GPU data-parallel computing of sequence alignment using CUDA
Jung, Sungbo
ProQuest Dissertations and Theses; 2008; ProQuest Dissertations & Theses (PQDT)
pg. n/a

GPU DATA-PARALLEL COMPUTING OF SEQUENCE ALIGNMENT USING CUDA

By

Sungbo Jung
B.A., Korea University, Korea, 2000
M.E., Korea University, Korea, 2003

A Thesis
Submitted to the faculty of the
Graduate School of the University of Louisville
In Partial Fulfillment of the Requirements
for the Degree of

Master of Science

Department of Computer Engineering and Computer Science
University of Louisville
Louisville, Kentucky

December 2008

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

UMI Number: 1468602

INFORMATION TO USERS

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleed-through, substandard margins, and improper alignment can adversely affect reproduction.
In the unlikely event that the author did not send a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

UMI®

UMI Microform 1468602
Copyright 2009 by ProQuest LLC
All rights reserved. This microform edition is protected against unauthorized copying under Title 17, United States Code.

ProQuest LLC
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106-1346

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Copyright 2008 by Sungbo Jung

All rights reserved

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

GPU DATA-PARALLEL COMPUTING OF SEQUENCE ALIGNMENT USING CUDA

By

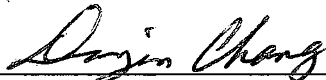
Sungbo Jung
B.A., Korea University, Korea, 2000
M.E., Korea University, Korea, 2003

A Thesis Approved on

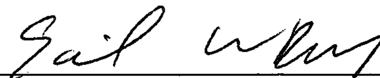
December 5, 2008

by the following Thesis Committee:

Thesis Director - Ming Ouyang, Ph.D.



Dar-jen Chang, Ph.D.



Gail W. DePuy, Ph.D.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

ACKNOWLEDGMENTS

Though only my name appears on the cover of this thesis, a great many people have contributed to its production. I owe my gratitude to all those people who have made this thesis possible and because of whom my graduate experience has been one that I will cherish forever.

My deepest gratitude is to my advisor, Dr. Ming Ouyang. I have been amazingly fortunate to have an advisor who gave me a vision of research, and at the same time the guidance to recover when my steps faltered. I am also thankful to him for encouraging the use of correct grammar and consistent notation in my writings and for carefully reading and commenting on countless revisions of this manuscript. Dr. Dar-jen Chang has been always there to take care of me and give advice. I am deeply grateful to him for helping me sort out the technical details of my work. Dr. Gail W. DePuy's insightful comments of my thesis were thought-provoking. I am grateful to her for holding me to a high research standard and enforcing strict validations for each research result, and thus teaching me how to do research.

I am also indebted to the members of the CECS Tech Service where I worked. Particularly, I would like to acknowledge Big Boss Mr. Ron Lile and Mrs. Susan Lile, My best mentor Sir Jeffrey Marean. Finally, I appreciate my family, father, mother, sister, and sister's family.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

ABSTRACT

GPU DATA-PARALLEL COMPUTING OF SEQUENCE ALIGNMENT
USING CUDA

Sungbo Jung

December 5, 2008

Recently rapid growth of the technology of the Graphics Processing Unit (GPU) has led to a surge of interest in using the GPU for general purpose applications. We can utilize the GPU in computation as a massive parallel co-processor because the GPU consists of multiple cores. In bioinformatics, finding the similarities in protein and DNA sequence databases has become a fundamental procedure. The Smith-Waterman algorithm based on the dynamic programming is one of the methods used to search for all of the possible local alignments between two sequences, enabling us to find the optimal local alignments. However, the dynamic programming requires a sequential calculation due to data dependency. Also required is a high number of computation steps proportional to the product of the lengths of the two sequences.

The approach of the present study is to implement the Smith-Waterman algorithm using the huge computational power of the GeForce 8800 series, based on NVIDIA's G80 architecture, to develop high performance solutions for local sequence alignment. To program the G80 hardware, the parallelized Smith-Waterman with the wavefront

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

algorithm is implemented in NVIDIA CUDA, an extended C language. Except the first row and the first column, every cell in the sequence matrix depends on previous calculations of the neighbor cells. However, all the cells in an anti-diagonal can be independently calculated. Thus we can parallelize the Smith-Waterman algorithm by using multiple streaming processors to calculate an anti-diagonal.

The computation is conducted on an NVIDIA GeForce 8800GT installed in a 3.0 GHz Intel Pentium D computer equipped with 2 GB RAM, running Microsoft Windows XP Professional. The results show that the parallelized Smith-Waterman with the wavefront algorithm on the GPU achieves from 1.5 to 3.7 times speed-up than the sequential Smith-Waterman on the CPU. Therefore, the GPU is shown to be sufficiently advanced to be used as an efficient high computing accelerator for a bioinformatics application.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

TABLE OF CONTENTS

PAGE

ACKNOWLEDGMENTS iii
ABSTRACT iv
LIST OF TABLES viii
LIST OF FIGURES ix

CHAPTER

1. INTRODUCTION 1
   1.1 Motivation 1
   1.2 Contributions of the Thesis 3
   1.3 Organization of the Thesis 3
2. GPU ARCHITECTURE 4
   2.1 The arise of the Graphics Processing Unit 4
   2.2 GPGPU 5
   2.3 GPU Programming Language 7
   2.4 G80 Architecture 11
   2.5 CUDA Programming Model 13
3. SEQUENCE ALIGNMENT 20
   3.1 Bioinformatics 20
   3.2 Sequence Alignment 20

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

3.3 Dynamic Programming........ 21

3.4 The Smith-Waterman Algorithm 22

3.5 Sequence Alignment on the GPU........ 24

4. SMITH-WATERMAN ON THE GPU USING CUDA 26

4.1 Parallelized Smith-Waterman Algorithm 26

4.2 Wavefront Algorithm 26

4.3 Smith-Waterman on the GPU System 27

4.4 CUDA Implementation 29

5. RESULTS AND CONCLUSION 37

5.1 Environment........ 37

5.2 Results........ 38

5.3 Conclusion........ 38

REFERENCES 40

CURRICULUM VITAE 43

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

LIST OF TABLES

TABLE PAGE

1. Memory types in CUDA 16

2. Protein conversion in substitution matrix 35

3. Experimental result between CPU-SW and GPU-SW with 1 block 38

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

LIST OF FIGURES

FIGURE PAGE

1. The growth of the GPU vs. the CPU 5
2. The graphics pipeline 6
3. An overview of the G80 architecture 12
4. Multiprocessor design for CUDA 14
5. Organization of CUDA threads 15
6. Memory architecture of the G80........ 18
7. Tracking the alignment scores 22
8. Smith-Waterman scoring schemes 23
9. Data dependency of each cell 24
10. Vector parallel model and pre-defined query-profile 25
11. Diagonal sweep wavefront 27
12. Diagram of Smith-Waterman CUDA system 28
13. The sequence matrix 30
14. Pre-defined macro code 30
15. BLOSUM50 substitution matrix for SW CUDA........ 31
16. Anti-diagonal matrix conversion 32
17. Data dependency of each anti-diagonal part 33
18. Bank conflict checker code 34
19. Mid anti-diagonal kernel code 35

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

CHAPTER 1

INTRODUCTION

1.1 Motivation

At the beginning of the personal computer era, the graphics card was created for the purpose of display. It had been designed as a 2-Dimensional (2D) graphics accelerator in order to use a palette of colors and fast 2D display. Then it became a multi-media device for playing high quality video. Recently the graphics card has been developed as a 3-Dimensional (3D) graphics accelerator to meet the demand of 3D gaming and 3D visualization. Furthermore, it has evolved into Graphics Processing Units (GPUs) with multiple streaming processors for high computing power. The trend of Central Processing Unit (CPU) has changed from the high clock cycle single core architecture to the multi-core architecture due to the problems of heat dissipation, power consumption, and physical limitation. GPUs have also evolved into multi-core streaming processors that can perform several Vertex shaders and Pixel shaders. So now GPUs can perform floating point computation faster than the CPU. Therefore there is attempt to use the computation power of GPUs not only in the 3D graphics applications, but also in the general purpose applications. This approach is called General Purpose computation on

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

GPU (GPGPU) [1]. It will be useful and efficient to utilize the parallel computing power of the GPU for applications that need high computational power with multiple data.

In bioinformatics, high computational power is required to process large amounts of data and complicated calculations. Finding sequence similarities in protein and DNA databases has become a routine procedure. Based on dynamic programming, the Smith-Waterman algorithm is one of the methods to search for all possible alignments between two sequences to find the optimal local alignments. However, dynamic programming is traditionally implemented as sequential calculations, and the number of the required operations is proportional to the product of the lengths of the two sequences. Many researchers have attempted to use high performance devices like super computers, cluster computers, or specially designed devices in order to reduce the computation time. Although they can deliver enough computational power, the costs are pretty high and they are hard to access. Thus, GPUs, being inexpensive commodity high performance hardware, may be a good solution towards bioinformatics problems.

NVIDIA, one of the major GPU manufacturers, developed a new graphics chipset architecture code-named G80 in 2007 [2]. It has 128 streaming processors with 768MB memory. The G80 supports CUDA, an extended C language environment, to implement GPGPU applications. Contrary to previous GPGPU approaches that use indirect operations through the vertex shader and the pixel shader, CUDA is easy to develop general purpose applications on the GPU.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

1.2 Contribution of the Thesis

The goal of this thesis is to implement the Smith-Waterman algorithm using the huge computational power of the GeForce 8800 series, which is based on NVIDIA's G80 architecture. The Smith-Waterman algorithm is parallelized via the wavefront pattern. Except the first row and the first column, every cell in the sequence matrix depends on previous calculations of the neighbor cells. However, all cells in an anti-diagonal can be independently calculated. Thus the anti-diagonals of the sequence matrix of the Smith-Waterman algorithm can simultaneously be processed by multiple streaming processors.

