An Efficient Implementation of Needleman Wunsch Algorithm on Graphical Processing Units

Chetan S. Khaladkar

This report is submitted as partial fulfilment
of the requirements for the Honours Programme of the
School of Computer Science and Software Engineering,
The University of Western Australia,
2009

Abstract

In bioinformatics, a sequence alignment is a way of arranging the sequences of DNA, RNA, or protein to classify regions of resemblance that may be the outcome of functional, structural, or evolutionary relationships between the sequences. This sequence alignment has become a routine procedure in Molecular Biology. Today, sequences can be obtained easily, but aligning the sequences remains a complicated aspect of comparative molecular biology. Here aligning refers to matching as many characters as possible from each sequence. A number of new programs for computing Multiple Sequence Alignment (MSA) have been developed in the last decade. Clustal is one such program which is extensively used for performing MSA.

ClustalW performs sequence alignment in three stages. In the first stage it uses the Needleman-Wunsch (NW) algorithm for pairwise alignment of sequences. In the second stage it implements the Neighbour-Joining tree for forming a guide tree of sequences. In the final stage, sequences are aligned with the help of progressive alignment. NW is based on a dynamic programming approach. Unfortunately, the computational cost required for using dynamic programming is very high as it requires a number of operations that are proportional to the product of the lengths of the sequences. The sequences used for MSA can be many hundreds or even thousands of residues or base pairs long. As molecular biologists may require the alignment of thousands of sequences, the current MSA algorithms would take longer period in computing a near optimal alignment. Thus parallelizing the algorithm is the natural step forward to reduce the running time required.

Modern graphics processing units (GPUs) can be used for high performance computing as they facilitate enhanced programmability. Furthermore, they provide good price/performance ratio. A wide range of tasks are computable on GPUs and their computational power has improved significantly. The main motivation of our work is to exploit the immense computational power of commonly available GPU, to develop an efficient solution for Multiple Sequence Alignment.

CUDA allows direct access to the hardware primitives of Graphics Processing Units (GPU). Hence we parallelized the ClustalW algorithm and implemented it on GPU with the help of CUDA, in order to explore the efficiency of GPUs for general purpose computing. We implemented the parallel approach for MSA on GTX 260 GPU. By comparing the execution times of ClustalW for CPU and our

parallel approach for GPU, we observed that there is a significant speedup for computing large number of sequences having small length. Our results confirm that MSA can benefit from the use of GPUs. These results acknowledge the fact that for efficient use of GPUs, utilization of hardware resources of GPU and the amount of parallelism exhibited by the problem are very crucial.

Keywords: GPU, CUDA, Multiple Sequence Alignment
CR Categories: C.1.4, C.1.2, I.3.1

Acknowledgements

I would like to express the deepest appreciation to my supervisor Prof Amitava Datta for his continually encouragement and support during this lengthy but exciting research process. I am grateful for his patience and the time he has dedicated to me during the course of this project. One simply could not wish for a better and friendly supervisor.

In my daily work I have been blessed with a friendly and cheerful group of fellow students who supported me throughout my thesis work. I would take this opportunity to thank Mr Prasad Lodha for helping me improve my writing and Mr Ravi Jadhav for his continual support throughout the project.

I would like to express my gratitude to all those gave me the possibility to complete this thesis. I want to thank the School of Computer Science and Software Engineering of The University of western Australia for provided the support and equipment I needed to produce and complete my thesis.

Finally, I thank my parents Shirish Khaladkar and Savita Khaladkar for their special support and love. To them I dedicate my thesis.

Contents

List of Tables

List of Figures

CHAPTER 1

Introduction

1.1 Overview of GPU

Traditionally, improving the performance of CPUs would mean increasing the clock frequency. But, it has reached a threshold level from where it is hard to meet the ever increasing demand of faster and faster computation speed just by increasing the clock speed. So now the main focus is on increasing the number of computing cores to increase the performance. Current top of the range CPUs have 4 to 6 cores and soon we will see 8 core CPUs making their way in the market as their designs are already under development. In the near future one can expect designs with more than 30 cores. AMD, one of the market leaders in manufacturing CPUs, have its roadmap leading to 12-core processors by 2010 and 16-core processors by 2011 [19].

Recently, GPU manufacturers are not only concentrating on developing GPUs for graphics oriented tasks but also making GPUs proficient in general purpose computing tasks. The idea of using a GPU as an additional calculation unit emerged several years ago, when GeForce FX, graphics card of NVIDIA came to market, which supported single precision floating-point calculations [21]. However, the use of GPU for general purpose computing did not materialise due to major hurdles posed by rigid architecture of the devices and the lack of adequate development platforms. However, the ever increasing demand of complex tasks paved way for development of unified architecture-based devices which largely supported general purpose applications.

There are various graphic cards manufactured by AMD, NVIDIA and other companies in the market that can be used for general purpose computing. AMD has introduced CTM (Close To Metal), and NVIDIA has introduced CUDA (Compute Unified Device Architecture) to take advantage of their respective graphics cards. Such graphics cards contain hundreds of computing cores in them. Large scale computations can be performed with the help of these graphics cards for where a data parallelism can be approached. The peak performance of these

cards is at least one order of magnitude higher than that of traditional CPUs. Various algorithms have been implemented on these platforms where a data parallelizing approach can be taken. In many cases it has shown improvement in performance. GPUs became an attractive option because they offer extensive resources even for non-visual, general-purpose computations: massive parallelism, high memory bandwidth, and general purpose instruction sets, including support for both single and double-precision IEEE floating point arithmetic [22].

1.2 Overview of Sequence Alignment

In bioinformatics, a sequence alignment is a way of arranging the sequences of DNA, RNA, or protein to classify regions of resemblance that may be the outcome of functional, structural, or evolutionary relationships between the sequences. Today, sequences can be obtained easily, but aligning the sequences remains a complicated aspect of comparative molecular biology. Here, aligning refers to matching as many characters as possible from each sequence. Many bioinformatics tasks like, detecting point mutations, predicting biological function, assessing gene and protein homology, constructing evolutionary trees, classifying genes and proteins, secondary and tertiary protein structure, etc. depend upon successful alignments. Alignments are conventionally shown as traces.

Amino acid sequences of some protein are closely related with each other but, in order to find the relation between the two sequences a comparison should be done. Sometimes the sequences are so closely related that simple visual comparison of the two sequences reveals the coincidence between them [2]. If there is any extraneous information related to the sequences, that can be taken into account while performing manual alignment. Automated alignment may suffer from the local minimum problem.

However, it is not always possible to detect the matches only with the help of visual comparison. It becomes very tedious and also the comparison is left to intuitive rationalization. Needless to say, this method alone will be very time consuming and impractical. Thus a computer adaptable method was designed by Needleman and Wunsch [2] in 1970, but it takes a considerable amount of time for larger sequence alignment. Various other heuristics algorithms were available like those of Fitch [13] in 1966 and Dayhoff [14] in 1969. There are various algorithms available to speed up the process of sequence alignment. We will discuss a few of them in Chapter 2.

There are two main areas of alignment: pairwise sequence alignment and multiple sequence alignment.

1.2.1 Pairwise sequence alignment

Pairwise sequence alignment is related with comparing two DNA or amino acid sequences and aligning the sequences in such a way that, they are padded by gaps to achieve the same length and maximum similarity between the sequences can be seen. Based on differences between the two sequences, one can calculate the cost of aligning the two sequences. The basic methods for producing pariwise alignments are dot-matrix method, dynamic programming, and words method. [2].

1.2.2 Multiple sequence alignment

Multiple sequence alignment (MSA) aims to find similarities between many sequences. MSA is hard and less tractable than pairwise alignment. Dynamic programming is often impractical for a large number of sequences.

1.3 Overview of Multiple sequence alignment

Multiple sequence alignment (MSA) is one of the fundamental tasks carried out in aligning DNA or amino acid sequences. Multiple sequence alignment refers to the sequence alignment of more than two biological sequences of DNA, RNA or protein. Multiple sequence alignment (MSA) has assumed a key role in comparative structure and function analysis of biological sequences. It often leads to fundamental biological insights into sequence-structure-function relationships of nucleotide or protein sequence families[3]. This alignment has to be carried out in an optimal way to match as many characters as possible from the sequence. However, aligning large number of sequences consumes plenty of time and resources. There is a rapid growth of biological sequence databases which exerts stress on computing the MSA in much shorter period of time.

Multiple sequence alignment (MSA) is a crucial task in bioinformatics. MSA of large sequences is computationally demanding and classified as an NP-hard problem. It can detect or demonstrate homology between new sequences and existing families of sequences. MSA is an active field of research in computational biology.

1.4 Thesis Outline

Chapter 1 provides general background information and thesis outline. In chapter 2, we will go through how multiple sequence alignment was carried out in the beginning and the evolution of different algorithms and techniques in order to reduce the computation time. Chapter 3 describes the working of ClustalW. Chapter 4 covers CUDA and GPU architecture in depth. Chapter 5 describes the approach we are adapting to implement MSA. In chapter 6 we evaluate the performance of our implementation. Conclusion of the thesis and future work is presented in chapter 7.

CHAPTER 2

Literature Review

2.1 Algorithmic Approaches

In this section, we will see the evolution of different sequence alignment methods and their weakneses and strengths over each other.

2.1.1 Needleman and Wunsch (NW) Algorithm

In the NW algorithm, the smallest unit of comparison is a pair of amino acids, one from each protein. The maximum match can be defined as the largest number of amino acids of one protein that can be matched with those of another protein while allowing for all possible deletions [2]. In this method, the designers realized that if you have two sequences, it is possible to align any member of one sequence to any member of the other sequence, thus the process lends itself to a matrix format. NW algorithm proved to be revolutionary because it made alignments quick enough to be used for reasonable size sequences. In the basic NW, first a similarity matrix is built where each row and column represents a sequence. If the corresponding row member and column member match, then a score of one is assigned to the cell (Figure 2.1). Figure 2.2 represents the best possible alignment till a particular element. After the second matrix is complete, the highest score is searched and a path is drawn which corresponds to the best alignment.

Later the basic NW algorithm was modified in order to improve efficiency and speed. Penalties were added for gaps in the sequence so extended gaps had higher penalties. Conservative and non-conservative mismatches were given different scores. There were various methods developed on similar lines of NW with a little variation to it. BLOSUM and PAM matrices were among the methods which were developed on similar scoring schemes as that of NW method. Then mathematically rigorous algorithms were suggested by Sankoff in 1972 , Reichert et al.[9] in 1973 and Beyer et al.[9] in 1979. They were good mathematically but they were not that satisfactory from a biological point of view.

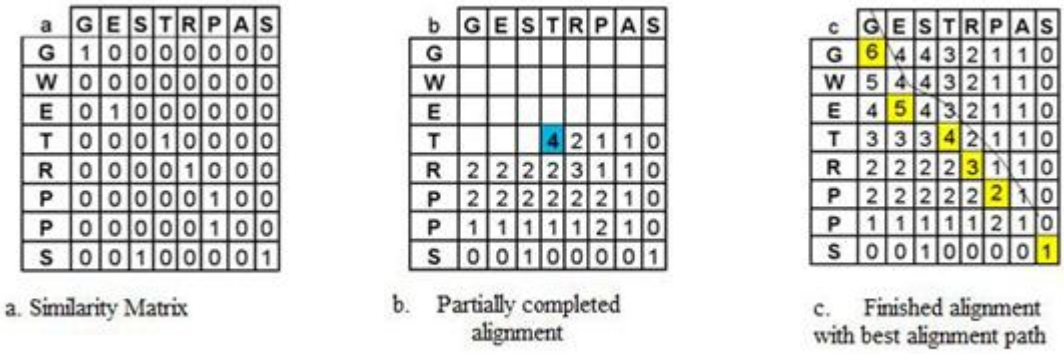

Figure 2.1: Different Matrix

G W E - T R P P - S
G - E S T R - P A S

Figure 2.2: Final Alignment

A biologically more appealing interpretation of gaps was proposed by mathematician Peter Seller [9] in 1974 where he did sequence alignment in terms of distance between sequences instead of similarity between sequences. This metric of distance between sequences was used by Waterman et al.[8] in 1976 where they included deletions and insertions of arbitrary length.

