Mapping Dynamic Programming Algorithms on Graphics Processing Units

Vom Promotionsausschuss der

Technischen Universität Hamburg-Harburg

zur Erlangung des akademischen Grades
Doktor der Naturwissenschaften
genehmigte Dissertation

von

Muhammad Kashif Hanif

aus

Faisalabad, Pakistan

2014

1. Gutachter: Prof. Dr. Karl-Heinz Zimmermann,
   Institut für Rechnertechnologie, Technische Universität Hamburg-Harburg
2. Gutachter: Prof. Dr. Sibylle Schupp,
   Institut für Softwaresysteme, Technische Universität Hamburg-Harburg

Tag der mündlichen Prüfung: 03.07.2014

Vorsitzender des Prüfungsausschusses: Prof. Dr. Ralf Möller,
Institut für Softwaresysteme, Technische Universität Hamburg-Harburg

Gedruckt mit Unterstützung des Higher Education Commission, Pakistan

To my beloved parents

Abstract

The Graphics Processing Unit (GPU) is a highly parallel, many-core streaming architecture that can execute hundreds of threads concurrently. The data parallel architecture of the GPU is suitable to perform computation intensive applications. In recent years, the use of GPUs for general purpose computation has increased and a large set of problems can be tackled by mapping onto GPUs. The programming model CUDA enables to design `C` like programs with some extensions which leverages programmers to efficiently use the graphics API.

Alignment is the fundamental operation used to compare biological sequences and in this way to identify regions of similarity that are eventually consequences of structural, functional, or evolutionary relationships. Multiple sequence alignment is an important tool for the simultaneous alignment of three or more sequences. Efficient heuristics exist to cope with this problem.

In the thesis, progressive alignment methods and their parallel implementation by GPUs are studied. More specifically, the dynamic programming algorithms of profile-profile and profile-sequence alignment are mapped onto GPU. Wavefront and matrix-matrix product techniques are discussed which can deal well with the data dependencies. The performance of these methods is analyzed. Simulations show that one order of magnitude of speed-up over the serial version can be achieved.

ClustalW is the most widely used progressive sequence alignment method which aligns more closely related sequences first and then gradually adds more divergent sequences. It consists of three stages: distance matrix calculation, guide tree compilation, and progressive alignment. In this work, the efficient mapping of the alignment stage onto GPU by using a combination of wavefront and matrix-matrix product techniques has been studied.

In the hidden Markov model, the Viterbi algorithm is used to find the most probable sequence of hidden states that has generated the observation. In the thesis, the parallelism exhibited by the compute intensive tasks is studied and a parallel solution based on the matrix-matrix product method onto GPU is devised. Moreover, the opportunity to use optimized BLAS library provided by CUDA is explored. Finally, the performance by fixing the number of states and changing the number of observations and vice versa is portrayed.

At the end, general principles and guidelines for GPU programming of matrix-matrix product algorithms are discussed.

Acknowledgements

Several people helped me to complete my thesis. Foremost, I would like to express my deepest gratitude to my advisor Prof. Dr. Karl-Heinz Zimmermann for the continuous support of my Ph.D study, for his patience, motivation, and enthusiasm. His guidance helped me in every single step of the research and writing of this thesis. I am extremely grateful for his help, without which it would have been impossible for me to complete this thesis.

I am thankful to the Higher Education Commission (HEC), Pakistan and the German Academic Exchange Service (DAAD) for financial support. The Institute of Computer Technology has provided me the necessary hardware to carry out my research.

I would like to thank all my colleagues and staff at the Institute of Computer Technology for their support. Special thanks to Mr. Stefan Just for providing technical support. I also acknowledge Svetlana Torgasin and Cem Bassoy who helped me during my work. I am grateful to Oscar, Mahwish, and Srinavasa for the pleasant conversation and positive feedback.

Many friends have helped me stay focused through these years. Special thanks to Sabeeh Raza who was always a great support.

Finally, I thank my parents who encouraged and helped me at every stage of my personal and academic life. I am also grateful to my brother, sisters, and wife for their support.

Contents

1 Introduction 1
1.1 Contribution of this Work 3
1.2 Organization 3

2 Fundamentals 5
2.1 Dynamic Programming 5
2.2 Biomolecular Sequences 6
  2.2.1 Pairwise Sequence Alignment 7
  2.2.2 Multiple Sequence Alignment 10
2.3 Tropical Algebra 16
2.4 Hidden Markov Model 18
  2.4.1 Viterbi Algorithm 21

3 Graphics Processing Unit 25
3.1 Design Policy of GPU 25
3.2 Programming Model 25
  3.2.1 Thread Hierarchy 26
  3.2.2 Memory Model 27
  3.2.3 Thread Synchronization 27
  3.2.4 CUBLAS Library 28
3.3 GPU Hardware Architecture 28
  3.3.1 Fermi Architecture 29
  3.3.2 Kepler Architecture 29
3.4 Hardware 30

4 Sequence Alignment 33
4.1 Related Work 33
4.2 Profile-Profile Alignment 34
  4.2.1 Matrix Approach 34
  4.2.2 Wavefront Approach 38
  4.2.3 Performance 40
4.3 Profile-Sequence Alignment 47
  4.3.1 Matrix Approach 47
  4.3.2 Performance 51
4.4 ClustalW Algorithm 56
  4.4.1 Performance 58

5 Viterbi Algorithm 61
5.1 Related Work 61
5.2 Parallel Viterbi Algorithm 62
5.3 Performance 65

6 Dynamic Programming 73
6.1 Mapping Dynamic Programming Problems on GPU 73
6.1.1 Serial Monadic Dynamic Programming 73
6.1.2 Non-serial Monadic Dynamic Programming 74
6.1.3 Serial Polyadic Dynamic Programming 76
6.1.4 Non-serial Polyadic Dynamic Programming 81
6.2 General guidelines for GPU programming 82

7 Conclusion 87

Bibliography 91

List of Figures

2.1 An example of pairwise sequence alignment 7
2.2 An example of Needleman-Wunsch algorithm 8
2.3 An example of Smith-Waterman algorithm 9
2.4 Two examples of multiple sequence alignment 11
2.5 An example profile 12
2.6 An example of profile-profile alignment 12
2.7 Multiple alignment resulting from profile-profile alignment 13
2.8 An example of profile-sequence alignment 14
2.9 Multiple alignment resulting from profile-sequence alignment 14
2.10 Stages of the ClustalW algorithm 17
2.11 An example of the ClustalW algorithm 18
2.12 Hidden Markov model at state σ_k. 20
2.13 The CpG island trellis diagram 21
2.14 An example of the Viterbi algorithm 23

3.1 Grid of thread blocks 26
3.2 CUDA memory model 28
3.3 Fermi streaming multiprocessor architecture 31
3.4 Kepler GK110 streaming multiprocessor (SMX) architecture 32

4.1 Data dependencies in the forward table. 35
4.2 The forward table for profile-profile alignment. 39
4.3 Portion of a decomposition of the forward table into blocks of size 2 × 2. 39
4.4 Block structure of the forward table. 40
4.5 Runtime of ProfProfAlignScalProd on NVIDIA. 41
4.6 Runtime of ProfProfAlignScalProd on NVIDIA and Intel CPU using MKL. 42
4.7 Runtime of profile-profile alignment algorithms on NVIDIA and Intel CPU (with and without MKL) by considering kernel execution and transfer of results back to CPU. 43
4.8 Runtime of profile-profile alignment algorithms on NVIDIA and Intel CPU (with and without MKL) by considering memory allocation, data transfer, and kernel execution. 44
4.9 Speed-ups of profile-profile alignment algorithms over CPU-MKL by considering kernel execution and transfer of results back to CPU memory. 45

4.10 Speed-ups of profile-profile alignment algorithms over CPU-MKL by considering kernel execution, memory allocation, and data transfer between CPU and GPU memory. 47
4.11 Performanc (MCUPS) of profile-profile alignment algorithms. 48
4.12 Runtime of profile-sequence alignment algorithms on NVIDIA and Intel CPU. 53
4.13 Runtime for calculation of the forward table for profile-sequence alignment algorithms. 55
4.14 Speed-ups of profile-sequence alignment algorithms. 56
4.15 Performance (MCUPS) of profile-sequence alignment algorithms. 58
4.16 Speed-up for the progressive alignment stage of the ClustalW algorithm. 59

5.1 Runtime of the Viterbi algorithm using fixed number of states. 69
5.2 Runtime of the Viterbi algorithm using fixed sequence length. 70
5.3 Speed-up for SMemViterbi using fixed number of states. 71
5.4 Speed-up for SMemViterbi using fixed sequence length. 72

6.1 The recursive step of the Viterbi algorithm. 74
6.2 The data dependencies in profile-sequence alignment. 75
6.3 The anti-diagonal data independence for profile-sequence alignment. . . 76
6.4 Runtime of FloydWarshall algorithm on NVIDIA and Intel CPU. . 78
6.5 Runtime of BipartiteAPSP algorithm on NVIDIA and Intel CPU. . 80
6.6 Runtime comparison of FloydWarshall and BipartiteAPSP algorithms for bipartite graphs on NVIDIA and Intel CPU. 80
6.7 Speed-up of BipartiteAPSP over FloydWarshall algorithm for bipartite graphs. 81
6.8 An example of the NPDP class. 82

List of Tables

2.1 Complexity of the ClustalW algorithm 16

3.1 Comparison of Fermi and Kepler architectures 30
3.2 NVIDIA GeForce GTX 560 Ti properties 30

4.1 Speed-ups of profile-profile alignment algorithms over CPU-MKL by considering kernel execution and transfer of results back to CPU memory 46
4.2 Speed-ups of profile-profile alignment algorithms over CPU-MKL by considering kernel execution, memory allocation, and data transfer between CPU and GPU memory. 46
4.3 Performanc (MCUPS) of profile-profile alignment algorithms. 46
4.4 Runtime of profile-sequence alignment algorithms on NVIDIA and Intel CPU. 54
4.5 Runtime of maximization step for profile-sequence alignment algorithms on CPU. 54
4.6 Speed-ups of profile-sequence alignment algorithms. 57
4.7 Performance (MCUPS) of profile-sequence alignment algorithms. 57

5.1 Runtime of the Viterbi algorithm by fixing number of states. 68
5.2 Runtime of the Viterbi algorithm by fixing sequence length. 68
5.3 Speed-up of the Viterbi algorithm by fixing number of states. 68
5.4 Speed-up of the Viterbi algorithm by fixing sequence length. 69

6.1 Runtime of BipartiteAPSP and FloydWarshall algorithms. . . . 81
6.2 Speed-up of parallel APSP algorithms over serial FloydWarshall algorithm for bipartite graphs. 82

Nomenclature

APSP	All-Pairs Shortest Path
BLAS	Basic Linear Algebra Subprograms
BLOSUM	BLOck SUbstitution Matrix
CUBLAS	Compute Unified Basic Linear Algebra Subprograms
CUDA	Compute Unified Device Architecture
CUPS	Cell Updates Per Second
DNA	DeoxyriboNucleic Acid
FLOPS	FLoating-point Operations Per Second
GPU	Graphics Processing Unit
NMDP	Non-serial Monadic Dynamic Programming
NPDP	Non-serial Ployadic Dynamic Programming
PAM	Point Accepted Mutation
RNA	RiboNucleic Acid
SIMT	Single Instruction Multiple Threads
SM	Streaming Multiprocessor
SMDP	Serial Monadic Dynamic Programming
SMX	Streaming Multiprocessor
SP	Streaming Processor
SPDP	Serial Polyadic Dynamic Programming

Chapter 1

Introduction

The *Graphics Processing Units* (GPUs) [78] are *many-core* massively parallel computing processors. GPUs offer orders of magnitude more computation power than CPUs and are becoming popular for general purpose computations to achieve good speed-up. Since the 1980s, GPUs have been used for graphics applications like 3D models and digital video related functions. NVIDIA introduced the flexible programming model for general purpose computation on graphics card, *Compute Unified Device Architecture* (CUDA), which provides extension to the `C` language. CUDA uses the *Single Instruction Multiple Threads* (SIMT) model to execute a single instruction with different data elements by parallel threads. GPUs have already been employed to general purpose computing in several areas such as molecular dynamics, physics simulations, and scientific computing [78, 80].

Synchronization is a major problem in current GPUs due to lack of inter-thread communication between multiprocessors. This can be done by re-launching the kernel again from CPU, which is a costly operation. Therefore, the applications having task or data parallelism with no inter-multiprocessor communication are well suited to GPU architecture [12, 126]. The Kepler architecture of GPU has dynamic parallelism to invoke a CUDA kernel from another, which provides more flexibility and efficiency [80, 82].

Dynamic programming is a technique to solve search and optimization problems [6]. This works by dividing the problem into sub-problems in a recursive way. Then smallest sub-problems are solved and the results are combined to achieve a solution of the initial problem. Dynamic programming can be divided into *Serial Monadic Dynamic Programming* (SMDP), *Non-serial Monadic Dynamic Programming* (NMDP), *Serial Polyadic Dynamic Programming* (SPDP), and *Non-serial Polyadic Dynamic Programming* (NPDP) classes corresponding to data dependencies and cost function [34, 60, 119].

However, high complexity restricts the use of dynamic programming. One approach is to utilize parallel processing to achieve massive speed-up. We can achieve huge performance boosts by mapping dynamic programming based algorithms onto GPUs. In this thesis, we will focus on the performance optimization of dynamic programming algorithms by discussing the opportunities to convert dynamic programming algorithms into a form which matches the vector-processing architecture of commodity GPUs. For this, the algorithms are redesigned into matrix-matrix product and the resultant algorithms are mapped to GPU architecture using custom kernel or by optimized BLAS routines. In addition, we will also present the wavefront technique to parallelize dynamic programming algorithms. In the wavefront approach, parallelism is exposed by breaking up the computations with recurrences into segments and pipelining the execution of segments. This is achieved by computing anti-diagonals of the forward matrix in order to avoid data dependencies [1].

Alignment is a fundamental operation in bioinformatics to compare sequences. Dynamic programming based algorithms are commonly used for sequence alignment. However, the computational complexity of these algorithms restricts their usage for longer sequence lengths [115]. Moreover, the availability of high throughput sequencing technologies results in exponential growth of the size of biomolecular sequence databases [102]. Therefore, fast alignment techniques are required to find similarities in large data sets. Simultaneous alignment of multiple sequences is a very costly operation in terms of computation and storage. Many heuristic methods run in reasonable time with less accuracy. The most popular method to generate multiple sequence alignment is progressive alignment in which the most similar sequences are aligned first and then less related sequences are added to the alignment [28, 43, 76].

In this work, we will investigate the use of GPUs to align multiple sequences to achieve better speed-up compared to contemporary CPUs. We will discuss profile-profile alignment and profile-sequence alignment using matrix-matrix product and wavefront methods. Previously, a matrix-matrix product based algorithm was designed to enhance the performance of profile-sequence alignment [5]. Due to the limited amount of GPU memory, it cannot handle longer sequences. For this, the algorithm will be redesigned to process large sequence lengths and to improve the performance. Furthermore, this algorithm will be implemented using the wavefront method. ClustalW is a widely used alignment method to align multiple sequences. In this thesis, we will present a parallelization strategy for the progressive alignment stage of the ClustalW using a mixture of wavefront and matrix-matrix multiplication.

Hidden Markov model [92] is a statistical tool to represent probability distribution for the sequence of observations. Hidden Markov models have been used in many areas including computational biology [52], speech recognition [91], and pattern recognition [29]. However, the algorithms using the hidden Markov model require a high computational complexity. For this purpose, there is a need to improve the processing time and to reduce the complexity. Three basic problems for hidden Markov model are evaluation, decoding, and learning. Decoding a hidden sequence of states is an important task and the most widely used method is the Viterbi algorithm. The Viterbi algorithm [31, 117] is a dynamic programming algorithm to find the most likely sequence of hidden states that generated the observed sequence. There will be l^n possible routes for n observations and l possible states for an observation. The Viterbi algorithm finds the route through l states that maximizes the probability of observations with time complexity $O(nl^2)$. To speed-up the Viterbi algorithm, we will formulate a matrix-matrix product based solution which is suitable for the GPU architecture.

We will show a performance comparison of different types of dynamic programming algorithms to better understand the mapping process onto GPU. Finally, general guidelines for implementing dynamic programming based algorithms will be discussed.

1.1 Contribution of this Work

The major contributions of this work are as follows:

- Multiple sequence alignment algorithms are studied. In particular, the parallel versions of profile-profile and profile-sequence alignment algorithms are designed using the wavefront and the matrix-matrix product techniques. We have discussed the performance of these methods for the GPU hardware architecture.

- Proposed an improved matrix-matrix product based solution for profile-sequence alignment. This method uses less memory space, which helps to align longer sequences.

- Parallelization of the progressive alignment stage of the ClustalW algorithm using a combination of wavefront and matrix-matrix product methods has been investigated.

- Provided the design, implementation, and experimental setup of matrix-matrix product based solution for the Viterbi algorithm. This is achieved by dividing the Viterbi algorithm into dependent and independent parts. The data independent part is executed on the GPU to improve performance.

- Explored the strategies for mapping of dynamic programming problems on GPUs and described the general principles for GPU programming of matrix-matrix product.

1.2 Organization

The rest of the thesis is organized as follows:

- Chapter 2 starts with a brief introduction to dynamic programming, followed by the fundamental concepts related to sequence alignment and hidden Markov model. First, dynamic programming algorithms for the pairwise and multiple sequence alignment are presented. Next, the most widely used progressive alignment algorithm ClustalW is discussed. We have also given a brief introduction to tropical algebra. This chapter ends by introducing the hidden Markov model and the Viterbi algorithm.

- Chapter 3 introduces the GPU computing model. This includes the programming model and GPU architecture. We have described the Fermi and Kepler architectures.

- Chapter 4 describes the parallel design of multiple sequence alignment algorithms. First, a parallel solution of profile-profile alignment is discussed and implemented onto GPU. Next, a parallel version for profile-sequence alignment is described. This chapter concludes with the parallel formulation of the progressive alignment stage of the ClustalW algorithm.

- Chapter 5 presents the strategies to parallelize the Viterbi algorithm. We have discussed both the matrix-vector and matrix-matrix product methods.

- Chapter 6 discusses the mapping of different types of dynamic programming problems onto GPU. General guidelines of matrix-matrix product based algorithms on GPU are also presented.

- Chapter 7 concludes the thesis with a discussion about the results and future work.

Chapter 2

Fundamentals

Dynamic programming is a widely used technique to solve optimization problems. Sequence alignment is the method to compare sequences for searching the regions of similar character patterns. Sequence alignment is use to determine protein family, pattern identification, phylogenetic analysis, and structure prediction [23, 115]. The sequence alignment problem can be tackled using dynamic programming. However, exponential growth in the size of sequence databases demands for sophisticated analysis techniques.

This chapter begins with a brief introduction to dynamic programming. We will present the necessary concepts related to sequence alignment and the associated algorithms for the sequence alignment. Then we will give basic concepts related to the tropical algebra. Finally, the Viterbi algorithm to solve the decoding problem of hidden Markov model will be discussed.

2.1 Dynamic Programming

Dynamic programming is a technique for solving a problem by dividing into interdependent sub-problems. The results of the sub-problems are used to solve larger sub-problems until the entire problem is solved. In the divide-and-conquer technique, a problem depends only on the solution of its sub-problems. However, there may be relationships between the sub-problems for dynamic programming technique [34].

A dynamic programming formulation is usually represented as recurrences on intermediate solutions to a problem. An optimal solution is defined in terms of the objective function involving the minimization (or maximization) of some cost function [119]. The computation of dynamic programming problems is based on establishing a matrix. An optimal solution of the smallest sub-problems is calculated first, then the optimal solution for the smaller sub-problems are used to solve larger sub-problems according to the recursive function. This property will guarantee the optimal solution for the whole problem.

