Comparative Analysis of Protein Alignment algorithms in Parallel environment using CUDA

BLAST verses Smith-Waterman

Supervisor: Dr. JiaUddin

Shadman Fahim-12301011

GulshanJubaed Prince-12301014

Shehabul Hossain-12301005

BRAC University

Submitted on: 18thApril 2016

Comparative Analysis of Protein Alignment algorithms in Parallel environment using CUDA

A THESIS

Submitted to the School of Computer Science and Engineering BRAC University Bangladesh.

In partial fulfillment of the requirements for the Bachelor's degree in Computer Science and Engineering

Signature of Supervisor

Dr. JiaUddin

Signatures of Author

ShadmanFahim

GulshanJubaed Prince

ShehabulHossain

DEDICATION

Wewouldliketodedicate (atthehighestlevel) thisthesistoAllah, theOmnipotent, theOmniscient, themostExalted, themostBeneficent, themostMerciful.

We would also like to mention the name of Prophet Muhammad (peace and blessings be upon him), who is our inspiration in every rightly guided things we do and every rightly guided choice we make as a role model of our entire life.

While writing this section, we can’t forget our parents who are the most precious family members to us. Furthermore, we keep in mind the rightly guided servants of Allah the Exalted, who assist us to be guided to the Path journeyed by; they are the reason we are here at all and made us who we are today.

ACKNOWLEDGEMENT

This thesis is an expedition that could not be completed without the wish and blessings of the almighty Allah, the most Exalted. We seek repentance from Allah for every mistake, we made, of course, during the period of writing this thesis.

Our journey towards the BSc. degree would not have been possible without the help of many people. It is our great pleasure to take this opportunity to thank them for the support and advice that we received over the past years.

We would like to express our sincere gratitude to our advisor, Dr. JiaUddin, for his support, encouragement, and guidance throughout this research. He has directed this research with competence, instilling his enthusiasm, providing support in uncountable occasions. This work would not have been completed without his help, support, and practically infinite supply of comments and ideas. It has been a great honor to work with him during our stay at the BRAC University.

Furthermore, we keep in mind the rightly guided servants of Allah the Exalted, who assist us to be guided to the Path journeyed by those whom Allah the Exalted showered blessings; they are the reason we are here at all and made us who we are today.

Finally, we would like to share a great deal of my achievement with our parents, our family, and our friends, who always support and encourage us in our life.

Contents

DEDICATION 3

ACKNOWLEDGEMENT 4

ABSTRACT 9

CHAPTER 1 10

Introduction 10

1.1 Overview 10

1.2 Objective and Goals 11

1.3 Motivation 12

1.4 Thesis Outline 12

CHAPTER 2 13

Background Study 13

2.1 Nvidia Gpu 13

2.2 CUDA 16

2.3 Blast Algorithm 19

2.4 Smith-Waterman Algorithm 23

CHAPTER 3 27

EXPERIMENTAL SETUP 27

3.1 CUDA Toolkit 27

3.2 GeForce GTX 660 GPU card 28

3.3 BLAST CUDA 30

3.4 Smith-Waterman CUDA 33

CHAPTER 4 35

EXPERIMENTAL RESULT ANALYSIS 35

CHAPTER 5 42

Conclusion and Future Work 42

REFERENCE 44

LIST OF TABLES

Figure 1. A general GPU architecture.. 14

Figure 2. Parallel computation between the host and the device 17

Figure 3. A CUDA code sample. 18

Figure 4. The scoring matrix; Blosum 62. 20

Figure 5. Un-gapped extension. 21

Figure 6. An overview of BLAST. 21

Figure 7. Another BLASTP example.. 22

Figure 8. The experimental setup of BLAST algorithm. 31

Figure 9. Kernel call. 32

Figure 10. The experimental setup of Smith-Waterman algorithm. 34

Figure 11. Comparison of execution time varying block and thread sizes. 35

Figure 12. Comparison of runtimes achieved by both the algorithm using Table 9..................... 37

Figure 13. Comparison of runtimes achieved by both the algorithm using Table 10.................. 38

Figure 14. Results comparison of BLAST and Smith-Waterman. 39

Figure 15. Comparison of runtimes achieved by CPU and GPU using Table 11........................ 41

LIST OF TABLES

TABLE 1. GPU ENGINE SPECIFICATIONS 28

TABLE 2. MEMORY SPECIFICATIONS 28

TABLE 3. FEATURE SUPPORT 29

TABLE 4. DISPLAY SUPPORT 29

TABLE 5. STANDARD GRPAHCIS CARD DIMENSIONS 30

TABLE 6. THERMAL AND POWER SPECIFICATIONS 30

TABLE 7. THERMAL AND POWER SPECIFICATIONS 30

TABLE 8. EXECUTION TIME OF UBP6P PROTEIN IN BLASTP. THREAD SIZE IS VARIED WITH BLOCK SIZE. 36

TABLE 9. RUNTIME IN SECONDS FOR BOTH THE ALGORITHMS USING 256 GPU BLOCKS AND 128 THREADS. 37

TABLE 10. RUNTIME IN SECONDS FOR BOTH THE ALGORITHMS USING 256 GPU BLOCKS AND 256 THREADS. 38

TABLE 11. RUNTIME OF BLASTP IN SECONDS FOR CPU AND GPU. 40

ABSTRACT

In bioinformatics to identify evolutionary relationships two sequences are matched to find similarity. Smith Waterman, a dynamic algorithm, is a common choice to carry out this alignment process. However, with the exponential growth of protein databases this algorithm's time complexity increases. The demand of bioinformatics for their tasks to speed up is very high. Even a slight seed up in computation would be very helpful in the field of bioinformatics. Thus, for a lot of the scientists this algorithm might not be the first choice. In today's world the most popular and used bioinformatics tool is the BLAST (Basic Local Alignment Tool). BLAST, similar to Smith Waterman algorithm, is an alignment algorithm for scanning proteins from protein databases. This thesis analyzes both the algorithms in a parallel environment with the help of NVIDIA GPU. For our experiments we utilized a GeForce GTX 660 NVIDIA GPU to execute both the algorithms. Experimental results show that BLAST is on average is 2.5 times faster than Smith-Waterman.

CHAPTER 1

INTRODUCTION

1.1 OVERVIEW

Basic Local Alignment Search Tool (BLAST) is the most popular alignment algorithm in the world of science today. The algorithm uses dynamic programing which utilizes well defined mutation scores. This method is more than an order of magnitude faster than the existing heuristic algorithms [1]. It has been cited over 25,000 [2] and over 21,000 [3]. For such popularity US National Center for Biotechnology Information (NCBI) plays the most important role. The reason for this is because NCBI provided a platform over the internet for everyone's reach. Now any general person can go on the website of NCBI and get results for their queries. It is noted that hundreds and thousands of queries are being processed every now and then using the platform provided by NCBI. This results in the increase of BLAST's usage by 2 to 3 times [4].

On the contrary, Smith Waterman is also a dynamic algorithm but isn't used as much as BLAST. This algorithm generates more accurate results than what BLAST produces. However, its accuracy is maintained at the expense of computation time and computer power [5]. Computation speed is the burning topic in today's world. How fast a task can be processed is the main challenge. One of such other challenges is searching through long detailed databases. With the exponential growth of protein databases demand to accelerate searching through such huge databases is very high. NCBI is having tremendous breakthroughs in this particular field. However, NCBI uses sequential search for the queries. With the availability of Graphics Processing Units (GPUs) it can be assumed that using its parallel techniques BLAST algorithm can have a faster processing time. An implementation of BLASTP algorithm is handled by GPU using Compute Unified Device Architecture (CUDA), CUDA-BLASTP. It is claimed that in CUDA architecture they have managed to achieve speedups of 10 times compared to sequential NCBI BLAST 2.2.22 on a GeForce GTX 295. It is also 3-4 times faster than multithreaded NCBI BLAST on an Intel Quad-Core processor [4]. Similarly, mpiBLAST is an open-sourcesequence tool that parallelizes the NCBI BLAST toolkit. It uses the database segmentation

approach and the master-worker style. It achieves significant speedups in small or moderate number of processes [6]. On the other hand researchers started to implement Smith-Waterman in GPUs as well. CUDASW++ 2.0 has managed to achieve an average performance of 9.509 GCUPS on single-GPU version and an average performance of 14.484 GCUPS on dual-GPU version [5]. A lotof efforts have been made to parallelize this computationon high-performance computing architectures rangingfrom loosely-coupled to tightly-coupled ones. Architecture examples include clouds [7], clusters [7] and accelerators [5]. Forfield programmable gate arrays (FPGAs), some approaches based on linear systolic arrays and custom instructions have been proposed. Oliver et al. [8] constructed a linear systolic array on a standard Virtex II FPGA board to perform the SW algorithm with affine gap penalties. Li et al. [9] designed custom instructions to support massively parallel computing of the SW algorithm on an Altera Stratix EP1S40 FPGA.

1.2 OBJECTIVE AND GOALS

There are scopes to speed-up the process and meet the demand of accelerating it. This thesis demonstrates both the alignment algorithms. Also, it illustrates how this task is handled by a GPU using CUDA. CUDA by NVIDIA, a parallel computing architecture uses parallel compute engine in NVIDIA GPUs to solve many computationally intensive problems in a more efficient way than on Central Processing Unit (CPU) [10]. Using its parallel techniques we demonstrate how computation can speed-up. The database used in this research is taken from the NCBI website [11]. Smith-Waterman algorithm is overshadowed by the widespread use of BLAST algorithm. Thus, this thesis focuses on the algorithms and their implementation on GPU. Later few factors are compared to see if the domination of BLAST is justified or not. Also, the purpose is to bring back Smith-Waterman if its performance is not too exhaustive.

1.3 MOTIVATION

In today's world it is all about speed. How fast you can do something is the challenge! For example you want to watch a movie that takes 1 hour to download. If that movie is downloaded within 30 minutes then you feel a sense of happiness. Similarly, for bioinformatics if they can perform their task with accurate results in a short time then they will be ecstatic. Thus, being computer engineers we want to be part of this evolutionary change in speed. We wanted to showcase that we can take a real life task and make it happen faster. Also, parallel computing seems very astonishing. Being able to lay our hand on parallel computing to implement an algorithm is what motivated us the most. Looking at all the work being done using GPU is just mind boggling. Three of the top 5 super computer is powered by GPU.

1.4 THESIS OUTLINE

Orientation of this thesis is as below:

- ✓ Chapter 2 displays the architecture of a NVIDIA GPU, features of CUDA, Smith-Waterman algorithm and BLAST algorithm.
- ✓ Chapter 3 provides a detailed explanation of the implementation of the algorithms.
- ✓ The results of the experiments carried out and their analysis are included in chapter 4.

Finally chapter 5 concludes and discusses about our future work.

CHAPTER 2

BACKGROUND STUDY

A simple way to understand the difference between a CPU and GPU is to compare how they process tasks. A CPU consists of a few cores optimized for sequential serial processing while a GPU has a massively parallel architecture consisting of thousands of smaller, more efficient cores designed for handling multiple tasks simultaneously.