2.1.2 Smith-Waterman method (SW)

SW method is a local alignment method. SW uses same algorithm as that of NW but instead of comparing a sequence as a whole it compares all the sub-segment of both sequences. This helps in finding better scores. SW differs with NW in the sense that negative scoring matrix are set to zero which helps in finding local alignments. In SW, backtracking starts with the highest scoring matrix cell till the zero score cell is reached [8].

In 1970s NW and SW were developed which proved sufficient for few years. However, as the number of DNA and protein sequences grew exponentially, it

quickly became infeasible to compute an optimal alignment between a query sequence and every sequence in the database of known sequences [25]. As a result, researchers developed heuristic techniques such as CLUSTAL [10] and BLAST [11] to more quickly discover highly similar alignments, while not reporting more distant alignments.

2.1.3 Blast

Today, the most commonly used sequence alignment program for pairwise alignment is Blast and various other programs derived from Blast [12]. Blast is most noted for its impressive speed. Blast works on a principle of hashing small matching sequences and then extending the hash matches to create high scoring segment pairs until you attend the highest possible score. BLAST is more than 50 times faster than any method using dynamic programming but BLAST does not guarantee the optimal alignment of the query and database as in dynamic programming.

2.1.4 Clustal

The next big innovation came with the development of various methods for performing multiple sequence alignments. A vast number of methods were developed for this, including programs such as CLUSTAL, which performs alignments based on clustering the sequences, and various programs that use Hidden Markov Models in order to create sequence profiles [10]. CLUSTAL has two main variations CLUSTAL-W and CLUSTAL-X. CLUSTAL-W is a command line interface and CLUSTAL-X has graphical user interface. Mostly approximation algorithms are used for multiple alignment tasks by CLUSTAL-W [11]. One major concern for CLUSTAL-W and other methods implementing progressive alignments is their dependency on initial pair wise sequence alignment. Due to this the error may propagate from initial alignments. ClustalW is explained in more depth in Chapter 3.

2.2 Different Hardware Approaches

In this section we will see implementation of various methods of multiple sequence alignment on different hardware platforms.

2.2.1 CPU

Datta and Ebedes [1] implemented Multiple Sequence Alignment in parallel on a cluster of workstations. They implemented CLUSTAL W. In the first stage, they calculated a distance matrix using the sequences themselves as input. The distance matrix is then used to construct the guide tree, which in turn is used in the final progressive alignment stage. With the help of CLUSTALW, they achieved an overall speedup of 5.5 times on total execution time with six processors, for an efficiency of approximately 90%.

Boukerche et al. [15] used an eight-machine cluster to implement parallel strategies for local biological sequence alignment. The results obtained with large sequence sizes show good speedups when compared with the sequential algorithm. For instance, to compare 50KBP (kilo-base pair) sequences using eight processors, heuristic-block provides a reduction of 304% over the execution time of the heuristic strategy on the same platform. Also, for the 80KBP sequences, the pre-process strategy runs approximately 12 times faster than a heuristic strategy. In this paper, they proposed and evaluated three parallel strategies to solve the DNA local sequence alignment problem. For all the three strategies the wavefront method was used. A distributed shared memory (DSM) system was also used in finding local biological sequence as it offers an easier programming model than other message passing models. The first two strategies (heuristic and heuristic block) used heuristics to reduce the space complexity to O(n), where n is the size of the sequences to be compared. In the third strategy, the original SW algorithm was used and the most relevant columns of the result matrix were saved to disk. These columns can be later processed in order to retrieve the actual alignments.

Size n x n	Serial	2 proc	4 proc	8 proc
15Kx15K	296	283.18	202.18	181.29
50Kx50K	3461	2884.15	1669.53	1107.02
80Kx80K	7967	6094.18	3370.40	2162.82
150Kx150K	24107	19522.95	10377.89	5991.79
400Kx400K	175295	141840.98	72770.99	38206.84

Figure 2.3: Execution Time (seconds) for four sequence sizes [15]

Figure 2.3 shows execution times for each n x n sequence comparisons, where n is the approximate size of both sequences, with 1, 2, 4 and 8 processors. An overall speedup of 4.58 times the total execution time was obtained. For instance, two sequences of 400 KBP where compared in just 10 hours as compared to 2

days required on a single CPU [15].

The above two papers proved that using clusters of computers in parallel will certainly speedup the process of sequence alignment. However, for larger sequences it may appear to be an expensive approach.

2.2.2 Other architectures

In the past, different parallel implementations of pair-wise sequence comparison algorithms using dynamic programming techniques range from the exploitation of instruction level parallelism in uniprocessor machines to SIMD (Single Instruction stream, Multiple Data stream) and MIMD implementations [16]. Parallel hardware such as special purpose VLSI, reconfigurable hardware and programmable co-processor designs have been extensively used for sequence alignment and database search.

Some work has been reported in the field of hardware accelerators and some basic software platforms for parallel computers have been developed which provide a general framework for sequence similarity searching [20, 23]. None of these implementations has made use of fine-grain multithreading. Martins et el.[16] came up with an innovative idea of using fine-grain multithreading to improve the performance of the dynamic programming algorithm. They efficiently implemented the dynamic programming algorithm on EARTH, which is a fine-grain event driven multithreaded execution and architecture model. Following results were achieved,

Speedup The multithreaded implementation has achieved impressive speedup: up to 90 on a 120-node platform .

Scalability. A good scalability is obtained from 4 up to 120 nodes.

Processor Efficiency Very good efficiency has been achieved; on 8 nodes, processors are utilized 99% as much as on one node running sequential code. Even on 120 nodes, processors achieve 75% utilization on the largest problem size.

Further, accelerator processors like APs and GPUs were introduced. GPUs became increasingly more powerful and ubiquitous. In general, the applications that have performed well in a GPGPU model are those that can decompose their problems into highly independent components and have a high arithmetic intensity [25]. GPU performance has been increasing from two to two-and-a-half times a year [26]. This growth rate is faster than Mooree's law, as it applies to

CPUs which correspond to about one-and-a-half times a year. Subsequently, the use of GPUs for high performance computing will increase in the near future. Thus we would try to exploit the GPU architecture in order to speed up the sequence alignment process.

Graphics Processing Unit (GPU)

The recent introduction of cost-effective accelerator processors (APs), such as the IBM Cell processor and Nvidia's and AMD's graphics processing units (GPUs), represents an important technological innovation which promises to unleash the full potential of atomistic molecular modeling and simulation for the biotechnology industry [14]. GPU's are low cost hardware with capability of high throughput, thus they are considered to be a good match for bioinformatics applications. The main advantage of GPU over other architectures such as FPGAs is that they are commodity components. Most users already have access to PC's with graphics cards so for those users it provides a zero cost solution. Even if the graphics card has to be bought, the installation of such a card is trivial.

As discussed above, two major vendors of GPUs are Nvidia and AMD. AMD uses Close to Metal (CTM) as their developing platform.CTM is a low level non-graphical application programming interface (API) which gives developers direct access to the native instructions set and memory of massively parallel computational elements in AMD video cards. Nvidia uses CUDA to exploit the core of Nvidia GPU family. CUDA is a C-like programming language. Both of the programming models have their advantages over their previous GPGPU programming models, in particular those set by traditional graphics pipeline and the graphics programming interface like OpenGL and Direct3D.

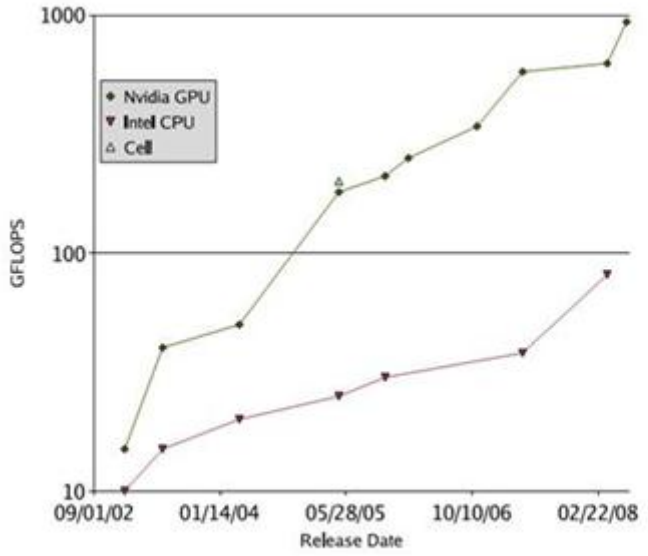

Figure 2.4: Comparison among Nvidia GPU, Intel CPU and cell [14]

Graph 2.4 depicts a theoretical peak performance trend of Nvidia and Cell processors in comparison to contemporary Intel processors. Since 2002, Nvidia GPU performance has increased by approximately 100% per annum while that of Intel CPUs by 50% per annum.

2.3 Related work

Liu et al.[13] effectively implemented a streaming architecture of GPUs to accelerate MSAs. They reformulated the computationally expensive first stage of CLUSTAL-W so that it suits GPU architecture. OpenGL Shading Language (GLSL) was used to evaluate the algorithm on nVidia GeForce 7800 GTX. It achieved speedups of almost ten compared to first stage of CLUSTAL-W and six as compared to the overall runtime. As compared with the approach of using clusters of PC, graphics hardware acceleration is superior in terms of price/performance. This solution is also scalable in the sense that several GPUs can be used by partitioning the individual pair-wise sequence comparisons. However, it shows that the first stage of CLUSTAL-W dominates the run time. If the first stage is mapped more efficiently onto graphics hardware then significant speedup can be achieved. The first stages of several other MSA tools such as

T-coffee [14] and MUSCLE [14] are also based on computation of pairwaise distances. Thus, these tools can also achieve similar speedups from the accelerator presented in this paper.

Michael et al. [25] presented a string matching program that runs on GPU. Sophisticated data structures called suffix trees were used. String matching algorithms were used in parallel in order to speed up the process. Its string matching kernel used Compute Unified Device Architecture(CUDA). Suffix tree operations are extremely low arithmetic intensity operations. Thus, string matching with suffix tree lookup is expected to perform poorly with parallel GPGPU architecture. However, Michael et al [25] with the use of cached texture memory and data reordering, achieved speed up of 35X. CUDA facilitated the use of GPU environment to formulate and implement the algorithms directly using a tree structure.

CHAPTER 3

CLUSTALW

A number of new approaches for computing MSA have been developed in the last decade. The Clustal program is extensively used for carrying out automatic multiple alignments of sets of nucleotide or amino acid sequences. There are two main variations of Clustal, ClustalW and ClustalX. ClustalW has a command line interface and ClustalX has a graphical user interface. The most familiar version is ClustalW (Thompson et. al., 1994), which uses a simple text menu system that is portable to most of the computers.

Clustal programs have become popular not only because they produce good alignments but also because of their portability to most computer platforms [8]. The simple text based menus of Clustal are easy to use. The basic idea behind the ClustalW algorithm for building multiple alignments is centered on aligning the most related sequences first. By aligning these sequences earlier on, one can minimize the instances of unnecessary gap insertions. It works well when the sequences are adequately closely related. However, a globally optimal solution cannot be guaranteed [1]. In order to identify similar sequences, the ClustalW program estimates all pairwise combinations for the input protein sequences. Thompson et. al. incorporated position-specific gap penalties, so that gap penalties can be lowered at hydrophilic residues and wherever gaps are already introduced into the alignment [2]. This method becomes less reliable when sequences have less than 30% residue identity. In such cases, the automatic alignments need to be refined manually.

