<th>Algorithm locate(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( t=s ), ( r=0; )</td>
</tr>
<tr>
<td>while row ( M[t] ) is not marked do</td>
</tr>
<tr>
<td>( c=L[t]; )</td>
</tr>
<tr>
<td>( t=-C[c]+OCC(c, 1, t-1)+1; )</td>
</tr>
<tr>
<td>( r=r+l; )</td>
</tr>
<tr>
<td>return ((t+r));</td>
</tr>
</tbody>
</table>

The above algorithm is a process of iterative search. If line \( M[t] \) is not marked, calculate the location of former charter at the step of three and four and add the number of iteration at the fifth line. Repeat the process until a marked line is find and then return to \((t+r)\) as the right location.

On the basis of previous experiment, in order to explore memory copy and data access time when using shared memory in more detail, the following experiment is designed.

4.2 \textbf{Reduce memory access}

In order to reduce memory access retrieval \textit{Precede}, we redesigned assistant data
structure two levels of samples, not with a simple single grade sampling. By selecting the right sampling rate, the SOAP3 space between the needs and memory access to reach a good balance. The implementation of the SOAP3, establish the auxiliary array of 128th for Precede function. The space is needed about 0.375 GB for the human genome. With the new design, the value of the retrieval Precede, only need a 32-bit memory visit auxiliary arrays and a 128-bit memory access BWT, half of the SOAP2 access operations. 128 sampling rate is beautiful clouds, designated choice, to limit the memory access to 128-bit, this is the memory of the width of the largest I/O operation support on Single GPU instructions. 128-bit data is read two 64bit words as a carrier to use of GPU 64bit operation of the primary support, including popcount. Population count (popcount) count 1, a string of, is a process..

Looking back, support electronic for approximate match, we need support retrieval Precede(i, c) for all character c efficiently. The new auxiliary array also accelerates the operation. Rather than visit the auxiliary array four times, we only need to visit once. All the data in one group, which is also in one segment (the biggest size of segment of GPU memory is 128bit), so it can search a 128 bit memory read in one instructions. So forward search need a lot of memory access backward search.

4.3 Coalescing memory accesses

Global memory access time is very sensitive memory access mode. The coalesced access is faster than without. For 128bit memory access, each of the thread access to their own position, the access operation can be done in one instruction.

For coalesced load the query sequences, the threads in a warp can read each of the words of one query at the same time. Fox example, there is a query Q1, which have more than 100bps reads. One warp of threads have 32 threads. Let $w_{i,j}$ denote the $j$-th word of the $i$-th read in the group ($1 \leq i \leq 32$). We store the words in the global memory in this order: $w_{1,1}, w_{1,2}, \ldots, w_{1,32}, w_{1,33}, w_{1,34}, \ldots, w_{2,1}, w_{2,2}, w_{2,3}, \ldots$. When the threads in a warp simultaneously access, say, the first words of the reads (i.e., $w_{1,1}, \ldots, w_{1,32}$), the memory locations accessed form a contiguous
128-byte segment. Then transform the data into shared memory. In this access model, we don’t need preprocess for the query sequences. This is different from SOAP3’s implementation.

4.4 **Reduce branching effect**

In practical, in each query sequence has different DNA data. Each thread process the data will go different branch. For some of the query will go more steps than others. So, we give a threshold steps value, when the branch steps bigger than the value, the process will exit and remand. For example, use terminate threshold value $1 = 8$ and $2 = 2048$, aimed at a length-100 read 1 don't match at 3.8657 seconds, and not terminate 9.7895 seconds.

4.5 **The division of the kernel**

Deal with different situations. First there are including exact match, 1 mismatch, 2 mismatch, or other mismatch. Second the respective branches are not the same. So in the implementation, we implement different situations into different kernels. It gets better than on the whole in one kernel.

4.6 **Experimental results**

Testing was performed on a 2.8 GHz 4-core machine with 16 GB main memory with one core used. The GPU card (model: NVIDIA Tesla C2070) was installed with 6 GB global memory and 448 processors. As shown in the Table 4-1, SOAP3 requires total 1.85 seconds to perform exact-match alignment for one million length-100 reads while SOAP2 requires 17.02 seconds. For 1-mismatch and 2-mismatch alignments, SOAP3 requires 3.87 and 14.92 seconds, while SOAP2 requires 42.04 and 329.58 seconds respectively. SOAP2 does not support alignments with 3 mismatches but the projected time is over 2000 seconds, while SOAP3 requires 45.82 seconds. The speed up of SOAP3 compared with SOAP2 ranges from 9 to 40.
Table 4-1 Time of align 1 million reads of length 100 against the human genome in seconds

<table>
<thead>
<tr>
<th>Type</th>
<th>GPU-BWT</th>
<th>SOAP2</th>
<th>Speed-up</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exact</td>
<td>1.85</td>
<td>17.02</td>
<td>9.21</td>
</tr>
<tr>
<td>1-mismatch</td>
<td>3.87</td>
<td>42.04</td>
<td>10.88</td>
</tr>
<tr>
<td>2-mismatch</td>
<td>14.92</td>
<td>329.58</td>
<td>22.1</td>
</tr>
<tr>
<td>3-mismatch</td>
<td>45.82</td>
<td>(&gt; 2000 ?)</td>
<td>(&gt; 40 ?)</td>
</tr>
</tbody>
</table>
Chapter 5  G-BLASTN: accelerating nucleotide alignment by graphics processors

5.1 Introduction

BLAST is one of the most fundamental software tools in bioinformatics for matching biological sequences [50] [72]. Due to the explosive growth of sequence data, improving the speed of BLAST has become increasingly critical. In the last decade, many attempts have been made to design and develop new BLAST software tools for specific hardware [73] [74] [75] or even parallel supercomputers [76]. Unfortunately, most researchers do not have access to these hardware platforms. Following the popularity of multicore processors, several BLAST software tools using multiple CPU cores for increased speed have been developed. One good example is the widely used National Center for Biotechnology Information (NCBI) BLAST, which supports multithreading in the preliminary stage of the BLAST algorithm [77]. Our experiments on a server with two quad-core Intel Xeon CPUs show that the multithreaded NCBI-BLAST can achieve an average speedup of 3~4X over the sequential version. NCBI-BLAST also supports an indexed Mega-BLAST module, which uses the database index to achieve an approximate speedup of 2~4X [78]. PLAST is a parallel implementation of BLAST [79] that applies a new indexing technique together with SSE instructions and multithreading to achieve better alignment speed. At present, PLAST only supports protein sequence alignment.