Consider the dynamic programming problem as multistage problem consisting of many sub-problems. If the solution of the sub-problems at all levels depends only on the immediately preceding levels, it is called serial; otherwise, non-serial. The cost function involving only single recursive term is monadic; otherwise, polyadic. Based on this criteria, Grama et al. [34] classifies the dynamic programming problems into four categories: serial monadic dynamic programming, serial polyadic dynamic programming, non-serial monadic dynamic programming, and non-serial polyadic dynamic programming. The sample problems for each of these classes and their parallelization strategies will be discussed later.

2.2 Biomolecular Sequences

The fundamental principle of molecular biology, which states that the genetic information flows from the *DeoxyriboNucleic Acid* (DNA) to the proteins, is known as the *central dogma of biology.* The genetic information required to make proteins resides in the DNA. First, this encoded information is transmitted into the *RiboNucleic Acid* (RNA). This process is called *transcription.* Then the RNA is translated into a protein by the *translation* process [88].

The DNA is the double-helical structure that encodes the genetic information required to build and functioning of an organism. The genetic information is coded by the sequence of four nucleotides: Adenine (A), Cytosine (C), Guanine (G), and Thymine (T). A *gene* is the basic physical and functional unit of heredity which is made up of DNA. It acts as an instruction to make a molecule called *protein.* The quantity and organization of the genes are different from one organism to another. The whole genetic information encoded in the DNA is called *genome.* DNA in the human cell is organized into large linear structures called *chromosomes.*

Sequence alignment is the technique in molecular biology used to compare sequences and to arrange sequences of biomolecules for identifying regions of similarity that are eventually consequences of structural, functional, or evolutionary relationships [23, 38, 122, 128]. Sequence alignment can be performed for DNA, RNA, or protein sequences.

The alignment of two sequences is called *pairwise alignment* which corresponds to either global alignment, where sequences are aligned to their entire length, or local alignment, where portion of sequences are aligned. *Multiple sequence alignment* is the technique to align three or more sequences simultaneously. The aligned sequences are obtained by inserting gaps and have equal length. A column of blanks is not allowed. Two identical characters in a column means match; otherwise, mismatch. There are different scoring schemes which give scores for match, mismatch, and gap.

The sequence alignment problem can be tackled using two computational approaches: optimal methods following the paradigm of dynamic programming [37] and heuristic methods [28, 115]. Note that these methods assume that the columns are aligned independent of each other. The size of biomolecular sequence databases grows exponentially due to the recent availability of high-throughput sequencing technologies [102]. This upsurge demands for fast alignment techniques rendering the more time-consuming optimal alignment techniques less useful for searching similarities in larger data sets. In particular, dynamic programming algorithms for simultaneously aligning k sequences of length $O(n)$ necessitates $O(2^k n^k)$ steps and thus are only feasible for a handful of sequences [128]. This is the reason why fast heuristic techniques such as BLAST [2] and FASTA [89] are preferred that are an order of magnitude faster than the optimal algorithms. However, the downside of heuristic approaches is that they are less sensitive (i.e., missing more homologous) than the optimal ones [128].

```
MNSFSTSAFGPVAFSLGLLLVLPAAFP-APVPPGEDSKDVAAPHRQPLTS
|...|...|.|||| |||:||...||| :.|..|:.::| ..|:| |:.:
MKFLSARDFHPVAF-LGLMLVTTTAFPTSQVRRGDFTED-TTPNR-PVYT

SERIDKQIRYILDGISALRKETCNKSNMCESSKEALAENNLNLPKMAEKD
:.::...|.::|..|..:|||.||.::.|.::.:|||||||.||::...|
TSQVGGLITHVLWEIVEMRKELCNGNSDCMNNDDALAENNLKLPEIQRND

GCFQSGFNEETCLVKIITGLLEFEVYLEYLQNRF-ESSEEQARAVQMSTK
||:|:|:|:|.||:||.:||||:..||||::|.. ::.:::||.:|..|:
GCYQTGYNQEICLLKISSGLLEYHSYLEYMKNNLKDNKKDKARVLQRDTE

VLIQFLQKKAKNLDAITTPDPTTNASLLTKLQAQNQWLQDMTTHLILRSF
.||....::.|:|..|..|.|.:||.|..||::|.:||:..|...||:|.
TLIHIFNQEVKDLHKIVLPTPISNALLTDKLESQKEWLRTKTIQFILKSL

KEFLQSSLRALRQM
:|||:.:||:.||.
EEFLKVTLRSTRQT
```

Fig. 2.1: Pairwise sequence alignment between Interleukin-6 proteins from human and mouse calculated by the EMBOSS pairwise alignment algorithm from EMBL-EBI taking the substitution matrix BLOSUM62, gap penalty 10.0, and gap extension penalty 5.0; the alignment has length 214 with 41.6% matches, 56.1% mismatches, and 2.3% indels.

2.2.1 Pairwise Sequence Alignment

Pairwise sequence alignment is the technique to compare and align two sequences. Aligned sequences are usually represented as rows of a matrix. Gaps are inserted between the residues such that identical or related residues can be aligned in corresponding columns (Fig. 2.1). Sequences can be represented in several text-based formats like the FASTA format that can be read by most web-based or GUI tools supporting alignment routines [128].

Pairwise sequence alignment for two sequences $\boldsymbol{x} = x_1 x_2 \dots x_m$ and $\boldsymbol{y} = y_1 y_2 \dots y_n$ is the method to find new sequences $\boldsymbol{x}^{'}$ and $\boldsymbol{y}^{'}$ of equal length such that there are minimum number of steps to transform $\boldsymbol{x}$ into $\boldsymbol{y}$ (vice versa) using insertion, deletion, or replacement of characters.

$$\begin{array}{rcccccc} \boldsymbol{x}^{'} & = & x_1^{'} & x_2^{'} & x_3^{'} & \dots & x_k^{'} \\ \boldsymbol{y}^{'} & = & y_1^{'} & y_2^{'} & y_3^{'} & \dots & y_k^{'} \end{array} \tag{2.1}$$

The minimum number of transformations to convert one sequence into another is called *edit distance*. The goal is to find the alignment with minimum edit distance. The maximum length of aligned sequences, in worst case, can be $m + n$ [128].

There are $(2n)!/(n!)^2 \approx (2)^{2n}/\sqrt{(\pi n)}$ possible alignments and to find the best alignment is a difficult task [23]. An optimal pairwise alignment is the pairwise alignment with highest alignment score. For this, we have to use a specific scoring scheme to score matches, mismatches, and gaps. The *BLOck SUbstitution Matrix* (BLOSUM) [41] and *Point Accepted Mutation* (PAM) [18] are most widely used scoring matrices for protein sequences. Choosing the different scoring matrices can lead to different optimal solutions. Here, the focus is to use an appropriate alignment technique. The scoring model for the sequence alignment algorithms presented in this work assumes that the aligned columns are statistically independent of each other.

Pairwise sequence alignment can be solved by dynamic programming in $O(n^2)$ steps if both sequences have length $O(n)$. The Needleman-Wunsch and Smith-Waterman algorithms are the basic pairwise sequence alignment methods following the paradigm of dynamic programming.

Needleman-Wunsch Algorithm

The Needleman-Wunsch algorithm provides a global alignment of two sequences aligning every residue in both sequences (Fig. 2.2). It is most useful when the sequences are similar and of roughly equal length [73]. This algorithm works in three steps. Firstly, the forward matrix is initialized. Then the forward matrix is calculated. Finally, alignment is deduced using traceback. The basic idea is to make optimal decisions at each stage and gradually adding sequences by one at a time using the optimal score at that stage. In this way, an optimal alignment can be determined.

```
V S P A G M A S G Y D C A
I − P − G K A S − Y D A C
```

Fig. 2.2: Global pairwise sequence alignment using Needleman-Wunsch algorithm of amino acid sequences `VSPAGMASGYDCA` and `IPGKASYDAC` taking scoring matrix BLOSUM50.

For two given sequences $\boldsymbol{x}$ and $\boldsymbol{y}$ over the alphabet Σ of length m and n, respectively and scoring matrix $\sigma(a, b)$, the Needleman-Wunsch algorithm can be defined as:

$$D_{i,j} = \begin{cases} 0, & \text{if } i = 0 \text{ and } j = 0, \\ \sum_{k=1}^{i} \sigma(x_k, -), & \text{if } 1 \leq i \leq m \text{ and } j = 0, \\ \sum_{k=1}^{j} \sigma(-, y_k), & \text{if } 1 \leq j \leq n \text{ and } i = 0, \\ \max\{D_{i-1,j-1} + \sigma(x_i, y_j), D_{i-1,j} + \\ \sigma(x_i, -), D_{i,j-1} + \sigma(-, y_j)\}, & \text{if } 1 \leq i \leq m \text{ and } 1 \leq j \leq n. \end{cases} \tag{2.2}$$

The routine NeedlemanWunschAlign implements the Needleman-Wunsch algorithm. Its input is given by two sequence $\boldsymbol{x}$ and $\boldsymbol{y}$. The algorithm begin by initializing $D_{0,0} = 0$. The first row and column are filled according to the selected scoring scheme which represents gap penalties at the start of either sequence. The $D_{i,j}$ value depends on $D_{i-1,j-1}$ (match or mismatch), $D_{i,j-1}$ (insertion), or $D_{i-1,j}$ (deletion). The entry $D_{m,n}$ gives the global alignment score. The global alignment can be retrieved by tracing back the optimal decision made at each step. There is also the possibility to have multiple alignment paths for the same optimal score. For sequences of comparable length, the runtime and memory complexity is $O(n^2)$ [128].

Algorithm 2.1 NeedlemanWunschAlign($\boldsymbol{x}, \boldsymbol{y}$)

```
Require: two sequences $\boldsymbol{x} = x_1, \ldots, x_m$ and $\boldsymbol{y} = y_1, \ldots, y_n$
Ensure: $\boldsymbol{D} = (D_{ij})$ forward matrix
$D_{0,0} \leftarrow 0$                                // initialization
for $i \leftarrow 1$ to $m$ do
    $D_{i,0} \leftarrow \sum_{k=1}^{i} \sigma(x_k, -)$
end for
for $j \leftarrow 1$ to $n$ do
    $D_{0,j} \leftarrow \sum_{k=1}^{j} \sigma(-, y_k)$
end for
for $i \leftarrow 1$ to $m$ do                      // computation and maximization
    for $j \leftarrow 1$ to $n$ do
        $D_{i,j} \leftarrow \max\{D_{i-1,j} + \sigma(x_i, -), D_{i,j-1} + \sigma(-, y_j), D_{i-1,j-1} + \sigma(x_i, y_j)\}$
    end for
end for
return $\boldsymbol{D}$
```

Smith-Waterman Algorithm

The Needleman-Wunsch algorithm aligns sequences for their entire length. However, finding the common sub-sequences for dissimilar sequences that are suspected to contain regions of similarity is also important [106]. Aligning similar regions of sequences while leaving highly divergent regions unaligned is referred as *local alignment.* A local alignment between two amino acid sequences `VSPAGMASGYDCA` and `IPGKASYDAC` is illustrated in Figure 2.3. The local alignment gives only the best matching sub-sequence.

```
P A G M A S G Y D − C
P − G K A S − Y D A C
```

Fig. 2.3: Local pairwise sequence alignments using Smith-Waterman algorithm of amino acid sequences `VSPAGMASGYDCA` and `IPGKASYDAC` taking scoring matrix BLOSUM50.

The Smith-Waterman algorithm yields a local alignment of two sequences in which only part of the residues participate [106]. For two given sequences $\boldsymbol{x}$ and $\boldsymbol{y}$ over the alphabet Σ having length m and n, respectively and scoring matrix $\sigma(a, b)$, the Smith-Waterman algorithm can be defines as:

$$D_{i,j} = \begin{cases} 0, & \text{if } i = 0 \text{ and } j = 0, \\ 0, & \text{if } 1 \leq i \leq m \text{ and } j = 0, \\ 0, & \text{if } 1 \leq j \leq n \text{ and } i = 0, \\ \max\{0, D_{i-1,j-1} + \sigma(x_i, y_j), D_{i-1,j} + \\ \sigma(x_i, -), D_{i,j-1} + \sigma(-, y_j)\}, & \text{if } 1 \leq i \leq m \text{ and } 1 \leq j \leq n. \end{cases} \tag{2.3}$$

The algorithm initializes the first row and column to zero instead of the gap penalties. The $D_{i,j}$ value depends on $D_{i-1,j-1}$ (match or mismatch), $D_{i,j-1}$ (insertion), or

Algorithm 2.2 SmithWatermanAlign($\boldsymbol{x}, \boldsymbol{y}$)

```
Require: two sequences $\boldsymbol{x} = x_1, \ldots, bx_m$ and $\boldsymbol{y} = y_1, \ldots, y_n$
Ensure: $\boldsymbol{D} = (D_{ij})$ forward matrix
$D_{0,0} \leftarrow 0$                                   // initialization
for $i \leftarrow 1$ to $m$ do
    $D_{i,0} \leftarrow 0$
end for
for $j \leftarrow 1$ to $n$ do
    $D_{0,j} \leftarrow 0$
end for
for $i \leftarrow 1$ to $m$ do                           // computation and maximization
    for $j \leftarrow 1$ to $n$ do
        $D_{i,j} \leftarrow \max\{0, D_{i-1,j} + \sigma(x_i, -), D_{i,j-1} + \sigma(-, y_j), D_{i-1,j-1} + \sigma(x_i, y_j)\}$
    end for
end for
return $\boldsymbol{D}$
```

$D_{i-1,j}$ (deletion). The recurrence to calculate $D_{i,j}$ also has zero which avoids negative scores. The reason is to start a new local alignment instead of extending it. The optimal local alignment can be anywhere in the matrix instead of $D_{m,n}$. Local alignment can be retrieved by starting at the maximum value of $D_{i,j}$ and tracing back the optimal decision made at each step until we reach an entry 0. There is also possibility to have multiple alignment paths for the same optimal score. The Smith-Waterman algorithm has the runtime and memory complexity $O(n^2)$ for the sequences of comparable length [128].

2.2.2 Multiple Sequence Alignment

Multiple sequence alignment corresponds to the simultaneous alignment of three or more sequences. It helps to establish evolutionary relationships that are useful for constructing phylogenies and revealing conserved and variable sites within protein families [23, 115, 128]. For k sequences to be aligned i.e., $\boldsymbol{x} = (\boldsymbol{x}_1 \ldots \boldsymbol{x}_k)$, multiple sequence alignment is the process such that all resulting sequences have equal length by inserting gaps.

$$\begin{array}{lcl} \boldsymbol{x}_1 & = & x_{11} \quad x_{12} \quad x_{13} \quad \ldots \quad x_{1,m_1} \\ \boldsymbol{x}_2 & = & x_{21} \quad x_{22} \quad x_{23} \quad \ldots \quad x_{2,m_2} \\ & \vdots & \\ \boldsymbol{x}_k & = & x_{k1} \quad x_{k2} \quad x_{k3} \quad \ldots \quad x_{k,m_k} \end{array} \tag{2.4}$$

Eqn. (2.5) gives the alignment of k sequences, where all the sequences have equal length i.e., n.

$$\begin{array}{lcl} \boldsymbol{x}_1^{'} & = & x_{11}^{'} \quad x_{12}^{'} \quad x_{13}^{'} \quad \ldots \quad x_{1,n}^{'} \\ \boldsymbol{x}_2^{'} & = & x_{21}^{'} \quad x_{22}^{'} \quad x_{23}^{'} \quad \ldots \quad x_{2,n}^{'} \\ & \vdots & \\ \boldsymbol{x}_k^{'} & = & x_{k1}^{'} \quad x_{k2}^{'} \quad x_{k3}^{'} \quad \ldots \quad x_{k,n}^{'} \end{array} \tag{2.5}$$

Columns consisting solely of blanks are not allowed [128]. The original sequences can be obtained by removing gaps (Fig. 2.4).

```
– A G C G G –     – – – G A A A
A C G T C G –     G T C T A – –
– T G C T G C     – C T C A T –
```

Fig. 2.4: Two multiple sequence alignments of DNA sequences. The first alignment is formed by `AGCGG`, `ACGTCG`, and `TGCTGC`, and the second by `GAAA`, `GTCTA`, and `CTCAT`. These alignments are calculated by the ClustalW algorithm [33, 55] from EMBL-EBI taking the substitution matrix IUB, gap penalty 5.0, and gap extension penalty 1.0.

However, multiple sequence alignment is very time consuming. Therefore, a variety of heuristic methods have been developed for the simultaneous alignment of three or more sequences [75]. The most popular method to generate a multiple sequence alignment is based on trees which are used to describe a relationship between the sequences based on their pairwise comparison. Progressive alignment is the most widely accepted heuristic method for aligning multiple sequences [42, 76]. The progressive method first aligns the most similar sequences and then less related sequences or groups of sequences are successively added to the alignment [28]. In order to improve alignment accuracy, some progressive methods additionally assess the sequences according to their relatedness.

Progressive alignment works in three steps. First, the optimal alignments between each pair of sequences are computed. Second, the so-called guide tree is built that reflects similarities (or distances) among the sequences. Third, the guide tree is used to combine the sequences into a multiple alignment. For this, intermediate alignments are formed from the leaves to the root such that two neighboring sequences are pairwisely aligned, a sequence and a neighboring alignment are aligned by profile-sequence alignment, and two neighboring alignments are aligned by profile-profile alignment [28].

The progressive alignment algorithms can cope with a larger number of sequences in practical time scales. The most widely used multiple sequence alignment programs are Clustal [15, 42, 114] and T-Coffee [76]. ClustalW is faster than T-Coffee but less sensitive [48].

Profile-Profile Alignment

Multiple sequence alignment is used to determine the protein structures, evolutionary relationship, and conservation of homologous regions. Classification of proteins into their respective protein families, finding relatedness of proteins within same family, and detecting similarity between proteins belonging to different families are important tasks [23, 115]. This can be achieved by aligning a multiple alignment against another multiple alignment. Profile-profile alignment is more accurate over profile-sequence and sequence-sequence alignment methods for multiple sequence alignment [26].

A multiple sequence alignment can be represented as a profile which can be helpful for more accurate alignments. A *profile* is a statistical representative produced from an alignment and contains the probability of finding each amino acid type at each position [26, 35]. It can be pictured by an $l \times m$ matrix $\mathbf{P} = (p_{ij})$, where l is the size of the extended alphabet $\Sigma' = \Sigma \cup \{-\}$ and m is the length of the alignment. The entry p_{ij} gives the relative frequency of the symbol (residue) i to occur in the j-th column of the alignment (Fig. 2.5). RNA and DNA alphabets consist of four respective nucleotides while the amino acid alphabet has 20 (naturally occurring) amino acids [50].

$$\begin{array}{c} A \\ C \\ G \\ T \\ - \end{array} \begin{pmatrix} 0.33 & 0.33 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 \\ 0.00 & 0.33 & 0.00 & 0.66 & 0.33 & 0.00 & 0.33 \\ 0.00 & 0.00 & 1.00 & 0.00 & 0.33 & 1.00 & 0.00 \\ 0.00 & 0.33 & 0.00 & 0.33 & 0.33 & 0.00 & 0.00 \\ 0.66 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.66 \end{pmatrix}$$

Fig. 2.5: Profile of the first multiple sequence alignment given in Fig. 2.4; the rows are labelled in turn by the symbols `A`, `C`, `G`, `T`, and $-$.