1.3 Organization of the Thesis

The rest of the thesis is organized as follows. Chapter 2 presents GPU architecture, including GPGPU, the G80 architecture, and the CUDA language. Chapter 3 describes sequence alignment including the Smith-Waterman algorithm and a review of the literature. Chapter 4 describes the implementation of the parallelized Smith-Waterman algorithm on the G80 with CUDA. In particular, the wavefront algorithm as a method of parallelizing dynamic programming is introduced in this chapter. Finally, Chapter 5 summarizes computational results, contribution of this thesis to the research field, and future works.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

CHAPTER 2

GPU ARCHITECTURE

2.1 The Arise of the Graphics Processing Unit

During the last decade, the performance of the Graphics Processing Units (GPU) has been dramatically increased by the development of new technology. In the beginning, a graphics card was designed for purpose of display, so the main features of a graphics card were 2D Graphics such as the number of colors, the quality of display, and the support of high resolution. In the mid 1990s, enhanced Operating Systems (O/S) with user friendly Graphics User Interface (GUI) led to demanding multi-media environments in order to play video files, to support 3D graphics games, and to manage multiple displays. The demands of 3D graphics led to the creation of the GPU, which had better integration and faster speed. In 2000, the multi-core platform was incorporated in the design of GPU. Major vendors such as ATI, NVIDIA, and 3D Labs competed to develop real time 3D graphics capable GPU and they used the multi-core technology for parallel processing. Now, floating point performance of the GPU is higher than performance of the CPU because the architecture of the GPU is dramatically changed via improvement of the chip design and manufacturing technology [3]. Thus, people want to use this great capability for general purpose applications.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

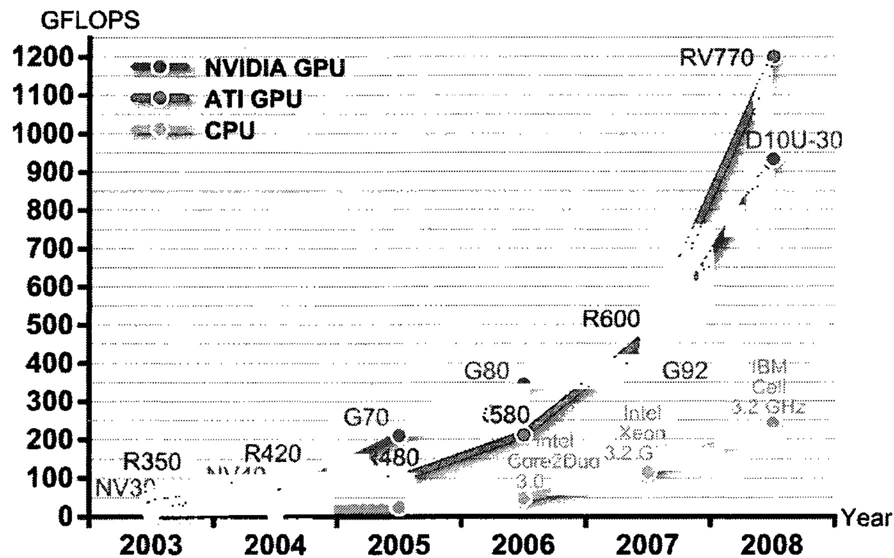


Figure 1. The growth of the GPU vs. the CPU

2.2 GPGPU

At the present time, the GPUs are the most economical and powerful computational hardware because they are inexpensive and user programmable, and they achieve high performance. The increased flexibility and high computing capabilities of GPUs have led a new research field that explores the performance of GPUs for general purpose computation. The general-purpose computation on the GPU (GPGPU) is getting the attention of many researchers and developers [4].

The processing of information in 3D graphics occurs in several serialized stages, where each stage generates results for the next, and a 3D image is transformed into 2D. Graphics processing in the GPU is like an assembly line with each stage affecting successive stages and all stages working in parallel. This architecture is called the graphics pipeline. The earlier design of the graphics pipeline was a "fixed function

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

pipeline, where the limited number of operations available at each stage of the graphics pipeline was hard-wired for specific tasks" [1]. The technology of the GPU has evolved into a more flexible programmable pipeline, and the graphics pipeline has been replaced by the user programmable vertex shader and pixel shader. "A programmer can now implement custom transformation, lighting, or texturing algorithms by writing programs called shaders" [5]. The pixel shader is more flexible than vertex shader to program the GPGPU applications. Recent GPUs have fully programmable unified processing units with support for single precision floating-point computation. Furthermore, the latest generation of GPUs, such as ATI's RV770 and NVIDIA's GT200, is expanding on its capabilities to support double precision floating point computation [6, 7]. High speed, increased precision, and rapidly expanding programmability of GPUs have transformed GPUs to a powerful platform for general purpose computations.

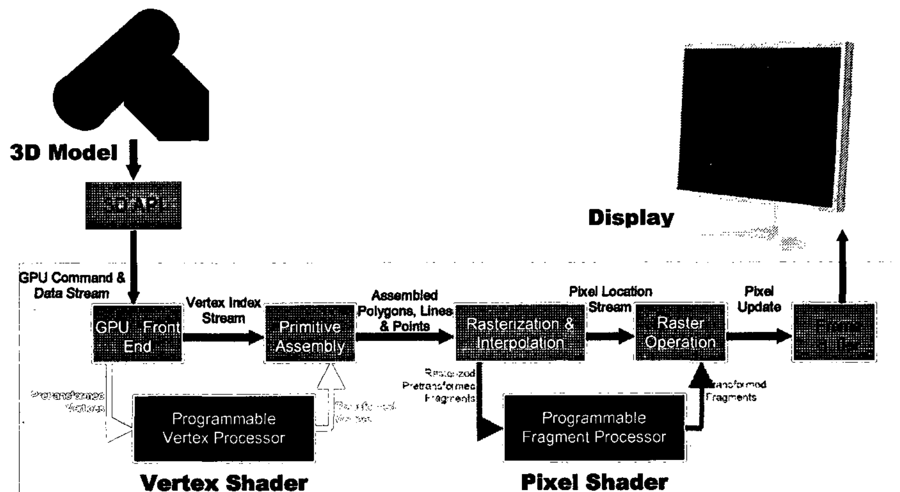


Figure 2. The graphics pipeline

Although GPUs have become a compelling platform for general-purpose computation, there are some limitations and difficulties to program them [1]. Originally,

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

GPUs are specifically designed for 3D graphics. The performance enhancement of these devices has mainly focused on the highly parallel tasks of the 3D graphics calculations. Some limitations of GPUs, such as the lack of integers, logical operations, and fixed-point arithmetic make GPUs ill-suited for some general purpose computation. In particular, programmers need to describe a general purpose calculation in graphics terms, which requires the programmer to understand both the GPU architecture and graphics programming. Thus it is desirable to hide many of the architectural features and simplify the programming methods, and vendors of GPUs and organizations of programmers have attempted to develop attractive GPGPU programming languages.

2.3 GPU Programming Language

The modern graphics programming languages were intended for the improvement of visual output effects. The graphics programming languages were also able to program the parts of the graphics pipeline in the GPU. Early GPU programming languages were based on shaders in various off-line rendering systems, such as the Renderman shading language by RenderMan [8]. The Renderman shading language consists of pre-defined standalone functions and five types of shaders: surface, light, volume, imager, and displacement shaders. In later GPU programming languages, shaders were divided into two parts, the vertex shader code and pixel shader code. Such division changed the structure of the GPU into one with two processing units for each shader. This feature became important when branching comes into play [9]. The resulting code was not always effective because sometimes branching caused inappropriate mixing of

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

instructions for each processing unit. So this division forced the programmer to manually optimize the code for higher performance.

GPU programming languages can be divided into two groups according to their intended purposes: graphical purpose and general purpose. The GPU programming languages that are intended for graphical purposes are the most popular languages for common graphics hardware. Several languages, such as Microsoft Direct3D-HLSL, NVIDIA Cg, SGI OpenGL-GLSL, and RapidMind Sh, were introduced by graphics hardware vendors and software developers [10, 11, 12, 13]. The early GPU programming languages that are intended for general purposes depend on modifying the codes in shaders. This arrangement makes the implementation complicated because it needs additional code to bypass the nature of GPUs. There are some attempts to add high level abstraction to GPU programming languages. The abstraction can hide the graphical nature of GPUs [14]. The high-level scheme also provides an attractive API for programmers. Although this effort simplifies the implementation process, the performance of abstracted high-level languages is usually lower than that of direct GPU languages. The followings sections review some recent approaches of abstracted high-level languages for GPGPU applications

CTM : AMD (originally ATI) Close To Metal (CTM) was an approach that enabled low-level efficient GPU programming without any graphics overhead [15]. CTM aimed to give developers direct access to the native instruction set and memory of the massively parallel computational elements in recent AMD GPUs. However, it was short-lived as a beta version and a predecessor of AMD Stream Computing SDK.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

BrookGPU : The Brook is the Stanford University graphics group's stream-oriented language intended for stream processing that is developed for specialized high performance stream machines [16]. The pipeline nature of the GPU allows almost direct conversion of the Brook to the GPU with only a few limitations based on features of the GPU thus creating a general tool for implementing general processing algorithms on the GPU.

Brook+ : The Brook+ is an AMD hardware optimized version of the Brook language [17]. It is an implementation by AMD of the BrookGPU spec on compute abstraction layer with some enhancement for AMD GPU. It supports integer and double precision processing for AMD GPU and it will also support NVIDIA GPU.

Stream Computing SDK : AMD Stream Computing SDK is their first production of GPGPU language to be run on Windows XP. The SDK includes Brook+, itself a variant of the C language, open-sourced and optimized for stream computing [17]. It includes the AMD Core Math Library (ACML), AMD Performance Library (APL) with optimizations for the AMD FireStream, and the COBRA video library for video transcoding acceleration. Another important part of the SDK, the Compute Abstraction Layer (CAL), is a software development layer aimed for low-level access, through the CTM hardware interface, to the GPU architecture for performance tuning of software written in various high-level programming languages.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

RapidMind : RapidMind, based on the programmable GPU language Sh, is a multi-core development platform for expressing data-parallel computations as C++ code and executing them on multi-core processors. RapidMind currently runs not only on GPUs from both ATI and NVIDIA but also on multi-core CPUs and on Cell Broadband Engine. RapidMind is commercial software.