In recent years, Graphics Processing Units (GPUs) have been widely accepted as low-cost, high-performance computing platforms [80]. Compared with traditional multi-core CPUs, GPUs have much higher computational horsepower and memory bandwidth. Many bioinformatics tools have been accelerated by GPUs in recent years [81] [82] [83] [84] [85] [86]. The significant difference between GPU and CPU architectures has created many challenges in developing highly efficient GPU software [87]. Without the development of carefully designed parallel algorithms and
sophisticated optimizations, the huge potential of GPUs may not be fully realized.

Some GPU-based software tools have been developed for protein sequence alignment. Ling’s GPU-based BLAST software can achieve a speedup of 1.7–2.7X, compared with NCBI-BLAST [88]. Recently, Vouzis and Sahinidis developed GPU-BLAST, which can typically achieve acceleration speedup of 3–4X relative to the sequential NCBI-BLAST [89]. The major advantage of GPU-BLAST is that it can produce the same results as NCBI-BLAST. Remarkably, this work was one of the top-ten highest downloaded articles published in Bioinformatics in 2011.

To the best of our knowledge, we are the first to provide an open-source GPU solution, namely G-BLASTN, for nucleotide sequence alignment that can produce the same results as NCBI-BLAST. G-BLASTN is developed on top of the NCBI-BLAST source code. It currently supports the megablast and blastn modes of NCBI-BLAST. Hereafter, we use BLASTN to refer to the nucleotide blast module of NCBI-BLAST, for simplicity. The major idea behind G-BLASTN is to store a small hash table in the fast GPU cache memory and then scan the DNA database in parallel using all of the available GPU cores. We have overcome several challenges to fully use the GPU horsepower. To achieve significant speedup, some other parts of BLASTN have also been optimized. We evaluate G-BLASTN’s performance by running a set of experiments on human and mouse genome databases, as well as a partial of the NCBI nucleotide collection (nt) database. Using a contemporary NVIDIA GTX780 GPU with a cost of $650, G-BLASTN under megablast mode can achieve significant speedups over the multithreaded BLASTN running on 4-core or 8-core CPUs. When running under the more sensitive blastn mode, G-BLASTN also achieves reasonable speedups. When processing a batch of queries, G-BLASTN supports a pipeline mode that can further improve the performance by up to 44%. We believe that G-BLASTN is an attractive and cost-effective option for bioinformatics practitioners.

The remainder of the article is organized as follows. In Section 2, we briefly review the main algorithms of BLASTN and present our design for G-BLASTN. In Section 5.3, we present the detailed implementation of G-BLASTN. In Section 5.4, we present the experimental results. We conclude the article in Section 5.5.
5.2 BLASTN algorithms

BLASTN is designed to efficiently search nucleotide databases using a nucleotide query sequence (Camacho et al., 2009). BLASTN’s high level pseudocode is given in Figure 1, which consists of four stages. The setup stage prepares search options, reads and prepares the query sequence and database sequence and builds the lookup table. The scanning stage performs a preliminary search comprising three steps: seeding, ungapped extensions and gapped extensions. The seeding step scans the database for hits (i.e., a match with some word in the lookup table). The hits are then extended by ungapped alignment. The alignments that exceed a threshold score will go through the gapped extensions. Only the gapped alignments that exceed another threshold score will be saved as “preliminary” matches. The trace-back stage takes the preliminary matches as input, considering ambiguous nucleotides, and finds the locations of insertions and deletions. The output stage displays the alignment results to the user.

Table 5-1 High Level Pseudocode of BLASTN

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><strong>[Setup]</strong> prepare the BLASTN options, query, database, lookup table</td>
</tr>
<tr>
<td>2</td>
<td><strong>[Scanning]</strong> for each of ( N ) threads {</td>
</tr>
<tr>
<td>2.1</td>
<td>while the database still has unsearched sequences {</td>
</tr>
<tr>
<td>2.2</td>
<td>Retrieve a group of sequences from the database</td>
</tr>
<tr>
<td>2.3</td>
<td>Seeding: find exact word matches</td>
</tr>
<tr>
<td>2.4</td>
<td>Ungapped extensions</td>
</tr>
<tr>
<td>2.5</td>
<td>Gapped extensions</td>
</tr>
<tr>
<td>2.6</td>
<td>}</td>
</tr>
<tr>
<td>2.7</td>
<td>}</td>
</tr>
<tr>
<td>3</td>
<td><strong>[Trace-back]</strong> for each database sequence containing alignments, perform trace-back</td>
</tr>
<tr>
<td>4</td>
<td><strong>[Output]</strong> print the alignment results</td>
</tr>
</tbody>
</table>

BLASTN’s efficiency relies on the assumption that any alignment of interest between the query and the database will contain at least one W-gram (i.e., a
subsequence of length W), where W is a parameter known as “BLASTN word size”. In practice, for any given query sequence BLASTN will construct a lookup table that stores the offsets into the query where each possible w-gram occurs, where w is a parameter known as the “lookup table word size” which is less than or equal to W. Because each letter can be one of {A, C, G, T}, the lookup table has 4w entries. The seeding procedure walks through the database sequence to find hits. In each round, it fetches a w-gram, calculates its hash value, looks into the lookup table and records all matched offset pairs (i.e., the pair of offsets of the matched w-gram in the query and database, respectively) if there are any. When W is larger than w, it is not necessary to scan the database letter by letter; instead, BLASTN scans the database in strides. The maximum stride size without missing any match is W-w+1. For extremely long nucleotide databases, the seeding procedure is usually the most time-consuming step.

After all w-gram hits have been found, we must determine whether each hit belongs to a W-gram match. This is done through the mini-extension procedure (a.k.a. exact match extension), which extends each w-gram in both the left and right directions to check the existence of exact W-gram matches. The mini-extension step can be time consuming if millions of w-gram hits must be extended. Once we find all of the W-gram hits, the ungapped extension step begins, allowing for mismatches. Ungapped alignments that exceed a threshold score are stored for gapped extension. In the scanning stage, gapped extension returns only the score and extent of the alignment while the number and position of insertions, deletions and matching letters are not stored. During the whole scanning stage, BLASTN processes the sequence in NCBI-NA2 format, in which each nucleic acid is represented by two bits. Hence, ambiguities cannot be handled. In the trace-back stage, ambiguous nucleotides are restored by converting NCBI-NA2 format into NCBI-NA8, and more sensitive heuristic parameters are used for the final gapped alignment. Finally, the output step formats the results according to the user options and prints the results for the user.
5.3 First method of G-BLASN

BLAST has been changed by several versions from 1990, such as the basic version, Megablast, indexed Megablast and discontiguous-Megablast. BLASTN is one important program of the BLAST family which can search a DNA sequence (i.e., the query sequence) in a DNA database (i.e., the subject sequences). In the current version, the default algorithm used by BLASTN is Megablast.