Progressive alignment describes alignments by profiles. The alignment between the two sequence alignments represented by profiles $\mathbf{P}$ and $\boldsymbol{Q}$ is [128]

$$\begin{array}{rcl} \mathbf{P}^{'} & = & \boldsymbol{p}_1^{'} \ \boldsymbol{p}_2^{'} \ \boldsymbol{p}_3^{'} \ \cdots \ \boldsymbol{p}_k^{'} \\ \boldsymbol{Q}^{'} & = & \boldsymbol{q}_1^{'} \ \boldsymbol{q}_2^{'} \ \boldsymbol{q}_3^{'} \ \cdots \ \boldsymbol{q}_k^{'} \end{array} \tag{2.6}$$

The resultant alignment has both sequences of equal length which are derived from the corresponding profiles by inserting blank columns (Figs. 2.6 and 2.7). There are several profile-profile alignment approaches. The major difference between them is choosing a particular scoring function [27, 84, 96, 118]. We have used the score of a profile-profile alignment $(\mathbf{P}', \boldsymbol{Q}')$ as the sum of so-called *column scores*

$$d(\mathbf{P}', \boldsymbol{Q}') = \sum_{i=1}^{k} d(\boldsymbol{p}_i', \boldsymbol{q}_i'), \tag{2.7}$$

where each column is scored by the Euclidean distance [128]

$$d(\boldsymbol{p}_i', \boldsymbol{q}_i') = \sqrt{\sum_{j=1}^{l} (p_{ij}' - q_{ij}')^2}. \tag{2.8}$$

$$\begin{array}{ccccccccc} \boldsymbol{p}_1 & \boldsymbol{p}_2 & \boldsymbol{p}_3 & \boldsymbol{p}_4 & \boldsymbol{p}_5 & \boldsymbol{p}_6 & \boldsymbol{p}_7 & -_p & -_p \\ -_p & -_p & \boldsymbol{q}_1 & \boldsymbol{q}_2 & \boldsymbol{q}_3 & \boldsymbol{q}_4 & \boldsymbol{q}_5 & \boldsymbol{q}_6 & \boldsymbol{q}_7 \end{array}$$

Fig. 2.6: A profile-profile alignment between the profiles describing the multiple alignments in Fig. 2.4.

The dynamic programming algorithm for profile-profile alignment is specified by the routine ProfProfAlign. Its input is given by two profiles, an $l \times m$ matrix $\mathbf{P} =$

```
– A G C G G – – –
A C G T C G – – –
– T G C T G C – –
– – – – – G A A A
– – G T C T A – –
– – – C T C A T –
```

Fig. 2.7: The overall multiple alignment resulting from profile-profile alignment in Fig. 2.6.

$(\boldsymbol{p}_1, \ldots, \boldsymbol{p}_m)$ and an $l \times n$ matrix $\boldsymbol{Q} = (\boldsymbol{q}_1, \ldots, \boldsymbol{q}_n)$. In particular, a column consisting solely of blanks is associated with the profile column $-_p = (0, \ldots, 0, 1)^T$, where blank occurs with relative frequency 1. The objective of a profile-profile alignment is to find an alignment with minimum score [38, 128].

Algorithm 2.3 ProfProfAlign($\mathbf{P}, \boldsymbol{Q}$)

```
Require: two profiles $\mathbf{P} = \boldsymbol{p}_1, \ldots, \boldsymbol{p}_m$ and $\boldsymbol{Q} = \boldsymbol{q}_1, \ldots, \boldsymbol{q}_n$
Ensure: $\boldsymbol{S} = (S_{ij})$ forward matrix
$S_{0,0} \leftarrow 0$                                   // initialization
for $i \leftarrow 1$ to $m$ do
    $S_{i,0} \leftarrow \sum_{k=1}^{i} d(\boldsymbol{p}_k, -_p)$
end for
for $j \leftarrow 1$ to $n$ do
    $S_{0,j} \leftarrow \sum_{k=1}^{j} d(-_p, \boldsymbol{q}_k)$
end for
for $i \leftarrow 1$ to $m$ do                          // computation and minimization
    for $j \leftarrow 1$ to $n$ do
        $S_{i,j} \leftarrow \min\{S_{i-1,j} + d(\boldsymbol{p}_i, -_p), S_{i,j-1} + d(-_p, \boldsymbol{q}_j), S_{i-1,j-1} + d(\boldsymbol{p}_i, \boldsymbol{q}_j)\}$
    end for
end for
return $\boldsymbol{S}$
```

The routine ProfProfAlign evaluates an $m \times n$ table $\boldsymbol{S} = (S_{ij})$. The first row and column are initilized by calculating the Euclidean distance between a profile and a gap. Then the minimum value of $S_{i,j}$ is obtained by aligning $\boldsymbol{p}_i$ and $\boldsymbol{q}_j$, where $S_{i,j} = S_{i-1,j-1} + d(\boldsymbol{p}_i, \boldsymbol{q}_j)$; aligning $\boldsymbol{p}_i$ to gap $-_p$, where $S_{i,j} = S_{i-1,j} + d(\boldsymbol{p}_i, -_p)$; or aligning $\boldsymbol{q}_j$ to gap $-_p$, where $S_{i,j} = S_{i,j-1} + d(-_p, \boldsymbol{q}_j)$. The backward algorithm retrieves the optimal alignments from the forward table. This is achieved by tracing back through the table from the last entry $S_{m,n}$ to the first entry $S_{0,0}$ considering the optimal decisions made at each step. The paths from the last entry $S_{m,n}$ to the first entry $S_{0,0}$ established in this way correspond one-to-one with the optimal alignments. In this work, we will focus on the parallelization of the algorithm by calculating the forward table. The implementation of traceback is not considered, since it has very low inherent parallelism.

Profile-Sequence Alignment

For the given sequences of a protein family, finding whether a new sequence belong to this family is an important feature which will be useful to determine the common structure and functionality of the sequence. The alignment between a sequence and a multiple alignment is more effective as it can match against a family of protein instead of single members of the protein family [35, 36].

The *profile-sequence alignment* is the alignment of a sequence with a profile which is statistical representation of an alignment. The alignment between the sequence $\boldsymbol{x}$ and the profile $\boldsymbol{O}$ is a pair of sequences [128]

$$\begin{array}{lcl} \boldsymbol{O}' & = & \boldsymbol{o}_1' \ \boldsymbol{o}_2' \ \boldsymbol{o}_3' \ \ldots \ \boldsymbol{o}_k' \\ \boldsymbol{x}' & = & x_1' \ x_2' \ x_3' \ \ldots \ x_k' \end{array} \tag{2.9}$$

Both sequences are of equal length and are derived from the corresponding sequences by inserting blank columns (Figs. 2.8 and 2.9).

$$\begin{array}{cccccccccccc} \boldsymbol{o}_1 & \boldsymbol{o}_2 & \boldsymbol{o}_3 & \boldsymbol{o}_4 & \boldsymbol{o}_5 & \boldsymbol{o}_6 & \boldsymbol{o}_7 & \boldsymbol{o}_8 & -_o & -_o & \boldsymbol{o}_9 & -_o \\ - & - & - & - & - & - & \mathtt{A} & \mathtt{T} & \mathtt{A} & \mathtt{C} & \mathtt{A} & \mathtt{C} \end{array}$$

Fig. 2.8: A profile-sequence alignment describing the multiple alignments corresponding to sequence `ATACAC` and alignment in Fig. 2.6.

–	A	G	C	G	G	–	–	–	–	–	–
A	C	G	T	C	G	–	–	–	–	–	–
–	T	G	C	T	G	C	–	–	–	–	–
–	–	–	–	–	G	A	A	–	–	A	–
–	–	G	T	C	T	A	–	–	–	–	–
–	–	–	C	T	C	A	T	–	–	–	–
–	–	–	–	–	–	A	T	A	C	A	C

Fig. 2.9: The overall multiple alignment resulting from profile-sequence alignment in Fig. 2.8.

The algorithm ProSeqAlign depicts the sequential version of profile-sequence alignment. The sequence $\boldsymbol{x} = x_1 \ldots x_n$ over the alphabet Σ and an $l \times m$ matrix $\boldsymbol{O} = (\boldsymbol{o}_1, \ldots, \boldsymbol{o}_m)$ that provides the profile of an alignment serves as input to the algorithm ProSeqAlign [5, 128]. In particular, the blank corresponding to the profile $\boldsymbol{O}$ is $-_o = (0, \ldots, 0, 1)^T$, where blank occurs with relative frequency 1. The score between a column $\boldsymbol{o}$ of the profile and a character $a \in \Sigma'$ is

$$\sigma(\boldsymbol{o}, a) = \sum_{b \in \Sigma'} \sigma(a, b) \cdot o_b. \tag{2.10}$$

A pre-defined substitution matrix (e.g., BLOSUM62, PAM240) is used to score matches, mismatches, and gaps. This algorithm fills an $m \times n$ table $\boldsymbol{S} = (S_{ij})$. The

Algorithm 2.4 ProSeqAlign($\boldsymbol{x}, \boldsymbol{O}$)

```
Require: sequence $\boldsymbol{x} = x_1 \ldots x_n$ and profile $\boldsymbol{O} = \boldsymbol{o}_1 \ldots \boldsymbol{o}_m$
Ensure: $\boldsymbol{S} = (S_{ij})$ forward matrix
$S_{0,0} \leftarrow 0$                                  // initialization
for $i \leftarrow 1$ to $m$ do
    $S_{i,0} \leftarrow \sum_{k=1}^{i} \sigma(\boldsymbol{o}_k, -)$
end for
for $j \leftarrow 1$ to $n$ do
    $S_{0,j} \leftarrow \sum_{k=1}^{j} \sigma(-_o, x_k)$
end for
for $i \leftarrow 1$ to $m$ do                         // computation and maximization
    for $j \leftarrow 1$ to $n$ do
        $S_{i,j} \leftarrow \max\{S_{i-1,j} + \sigma(\boldsymbol{o}_i, -), S_{i,j-1} + \sigma(-_o, x_j), S_{i-1,j-1} + \sigma(\boldsymbol{o}_i, x_j)\}$
    end for
end for
return $\boldsymbol{D}$
```

first row and column are calculated by aligning the profile and the sequence to gaps, respectively. The computation of $S_{i,j}$ depends on the upper-left, upper, and left table entries. The optimal alignment path is established using the backward algorithm which traces backward through the table from the last entry $S_{m,n}$ to the first entry $S_{0,0}$ by considering optimal decisions made at each step.

ClustalW **Algorithm**

ClustalW is a progressive alignment algorithm making use of the policy "once a gap, always a gap" i.e., gaps introduced earlier in the alignment remain valid as new sequences are added. This approach first aligns more closely related sequences, gradually adding divergent sequences [43, 114]. This algorithm consists of three main stages (Fig. 2.10). The first stage calculates the distances between each pair of sequences by pairwise sequence alignment. Due to symmetry, only the upper- or lower-triangular part of the distance matrix is required. Pairwise sequence alignment can be calculated by a slower and accurate dynamic programming based method or a fast heuristic method [4, 62, 63, 114]. The fast pairwise alignment method calculates scores using exactly matching fragments (k-tuples) in the best alignment minus fixed penalty for gaps. The value of k-tuples can be increased for speed or decreased for sensitivity. The slower full pairwise alignment calculate scores by counting the number of identities in the optimal alignment and dividing them by the number of residues. The scores of attained pairwise alignments are converted into distances which are input for the next stage [114].

The second stage of ClustalW uses the distance matrix calculated in the first stage to build the guide tree which serves as a guide for the calculation of the overall multiple sequence alignment. This tree can be constructed by the *Neighbour-Joining* (NJ) method [97] or by the *Unweighted Pair Group Method with Arithmetic mean* (UPGMA) method [55].

The last stage uses the guide tree to progressively align the sequences. The se-

Table 2.1: Complexity of the ClustalW algorithm [63].

Stage	$\boldsymbol{O}$**(Time)**
Distance matrix	$O(n^2l^2)$
Guide tree	$O(n^3)$
Progressive alignment	$O(nl^2 + n^2l)$
Total	$O(n^2l^2 + n^3)$

quences correspond one-to-one with the leaves of the tree. The inner nodes represent the alignment of two existing alignments or sequences. Three cases can occur:

- An inner node whose descendants are leaves is associated with the pairwise alignment of the sequences corresponding to these leaves.
- An inner node whose descendants are a leaf and an inner node is associated to the alignment given by the sequence and the multiple alignment. This can be achieved by profile-sequence alignment where the given multiple alignment is represented by a statistical representative called profile.
- An inner node whose descendants are two inner nodes is associated to the alignment given by the corresponding multiple alignments. This can be attained by profile-profile alignment where the given multiple alignments are represented by statistical representatives.

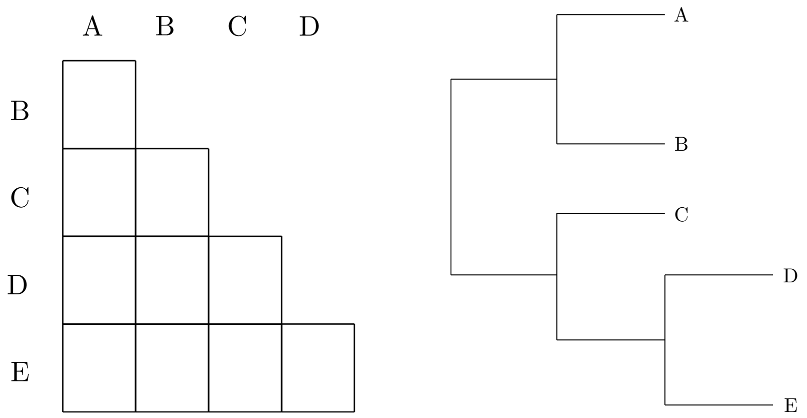
The root of the tree corresponds to the overall multiple sequence alignment. The basic algorithm uses one weight matrix and fixed gap opening and extension penalties.

However, this approach is not suitable for more divergent sequences. In this case, sequence weights are calculated from the guide tree. Closely related sequences have lower weights while the divergent ones have higher weights. Moreover, different substitution matrices are used at different alignment stages. The initial gap opening and gap extending penalties are given. New penalties are calculated based on the length of sequences, similarity of sequences, weight matrix, and existing gap positions [114, 128]. An example using *tat* and *vpu* proteins from HIV 1 (Human Immunodeficiency Virus) is shown in Figure 2.11. Table 2.1 presents the complexity of the ClustalW algorithm where n is the number of sequences and l is the average sequence length [62, 63].

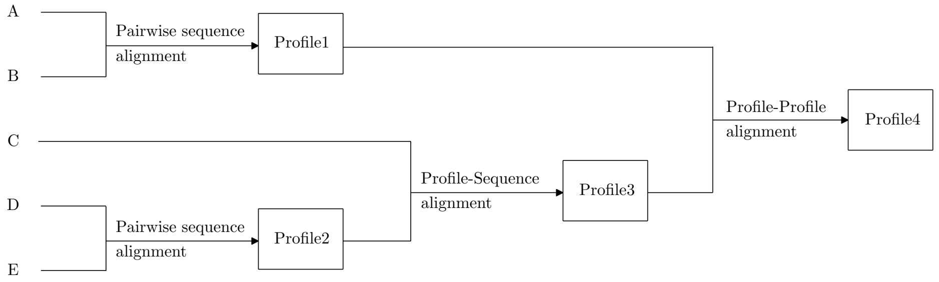
2.3 Tropical Algebra

The term *tropical* was given in honor of the Brazilian mathematician Imre Simon [105], who was one of the pioneers of *min-plus algebra*. It is employed in many optimization problems, e.g., Floyd-Warshall and Dijkstra algorithms and found extremely useful. The tropical algebra or min-plus algebra $(\mathbb{R} \cup \{\infty\}, \oplus, \odot)$ is a semiring that consists of the field of real numbers $\mathbb{R}$ together with an extra symbol ∞ representing infinity, and two arithmetic operations $\oplus$ and $\odot$, called addition and multiplication [88], defined as

$$x \oplus y = \min\{x, y\}, \quad x, y \in \mathbb{R} \cup \{\infty\}, \tag{2.11}$$



(a) Stage 1: Distance matrix (b) Stage 2: Guide tree


A
B
C
D
E
Pairwise sequence alignment
Profile1
Pairwise sequence alignment
Profile2
Profile-Sequence alignment
Profile3
Profile-Profile alignment
Profile4

(c) Stage 3: Progressive alignment

Fig. 2.10: Stages of the ClustalW algorithm. The first stage computes pairwise distance between sequences. The guide tree is built in stage two using the distance matrix. In stage three, the sequences are progressively aligned.

$$x \odot y = x + y, \quad x, y \in \mathbb{R} \cup \{\infty\}. \tag{2.12}$$

The tropical sum of two elements is their minimum and the tropical product of two elements is their ordinary sum. The neutral elements for addition and multiplication are ∞ and zero, respectively.

$$x \oplus \infty = x \text{ and } x \odot 0 = x, \quad x \in \mathbb{R} \cup \{\infty\}.$$

The additive and multiplicative inverses may not exist. For instance, there are no solutions for equations $5 \oplus x = 8$ and $\infty \odot x = 2$, $x \in \mathbb{R} \cup \{\infty\}$ [129]. The addition of vectors in the two-dimensional space over the tropical algebra is

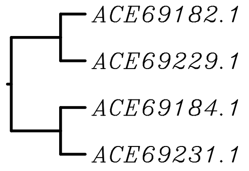
$$\begin{pmatrix} u_1 \\ u_2 \end{pmatrix} \oplus \begin{pmatrix} v_1 \\ v_2 \end{pmatrix} = \begin{pmatrix} u_1 \oplus v_1 \\ u_2 \oplus v_2 \end{pmatrix} = \begin{pmatrix} \min\{u_1, v_1\} \\ \min\{u_2, v_2\} \end{pmatrix}$$

and the multiplication of matrices is

$$\begin{aligned} \begin{pmatrix} u_{11} & u_{12} \\ u_{21} & u_{22} \end{pmatrix} \odot \begin{pmatrix} v_{11} & v_{12} \\ v_{21} & v_{22} \end{pmatrix} &= \begin{pmatrix} u_{11} \odot v_{11} \oplus u_{12} \odot v_{21} & u_{11} \odot v_{12} \oplus u_{12} \odot v_{22} \\ u_{21} \odot v_{11} \oplus u_{22} \odot v_{21} & u_{21} \odot v_{12} \oplus u_{22} \odot v_{22} \end{pmatrix} \\ &= \begin{pmatrix} \min\{u_{11} + v_{11}, u_{12} + v_{21}\} & \min\{u_{11} + v_{12}, u_{12} + v_{22}\} \\ \min\{u_{21} + v_{11}, u_{22} + v_{21}\} & \min\{u_{21} + v_{12}, u_{22} + v_{22}\} \end{pmatrix}. \end{aligned}$$

Sequence	ACE69182.1	ACE69229.1	ACE69184.1
ACE69229.1	15.38		
ACE69184.1	79.41	15.38	
ACE69231.1	15.38	80.77	15.38

(a) Distance matrix



(b) Rooted tree using UPGMA method

```
ACE69182.1      --------ESEGDQEELSALVEMGHHAPWDIDDL-------
ACE69229.1      --------ESDGDQEELSALVEMGDHAPLVINDL-------
ACE69184.1      PTSQPRGDQTGPKESEKKVERETATDQEDQWMDSYHLSGSI
ACE69231.1      PASQPRGDPTGPKESKKKVESETETDQGDQQMDS-------
                         :  .:.: ..  *   .      *
```

(c) Multiple alignment

Fig. 2.11: ClustalW based sequence alignment between the *tat* and *vpu* proteins from HIV 1 calculated from EMBL-EBI using the BLOSUM substitution matrix. The gap opening and the gap extension penalties for the full pairwise alignment are 10 and 0.1, respectively, and the initial gap penalty and the gap extension penalty for multiple alignments are 25 and 0.2, respectively.

These operations can be extended to matrices of arbitrary dimension. The identity matrix for the tropical matrix multiplication is

$$\boldsymbol{I} = \begin{pmatrix} 0 & \infty & \cdots & \infty \\ \infty & 0 & \ddots & \vdots \\ \vdots & \ddots & \ddots & \infty \\ \infty & \cdots & \infty & 0 \end{pmatrix}.$$

An antitone bijective mapping ($x \mapsto -\log x$) from the natural semiring ($\mathbb{R}_{\geq 0}, +, .$) onto the tropical semiring is called *tropicalization* of the natural semiring [129].

2.4 Hidden Markov Model

The *hidden Markov model* is a stochastic model widely used in areas such as speech recognition [91], computational biology [52], and pattern recognition [29]. The hidden Markov model represents the probability distribution of a sequence of observations and has two basic properties: the states that generated the observation sequence are hidden and the Markov process in which the future state depends only on the current state, not on the past states. In this thesis, we assume limited, discrete, and countable number of states.