PeakStream : PeakStream is also a commercial platform enabled to program a high performance, multi-core, and parallel processors, and convert them into radically powerful computing engines for computationally intense applications. PeakStream originally targeted the ATI GPUs. Now it is available in various types of multi-core processors and is supported in Linux and Windows environments.

CUDA : NVIDIA CUDA is an extended C language environment that unlocks the processing power of GPUs to solve the most complex computation-intensive challenges. CUDA provides developers with both the deterministic low-level API and the high-level API for repeatable access to hardware, which is necessary to develop essential high-level programming tools such as compilers, debuggers, math libraries, and application platforms. At the present time it works only with recent NVIDIA G80 GPU series.

OpenCL : OpenCL (Open Computing Language) is a language for GPGPU based on the C language and proposed by Apple in cooperation with others [18]. The purpose is to make open industry standards for GPGPU, 3D graphics, and computer audio, and to extend the power of the GPU beyond graphics. OpenCL is scheduled to be introduced in

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Mac OS X v10.6 ('Snow Leopard'). According to the sources, Snow Leopard further extends support for modern hardware with OpenCL, which lets any application tap into the vast gigaflops of GPU computing power previously available only to graphics applications. The initial OpenCL implementation is reportedly built on LLVM and Clang compiler technology.

2.4 G80 Architecture

In this section, brief descriptions of the architecture of the NVIDIA G80 GPU, the GeForce 8800 series graphics card used in the experiments, and the Compute Unified Device Architecture (CUDA) language, which is used to program the G80 or later GPUs, are provided. This section is a summary of the information available in the CUDA documentation by NVIDIA [19]. The G80 uses the standard 575Mhz clock core, a high speed 1.35 GHz clock shader, and 1.8GHz 384-bit DDR3 Memory with a theoretical bandwidth of 86.4GB/s. NVIDIA claimed that G80 can perform nearly 518 GFLOPS (1.35 GHz * 128 SPs * 3 FLOPS).

Figure 3 illustrates the high level design of the G80 architecture. The G80 contains 16 streaming multiprocessors (SM) and each SM has 8 streaming processors (SP), for a total of 128 SPs. These SPs run at a 1.35GHz clock, and they are mated with 1.8GHz 384-bit DDR3 memory. Each SP can access the registers and perform 32-bit single precision floating point calculation. In addition, the arithmetic unit is also able to perform integer arithmetic operations on the G80. Each SM contains two Special Function Units (SFUs) that support more complicated float pointing calculations for the mathematical functions. Each SM has a total of 8,192 registers and they can be

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

dynamically partitioned when the threads are running. The latency rate of the register is assumed to be about 2 clock cycles; however, NVIDIA has not provided more detailed information. Each SM can execute one or more threads and blocks within the warp size. A warp is a group of 32 threads and the SM executes the threads by the size of a warp sequentially. That is, a SM can run only one instruction at the same time for all threads. Because of this property, execution could be slowed when the threads follow different control paths. The thread execution will be serialized in this case. The consideration of warp size is also important in preventing bank conflicts.

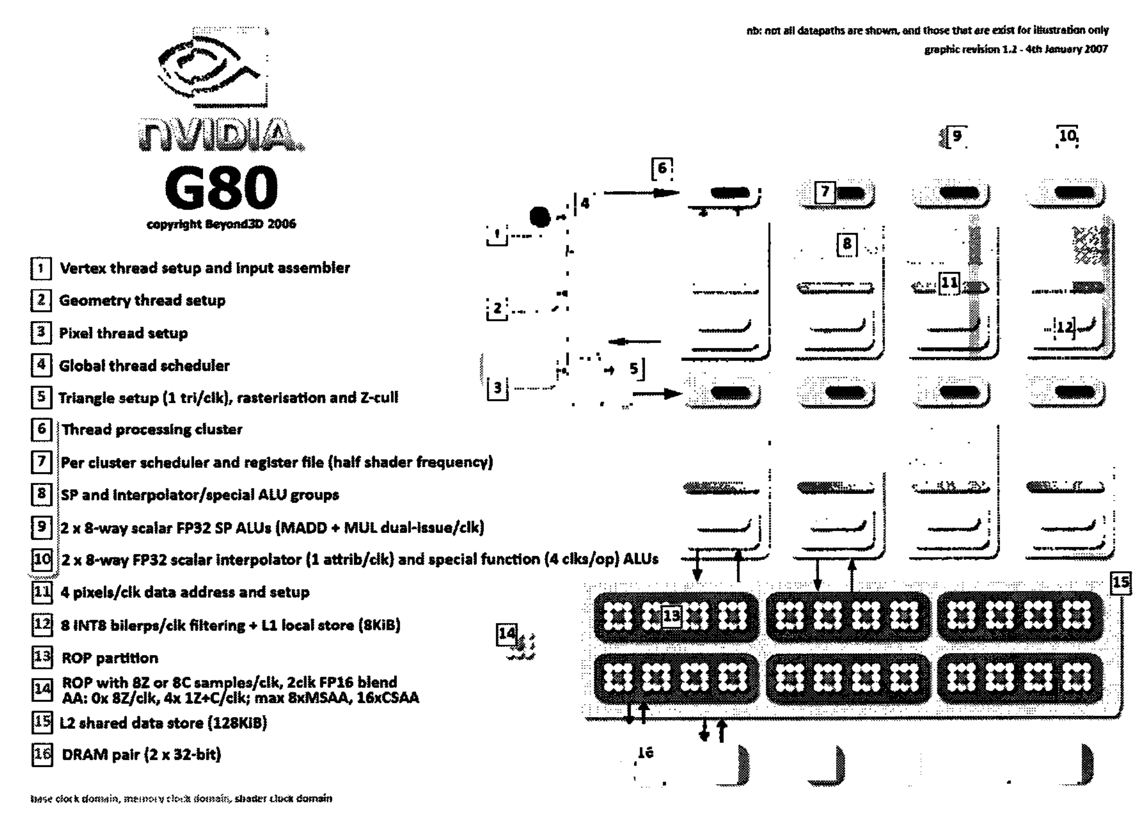


Figure 3. An overview of the G80 architecture [20]

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

2.5 CUDA Programming Model

NVIDIA CUDA is a general purpose scalable parallelized programming model for highly parallel processing applications. It is an abstract high-level language that has distinctive abstractions, such as a hierarchy of thread blocks, barrier synchronization, and shared memory structures. CUDA is well-suited for programming on multiple threaded multi-core GPUs. Many researchers and developers attempt to use CUDA for demanding computational applications in order to achieve dramatic speedups.

2.5.1 CUDA Structure

In earlier GPGPU designs, general purpose applications on the GPU must be mapped through the graphics Application Programming Interface (API) because traditional GPUs had highly specialized pipeline designs. This structural property made it necessary for a programmer to write the programs to fit the graphics API. Sometimes the programmer needed to rework all the programs. The G80 has a global memory that can be addressed directly from the multiple sets of processor cores. The global memory makes the G80 architecture a more flexible and general programming model than previous GPGPU models. The global memory also allows programmers to implement data-parallelized kernels for the GPU easily. A processor core in the GPU can share the same traits with other processor cores. Thus, multiple processor cores can run independent threads in parallel at the same time.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

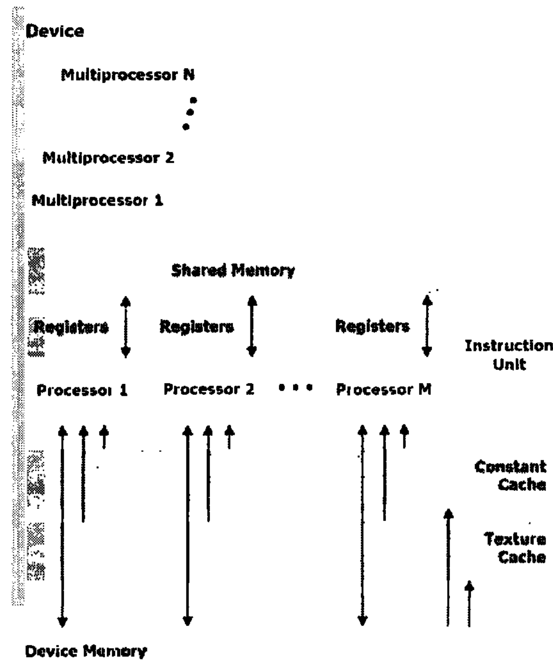


Figure 4. Multiprocessor design for CUDA

2.5.2 SPMD Design

The GPGPU application systems use the GPU as a group of fast multiple coprocessors that execute data-parallelized kernel code. So the programmers can access the GPU cores via a single source code encompassing both CPU and GPU code. A kernel function operates in a Single Program Multiple Data (SPMD) fashion [21]. The SPMD concept extends Single Instruction Multiple Data (SIMD) by executing several instructions for each piece of data. A kernel function can be executed by the threads in order to run the data-parallelized operations. It is very efficient to utilize many threads in one operation. For full utilization of the GPU, fine-grained decomposition of work is required, which might cause redundant instructions in the threads. However, there are several restrictions in using the kernel functions. A CUDA kernel function can not be

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

recursive and it can not use static variables. Kernel functions also need a non-variable type of parameters. The host (CPU) code can copy data between the CPU's memory and the GPU's global memory via API calls.

2.5.3 CUDA Threads

Thread execution on the G80 architecture consists of a three-level hierarchy, grid, block, and thread. The grid is the highest level. A block is a part of the grid, and there can be a maximum of $2^{16} - 1$ blocks in the grid, organized in a one or two dimensional array. A thread is a part of a block, and there can be up to 512 threads in a block, organized in a one, two, or three dimensional array. Threads and blocks have their unique location numbers as threadID and blockID. The threads in the same block share the data through the shared memory. In CUDA, the function __syncthreads() performs the barrier synchronization, which is the only synchronization method in CUDA. Additionally, the threads can be grouped into warps of up to 32 threads.