We had done some preliminary research on accelerating the scanning process of BLASTN by GPU. First we had analyzed the hotspots of BLASTN. The test results show that, for smaller reference database, the time is more dispersed and the main hotspots are not obvious; but when the database reaches a certain size, computational hotspots are mainly concentrated in the s_SeqDBMapNA2ToNA8 function and s_BlastSmallNaScanSubject_8_1Mod4 function (hereinafter referred to as SCAN function). The former is used for library encoding extension of the matching process to convert function, while the latter is the core of BLASTN, an exact seed match function. Using Human_ref_contig reference as an example, the running time of these two functions accounted for more than 74% of the total running time.

S_SeqDBMapNA2ToNA8 function feature is only SCAN function includes typical BLAST match the calculation process, so in this study, we choose the SCAN function for parallel transformation.

SCAN function is scanning cross through the reference database (hereinafter referred to as the SUBJECT), block by block match with the query sequence (hereinafter referred to as QUERY) accurate, when hit (HIT), QUERY and SUBJECT the current offset the position information (OFFSET_PAIR,) added to the list of records to the subsequent extension of the pair.

The data access pattern of SCAN function is shown in Figure 5-1. Under normal circumstances, SCAN function to the offset of the 84 characters for the step cycle traverse SUBJECT. The number of bytes processed per cycle for 21 charactors. Each cycle of four steps called base_0 base_1 base_2 and base_3. Each step, SCAN HASH operator from SUBJECT block of data to extract two or three bytes, the KEY value to
be generated in the QUERY sequence then look up in HASH table for matching. Between each cycle and each step there is no data related, for the parallel transformation of the process provides a convenient.

![Figure 5-1 SCAN function data access pattern](image)

In parallel programing there are two ways to implement the SCAN function:

1) Parallel between the cycle, each cycle as a separate task parallel execution;

2) the base between the parallel, each processing step (base_0 . .3) as a separate task performed in parallel.

We choose the second method to implement. This is because the base between the parallel can calculate the distribution to the more threads, more parallel speedup in CUDA programing model.

### 5.3.1 The main parameters for CUDA implementation

1) Number of threads (threads): previously known, the SCAN algorithm, each base has 21 bytes for the cell block sequence, reaching the length of exit. Therefore, the thread number of the SCAN is equal to the SUBJECT length (hereinafter referred to SUBJ_LEN) / 21.

2) Thread block (block) number: each thread block consists of 256 threads, the block number for SUBJ_LEN/21/256.

In theory, assume that the CPU and GPU computing unit byte computing power equivalent to the program SCAN process can get the maximum equivalent to the
number of threads (SUBJ_LEN/21) relative speedup. However, because of the GPU kernel calls, GPU memory access violation delay consumption of thread management, and data aggregation time-consuming cost, the actual acceleration a larger gap than with the theoretical value.

In order to avoid disk I/O's influence, we also applied to optimize the transformation of pre-read 2.2 hot spot analysis of library data.

![Figure 5-2 Speed up with CUDA version](image)

### 5.3.2 Conclusions from this research

1) When the average SUBJECT sequence length is long, the GPU version of the SCAN module has a better computational performance than the CPU version, and with the average length increases, the speedup increasing the maximum speed up to 35 times (human_genomic reference).

2) SUBJECT average length of the reference is short, the GPU version of the performance degradation, even lower than the CPU version. This is the SCAN function each call can only deal with the a SUBJECT sequence, the main program every time call a CUDA function must initialize and exit the GPU core, so when the library contains many SUBJECT sequence, the kernel out of the overhead will occupy a high percentage, greatly reduces the performance of the GPU version.

3) When dealing with longer QUERY sequence, SCAN function achieves better speedup, but the increase is not obvious. This is because no matter how long
the query sequence is, BLASTN after the transformation will only occur one memory copy from CPU memory to GPU memory. In the case of processing a large number of query sequences, the time consumption of the impact on performance can be ignored.

Even without considering the other software in the BLAST package, BLASTN is still a very complex system. We found the following challenging questions to achieve a good acceleration of BLASTN by GPU:

1) GPU embedded optimization

The current implementation of the SCAN function as the smallest unit, ie, each scanning a SUBJECT sequence of copies of time-series data to the GPU, and call a CUDA kernel function. In the case of multiple SUBJECT sequence will be processed, which led to the entry of unnecessary kernel exit consumption and memory copy operations. In future work, we consider the time to deal with all reference sequences in the GPU memory, which requires that at first one-time read reference sequence into the GPUs, and the other will be renovated and transplantation more code.

2) GPU memory access optimization

Each thread may find matches (HIT) program between each thread and block statistics will be the summary of HIT. HIT is sometimes a very large number of Share Memory capacity may be unable to meet their needs, so we directly use the Global Memory to store all of HIT. Global Memory existence of access delay and other issues, the program's performance is also affected. Future we will be in-depth analysis of HIT memory needs to make full use of the Share Memory and other high-speed storage, in order to improve memory access performance.

3) Other modules GPU parallel transformation

By the hotspots analysis show, the SCAN function is the largest hotspot, but it only takes up part of the total CPU time such as human genomic libraries account for only about 20%. In fact, the BLAST algorithm, there are other high computational load of the module, to obtain the overall performance significantly improved, the need to try to optimize the hot module. The current problem is that the BLASTN both for parallel transformation of the module, there are also a number of complex serial
processing module, including algorithms and data streams, and inclusion in the parallelization process. If these serial process of transplantation to the GPU since the GPU's inadequate in dealing with complex statements will greatly degrade the program's overall performance; If you do not transplant, will face a huge CPU-GPU I/O and GPU kernel call overhead. This is the most difficult of further work.

The BLASTN program hotspots module s_BlastSmallNaScanSubject_8_1Mod4 preliminary GPU parallelization transplantation, and the CPU version of the comparison results indicate that, using CUDA technology and GPU parallelization can improve the speed of the BLASTN accurate matching process, and for the long SUBJECT sequence, performance is obviously, the largest about 35 times. This shows that using the GPU on the BLASTN program parallelization transplantation is an effective program to address high-performance BLASTN needs. The same time, we also found in the work GPUs kernel call overhead between the CPU-GPU I/O, the use of the GPU memory optimization, as well as the problem that is embedded in the serial process, the impact of BLASTN GPU parallelization transplant performance factors issues.

5.4 Design of G-BLASTN

GPUs have become mature, many-core processors with much higher computational power and memory bandwidth than today’s CPUs. A GPU consists of a...
scalable number of streaming multiprocessors (SMs), each containing some streaming processors (SPs), special function units (SFUs), a multithreaded instruction fetch and issue unit, registers and a read/write shared memory. CUDA is currently the most popular programming model for general purpose GPU computing. The best way to use the hundreds to thousands of GPU cores is to generate a large number of CUDA threads that can access data from multiple memory spaces during their execution, as illustrated in Figure 5-3. Each thread has its private registers and local memory. Each GPU kernel function generates a grid of threads that are organized into thread blocks. Each thread block has shared memory visible to all threads within the block and with the same lifetime as the block. All threads have access to the same global memory. Two additional read-only memory spaces are accessible by all threads: the constant and texture memory spaces, both of which have limited caches.