The initial probability is the probability of an inital state. It is given by a vector $\boldsymbol{\pi}$ of length l, where l is the number of states.

$$\boldsymbol{\pi} = \begin{pmatrix} \pi_1, & \pi_2, & \ldots, & \pi_l \end{pmatrix}.$$

The *transition probability* is the probability to transit from one state to another and must be equal or greater than 0. The sum of all the probabilities from one state to all other states must be 1. The matrix $\boldsymbol{T}$ of size $l \times l$ holds the state transition probabilities, where l is the finite number of states.

$$\boldsymbol{T} = \begin{pmatrix} t_{1,1} & t_{1,2} & \dots & t_{1,l} \\ t_{2,1} & t_{2,2} & \dots & t_{2,l} \\ \vdots & \vdots & \ddots & \vdots \\ t_{l,1} & t_{l,2} & \dots & t_{l,l} \end{pmatrix}.$$

The probability to emit a particular symbol from a given state is called *emission probability*. The emission probability matrix $\boldsymbol{E}$ has size $l \times l^{'}$, where l is the number of states and $l^{'}$ is the number of output symbols. The row sums for the transition probability and emission probability matrices must be equal to 1.

$$\boldsymbol{E} = \begin{pmatrix} e_{1,1} & e_{1,2} & \dots & e_{1,l'} \\ e_{2,1} & e_{2,2} & \dots & e_{2,l'} \\ \vdots & \vdots & \ddots & \vdots \\ e_{l,1} & e_{l,2} & \dots & e_{l,l'} \end{pmatrix}.$$

The formal definition of hidden Markov mode is [23, 92]:

- Finite number of states l drawn from an alphabet $\Sigma = \{\sigma_1, \sigma_2, \dots, \sigma_l\}$;
- Number of observation symbols $l^{'}$ per state drawn from an alphabet $\Sigma^{'} = \{o_1, o_2, \dots, o_{l'}\}$;
- Length of observation sequence n;
- Observed sequence $\boldsymbol{\tau} = \tau_1, \dots, \tau_n \in \Sigma^{'^n}$;
- State transition probability matrix $\boldsymbol{T}$;
- Emission probability matrix $\boldsymbol{E}$;
- Initial probability vector $\boldsymbol{\pi}$.

The hidden Markov model assumes that the transition probability and emission probability matrices are *time invariant*. The large number of states are inefficient to use as they are difficult to analyze and a large transition matrix is required. An example of a hidden Markov model at time k is illustrated in Figure 2.12. Here, the states σ_i are hidden (shaded area) from the observations τ_i. The transition (shown as rectangles) from one states to the next one depends on the transition probability. At each step, the hidden Markov model selects a new state according to the transition probability and emits a symbol according to the emission probability. Some major applications of hidden Markov models in the field of bioinformatics are pairwise and multiple sequence alignment, gene finding, secondary structure prediction, and phylogenetic analysis [23, 25, 53, 65, 87, 103, 104, 113, 124].

Example 2.4.1. The *CpG islands* are regions of DNA characterized by a large number of adjacent CG nucleotides linked by the phosphodiester bonds. The CpG islands are typically 300-3,000 base pairs in length and found in approximately 40% of the promoter region of the mammalian genes (70% in the human promoter region). The

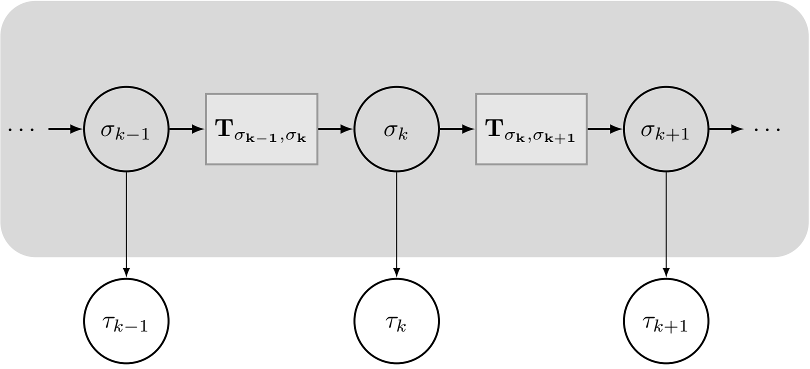


Fig. 2.12: Hidden Markov model at state σ_k.

methylation modifies C in a CpG pair i.e., `CH3`-group replaces the H-atom to mutate it into T. This methylation process is suppressed in a short region called CpG islands [7, 23].

The problem, whether a nucleotide comes from a CpG island or not, can be implemented by a hidden Markov model. For this, we have eight hidden states drawn from the alphabet $\Sigma = \{A+, C+, G+, T+, A-, C-, G-, T-\}$ where + represents the nucleotide emitted by the CpG island and - represents the nucleotide emitted by the non CpG island, four observed symbols $\Sigma' = \{A, C, G, T\}$, transition probability matrix, and emission probability matrix (Fig. 2.13). $\square$

The hidden Markov model addresses three basic problems [91, 92]:

- **Evaluation:**
  Given the hidden Markov model $M = (\boldsymbol{T}, \boldsymbol{E}, \boldsymbol{\pi})$ and the observation sequence $\boldsymbol{\tau} = \tau_1\tau_2 \dots \tau_n$, calculate the probability that the model M has generated the observation sequence $\boldsymbol{\tau}$.
- **Decoding:**
  Given the hidden Markov model $M = (\boldsymbol{T}, \boldsymbol{E}, \boldsymbol{\pi})$ and the observation sequence $\boldsymbol{\tau} = \tau_1\tau_2 \dots \tau_n$, calculate the most likely sequence of hidden states that produced this observation sequence $\boldsymbol{\tau}$.
- **Learning:**
  Given the training observation sequence $\boldsymbol{\tau} = \tau_1\tau_2 \dots \tau_n$ and the general structure of the HMM (numbers of hidden and visible states), determine the hidden Markov model parameters $M = (\boldsymbol{T}, \boldsymbol{E}, \boldsymbol{\pi})$ that best fit training data.

The decoding problem can be solved by the *Viterbi algorithm*. For the CpG island problem, the Viterbi algorithm is used to find the most likely sequence of states that generated the given sequence. In this work, we will focus on the Viterbi algorithm by discussing the parallel design suitable for the GPU architecture.

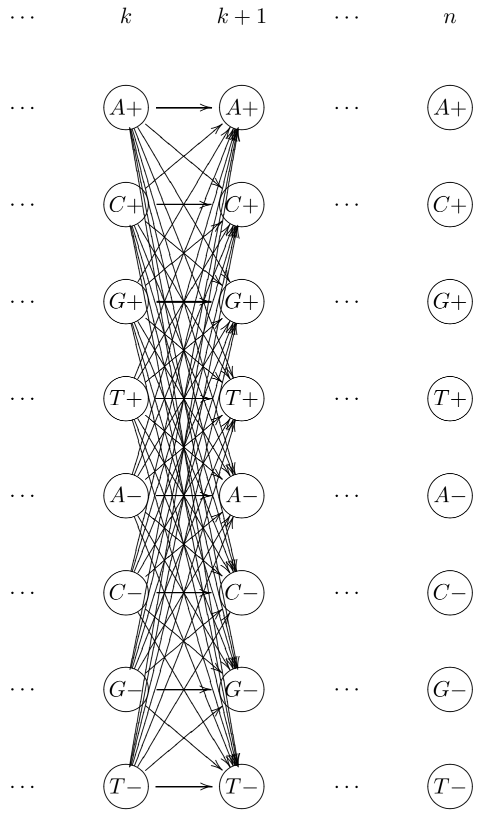


Fig. 2.13: The CpG island trellis diagram.

2.4.1 Viterbi Algorithm

Given a hidden Markov model and an observed sequence, deducing the sequence of (hidden) states that generated the observed sequence is an important task. There are several approaches which can predict the sequence of states that generated the observed sequence. The dynamic programming based Viterbi algorithm is the most widely used method to find the most probable sequence of (hidden) states that generated the observed sequence [31, 117]. Figure 2.14 shows an example of the most probable path for the CpG island problem for a given observed sequence.

The most probable path of hidden data that generated the observed sequence is the path with maximum posteriori probability [23, 92, 129].

$$\theta^{'} = \mathrm{argmax}_{\sigma}\{p_{\sigma,\tau}\}. \tag{2.13}$$

The Viterbi algorithm is given by

$$\begin{aligned}
M[0,\sigma] &= \pi_{\sigma},\ \sigma \in \Sigma, && (2.14)\\
M[k,\sigma] &= \max_{\sigma^{'}}(t_{\sigma^{'},\sigma} \cdot e_{\sigma^{'},\tau_k} \cdot M[k-1,\sigma^{'}]),\ 1 \leq k \leq n-1, && (2.15)\\
M[n,\sigma] &= e_{\sigma,\tau_n} \cdot M[n-1,\sigma],\ \sigma \in \Sigma, && (2.16)\\
p_{\tau} &= \max_{\sigma_n} M[n,\sigma_n]. && (2.17)
\end{aligned}$$

The value of $M[k,\sigma]$ is calculated using the previous values of $M[k-1,\sigma]$, the transition probability from the state $\sigma - 1$ to the state σ, and the probability of emitting the sequence τ_k. The last step computes the probability p_τ of the most probable path. This gives the algorithm VITERBI. The inputs are the observation sequence $\boldsymbol{\tau}$, the $l \times l$ transition probability matrix $\boldsymbol{T}$, and the $l \times l^{'}$ emission probability matrix $\boldsymbol{E}$. The algorithm fills in a dynamic programming matrix. The elements of the matrix are derived from one of the elements in previous row. This probability denotes the most probable path to generate the prefix of the observation sequence ending in the corresponding symbol τ_k.

Algorithm 2.5 VITERBI($\boldsymbol{\tau}, \boldsymbol{T}, \boldsymbol{E}, \boldsymbol{\pi}$)

```
Require: sequence $\boldsymbol{\tau} \in \Sigma^{'n}$, probabilities $\boldsymbol{T}$, $\boldsymbol{E}$, and $\boldsymbol{\pi}$
Ensure: term $p_\tau$
$\boldsymbol{M} \leftarrow \mathrm{matrix}[0 \dots n, 1 \dots l]$
for $\sigma \leftarrow 1$ to $l$ do
    $M[0, \sigma] \leftarrow \pi[\sigma]$
end for
for $k \leftarrow 1$ to $n - 1$ do
    for $\sigma \leftarrow 1$ to $l$ do
        $M[k, \sigma] \leftarrow 0$
        for $\sigma^{'} \leftarrow 1$ to $l$ do
            $M[k, \sigma] \leftarrow \max\{M[k, \sigma], T[\sigma^{'}, \sigma] \cdot E[\sigma^{'}, \tau_k] \cdot M[k-1, \sigma^{'}]\}$
        end for
    end for
end for
for $\sigma \leftarrow 1$ to $l$ do
    $M[n, \sigma] \leftarrow E[\sigma, \tau_n] \cdot M[n-1, \sigma]$
end for
$p_\tau \leftarrow 0$
for $\sigma \leftarrow 1$ to $l$ do
    $p_\tau \leftarrow \max\{p_\tau, M[n, \sigma]\}$
end for
```

The multiplication of many probabilities can yield very small numbers. To avoid this problem, the logarithmic workspace is used. Another advantage is that the computational expensive multiplication operation is replaced by addition. The maximum probability that generated the observed sequence can be evaluated using semiring homomorphism in Section 2.3. $\theta^{'}$ is evaluated in tropical algebra by putting $q_{\sigma,\tau} = -\log(p_{\sigma,\tau})$ [88, 129]

$$\theta^{'} = \mathrm{argmin}_\sigma\{q_{\sigma,\tau}\}. \tag{2.18}$$

For this, take the transition probability and emission probability matrices $\boldsymbol{T}^{'}$ and $\boldsymbol{E}^{'}$, where $t^{'}_{ij} = -\log(t_{ij})$ and $e^{'}_{ij} = -\log(e_{ij})$. The initial probability vector $\boldsymbol{\pi}^{'}$ is $\pi^{'}_i = -\log(\pi_i)$.

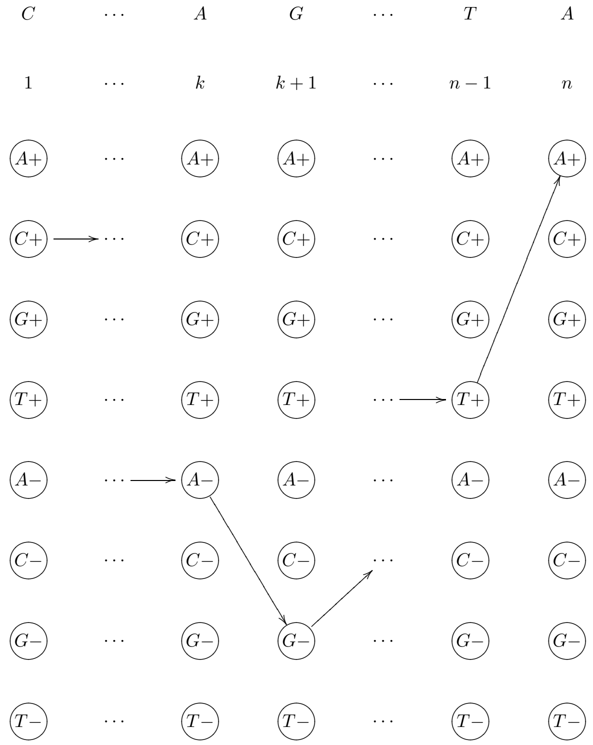


Fig. 2.14: The Viterbi algorithm shows an example of the most probable path for the CpG island problem.

The computation of tropicalized term q_τ for the Viterbi algorithm is [88, 129]

$$\begin{aligned}
M[0,\sigma] &= \pi'_{\sigma},\ \sigma \in \Sigma, && (2.19)\\
M[k,\sigma] &= \bigoplus_{\sigma'} (t'_{\sigma',\sigma} \odot e'_{\sigma',\tau_k} \odot M[k-1,\sigma']),\ 1 \le k \le n-1, && (2.20)\\
M[n,\sigma] &= e'_{\sigma,\tau_n} \odot M[n-1,\sigma],\ \sigma \in \Sigma, && (2.21)\\
q_\tau &= \bigoplus_{\sigma_n} M[n,\sigma_n]. && (2.22)
\end{aligned}$$

The $l \times l$ matrix $\boldsymbol{T}'$ is

$$\boldsymbol{T}' = -\log(\boldsymbol{T}) = \begin{pmatrix} -\log(t_{1,1}) & -\log(t_{1,2}) & \dots & -\log(t_{1,l}) \\ -\log(t_{2,1}) & -\log(t_{2,2}) & \dots & -\log(t_{2,l}) \\ \vdots & \vdots & \ddots & \vdots \\ -\log(t_{l,1}) & -\log(t_{l,2}) & \dots & -\log(t_{l,l}) \end{pmatrix},$$

the $l \times l'$ matrix $\boldsymbol{E}'$ is

$$\boldsymbol{E}' = -\log(\boldsymbol{E}) = \begin{pmatrix} -\log(e_{1,1}) & -\log(e_{1,2}) & \dots & -\log(e_{1,l'}) \\ -\log(e_{2,1}) & -\log(e_{2,2}) & \dots & -\log(e_{2,l'}) \\ \vdots & \vdots & \ddots & \vdots \\ -\log(e_{l,1}) & -\log(e_{l,2}) & \dots & -\log(e_{l,l'}) \end{pmatrix},$$

and the initial probability vector $\boldsymbol{\pi}^{'}$ is

$$\boldsymbol{\pi}^{'} = -\log(\boldsymbol{\pi}) = \begin{pmatrix} -\log(\pi_1) & -\log(\pi_2) & \dots & -\log(\pi_l) \end{pmatrix}.$$

This gives the algorithm TropViterbi. Its input is given by the observation sequence $\boldsymbol{\tau}$, the transition probability matrix $\boldsymbol{T}^{'}$, the emission probability matrix $\boldsymbol{E}^{'}$, and the initial probability vector $\boldsymbol{\pi}^{'}$. The algorithm initialize the first row of the matrix $\boldsymbol{M}$ with initial probability. Next, entry $M[k, \sigma]$ is computed by having minimum value at the current state using the transition and emission probabilities. Finally, the tropicalized term q_τ is calculated. The computational complexity of Viterbi and TropViterbi algorithms are $O(nl^2)$.

Algorithm 2.6 TropViterbi($\boldsymbol{\tau}, \boldsymbol{T}^{'}, \boldsymbol{E}^{'}, \boldsymbol{\pi}^{'}$)

```
Require: sequence $\boldsymbol{\tau} \in \Sigma'^n$, probabilities $\boldsymbol{T}^{'}$, $\boldsymbol{E}^{'}$, and $\boldsymbol{\pi}^{'}$
Ensure: tropicalized term $q_\tau$
$\boldsymbol{M} \leftarrow$ matrix$[0 \dots n, 1 \dots l]$
for $\sigma \leftarrow 1$ to $l$ do
    $M[0, \sigma] \leftarrow \pi^{'}[\sigma]$
end for
for $k \leftarrow 1$ to $n - 1$ do
    for $\sigma \leftarrow 1$ to $l$ do
        $M[k, \sigma] \leftarrow \infty$
        for $\sigma^{'} \leftarrow 1$ to $l$ do
            $M[k, \sigma] \leftarrow \min\{M[k, \sigma], T^{'}[\sigma^{'}, \sigma] + E^{'}[\sigma^{'}, \tau_k] + M[k-1, \sigma^{'}]\}$
        end for
    end for
end for
for $\sigma \leftarrow 1$ to $l$ do
    $M[n, \sigma] \leftarrow E^{'}[\sigma, \tau_n] + M[n-1, \sigma]$
end for
$q_\tau \leftarrow \infty$
for $\sigma \leftarrow 1$ to $l$ do
    $q_\tau \leftarrow \min\{q_\tau, M[n, \sigma]\}$
end for
```

Chapter 3

Graphics Processing Unit

The growth of microprocessor systems is hugely limited by the heat-dissipation and energy consumption factors. Due to these restrictions, the silicon based semiconductor industry has switched from single core to the multiple processing cores. This change enabled the design of parallel programs onto desktop computers. NVIDIA introduced the application of the graphics processor for general purpose computations that are traditionally treated by personal computers or workstations. A large set of problems in molecular dynamics, physics simulations, and scientific computing have been tackled by mapping them onto a GPU [78].

We will begin by outlining the design policy of the GPU. Then we will describe the programming model of the GPU. Finally, we will present the general GPU architecture and discuss the Fermi and Kepler architectures.

3.1 Design Policy of GPU

The fundamental difference between CPU and GPU is the design policy. The CPU is latency oriented while the GPU is throughput oriented. The CPU is designed to have sophisticated control logic and large cache memories to reduce instruction and data access latencies.

However, the GPU is used for data parallel applications which appeal high arithmetic intensity. Therefore, small cache memory and simple control logic is required to achieve high performance [51, 78]. The memory access latency can be minimized by overlapping computation and data transfer.

Both CPU and GPU should be used to achieve better performance i.e., CPU for the sequential part and GPU for the parallel part. The CUDA programming model introduced by NVIDIA supports the CPU/GPU execution.

3.2 Programming Model

Earlier, OpenGL and DirectX exploited the power of GPU for many data parallel algorithms to achieve huge speed-up. However, this approach has several drawbacks [77]:

- Random memory reads and writes are not supported.

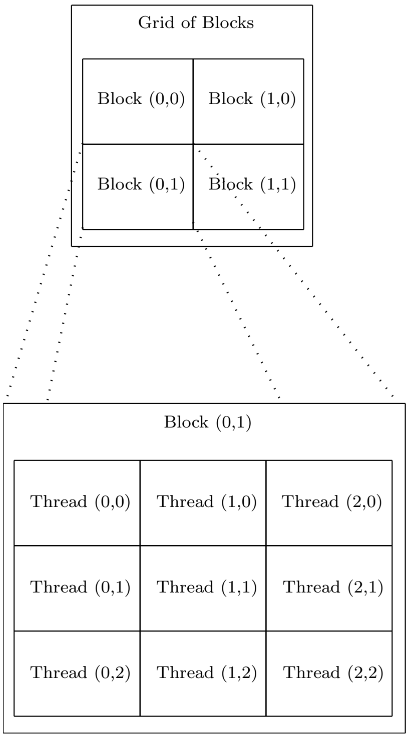


Fig. 3.1: Grid of thread blocks.