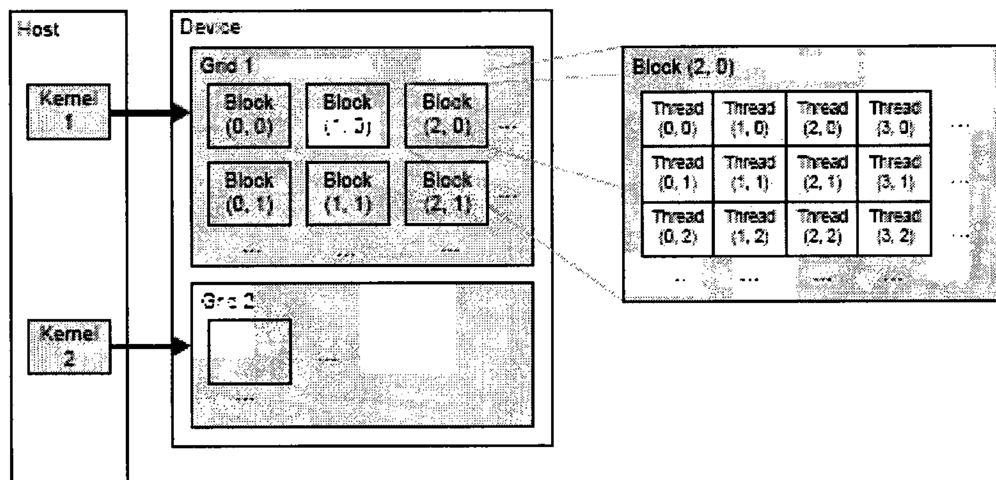


Figure 5. Organization of CUDA threads

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

2.4.2 CUDA Memory Architecture

The G80 has several specially designed types of memory that have different latencies and different limitations [22]. The registers are fast and very limited size read-write per-thread memory in each SP. The local memory is slow, not cached, limited size read-write per-thread memory. The shared memory is a low-latency, fast, very limited

Table 1

Memory types in CUDA [22]

Memory	Location	Size	Latency	Read-Only	Description
Global	off-chip	768 MB total	200~300 cycles	No	Large DRAM. All data reside here at the beginning of kernel execution. Directly addressable from a kernel using pointers. Backing store for constant and texture memories. Used more efficiently when multiple threads simultaneously access contiguous elements of memory, enabling the hardware to coalesce memory accesses to the same DRAM page.
Shared	on-chip	16 KB per SM	≈ register latency	No	Local scratched that can be shared among threads in a thread block. Organized into 16 banks. It is often possible to organize both threads and data so that bank conflict seldom or never occurs.
Constant	on-chip cache	64 KB Total	≈ register latency	Yes	8KB cache per SM, with data originally residing in global memory. The 64KB limit is set by programming model. Often used for lookup table The cache is single-ported, so simultaneous requests within an SM must be to the same address or delay occurs.
Texture	on-chip cache	Up to Global	> 100 cycles	Yes	16KB cache per two SMs, with data originally residing in global memory. Capitalizes on 2D locality. Can perform hardware interpolation and have configurable returned-value behavior at the edge of textures, both of which are useful in certain applications such as video encoder
Local	off-chip	Up to Global	Same as global	No	Space for register spilling, etc

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

size, read-write per-block memory in each SM. Shared memory is useful for data among the threads in a block. The global memory is a large, long-latency, slow, non-cached, read-write per-grid memory. It is used for communication between CPU and GPU as a default storage location. Constant and texture memory is used for only one-sided communication. The constant memory is slow, cached, limited size, read-only per-grid memory. The texture memory is slow, cached, large size, read-only per-grid memory. Table 1 indicates the CUDA memory types.

L. Marziale, et al.[23] summarized the memory architecture of CUDA with restrictions and associated costs as follows:

* Private registers are local to a particular thread and readable and writeable only by that thread.
* Constant memory is initialized by the host and readable by all threads in a kernel. Constant memory is cached and a read costs one memory read from device memory only on a cache miss, otherwise it costs one read from the constant cache. For all threads of a particular warp, reading from the constant cache is as fast as reading from a register as long as all threads read the same address. The cost scales linearly with the number of different addresses read by all threads.
* Shared memory can be read and written by threads executing within a particular thread block. The shared memory space is divided into distinct, equal-sized banks which can be accessed simultaneously. This memory is on-chip and can

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

be accessed by threads within a warp as quickly as accessing registers, assuming there are no bank conflicts. Requests to different banks can be serviced in one clock cycle. Requests to a single bank are serialized, resulting in reduced memory bandwidth.

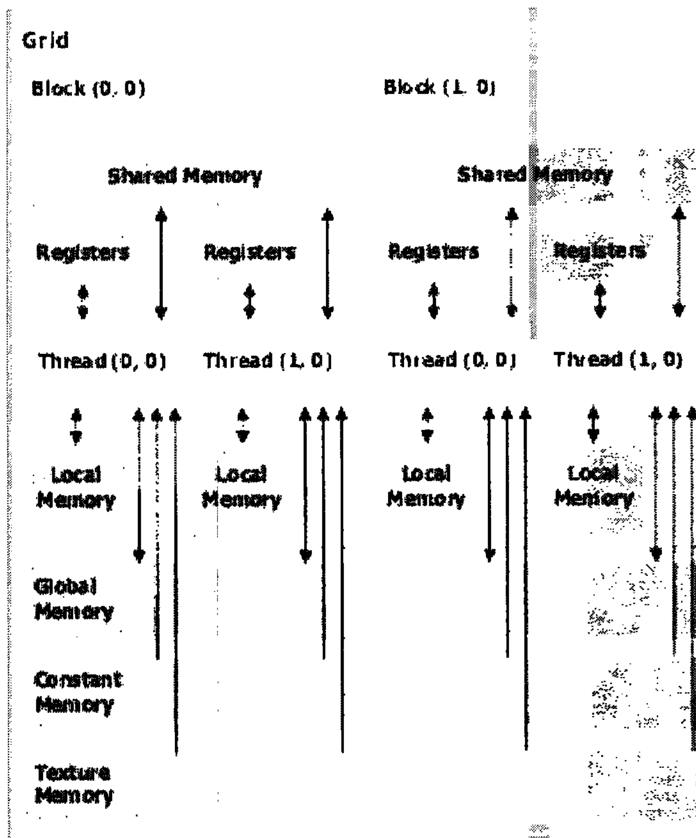


Figure 6. Memory architecture of the G80

* Texture memory is a global, read-only memory space shared by all threads. Texture memory is cached and texture accesses cost one read from device memory only on a texture cache miss. Texture memory is initialized by the host.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Hardware texture units can apply various transformations at the point of texture memory access.

* Finally, global memory is uncached device memory, readable and writeable by all threads in a kernel and by the host. Accesses to global memory are expensive, requiring 200 or more cycles of memory latency.

2.5.3 CUDA Toolkit and SDK

Nvidia provides the CUDA toolkit and CUDA SDK as the interface and the examples for the CUDA programming environment. They are supported on Windows XP, Windows Vista, Mac OS X, and Linux platforms. The CUDA toolkit contains several libraries and the CUDA compiler NVCC. A CUDA program is written as a C/C++ program. NVCC separates the code for the host (regular C/C++ code) and the device (a CUDA native). The CUDA toolkit also supports a simulation mode of CUDA. In the simulation mode, a programmer can debug the code. The CUDA toolkit and SDK also provide GPU management functions, memory management functions, external graphical API supported functions, and some useful sample codes.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

CHAPTER 3

SEQUENCE ALIGNMENT

3.1 Bioinformatics

Bioinformatics is an interdisciplinary research area among many scientific fields, including biology, chemistry, computer science, mathematics, physics, and statistics. Bioinformatics can be defined as the application of information technology to the collection, identification, classification, organization, and analysis of biological data. Due to the rapidly increasing amounts of the biological data, the high computational resources are required in bioinformatics research in order to process the large data sets and to reduce the processing time. The main research domains in bioinformatics include sequence alignment, gene finding, genome assembly, protein structure alignment, protein structure prediction, analysis of gene and protein expression and protein-protein interactions, and the modeling of evolution.

3.2 Sequence Alignment

Sequence alignment has become a basic tool of molecular biology. Its goal is to find the similarity between two sequences in protein or nucleotide databases, and such

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

similarity may help identify the functional, structural, or evolutionary relationships between the sequences. The similarity is generally called homology. Sequence alignment algorithms are mostly used to align two sequences at a time. The quality of the alignment is represented as an aligned score, calculated from matches, mismatches, and gaps in the sequence alignment matrix. An optimal alignment algorithm finds the alignments that maximize the score.

Pairwise sequence alignment can be generally classified as global alignment and local alignment. In general, global alignment is applied when the lengths of two sequences are roughly equal. It can find the similarity throughout the entire lengths of the sequences. Based on dynamic programming, the Needleman-Wunsch algorithm [24] finds the optimal global alignment between two sequences. On the contrary, local alignment finds short segments in the sequences that have high similarity. The Smith-Waterman algorithm [25], also based on dynamic programming, finds the optimal local alignments.

3.3 Dynamic Programming

Dynamic programming is a technique to find the optimal solutions from complex problems that have overlapping sub-problems and optimal sub-structures [26]. Dynamic programming can speed up the computation when it is costly to enumerate all possible sub-structures. When aligning DNA or RNA sequences, a match of residues is assigned a positive score, a mismatch of residues receives a negative score, and a gap incurs a negative penalty. When aligning protein sequences, a substitution matrix, constructed from observed substitutions between different amino acids in evolutionarily related

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

sequences, is employed instead of the simple match and mismatch scores. Dynamic programming is guaranteed to find the maximum aligned score of two sequences with respect to a particular scoring scheme. Although dynamic programming can find the optimal alignments, it is slow for long sequences.

3.4 The Smith-Waterman Algorithm

The Smith-Waterman algorithm, proposed by T. Smith and M. Waterman in 1981, finds the optimal local alignment between two sequences. This algorithm works by filling the sequence alignment matrix with scores. One sequence is placed at the top of the sequence alignment matrix, and the other sequence is place at the left side of the matrix. The cells in the first row and the first column are filled with 0 for initialization of the sequence alignment matrix. The configuration is illustrated in Figure 7.