At the start of multicore CPUs and GPUs the processor chips have become parallel systems. But speed of the program will be increased if software exploits parallelism provided by the underlying multiprocessor architecture. Hence there is a big need to design and develop the software so that it uses multithreading, each thread running concurrently on a processor, potentially increasing the speed of the program dramatically. To develop such a scalable parallel applications, a parallel programming model is required that supports parallel multicore programming environment.

NVIDIA is focusing on professional visualization, Data centers, gaming and Auto are the four markets where NVIDIA from 2014. The researchers and scientists are given the capability of parallel processing by NVIDIA. Thus the applications which need high performance can be run efficiently using NVIDIA GPUs. This NVIDIA GPU power millions of devices like desktops, notebooks, workstations and supercomputer all over the world.

2.1 NVIDIA GPU

NVIDIA Corporation began as an American technology company based in Santa Clara, California. NVIDIA designs graphics processing units (GPUs) for the gaming market, as well as system on chip units (SOCs) for the mobile computing and automotive market. NVIDIA's primary GPU product line, labeled "GeForce", is in direct competition with Advanced Micro Devices' (AMD) "Radeon" products. NVIDIA expanded its presence in the gaming industry with its handheld SHIELD Portable, SHIELD Tablet, and SHIELD Android TV. In addition to GPU

manufacturing, NVIDIA provides parallel processing capabilities to researchers and scientists that allow them to efficiently run high-performance applications. They are deployed in supercomputing sites around the world [12].

When we consider a processor that is more power efficient and have a better performance, GPU comes in top of that list. Comparing to a CPU, a GPU provides a better performance because it offers a higher peak GFLOPS (Giga floating-point operations per second) [13]. The GPU that we used for the experimentations is GeForce GTX 660. Generally a GPU device has several multiprocessors with several processors inside each of them. The Figure 1 enlightens it. There are mainly two types of memory in GPU. One is on-chip memory and the other is off-chip memory. The on-chip memory has low access latency but a relatively small size. On the other hand the off-chip memory has larger size and also higher access latency [14]. Moreover, these microprocessors contain the shared memory and caches, along with registers.

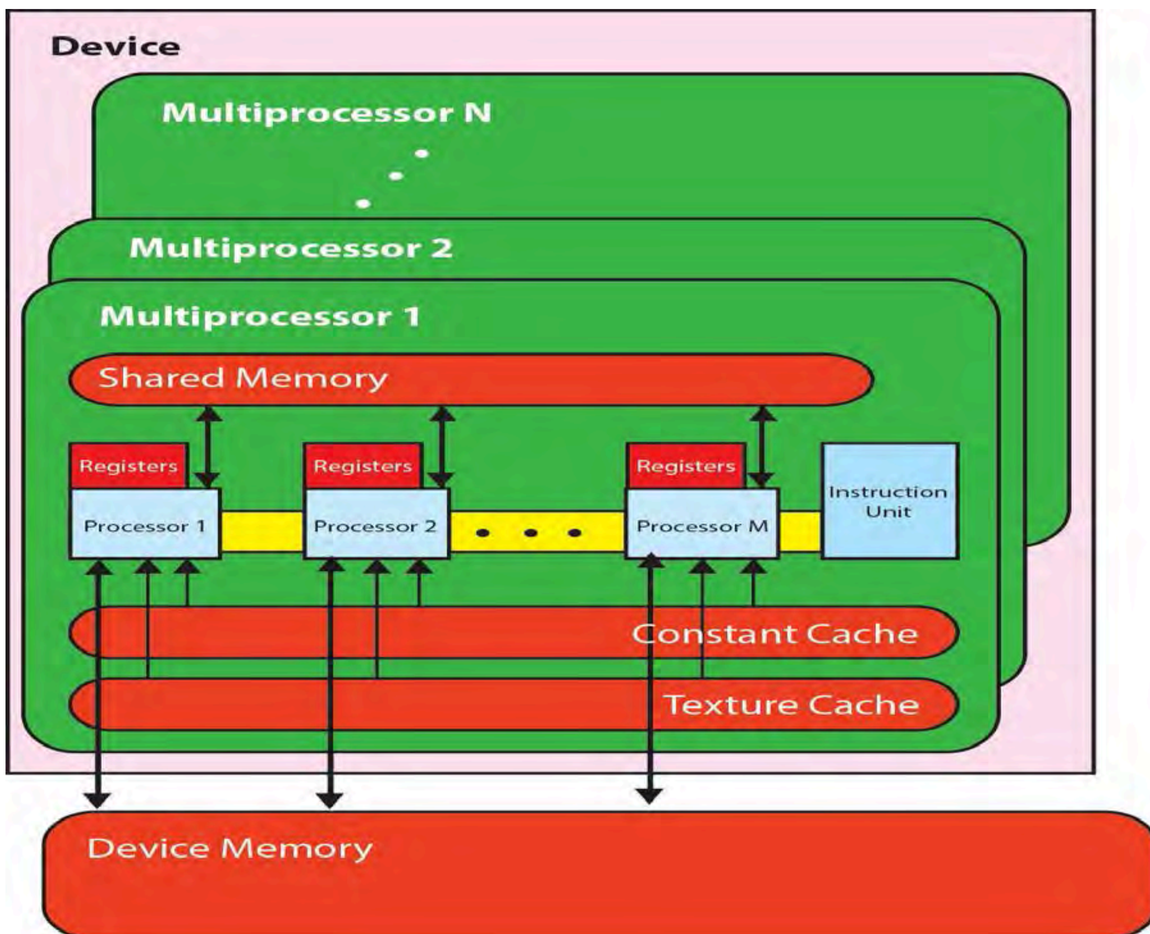

Figure 1.A general GPU architecture.

The following fields explains about the GPU,

- Model - The marketing name for the processor assigned by NVIDIA.
- Launch - Date of release for the processor.
- Code name - The internal engineering codename for the processor (typically designated by an NVXY name and later GXY where X is the series number and Y is the schedule of the project for that generation).
- Fab - Fabrication process. Average feature size of components of the processor.
- Bus interface - Bus by which the graphics processor is attached to the system (typically an expansion slot, such as PCI, AGP, or PCI-Express).
- Memory - The amount of graphics memory available to the processor.
- SM Count - Number of streaming multiprocessors.
- Core clock - The factory core clock frequency (while some manufacturers adjust clocks lower and higher, this number will always be the reference clocks used by NVIDIA).
- Memory clock - The factory effective memory clock frequency (while some manufacturers adjust clocks lower and higher, this number will always be the reference clocks used by NVIDIA). All DDR/GDDR memories operate at half this frequency, except for GDDR5, which operates at one quarter of this frequency.
- Core configuration - The layout of the graphics pipeline, in terms of functional units. Over time the number, type, and variety of functional units in the GPU core has changed significantly; before each section in the list there is an explanation as to what functional units are present in each generation of processors. In later models, shaders are integrated into a unified shader architecture, where any one shader can perform any of the functions listed.
- Fill rate - Maximum theoretical fillrate in textured pixels per second. This number is generally used as a "maximum throughput number" for the GPU and generally, a higher fillrate corresponds to a more powerful (and faster) GPU.
- Memory subsection
  - Bandwidth - Maximum theoretical bandwidth for the processor at factory clock with factory bus width. GB=10^9 bytes.
  - Bus type - Type of memory bus or buses utilized.

  - Bus width - Maximum bit width of the memory bus or buses utilized. This will always be a factory bus width.
- API support section
  - Direct3D - Maximum version of Direct3D fully supported.
  - OpenGL - Maximum version of OpenGL fully supported.
- Features - Additional features that are not standard as a part of the two graphics libraries.

2.2 CUDA

CUDA introduced by NVIDIA is a general purpose parallel computing platform and programming model [15]. It enables dramatic increases in computing performance by harnessing the power of the graphics processing unit (GPU). The CUDA platform is a software layer that gives direct access to the GPU's virtual instruction set and parallel computational elements, for the execution of compute kernels. The CUDA platform is designed to work with programming languages such as C, C++ and Fortran. This accessibility makes it easier for specialists in parallel programming to utilize GPU resources, as opposed to previous API solutions like Direct3D and OpenGL, which required advanced skills in graphics programming. Also, CUDA supports programming frameworks such as OpenACC and OpenCL The figure 2 below shows how host and device communicate with each other.

One of the main advantages of CUDA is shared memory. CUDA exposes a fast shared memory region that can be shared amongst threads. This can be used as a user-managed cache, enabling higher bandwidth than is possible using texture lookups [16]. CUDA C extends C by allowing the programmer to define C functions, called kernels. Unlike C functions that run only once this kernel runs N times in parallel by N different CUDA threads. Each thread that executes the kernel is given a unique thread ID.The threads are organized in a hierarchy consisting of blocks and grids. When calling a kernel function the size of the blocks and the number of threads per block are specified. An example of the function to call kernels is presented as:

kernel<<<numBlocks, numThreads>>>(parameter’s list).

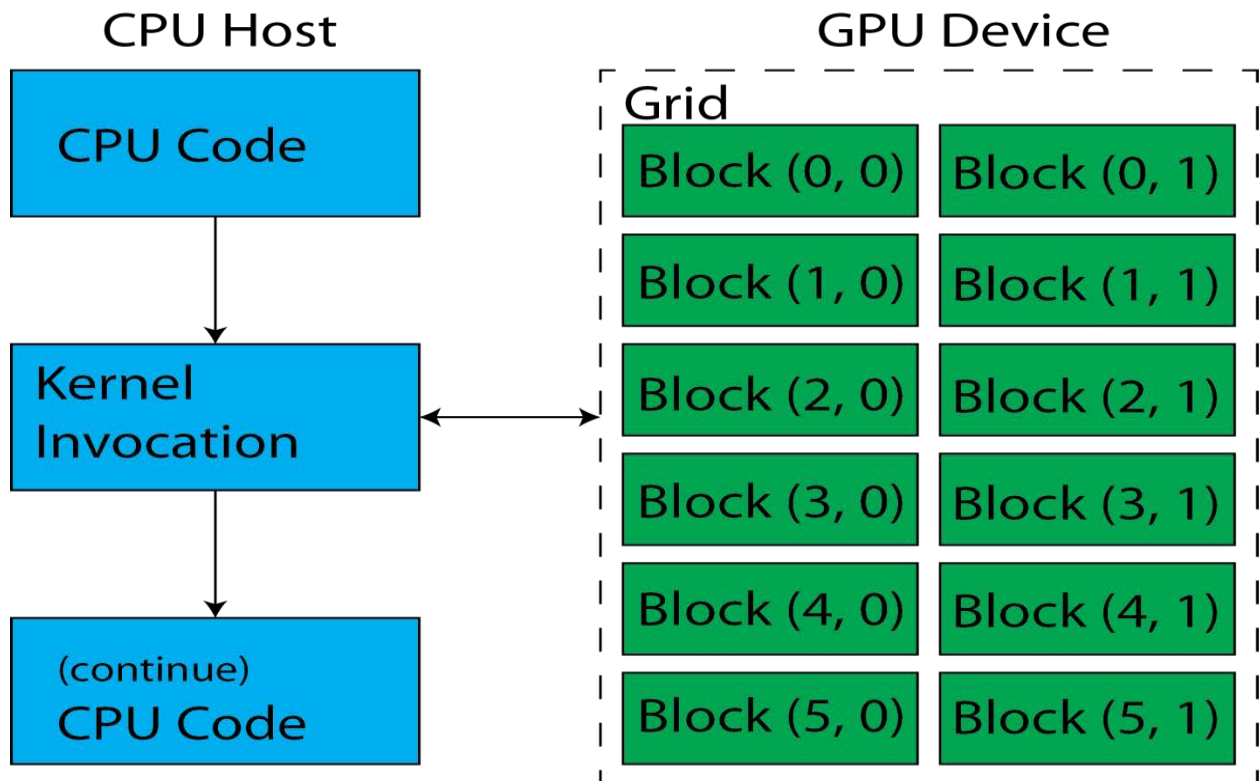

Figure 2. Parallel computation between the host and the device

With millions of CUDA-enabled GPUs sold to date, software developers, scientists and researchers are finding broad-ranging uses for GPU computing with CUDA. Here are a few examples:

1. **Identify hidden plaque in arteries**

Heart attacks are the leading cause of death worldwide. Harvard Engineering, Harvard Medical School and Brigham & Women's Hospital have teamed up to use GPUs to simulate blood flow and identify hidden arterial plaque without invasive imaging techniques or exploratory surgery.