---

Figure 5-4 Profiling of BLASTN for 300 query sequences (ranging from 500 to 100,000 bases) against human build 36 genome database under megablast mode. The lengths of the query sequences can be found in Figure 5-12.

Due to the complexity of BLASTN software, exploiting GPUs to accelerate BLASTN is a non-trivial task. The main challenge is that not all of the steps involved in BLASTN are suitable to be parallelized by GPUs. To identify which steps should be parallelized, we conducted a profiling study by running 300 different queries with
a broad range of lengths against the human build 36 genome database to analyze the
time distribution of different BLASTN steps under megablast mode (Figure 5-4). We
mainly observed the following details. The scanning stage is the most time-consuming
and accounts for 69-93% of the total execution time. Surprisingly, BLASTN spends
5-25% of the total execution time in the setup stage, mainly initializing the mask
database. The trace-back stage takes negligible time for most queries, but can occasionally take a very long time.

To achieve a good overall speedup, we designed G-BLASTN as follows. Its
major component is a set of CUDA kernel functions that run on GPUs to significantly
accelerate the seeding and mini-extension steps in the scanning stage. It is designed to
initialize the mask database once and then serve a large number of queries. Therefore,
the time spent in database initialization can be largely removed. We optimized the two
most time consuming functions in the trace-back stage and further designed a pipeline
mode under which the trace-back, output and scanning stages can run simultaneously.
The general framework of G-BLASTN is shown in Figure 5-5.
5.5 Implementation

We use CUDA C language to implement G-BLASTN based on NCBI BLAST 2.2.28 software package. It supports both Windows and Linux platforms. In the following, we present the detailed implementation of the major modules of G-BLASTN.

5.5.1 Accelerating the seeding step by GPU

The main task of the seeding step is to scan the database sequences and identify all w-gram matches. Due to the large database sizes, the seeding step is the most time consuming in BLASTN. Fortunately, there is a good chance that the seeding step can be parallelized due to the independence of the tasks at different offsets of the database. G-BLASTN first loads the database sequences to GPU global memory. Then for each query sequence, it stores a copy of the lookup table in GPU texture memory to achieve ultrafast table lookup. For each database sequence, it invokes a GPU kernel function that generates a large number of GPU threads to scan the database sequence in parallel; and hence the large number of GPU cores can be fully utilized to speed up the seeding step.

The implementation of the seeding step on the GPU is a major challenge, however. In CUDA, each thread block is organized as a number of warps, and each warp of threads is executed by a Single Instruction, Multiple Data (SIMD) hardware. When threads within a warp take different execution paths, the SIMD hardware will take multiple runs to go through these divergent paths, which will significantly decrease the utilization of GPU cores. In the case of BLASTN, the w-grams at different offsets of the database sequence may have no match or many matches to the query sequence, which can lead to severe thread branch divergence that decreases the GPU performance significantly. To conquer this challenge, we divide the seeding step into two sub-steps: scan and lookup. In the scan sub-step, we go through the whole database sequence in parallel and record all offsets of the database that have at least
one match to the query. Notice that we do not need to know how many matches have been found and where they are for each offset. Thus, each GPU thread can perform almost the same execution path and the effect of thread branch divergence can be minimized. In the lookup sub-step, we use another GPU kernel function to recheck all matched offsets and construct the complete set of matched offset pairs. This strategy works very well because the scan sub-step dominates the time of the seeding step.

There is yet another challenge in efficiently implementing the scan sub-step on the GPU. Once a thread finds a w-gram match, it has to increase a global counter and then write the matched offset into a global array. There are two negative consequences: (1) increasing the global counter must be an atomic operation, which means only one among all threads can operate while others have to wait; and (2) writing a single offset pair into the global array can waste a lot of GPU memory bandwidth. To overcome this challenge, we use a local counter and a local array for each thread block as temporary storage for the global counter and global array. The local counter and array are held in GPU shared memory, which is very fast. Now all thread blocks can operate on their own local counters and arrays simultaneously, boosting the overall performance. Once a local array becomes full, the set of offset pairs is written into global array as a whole and the global counter is updated by an atomic operation. We exploit coalesced memory write operations to achieve very high memory bandwidth. Meanwhile, the number of atomic operations on the global counter can be significantly reduced. The framework of the scan sub-step on GPU is shown in Figure 5-6.
For performance consideration, BLASTN supports two types of lookup tables for different types of queries: small and megablast\(^1\). Each type of lookup table has its own set of algorithms. Therefore, we have to implement different GPU kernel functions for different types of lookup tables.

A small lookup table contains a simple backbone array and an overflow array, both of which are simply an array of 16-bit integers. If the value of a backbone cell is nonnegative, it means that position in the lookup table contains exactly one query offset, which equals the cell value. If the value is \(-1\), the corresponding w-gram does not exist in the query sequence. If the value is \(-x\) (\(x > 1\)), the corresponding w-gram appears multiple times in the query sequence and their offsets begin at offset \(x\) of the overflow array and continue until a negative value is encountered. The pseudocode of our GPU scan and lookup kernel functions using a small lookup table are shown in Figures 6 and 7, respectively. The backbone array is held in GPU texture memory. Notice that a GPU kernel function specifies the behavior of a single GPU thread. There are hundreds of thousands of GPU threads simultaneously active, each of which executes the same instructions while working on different data items.

\(^1\) In current NCBI-BLAST, both types of lookup tables are supported by blastn mode and megablast mode.
Table 5-2 The GPU scan kernel function using small lookup table

**Input:** backbone[] // in texture memory

**Output:** P1[], P2[], globalCounter // P1 stores exact offset pairs, P2 stores overflow offset pairs, globalCounter stores the number of matches

**Key Variables:** BlastOffsetPair localArray[K]; // in shared memory
uint localCounter; // in shared memory

```c
s_index = blockIdx.x*blockDim.x + threadIdx.x;
do
    load base pairs into s from database sequence;
    h = hash_function(s);
    hv = backbone[h];
    calculate db_offset;
    if hv > -1 then
        atomicAdd(localCounter, 1);
        write offset pair (hv, db_offset) into localArray;
    end if
    if hv < -1 then
        atomicAdd(overflowCounter, 1);
        write offset pair (-hv, db_offset) into P2;
    end if
    __syncthreads(); // local barrier
if localCounter >= K/2 then
    if threadIdx.x == 0 atomicAdd(globalCounter, localCounter);
    __syncthreads(); // local barrier
    copy the offset pairs in localArray to P1;
if threadIdx.x == 0 localCounter = 0;
    __syncthreads(); // local barrier
end if
update s_index;
repeat until out of range
if localCounter > 0 then
    if threadIdx.x == 0 atomicAdd(globalCounter, localCounter);
    __syncthreads(); // local barrier
    copy offset pairs in localArray to P1;
end if
```