The main advantages of Clustal programs are their ability to work in very little core memory, and the combination of speed, sensitivity and capacity. In Clustal programs, the number of gaps and the length of gaps are considered to be two distinct quantities, thus different weights are given to each. Different types of mismatches are also given different weights. For example a transaction (leu - val) is more probable than transversion (leu - asp), and thus is treated with different weights [9]. Sequences that are recently related through evolution are more likely to be similar because they haven't had time to diverge. Thus Clustal takes into consideration the evolutionary distance while giving weights

to sequences. Regions of hydrophobicity and proximity to other groups are the secondary structure features, which are incorporated in Clustal programs. The Clustal program takes advantage of the profile alignment rather than only doing pariwise alignment. Here, profile alignment refers to aligning two existing alignments or adding a series of new sequences to an existing alignment.

The alignment is carried out in three steps, pairwise alignment, guide-tree generation and progressive alignment. The pairwise calculation represents the first stage of the Clustal algorithm. In this stage for N input sequences there are N(N-1)/2 pairs of sequences because of the pairwise matrix symmetry [4]. This is then followed by the construction of a guide tree; this is the second stage of the algorithm. The guide tree is calculated by the neighbor-joining method, which improves tree topology and provides a method for weighting sequences on the basis of their divergences. The guide tree contains the most closely matching sequences located on the same branch. The guide tree is further used as a guide during the third stage that is the progressive alignment. In this thesis we will analyze ClustalW with respect to the time required for alignment of various amino acids sequences. We will analyze the time required for all the three steps mentioned above.

3.1 Method

The Clustal series of programs are widely used for multiple sequence alignments and for preparing phylogenetic trees. The Clustal programs have undergone several incarnations, in 1997 ClustalW 1.7 (upgrade) and ClustalX was released, which has a windows interface. ClustalW program implements a progressive method for multiple sequence alignment. In ClustalW sequences are added one by one to the aligned sequences in order to form a new alignment from that. Sequences are added to the new alignment with the help of a phylogenetic tree, which is called a guide tree. The similarity between all the possible pairs of sequences constitute for the construction of a guide tree.

The Clustal algorithm consists of three phases,

3.1.1 Stage 1 - Distance Matrix

In this stage, pairwise alignment of all sequences is carried out based on NW algorithm. In NW algorithm an alignment matrix is computed. In this matrix the number of columns and rows are decided by the query sequence and the matching sequence respectively. The computation is based on a substitution matrix and on

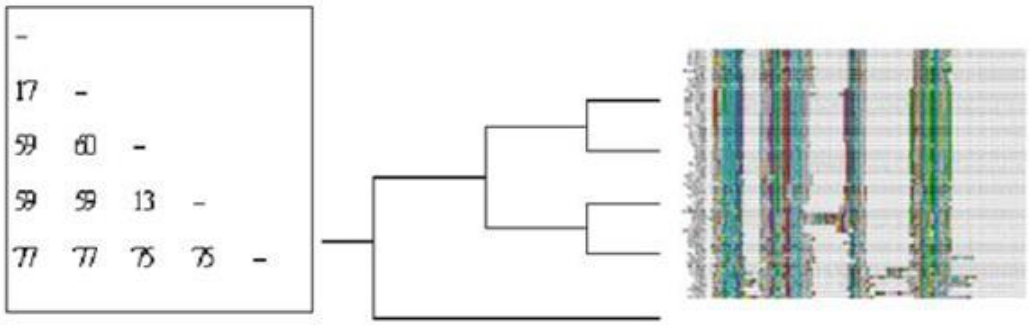

Figure 3.1: Stages of ClustalW algorithm. Left to right: pairwise alignment, Guide Tree, Progressive Alignment

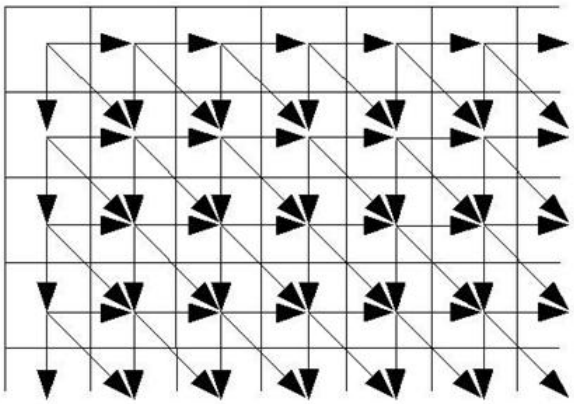

Figure 3.2: Matrix Dependency

a gap-penalty function. This pairwise alignment is used to compute a distance between all pairs of sequences. There are various ways to compute this distance. One of the widely used methods for assigning distance is as follows, for each pairwise alignment, count the number of mismatches between the two sequences, and then divide this value by the number of non-gapped pairs to calculate the distance. An important point to be considered is that any cell of the alignment matrix can be computed only after the values to the left, above and top left are known, as shown in figure 3.2. Different cells can be simultaneously processed only if they are on the same anti-diagonal. The anti-diagonal pattern is shown in figure 3.3.

The NW algorithm consists of three steps - Matrix Initialization, Matrix

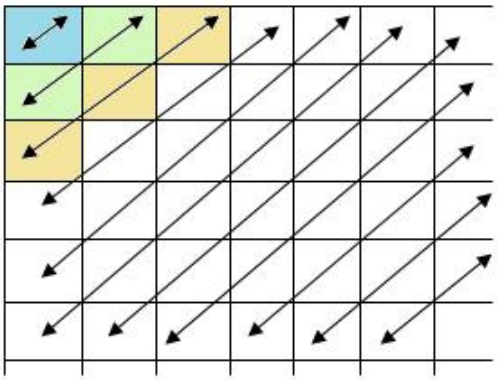

Figure 3.3: Anti-Diagonal Pattern

Calculation and Traceback. Here we are concerned with the first two steps as the traceback is performed in MSA after the calculation of guide tree. The following example will illustrate the working of algorithm,

Matrix Initialization

Consider sequences A and B. The first step consists of creating a matrix with m rows and n columns, where n is the length of sequence A and m is the length of sequence B. Considering the worst case there can always be a possibility of starting the alignment with a gap. Thus, the first row and first column are assigned 0, -2, -4 etc.

Matrix Calculation

The matrix calculation starts from the top left column. As discussed above, in order to find out the value a cell Matrix(i,j), for all pairs of i and j, the values to its top (Matrix(i-1,j)) , left (Matrix(i,j-1)) and top left (Matrix (i-1,j-1)) must be known in advance. Where, Matrix (i,j) can be calculated with the help of following equation,

Matrix(i,j) = Maximum of { Matrix(i-1,j-1) $\pm$ 1, Matrix(i-1,j) -2, Matrix (i,j-1) -2 }

In the example consider calculating the value of Matrix(1,1),
Matrix(1,1) = Maximum{ Matrix(0,0)-1, Matrix(0,1)-2, matrix(1,0)-2}

= Maximum {-1, -4, -4}
= -1

Similarly, we calculate the values for the entire matrix.

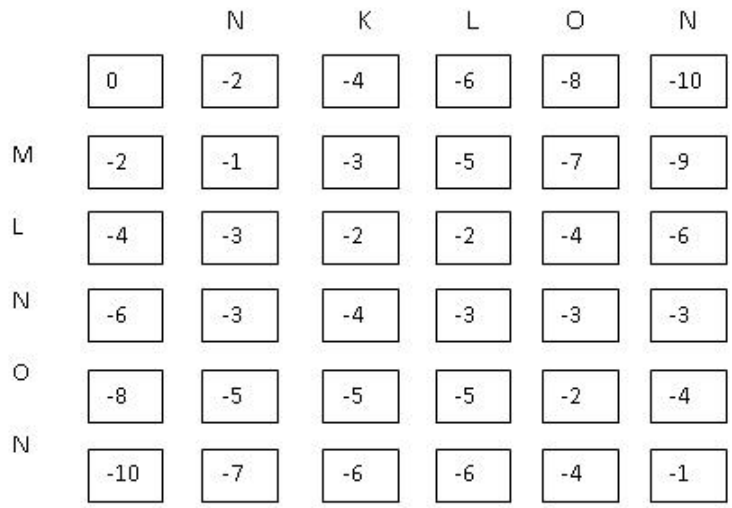

Figure 3.4: Distance Matrix

A score of the matrix is the last cell of the matrix. For every pair of sequences we are interested in this score of matrix. Similarly, we calculate the score for all pair of sequences in this stage.

After computing this distance for all pairs of sequences, they are put into the matrix (Fig 3.1- left).

3.1.2 Stage 2 - Guide Tree

The guide tree is calculated from the distance matrix using a neighbor joining algorithm (Fig 3.1 Middle). It defines the order in which the sequences are aligned in the next stage. The root of the tree is placed at the midpoint of the longest chain of consecutive edges. ClustalW also offers other methods for tree calculation but neighbor joining is widely used as it is known to be a more robust method. In neighbor joining tree method the initial input is the distance matrix and the initial tree is the start tree. Further, a new distance matrix is constructed. In this new distance matrix each pair of nodes is altered depending upon their mean divergence from the rest of nodes. The least-distant nodes are found from the new distance matrix with the help of which a guide tree is constructed. While linking the two distinct-nodes their common node is appended to the guide tree.

The end nodes are removed from their respective branches. Thus now the new nodes become the terminal nodes of the tree.Hence at every stage, a new node is added in the place of two terminal nodes. The end of the process is achieved when only two nodes are left which are separated by a single branch.Ideally, the distance between the elements in the distance matrix is same as that of the sum of branch lengths between these elements in the tree. However, this case is seldom possible in practice.

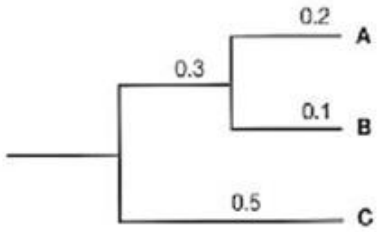

Figure 3.5: Guide Tree

Figure 3.5 shows a simple example of a guide tree. The number associated with each branch denotes the distance as calculated in stage one.

3.1.3 Stage 3 - Progressive alignment

In this stage the sequences are progressively aligned following the guide tree. The term "progressive" is used because ClustalW determines the most related sequences first and then progressively adds less related sequences or groups of sequences to the initial alignment. For example, if we consider figure 3.5, then in the progressive alignment step first sequences A and B are aligned, then sequence C is aligned with the output of previous alignment. And the result of Progressive alignment is the final multiple sequence alignment. Progressive alignment also consists of a traceback step, wherein the optimal alignment for the two sequences is achieved. In traceback step, traversing is done from the last element in the matrix (Matrix(n,m)) until we reach the left most or the top most position of the table.

For the above example progressively aligned sequences are as follows:
Sequence1 NK-LO-N

Sequence2 NKOL–M
Sequence3 -M-LNON

3.2 Other issues

In order to consider the evolutionary relationship, each sequence is not assigned equal weight. Two sequences that are closely related receive less weight than two sequences which are less closely related. The closely related sequences contain duplicate information so less weight is given to these sequences.

There are no guarantees in respect to the quality of alignment. The alignment found does not say anything about the optimum MSA. The errors at the initial stage are bound to propagate. There can be more than one optimum pairwise alignment possible, yet by using ClustalW we are committing ourselves to only one at the outset [7]. The order in which ClustalW adds the sequences to the alignment (e.g. based on the guide tree) affects the alignment. In general if any pair of sequences is less than 25% identical, then the alignment is prone to be bad.

ClustalW can work in two different types of modes, fast alignment and slow alignment. Fast alignment is more approximate alignment. In this type of alignment gap-penalties or total weighted matrix are not considered. Thus, this algorithm works much faster than the slow alignment. Slow alignment gives more exact alignments. We are using ClustalW with slow alignment in our experiment.

3.3 Experiment

We analyzed the performance of ClustalW by measuring the time required for ClustalW to align different nucleotide or protein sequences. We randomly selected a few sequences from the data available at national centre for biotechnology information (NCBI) website [6]. We timed all the three phases for sequence alignment, that are pairwise calculation (PW), guide tree calculation (GT), and progressive alignment (PA). We implemented the experiment under the following environment,

ClustalW version ClustalW-2.0.11

Processor Intel(R) Core(TM)2 DUO CPU 2.16 GHz.