2. **Analyze air traffic flow**

The National Airspace System manages the nationwide coordination of air traffic flow. Computer models help identify new ways to alleviate congestion and keep airplane traffic

moving efficiently. Using the computational power of GPUs, a team at NASA obtained a large performance gain, reducing analysis time from ten minutes to three seconds.

3. Visualize molecules

A molecular simulation called NAMD (nanoscale molecular dynamics) gets a large performance boost with GPUs. The speed-up is a result of the parallel architecture of GPUs, which enables NAMD developers to port compute-intensive portions of the application to the GPU using the CUDA Toolkit.

Figure 3 compares a standard C code with CUDA code.

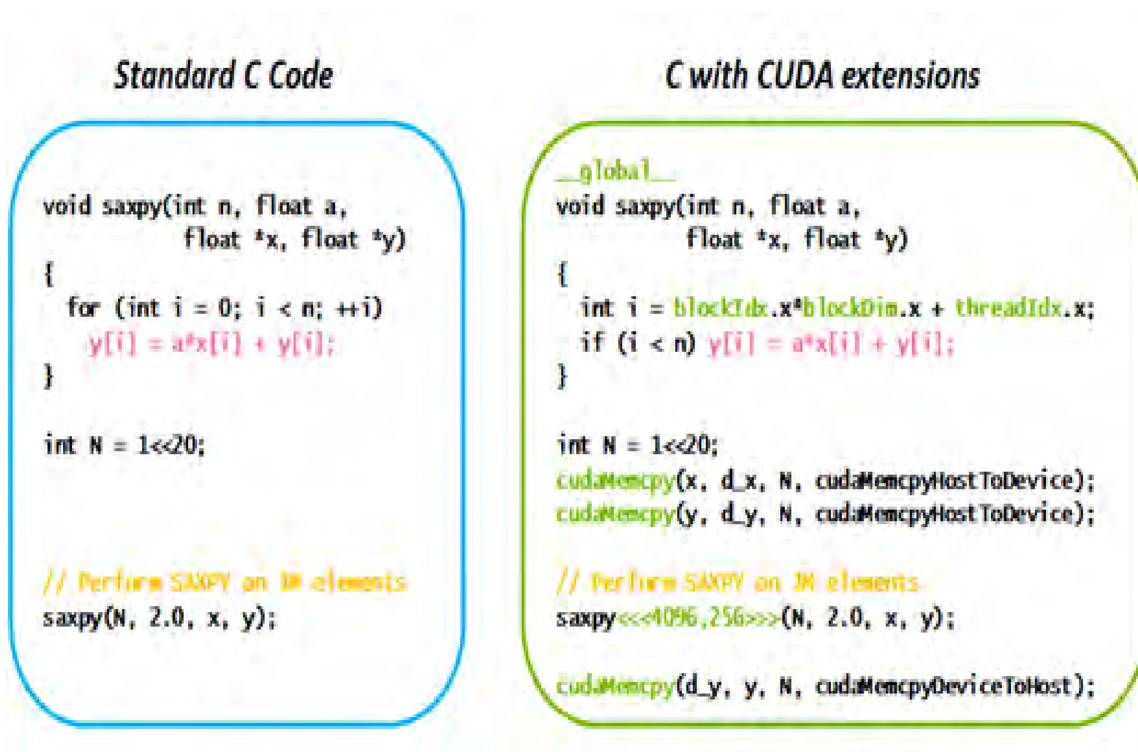

Figure 3.A CUDA code sample.

2.3 BLAST ALGORITHM

Before BLAST, FASTA was developed by David J. Lipman and William R. Pearson in 1985 [17]. Besides fast algorithms like BLAST and FASTA, Smith-Waterman algorithm was used to search protein databases which guarantee the optimal alignments of the query and database sequences unlike BLAST and FASTA. However, the heuristic approach of BLAST algorithm is overall a lot faster. So, due to such highly populated protein databases Smith-Waterman search is both time consuming and computer power intensive. The actual alignment part of the algorithm which is performed for every database sequence has a very different complexity. To find the seeds, each of the words in the database sequence must be compared to hash table created for the neighbors of the query sequence words, and thus we must perform M lookups. The end product of the M lookups is on the order of N seeds total, because there are only $N–w+1$ words in the query. Each of these seeds starts an alignment, and the maximum length of the alignment is the length of the query sequence, M, assuming $M<N$. Since calculating λ must only be done once, and calculating the statistical significance of each HSP is a constant time operation, these have a complexity of O(1).The time complexity of this particular algorithm is [18]:

$$O(M) + O(MN) + O(1) = O(MN).$$

We must first take into consideration the hash table. The table contains 20^wrows, one for every possible word of length w. The rows contain the locations for each of the words, and the total number of positions is on the order of N. Thus, there should on the order of N seeds which can each lead to a local alignment of a maximum of length M. The total space complexity is [18],

$$O(20^w)+O(N)+O(MN)=O(20^w+MN)$$

Thus, the space complexity is slightly higher than the other algorithms, however the actual space used may not be significantly larger than the dynamic programming algorithms. This is because many of the local alignments will be discarded because they do not meet the threshold, and also because the alignments which do meet the threshold will significantly shorter than length M.

So as mentioned earlier BLAST is the most popular heuristic search algorithm for protein scanning. Unlike Smith-Waterman algorithm where the entire sequence is compared BLAST locate high scoring short matches between the query sequence and the subject sequence [14].

Due to this the accuracy of BLAST decreases to some extent but then the processing speed increases exceptionally than Smith-Waterman. The Blast Algorithm mainly has four stages [19], such as in the first stage the query sequence is matched with the subject sequence in order to find matches. For this the query sequence is broken down to similar size world lengths (W). For our experiment W is 3. For example a query sequence is ARNDCQEGHFPYVWTSKMLI. This sequence is extracted to three letters word; i.e. ARN, RND, NDC, DCQ, CQE and so on. Later these extracted words are matched with the subject query one by one to identify matches known as hits. This process is called the hit detection. The hits are then scored using the blosum62 scoring matrix. A sample snap shot of blossom62 is presented in Figure 4. The hits are not necessary to be exact matches similar matches is also accepted as long as the score of that hit is greater than a certain threshold (T). Otherwise the hits that do not overcome the threshold are filtered out. For example, we obtain a score of 15 and 12 by comparing PQG with PEG and PQA, respectively. Let's assume that T is 13. As a result PEG is kept and PQA is cut off.

	A	R	N	D	C	Q	E	G	H	I	L	K	M	F	P	S	T	W	Y	V
A	4																			
R	-1	5																		
N	-2	0	6																	
D	-2	-2	1	6																
C	0	-3	-3	-3	9															
Q	-1	1	0	0	-3	5														
E	-1	0	0	2	-4	2	5													
G	0	-2	0	-1	-3	-2	-2	6												
H	-2	0	1	-1	-3	0	0	-2	8											
I	-1	-3	-3	-3	-1	-3	-3	-4	-3	4										
L	-1	-2	-4	-4	-1	-2	-3	-4	-3	2	4									
K	-1	2	-1	-1	-3	1	1	-2	-1	-3	2	5								
M	-1	-1	-3	-3	-1	0	-2	-3	-2	1	2	-1	5							
F	-2	-3	-3	-3	-2	-3	-3	-3	-1	0	0	-3	0	6						
P	-1	-2	-1	-1	-3	-1	-1	-2	-2	-3	-3	-1	-2	-4	7					
S	1	-1	0	0	-1	0	0	0	-1	-2	-2	0	-1	-2	-1	4				
T	0	-1	-1	-1	-1	-1	-1	-2	-2	-1	-1	-1	-1	-2	-1	1	5			
W	-3	-3	-4	-4	-2	-2	-3	-2	-2	-3	-2	-3	-1	1	-4	-3	-2	11		
Y	-2	-2	-3	-3	-2	-1	-2	-3	2	-1	-1	-2	-1	3	-3	-2	-2	2	7	
V	0	-3	-3	-3	-1	-2	-2	-3	-3	3	1	-2	1	-1	-2	-2	0	-3	-1	4

Figure 4.The scoring matrix; Blosum 62.

In stage 2, the remaining hits are now sent to this stage for further processing. The hits are extended in both directions and as long as the accumulated score is increasing the extension is carried on. As soon as the score starts to decrease we stop. This is called un-gapped extension. The result is HSPs (highest scoring pairs). A sample of un-gapped extension is presented in Figure 5.

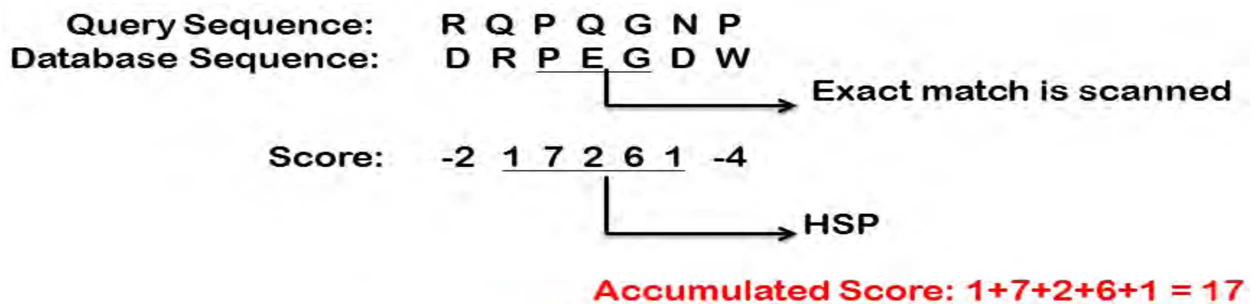

Figure 5.Un-gapped extension.

Consequently in stage 3, the HSPs sent to this stage are then extended further. However, unlike the last stage here gaps are allowed. With each gap there is a penalty. This is called the gapped extension. Finally in stage 4, scores all the alignments again from the previous stage. Once done scoring it produces the top scores. This is called the gapped alignment with trace-back. The entire BLASTP algorithm is enlightened in Figure 6.