	∅	A	T	C	T	C	G	T	A	T	G	A	T	G
∅	0	0	0	0	0	0	0	0	0	0	0	0	0	0
G	0	0	0	0	0	0	2	1	0	0	2	1	0	2
T	0	0	2	1	2	1	1	4	3	2	1	1	3	2
C	0	0	1	4	3	4	3	3	3	2	1	0	2	2
T	0	0	2	3	6	5	4	5	4	5	4	3	2	1
A	0	2	2	2	5	5	4	4	7	6	5	6	5	4
T	0	1	4	3	4	4	4	6	5	9	8	7	8	7
C	0	0	3	6	5	6	5	5	5	8	8	7	7	7
A	0	2	2	5	5	5	5	4	7	7	7	10	9	8
C	0	1	1	4	4	7	6	5	6	6	6	9	9	8

Figure 7. Tracking the alignment scores

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

After initialization, the cells in the matrix are filled with scores in a top-down and left-to-right fashion where the scores are calculated according to the formulas in Figure 8. After filling all the cells in matrix, the maximum scoring cells are selected, and they represent the ends of the optimal local alignments. Backtracking is applied till a cell with the score 0, which represents the beginning of the optimal local alignment. The Smith-Waterman algorithm takes O(MN), where M and N are the lengths of the sequences.

$A: a_1, a_2, a_3, \dots, a_{n-1}, a_n$ The first sequence string

$B: b_1, b_2, b_3, \dots, b_{m-1}, b_m$ The second sequence string

α : Initial Gap penalty

β : Extension Gap penalty

$sbt(a_i, b_j)$: A substitution matrix of a_i and b_j

$$E_{i,j} = MAX\begin{pmatrix} H_{i,j-1} - \alpha \\ E_{i,j-1} - \beta \end{pmatrix}$$

$$F_{i,j} = MAX\begin{pmatrix} H_{i-1,j} - \alpha \\ F_{i-1,j} - \beta \end{pmatrix}$$

$$H_{i,j} = MAX\begin{pmatrix} 0 \\ E_{i,j} \\ F_{i,j} \\ H_{i-1,j-1} + sbt(a_i, b_j) \end{pmatrix}$$

Where, $1 \le i \le m$

$1 \le j \le n$

Figure 8. Smith-Waterman scoring schemes

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

During the computation of the Smith-Waterman algorithm, a cell in the sequence alignment matrix has a dependency on three cells, the one to the left, the one above, and the one from the upper left diagonal, illustrated in Figure 9. However, the elements in the same anti-diagonal are all independent of each other, and they are dependent on the previous two anti-diagonals.

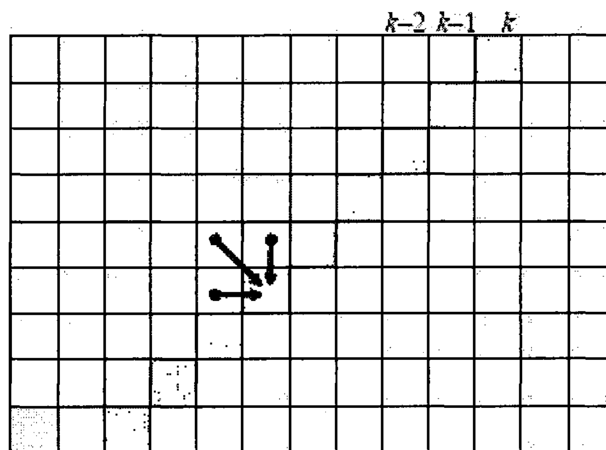


Figure 9. Data dependency of each cell [27]

3.5 Sequence Alignment on the GPU

There were several studies to conduct sequence alignment on GPU in the literature. W. Liu et al.[27] implemented sequence alignment in a high-level GPU programming language, GLSL, using texture buffer on Geforce 6800GTO and Geforce 7900GTX. They used a shifted sequence matrix for processing anti-diagonals, and RGBA channels of 2D buffers for the computation of H, E, F, and the maximum values. M.C. Schatz et al.[28] proposed high-throughput sequence alignment system using CUDA on GeForce 8800GTX. They implemented MUMmer (Maximal Unique Matches), a genome alignment system, on GPU using CUDA. MUMmer used a suffix tree data structure to

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

align multiple query sequences against a single reference sequence. Their MUMmerGPU was more than 10 times faster than the CPU version. Manavski and Valle succeeded to implement the Smith-Waterman algorithm on GeForce 8800GTX using CUDA [29]. They adopted the vector parallel model, a pre-defined query-profile in order to reduce the substitution matrix lookup, a pre-ordered database for fast computation, and an adequate maximal use of memory per cycle. They used 64 threads in a block and 450 blocks in the grid for a total of 28,800 threads on GeForce 8800GTX. In addition, they used two GeForce 8800GTX as multiple GPUs. Their implementation was from 2 to 30 times faster than other previous sequence alignment systems.

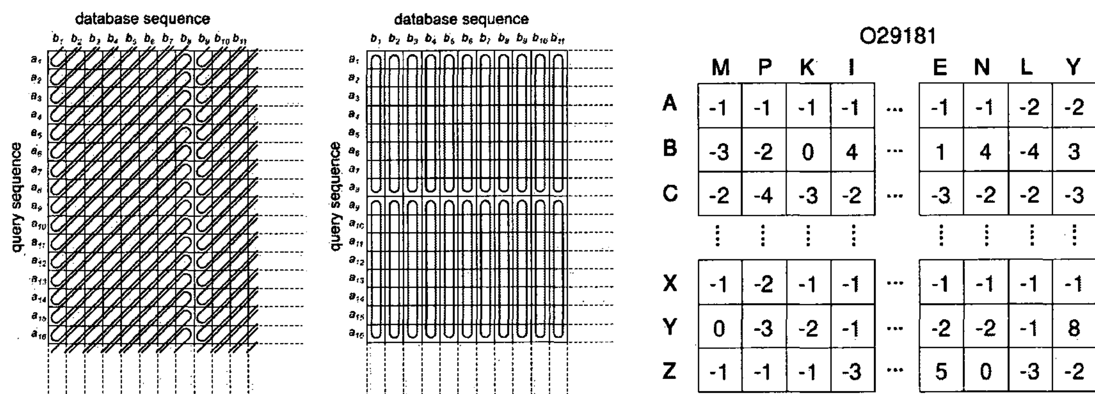


Figure 10. Vector parallel model and Pre-defined Query-profile [29, 30]

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

CHAPTER 4

SMITH-WATERMAN ON THE GPU USING CUDA

4.1 Parallelized Smith-Waterman Algorithm

In this chapter, we describe the implementation of the parallelized Smith-Waterman algorithm on GPU. The Smith-Waterman algorithm constructs a sequence alignment matrix from which the optimal local alignments are extracted. In the sequence alignment matrix, the value of a cell depends on three other cells: the one to the left, the one above, and the one from the upper left diagonal. However, the cells in the same anti-diagonal are all independent, and they are dependent on the previous two anti-diagonals. Thus, we will apply parallel computation to the anti-diagonals using multiple streaming processors in the GPU.

4.2 Wavefront Algorithm

There is an approach called wavefront that uses diagonal sweeps to implement a parallelized dynamic programming [31]. In wavefront using diagonal sweep, the memorization method can be used for the multi-thread parallelized Smith-Waterman algorithm on GPU. This method involves more than one dimensional memorization

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

variables, and it also reduces the recursive calls between parameters. The method can create a pattern of computation, advancing diagonally on the multidimensional memorization variable space. For the parallel formulation of the wavefront algorithm, the distribution of threads is very important. Thread idle time needs to be minimized and an equal amount of work needs to be assigned to each thread.

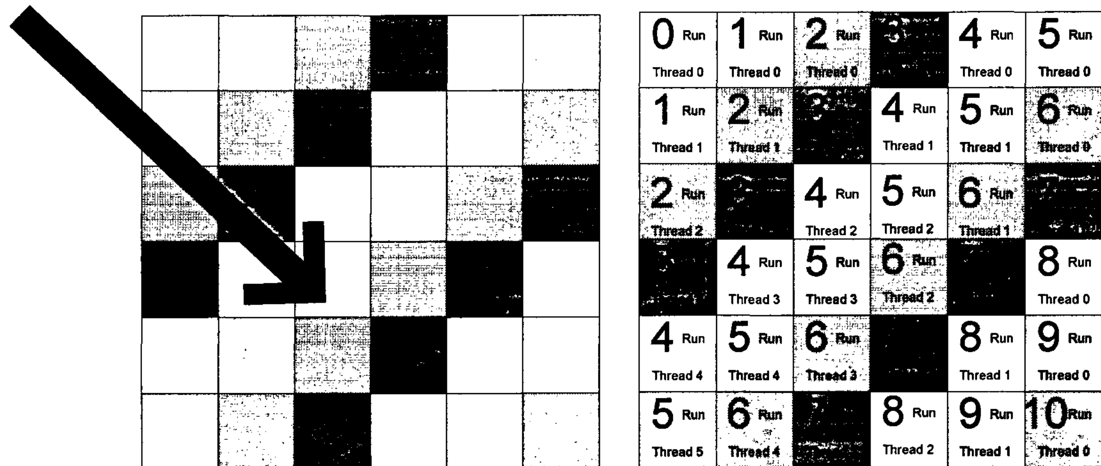


Figure 11. Diagonal sweep wavefront

Same number will be calculated in one run time

4.3 Smith-Waterman CUDA System

System Structure

The Smith-Waterman implementation consists of five modules: sequence reading module, pre-processing module, CPU Smith-Waterman processing module, GPU Smith-Waterman processing module, and output display module.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