Memory (RAM) 1 GB.

System Type (operating system): Ubuntu Linux version 8.04.1.

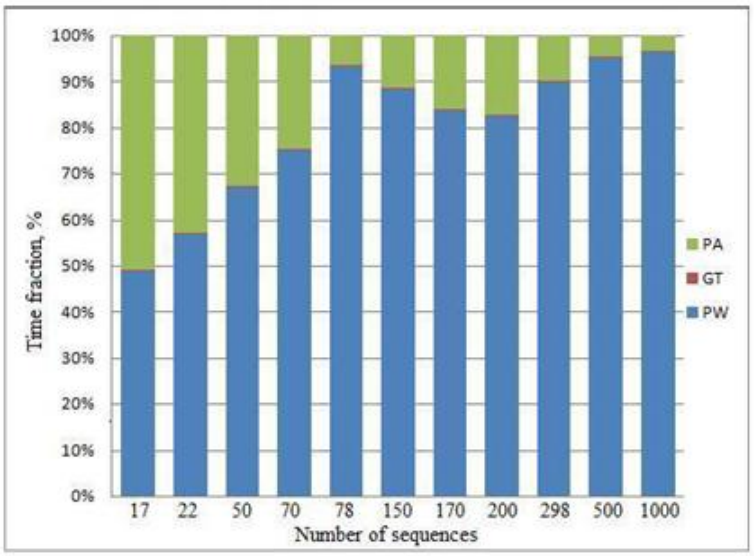

Figure 3.6: Time fraction spent in three ClustalW stages for various input sizes.

The timing results used to plot the graphs are presented in the appendix.

Time profile analysis of the ClustalW, using different number of sequences is depicted in the figure 3.6. It shows the time fraction spent on each of the three stages, that are PA (pairwise alignment), GT (Guide Tree), and the PW(progressive alignment). Graph 3.6 indicates that, when the numbers of sequences is less, then the time required for PA and PW stages is nearly the same. But, as gradually the number of sequences increases, the PA stage dominates in the time fraction percentage. As seen in the above graph when the number of sequences is 17, the PA stage took around 49% of total time, but when the numbers of sequences are 1000 it took nearly 97% of total time. Also it is visible that, the Guide tree takes negligible amount of time as compared to the other two stages.

Graph 3.7 shows that, as the number of sequences goes on increasing, the total time required for computing complete sequence alignment increases exponentially. Small numbers of sequences are computed very fast by ClustalW. But, as we can see when the number of sequences grows beyond 300, it requires a lot of

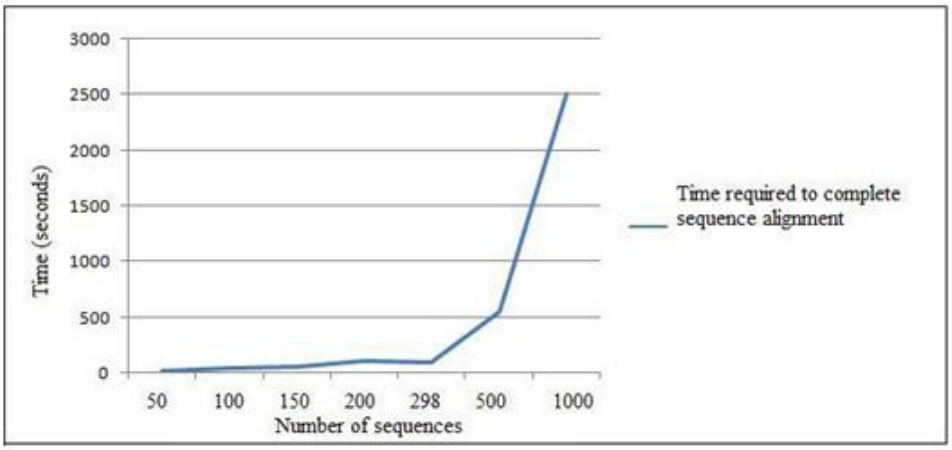

Figure 3.7: Time required for complete sequence alignment with different number of sequences

computation time. Large number of sequences takes enormous amount of time for sequences alignment.

3.4 Discussion

Multiple sequence alignment methods should ideally guarantee to find biologically correct alignment. But in practice this is very difficult to achieve. Even after getting tertiary information, it is difficult to define an optimal alignment between divergent nucleotide or protein sequences. Also the methods for finding an optimal sequence alignment have been impractical to implement, due to the high computational cost.

This chapter shows two main limitations of ClustalW. One of the limitations is that the time taken by the pairwise alignment is too high. Heuristic approaches like parallel implementation on clusters of computers or GPU implementation should be taken to minimize this time required by the first stage. The second limitation is that, as the number of sequences goes on increasing, the

time taken for pairwise alignment increases tremendously and thus resulting in overall increase in the time taken for multiple sequence alignment. As each pairwise alignment is completely independent of all other pairwise alignments we can parallelize this stage. However, parallel implementation of the second and third stages is not that simple due to the data dependency problems. But parallel implementation of just the first stage will considerably reduce the time required for the overall alignment process. Modern graphics processing units (GPUs) can be used for high performance computing as they facilitate enhanced programmability, and also they provide good price/performance ratio. The power of graphics processing units is rapidly increasing. Thus we can conclude that the ClustalW program can be optimized if we use GPUs.

CHAPTER 4

CUDA and GPU Architecture

In this chapter we will explore CUDA and GPU architecture.

4.1 Compute Unified Device Architecture (CUDA)

CUDA is developed by NVIDIA Corporation, to allow direct access to the GPU's graphics hardware. CUDA is a general purpose parallel computing architecture. CUDA comes with a software environment that allows developers to use C as a high-level programming language. CUDA makes use of threads for parallel execution. Unlike CPUs, where they have less threads, GPUs allow thousands of parallel threads to run at the same time. Prior to development of CUDA technology, programming GPGPUs required extensive knowledge of fragment shader languages, it hardly had any debugging tools. A few key CUDA abstractions are listed below,

- Thread: A basic element of execution context [7]. CUDA threads are extremely lightweight as compared to that of the CPU thread. Thus a context change between two threads is not a costly operation.
- Block: A three dimensional collection of threads is called a block. Same block threads share data through fast shared on-chip memory.
- Grid: A two dimensional collection of blocks is called a grid.
- Device: A physical GPU. It is capable of executing several grids simultaneously.
- Kernel: In kernel the code is executed by each thread in parallel. The kernel is written in straight forward scalar C like code.

4.2 CUDA Memory Model

In CUDA there are different memory spaces that have different characteristics and performances:

Type	Read/Write	Scopee	Speed	Size	Location
Register	R/W	Per thread	Very Fast	Very Limited	On-chip
Local	R/W	Per Thread	Fast	Limited	Off-chip
Shared Memory	R/W	Per Block	Fast	Very Limited	On-chip
Global Memory	R/W	Per Grid	Slow	Large	Off-chip
Constant Memory	Read Only	Per Grid	Slow	Limited	Off-chip
Texture Memory	Read Only	Per Grid	Slow	Large	Off-chip

Table 4.1: Properties of the CUDA memory types

Data communication between CUDA main memory and GPU memory has to be explicitly managed by the programmer. This communication overhead can have a significant impact on the overall performance of the application. With the increase in amount of data to be transferred between memories, the time to transfer this data will increase linearly.

The speed, the amount of memory needed, and the operations performed on the data, drives the kind of memory we choose. The overall performance of the applications is highly correlated with the management of memory. Limited size of shared memory proves to be one of the major limitations as it is the only available read-write cache. In CUDA, the programmer needs to copy the data from global memory to the shared memory and from shared memory to global memory.

Figure 4.1 that is given by NVIDIA shows simple arrangement of threads and associated memory. It shows a single thread which resides in a local memory. A group of threads packed in a block is called a thread block which can use shared memory. And a grid, which consists of a number of blocks can share the data from global memory. This memory management is very important for a programmer in order to implement the code on GPU with the help of CUDA. GPUs can run a very high number of threads in parallel. Threads of the same block share data through fast shared on-chip memory. GPUs are specialized for compute intensive, highly parallel computation, thus there is a discrepancy in

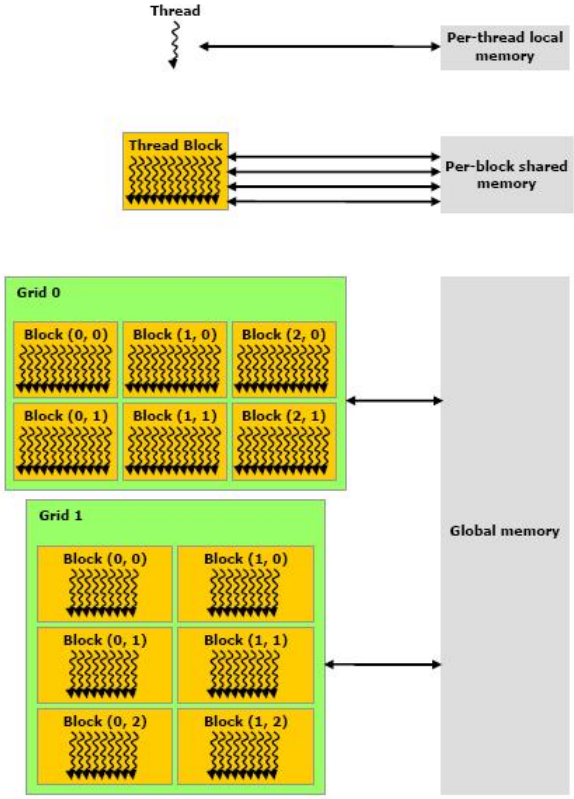

Figure 4.1: Memory Model [24]

floating-point capability between CPU and GPU. GPUs are efficient for a number of problems that can be articulated as data-parallel computations and problems which have high arithmetic intensity (Ratio of arithmetic operation to memory operation).

The threads in a GPU are organised in a structured hierarchical way. In order to achieve good performance it should be in a multiple of 8. Each block is executed on a single multiprocessor. It is advisable to use more number of threads so that the latency of the main memory can be hidden. Within a block the threads are organised in warps, each consisting of 32 threads. All threads in the same warp are executed concurrently. Thread synchronization is completely handled in the hardware.

4.3 GPU Hardware

Figure 4.2 below shows a high-level, simplified view of a GeForce GTX 260GPU. NVIDIA packs 8 texture processing clusters in GTX 260 GPU. As shown in the diagram each cluster is sub-divided into three Streaming multiprocessors (High end GeForce 8-series and 9-series had two Streaming Multiprocessors).

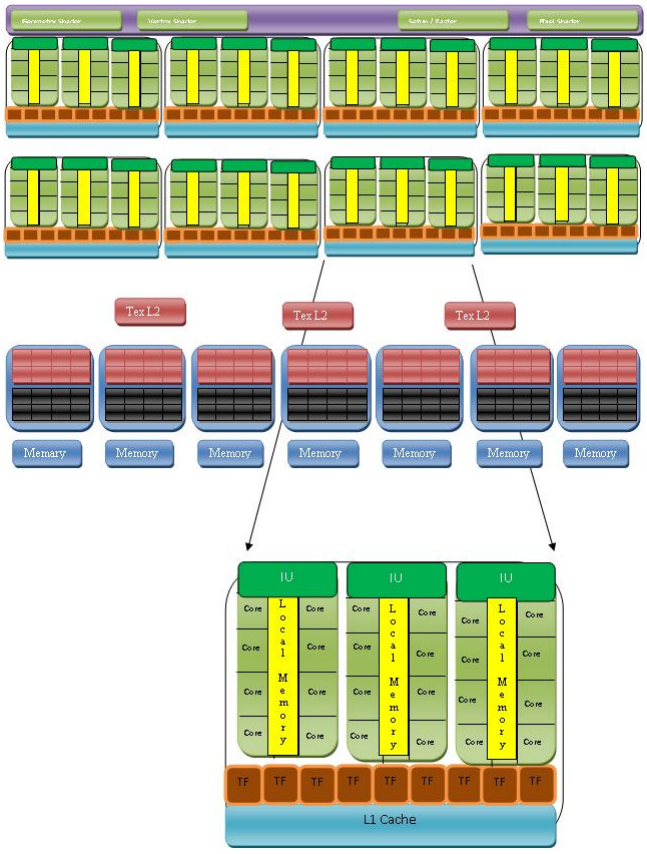

Figure 4.2: (a) Layout of GPU 260 and (b) enlarged layout of texture processing cluster

We can understand the layout of GTX260 GPU with the help of fig 4.2 (a). An enlarged view of texture processing cluster (TPC) is shown in the Fig 4.2 (b). As we can see it has eight texture-filter portions that can filter, between them, eight billion pixels or four 16-bit floating-point pixels per clock cycle. It also has three Streaming Processors.