- Programmers need to have knowledge of the graphics API.
- Problems need to be expressed in terms of vertex coordinates, textures, and shader programs.

For this, NVIDIA introduced the CUDA programming model which enables the programmer to write `C` – like functions called kernels with some extensions. Each kernel is executed by a batch of parallel threads. CUDA provides three key abstractions: a hierarchy of thread groups, shared memories, and barrier synchronization [78].

3.2.1 Thread Hierarchy

CUDA enables programmers to generate lightweight threads grouped into blocks that are executed in parallel. Each block can be organized as a one-, two-, or three-dimensional array of threads. The maximum number of threads per block is limited by the hardware architecture [78].

Each block is assigned to only one streaming multiprocessor running until completion without pre-emption. A grid of size 2×2 is shown in Figure 3.1, where each block has 3×3 threads. A kernel launch with this configuration can execute maximum 36 parallel threads. The number of thread blocks in a grid usually depends on the size of data or the number of processors. The blocks in a kernel are subject to a scheduler in order to assign them to the streaming multiprocessors. The thread blocks are executed in serial or parallel manner which enables to schedule them in any order requiring special attention of the programmers about block dependencies. The threads belonging to different blocks within a kernel cannot communicate during the execution [78].

3.2.2 Memory Model

The data need to be transferred into the GPU memory in order to be processed. CUDA memory size is a limiting factor to achieve good performance. CUDA provides access to several types of memory that helps the programmer to develop efficient parallel programs (Fig. 3.2). The choice to use memory type typically depends on the factors such as address space, scope, lifetime, and access latency [78].

The on-chip memories such as registers and shared memory can be accessed at very high speed. The registers are allocated to the individual threads and have lifetime of thread execution. The private variables to each thread are typically stored in the registers. The threads within the thread block can share data using shared memory and can read from and write to the shared memory within the kernel. The shared memory has lifetime of a block and cannot be accessed after the block finishes its execution. The size of shared memory is a limiting factor. Synchronization mechanism should be provided to avoid concurrent read and write problems [78].

The variables declared in the global scope are stored in global memory and have the lifetime of a whole application. The threads within a grid can read from and write to the global memory. The data can be transferred directly from host memory to the global memory. The size of the global memory is much larger than the shared memory. The global memory access latency is very high (almost 100 times slower than the shared memory). Therefore, the number of accesses to global memory should be reduced within the kernel [51, 78].

The constant variables should be declared in the global scope and have the lifetime of application. It can be accessed by all threads and cannot be written from within the kernel. The access latency of the constant memory is faster than the global memory because it is cached but there is only limited amount of the constant memory available [78].

The texture memory is a read-only memory that resides in the device memory and is cached in the texture cache. The process of reading a texture is called a texture fetch. The texture, which is a piece of the texture memory that is fetched, can be a one-, two-, or three-dimensional array where the elements of the arrays are called texels [78].

3.2.3 Thread Synchronization

In order to avoid the concurrent read/write problem for shared data, CUDA provides a synchronization mechanism. To share the data by means of shared memory among threads within a thread block, use `syncthreads()`, which is for synchronization purposes. For the threads belonging to different thread blocks, data must be shared by virtue of the global memory using separate kernel invocations. However, the performance is degraded by the additional kernel invocation and global memory traffic [78].

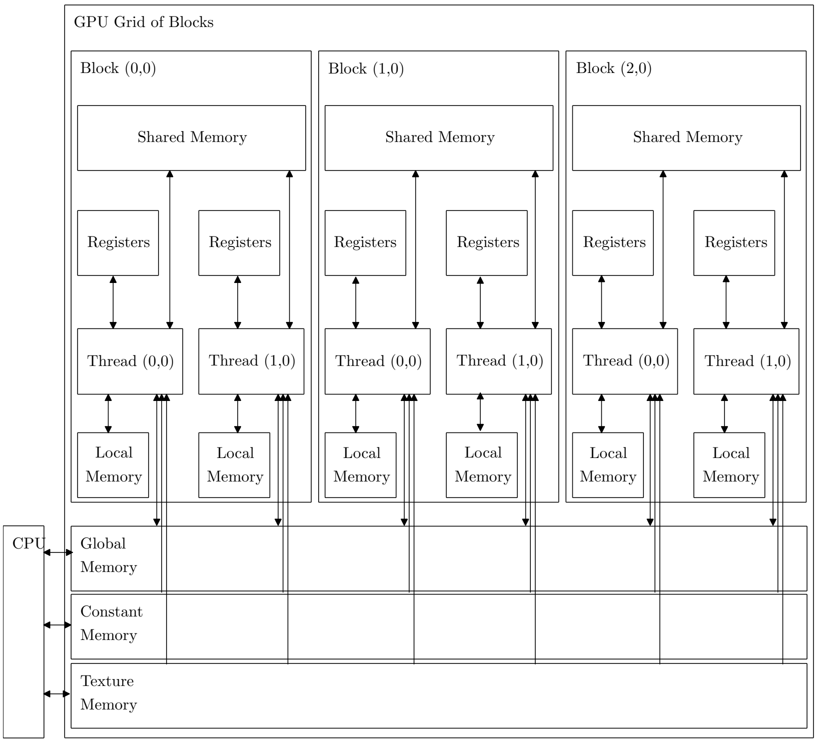


Fig. 3.2: CUDA memory model.

The efficiency of GPU programs depends on the utilization of the allocated hardware, hardware configuration (number of threads, number of blocks, and memory type), and the amount of parallelism exhibited by the problem [78, 90].

3.2.4 CUBLAS Library

CUDA also provides *Compute Unified Basic Linear Algebra Subprograms* (CUBLAS) library [79, 81] which is an implementation of *Basic Linear Algebra Subprograms* (BLAS). It enables the programmer to use the GPU without direct operation of the CUDA drivers. In order to utilize the optimized library, vectors and matrices should be created and filled in the device memory, invoking the required CUBLAS routines, and transferring the results in host memory.

The CUBLAS library provides helper functions and BLAS routines. Helper functions are used to move the data to or from the GPU memory. There are also methods to create and destroy objects. BLAS functions are organized in three levels i.e., vector-vector, matrix-vector, and matrix-matrix operations. The CUBLAS library uses the column-major storage with 1-based indexing.

3.3 GPU Hardware Architecture

The NVIDIA GPU consists of *Streaming Multiprocessors* (SMs) each of which consists of many *Streaming Processors* (SPs). The thread blocks are allocated to multiprocessors with the available execution capacity. Multiple thread blocks can execute concurrently on one SM and new blocks are allocated on the SM after some blocks have finished their execution. However, one thread block is executed only on one allocated SM [78].

A multiprocessor works in the SIMT manner to execute hundreds of threads concurrently which helps to write thread level parallel code for independent threads as well as data parallel code for coordinated threads. The threads are executed in groups of 32 parallel threads called *warps* [78]. Next, we discuss briefly the Fermi and Kepler based GPU hardware architectures.

3.3.1 Fermi Architecture

The first Fermi based GPU features up to 512 CUDA cores which are organized in 16 SMs of 32 cores each. Floating point and integer instructions are exectuted on a CUDA core per clock for a thread. GPU is connected to CPU via PCI-Express [77].

A Fermi based streaming multiprocessor architecture is illustrated in Figure 3.3. Each SM features 32 CUDA processors having a fully pipelined integer *Arithmetic Logic Unit* (ALU) and *Floating Point Unit* (FPU) and 16 load/store units (LD/ST), allowing source and destination addresses to be calculated for 16 threads per clock. *Special Function Units* (SFUs) execute transcendental instructions such as sin, cosine, reciprocal, and square root. Each SM features two warp schedulers and two instruction dispatch units, allowing two warps to be issued and executed concurrently. The Fermi architecture also allows a 64 KB configurable shared memory and L1 cache which can be adjusted according to the problem nature [77].

3.3.2 Kepler Architecture

The next generation CUDA compute architecture called *Kepler* provides the capability to invoke the kernel or other library routines such as BLAS functions within the kernel. The Kepler architecture also provide a feature called *Hyper-Q* which enables multiple CPU cores to launch work on a single GPU by increasing the total number of work queues (32) between host and GPU. This will allow efficient hardware utilization [82].

Kepler GK110 is designed for power efficiency to provide high performance per Watt. A full Kepler GK110 implementation includes 15 *Streaming Multiprocessor* (SMX) units each with 192 single precision CUDA cores and six 64 bit memory controllers [82].

The Kepler GK110 SMX features 192 single precision CUDA cores and 64 double precision math units (DP unit). Each core has fully pipelined floating point and integer arithmetic logic units (Fig. 3.4). Each SMX also has 32 load/store units (LD/ST) and 32 *Special Function Units* (SFU). Four warp schedulers and eight instruction dispatch units allow four concurrent warps executions. The memory hierarchy of the Kepler architecture is similar to the Fermi architecture except it allows 32KB / 32KB split between the shared memory and L1 cache and 48 KB read-only data cache [82]. Table 3.1 shows a comparison between Fermi and Kepler architectures.

Table 3.1: Comparison of Fermi and Kepler architectures [82].

	Fermi GF104	**Kepler GK110**
compute capability	2.1	3.5
threads/warp	32	32
max warps/multiprocessor	48	64
max threads/multiprocessor	1536	2048
max thread blocks/multiprocessor	8	16
32bit registers/multiprocessor	32768	65536
max registers/thread	63	255
max threads/thread block	1024	1024
max x grid dimension	$2^{16} - 1$	$2^{32} - 1$

Table 3.2: NVIDIA GeForce GTX 560 Ti properties.

CUDA version	4.0
compute capability	2.1
multiprocessors	8
CUDA cores per multiprocessor	48
GPU clock speed	1.8 GHz
constant memory	64 KB
global memory	1024 MB
shared memory per block	48 KB
registers per block	32768
maximum number of threads per block	1024
maximum sizes of each dimension of a block	1024 x 1024 x 64
maximum sizes of each dimension of a grid	65535 x 65535 x 65535

3.4 Hardware

The hardware and software used for implementing the parallel algorithms is an Intel Core 2 Duo 6600 CPU (2.40 GHz) running openSUSE 11.4 linux distribution using the Intel Math Kernel Library (MKL) 10.3, and the CUDA version 4.0 on an NVIDIA GeForce GTX 560 Ti graphics card. The tests are conducted using a serial gcc compiler (version 4.4.1) and an NVIDIA nvcc compiler. Table 3.2 provides the key features of NVIDIA GeForce GTX 560 Ti.

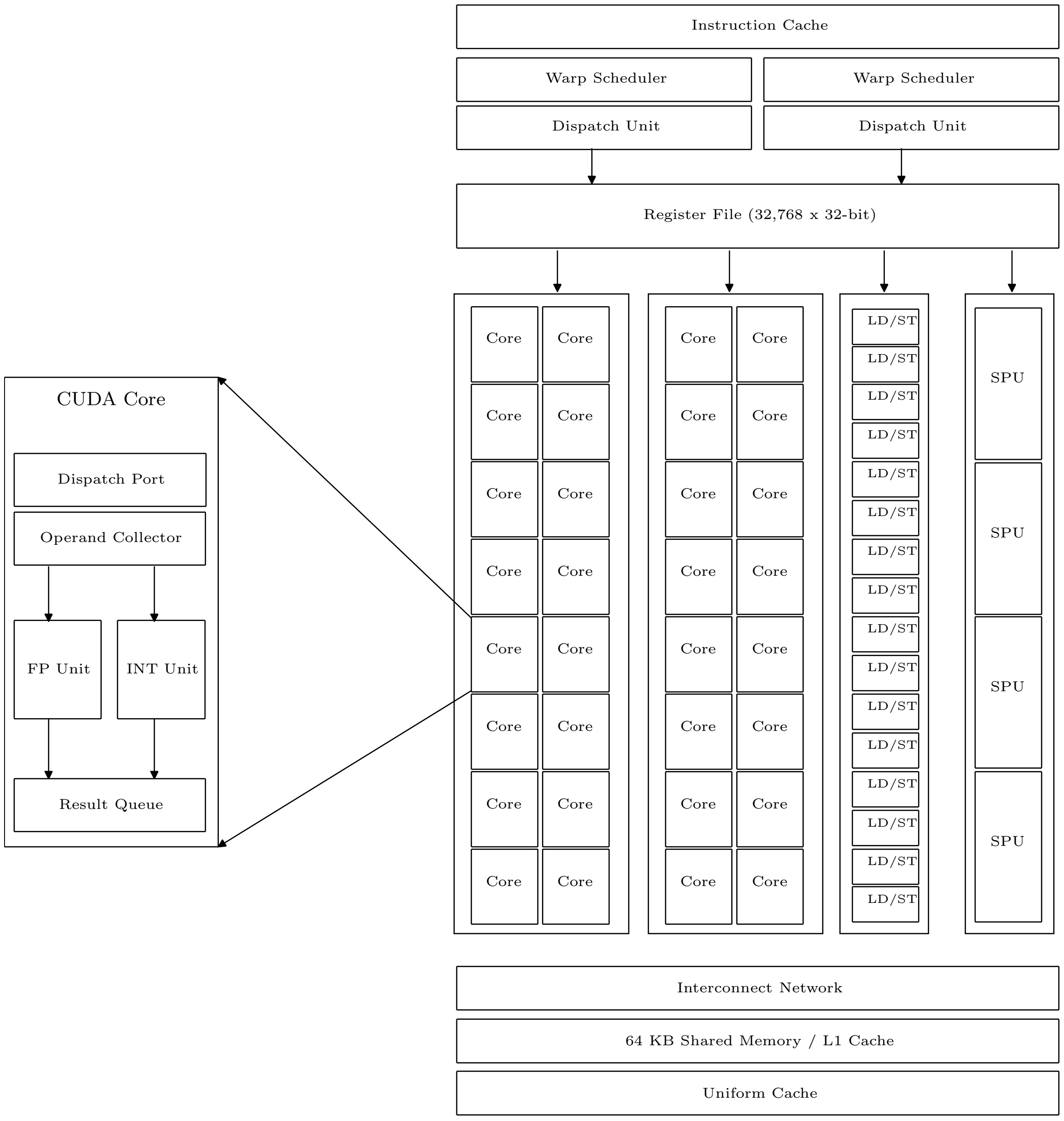


Fig. 3.3: Fermi streaming multiprocessor architecture [77].

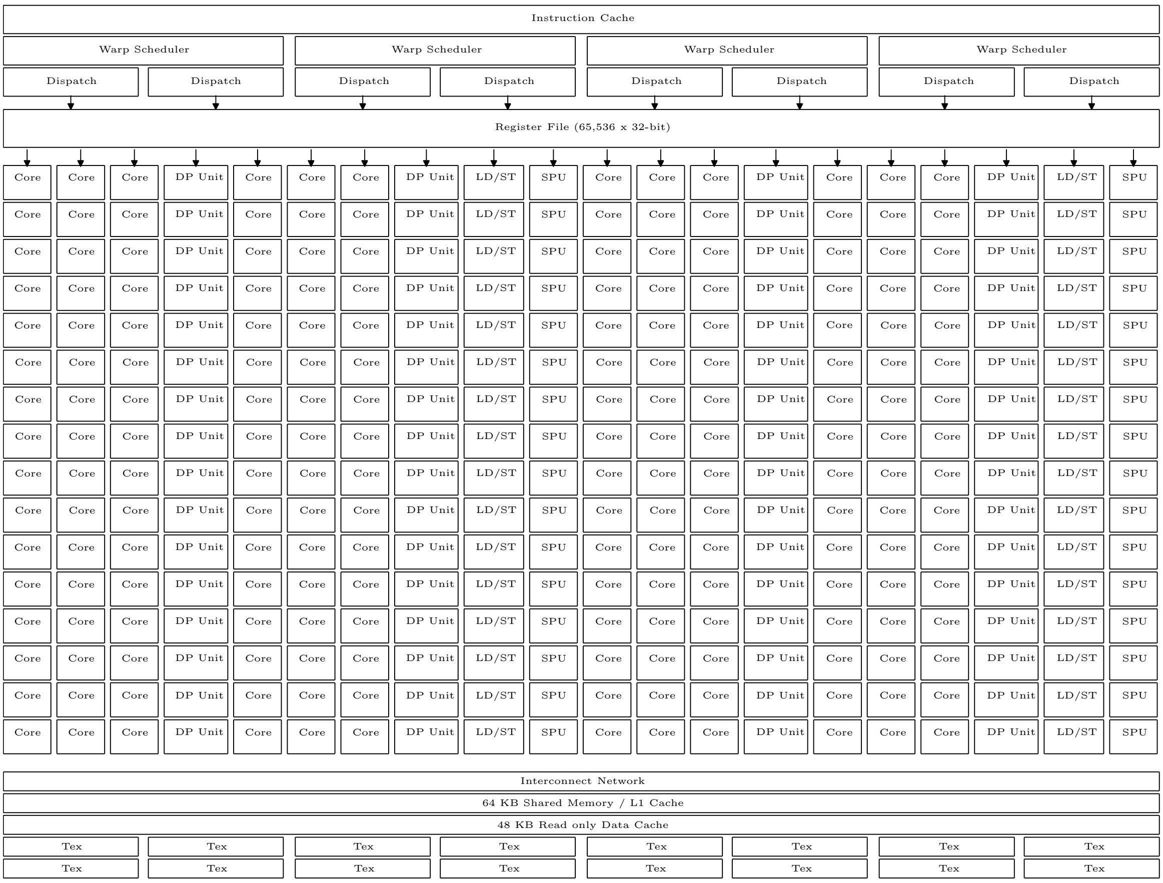


Fig. 3.4: Kepler GK110 streaming multiprocessor (SMX) architecture [82].

Chapter 4

Sequence Alignment

Alignment is the basic operation in molecular biology for comparing sequences. It provides a means to arrange biomolecular sequences in order to identify regions of similarity that may be a consequence of functional, structural, or evolutionary relatedness. Although, the heuristic methods are faster but they are still costly due to rapid increase in sequence databases. Therefore, a variety of techniques are employed to parallelize heuristic methods.

In this chapter, we will present different strategies to implement progressive sequence alignment on the GPU. We will investigate the profile-profile alignment and profile-sequence alignment by using the matrix-matrix product and the wavefront methods. We will conclude with a parallelization technique to map the progressive alignment stage of the ClustalW algorithm on the GPU.

4.1 Related Work

In recent years, there were several attempts to improve the execution speed of sequence alignment algorithms using GPUs. Manavski et al. [67] and Munekawa et al. [71] have accelerated the Smith-Waterman algorithm on a GPU gaining moderate performance boosts. The methods to reduce the amount of data transfer and data fetches help further to increase the speed-up. Schatz et al. [100] have provided an implementation of a local sequence alignment algorithm (MUMmer) on a GPU attaining a 10-fold speed-up over a serial CPU version. Similarly, Dzivi [90] has implemented the Needleman-Wunsch algorithm and gained performance peaks of an 80-fold speed-up. All these algorithms follow the wavefront approach utilizing the fact that the anti-diagonals in the forward table are independent of each other. Xiao et al. [126] used a fine-grained parallelization strategy by distributing the tasks of a single problem across all threads on the GPU. These are just few examples of the work which is done to parallelize the sequence alignment on the GPU architecture.

Several parallel algorithms have been developed to improve the efficiency of multiple sequence alignment [3, 19, 46, 62, 64]. However, most of these methods discuss the profile-profile and profile-sequence alignment as part of progressive sequence alignment. There is hardly any GPU based implementation that solely focuses the profile-profile alignment.