Table 5-3 The GPU lookup kernel function using small lookup table

**Input:** P1[], P2[], overflowTable[], globalCounter // P1 is exact offset pair array, P2 is overflow offset pair array, overflowTable is in texture memory

**Output:** P1[], globalCounter;

```c
index = blockIdx.x*blockDim.x + threadIdx.x;
read pair (hv, db_offset) from P2[index];
```
q_offset = overflowTable[hv++]; // overflow table lookup

do

doi
atomicAdd(globalCounter, 1);
write offset pair (q_offset, db_offset) into P1;
if hv <= the length of overflow table then
q_offset = overflowTable[hv++];
else
break;
end if
repeat until q_offset < 0

The megablast lookup table comprises three arrays: presence vector (PV array), hash table (hashtable[]) and next position (next_pos[]). The PV array is a bit field with one bit for each hash table entry. If a hash table entry contains a query offset, the corresponding bit in the PV array is set. The scanning process first checks the PV array to see whether there are any query offsets in a particular lookup table entry. The hashtable[] array is a thick backbone with one word for each of the lookup table entries. If a lookup table entry has no query offsets, the corresponding entry in hashtable[] is zero; otherwise, it is an offset into next_pos[]. The position in next_pos[] is in fact the query offset, and the actual value at that position is a pointer to the succeeding query offset in the chain. A value of zero means the end of the chain. The pseudocode of our GPU scan and lookup kernel functions using the megablast lookup table are shown in Figures 8 and 9, respectively. The scan kernel function checks the PV array to quickly determine whether there is a match. To achieve the best table lookup performance, the PV array is held in texture memory. The lookup kernel function takes the output of scan function as input and checks the hashtable[] and next_pos[] to find the complete set of matched offset pairs.

5.5.2 Accelerating the mini-extension step by GPU

It is not uncommon for the scan sub-step to return millions of seed matches. The mini-extension step is designed to verify whether each w-gram match can be extended to a W-gram match when w < W. We can create a huge number of GPU threads to extend those w-gram matches simultaneously. Each GPU thread reads one offset pair
from the matched offset pair array, extends on the left side and then extends on the right side. If it finds a W-gram match, this offset pair will be recorded for further gapped extension. Given that the mini-extension algorithm exhibits no big difference from the original BLASTN, we do not provide the pseudocode here. We note that there are two versions of mini-extension, one for the small lookup table and another for the megablast lookup table.

Table 5-4 The GPU scan kernel function using megablast lookup table

| Input: PV // presence vector; in texture memory |
| Output: P[], globalCounter // P stores all matched offset pairs |
| Key Variables: BlastOffsetPair localArray[K]; // in shared memory |

```plaintext
1 s_index = blockIdx.x*blockDim.x + threadIdx.x;
2 do
3     load base paz into s from database sequence;
4     h = hash_function(s);
5     if BlastMBLookupHasHits(h) == 1 then
6         calculate db_offset;
7         atomicAdd(localCounter, 1);
8         write offset pair (h, db_offset) into localArray
9     end if
10    __syncthreads(); // local barrier
11    if localCounter >= K/2 then
12        if threadIdx.x == 0 atomicAdd(globalCounter, localCounter);
13        __syncthreads(); // local barrier
14        copy offset pairs in localArray to P;
15        if threadIdx.x == 0 localCounter = 0;
16        __syncthreads(); // local barrier
17    end if
18    update s_index;
19 repeat until out of range
20    if localCounter > 0 then
21        if threadIdx.x == 0 atomicAdd(globalCounter, localCounter);
22        __syncthreads(); // local barrier
23        copy offset pairs in localArray to P;
24    end if
```

Table 5-5 The GPU lookup kernel function using megablast lookup table

| Input: P, hashtable, next_pos |
| Output: P1 |

```plaintext
1     index = blockIdx.x*blockDim.x + threadIdx.x;
```
read pair (h, db_offset) from P[index];
q_offset = hashtable[h];

while q_offset > 0
    atomicAdd(globalCounter, 1);
    write (q_offset-1, db_offset) to P1;
    if q_offset < the length of next_pos table then
        q_offset = next_pos[q_offset];
    else
        break;
end if
end while

5.5.3 Optimizing the trace-back step

As mentioned in Section 2.2, occasionally the trace-back step takes quite a long time, which may counteract the speedup achieved by the previous steps. Unfortunately, the trace-back step is not naturally suitable for GPUs. We therefore resort to the following optimization techniques. First, function s_SeqDBMap NA2ToNA8() uses a translation table to convert sequence data from NCBI-NA2 to NCBI-NA8 format. BLASTN translates the data character by character, which does not fully use the CPU memory bandwidth. In G-BLASTN, we replace four 8-bit memory writes with a single 32-bit memory write, which boosts the speed by 2 to 3 times. Second, function s_SeqDBMapNcbiNA8ToBlastNA8() uses a 16-byte translation table to convert sequence data from NCBI-NA8 to BLAST-NA8 format, character by character. G-BLASTN uses a 128-bit union (denoted by ntob_table) to hold the 16-byte translation table, and then SSE instructions to write 16 bytes as a whole, which achieves a speedup of 3~4X. The SSE instructions are shown in Table 5-6.

<table>
<thead>
<tr>
<th>Table 5-6 SSE instructions used by s_SeqDBMapNcbiNA8ToBlastNA8()</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 set pointer p_buf to the address of 128-bit data;</td>
</tr>
<tr>
<td>2 __m128i t_buf = _mm_loadu_si128(p_buf); // load data into register</td>
</tr>
<tr>
<td>3 t_buf=_mm_shuffle_epi8(ntob_table, t_buf); // translate the data</td>
</tr>
<tr>
<td>4 _mm_storeu_si128(p_buf,  t_buf); // write back data</td>
</tr>
</tbody>
</table>
5.5.4 Pipeline mode for multiple queries

Once we have accelerated the scanning stage by GPU, other stages such as trace-back and output may start to occupy a relatively large portion of the total execution time, especially when there are many final hits. G-BLASTN supports a pipeline mode when handling a batch of queries. The main advantage of the pipeline mode is that the GPU and CPU can work on different tasks simultaneously, as shown in Figure 5-7. In short, when the GPU is busy with seeding or mini-extension, the CPU can execute the trace-back or output steps for a previous query. To achieve this purpose, G-BLASTN uses multithreading to maintain four queues: query, job, prelim and result. A master thread reads the queries and puts them into the query queue, and then creates the job queue. The prelim thread(s) fetches jobs from the job queue and uses GPU to execute the preliminary search, storing the results in the prelim queue. The trace-back thread(s) reads from the prelim queue, executes the trace-back step and stores the results in the results queue. Finally, the print thread prints the results. This pipeline design can efficiently use both GPU and CPU resources.