Each Streaming processor has 8 cores. To maximize the number of processing elements that can be accommodated within the GPU die, these 8 cores operate in Single Instruction Multiple Data (SIMD) fashion under the control of a single instruction sequencer [22]. The threads in a thread block are time sliced onto these 8 cores in group of 32 called wraps. Each warp of 32 threads operates in lockstep and these 32 threads are quad-pumped on the 8 cores[22]. A hardware thread scheduler is used to perform multithreading. One by one each time next wraps are executed. Hardware masking is used in order to manage divergent threads. Different warps in a thread block need not operate in lockstep, but if threads within a warp follow divergent paths, only threads on the same path can be executed simultaneously[22]. In the worst case, if all 32 threads in a warp follow different paths without reconverging (effectively resulting in a sequential execution of the threads across the warp) a 32X penalty will be incurred. With the help of SIMD divergence, programme can be optimized; however it is not necessary to know SIMD in deep.

The number of cores in a GeForce GTX 260, are Number of Multiprocessors $\times$ Number of Stream processors per Multiprocessor $\times$ Number of cores per stream processor i.e. $8 \times 3 \times 8 = 192$ Cores. The main difference between the GPU and CPU is the area on chip dedicated to memory processing and processing functions. For example an Intel Core 2 Quad Q9650 has only 4 processing cores as compared to 192 of GTX 260[4]. More processing cores results in a higher theoretical computation rate. For on-chip memory, the Q9650 has 12Mb of Level 2 cache [6] whereas the GTX 260 has only 1MB of total on-chip memory [3] which includes registers , shared memory, texture cache, constant cache. The result of this is a large difference in the amount of on-chip memory per processor for each processor. Q9650 achieves 3MB per processor whereas GTX 260 has around 8KB per processor. Thus to hide the pipeline stall, a lot of parallel threads should be run concurrently. In general, as the GPUs have a larger number of processors on board, they are able to compute much faster than CPUs because instructions can be executed in parallel.

The large increase in processor count does not yield an equally sizeable increase in power consumption. At full load, the Q9650 consumes 95W [6] compared to the GTX 260 with 182W[5]. The GTX 260 is capable of processing at 715.392 GFLOPs [5], while the Q9650 has only been measured to achieve 49.3 GFLOPs

[6]. As compared to power consumption, the processing speedup achieved is very high in GTX 260.

CHAPTER 5

Methodology

This chapter briefly describes the approach for implementing Multiple Sequence Alignment on GPUs and CPU.

5.1 CPU Implementation

ClustalW is an open source program. It is a very fine tuned implementation of MSA. In the CPU implementation we are using ClustalW-2.0.11 that serially executes on CPU. The pairwise alignment is the first stage of MSA. In this stage, the CPU implementation takes a number of iterations to calculate the score matrix for each pair of sequence which is simply the product of length of sequence1 and sequence2. In the second stage of MSA, neighbor joining tree is implemented. The progressive alignment is the final stage of MSA. In this stage, the individual score matrixes based on the guide tree are computed. The calculation of this score matrix requires the same number of iterations as computed in the pairwise alignment. The individual score matrix is further used for the traceback phase to align the sequences.

5.2 GPU Implementation

An efficient implementation of any algorithm on GPU requires in-depth understanding of the device architecture and its constraints. As discussed in chapter 3, MSA has three steps: pairwise alignment, building guide tree and progressive alignment. Figure 3.6 depicts that the pairwise alignment is the most computationally expensive stage in ClustalW for both large and small number of sequences. On the other hand, progressive alignment is computationally expensive only for a small number of sequences. Thus we parallelize and implement these two stages in GPU in order to reduce the time required for matrix calculation.

5.2.1 Pairwise Alignment in CUDA

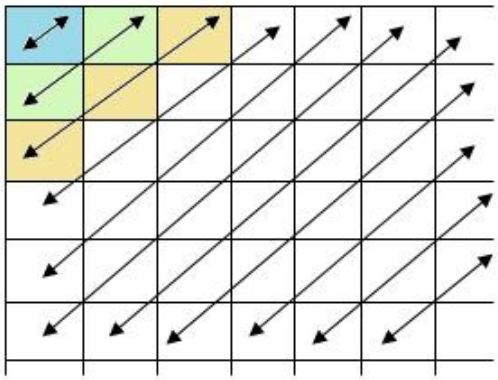

Figure 5.1: Matrix Dependency [9]

As the pairwise alignment requires computation of the score matrix for every pair of sequences, it proves to be the costliest step of all. Each cell in the matrix depends upon its neighboring three cells which leads to data dependency between them. In order for parallel implementation of this algorithm in GPU, we explored different ways in finding data dependency between the cells. However, careful observation of the calculation matrix revealed that the anti-diagonals of the matrix are dependent on the previous anti-diagonal as shown in figure 5.1. Thus we can calculate the matrix on an anti-diagonal by anti-diagonal basis. The key point in implementing pairwise alignment in this fashion is that any anti-diagonal depends upon only the previous anti-diagonal.

The number of threads per block and the number of blocks per grid are to be declared in advance before launching any CUDA kernel. A large number of parallel threads have to be launched simultaneously to fully exploit the immense computational power of the GPU. Thus configuring the blocks and grids is a crucial step. In this implementation, the number of threads in a block is set to 16×16. Furthermore, the number of blocks in a grid is dependent on the length of sequence and the number of threads per block.

```
NUMBER OF THREAD = 16
NUMBER OF BLOCKS = LENGTH OF SEQUENCE / NUMBER OF THREADS

If the NUMBER OF THREADS is not a multiple of LENGTH OF SEQUENCE then we increase the
NUMBER OF BLOCKS BY 1.
```

Figure 5.2: Pseudo code for initialization, calculating number of threads and blocks.

```
dim3 threads(NUMBER OF THREADS,NUMBER OF THREADS);
dim3 blocks(NUMBER OF BLOCKS, NUMBER OF BOCKS);
```

Figure 5.3: Pseudo code to initialize threads and blocks.

Figure 5.4 illustrates the border cells for any alignment. These cells always appear in the same fashion. As we are implementing the matrix calculations on GPU, in order to achieve this form of matrix initialization we write a CUDA kernel.

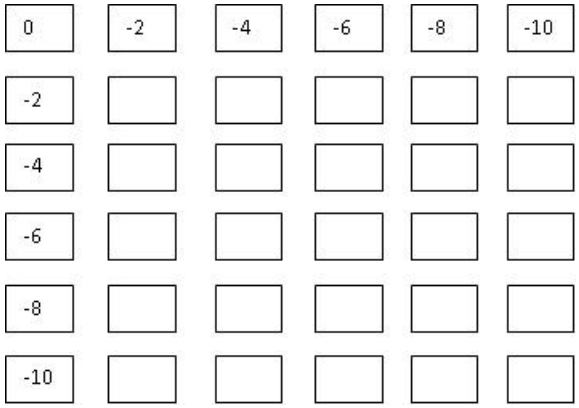

Figure 5.4: Initialized Matrix

intialiseMatrix kernel:

Thread x;
Thread y;
Thread Index = x * LENGTH_OF_SEQUENCES + y;

Step 1: check if x = 0. If yes then Matrix[Index] = x *-2;
Step2: check if y = 0. If yes then Matrix[Index] = y*-2;
Step3: if it is not in step1 and step2 then Matrix[Index] = 0;

X is used to initialize the first column with multiples of -2. And Y is used to initialize the first row with the multiples of -2.

Figure 5.5: pseudo code for matrix Initialization .

A GPU kernel can only access the memory which resides on the device. In order to perform the operation we transfer the data to device and then call the GPU kernel for matrix initialization. After performing this operation the data is retrieved by the CPU function. Thus, while performing the matrix calculation for each pair of sequences it needs to copy the sequences to the device which generates a large volume of data transfer during each kernel call. In order to reduce this large amount of data transfer the sequences are copied only once to the device memory, so that it can be reused without any need to transfer it each time. Here, we have used device memory for storing the sequences and the DistanceMatrix while performing the calculations.

Kernel call:
calculateMatrix <<<blocks, threads>>> (LENGTH OF SEQUENCES, LOOP NUMBER)

The pseudo code for calculateMatrix is as follows,

Let us consider "BlockX" and "BlockY" as the threads for block X-index and Block Y-index respectively and "INDEX" as the index thread used across the matrix. TOP_LEFT, TOP, LEFT and MXIMUM are the four variables where we store our values.

Step 1: Check if the sum of BlockX and BlockY is same as that of the LOOP NUMBER. The LOOP NUMBER is twice as much as the block size calculated above.

Step2: Check if the particular characters of two sequences are same or not. If we consider Sequence1 and Sequence2 as two sequences and LENGTH OF SEQUENCES then we can check if the particular characters pointed by the thread index are same or not by following statement,
This will give the exact character of the first sequence: Sequence1[INDEX divide by SEQUENCE LENGTH]
This will give the exact character of the second sequence: Sequence2[INDEX mod SEQUENCE LENGTH]

Step 3: Both the characters obtained in the above step are same or not. If they are same then assign MATRIX [Top left of the current location] +1 to TOP_LEFT

Step4: Assign MATRIX [left of current position] -2 to LEFT
Step5: Assign MATRIX [top of current position] -2 to TOP

Step6: Find which is maximum among LEFT, TOP and TOP_LEFT and assign this value to MAXIMUM.

Step7: Copy this MAXIMUM to the MATRIX [INDEX]

Figure 5.6: Pseudo code for calculating matrix.

In the pairwise alignment we only require the score of two sequences which is the last entry of the distance matrix. So, instead of transferring the whole distance matrix back to CPU it only returns the scores of two sequences. This approach considerably reduces the memory transfer time. By calculating the score of each pair of sequences, a matrix of scores of each sequence is formed against every other sequence. It will require N $\times$ N time calculation of pairwise alignment. This is illustrated in figure 5.7.

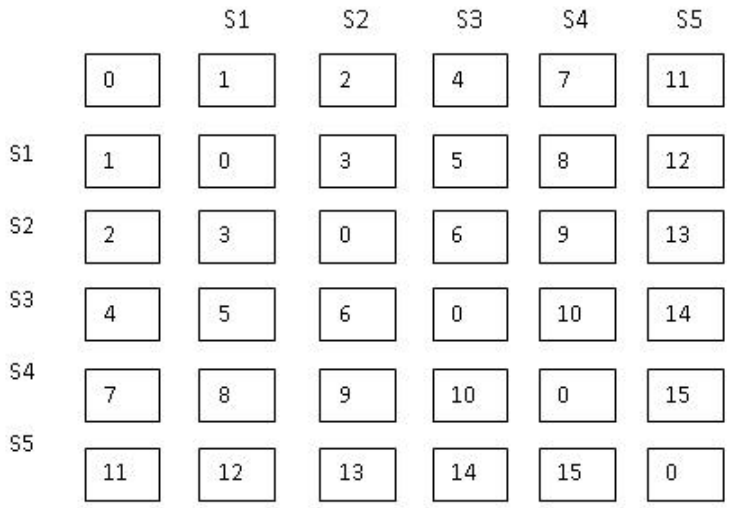

Figure 5.7: Symetric Matrix

As the matrix in figure 5.7 is diagonally symmetric, only the lower half of the diagonal can be maintained in order to avoid the redundancy. Thus in order to calculate a matrix for N sequences, it will take (N X (N-1)) / 2 pairwise alignments. Using this approach, the pairwise alignment can be calculated in half of the time.