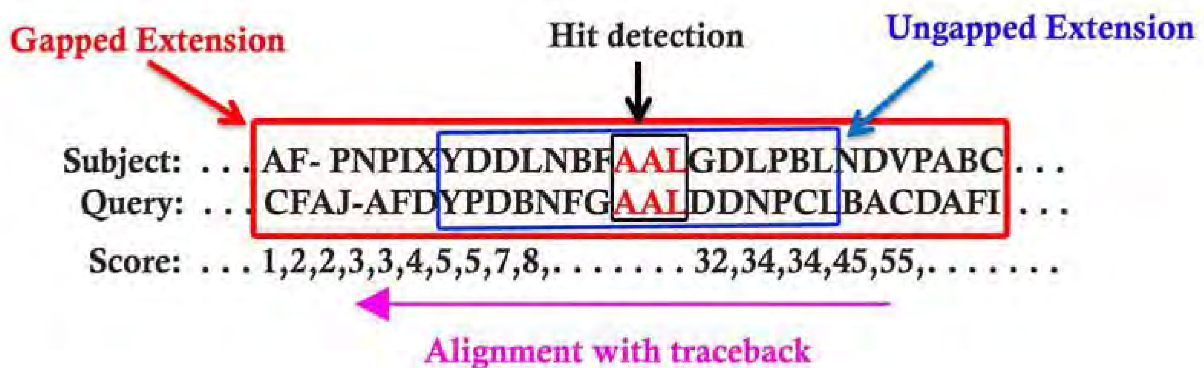

Figure 6.An overview of BLAST.

In Figure 6, the query sequence and the subject sequence is compared. We can see there is an exact match in the sequence AAL. This is the stage one where the hit is detected. In the next stage the un-gapped extension is performed on pairs of high-scoring segment pairs (HSPs). Here, the initial match in extended in both directions until there the overall accumulated score starts to decrease. In stage 3, the un-gapped extension is extended using a gapped alignment [20]. To determine the level of the alignment a scoring matrix and a threshold value is used. Finally, in the last stage a trace-back algorithm is used to produce and score the alignments.

The figure 7 below provides more details of the process [21]. In stage 1, short, matches areidentified(black lines in the left figure). In stage 2, matches along the same diagonal are extended (non-gapped) ifthe resulting score exceeds a specified threshold. The extensions are shown as grey lines in the leftfigure. Next, stage 3 extends (typically using Smith-Waterman) the non-gapped sequences using gappedalignment, as shown by the grey line in the right figure. Finally, stage 4 generates and scores thesequence for the end user using alignment traceback algorithms.

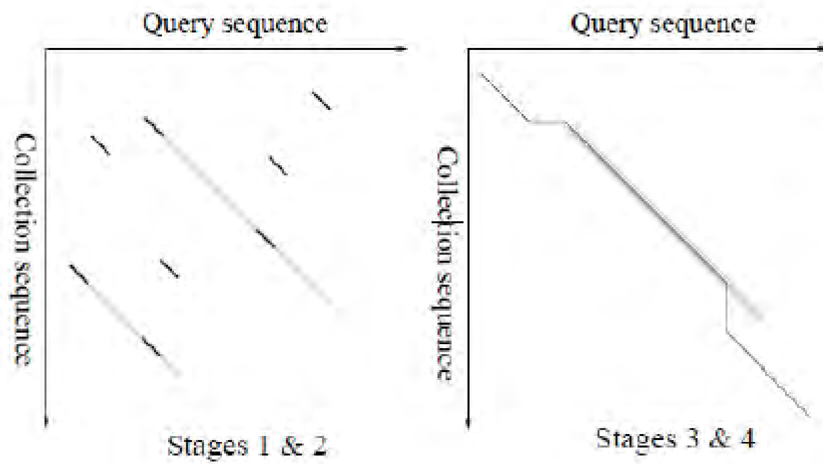

Figure 7.Another BLASTP example.

2.4 SMITH-WATERMAN ALGORITHM

In 1981 before BLAST or FASTA were written Smith and Waterman suggested Smith-Waterman algorithm [22]. Later in 1982 Gotoh improved the algorithm [24]. This is a local alignment algorithm. Thus it matches the highest similarities between two proteins instead of the aligning the entire two proteins. Instead of aligning the entire length of two protein sequences, this algorithm finds the region of highest similarity between two proteins. This is potentially more biologically relevant due to the fact that the ends of proteins tend to be less highly conserved than the middle portions, leading to higher mutation, deletion, and insertion rates at the ends of the protein. The Smith-Waterman algorithm allows us to align proteins more accurately without having to align the ends of related protein which may be highly different. Assuming two query sequences S_1 and S_2 having lengths l_1 and l_2. The two sequences are arranged in a matrix form with $l_1 + 1$ row and $l_2 + 1$ column. Initially the first row and column are set to 0. Then the similarity matrix is computed for $1 <= i <= l_1$, $1 <= j <= l_2$ using the formula as shown below. At last the trace back is performed to calculate the final overall score.There are mainly three steps to run this algorithm, they are:

1. Initialization.

H (0, j) = 0;

H (i, 0) = 0; //where H is the similarity score matrix

2. Filling the matrix, H.

$$H(i,j) = \max \begin{Bmatrix} 0 & \\ H(i-1,j-1) + s(a_i,b_j) & \text{Match/Mismatch} \\ \max_{k\geq 1}\{H(i-k,j) + W_k\} & \text{Deletion} \\ \max_{l\geq 1}\{H(i,j-l) + W_l\} & \text{Insertion} \end{Bmatrix}, 1 \leq i \leq m, 1 \leq j \leq n$$

Where:

- a, b = Strings over the alphabet
- m = length (a)
- n = length (b)
- s(a, b) is a similarity function on the alphabet
- H (i, j) – is the maximum similarity score between a suffix of [1..i] and a suffix of b [1..j]
- W_i is the gap-scoring scheme

3. Trace back the sequences for a suitable alignment.

F = max {H(i, j)};

traceback(F);

The complexity of the Smith-Waterman algorithm can also be computed. The time complexity of the initialization is *O(M+N)* because we need to initialize row 0 and column 0. In filling the matrix, we traverse each cell of the matrix and perform a constant number of operations in each cell, and thus the time complexity for this part is *O(MN)*. However, in the traceback, the algorithm requires the maximum cell be found, and this must be done by traversing the entire matrix, making the time complexity for the traceback *O(MN)*. It is also possible to keep track of the largest cell during the matrix filling segment of the algorithm, although this will not change the overall complexity.
The time complexity of this algorithm is [18],

$$O(M+N) + O(MN) + O(MN) = O(MN).$$

Before Smith Waterman algorithm the Needleman-Wunsch algorithm [24], published in 1970, provides a method of finding theoptimal global alignment of two sequences by maximizing the number of amino acid matches andminimizing the number of gaps necessary to align the two sequences. Because the Needleman-Wunsch algorithm finds the optimal alignment of the entire sequence of both proteins, it is a globalalignment technique, and cannot be used to find local regions of high similarity.

The space complexity of the Smith-Waterman algorithm is also unchanged from theNeedleman-Wunsch algorithm. This is due to the fact that the same matrix is used and the sameamount of space is needed for the traceback. Thus, there is no definite space or time advantage of onealgorithm over the other. However, the Smith-Waterman algorithm tends to model protein homologybetter because it ignores misalignments at the ends of the proteins which are often not highlyconserved. Thus, database searches are usually done with the Smith-Waterman algorithm over theNeedleman-Wunsch algorithm which tends to model homology better in distantly related proteins.The Needleman-Wunsch algorithm will tend to be better for proteins which are closely related, withfewer mutations because the ends of the protein in closely related sequences will not be changed significantly. Since Smith Waterman algorithm fills a single matrix of size *MN* and stores at most *N* positions for the traceback,the total space complexity of this algorithm is given below [18],

$$O(MN)+O(N)=O(MN).$$

An example of Smith Waterman algorithm,

- Sequence 1 = ACACACTA
- Sequence 2 = AGCACACA
- s(a, b) = +2 if a = b (match), -1 if a ≠ b (mismatch)
- $W_i = -1$

The following matrices are computed using the values above.

$$H = \begin{pmatrix} & - & A & C & A & C & A & C & T & A \\ - & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ A & 0 & 2 & 1 & 2 & 1 & 2 & 1 & 1 & 2 \\ G & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 \\ C & 0 & 1 & 3 & 2 & 3 & 2 & 3 & 2 & 2 \\ A & 0 & 2 & 2 & 5 & 4 & 5 & 4 & 4 & 4 \\ C & 0 & 1 & 4 & 4 & 7 & 6 & 7 & 6 & 6 \\ A & 0 & 2 & 3 & 6 & 6 & 9 & 8 & 8 & 8 \\ C & 0 & 1 & 4 & 5 & 8 & 8 & 11 & 10 & 10 \\ A & 0 & 2 & 3 & 6 & 7 & 10 & 10 & 10 & 12 \end{pmatrix} \quad T = \begin{pmatrix} & - & A & C & A & C & A & C & T & A \\ - & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ A & 0 & \nwarrow & \leftarrow & \nwarrow & \leftarrow & \nwarrow & \leftarrow & \leftarrow & \nwarrow \\ G & 0 & \uparrow & \nwarrow & \uparrow & \nwarrow & \uparrow & \nwarrow & \nwarrow & \uparrow \\ C & 0 & \uparrow & \nwarrow & \leftarrow & \nwarrow & \leftarrow & \nwarrow & \leftarrow & \leftarrow \\ A & 0 & \nwarrow & \uparrow & \nwarrow & \leftarrow & \nwarrow & \leftarrow & \leftarrow & \nwarrow \\ C & 0 & \uparrow & \nwarrow & \uparrow & \nwarrow & \leftarrow & \nwarrow & \leftarrow & \leftarrow \\ A & 0 & \nwarrow & \uparrow & \nwarrow & \uparrow & \nwarrow & \leftarrow & \leftarrow & \nwarrow \\ C & 0 & \uparrow & \nwarrow & \uparrow & \nwarrow & \uparrow & \nwarrow & \leftarrow & \leftarrow \\ A & 0 & \nwarrow & \uparrow & \nwarrow & \uparrow & \nwarrow & \uparrow & \nwarrow & \nwarrow \end{pmatrix}$$

To obtain the optimum local alignment, start with the highest value in the matrix (i,j). Then, go backwards to one of positions $(i-1,j)$, $(i,j-1)$, and $(i-1,j-1)$ depending on the direction of movement used to construct the matrix. This methodology is maintained until a matrix cell with zero value is reached.In the example, the highest value corresponds to the cell in position (8,8). The walk back corresponds to (8,8), (7,7), (7,6), (6,5), (5,4), (4,3), (3,2), (2,1), (1,1), and (0,0). Once finished, the alignment is reconstructed as follows: Starting with the last value, reach (i,j) using the previously calculated path. A diagonal jump implies there is an alignment (either a match or a mismatch). A top-down jump implies there is a deletion. A left-right jump implies there is an insertion.