Figure 5-7 The pipeline mode of G-BLASTN
5.6 Results

5.6.1 General setup and data sets

The GPU experiments were performed on a desktop computer with an Intel quad-core CPU and Nvidia GTX780 GPU. The CPU experiments were performed on two different platforms: a 4-core platform which is the same computer that runs the GPU experiments; and an 8-core platform which is a server with two Intel Xeon CPUs. The detailed system configuration is shown in Table 5-7.

Table 5-7 System configuration

<table>
<thead>
<tr>
<th>CPU</th>
<th>Memory</th>
<th>GPU</th>
<th>Storage</th>
<th>OS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel Core i7-3820 (4-core, 3.6GHz)</td>
<td>32GB (DDR3 1600)</td>
<td>Nvidia GTX780</td>
<td>SATA</td>
<td>CentOS 6.4 (Linux kernel 2.6.32)</td>
</tr>
<tr>
<td>2 x Intel Xeon E5620 (8-core, 2.4GHz)</td>
<td>24GB (DDR3 1333)</td>
<td>N/A</td>
<td>SATA</td>
<td>Redhat 5.5 (Linux kernel 2.6.18)</td>
</tr>
</tbody>
</table>

We used the following two command lines to run NCBI BLASTN and G-BLASTN, respectively. More details about the command options of G-BLASTN can be found on our website.

\`
$\text{blastn}\ -db\ <\text{database}>\ -query\ <\text{query}>\ -\text{task\ megablast}|\text{blastn}\ -\text{outfmt}\ 7\ -\text{out}\ <\text{file}>\ -\text{dust}\ \text{yes}\ -\text{window}\_\text{masker}\_\text{db}\ <\text{masker}\_\text{db}>\ -\text{num}\_\text{threads}\ <1|4|8>
$

\$
\text{gblastn}\ -db\ <\text{database}>\ -\text{query}\_\text{list}\ <\text{query}\_\text{list}>\ -\text{task\ megablast}|\text{blastn}\ -\text{outfmt}\ 7\ -\text{out}\ <\text{file}>\ -\text{dust}\ \text{yes}\ -\text{window}\_\text{masker}\_\text{db}\ <\text{masker}\_\text{db}>\ -\text{use}\_\text{gpu}\ \text{true}\ -\text{mode}\ <1|2>\ -\text{num}\_\text{threads}\ <1|4|8>
$

We used gettimeofday() functions to measure the program execution time. Each experiment was run 10 times and the average results are reported in this paper.

Databases. We chose human build 36 and mouse build 36 genome databases for the experiments of megablast mode. In addition, we constructed a database to test the blastn mode by selecting all sequences with length no less than 2million from NCBI nt database. This partial NCBI nt database has a raw size of 8.4GB and can well fit into a single GPU card with 3GB memory after compression. All databases were masked with WindowMasker [90], including low-complexity filtering by DUST [91]. The length information of all database sequences can be found in Figure 5-8.
(a) Human Build 36

(b) Mouse Build 36

(c) The longest 2181 sequences from NCBI non-redundant nucleotide database

Figure 5-8 Lengths of database sequences
Queries. To test the megablast mode, we chose queries from the NCBI ftp server: ftp://ftp.ncbi.nlm.nih.gov/pub/agarwala/indexed_megablast/queries [78]. Six query sets, each containing 100 queries, were used, which are referred to as Qsmall (∼500 bases, range: 501-506), Qmedium (∼10 Kbases, range: 10000-10446) and Qlarge (∼100 Kbases, range: 100001-102087). To test the blastn mode, we chose the first 500 bacterial sequences from the NCBI server: http://www.ncbi.nlm.nih.gov/sra/SRX338063, namely Qbac. The length information of all query sequences can be found in Figure 5-9.

(a) Human Query Sequences

(b) Mouse Query Sequences
5.7 Experimental results

5.7.1 Performance under normal mode

Under normal mode, G-BLASTN handles the queries one at a time. We first present the experimental results of megablast. The speedups over 8-core platform on human genome database are shown in Figure 12. The speedups of other experiments are shown in Figure 5-10 Figure 5-11 Figure 5-12 Figure 5-13. We also show the average speedup of each query set in Table 2. The overall speedups are calculated as the average of all 1600 query experiments for each hardware setting. As compared with 4-core Intel i7-3820, G-BLASTN achieves an overall speedup of 7.15X. As compared with the 8-core platform, G-BLASTN achieves an overall speedup of 13.76X. There are several reasons why BLASTN runs much faster on i7-3820 than on Xeon E5620. Firstly, i7-3820 has a much higher working frequency than E5620. Secondly, the memory bandwidth of i7-3820 is twice of E5620. Thirdly, the memory module of our i7-3820 platform is faster than that of E5620 platform. Based on the results of E5620, we can notice that the speedups achieved using 8 cores are only slightly better than using 4 cores.
(a) speedup of $Q_{small}$ queries on human genome database

(b) speedup of $Q_{medium}$ queries on human genome database

(c) speedup of $Q_{large}$ queries on human genome database
(d) speedup of Qbac queries on human genome database

(e) speedup of Qsmall queries on mouse genome database

(f) speedup of Qmedium queries on mouse genome database
Figure 5-10 Speedup of megablast mode (G780 vs. i7-3820; speedup-1 is for 1 CPU core, speedup-4 is for 4 CPU cores)
(a) speedup of Qbac queries on human genome database

(b) speedup of Qsmall queries on mouse genome database

(c) speedup of Qmedium queries on mouse genome database
(d) speedup of Qlarge queries on mouse genome database

(e) speedup of Qbac queries on mouse genome database

Figure 5-11 Speedup of megablast mode (G780 vs. E5620; speedup-1 is for 1 CPU core, speedup-4 is for 4 CPU cores, speedup-8 is for 8 CPU cores)
(a) Speedup of Qbac on human genome database

(b) Speedup of Qbac on mouse genome database

(c) Speedup of Qbac on NCBI nt genome database

Figure 5-12 Speedups of blastn mode (G780 vs. i7-3820; speedup-1 is for 1 CPU core, speedup-4 is for 4 CPU cores)
Figure 5-13  Speedups of blastn mode (G780 vs. E5620; speedup-1 is for 1 CPU core, speedup-4 is for 4 CPU cores, speedup-8 is for 8 CPU cores)
We also notice that the speedups on the human database are less than those on the mouse database. This is mainly because the human build 36 database consists of 367 sequences, more than 200 of which are short sequences (less than 1 million bps). In contrast, the mouse build 36 database consists of 21 very long sequences. We will discuss more on this issue in Section 5.