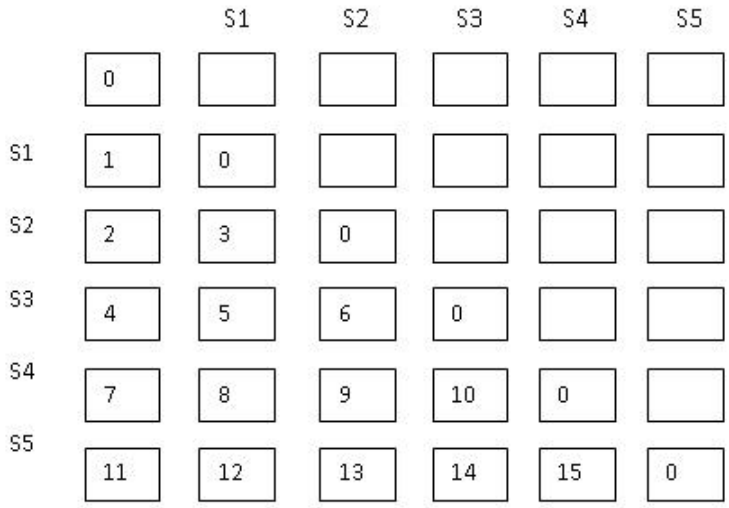

Figure 5.8: Half Symmetric Matrix

5.2.2 Constructing Guide tree using C language

In this project, we are implementing the Neighbor-Joining method for construction of the Guide Tree. It is a complex but a less expensive task as compared to other stages of MSA. Furthermore, it is hard to parallelize the construction of the guide tree. Thus, we implement it on CPU using C language. Figure 5.9 shows the steps for constructing the guide tree.

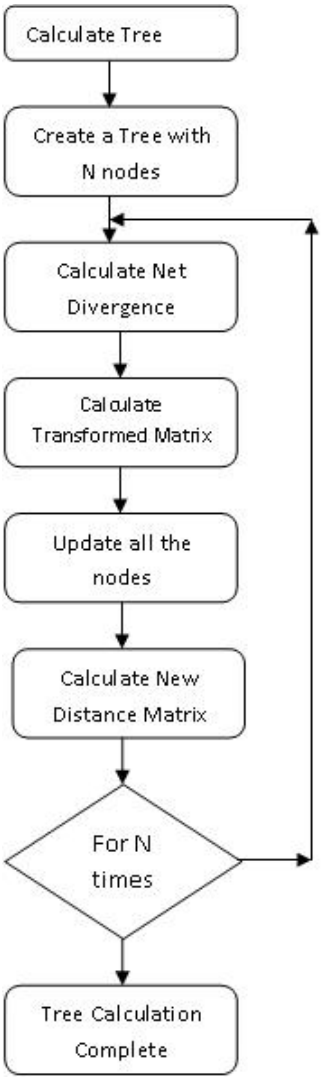

Figure 5.9: Flow Chart of steps followed while constructing Guide Tree

5.2.3 Progressive alignment in CUDA

Progressive alignment consists of calculating score matrix for the sequences aligned in the guide tree and then implementing traceback in order to align the sequences. However, storing the matrix for backtracking, significantly increases the number of arithmetical and memory operations. The following section describes the two steps for calculating progressive alignment.

```
CUDA_SAFE_CALL (cudaMemcpy(host_matrix,dev_matrix,sizeof(int)*Length-of-
Sequence1 x Length-of-Sequence2,cudaMemcpyDeviceToHost));
```

Figure 5.10: Memory transfer statement

5.2.4 Calculating score matrix

Calculating the score matrix in progressive alignment stage is very similar to calculating the score matrix in pairwise alignment. The only difference is, in pairwise alignment we have to calculate the score matrix of every sequence with every other sequence, whereas in progressive alignment we calculate the score matrix of the sequences that are aligned in the guide tree. This significantly reduces the number of matrix calculations required. However, in pairwise alignment we only transfer the score of matrix from the device to CPU, whereas, in progressive alignment we have to transfer the complete set of matrices. This matrix is sent to the traceback phase for alignment of sequences.

5.2.5 Trace back using C language

In trace back, the sequences are aligned depending upon their score matrix and the other sequences. Implementing it on GPU is expensive as it will run linearly on GPU due to the data dependency. Thus this stage is implemented on CPU.

Let us consider Q1 and R1 as the final aligned sequences. Let q1index, and r1index be the variables that keep track of the lengths of the final sequences.

(a) Initialize q1index and r1index to 0; i and j to qlength-1, and dlength-1.

(b) While i and j are greater than 0 do

i. Calculate the match score for the position.

ii. If Score[idx] equals Score[idx-qlength-1] + match score, then

Copy Q[i] to Q1[q1index], and R[j] to R1[r1index]

Increment q1index and r1index by 1.

Decrement i and j by 1.

iii. Else if Score[idx] equals Score[idx-length] + gap penalty,

then

Copy Q[i] to Q1[q1index], and insert a gap (-) in R1[r1index]

Increment q1index and r1index by 1.

Decrement i by 1.

iv. Else if Score[idx] equals Score[idx-1] + gap penalty, then

Insert a gap (-) in Q1[q1index], and copy R[j] to R1[r1index]

Increment q1index and r1index by 1.

Decrement j by 1.

(c) While i is greater than 0,

Copy Q[i] to Q1[q1index], and insert a gap (-) in R1[r1index]

Increment q1index and r1index by 1.

Decrement i by 1.

(d) While j is greater than 0,

Copy R[j] to R1[r1index] and insert a gap (-) in Q1[q1index]

Increment q1index and r1index by 1.

Decrement j by 1.

(e) Reverse both the strings Q1, and R1.

Figure 5.11: Pseudo code for traceback [7]

CHAPTER 6

Experimental results and analysis

In this chapter, the results that are obtained by timing ClustalW on CPU and our GPU implementation are analyzed. Comprehensive tests have been performed to compare the implementation of MSA in CUDA with ClustalW. Furthermore, the effect of altering the code to suit the hardware by using different memory options available has been analyzed.

6.1 Environmental Details for CPU and GPU

- Processor Intel Core 2 Duo CPU, T5850 @2.16GHz @2.17GHz
- RAM 2.00 GB
- System type 32-bit UBUNTU Hardy Heron-Version 8.04 Operating System.
- Programming language C-99 (having gcc compiler).

The GPU we used for performance measurements is the GTX 260, in which the Streaming Processors (SPs) are clocked at 1.24 GHz. The GPU has 896 MB of device memory. The GTX 260 contains 24 streaming multiprocessors (SMs). Each multiprocessor comprises of 8 SPs. Thus the total numbers of SPs are 192. A 16kb of fast per-block shared memory is shared among each group of 8 SPs. And each group of three SMs, that is the thread processing cluster, shares a texture unit. Single Instruction Multiple Data (SIMD) is used for execution of instructions in all SPs which contains a scalar ALU. SPs can also perform floating point operations.

6.2 Performance analysis of CPU version

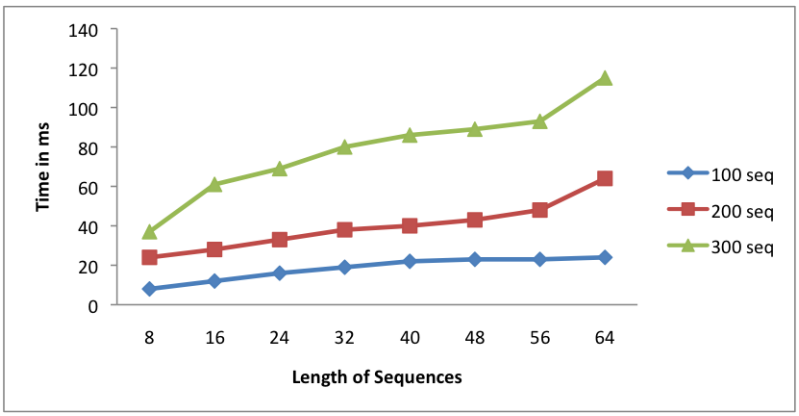

Figure 6.1: CPU performance for different inputs

Graph 6.1 illustrates the time required for sequences of length in multiple of 8. The graph shows an upward trend with increase in length and number of sequences. The CPU version implemented is ClustalW.

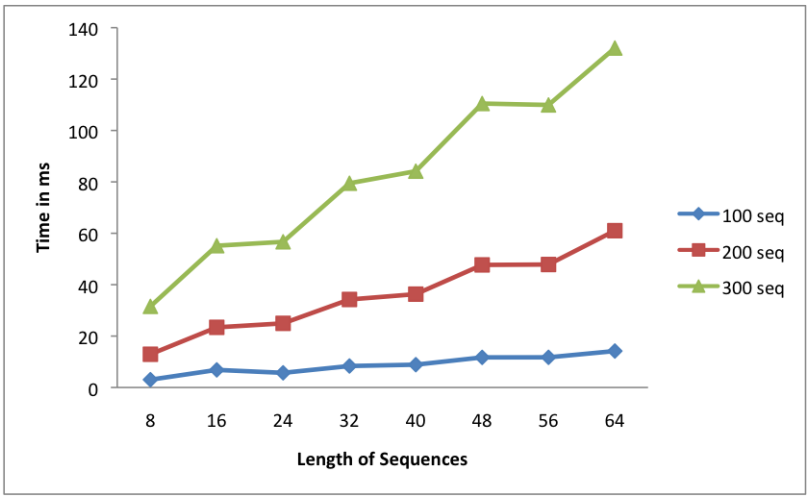

Figure 6.2: GPU performance for different inputs

Graph 6.2 shows the performance of GPU for calculating MSA. The graph shows an upward trend with increasing length of sequences. This clearly shows that as the length of the sequences goes on increasing, the time it takes to compute MSA grows.

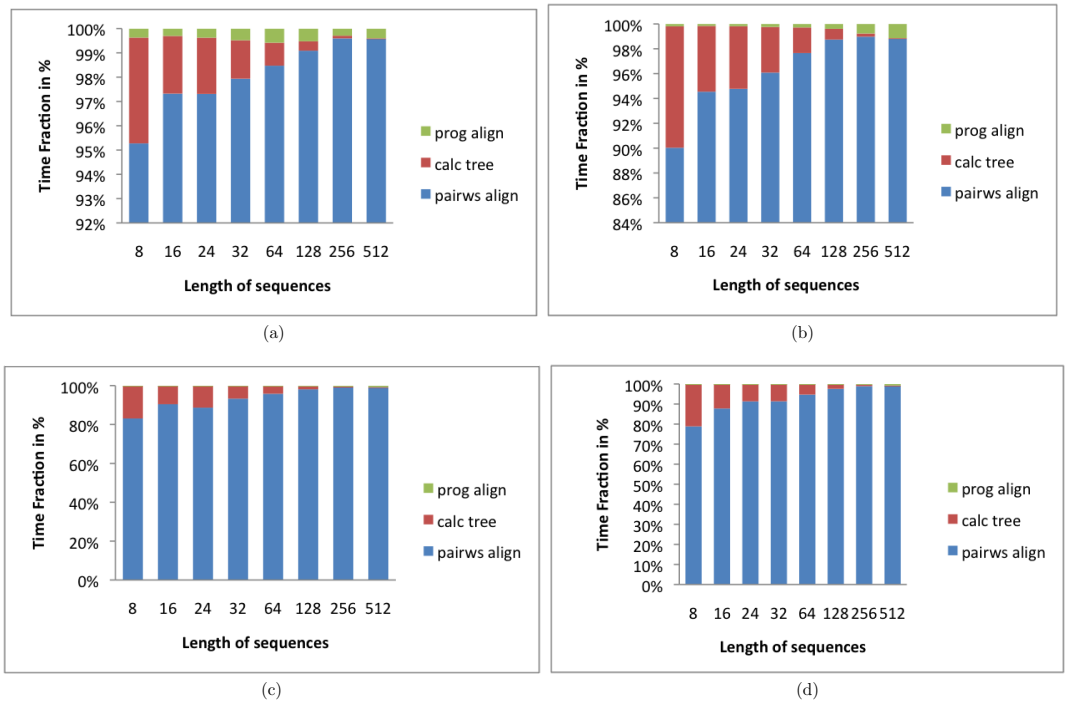

Figure 6.3: Comparison of different Phases of MSA. (a) 100 sequences (b) 200 sequences (c) 300 sequences (d) 400 sequences

Graph 6.3 depicts the time taken by the three phases of MSA for different inputs. These graphs show a particular trend where the calculation of tree for very small sequences takes majority of the time. However, as the length of sequences goes on increasing, the pairwise alignment stage dominates the overall MSA and also the calculation of guide tree takes less time.