The results are:

- Sequence 1 = A-CACACTA
- Sequence 2 = AGCACAC-A

CHAPTER 3

EXPERIMENTAL SETUP

For our experiments CUDA Toolkit 7.5 is used and NVIDIA GeForce GTX 660. All the experiments are conducted in a personal computer (PC) with the configuration Intel(R) Core i3-4160 CPU @ 3.6 GHz, 8GB RAM, running Ubuntu 14.04.

3.1 CUDA TOOLKIT

The NVIDIA CUDA Toolkit provides a comprehensive development environment for C and C++ developers building GPU-accelerated applications. The CUDA Toolkit includes a compiler for NVIDIA GPUs, math libraries, and tools for debugging and optimizing the performance of your applications. You'll also find programming guides, user manuals, API reference, and other documentation to help you get started quickly accelerating your application with GPUs.The new features of CUDA 7.5 are,

1. 16-bit floating point (FP16) data format

- Store up to 2x larger datasets in GPU memory.
- Reduce memory bandwidth requirements by up to 2x.
- New mixed precision cublasSgemmEX() routine supports 2x larger matrices.

2. New cuSPARSE GEMVI routines

- Optimized dense matrix x sparse vector routines - ideal for Natural Language Processing.

3. Instruction-level profiling helps pinpoint performance bottlenecks

- Quickly identify the specific lines of source code limiting the performance of GPU code.
- Apply advanced performance optimizations more easily.

3.2 GeForce GTX 660 GPU card

For our experiment this is the only GPU available to us. The specification of this GPU care is displayed.

TABLE 1. GPU ENGINE SPECIFICATIONS

CUDA Cores	960
Base Clock (MHz)	980
Boost Clock (MHz)	1033
Texture fill rate (billion/sec)	78.4

TABLE 2. MEMORY SPECIFICATIONS

Memory speed	6.0 Gbps
Standard memory configuration	2048 MB
Memory interface	GDDR5
Memory interface width	192-bit GDDR5
Memory bandwidth (GB/sec)	144.2

TABLE 3. FEATURE SUPPORT

Important technologies	GPU Boost, PhysX, TXAA, NVIDIA G-SYNC-ready
Other supported technologies	3D vision, CUDA, Adaptive VSync, FXAA, 3D Vision Surround, SLI
OpenGL	4.3
Microsoft directX	12 API
Bus support	PCI express 3.0
Certified for windows 7, 8, vista, XP	Yes
3D vision ready	Yes

TABLE 4. DISPLAY SUPPORT

Maximum digital resolution	4096 x 2160
Maximum VGA resolution	2048 x 1536
Standard Display Connectors	One dual link DVI-I, one dual link DVI-D, one HDMI, one display port
Multi monitors	4 displays
HDCP	Yes
HDMI	Yes
Audio input for HDMI	Internal

TABLE 5. STANDARD GRPAHCIS CARD DIMENSIONS

Height	4.376 inches
Length	9.5 inches
Width	Dual-slot

TABLE 6. THERMAL AND POWER SPECIFICATIONS

Maximum GPU temperature (in C)	97 C
Graphics card power (W)	140 W
Minimum recommended system power (W)	450 W
Supplementary power connectors	One 6 pin

TABLE 7. THERMAL AND POWER SPECIFICATIONS

3D Blu-Ray	Yes
3D Gaming	Yes
3D Photos	Yes

3.3 BLAST CUDA

Figure 8 demonstrates a detailed implementation of BLASTP algorithm. It brings light to what part of the code is sent to GPU for execution. Stages 1 and 2 of BLAST algorithm are processed in GPU namely hit detection stage and un-gapped extension. First and foremost the CPU takes the query sequences. Then it sorts the database according to the number of subject sequences it contains. This helps in balancing the load among the threads. So, no threads in the

same wrap (cluster of threads that can execute in parallel) work on subject sequences with large length difference. Later, the database is sent to kernel for calculating the HSP pairs. Once done, High Scoring Alignments (HSAs) are computed out in the CPU using gapped extension. At last final calculations are made and the results identical to NCBI-BLAST are displayed.

3.3.1 Input

The three main inputs of BLAST are the query sequence, database where the subject sequences are stored, threshold value at which an alignment must score to avoid being cut off.

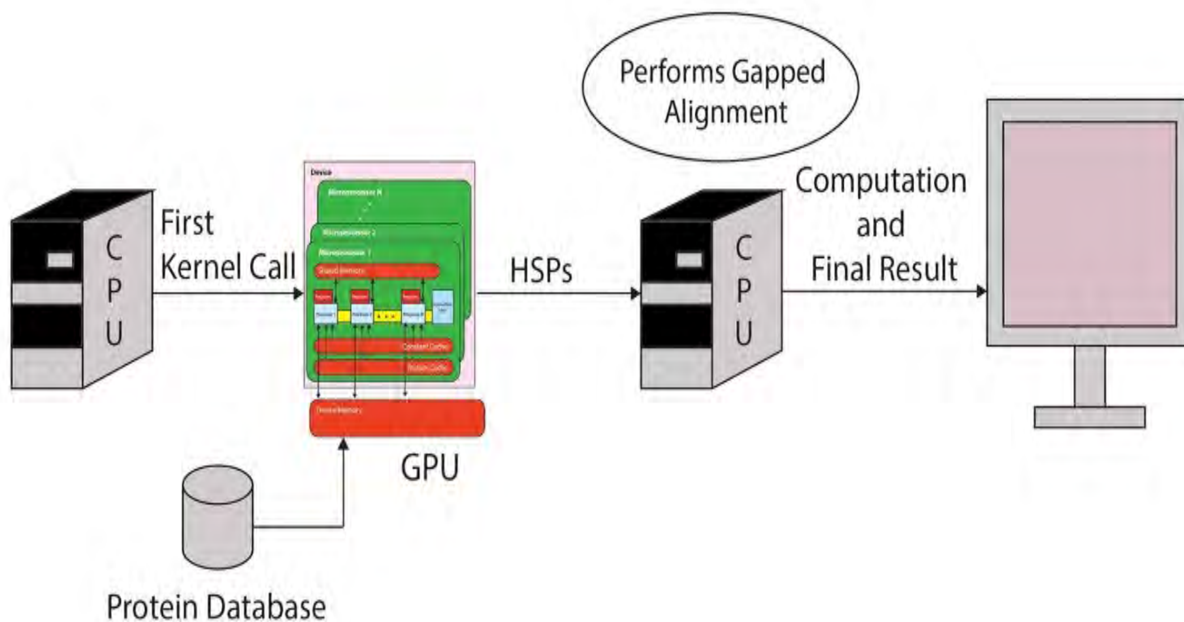

Figure 8.The experimental setup of BLAST algorithm.

3.3.2 Kernel Call

Stage 1 and stage 2 are mainly performed in this kernel. At first the query sequence inputted is broken down to several words of length 3. The protein database is stored in the global memory

ofthe GPU. Kernel is then called and the 3 letter words are sent to corresponding threads with a batch of the database. There each word is matched with the subject sequence. Whenever a match is confirmed un-gapped extension is carried out. This generates HSPs. Finally, the HSPs are read back to the host (CPU) for further processing. An overview of the first kernel is portrayed in figure 9.

3.3.3 CPU Readback

The HSPs are read backed to the CPU. Here the later part which is the gapped extension is processed. The results of gapped extension are HSAs. HSAs are filtered if they fail to overcome the threshold value. Finally, after trace backing the final results are outputted on the display.

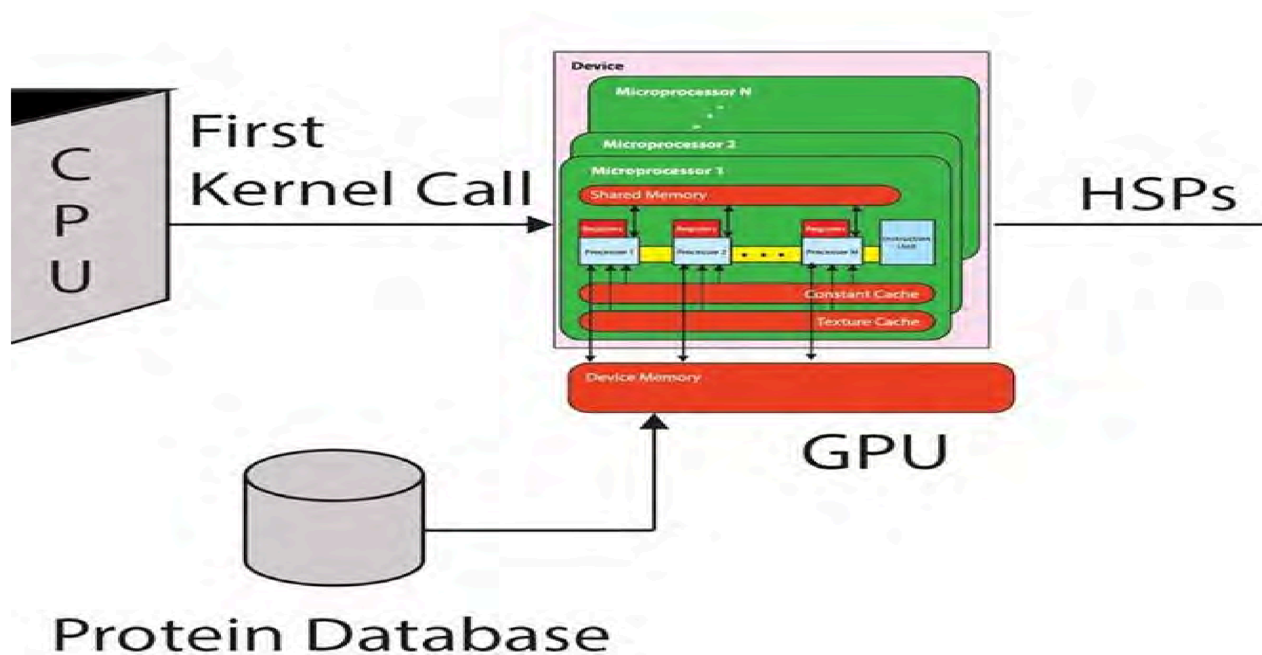

Figure 9.Kernel call.

3.4 SMITH-WATERMAN CUDA

This algorithm is very time consuming as matrices are generated against every subject sequence from the database. Thus running this algorithm in GPUs is preferable as GPUs are designed to compute matrices. The implementation of smith-waterman algorithm is illustrated in Figure 10.

The operations in doted blocks are carried out by GPU and the rest of the blocks are performed by CPU. Similar to the BLAST CUDA implementation the database is sorted so no threads on the same cluster work on subject sequences with large length difference. Each thread in the GPU is assigned to fill in the similarity matrix against one sequence from the database. Once done matching the similarity between the sequences the matrix is saved in the local memory. Then the third step of the algorithm is performed which is trace back. First the thread figures out the maximum value in the matrix and starts tracing back till it reaches zero. At last along the line of the trace back the alignment found is scored. Then the alignments with scores are read back to CPU where it organizes the results and displays it.