The conversion of the alignment problem into a form that matches the vector-processing architecture of GPU can result in a huge performance boost. Recently, Bassoy et al. [5] formulated a matrix-matrix product algorithm by separating the profile-sequence alignment algorithm into a data dependent and a data independent part to attain an order of magnitude speed-up on an GPU. However, they have ignored the time taken by executing the data dependent part on the CPU which is the reason for their huge speed-up given. In this work, profile-profile alignment problem will be transformed into matrix-matrix product to utilize the parallel GPU architecture. Moreover, the space requirements for the matrix-matrix product based profile-sequence alignment algorithm presented by Bassoy et al. [5] will be improved to process sequences of larger length.

Several efforts have been made to accelerate the performance of the ClustalW algorithm. ClustalW-MPI [57], Ebedes et al. [24], and pCLUSTAL [14] use MPI to parallelize ClustalW on a cluster. ClustalW-MPI parallelizes all three stages and achieves approximately 4.3 speed-up using 16 processors. Ebedes et al. demonstrate speed-up of 5.5 by parallelizing the stages one and three. Similarly, Tan et al. [111] use MPI/OpenMP for symmetric multiprocessors to parallelize the stages one and three. Mikhailov et al. [69] show a 10-fold speed-up by parallelizing all three stages with OpenMP on a shared-memory SGI machine. Oliver et al. [86] map stage one on FPGA and attain speed-up between 45 and 50. MT-ClustalW [10] utilize pthreads to parallelize all three stages. GPU-ClustalW [61] parallelize the first stage on a GPU with OpenGL to obtain approximately 7 speed-up. MSA-CUDA [62] exploits the parallel architecture of the GPU by implementing all three stages and achieve maximum average speed-up of approximately 37 for a small number of long sequences. In this chapter, a combination of matrix-matrix product and wavefront methods will be used to parallelize the progressive alignment stage of ClustalW.

4.2 Profile-Profile Alignment

The operation to align two existing alignments is called profile-profile alignment. Profile-profile alignment is better to detect distantly related proteins and provide better alignment accuracy when compared with profile-sequence alignment [26]. There are many strategies to implement profile-profile alignment depending on scoring function, gap-penalties, and alignment method [84].

In this section, we will formulate parallel solutions for profile-profile alignment on the GPU. It is straight-forward to design parallel algorithms when there is no data dependency between elements. However, the algorithm ProfProfAlign shows that the cell $S_{i,j}$ in the interior of the forward table can only be computed if the neighboring entries $S_{i-1,j}$, $S_{i,j-1}$, and $S_{i-1,j-1}$ are already known. The entries of the forward table depend on one or three previous entries (Fig. 4.1). These data dependent elements impact the algorithm design and performance. In the following, we will present two approaches that fit to the parallel architecture of the GPU.

4.2.1 Matrix Approach

The algorithm ProfProfAlign will be redesigned and implemented onto GPU by separating the data independent and data dependent parts. The score of a profile-

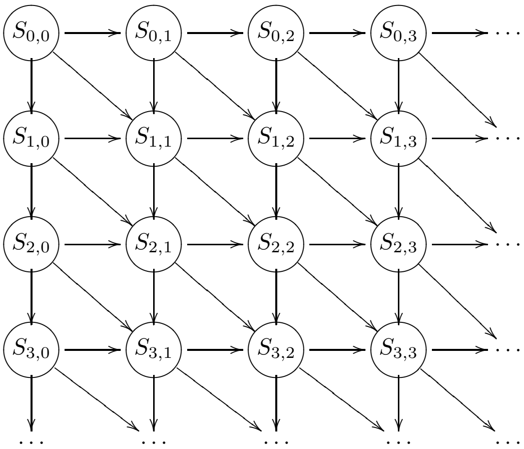


Fig. 4.1: Data dependencies in the forward table.

profile alignment $(\mathbf{P}', \boldsymbol{Q}')$ defined by Eqn. (2.7) can also be calculated using the squared Euclidean distance which amounts to a scalar product,

$$d(\boldsymbol{p}, \boldsymbol{q})^2 = (\boldsymbol{p} - \boldsymbol{q})^T \cdot (\boldsymbol{p} - \boldsymbol{q}). \tag{4.1}$$

First, the data independent part is implemented on GPU by calculating Euclidean distances $d(\boldsymbol{p}_i, -_p)$, $d(-_p, \boldsymbol{q}_j)$, and $d(\boldsymbol{p}_i, \boldsymbol{q}_j)$ making use of scalar products. These values are pre-stored in three $m \times n$ matrices $\boldsymbol{V} = (V_{ij})$, $\boldsymbol{H} = (H_{ij})$, and $\boldsymbol{D} = (D_{ij})$. Second, the data dependent part uses these matrices to compute the entries of the forward table. This part requires to take the minimum of three values and is implemented onto CPU. This gives the algorithm ProfProfAlignScalProd.

The algorithm ProfProfAlignScalProd can be reformulated by using matrix multiplications. For this, define the $l \times m$ matrix

$$\mathbf{P}_- = (-_p, \ldots, -_p) \tag{4.2}$$

and the $l \times n$ matrix

$$\boldsymbol{Q}_- = (-_p, \ldots, -_p)\,, \tag{4.3}$$

and form the $l \times n$ matrix

$$\begin{aligned} \boldsymbol{H}_1 &= \boldsymbol{Q}_- - \boldsymbol{Q} \\ &= (-_p - \boldsymbol{q}_1, \ldots, -_p - \boldsymbol{q}_n)\,, \end{aligned} \tag{4.4}$$

the $l \times m$ matrix

$$\begin{aligned} \boldsymbol{V}_1 &= \mathbf{P} - \mathbf{P}_- \\ &= (\boldsymbol{p}_1 - -_p, \ldots, \boldsymbol{p}_m - -_p)\,, \end{aligned} \tag{4.5}$$

and the $l \times (m \cdot n)$ matrix

$$\boldsymbol{D}_1 = (\boldsymbol{p}_1 - \boldsymbol{q}_1, \ldots, \boldsymbol{p}_1 - \boldsymbol{q}_n, \ldots, \boldsymbol{p}_m - \boldsymbol{q}_1, \ldots, \boldsymbol{p}_m - \boldsymbol{q}_n)\,. \tag{4.6}$$

Algorithm 4.1 ProfProfAlignScalProd($\mathbf{P}, \boldsymbol{Q}$)

```
Require: two profiles $\mathbf{P} = \boldsymbol{p}_1, \ldots, \boldsymbol{p}_m$ and $\boldsymbol{Q} = \boldsymbol{q}_1, \ldots, \boldsymbol{q}_n$
Ensure: $\boldsymbol{S} = (S_{ij})$ forward matrix
$S_{0,0} \leftarrow 0$                                        // initialization
for $i \leftarrow 1$ to $m$ do
    $S_{i,0} \leftarrow \sum_{k=1}^{i} d(\boldsymbol{p}_k, -_p)$
end for
for $j \leftarrow 1$ to $n$ do
    $S_{0,j} \leftarrow \sum_{k=1}^{j} d(-_p, \boldsymbol{q}_k)$
end for
for $i \leftarrow 1$ to $m$ do                                // calculation
    for $j \leftarrow 1$ to $n$ do
        $V_{i,j} \leftarrow d(\boldsymbol{p}_i, -_p)$
        $H_{i,j} \leftarrow d(-_p, \boldsymbol{q}_j)$
        $D_{i,j} \leftarrow d(\boldsymbol{p}_i, \boldsymbol{q}_j)$
    end for
end for
for $i \leftarrow 1$ to $m$ do                                // minimization
    for $j \leftarrow 1$ to $n$ do
        $S_{i,j} \leftarrow \min\{S_{i-1,j} + V_{ij}, S_{i,j-1} + H_{ij}, S_{i-1,j-1} + D_{ij}\}$
    end for
end for
return $\boldsymbol{S}$
```

Then the squared Euclidean distances of the entries in the matrix $\boldsymbol{H}$ can be calculated by multiplying the newly formed matrix $\boldsymbol{H}_1$ with its transpose and extracting the diagonal entries. This similarly holds for the matrices $\boldsymbol{V}$ and $\boldsymbol{D}$.

$$\boldsymbol{H} = \mathrm{diag}[\boldsymbol{H}_1^T \cdot \boldsymbol{H}_1] = \big(d(-_p - \boldsymbol{q}_j)^2\big)_j\,, \tag{4.7}$$

$$\boldsymbol{V} = \mathrm{diag}[\boldsymbol{V}_1^T \cdot \boldsymbol{V}_1] = \big(d(\boldsymbol{p}_i - -_p)^2\big)_i\,, \tag{4.8}$$

$$\boldsymbol{D} = \mathrm{diag}[\boldsymbol{D}_1^T \cdot \boldsymbol{D}_1] = \big(d(\boldsymbol{p}_i - \boldsymbol{q}_j)^2\big)_{i,j}\,. \tag{4.9}$$

Theorem 4.1. *The matrix identity in (4.7) holds.*

Proof. Take the $l \times n$ matrix

$$\boldsymbol{H}_1 \quad = \quad \boldsymbol{Q}_- - \boldsymbol{Q} = (-_p - \boldsymbol{q}_1, \ldots, -_p - \boldsymbol{q}_n)\,,$$

where $\boldsymbol{Q}_-$ is the $l \times n$ matrix defined in (4.3). Expanding the matrix $\boldsymbol{H}$ in (4.7) gives

$$\begin{aligned}
\boldsymbol{H} &= \mathrm{diag}[\boldsymbol{H}_1^T \cdot \boldsymbol{H}_1] \\
&= \mathrm{diag}\left[\begin{pmatrix} -_{p_1} - q_{1,1} & \cdots & -_{p_l} - q_{1,l} \\ \vdots & \ddots & \vdots \\ -_{p_1} - q_{n,1} & \cdots & -_{p_l} - q_{n,l} \end{pmatrix} \cdot \begin{pmatrix} -_{p_1} - q_{1,1} & \cdots & -_{p_1} - q_{n,1} \\ \vdots & \ddots & \vdots \\ -_{p_l} - q_{1,l} & \cdots & -_{p_l} - q_{n,l} \end{pmatrix}\right] \\
&= \mathrm{diag}\begin{bmatrix} (-_{p_1} - q_{1,1})^2 + \ldots + (-_{p_l} - q_{1,l})^2 & \cdots & \\ \vdots & \ddots & \vdots \\ & \cdots & (-_{p_1} - q_{n,1})^2 + \ldots + (-_{p_l} - q_{n,l})^2 \end{bmatrix} \\
&= \mathrm{diag}\begin{bmatrix} d(-_p - \boldsymbol{q}_1)^2 & \cdots & \\ \vdots & \ddots & \vdots \\ & \cdots & d(-_p - \boldsymbol{q}_n)^2 \end{bmatrix}.
\end{aligned}$$

Taking diagonal entries, we obtain the squared Euclidean distances

$$\boldsymbol{H} \quad = \quad (d(-_p - \boldsymbol{q}_j)^2)_j.$$

□

Theorem 4.2. *The matrix identity in (4.8) is valid.*

The proof is similar to that of Theorem 4.1.

Theorem 4.3. *The matrix identity in (4.9) holds.*

Proof. Pick the $l \times (m \cdot n)$ matrix

$$\boldsymbol{D}_1 \quad = \quad (\boldsymbol{p}_1 - \boldsymbol{q}_1, \ldots, \boldsymbol{p}_1 - \boldsymbol{q}_n, \ldots, \boldsymbol{p}_m - \boldsymbol{q}_1, \ldots, \boldsymbol{p}_m - \boldsymbol{q}_n)\,.$$

By (4.9), we obtain

$$\begin{aligned}
\boldsymbol{D} &= \mathrm{diag}[\boldsymbol{D}_1^T \cdot \boldsymbol{D}_1] \\
&= \mathrm{diag}\left[\begin{pmatrix} p_{1,1} - q_{1,1} & \cdots & p_{1,l} - q_{1,l} \\ & \vdots & \\ p_{m,1} - q_{n,1} & \cdots & p_{m,l} - q_{n,l} \end{pmatrix} \cdot \begin{pmatrix} p_{1,1} - q_{1,1} & \cdots & p_{m,1} - q_{n,1} \\ & \vdots & \\ p_{1,l} - q_{1,l} & \cdots & p_{m,l} - q_{n,l} \end{pmatrix}\right] \\
&= \mathrm{diag}\begin{bmatrix} d(\boldsymbol{p}_1 - \boldsymbol{q}_1)^2 & \cdots & \\ \vdots & \ddots & \vdots \\ & \vdots & d(\boldsymbol{p}_m - \boldsymbol{q}_n)^2 \end{bmatrix}.
\end{aligned}$$

By taking diagonal entries, we obtain the squared Euclidean distances given by

$$\boldsymbol{D} \quad = \quad (d(\boldsymbol{p}_i - \boldsymbol{q}_j)^2)_{i,j}.$$

□

These identities give rise to algorithm ProfProfAlignMatProd [39].

Algorithm 4.2 ProfProfAlignMatProd($\mathbf{P}, \boldsymbol{Q}$)

```
Require: two profiles $\mathbf{P} = \boldsymbol{p}_1, \ldots, \boldsymbol{p}_m$ and $\boldsymbol{Q} = \boldsymbol{q}_1, \ldots, \boldsymbol{q}_n$
Ensure: $\boldsymbol{S} = (S_{ij})$ forward matrix
$S_{0,0} \leftarrow 0$                                   // initialization
for $i \leftarrow 1$ to $m$ do
    $S_{i,0} \leftarrow \sum_{k=1}^{i} d(\boldsymbol{p}_k, -_p)$
end for
for $j \leftarrow 1$ to $n$ do
    $S_{0,j} \leftarrow \sum_{k=1}^{j} d(-_p, \boldsymbol{q}_k)$
end for
$\boldsymbol{H}_1 \leftarrow \boldsymbol{Q}_- - \boldsymbol{Q}$                  // calculation
$\boldsymbol{V}_1 \leftarrow \mathbf{P} - \mathbf{P}_-$
$\boldsymbol{D}_1 \leftarrow \left(\boldsymbol{p}_i - \boldsymbol{q}_j\right)_{ij}$
$\boldsymbol{H} \leftarrow \mathrm{diag}[\boldsymbol{H}_1^T \cdot \boldsymbol{H}_1]$
$\boldsymbol{V} \leftarrow \mathrm{diag}[\boldsymbol{V}_1^T \cdot \boldsymbol{V}_1]$
$\boldsymbol{D} \leftarrow \mathrm{diag}[\boldsymbol{D}_1^T \cdot \boldsymbol{D}_1]$
for $i \leftarrow 1$ to $m$ do                      // minimization
    for $j \leftarrow 1$ to $n$ do
        $S_{i,j} \leftarrow \min\{S_{i-1,j} + V_i, S_{i,j-1} + H_j, S_{i-1,j-1} + D_{ij}\}$
    end for
end for
return $\boldsymbol{S}$
```

4.2.2 Wavefront Approach

The entries of the forward table depend on one or three previous entries (Fig. 4.1). The cells can be filled column by column, row by row, or anti-diagonal by anti-diagonal. The first two approaches limit the number of cells that can be simultaneously calculated, since entries in one column depend on other entries in the same or previous columns. The situation is similar for rows.

However, an anti-diagonal consists of all cells S_{ij} such that $i+j$ is constant. Therefore, the elements on the same anti-diagonal are independent of each other and only depend on the previous two anti-diagonals. The anti-diagonals of the 6×6 forward table are shown in Figure 4.2. In order to calculate value for $S_{2,2}$, we require the cells $S_{1,1}$, $S_{1,2}$, and $S_{2,1}$. Therefore, we need the previous two anti-diagonals i.e., $k-1$ and $k-2$. The approach of calculating all entries in each anti-diagonal at once is called wavefront method [54]. Most of the GPU implementations of sequence alignment follow this paradigm [13, 67, 71, 90, 100].

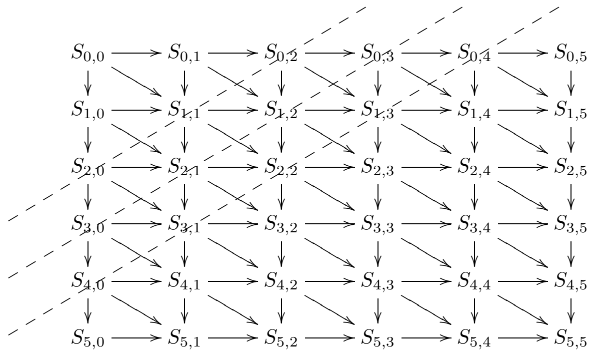

$S_{0,0}$ $S_{0,1}$ $S_{0,2}$ $S_{0,3}$ $S_{0,4}$ $S_{0,5}$
$S_{1,0}$ $S_{1,1}$ $S_{1,2}$ $S_{1,3}$ $S_{1,4}$ $S_{1,5}$
$S_{2,0}$ $S_{2,1}$ $S_{2,2}$ $S_{2,3}$ $S_{2,4}$ $S_{2,5}$
$S_{3,0}$ $S_{3,1}$ $S_{3,2}$ $S_{3,3}$ $S_{3,4}$ $S_{3,5}$
$S_{4,0}$ $S_{4,1}$ $S_{4,2}$ $S_{4,3}$ $S_{4,4}$ $S_{4,5}$
$S_{5,0}$ $S_{5,1}$ $S_{5,2}$ $S_{5,3}$ $S_{5,4}$ $S_{5,5}$

Fig. 4.2: The forward table for profile-profile alignment showing the anti-diagonals.

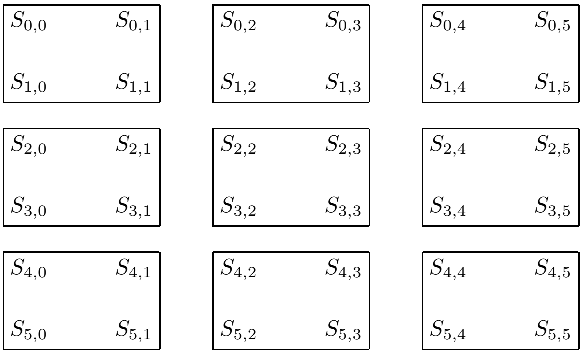

$S_{0,0}$ $S_{0,1}$ $S_{1,0}$ $S_{1,1}$
$S_{0,2}$ $S_{0,3}$ $S_{1,2}$ $S_{1,3}$
$S_{0,4}$ $S_{0,5}$ $S_{1,4}$ $S_{1,5}$
$S_{2,0}$ $S_{2,1}$ $S_{3,0}$ $S_{3,1}$
$S_{2,2}$ $S_{2,3}$ $S_{3,2}$ $S_{3,3}$
$S_{2,4}$ $S_{2,5}$ $S_{3,4}$ $S_{3,5}$
$S_{4,0}$ $S_{4,1}$ $S_{5,0}$ $S_{5,1}$
$S_{4,2}$ $S_{4,3}$ $S_{5,2}$ $S_{5,3}$
$S_{4,4}$ $S_{4,5}$ $S_{5,4}$ $S_{5,5}$

Fig. 4.3: Portion of a decomposition of the forward table into blocks of size 2 × 2.

In the basic wavefront approach, the forward table is divided into blocks of the same size (Fig. 4.3). Each block inherits the data dependencies from the forward table. Moreover, there are data dependencies between the blocks (Fig. 4.4). For instance, at the beginning all entries of the block $B_{1,1}$ can be computed. Then the cells of the blocks $B_{1,2}$ and $B_{2,1}$ can be filled depending on the availability of the boundary entries of block $B_{1,1}$. Moreover, block $B_{2,2}$ needs both the boundary entries of $B_{1,2}$ and $B_{2,1}$ as well as the last entry of $B_{1,1}$. Thus the complete forward matrix exhibits parallelism at two levels: intra-block and inter-block. Both forms show a similar anti-diagonal pattern of parallelism.

Suppose each block has r rows and c columns. Then by the wavefront approach, there are $r + c - 1$ anti-diagonals such that the block can be computed in $r + c - 1$ parallel steps. To compute $r \cdot c$ cell entries in each block, $r+c-1$ boundary cells requires values from adjacent blocks. Each block takes $r + c + 1$ boundary cells to compute $r \cdot c$ cell entries. Thus the communication-to-computation ratio becomes $(r + c + 1)/(r \cdot c)$. This ratio decreases when the block size increases.