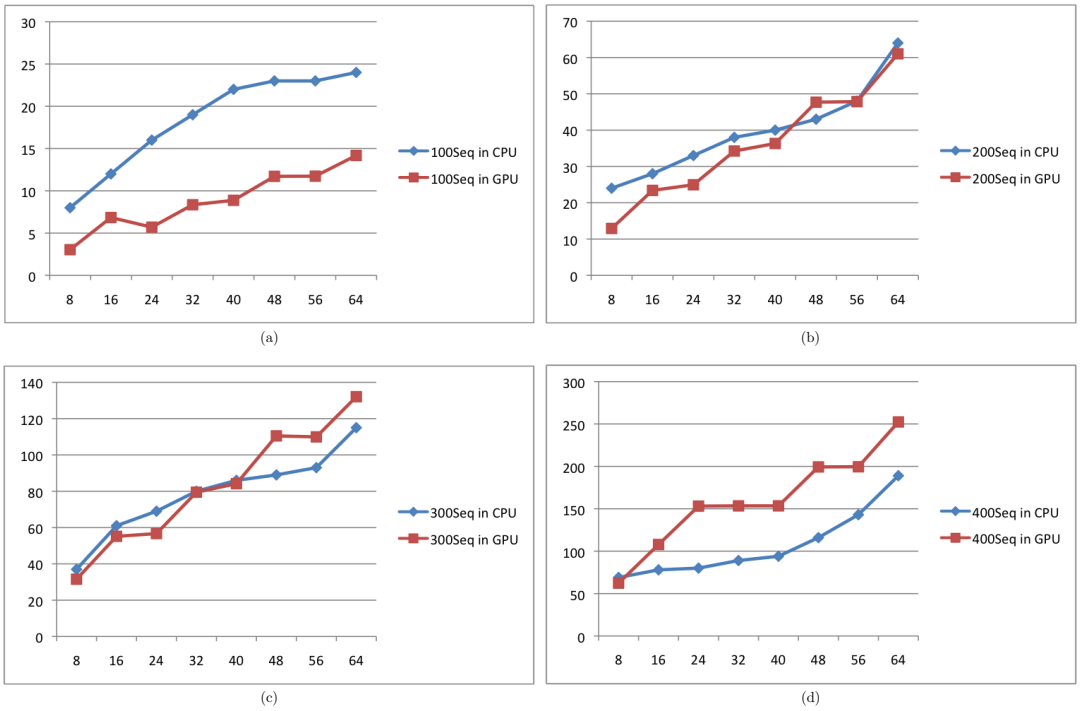

Figure 6.4: GPU vs CPU for (a) 100 sequences (b) 200 sequences (c) 300 sequences (d) 400 sequences

Graph 6.4 illustrates the performance comparison between GPU and CPU for same input. As seen in this graph, for input of 100 sequences, GPU performs better than CPU for all the sequence lengths. However, as the number of sequences goes on increasing, the performance of CPU is better than that of GPU. Further, when the input is 300 sequences, the CPU and GPU is on par with each other for small lengths of sequences. However, as the sequence length goes on increasing the CPU outperforms the GPU. This limited speedup is due to the fact that this application is not basically computationally intensive.

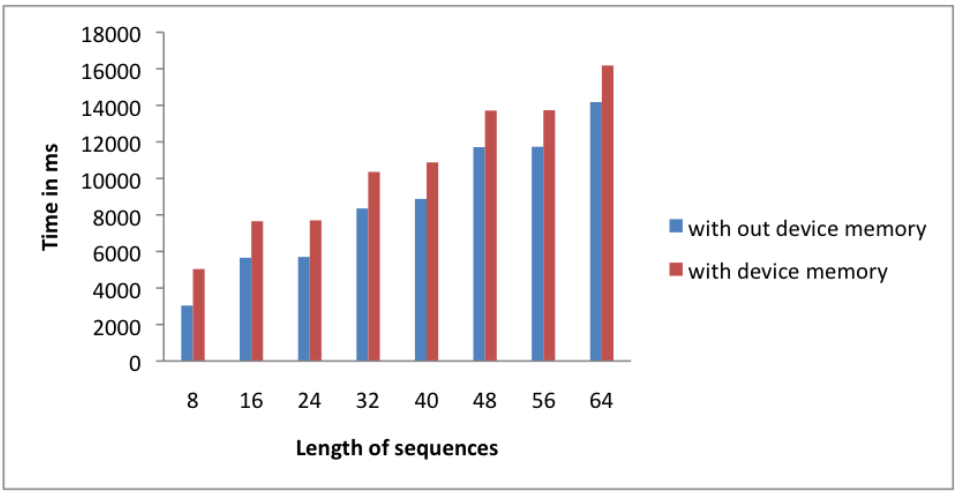

Figure 6.5: GPU without device memory v/s with device memory.

Graph 6.5 shows the speedup achieved when the device memory is used to store the sequences and the matrix, instead of passing it each time in the kernel call. The difference between the two approaches clearly shows that the memory transfer overhead of sending the sequences and matrix during each kernel call is significant. Initially, the data is copied to the GPU and then the kernel is called in order to reduce the memory transfer overhead. The multi-processor occupancy of GPU for the calculateMatrix kernel has 256 threads per block and it uses 14 registers per thread. It uses 28bytes of shared memory per block. Thus with the help of CUDA GPU occupancy calculator the following data was collected,
Active Threads per Multiprocessor:1024
Active Warps per Multiprocessor:32
Active Thread Blocks per Multiprocessor:4
Occupancy of each Multiprocessor:100%

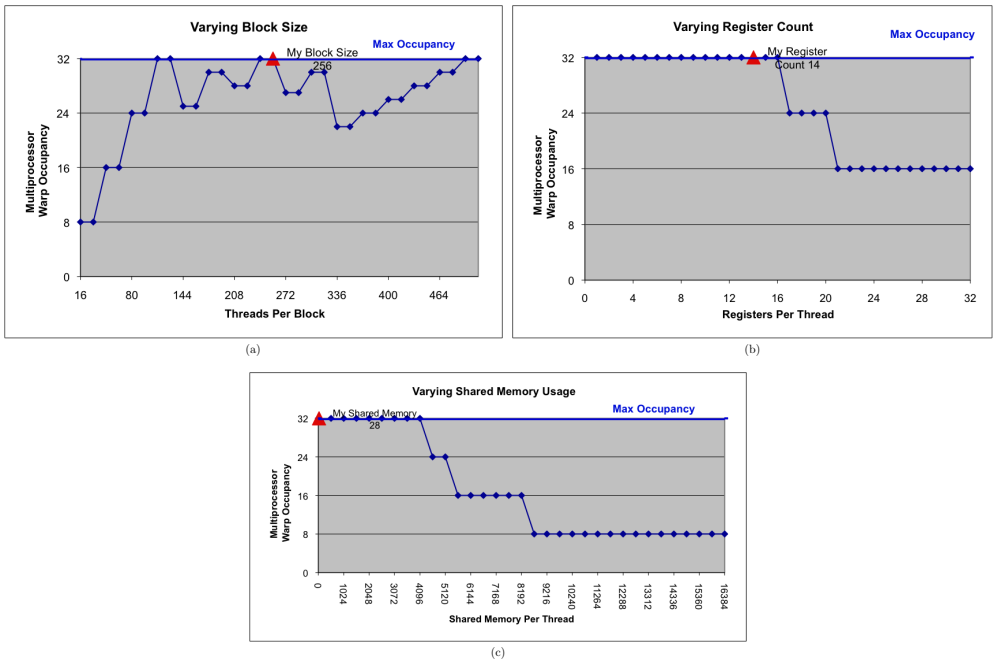

Figure 6.6: (a) varying block size , (b) Varying Register count, (c) Varying Shared memory usage.

Graph 6.6 illustrates the occupancy of threads, registers and shared memory for the configuration used for implementing the calculateMatrix kernel. It is clearly visible from the above graphs that the GPU has maximum occupancy for threads per block, registers per thread and shared memory per thread. However, achieving higher occupancy always does not yield higher performance. In order to accomplish high performance the kernel should be bandwidth bound. There can be many cases where increasing the occupancy may either have no or adverse effects. If a multiprocessor in GPU is already running at least one thread block per multiprocessor, and it is bottlenecked by computation and not by global memory accesses, then increasing occupancy may have no effect [1]. Increasing the occupancy may give rise to various problems such as additional instructions, spills to local memory (which is off chip), divergent branches, etc. For bandwidth bound applications, increasing occupancy can help better hide the latency of memory accesses, and therefore improve performance [1]. Thus the optimal way to measure the performance of any application is to time it and experiment it with different configuration and input.

6.3 Discussion

While developing and implementing this parallel version of ClustalW, we have gained some insights into GPU architecture and the CUDA programming model in particular:

- Along with the programming model of CUDA, some inherent properties of the architecture are also important in order to extract sound performance from GPU. The most significant characteristic is the deep focus on throughput at the cost of individual thread performance. This solicits the programmer to exploit large quantities of fine-grained parallelism. Other significant aspects are SIMD fashion execution of threads, wrap concept, use of local memories. All this requires a deep focus on data parallelism.

- Control flow instructions such as if, if-else, switch can have a noteworthy impact on performance.

- Efficient mappings of data structures to CUDA's domain based model are necessary. For example, matrix like data structures are quite straightforward to implement in GPU. However, for applications using directed graph traversal, tree traversal and other complex data structures, it is quite hard to make use of GPUs, without redesigning the algorithms.

- The location of data and its memory access patterns signified by the implementation of the algorithms holds a key in order to take benefit of GPU memory hierarchy. For example for data which is accessed frequently, read-only memory, constant memory and shared memory can prove to be beneficial depending upon the size of data. Exorbitant global memory access is undesirable as it consumes a lot of bandwidth and also it has high latency. Thus most of the time appropriate use of shared memory is profitable.

- As shared memory is non-persistent it leads to less efficient communication between the kernels. For example, a kernel has to store the data from shared memory to device memory and then the other kernel has to fetch it from the device memory in order to use it.

CHAPTER 7

Conclusion and future work

7.1 Conclusion

The inadequate programming model of OpenGL was ill-suited for efficient general-purpose computing. This was one of the main hindrance for exploiting immense computational power of GPUs. However, the introduction of new architectures like CUDA, CTM has opened doors to achieve high performance for general purpose computing from GPUs. A number of algorithms in bio-informatics can be accelerated by using GPUs.

While implementing ClustalW, we performed the entire pairwise alignment stage in parallel in GPU. In order to parallelize the pairwise stage, deep knowledge of the algorithm and understanding of the data dependencies was necessary. This stage requires calculation of N X (N-1)/2 matrixes, where N is the number of sequences. In CPU, for calculating the matrix, it performs q1 X q2 iterations, where q1 is the length of sequence one and q2 is the length of sequence 2. In our GPU implementation, the numbers of iterations were drastically reduced. In GPU the number of iterations depends on number of blocks rather than the direct length of sequences. The observations from the GPU implementation of the algorithm illustrate better performance by GPU for smaller sequences as compared to that of large number of bigger sequences.

The results obtained were as follows,

- GPU speedup was 9 X as that of the CPU for smaller sequences.
- CPU outperformed the GPU implementation for higher length of sequences.