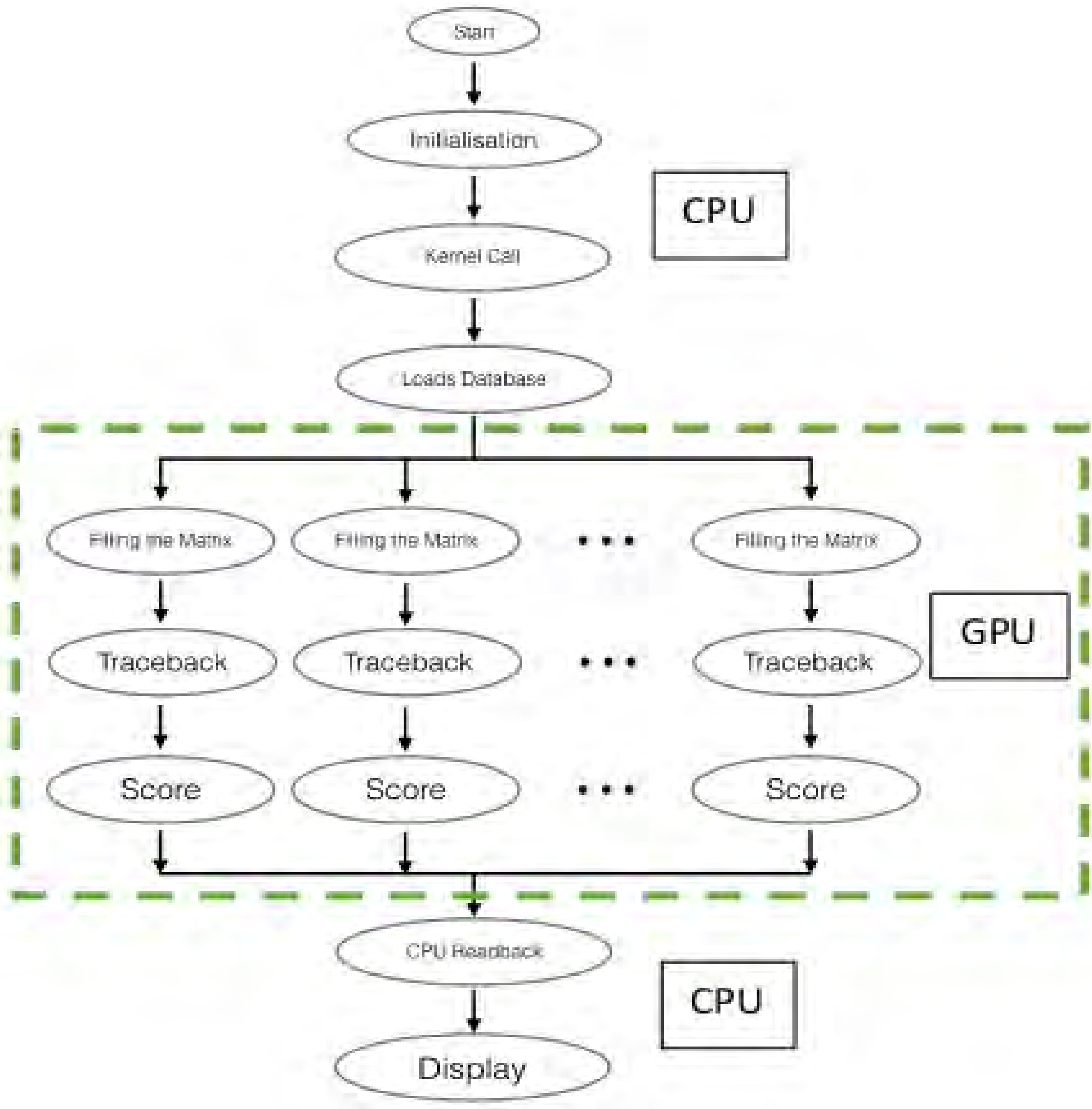

Figure 10.The experimental setup of Smith-Waterman algorithm.

CHAPTER 4

EXPERIMENTAL RESULT ANALYSIS

The env_nr database we used is of 1.5 GB. Our input query sequences are of yeast and that too retrieved from the NCBI website. The database has a total of 6,891,928 sequences; 1,364,236,057 letters. We varied the query length sequence from 26 to 1002.

At first we carried out a test to figure out the optimal block and thread size to carry out our experiments. Firstly, we kept block size constant and varied the thread size. Then we changed the block size and completed the task again.

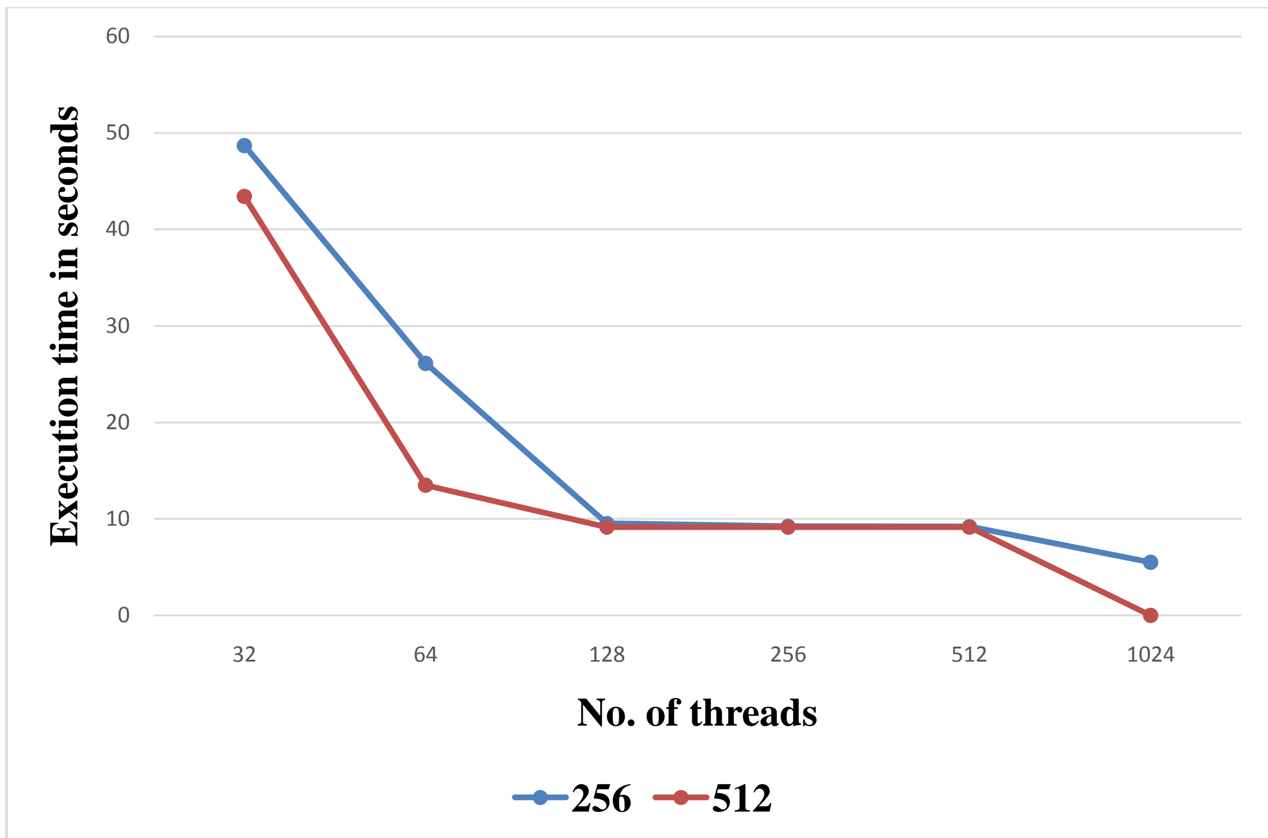

Figure 11. Comparison of execution time varying block and thread sizes.

To find the optimal thread and block sizes we used the Ubp6p protein sequence which is of length 499. The graph generated is shown in figure 11. For the last value which is marked as X, the GPU we used runs out of global memory to finish the task. GeForce GTX 660 has a memory space of 2 GB. Thus a GPU with a higher global memory will give the result. Similarly when we tried the same job with a larger sequence of length 1002 any thread size greater than 256 shows the unavailability of memory space. Finally, we conclude to complete the experiments with keeping the block size constant at 256 while changing the thread size twice 128 and 256.

TABLE 8.EXECUTION TIME OF UBP6P PROTEIN IN BLASTP. THREAD SIZE IS VARIED WITH BLOCK SIZE.

Thread size	Time for block size 256	Time for block size 512
32	48.709	43.422
64	16.131	13.495
128	9.538	9.141
256	9.235	9.154
512	9.201	9.133
1024	5.505	X

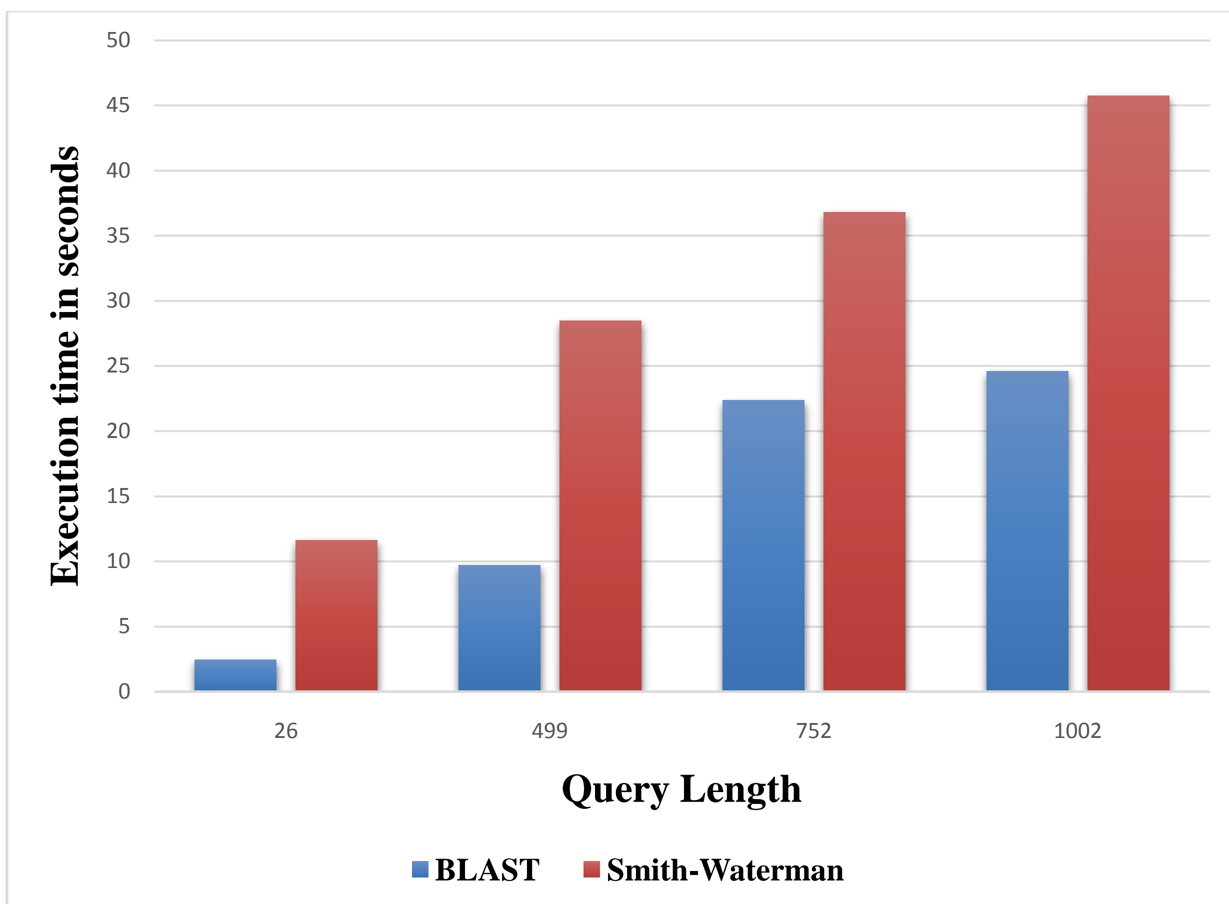

Figure 12.Comparison of runtimes achieved by both the algorithm using Table 9.