A typical implementation of a wavefront algorithm on a GPU launches two kernels, one for initialization and one for filling the forward table [90]. The kernel for initialization calculates the values of the boundary cells, while the kernel for computation of the forward table provides a grid of blocks such that the blocks (of threads) correspond one-to-one with the blocks in the decomposition of the forward table. Moreover, the threads in a block are associated one-to-one with the cells of the corresponding block

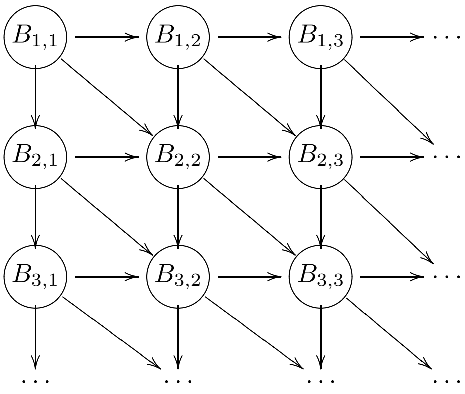


Fig. 4.4: Block structure of the forward table.

in the forward table each of which calculating the value of the cell. The kernel naturally emulates the anti-diagonal parallelism inside each block. For this, the threads in each block need to synchronize due to the dependencies among the anti-diagonals using the GPU function `syncthreads`.

However, CUDA does not provide global barrier synchronization between blocks and blocks exhibit producer-consumer relationship (due to data dependencies). Therefore, the anti-diagonal parallelism among blocks needs to be implemented on the host CPU.

4.2.3 Performance

In order to establish which approach is preferable for implementation of profile-profile alignment, we have compared matrix and wavefront approaches using execution time, speed-up, and *Cell Updates Per Second* (CUPS). *FLoating-point Operations Per Second* (FLOPS) depicts one dimension of algorithm efficiency. It ignores memory traffic and other overheads related to program execution. Moreover, the number of actual FLOPS are hardware architecture dependent. For these reasons, we have not considered FLOPS as performance measure.

The profiles have been generated at random for various lengths ranging from 32 to 992 with a step size of 32. The reasons are that CUDA has a fixed warp size of 32 threads. Additionally, profiles of higher length up to 10,000 are considered to analyze the behaviour of the different versions of the profile-profile alignment. Profiles of length longer than 10,000 are not taken into account due to hardware limitations since then the tables become so huge that they may occupy the whole GPU memory. Moreover, only profiles of comparable length are considered. Execution times have been averaged over ten runs for each profile length.

Four basic implementations of the algorithm ProfProfAlignScalProd have been evaluated; in each case, the forward table is decomposed into blocks of size $k \times k$:

- Simple k: each thread calculates one cell of the forward table (k^2 threads per block).
- Row k: each thread computes one row of a block (k threads per block).

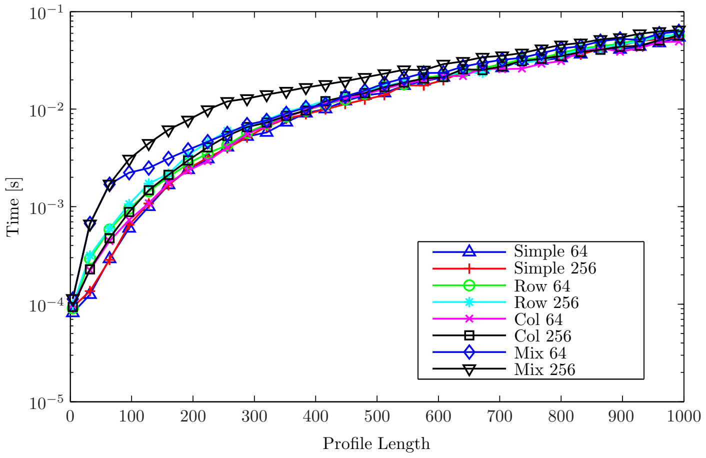


Fig. 4.5: Runtime (in seconds) of ProfProfAlignScalProd on NVIDIA.

- Column k: each thread yields one column of a block (k threads per block).
- Mix k: each thread evaluates one row and one column of a block (k threads per block).

The results are illustrated in Figure 4.5 for block sizes $k = 64$ and 256 by considering kernel execution and transfer of results back to CPU. It appears that all four approaches yield similar execution times for alignments of profiles longer than 500. Moreover, the block size seems to have no influence on the performance. The reason is that the four variants implementing the data independent part exploit parallelism quite similarly.

Next, the algorithm ProfProfAlignScalProd has been implemented by calculating the forward table using the CUBLAS library functions `SAXPY` (which takes the difference between two profiles) and `SNRM2` (which calculates Euclidean distance). A comparison of this implementation with two implementations of Simple k (considering only kernel execution and transfer of results back to host memory) and an Intel CPU implementation via MKL using the library functions `SAXPY` and `SNRM2` to establish the forward table is given in Figure 4.6. It appears that Simple k outperforms both the CUBLAS and the Intel CPU implementations by one order of magnitude. Moreover, the Intel CPU implementation is faster than the CUBLAS one up to profiles of length 700. The reason is that the CUBLAS function `SNRM2` stores results back into the host memory for each cell.

The algorithm ProfProfAlignMatProd is difficult to implement for larger profiles. Indeed, the memory required to store the intermediate matrices on the device can become huge when compared with the size of the results, since only the data on the major diagonal are used. However, the algorithm ProfProfAlignMatProd can be realized using vector multiplication.

For this purpose, two variants have been considered:

- MatProd V1: use CUBLAS functions.
- MatProd V2 k: perform all calculations on the GPU (k threads per block).

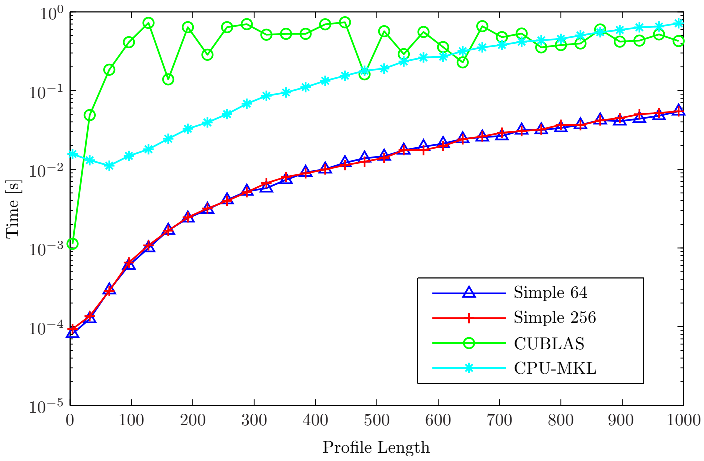


Fig. 4.6: Runtime (in seconds) of ProfProfAlignScalProd on NVIDIA and Intel CPU using MKL.

Note that MatProd V1 can be implemented with the CUBLAS functions `gemm` for subtraction of matrices and `dot` for component-wise multiplication of vectors, which corresponds to the multiplication of the diagonals of the involved matrices. Indeed, `gemm` performs the operation

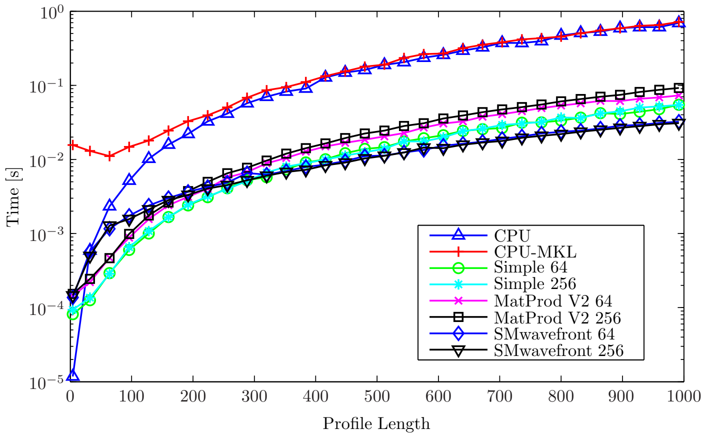
$$\boldsymbol{C} \leftarrow \alpha \boldsymbol{A}\boldsymbol{B} + \beta \boldsymbol{C}. \tag{4.10}$$

Thus the subtraction of two matrices can be carried out by setting $\alpha = 1$, $\beta = -1$, and taking the identity matrix for $\boldsymbol{B}$. But the identity matrix has the same size as the matrix $\boldsymbol{A}$ so that unnecessary computations are performed. Hence, the idea to implement ProfProfAlignMatProd by MatProd V1 has been discarded.

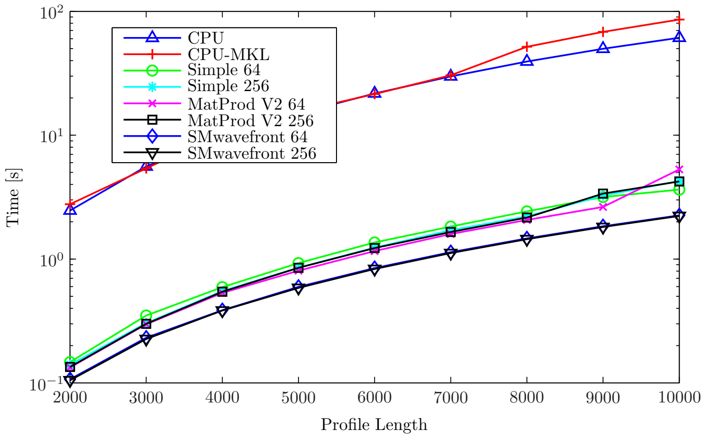
On the other hand, we have specified the wavefront approach in Section 4.2.2. This approach is implemented by the SMwavefront k method where the data produced by the blocks are stored in global memory. These data will be transferred to shared memory when a new block is being launched that requires access to these data. The anti-diagonal parallelism among blocks is exploited by the CPU. This will incur some delay due to switching between GPU and CPU.

Furthermore, we have compared five implementations of the profile-profile alignment algorithm: Simple k, MatProd V2 k, SMwavefront k (for block sizes $k = 64$ and 256) and the Intel CPU implementation with and without the MKL. First, kernel execution and transfer of results back to host have been considered. The results in Figure. 4.7 exhibit that the Simple k approach performs best for profile lengths up to 500 while Simple k and SMwavefront k have almost similar execution times with a slight edge in performance to SMwavefront k for lengths longer than 500. The performance degradation of SMwavefront k for smaller length is due to the computation-to-communication cost since a smaller number of blocks is executed concurrently. As the profile lengths increase, multiple anti-diagonal block executions for SMwavefront k result in superior performance when compared with Simple k and MatProd V2 k.

Second, kernel execution, memory allocation, and transfer of results back to host have been taken into account (Fig. 4.8). MatProd V2 k has almost the same performance as the Intel CPU implementation (with and without MKL) because duplication



(a) Profile length < 1000

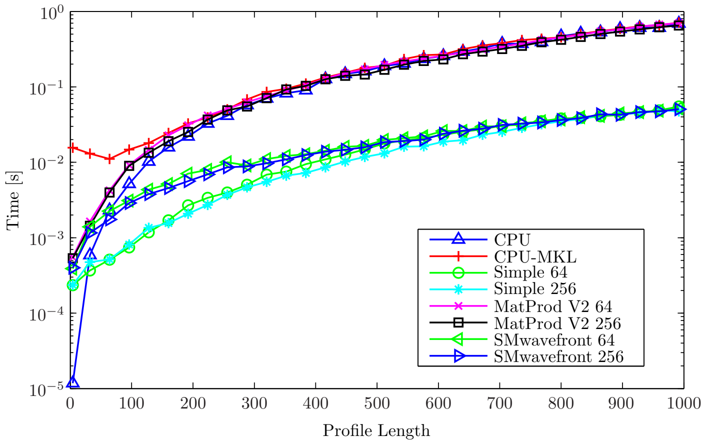


(b) Profile length > 1000

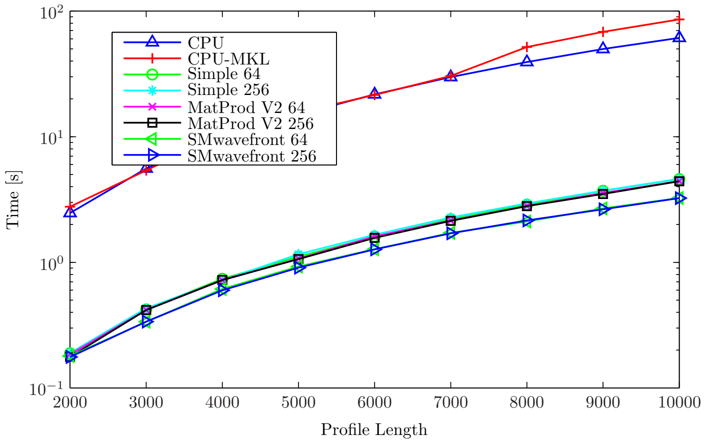
Fig. 4.7: Runtime (in seconds) of profile-profile alignment algorithms on NVIDIA and Intel CPU (with and without MKL) by considering kernel execution and transfer of results back to CPU.

of profiles for the purpose of vector subtraction is time consuming. When using the MKL BLAS routine `SAXPY` to calculate the difference between profile vectors, the new data will overwrite the old ones. To avoid this, the profiles should be pre-stored causing a degradation in performance. The SMwavefront k execution time is calculated by excluding switching delay between CPU and GPU. However, the impact of this delay on the performance of SMwavefront k is not significant for profiles of length < 1000, but the switching delay for longer profile lengths becomes a significant factor for the superior performance of SMwavefront k when compared with Simple k. For profiles of length > 1000, the runtime performance of MatProd V2 k is almost similar to that of Simple k and SMwavefront k.

Next, the speed-ups attained with the NVIDIA implementations when compared with the Intel CPU implementation using MKL have been calculated. First, kernel execution and transfer of results back to the host have been considered. The results in Figure. 4.9 illustrate that NVIDIA implementations of Simple k and SMwavefront k achieve speed-up factors of one order of magnitude when compared with the Intel CPU



(a) Profile length < 1000

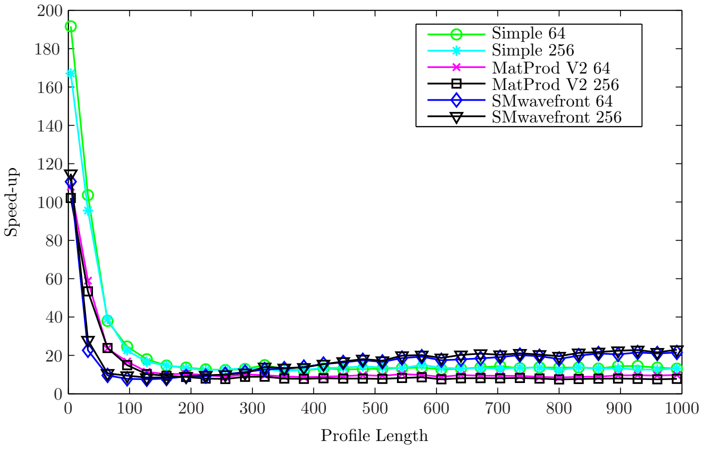


(b) Profile length > 1000

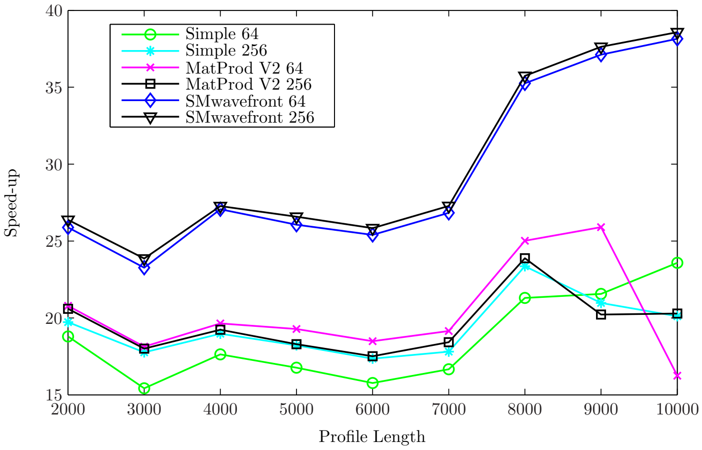
Fig. 4.8: Runtime (in seconds) of profile-profile alignment algorithms on NVIDIA and Intel CPU (with and without MKL) by considering memory allocation, data transfer, and kernel execution.

implementation using MKL. For the profile lengths > 1000, SMwavefront k exhibits maximum speed-up factor of about 38.5 (mean 30) while Simple k and MatProd V2 k achieve average speed-up factor of about 20. Second, this result remains valid when overheads for memory allocation are taken into account (Fig. 4.10). Note that MatProd V2 k does not have a significant speed-up due to duplication of profiles for subtraction purposes. Tables 4.1 and 4.2 provide more details which depicts that SMwavefront k performs much better than other implementations for longer profiles.

To sum up, the CUPS attained with NVIDIA implementations and CPU-MKL have been calculated (Fig. 4.11). For each cell to calculate the squared Euclidean distance, we require three times 21 multiplications, 21 additions, and 21 subtractions (20 amino acids plus blank). This large number of floating point operations is the reason for smaller values of CUPS. Another factor that contributes to small values of CUPS for Simple k and MatProd V2 k is due to the processing of the data dependent part on the host CPU. Table 4.3 provide more details about CUPS. SMwavefront k



(a) Profile length < 1000



(b) Profile length > 1000

Fig. 4.9: Speed-ups of profile-profile alignment algorithms for Simple k, MatProd V2 k, and SMwavefront k over CPU-MKL by considering kernel execution and transfer of results back to CPU memory.

achieves about 42 MCUPS on average and clearly outperforms Simple k and MatProd V2 k implementations.

In this section, the parallel formulation of profile-profile alignment was studied using the matrix-matrix product and the wavefront methods. These results exhibit that the SMwavefront 256 is a very good candidate for the implementation of profile-profile alignment on a GPU because it has better hardware utilization and speed-up compared to Simple k and MatProd V2 k. The matrix approach is not a suitable option since the elements at diagonal are of interest and unnecessary operations are performed for other matrix entries.

Table 4.1: Maximum and average speed-ups of profile-profile alignment algorithms for Simple k, MatProd V2 k, and SMwavefront k over CPU-MKL by considering kernel execution and transfer of results back to CPU memory.

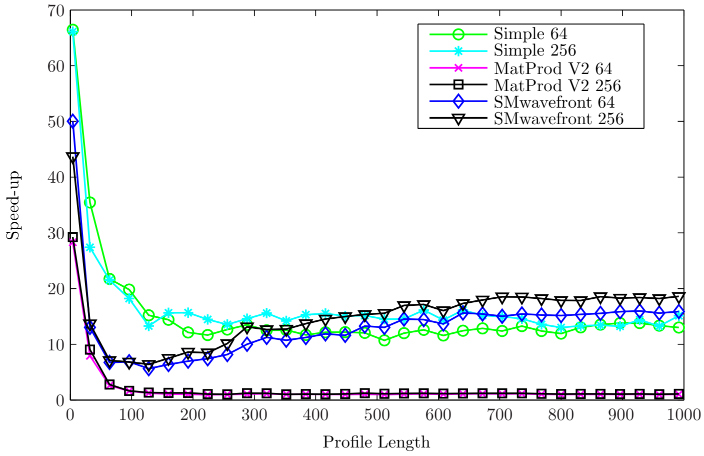
Speed-up	**profile length < 1000**		**profile length > 1000**	
	max	**mean**	**max**	**mean**
Simple 64	191.6012	23.0617	23.5844	18.6107
Simple 256	167.0107	21.8563	23.3564	19.3647
MatProd V2 64	107.9910	14.7831	25.8997	20.2941
MatProd V2 256	102.1288	13.1962	23.8790	19.6056
SMwavefront 64	110.5914	18.9352	38.1476	29.4485
SMwavefront 256	114.7355	20.0702	38.5810	29.9065

Table 4.2: Maximum, and average speed-ups of profile-profile alignment algorithms for Simple k, MatProd V2 k, and SMwavefront over CPU-MKL by considering kernel execution, memory allocation, and data transfer between CPU and GPU memory.