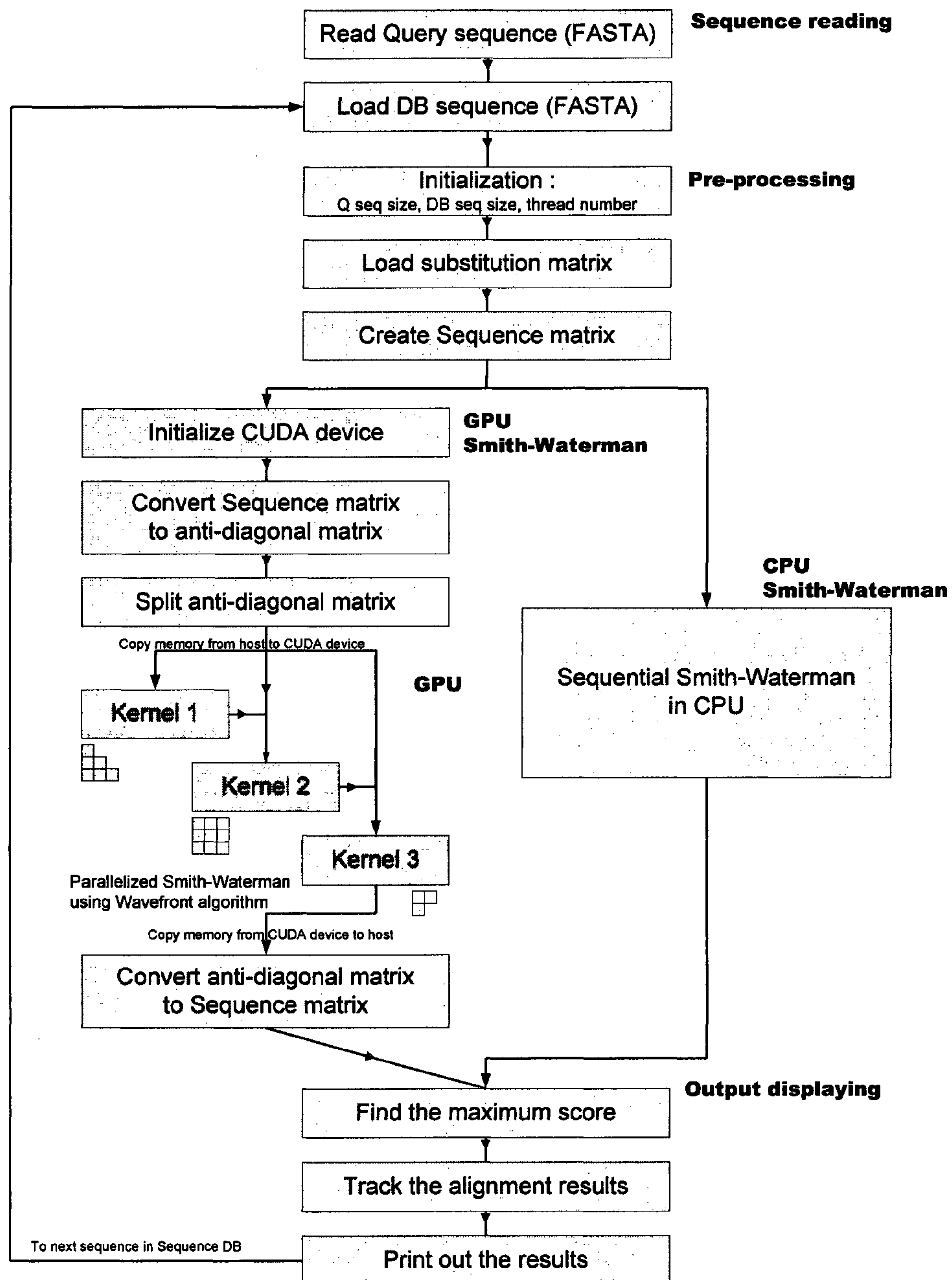


Figure 12. Diagram of Smith-Waterman CUDA system

Sequence Reading : The sequence reading module reads a query sequence file and the database files in the FASTA format [32]. It checks that the files are correct FASTA files.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

It reads the protein (or DNA) sequences from the files, and stores the sequences to variables containing sequence names, sequence lengths, and the sequences.

Pre-processing : The second module is the pre-processing module. This module prepares all the required variables. It initializes the sequence alignment matrix, and it loads the substitution matrix in order to create the substitution matrix array.

CPU Smith-Waterman : The CPU implementation of Smith-Waterman is the third module. It uses the classic method in the implementation.

GPU Smith-Waterman : The fourth module is the GPU implementation of Smith-Waterman. The code is written in CUDA, utilizing multiple streaming processors in the GPU. G80 and CUDA support the Single Programming Multiple Data (SPMD) parallelization.

Output Display : The last module is output display. It shows the results of the alignment: the number of matched or ignored sequences, the running time, alignment scores, and so on.

4.4 CUDA Implementation

An advantage of using CUDA is that G80 supports many simultaneous threads. The basic concept of our implementation is how to process all cells in an anti-diagonal at the same time using threads.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

4.4.1 Pre-processing

The Sequence matrix: From two input sequences of lengths m and n, we can create the sequence alignment matrix using an `int4` type 1-dimensional dynamic array. The size of the matrix is m plus one (all 0 in the first row) times n plus one (all 0 in the first column). It stores the score, the x coordinate, the y coordinate, and the direction (from the top, from the left, or from the upper left diagonal) for the purpose of backtracking.

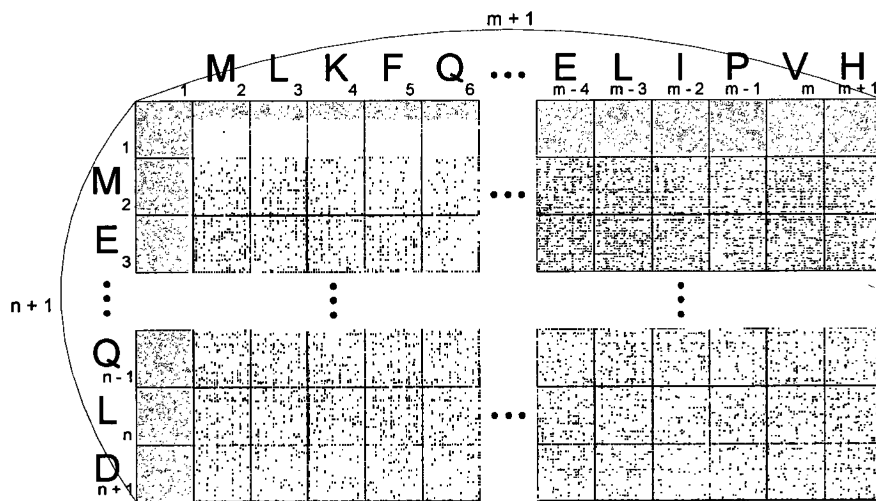


Figure 13. The sequence matrix

Pre-defined macro: Smith-Waterman needs a lot of comparisons of scores in each cell. We use a pre-defined C macro to speed up the comparison of two numbers. In GPU programming, it may be advantageous to avoid function calls.

```
#define max(A,B) ((A)>(B)?(A):(B))
```