7.2 Future Work

Future Work: This application would benefit from further optimization. Some refinement in the algorithm implemented would be beneficial to overcome the

limitation of taking large amount of time for larger sequences. One of the probable solutions for this is to use shared memory. Shared memory is very fast form of memory as compared to global memory. It can be as proficient as registers if a proper access pattern is used. But the amount of shared memory available is limited thus it would require extensive swapping of chunks of memory. Thus in order to utilize the shared memory we need to copy the essential data from global memory to shared memory. Then perform as many operations as possible using shared memory. Then write back the results to global memory.

We still have not performed any extensive performance tuning, such as working to reduce bank conflicts, and our CUDA version is wasting some of resources.

Currently, the guide tree step is implemented in CPU. However, for larger number of sequences it has to be seen if its implementation in GPU can be beneficial or not.

APPENDIX A

Research Proposal

Title: Use of Graphical processing units to accelerate multiple sequence alignment.

Author: Chetan Khaladkar

Supervisor: Prof. Amitava Datta

A.1 Background

In molecular Biology one of the fundamental tasks carried out is searching similarities in protein and DNA databases. Sequence alignment is the process of comparing the sequences of DNA, RNA or protein in order to and the similarities between two or more sequences. Multiple sequence alignment (MSA) is a sequence alignment of usually more than two such biological sequences. Sequence alignment is used for many applications. One of the main purposes of sequence alignment is to and the homologous characteristic between the sequences.

Aligning two or more biological sequences can be difficult as well as time consuming due to the length of biological sequences [1]. Pair wise alignment is used to match two query sequences. As MSA is more complex so a pair wise alignment is not useful in that case. Multi-Dimensional programming can be used for MSA. But that is time consuming for calculating MSAs as the time required is exponential in the number of sequences. Thus many heuristics are used to compute MSAs so that MSAs can be calculated in a reasonable time[7].Progressive alignment is one of the widely used heuristic. Clustal W[5], RALINE[8], MUSCLE[9] and T-Coffee[10] are few popular tools which use the progressive alignment technique.

A.2 Limitations

Dynamic programming generally uses a gap penalty and substitution matrix for alignment of amino acids or nucleotides. A simple method requires construction of an N dimensional matrix if there are n individual sequences. Thus the computationalspace required increases exponentially with the increase in N. Thus it consumes lot of time for multiple sequence alignment. One of the alternatives can be parallel processing on different computers but it also does not gain much speed as compared to the cost incurred.

The biological databases are growing at a rapid rate [2], as compared to the improvement in micro-processors. Thus in the past research have implemented Multiple Sequence alignment on different hardware platforms like playstations to increase the computation speed. But compared to other hardware devices GPU (Graphics processing unit) are relatively cheap and also they have high performance [1,3]. With the help of CUDA (Compute Unified device Architecture) we can read as well as write on device memory. Also memory-intensive application can run significantly faster on low-cost-GPU as compared to CPU.[3]

A.3 Aim

The aim of this project is to use the high speed and computational power of GPU to create high speed/performance solutions for implementing sequence alignment at a relatively cheaper cost. We will implement the progressive alignment algorithm as used in Clustal W. Then we will parallelize this process in order to improve the speed.

A.4 Method

The Algorithm is implemented in 3 parts,

- In the first part a distance matrix is calculated. Using dynamic programming algorithm of Needleman and Wunsch[4] , pair wise alignment is performed. This is the most dominant steep among all three steps.
- In the second part a guide tree is constructed with the help of the distance matrix. With the help of the distance matrix calculated in the

  rst part a phylogenetic or guide tree is constructed.

- In last part progressive alignment is performed. In this step a group of sequences which are closely related are aligned to get the MSAs.

And all this is implemented on GPU with the help of C and CUDA languages.

A.5 Software and Hardware Requirements

We will be using C and CUDA for programming on GPU. For documentation and report writing work, LATEX will be used. Also a standard PC hardware with a GeForce 8800GTX will be required for the project.

APPENDIX B

Alignment output

B.1 Input Sequence:

seqNum0 This is random sequence 0
DBCADABCEBCCAEDBABCBBDCECACDCAECCDECDBBCEDBEE
seqNum1 This is random sequence 1
CDBDDEDBEECABDECBBEEAAEDCBCCCCCBBABBAEEEAEBCC
seqNum2 This is random sequence 2
EDEDBADAEEEECBDBCAEBCBCDACEBDAECADDDDBDEBEDDD
seqNum3 This is random sequence 3
CDEDBDACBAABBADAABEBCADBCDECCEABCEEADBDBBADEA
seqNum4 This is random sequence 4
BBDCBAEEBEEBACADCAEDAECEBBAECCCAAACEACDDDAEAE

B.2 Output Sequence

seqNum1
CDBDDEDBEECABDECBBEEAAEDCBCCC-CCBBABB-AEEEAEBCC—-
seqNum2
—EDEDBADAEEEECB-DBCAEBCBCDA-CEBDAEC-ADDDDBDEBEDDD
seqNum0
DBCADABCEBCCAEDB–ABCBBDCECACDCAECCDE-CDBBCEDB–EE-
seqNum3
–CDED-BDACBAABB–ADAABEB-CADBCDECCEABCEEADBDBBADEA
seqNum4
–BBDCBAEEBEEBAC–ADCAEDAECEBBAECCCAAACEACDDDAEAE–

APPENDIX C

Timing results

Sequences	Time required for PW (milliseconds)	Time required for Gt (milliseconds)	Time required for PA (milliseconds)	Total time required (milliseconds)	No of sequences
1	722.90	0.811	749.61	1,473.324	17
2	1475.91	1.319	1109.41	2,586.639	22
3	12260.00	3.429	5949.62	18,213.049	50
4	19154.70	6.880	6313.10	25,474.680	70
5	39375.30	8.944	2663.22	42,047.464	78
6	56508.40	43.688	7237.14	63,789.228	150
7	65052.30	48.110	12357.10	77,457.510	170
8	91976.10	71.795	19148.40	1,11,196.295	200
9	91732.40	198.057	9980.86	1,01,911.317	298
10	526816.00	767.826	26419.50	5,54,003.326	500
11	2.42E+06	4863.330	85991.80	2.51E+06	1000

Table C.1: Time analyzed during all 3 phases of Clustal W

Sequences	Time required for PW (milliseconds)	Time required for Gt (milliseconds)	Time required for PA (milliseconds)	Total time required (milliseconds)	No of sequences
1	12260.00	3.429	5949.62	18,213.049	50
2	30868.50	14.423	17075.00	47,957.923	100
3	56508.40	43.688	7237.14	63,789.228	150
4	91976.10	71.795	19148.40	1,11,196.295	200
5	91732.40	198.057	9980.86	1,01,911.317	298
6	526816.00	767.826	26419.50	5,54,003.326	500
7	2.42E+06	4863.330	85991.80	2.51E+06	1000

Table C.2: Time analyzed during all 3 phases of Clustal W

Number of Sequences	With out device memory	With device memory
8	3037.9	5037.9
16	5655.29	7655.29
24	5702.89	7702.89
32	8356.26	10356.26
40	8877.26	10877.26
48	11714.89	13714.89
56	11734.16	13734.16
64	14181.74	16181.74
128	34800.36	36800.36
256	121788.19	123788.19
512	589624.16	591624.16

Table C.3: CPU for different input lines

Bibliography

[1] Justin Ebedes, and Amitava Datta. Multiple sequence alignment in parallel on a workstation cluster. *Bioinformatics* (2004), 20(7):1193-5.

[2] S.B. Needleman and C. D. Wunsch. A general method applicable to search for similarities in the amino acid sequences of the two proteins. In *J. Mol. Biol* (1970),48, 443-453.

[3] Svetlin A Manavski and Giorgio Valle. CUDA compatible GPU cards as efficient hardware accelerators for Smith-Waterman sequence alignment. *From Italian Society of Bioinformatics (BITS) : Annual Meeting 2007 Naples, Italy* (April 2007), 26-28.

[4] Ryoo, S., Rodrigues, C. I., Baghsorkhi, S. S., Stone, S. S., Kirk, D. B., and mei W. Hwu, W.. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. *ACM SIGPLAN Symposium on Principles and practice of parallel programming, USA/* (2008), 73-82.

[5] Fedy Abi-Chahla, Florian Charpentier. Nvidia GeForce GTX 260/280 Review `http://www.tomshardware.com/reviews/nvidia-gtx-280,1953.html`, (June, 2008)

[6] Intel. Intel R coreTM2 quad processor - specifications `http://www.intel.com/products/processor/core2quad/specifications.htm`, (Oct, 2008)

[7] Diviz PS. Sequence Alignment Using Graphics Processing Units. *The University of Western Australia. School of Computer Scince* (2008).

[8] T.F. Smith and M.S.Waterman. Identification of Common Molecular Sub-sequences *J. Mol. Biol.* (1981), 147,195-197.

[9] David W. Mount. Using Gaps and Gap penalties to Optimize Pair wise Se-quence Alignments Cold Spring Harb. Protoc. In *doi:10.1101/pdb.ip59* (2008).

[10] Jonathan Light. A brief history of Sequence Alignment Methods. *MB & B 452a.*

[11] J.D. Thompson, D.G. Higgins and T.J. Gibson. CLUSTAL W: improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-specific gap penalties and weight matrix choice *Nucleic Acids Res.*,(1994), 4672-4680 .

[12] Casey. RM BLAST Sequences Aid in Genomics and Proteomics In *Business Intelligence Network* (2005).

[13] Weiguo Liu, Bertil Schmidt, Gerrit Voss, and Wolfgang Muller-Wittig. Gpu clustal w: Using graphics hardware to accelerate multiple sequence alignment. In *School of computer Engineering, Nayang Technological University*.

[14] G. Giupponi, M. J. Harvey, G. De Fabritiis. The impact of accelerator processors for high-throughput molecular modeling and simulation In *Drug Discovery Today* (2008), 23-24.

[15] Azzedine Boukerche , Alba Cristina , Mauricio , Thomas. Parallel Strategies for Local Biological Sequence Alignment in a Cluster of Workstations In *Journal of Parallel and Distributed Computing*, 12 (February 2007), 170-185.

[16] W.S. Martins, J.B. Del Cuvillo, F.J. Useche, K.B. multithreaded parallel implementation of dynamic programming algorithm for sequence comparison. In *Department of Electrical and Computer Engineering University of Delaware, Newark* (DE 19716, USA.).

[17] W. Fitch. Towards defining the course of evolution: minimum change for a specific tree topology In *Systematic Zoology* (1971), 20.406-416.

[18] Dayhoff, M. O. Computer analysis of protein evolution. *Facets of Genetics, Readings from Scientific American.* (1969), 86-95.

[19] By Wolfgang Gruener *Computer Applications in the Biosciences.* (April 22, 2009), http://www.tgdaily.com/content/view/42125/135/

[20] P. Guerdoux-Jamet and D. Lavenier. SAMBA: Hardware accelerator for biological sequence comparison. *Computer Applications in the Biosciences.* (1997), 609-615.

[21] Balazs Tukora, Tibor Szalay High Performance computing on Graphics processing unit. *An International Journal for Engineering and Information Sciences* (2008), 27-34.

[22] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Kevin Skadron A Performance Study of General-Purpose Applications on Graphics Processors Using CUDA. *University of Virginia, Department of Computer Science*

[23] A. Deshpande, D. Richards, and W. Pearson. A platform for biological sequence comparison of parallel computers. *Computer Applications in the Biosciences*,(1991),7:237-247.

[24] Nvidia Cuda. *Programming Guide* (26 May 2009),11.

[25] Michael C. Schatz and Cole Trapnell. Fast exact string matching on the gpu *Centre for bioinformatics and computational biology.*(2008).

[26] D. Manocha. Genral purpose computations using graphics processors. In *University of North Carolina at Chapel Hill* (Aug 2005),85-88.

[27] D. Manocha. Genral purpose computations using graphics processors. In *University of North Carolina at Chapel Hill* (Aug 2005),85-88.