TABLE 9.RUNTIME IN SECONDS FOR BOTH THE ALGORITHMS USING 256 GPU BLOCKS AND 128 THREADS.

Query Sequence Length	GPU blocks	GPU threads	BLAST CUDA	SW CUDA
26 (SCY_4187)	256	128	2.432	11.615
499 (Ubp6p)	256	128	9.699	28.482
752 (Gcn20p)	256	128	22.370	36.799
1002 (SAP155)	256	128	24.585	45.749

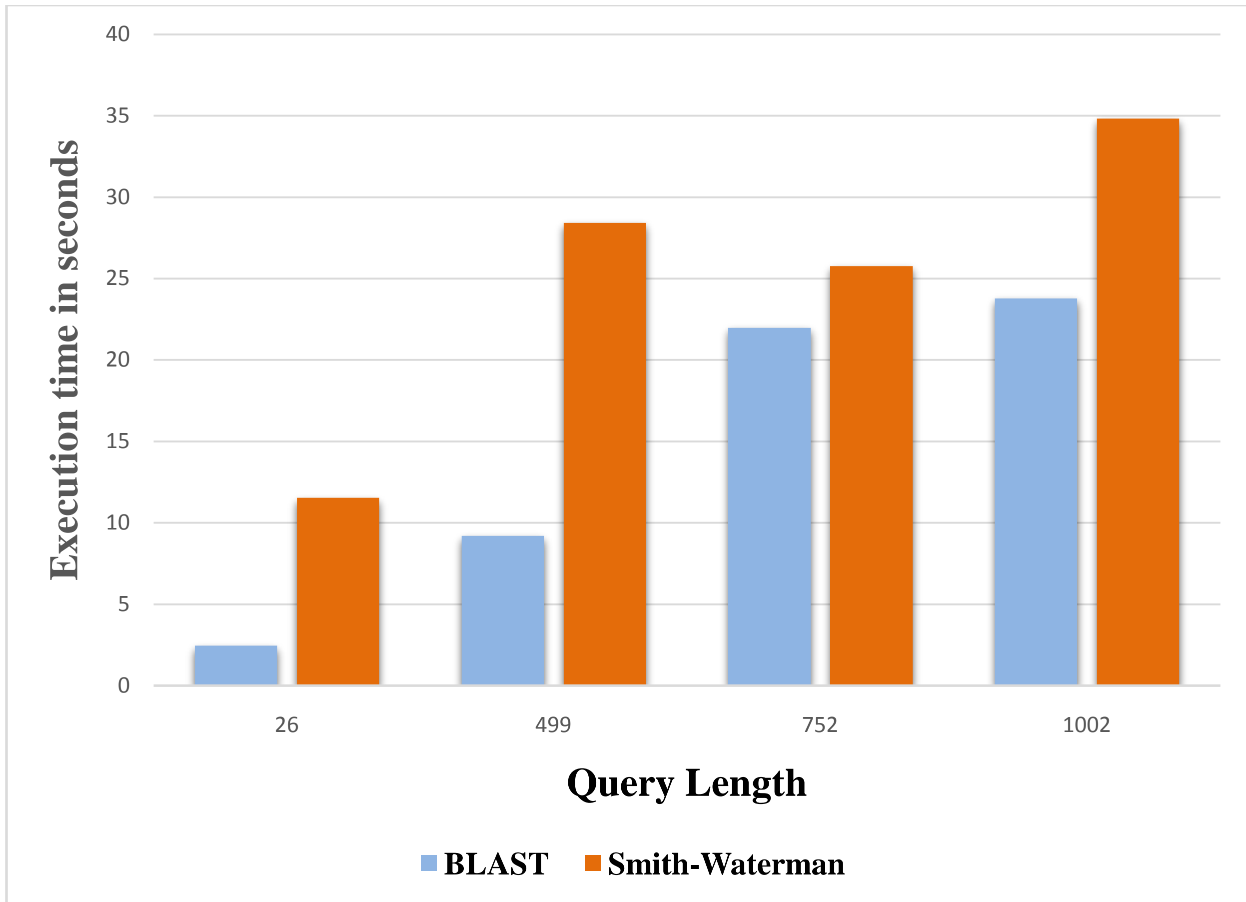

Figure 13.Comparison of runtimes achieved by both the algorithm using Table 10.

TABLE 10.RUNTIME IN SECONDS FOR BOTH THE ALGORITHMS USING 256 GPU BLOCKS AND 256 THREADS.

Query Sequence Length	GPU blocks	GPU threads	BLAST CUDA	SW CUDA
26 (SCY_4187)	256	256	2.446	11.531
499 (Ubp6p)	256	256	9.185	28.420
752 (Gcn20p)	256	256	21.963	25.764
1002 (SAP155)	256	256	23.773	34.821

The figure 14 shows the snap of some top few of the results obtained by both the algorithms. It can clearly be seen that the results vary. Smith-Waterman produces more results that BLAST cannot even compute. Even though BLAST is fast Smith-Waterman is accurate.

<table>
<tr><th>BLASTP</th><th>Smith Waterman</th></tr>
<tr><td colspan="2">Length: 499</td></tr>
<tr><td>gi|139322119|gb|ECE52589.1| hypothetical protein GOS_6347887</td><td>gi|139322119|gb|ECE52589.1| hypothetical protein GOS_6347887</td></tr>
<tr><td>gi|139946120|gb|ECI55741.1| hypothetical protein GOS_4322971</td><td>gi|139946120|gb|ECI55741.1| hypothetical protein GOS_4322971</td></tr>
<tr><td>gi|137997043|gb|EBX18915.1| hypothetical protein GOS_6626877</td><td>gi|134681576|gb|EBC79177.1| hypothetical protein GOS_13631</td></tr>
<tr><td>gi|143993272|gb|EDI09956.1| hypothetical protein GOS_488690</td><td>gi|139127642|gb|ECD59672.1| hypothetical protein GOS_6323963</td></tr>
<tr><td>gi|137118211|gb|EBS30477.1| hypothetical protein GOS_7458561</td><td>gi|140648037|gb|ECM54753.1| hypothetical protein GOS_5989641</td></tr>
<tr><td>gi|140967368|gb|ECO72834.1| hypothetical protein GOS_4287286</td><td>gi|136641435|gb|EBP45728.1| hypothetical protein GOS_7908304</td></tr>
<tr><td>gi|134681576|gb|EBC79177.1| hypothetical protein GOS_13631</td><td>gi|137520397|gb|EBU53522.1| hypothetical protein GOS_7046189</td></tr>
<tr><td>gi|139127642|gb|ECD59672.1| hypothetical protein GOS_6323963</td><td>gi|141472787|gb|ECS01292.1| hypothetical protein GOS_6389549</td></tr>
<tr><td>gi|138573566|gb|ECA53547.1| hypothetical protein GOS_446765</td><td>gi|143993272|gb|EDI09956.1| hypothetical protein GOS_488690</td></tr>
<tr><td>gi|136008245|gb|EBL29035.1| hypothetical protein GOS_8595530</td><td>gi|140866693|gb|ECO04000.1| hypothetical protein GOS_3557831</td></tr>
<tr><td>gi|141472787|gb|ECS01292.1| hypothetical protein GOS_6389549</td><td>gi|139741775|gb|ECH15721.1| hypothetical protein GOS_6394985</td></tr>
<tr><td>gi|138639899|gb|ECA98147.1| hypothetical protein GOS_6191547</td><td>gi|139060890|gb|ECD14048.1| hypothetical protein GOS_4610292</td></tr>
<tr><td colspan="2">Length:752</td></tr>
<tr><td>gi|144064981|gb|EDI61499.1| hypothetical protein GOS_402696</td><td>gi|144064981|gb|EDI61499.1| hypothetical protein GOS_402696</td></tr>
<tr><td>gi|143003590|gb|EDC10971.1| hypothetical protein GOS_1534601</td><td>gi|142069991|gb|ECV34131.1| hypothetical protein GOS_2917661</td></tr>
<tr><td>gi|142048767|gb|ECV16325.1| hypothetical protein GOS_2951825</td><td>gi|816824943|gb|KKN99364.1| hypothetical protein LCGC14_0138080</td></tr>
<tr><td>gi|142329538|gb|ECX30186.1| hypothetical protein GOS_2563185</td><td>gi|142048767|gb|ECV16325.1| hypothetical protein GOS_2951825</td></tr>
<tr><td>gi|142069991|gb|ECV34131.1| hypothetical protein GOS_2917661</td><td>gi|816731399|gb|KKN15156.1| hypothetical protein LCGC14_0988780</td></tr>
<tr><td>gi|816824943|gb|KKN99364.1| hypothetical protein LCGC14_0138080</td><td>gi|143003590|gb|EDC10971.1| hypothetical protein GOS_1534601</td></tr>
<tr><td>gi|142943356|gb|EDB69029.1| hypothetical protein GOS_1611030</td><td>gi|142044026|gb|ECV12593.1| hypothetical protein GOS_2958253</td></tr>
<tr><td>gi|816731399|gb|KKN15156.1| hypothetical protein LCGC14_0988780</td><td>gi|142329538|gb|ECX30186.1| hypothetical protein GOS_2563185</td></tr>
<tr><td>gi|144143966|gb|EDJ18415.1| hypothetical protein GOS_1740407</td><td>gi|142040101|gb|ECV08679.1| hypothetical protein GOS_2963607</td></tr>
<tr><td>gi|144215775|gb|EDJ70855.1| hypothetical protein GOS_1647075</td><td>gi|142943356|gb|EDB69029.1| hypothetical protein GOS_1611030</td></tr>
<tr><td>gi|136801936|gb|EBQ50170.1| hypothetical protein GOS_7741737</td><td>gi|142047767|gb|ECV15465.1| hypothetical protein GOS_2953538</td></tr>
<tr><td>gi|144048220|gb|EDI49090.1| hypothetical protein GOS_423864</td><td>gi|144143966|gb|EDJ18415.1| hypothetical protein GOS_1740407</td></tr>
</table>

Figure 14.Results comparison of BLAST and Smith-Waterman.

According to our results BLAST performs much faster than smith-waterman algorithm. When using 256 block size and 128 number of threads on each block BLAST is around 2-5 times faster than smith-waterman shown in figure 12. On the other hand in figure 13 when the block and thread sizes are changed to 256 and 256 respectively BLAST performs better with a speed of 1.5-4.5 times faster.