Figure 14. Pre-defined macro code

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
/*A*/   5,-2,-1,-2,-1,-1,-1, 0,-2,-1,-2,-1,-1,-3,-1, 1, 0,-3,-2, 0,-2,-1,-1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
/*R*/  -2, 7,-1,-2,-4, 1, 0,-3, 0,-4,-3, 3,-2,-3,-3,-1,-1,-3,-1,-3,-1, 0,-1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
/*N*/  -1,-1, 7, 2,-2, 0, 0, 0, 1,-3,-4, 0,-2,-4,-2, 1, 0,-4,-2,-3, 4, 0,-1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
/*D*/  -2,-2, 2, 8,-4, 0, 2,-1,-1,-4,-4,-1,-4,-5,-1, 0,-1,-5,-3,-4, 5, 1,-1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
/*C*/  -1,-4,-2,-4,13,-3,-3,-3,-3,-2,-2,-3,-2,-2,-4,-1,-1,-5,-3,-1,-3,-3,-2, 0, 0, 0, 0, 0, 0, 0, 0, 0,
/*Q*/  -1, 1, 0, 0,-3, 7, 2,-2, 1,-3,-2, 2, 0,-4,-1, 0,-1,-1,-1,-3, 0, 4,-1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
/*E*/  -1, 0, 0, 2,-3, 2, 6,-3, 0,-4,-3, 1,-2,-3,-1,-1,-1,-3,-2,-3, 1, 5,-1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
/*G*/   0,-3, 0,-1,-3,-2,-3, 8,-2,-4,-4,-2,-3,-4,-2, 0,-2,-3,-3,-4,-1,-2,-2, 0, 0, 0, 0, 0, 0, 0, 0, 0,
/*H*/  -2, 0, 1,-1,-3, 1, 0,-2,10,-4,-3, 0,-1,-1,-2,-1,-2,-3, 2,-4, 0, 0,-1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
/*I*/  -1,-4,-3,-4,-2,-3,-4,-4,-4, 5, 2,-3, 2, 0,-3,-3,-1,-3, 2,-4, 0, 0,-1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
/*L*/  -2,-3,-4,-4,-2,-2,-3,-4,-3, 2, 5,-3, 3, 1,-4,-3,-1,-2,-1, 1,-4,-3,-1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
/*K*/  -1, 3, 0,-1,-3, 2, 1,-2, 0,-3,-3, 6,-2,-4,-1, 0,-1,-3,-2,-3, 0, 1,-1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
/*M*/  -1,-2,-2,-4,-2, 0,-2,-3,-1, 2, 3,-2, 7, 0,-3,-2,-1,-1, 0, 1,-3,-1,-1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
/*F*/  -3,-3,-4,-5,-2,-4,-3,-4,-1, 0, 1,-4, 0, 8,-4,-3,-2, 1, 4,-1,-4,-4,-2, 0, 0, 0, 0, 0, 0, 0, 0, 0,
/*P*/  -1,-3,-2,-1,-4,-1,-1,-2,-2,-3,-4,-1,-3,-4,10,-1,-1,-4,-3,-3,-2,-1,-2, 0, 0, 0, 0, 0, 0, 0, 0, 0,
/*S*/   1,-1, 1, 0,-1, 0,-1, 0,-1,-3,-3, 0,-2,-3,-1, 5, 2,-4,-2,-2, 0, 0,-1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
/*T*/   0,-1, 0,-1,-1,-1,-1,-2,-2,-1,-1,-1,-1,-2,-1, 2, 5,-3,-2, 0, 0,-1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
/*W*/  -3,-3,-4,-5,-5,-1,-3,-3,-3,-3,-2,-3,-1, 1,-4,-4,-3,15, 2,-3,-5,-2,-3, 0, 0, 0, 0, 0, 0, 0, 0, 0,
/*Y*/  -2,-1,-2,-3,-3,-1,-2,-3, 2,-1,-1,-2, 0, 4,-3,-2,-2, 2, 8,-1,-3,-2,-1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
/*V*/   0,-3,-3,-4,-1,-3,-3,-4,-4, 4, 1,-3, 1,-1,-3,-2, 0,-3,-1, 5,-4,-3,-1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
/*B*/  -2,-1, 4, 5,-3, 0, 1,-1, 0,-4,-4, 0,-3,-4,-2, 0, 0,-5,-3,-4, 5, 2,-1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
/*Z*/  -1, 0, 0, 1,-3, 4, 5,-2, 0,-3,-3, 1,-1,-4,-1, 0,-1,-2,-2,-3, 2, 5,-1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
/*X*/  -1,-1,-1,-1,-2,-1,-1,-2,-1,-1,-1,-1,-1,-2,-2,-1, 0,-3,-1,-1,-1,-1,-1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
/*-*/   0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
/*-*/   0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
/*-*/   0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
/*-*/   0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
/*-*/   0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
/*-*/   0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
/*-*/   0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
/*-*/   0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
/*-*/   0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
      //A  R  N  D  C  Q  E  G  H  I  L  K  M  F  P  S  T  W  Y  V  B  Z  X  -  -  -  -  -  -  -  -  -
```

Figure 15. BLOSUM50 Substitution matrix for SW CUDA

Substitution matrix: The substitution matrix is stored as a 32 by 32 array as shown in Figure 16. There are only 23 symbols in the alphabet. However, we add 9 padding symbols so that the size of the matrix is a power of 2 for faster addressing. In CUDA, float calculation is faster than integer calculation. Thus, we convert an int type substitution matrix array to a float type array.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

4.4.2 Extraction of the Anti-diagonals

In using a diagonal sweep wavefront algorithm, we need to extract anti-diagonals and form an anti-diagonal matrix. The conversion from the original sequence alignment matrix to the anti-diagonal matrix is illustrated in Figure 17. The number of threads will be n, and the number of anti-diagonals will be m-n. We store this matrix in a float 1-dimensional array, and the array consists of three parts.

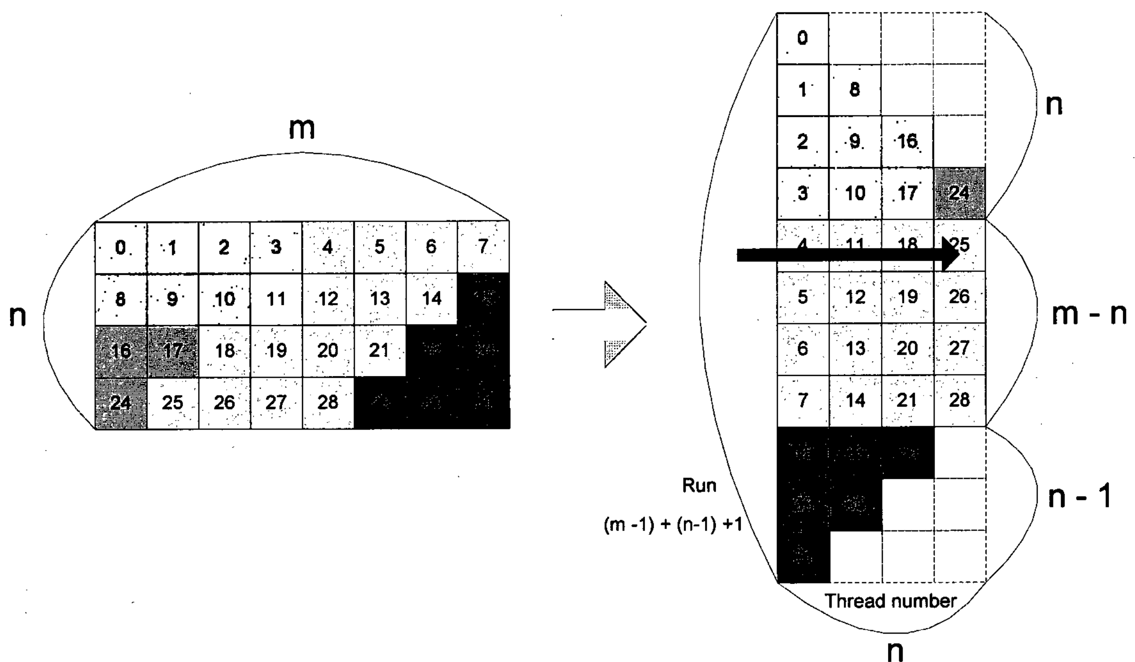


Figure 16. Anti-diagonal matrix conversion

First anti-diagonals: From the upper left corner of the sequence alignment matrix to the first full anti-diagonal, the code uses fewer threads than what can be deployed and will increase the number of threads one by one up to the end of the first part. The first and last

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

values of each row are 0, because they are initialized with 0 in the sequence alignment matrix.

Mid anti-diagonals: From the second row of full anti-diagonals to the end of the full anti-diagonals, the code uses all available threads.

Last anti-diagonals: From the end of the full anti-diagonals to the lower right corner of the matrix, the code does not use all threads. The number of threads is decreased one by one.

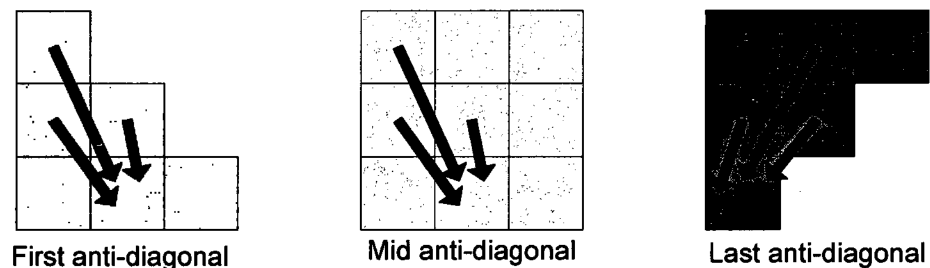


Figure 17. Data dependency of each anti-diagonal part

4.4.3 CUDA Kernels

The sizes of the three anti-diagonal parts are different, and the data dependency of the last anti-diagonal part is different from the first two parts. If we use the same CUDA kernel for all three parts, several conditional statements will be included to distinguish the parts. However, when conditional statements are used in a kernel function, different threads may follow different conditional paths during execution, leading to divergence of SPMD. For this reason we use three kernels for the three parts without conditional statements in the code. The 32 bit integer multiplication on the GPU takes 16 clock cycles, whereas floating point addition, subtraction, and multiplication take 4 clock

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

cycles [19]. Thus integer multiplication should be avoided. Since two previous anti-diagonals are needed to calculate the current anti-diagonal, calculation should start from the third row of each part. Because the first and last elements in the first part of the anti-diagonal matrix are initialized to zeroes, we do not need to calculate them.

The kernels use many variables to store data. Before calling the kernels, we need to allocate memory in the GPU device memory, and copy all the data from the memory of the host computer to the device using the function cudaMemcpy(). Since the partial anti-dagonal 1-dimensional array contains critical data, it will lead to good performance if bank conflicts are avoided. We can check for bank conflict using the function CUT_BANK_CHECKER().

```
#define CHECK_BANK_CONFLICTS 0
#if CHECK_BANK_CONFLICTS
#define H(i) CUT_BANK_CHECKER(((float*)&d_H[0], i)
#else
#define H(i) d_H[i]
#endif
```

Figure 18. Bank conflict checker code

4.4.4 Substitution Matrix

The location of the current cell is stored as the X and Y coordinates in an int2 type variable, and actual sequences are stored as strings in a char type array. We can use a subscript to the sequence array to retrieve the actual sequence symbol. The substitution matrix is a 32 by 32 array, and each amino acid is converted to a number as shown in

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Table 2. Therefore, when we look up the score in the substitution matrix, we use the numbers that correspond to the symbols of the amino acids. The scores are stored as floating point numbers.

Table 2

Protein conversion in Substitution matrix

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
A	R	N	D	C	Q	E	G	H	I	L	K	M	F	P	S	T	W	Y	V	B	Z	X	-