(a) speedup of Qsmall queries

(b) speedup of Qmedium queries

(c) speedup of Qlarge queries

Figure 5-14 Speedup of G-BLASTN on human genome database (GTX780 vs. Two Xeon E5620)
We evaluate the performance of blastn mode using Qbac query set against human,
mouse, and the partial NCBI nt databases. We show the average speedups of each
database and the overall speedups in Table 3. G-BLASTN achieves an overall
speedup of 1.56 and 2.95 as compared with 4-core i7-3820 and 8-core E5620,
respectively. As compared with megablast mode, blastn mode has a much smaller
value of stride size, which results in more scanning workload and more seed hits.
Therefore the ungapped extension step under blastn mode takes much longer time
than under megablast mode, as shown in Figure 5-15. Since the ungapped extension
is sequentially executed on CPU, the speedups achieves under blastn mode are much
less than megablast.

<table>
<thead>
<tr>
<th>Database</th>
<th>Query</th>
<th>Intel i7-3820 1-core</th>
<th>Intel i7-3820 4-core</th>
<th>Intel Xeon E5620 1-core</th>
<th>Intel Xeon E5620 4-core</th>
<th>Intel Xeon E5620 8-core</th>
</tr>
</thead>
<tbody>
<tr>
<td>Human</td>
<td>Qsmall</td>
<td>10.47</td>
<td>5.11</td>
<td>26.90</td>
<td>12.88</td>
<td>10.52</td>
</tr>
<tr>
<td></td>
<td>Qmedium</td>
<td>11.49</td>
<td>4.53</td>
<td>32.68</td>
<td>11.98</td>
<td>8.50</td>
</tr>
<tr>
<td></td>
<td>Qlarge</td>
<td>9.22</td>
<td>3.37</td>
<td>21.07</td>
<td>7.54</td>
<td>5.25</td>
</tr>
<tr>
<td></td>
<td>Qbac</td>
<td>10.80</td>
<td>5.37</td>
<td>26.04</td>
<td>12.71</td>
<td>10.47</td>
</tr>
<tr>
<td>Mouse</td>
<td>Qsmall</td>
<td>18.50</td>
<td>9.37</td>
<td>44.12</td>
<td>21.87</td>
<td>18.28</td>
</tr>
<tr>
<td></td>
<td>Qmedium</td>
<td>17.84</td>
<td>7.39</td>
<td>49.16</td>
<td>19.11</td>
<td>14.32</td>
</tr>
<tr>
<td></td>
<td>Qlarge</td>
<td>10.44</td>
<td>4.14</td>
<td>23.16</td>
<td>9.14</td>
<td>6.92</td>
</tr>
<tr>
<td></td>
<td>Qbac</td>
<td>20.97</td>
<td>10.73</td>
<td>49.28</td>
<td>24.71</td>
<td>20.80</td>
</tr>
<tr>
<td>Overall</td>
<td></td>
<td>14.80</td>
<td>7.15</td>
<td>35.85</td>
<td>16.85</td>
<td>13.76</td>
</tr>
</tbody>
</table>

Table 5-9 Speedup of G-BLASTN under blastn mode using Qbac query set

<table>
<thead>
<tr>
<th>Database</th>
<th>Intel i7-3820 1-core</th>
<th>Intel i7-3820 4-core</th>
<th>Intel Xeon E5620 1-core</th>
<th>Intel Xeon E5620 4-core</th>
<th>Intel Xeon E5620 8-core</th>
</tr>
</thead>
<tbody>
<tr>
<td>Human</td>
<td>4.58</td>
<td>1.57</td>
<td>8.01</td>
<td>2.99</td>
<td>2.15</td>
</tr>
<tr>
<td>Mouse</td>
<td>5.02</td>
<td>1.83</td>
<td>8.84</td>
<td>3.51</td>
<td>2.70</td>
</tr>
<tr>
<td>NCBI nt</td>
<td>3.37</td>
<td>1.29</td>
<td>5.71</td>
<td>2.36</td>
<td>1.74</td>
</tr>
<tr>
<td>Overall</td>
<td>4.32</td>
<td>1.56</td>
<td>7.52</td>
<td>2.95</td>
<td>2.20</td>
</tr>
</tbody>
</table>
Figure 5-15 The time distribution of Qbac on Human genome database. Remark: In (b), the time of ungapped extension is too short to be noticeable.
5.7.2 Performance under pipeline mode

To evaluate the performance of pipelined G-BLASTN, we use all queries in each data set as a single input to G-BLASTN. The speedups against the NCBI BLAST on Intel i7-3820 are shown in Table 4. If we compare Table 4 with Tables 2 & 3, we can observe a significant improvement on the speedups for many data sets. For small and medium queries under megablast, the trace-back and output steps account for a very small portion of the total time, and hence the pipeline design does not offer much of an advantage. For large queries using megablast, however, the trace-back and output steps take a much longer time due to a greater number of final hits, and thus the pipeline design hides a significant portion of time. Under blastn mode, the pipeline design can further improve the speedups of the Qbac query set by 19-44%.

Table 5-10 Speedup of pipelined G-BLASTN (GTX780 vs. Intel i7-3820)

<table>
<thead>
<tr>
<th>Mode</th>
<th>Database</th>
<th>Query</th>
<th>1-core</th>
<th>4-core</th>
</tr>
</thead>
<tbody>
<tr>
<td>megablast</td>
<td>Human</td>
<td>Qsmall</td>
<td>10.83</td>
<td>5.28</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Qmedium</td>
<td>12.67</td>
<td>5.05</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Qlarge</td>
<td>12.19</td>
<td>4.68</td>
</tr>
<tr>
<td></td>
<td>Mouse</td>
<td>Qsmall</td>
<td>20.49</td>
<td>10.37</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Qmedium</td>
<td>20.12</td>
<td>8.51</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Qlarge</td>
<td>12.09</td>
<td>5.47</td>
</tr>
<tr>
<td>blastn</td>
<td>Human</td>
<td>Qbac</td>
<td>5.49</td>
<td>1.87</td>
</tr>
<tr>
<td></td>
<td>Mouse</td>
<td>Qbac</td>
<td>6.49</td>
<td>2.36</td>
</tr>
<tr>
<td></td>
<td>NCBI nt</td>
<td>Qbac</td>
<td>4.73</td>
<td>1.86</td>
</tr>
</tbody>
</table>

5.8 Discussions and Conclusions

In this thesis, we describe our design and implementation of G-BLASTN, an open source software tool for nucleotide alignment based on the widely used NCBI-BLAST. G-BLASTN exploits the power of GPUs to accelerate nucleotide alignments. Compared with a contemporary quad-core Intel CPU running at 3.6GHz, G-BLASTN on a single $650 GPU card can achieve overall speedups of 14.8X and 4.32X under megablast mode and blastn mode respectively. When compared with multithreaded
NCBI-BLAST that uses four CPU cores, G-BLASTN can still achieve overall speedups of 7.15X (megablast) and 1.56X (blastn). G-BLASTN also supports a pipeline mode that further improves the overall performance by up to 44% when handling multiple queries.