TABLE 11.RUNTIME OF BLASTP IN SECONDS FOR CPU AND GPU.

Query Sequence Length	BLAST 128	BLAST 256	BLAST CPU
26 (SCY_4187)	2.432	2.446	4.189
499 (Ubp6p)	9.699	9.185	27.474
752 (Gcn20p)	22.370	21.963	45.494
1002 (SAP155)	24.585	23.773	54.652

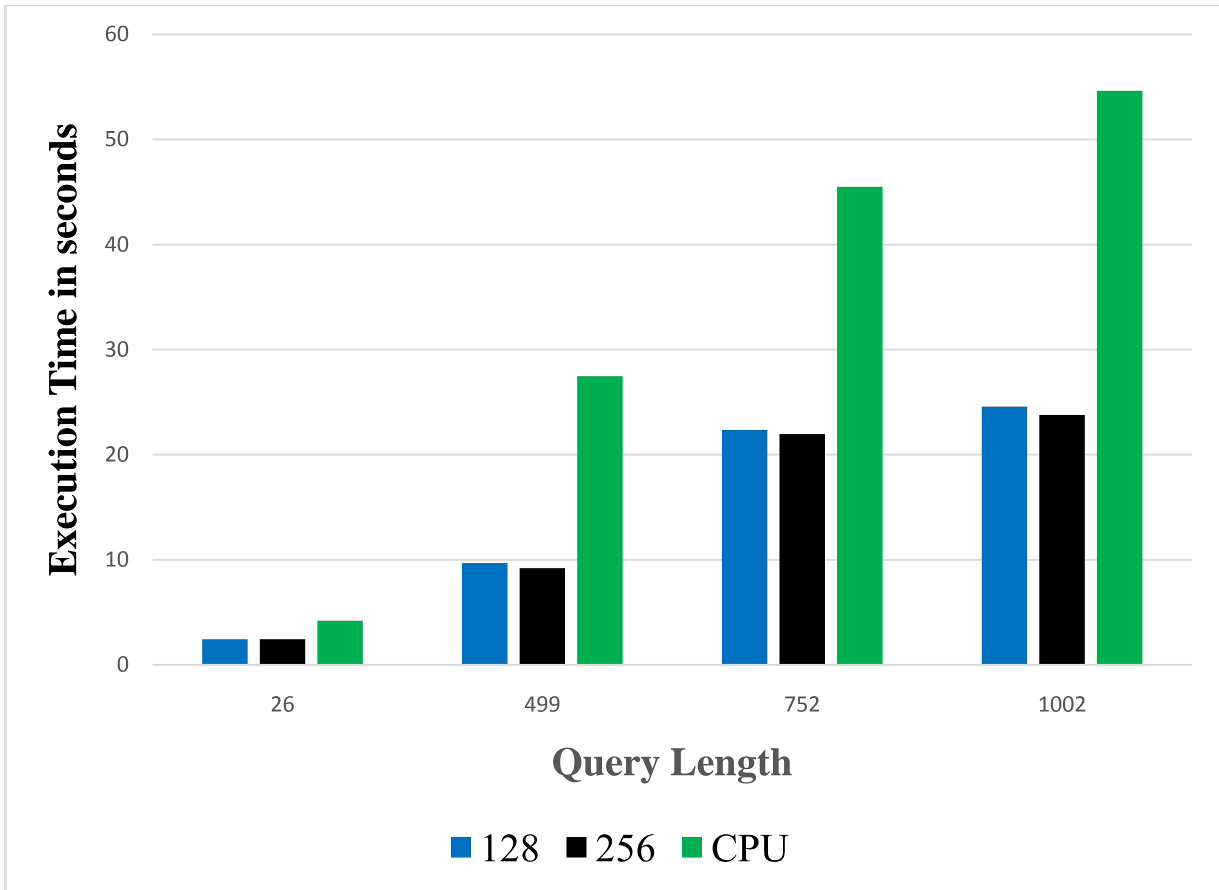

Figure 15.Comparison of runtimes achieved by CPU and GPU using Table 11.

Since BLAST is the most used algorithm we did more experimenting with it. We ran the entire algorithm in CPU and compared the results with that obtained using GPU. The results are portrayed in figure 15. BLAST 128 means block size 256 and thread size 128. While BLAST 256 means block size 256 and thread size 256.

CHAPTER 5

CONCLUSION AND FUTURE WORK

5.1 CONCLUSION

This thesis examined how much BLASTP algorithm and Smith-Waterman varies from each other in computation time using the parallel techniques of CUDA. We have collected the database of protein from NCBI and CUDA Tool kit 7.5 is used. It is on average 2.5 times faster than the BLAST. Smith-Waterman being a very exhaustive algorithm while performing in CPU is being handled pretty well in GPU. It gives a tough fight to BLAST CUDA. However, falls short in execution time. On the bright side, smith-waterman gives more accurate results than BLAST. Thus, while choosing which algorithm for their alignment task one has to decide based on accuracy or execution time. We hope our results will motivate others to work on GPUs because in today's world being fast is important. Also, GPU is providing a faster execution of BLAST CUDA than CPU we believe in the coming years with more upgrades and improvements GPU will be a force to reckon with. All in all, the next generation of GPUs will have even a better performance of BLASTP algorithm and so will Smith-Waterman.

5.2FUTURE WORKS

While doing our thesis we faced a lot of problems. To start off was to code in CUDA. We learned as much as we could but still a long way to go. Secondly, we carried our thesis on not so powerful of a GPU. It had a memory of around 2 GB, which wasn't enough. We worked with massive databases. Most of the time we exceeded the memory capacity of GPU and got error in our experimental results. Thus we had to narrow our inputs as well. Thus, in the future we hope to work on better and powerful GPUs. The more we could vary our inputs the more we can gather knowledge and move forward on GPU and CUDA.

Also, in the coming years we would like to pursue our journey towards parallel computing. When the first time we executed a CUDA code and ran the same code for CPU we were so amazed. As for the first time we managed to speed up a code by 5 seconds. That joy is what still motivates us to have a better understanding of this topic. Moreover, we have plans to use BLAST algorithm to use it as whatever alignment algorithm Google uses for its autocomplete when we search the web.

REFERENCE

[1] S. Altschul, “Basic Local Alignment Search Tool,” J. of Molecular Biology, vol. 215, no. 3, pp. 403-410, 1990.

[2] S.F. Altschul, W. Gish, W. Miller, E.W. Myers, and D.J. Lipman, “Basic Local Alignment Search Tool,” J. Molecular Biology, vol. 215, pp. 403-410, 1990.

[3] S.F. Altschul, T.L. Madden, A.A. Scha¨ffer, J. Zhang, Z. Zhang, W. Miller, and D.J. Lipman, “Gapped BLAST and PSI-BLAST: A New Generation of Protein Database Search Programs,” Nucleic Acids Research, vol. 25, no. 17, pp. 3389-3402, 1997.

[4] L. Weiguo, B. Schmidt, and W. Muller-Wittig, “CUDA-BLASTP: Accelerating BLASTP on CUDA-Enabled Graphics Hardware,” IEEE/ACM Transactions on Computational Biology and Bioinformatics, vol. 8, no. 6, pp. 1678-1684, Nov.-Dec. 2011.

[5] L. Yongchao, L.M. Douglas, and S. Bertil, “CUDASW++ optimizing Smith-Waterman sequence database searches for CUDA-enabled graphics processing units,” BMC Reseach Notes, vol. 2, no. 73, 2009.

[6] Lin, Heshan et al. “Coordinating Computation And I/O In Massively Parallel Sequence Search,” in IEEE Transactions on Parallel Distributed Systems, vol. 22, no. 4, pp. 529-543, 2011.

[7] J. Qui, J. Ekanayake, T. Gunarathne, J. Y.Choi, S.H. Bae, H. Li, B. Zhang, T. L. Wu, Y. Ruan, S. Ekanayake, A. Hughes, G. Fox, “Hybrid cloud and cluster computing paradigms for life science applications,” BMC Bioinformatics, vol. l12, no.11, 2010.

[8] T. Oliver, B. Schmidt, and D. L., “Maskell : Reconfigurable architectures for bio sequence database scanning on FPGAs,” vol. 52, pp. 851–855,2005.

[9] T. I. Li , W. Shum, K. Truong, “160-fold acceleration of the Smith-Waterman algorithm using a field programmable gate array (FPGA),” vol. 8, pp. 185, 2007.

[10] Cheng, John, Max Grossman, and Ty KcKercher. Professional CUDA C Programming. Indianapolis: Wrox, 2014. Print.

[11] Database Env_nr retrieved from ftp://ftp.ncbi.nlm.nih.gov/blast/db/FASTA/env_nr_gz.

[12] Clark, Don, "J.P. Morgan Shows Benefits from Chip Change". WSJ Digits Blog, 2011. Retrieved February 14, 2016.

[13] NVIDIA GTX 680 Whitepaper. Retrived December 30,2015 from http://www.nvidia.es/content/PDF/product-specifications/GeForce_GTX_60_Whitepaper_FINAL.pdf .

[14] Z. Jing, W. Hao, L. Heshan, and F. Wu-chun, "cuBLASTP: Fine-Grained Parallelization of Protein Sequence Search on a GPU," in Parallel and Distributed Processing Symposium, 2014 IEEE 28th International , vol., no., pp.251-260, 2014.

[15] NVIDIA Corporation. NVIDIA CUDA C Programming Guide, 2015. Version 7.0.

[16] Silberstein, Mark, Schuster, Assaf, Geiger, Dan, Patney, Anjul, Owens, D. John, "Efficient computation of sum-products on GPUs through software-managed cache," 2008, pp. 309–318.

[17] DJ Lipman, and WR Pearson, "Rapid and sensitive protein similarity search," vol. 227, pp. 1435-41.

[18] A. Chan, "An analysis of pairwise sequence alignment algorithm complexities: Needleman-wunsch, smith-waterman, fasta, blast and gapped blast," 2007, [online]: http://biochem218.stanford.edu/Projects%202004/Chan.pdf.

[19] S. Xiao, H. Lin, and W. Feng, "Accelerating Protein Sequence Search in a Heterogeneous Computing System," 2011 IEEE International Parallel & Distributed Processing Symposium, IPDPS, pp. 1212 - 1222.

[20] A. David, "An Analysis of BLASTP Implementation on NVIDIA GPUs," 2012, [online]: http://cmgm.stanford.edu/biochem218/Projects%202012/Glasco.pdf.

[21] M.Cameron, H.E.Williams, and A.Cannane "A deterministic finite automaton for faster protein hit detection in BLAST," vol. 13, pp. 965-78, 2007.

[22] T.F. Smith, and M.S. Waterman, "Identification of common molecular subsequences," J. Mol. Biol, vol.147, pp.195-197, 1981.

[23] O. Gotoh, "An improved algorithm for matching biological sequences," J. Mol. Biol, vol. 162, pp. 705-708, 1982.

[24] S. B. Needleman, and C. D. Wunsch, "A general method applicable to the search for similarities in the amino acid sequence of two proteins," J. Mol. Biol., Vol. 48, pp. 443-453, 1970.