```
__global__ void
swTestmid(float* d_H, int aSize, int bSize, int2* qLoc, float* sbt)
{
    const unsigned int t_num = blockDim.x + 1; // Block index
    const unsigned int tid = threadIdx.x + 1; // Thread index
    unsigned int num = t_num + t_num;
    for(unsigned int i=3; i < (aSize - (bSize - 1) + 2); i++){
       num += t_num; // To reduce integer multiplication
       H(num + tid) = max(0.0, max(H((num - t_num) + tid - 1),
                       max(H( (num - t_num) + tid), H( (num - t_num - t_num) + tid -1)
                       - sbt[( (qLoc[num + tid].y) << 5) + (qLoc[num + tid].x) ] )));
       __syncthreads();
       }
}
```

Figure 19. Mid anti-diagonal kernel code

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

4.4.5 CUDA Configuration

In our implementation of the CUDA Smith-Waterman, there is only one block of 512 threads in the grid. The for loop in the kernel processes one row of the anti-diagonal matrix at a time, and the 512 cells in the row are computed by 512 threads simultaneously.

After running the three kernels, the resulted data in the GPU are copied to the host memory using the function cudaMemcpy(). The CUDA utility cutTimer() is used to determine the time used by the computation and data transfer. The results can be displayed on a screen or saved as a file.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

CHAPTER 5

RESULTS AND CONCLUSION

The performance of the GPU implementation of Smith-Waterman is affected by the number of simultaneous threads and the time for data transfers between the host and the device. Furthermore, the implementation is constrained by several hardware specific properties of the G80 and the CUDA language. In the present work, the efforts are focused on reducing the calculation cycles and the running time by making simple code and using the appropriate data types.

5.1 Environment

We have implemented the Smith-Waterman algorithm using CUDA on the Geforce 8800GT (600MHz G92 core with total 112 SPs in 14 SMs, 1.5GHz shader clocks, and 1GB 1.8GHz 256-bit DDR3 memory), a later version of G80, in an Alienware machine (Pentium-D 3.0GHz with 2GB RAM). We compared computation performance on three different lengths of query and DB sequences.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

5.2 Results

Table 3 shows that the parallelized Smith-Waterman with the wavefront algorithm on the GPU using 1 grid and 1 block can be from 1.5 to 3.7 times faster than the sequential Smith-Waterman on the CPU.

Table 3

Experimental result between CPU-SW and GPU-SW with 1 block

DB	Query (threads)	CPU-SW (ms)	GPU-SW (ms)	Speed
512	256	4.9055	1.4638	3.3
1024	256	8.9378	4.1006	2.1
2048	256	16.5376	9.3348	1.7
4096	256	31.7367	19.7967	1.6
8192	256	61.3310	40.2662	1.5
1024	512	18.6674	4.9486	3.7
2048	512	34.2762	14.4870	2.3
4096	512	66.0518	34.9783	1.8
8192	512	124.5451	77.9419	1.6

5.3 Conclusion

The present work is focused on parallelizing the Smith-Waterman algorithm with the following results. First, we presented a simple approach to parallelize Smith-Waterman with the wavefront algorithm. Second, we conducted preliminary experiments and found that the parallelized Smith-Waterman algorithm on the GPU is at least 1.5 times faster than Smith-Waterman on the CPU. It is likely that an even better

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

performance may be achieved by employing a more sophisticated arrangement of threads, blocks, and the grid.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

REFERENCES

[1] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Kruger, A. E. Lefohn, and T. J. Purcell, "A SURVEY OF GENERAL-PURPOSE COMPUTATION ON GRAPHICS HARDWARE", Computer Graphics Forum, 26(1): pp. 80 - 113, 2007

[2] V. Anand, "NVIDIA GEFORCE 8800 GTX / GTS (G80) - THE WORLDS'S FIRST DX10 GPU", <http://www.hardwarezone.com/articles/view.php?cid=3&id=2107>, Web article, Nov. 2006

[3] P. Trancoso and M. Charalambous, "EXPLORING GRAPHICS PROCESSOR PERFORMANCE FOR GENERAL PURPOSE APPLICATIONS", In Proceedings of the Euromicro Symposium on Digital System Design, Architectures, Methods and Tools (DSD 2005), IEEE Computer Society, pp. 306 - 313, Porto, Portugal, August 2005.

[4] GPGPU.org, "GENERAL-PURPOSE COMPUTATION USING GRAPHICS HARDWARE", <http://www.gpgpu.org/>

[5] N. Goodnight, R. Wang, and G. Humphreys, "COMPUTATION ON PROGRAMMABLE GRAPHICS HARDWARE", IEEE Computer Graphics and Applications, 25(5): pp. 12 – 15, 2005

[6] A. L. Shimpi, and D. Wilson, "NVIDIA'S 1.4 BILLION TRANSISTOR GPU", <http://www.anandtech.com/video/showdoc.aspx?i=3334&p=21>, Web article, Jun. 2008

[7] AMD, "ATI RADEON™ HD 4800 SERIES", <http://ati.amd.com/products/radeonhd4800/specs2.html>

[8] P. Hanrahan, and J. Lawson, "A LANGUAGE FOR SHADING AND LIGHTING CALCULATIONS", Proceedings of the 17th conference on Computer graphics and interactive techniques, Dallas, USA, pp. 289 – 298, 1990

[9] K. Moreland, and E. Angel, "THE FFT ON A GPU", Proceedings of the ACM SIGGRAPH/Eurographics Workshop on Graphics Hardware, San Diego, USA, pp. 112 – 119, 2003

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

[10] D. Blythe, "THE DIRECT3D 10 SYSTEM", ACM Transactions on Graphics, 25(3): pp. 724 – 734, Jul. 2006

[11] W. R. Mark, R. S. Glanville, K. Akeley, and M. J. Kilgard, "CG: A SYSTEM FOR PROGRAMMING GRAPHICS HARDWARE IN A C-LIKE LANGUAGE", ACM Transactions on Graphics, 22(3): pp. 896 – 907, 2003

[12] J. Kessenich, D. Baldwin, and R. Rost, "THE OPENGL SHADING LANGUAGE", <http://www.opengl.org/registry/doc/GLSLangSpec.Full.1.20.8.pdf>

[13] M. D. McCool, Z. Qin, and T. S. Popa, "SHADER METAPROGRAMMING", Proceedings of the ACM SIGGRAPH/Eurographics Workshop on Graphics Hardware, Saarbruecken, Germany, pp. 57 - 68, 2002

[14] T. R. Halfhill, "PARALLEL PROCESSING WITH CUDA", Microprocessor Report, <http://www.mdronline.com>, Jan. 2008

[15] AMD, "ATI CTM GUIDE", <http://ati.amd.com/companyinfo/researcher/documents/ATI_CTM_Guide.pdf>

[16] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan, "BROOK FOR GPUS: STREAM COMPUTING ON GRAPHICS HARDWARE", ACM Transactions on Graphics, 23(3): pp. 777–786, 2004.

[17] AMD, "AMD STREAM SDK", <http://ati.amd.com/technology/streamcomputing/index.html>

[18] T. Krazit, "INDUSTRY GROUP TO EVALUATE APPLE'S OPENCL AS STANDARD", <http://news.cnet.com/8301-13579_3-9970617-37.html>, Jun. 2008

[19] NVIDIA, "COMPUTE UNIFIED DEVICE ARCHITECTURE PROGRAMMING GUIDE VER 1.1", <http://developer.download.nvidia.com/compute/cuda/1_1/NVIDIA_CUDA_Programming_Guide_1.1.pdf>, Nov. 2007

[20] Rys, "NVIDIA G80: ARCHITECTURE AND GPU ANALYSIS", <http://www.beyond3d.com/content/reviews/1/5>, Nov. 2006

[21] M. J. Atallah, "ALGORITHM AND THEORY OF COMPUTATION HANDBOOK" Boca Raton, FL: CRC Press, 1998.

[22] S. Ryoo, "PROGRAM OPTIMIZATION STRATEGIES FOR DATA-PARALLEL MANY-CORE PROCESSORS", PhD Dissertation, University of Illinois, Urbana, IL, 2008.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

[23] L. Marziale, G. G. Richard III, and V. Roussev, "MASSIVE THREADING: USING GPUS TO INCREASE THE PERFORMANCE OF DIGITAL FORENSICS TOOLS", Proceedings of the 7th Annual Digital Forensics Research Workshop (DFRWS 2007), Boston, MA, 2007

[24] S. Needleman, and C. Wunsch. "A GENERAL METHOD APPLICABLE TO THE SEARCH FOR SIMILARITIES IN THE AMINO ACID SEQUENCES OF TWO PROTEINS", Journal of Molecular Biology, 48: pp. 443 - 453, 1970.

[25] T. F. Smith, and M. S. Waterman, "IDENTIFICATION OF COMMON MOLECULAR SUBSEQUENCES", Journal of Molecular Biology, 147: pp. 195 - 197, 1981

[26] Wikipedia, "DYNAMIC PROGRAMMING", <http://en.wikipedia.org/wiki/Dynamic_Programming>

[27] W. Liu, B. Schmidt, G. Voss, and W. Muller-Wittig, "STREAMING ALGORITHMS FOR BIOLOGICAL SEQUENCE ALIGNMENT ON GPUS", IEEE Transactions on Parallel and Distributed Systems, 18(9): pp. 1270 - 1281, Jun. 2007

[28] M. C. Schatz, C. Trapnell, A.L. Delcher, and A. Varshney, "HIGH-THROUGHPUT SEQUENCE ALIGNMENT USING GRPHICS PROCESSING UNITS", BMC Bioinformatics, 8: pp. 474., 2007

[29] S. A. Manavski, and G. Valle, "CUDA COMPATIBLE GPU CARDS AS EFFICIENT HARDWARE ACCELERATORS FOR SMITH-WATERMAN SEQUENCE ALIGNMENT ", BMC Bioinformatics, 9(2): pp. S10, Mar. 2008

[30] T. Rognes, and E. Seeberg, "SIX FOLD SPEED-UP OF SMITH-WATERMAN SEQUENCE DATABASE SEARCHES USING PARALLEL PROCESSING ON COMMON MICROPROCESSORS", Bioinformatics. 2000 Aug;16(8):699-706

[31] M. Snir, "WAVEFRONT PATTERN", <http://www.cs.uiuc.edu/homes/snir/PPP/patterns/wavefront.pdf>

[32] NCBI, "FASTA FORMAT DESCRIPTION", <http://www.ncbi.nlm.nih.gov/blast/fasta.shtml>

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

CURRICULUM VITAE

Sungbo Jung

PERSONAL INFORMATION

Date of Birth: November 24th, 1976
Place of Birth: Bucheon, Republic of Korea
E-mail: s0jung03@louisville.edu

EDUCATION & TRAINING

Aug. 2005 - Present	M.S., Computer Engineering and Computer Science University of Louisville, KY, U.S.A
Sep. 2001 - Aug. 2003	M.E., Industrial Engineering Korea University, Korea
Mar. 1996 - Feb. 2000	B.A., Mass Communication Korea University, Korea

AWARDS

Nov. 2008	Computer and Information Science - 1st Prize, Graduate Research Competition Award, Kentucky Academy of Science at Univ. of Kentucky
Oct. 2002	Bronze Medal, Technological Automobile Competition, Hyundai-Kia Motors
Aug. 2002	Research Prize, Korea Intelligent Robot Competition at the Postech.

PROFESSIONAL SOCIETY MEMBERSHIP

ACM Student Member

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

PROFESSIONAL EXPERIENCE

Sep. 2002 - Aug. 2003	WEB administrator, College of Engineering, Korea University, Seoul, Korea
Sep. 2001- Aug. 2002	Instructor and Teaching Assistant, Department of Industrial Engineering, Korea University, Seoul, Korea
Jan. 2000 - Apr. 2001	Reporter, MacMadang – Macintosh Magazine. Seoul, Korea

PUBLICATIONS

Journal:

S. Jung, K. Lee, and D. Jang, "A Study on Stand-alone Autonomous Mobile Robot using Mono Camera", The Korea Institute of Signal Processing and Systems, Korea, Feb 2003

UNIVERSITY SERVICE

President of Korean Student Association at University of Louisville, 2006

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.