G-BLASTN can be improved in the following directions. At present, G-BLASTN invokes a kernel function for each database sequence, which is not efficient when the length of the database sequence is shorter than one million bps. Besides the constant overhead of invoking a kernel function, another reason is that short database sequence only generates a small number of hits during the scanning stage, and hence the effect of our optimization strategy on the atomic operations will diminish. There are several possible solutions to this problem. One possibility is to aggregate short database sequences into longer ones. Another solution is to process multiple database sequences in each kernel function call. G-BLASTN is also limited by the GPU memory size. We plan to extend G-BLASTN to support multiple GPU cards. Doing so can not only support much larger databases, but also achieve better speedups. A more challenging task is to accelerate other steps such as ungapped extension, gapped extension, and trace-back, which will improve the performance of blastn mode significantly. Finally, we also plan to support discontiguous megablast mode in our future work.
Chapter 6  Conclusion and future work

DNA sequence alignment always plays an important role in bioinformatics. The research about how to accelerate the DNA sequence alignment attracts lots of attention from the academia and industry. During my doctoral research, several DNA alignment algorithms are studied. Since the DNA sequence alignment is a data intensive computing problem, I made some deep research of the data access based on the current GPU architecture and proposed a new memory access based parallel computing model. In Chapter 3, based on GPU architecture I had designed and implemented various large data access testing on GPU memories. The performance of memory access on GPU in parallel computing was tested for different memory layout and different tread access methods. The conclusion was obtained, that for data intensive problems the GPU parallel computing can be simplified as a model of multiple threads accessing the memory IO at the same time, thus the computing and memory IO is united, what’s more, the design flow on parallel algorithm on GPU architecture is simplified.

Based on the memory model of GPU parallel architecture, in Chapter 4, the SOAP2 algorithm is ported to GPU, thereby the SOAP3 algorithm is implemented and improved. The performance of SOAP3 is much better than SOAP2 based on CPU. According to the data collected in this thesis, the speed of the sequence alignment on SOAP3 based on GPU is almost ten times faster that it on SOAP2 based on CPU. In Chapter 4, based on the research, a new method of optimized GPU parallel computing algorithm is proposed. The BLASTN algorithm studied in Chapter 5 has been developed over 30 years in industry. It has million lines of codes and thus the system and architecture is very complex. In the past few years, I have done lots of research on the algorithm of BLASTN, and implicated the porting of BLASTN on GPU, and put forward the new “LOOK UP TABLE” method based on GPU architecture. Based on the memory architecture of the GPU, this new method can change the previous multiple memory accesses to two cache accesses, thereby makes the memory data operation much more convenient. Meanwhile, based on the different threads having
different processing time, a new multi kernels optimization strategy is proposed, which insures most threads work in the saturation area. For the whole framework of BLASTN, through multiple GPUs running simultaneously at a higher level and designing a higher level multi-threaded asynchronous GPU-GPU runtime architecture, CPU-GPU heterogeneous computing framework can be optimized. In the highest degree of parallelism, the performance of the GPU is 10 – 30 times faster than the CPU.

DNA sequence alignment was, is, and will be one of the most important research problems in bioinformatics. As the architecture of GPU is constantly innovating, for the new GPU architecture and parallel computing model, I hope to further enrich the GPU memory parallel computing model, to unify computing and IO, and develop a deeper theory, in order to simplify the porting data intensive algorithms to GPU based computing architecture. At the same time, I hope that the parallel memory computing model can be unified into the future of chip design.
Publication List

Edited Book


Translated book


Journal Papers

- Beini Jiang, Allan Struthers, Zhe Sun, Zhuo Feng, Xuqian Zhao, Kaiyong Zhao, Weizhong Dai, Xiaobo Zhou, Michael E. Berens, Le Zhang, “Employing graphics processing unit technology, alternating direction implicit method and domain decomposition to speed up the numerical diffusion solver for the biomedical engineering research”, Article first published online: 2 MAY 2011.

Conference Papers


• K.-Y Zhao, Shifu Chen, Y.-B Zhang, Y.-L Chu, Photo Editing on the GPU with MuseMage, GTC 2010, CA, USA


In Chinese

• Guangyong Zhang, Xiaowei Lu, Bo Shen, Kai Zhang, Qi Chu, Kaiyong Zhao," Design and Optimization of Simultaneous Iterative Reconstruction Technique Based on GPU Platform ",<Computer Science>,2012, 39(5)


• ZHANG Qing, CHI Xu-Guang, XIE Hai-Bo, ZHAO Kai-Yong, WU Qing,CHEN Wei, WANG Shi-Hu, ZHU Xiao-Wen, ” Parallel Algorithm Based on the Travel Time Computing of Pre-stack Time Migration Using GPU”,< Microcomputer & Its Applications>, 42-46, 2011.8


• CHU Qi, ZHANG Kai, WANG Xiaohua, ZHANG Ci, ZHANG Yan, ZHANG Guangyong, SHEN Bo,LU Xiaowei, ZHAO Kaiyong,CHU Xiaowen,ZHANG Fa,SUN Fei, “ATOM 1.0:A GPU Powered Package for Electron Tomography Reconstruction", 《Acta Biophysica Sinica》, 231-241, 2011.3


Poster
• K. Zhao and X.-W. Chu, “GPU-BLASTN: Accelerating Nucleotide Sequence Alignment by GPUs,” Poster, RECOMB 2013, Beijing, China.
References


[27] HIGH PERFORMANCE COMPUTING - HIPC 2006
[31] 万宁，谢海波，张清，赵开勇，褚晓文，于军，使用 GPU 加速 BLAST 算法初探，《计算机工程与科学》2009 年 11 期。
Sequence Alignment Algorithms: A Survey, 96-103, ISTA '09


[40] H. Li, N. Homer, A survey of sequence alignment algorithms for next-generation sequencing, Brief Bioinform (2010) http://dx.doi.org/10.1093/bib/bbq015


[42] Fabiano C. Botelho, Yoshiharu Kohayakawa, Nivio Ziviani, A practical minimal perfect hashing method (2005), in Proc. of the 4th International Workshop on Efficient and Experimental Algorithms (WEA'05)


CURRICULUM VITAE

Academic qualifications of the thesis author, Mr. Zhao Kaiyong, David:

- Received the degree of Bachelor of engineering from Beijing Institute of Technology, June 2005.
- Received the degree of Master of Philosophy from Hong Kong Baptist University, November 2010.

November 2016