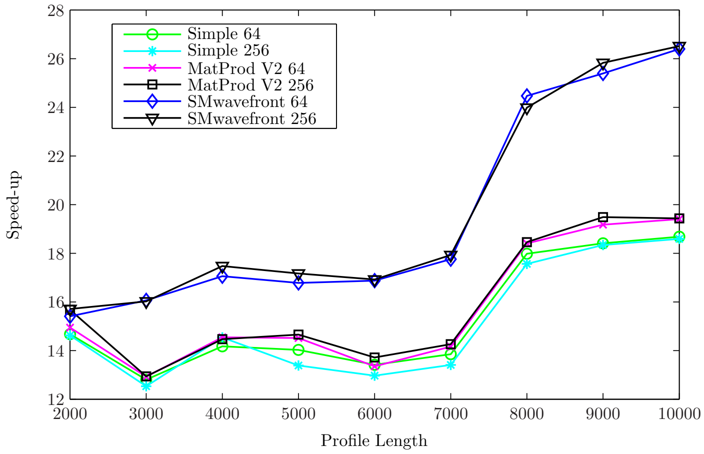
Speed-up	**profile length < 1000**		**profile length > 1000**	
	max	**mean**	**max**	**mean**
Simple 64	66.4489	15.5892	18.6781	15.3341
Simple 256	66.1113	16.9095	18.5901	15.1040
MatProd V2 64	28.3095	2.1860	19.4008	15.7134
MatProd V2 256	29.2206	2.3378	19.4863	15.8976
SMwavefront 64	50.0176	13.5337	26.3981	19.5776
SMwavefront 256	43.6919	15.4248	26.5118	19.7270

Table 4.3: Maximum, and average MCUPS of profile-profile alignment algorithms.

MCUPS	**profile length < 1000**		**profile length > 1000**	
	max	**mean**	**max**	**mean**
CPU-MKL	1.4146	1.1668	1.6275	1.4588
Simple 64	19.6801	16.5991	27.4666	26.6164
Simple 256	19.1384	16.2904	29.6419	27.9431
MatProd V2 64	13.5027	11.4729	31.0323	29.3160
MatProd V2 256	10.8376	9.8933	30.0492	28.3202
SMwavefront 64	29.6980	19.0420	44.4270	41.9456
SMwavefront 256	31.6278	20.0917	44.9318	42.6131



(a) Profile length < 1000



(b) Profile length > 1000

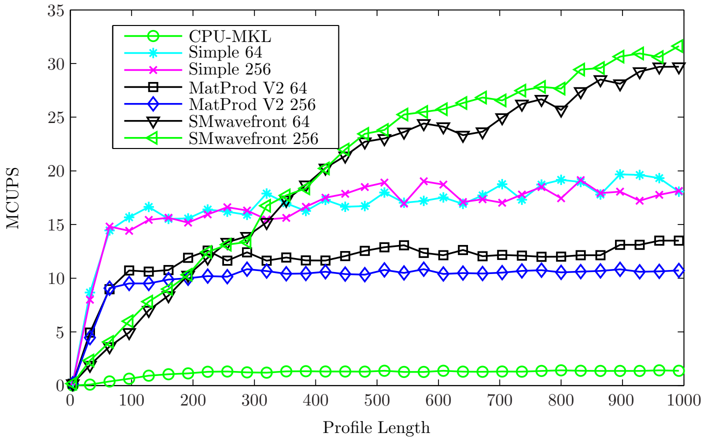
Fig. 4.10: Speed-ups of of profile-profile alignment algorithms for Simple k, MatProd V2 k, and SMwavefront k over CPU-MKL by considering kernel execution, memory allocation, and data transfer between CPU and GPU memory.

4.3 Profile-Sequence Alignment

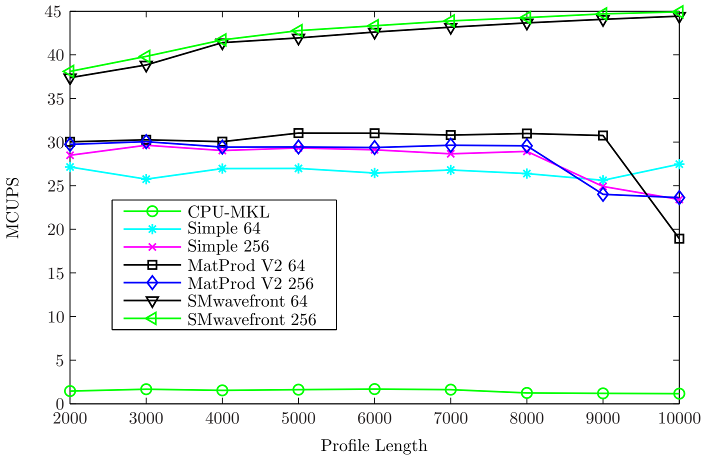
The algorithm ProSeqAlign exhibits similar properties to ProfProfAlign. The data dependency among the forward table entries impacts the parallel design of the algorithm and its performance (Fig. 4.1). For this reason, two approaches are discussed to parallelize profile-sequence alignment i.e., wavefront and matrix approaches. The wavefront approach for profile-sequence alignment is similar to profile-profile alignment (Section 4.2.2). The matrix approach is discussed in the following section.

4.3.1 Matrix Approach

Profile-sequence alignment was parallelized using the matrix approach by Bassoy et. al. [5]. The alignment algorithm was designed using matrix-vector and matrix-matrix product methods. However, the methods discussed have a drawback by introducing



(a) Profile length < 1000



(b) Profile length > 1000

Fig. 4.11: Performance (MCUPS) of profile-profile alignment algorithms.

some matrices which require additional memory and clock cycles. Next, we will discuss these methods and redesign them to achieve better performance.

To gain the benefits of parallelization, the algorithm ProSeqAlign is divided into the data independent and data dependent parts. First, the data independent part calculates three scalar products and store the results in $m \times n$ matrices $\boldsymbol{H}$, $\boldsymbol{V}$, and $\boldsymbol{D}$. Second, these matrices are used in the data dependent part for calculation of the forward table entries. Take the extended alphabet $\Sigma' = \{a_1, \ldots, a_l\}$, where a_l equals blank, and assign

$$\boldsymbol{w}_a = (\sigma(a, a_1), \ldots, \sigma(a, a_l))^T\,, \quad a \in \Sigma'. \tag{4.11}$$

The matrices are computed as

$$\boldsymbol{H}_{i,j} = \sigma(-_o, x_j) = \sum_b \sigma(x_j, b) \cdot -_o = -_o^T \cdot \boldsymbol{w}_{x_j}, \tag{4.12}$$

$$\boldsymbol{V}_{i,j} = \sigma(\boldsymbol{o}_i, -) = \sum_b \sigma(-, b) \cdot o_{i,b} = \boldsymbol{o}_i^T \cdot \boldsymbol{w}_-, \tag{4.13}$$

$$\boldsymbol{D}_{i,j} = \sigma(\boldsymbol{o}_i, x_j) = \sum_b \sigma(x_j, b) \cdot o_{i,b} = \boldsymbol{o}_i^T \cdot \boldsymbol{w}_{x_j}. \tag{4.14}$$

Now, the algorithm ProSeqAlign can be easily converted into matrix-vector product. For this, take the $l \times n$ matrix $\boldsymbol{W}$ given as

$$\boldsymbol{W} = (\boldsymbol{w}_{x_1}, \ldots, \boldsymbol{w}_{x_n}). \tag{4.15}$$

The i-th row of matrices $\boldsymbol{H}$, $\boldsymbol{V}$, and $\boldsymbol{D}$ can be calculated using matrix-vector product

$$\boldsymbol{H}_i = -_o^T \cdot \boldsymbol{W}, \tag{4.16}$$

$$\boldsymbol{V}_i = \boldsymbol{o}_i^T \cdot \boldsymbol{w}_-, \tag{4.17}$$

$$\boldsymbol{D}_i = \boldsymbol{o}_i^T \cdot \boldsymbol{W}. \tag{4.18}$$

To calculate first column of the table $\boldsymbol{S}$, take the lower triangular $m \times m$ matrix $\boldsymbol{B}_m$

$$\begin{aligned}
\boldsymbol{B}_m[\boldsymbol{O}^T \cdot \boldsymbol{w}_-] &= \begin{pmatrix} 1 & & & \\ 1 & 1 & & \\ \vdots & & \ddots & \\ 1 & 1 & \ldots & 1 \end{pmatrix} \left[\begin{pmatrix} \boldsymbol{o}_1^T \\ \vdots \\ \boldsymbol{o}_m^T \end{pmatrix} \cdot \boldsymbol{w}_- \right] \\
&= \begin{pmatrix} \boldsymbol{o}_1^T \cdot \boldsymbol{w}_- \\ \boldsymbol{o}_1^T \cdot \boldsymbol{w}_- + \boldsymbol{o}_2^T \cdot \boldsymbol{w}_- \\ \ldots \\ \boldsymbol{o}_1^T \cdot \boldsymbol{w}_- + \boldsymbol{o}_2^T \cdot \boldsymbol{w}_- + \ldots + \boldsymbol{o}_m^T \cdot \boldsymbol{w}_- \end{pmatrix}.
\end{aligned} \tag{4.19}$$

Similarly, the first row is calculated by having the $n \times n$ upper triangular matrix $\boldsymbol{B}_n$

$$\begin{aligned}
[-_o^T \cdot \boldsymbol{W}]\boldsymbol{B}_n &= \left[-_o^T \cdot \begin{pmatrix} \boldsymbol{w}_{x_1} & \boldsymbol{w}_{x_2} & \ldots & \boldsymbol{w}_{x_n} \end{pmatrix}\right] \begin{pmatrix} 1 & 1 & \ldots & 1 \\ & 1 & \ldots & 1 \\ & & \ddots & \vdots \\ & & & 1 \end{pmatrix} \\
&= \begin{pmatrix} -_o^T \cdot \boldsymbol{w}_{x_1} & \ldots & -_o^T \cdot \boldsymbol{w}_{x_1} + \ldots + -_o^T \cdot \boldsymbol{w}_{x_n} \end{pmatrix}.
\end{aligned} \tag{4.20}$$

The matrix-vector product based algorithm in [5] have $n \times n$ matrices $\boldsymbol{H}$ and $\boldsymbol{D}$. This will generate error when profile and sequence lengths are not equal. This deficiency is corrected in algorithm ProSeqAlignMatVecProdV1.

The algorithm ProSeqAlignMatVecProdV1 can be redesigned as matrix multiplication by having the $l \times n$ matrix

$$\boldsymbol{W}_- = (\boldsymbol{w}_-, \ldots, \boldsymbol{w}_-) \tag{4.21}$$

and the $l \times m$ matrix

$$\boldsymbol{O}_- = (-_o, \ldots, -_o). \tag{4.22}$$

Algorithm 4.3 ProSeqAlignMatVecProdV1($\boldsymbol{x}, \boldsymbol{O}$)

```
Require: sequence $\boldsymbol{x} = x_1 \ldots x_n$ and profile $\boldsymbol{O} = \boldsymbol{o}_1 \ldots \boldsymbol{o}_m$
$S_{0,0} \leftarrow 0$                                   // initialization
$\boldsymbol{S}_{*,0} \leftarrow \boldsymbol{B}_m[\boldsymbol{O}^T \cdot \boldsymbol{w}_-]$
$\boldsymbol{S}_{0,*} \leftarrow [-_o^T \cdots \boldsymbol{W}]\boldsymbol{B}_n$
for $i \leftarrow 1$ to $n$ do                           // calculation
    $\boldsymbol{V}_i \leftarrow \boldsymbol{O}^T \cdot \boldsymbol{w}_-$
end for
for $i \leftarrow 1$ to $m$ do
    $\boldsymbol{H}_i \leftarrow -_o^T \cdot \boldsymbol{W}$
    $\boldsymbol{D}_i \leftarrow \boldsymbol{o}_i^T \cdot \boldsymbol{W}$
end for
for $i \leftarrow 1$ to $m$ do                           // maximization
    for $j \leftarrow 1$ to $n$ do
        $S_{i,j} \leftarrow \max\{S_{i-1,j} + V_{ij}, S_{i,j-1} + H_{ij}, S_{i-1,j-1} + D_{ij}\}$
    end for
end for
return $\boldsymbol{S}$
```

The matrices $\boldsymbol{H}$, $\boldsymbol{V}$, and $\boldsymbol{D}$ can be computed by matrix multiplication as

$$\boldsymbol{H} = \boldsymbol{O}_-^T \cdot \boldsymbol{W}, \tag{4.23}$$

$$\boldsymbol{V} = \boldsymbol{O}^T \cdot \boldsymbol{W}_-, \tag{4.24}$$

$$\boldsymbol{D} = \boldsymbol{O}^T \cdot \boldsymbol{W}. \tag{4.25}$$

This gives the algorithm ProSeqAlignMatProdV1.

Algorithm 4.4 ProSeqAlignMatProdV1($\boldsymbol{x}, \boldsymbol{O}$)

```
Require: sequence $\boldsymbol{x} = x_1 \ldots x_n$ and profile $\boldsymbol{O} = \boldsymbol{o}_1 \ldots \boldsymbol{o}_m$
$S_{0,0} \leftarrow 0$                                   // initialization
$\boldsymbol{S}_{*,0} \leftarrow \boldsymbol{B}_m[\boldsymbol{O}^T \cdot \boldsymbol{w}_-]$
$\boldsymbol{S}_{0,*} \leftarrow [-_o^T \cdots \boldsymbol{W}]\boldsymbol{B}_n$
$\boldsymbol{V} \leftarrow \boldsymbol{O}^T \cdot \boldsymbol{W}_-$      // calculation
$\boldsymbol{H} \leftarrow \boldsymbol{O}_-^T \cdot \boldsymbol{W}$
$\boldsymbol{D} \leftarrow O^T \cdot \boldsymbol{W}$
for $i \leftarrow 1$ to $m$ do                           // maximization
    for $j \leftarrow 1$ to $n$ do
        $S_{i,j} \leftarrow \max\{S_{i-1,j} + V_{ij}, S_{i,j-1} + H_{ij}, S_{i-1,j-1} + D_{ij}\}$
    end for
end for
return $\boldsymbol{S}$
```

These algorithms are taken from [5] with some modifications. However, these two algorithms have redundant data calculations for matrices $\boldsymbol{H}$ and $\boldsymbol{V}$. The calculation of matrix $\boldsymbol{H}$

$$\boldsymbol{H} = \begin{bmatrix} -_o^T \cdot \boldsymbol{w}_{x_1} & \dots & -_o^T \cdot \boldsymbol{w}_{x_n} \\ -_o^T \cdot \boldsymbol{w}_{x_1} & \dots & -_o^T \cdot \boldsymbol{w}_{x_n} \\ & \vdots & \\ -_o^T \cdot \boldsymbol{w}_{x_1} & \dots & -_o^T \cdot \boldsymbol{w}_{x_n} \end{bmatrix}$$

and matrix $\boldsymbol{V}$ is

$$\boldsymbol{V} = \begin{bmatrix} \boldsymbol{o}_1^T \cdot \boldsymbol{w}_- & \dots & \boldsymbol{o}_1^T \cdot \boldsymbol{w}_- \\ \boldsymbol{o}_2^T \cdot \boldsymbol{w}_- & \dots & \boldsymbol{o}_2^T \cdot \boldsymbol{w}_- \\ & \vdots & \\ \boldsymbol{o}_m^T \cdot \boldsymbol{w}_- & \dots & \boldsymbol{o}_m^T \cdot \boldsymbol{w}_- \end{bmatrix}.$$

Here, the first row of matrix $\boldsymbol{H}$ is repeated m times. For matrix $\boldsymbol{V}$, the first column is repeated n times. These redundant calculations will affect the overall performance. The storage of these matrices is a problem for large profile and sequence lengths. Therefore, vectors $\boldsymbol{h}$ and $\boldsymbol{v}$ are used to hold these values. In this way, we have avoided unnecessary multiplications and memory space is reduced to n and m, respectively. The resulting matrix-vector product and matrix product algorithms are ProSeqAlignMatVecProdV2 and ProSeqAlignMatProdV2.

Algorithm 4.5 ProSeqAlignMatVecProdV2($\boldsymbol{x}, \boldsymbol{O}$)

```
Require: sequence $\boldsymbol{x} = x_1 \dots x_n$ and profile $\boldsymbol{O} = \boldsymbol{o}_1 \dots \boldsymbol{o}_m$
$S_{0,0} \leftarrow 0$                                   // initialization
$\boldsymbol{v} \leftarrow \boldsymbol{O}^T \cdot \boldsymbol{w}_-$
$\boldsymbol{h} \leftarrow -_o^T \cdot \boldsymbol{W}$
$\boldsymbol{S}_{*,0} \leftarrow \boldsymbol{B}_m \boldsymbol{v}$
$\boldsymbol{S}_{0,*} \leftarrow \boldsymbol{h} \boldsymbol{B}_n$
for $i \leftarrow 1$ to $m$ do                          // calculation
    $\boldsymbol{D}_i \leftarrow \boldsymbol{o}_i^T \cdot \boldsymbol{W}$
end for
for $i \leftarrow 1$ to $m$ do                          // maximization
    for $j \leftarrow 1$ to $n$ do
        $S_{i,j} \leftarrow \max\{S_{i-1,j} + v_i, S_{i,j-1} + h_j, S_{i-1,j-1} + D_{ij}\}$
    end for
end for
return $\boldsymbol{S}$
```

4.3.2 Performance

The performance of the above mentioned approaches is compared using execution time, speed-up, and CUPS. The profiles and amino acid sequences have been generated at random for various lengths ranging from 32 to 992 with a step size of 32. We have considered sequence length up to 10,000 for comparing different versions of profile-sequence alignment. Moreover, only profiles of comparable length are considered. Execution times have been averaged over ten runs for each sequence length. We have also neglected the time for memory allocation and data transfer to or from GPU.

Algorithm 4.6 PROSEQALIGNMATPRODV2($\boldsymbol{x}, \boldsymbol{O}$)

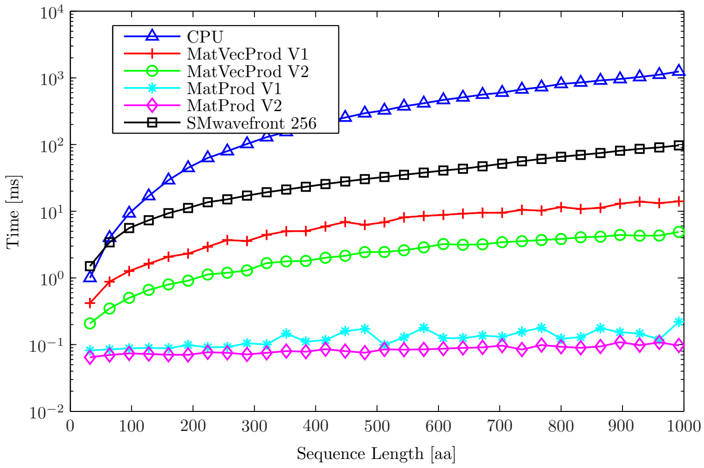
```
Require: sequence $\boldsymbol{x} = x_1 \dots x_n$ and profile $\boldsymbol{O} = \boldsymbol{o}_1 \dots \boldsymbol{o}_m$
$S_{0,0} \leftarrow 0$                                        // initialization
$\boldsymbol{v} \leftarrow \boldsymbol{O}^T \cdot \boldsymbol{w}_-$
$\boldsymbol{h} \leftarrow -_o^T \cdot \boldsymbol{W}$
$\boldsymbol{S}_{*,0} \leftarrow \boldsymbol{B}_m \boldsymbol{v}$
$\boldsymbol{S}_{0,*} \leftarrow \boldsymbol{h} \boldsymbol{B}_n$
$\boldsymbol{D} \leftarrow \boldsymbol{O}^T \cdot \boldsymbol{W}$
for $i \leftarrow 1$ to $m$ do                                // maximization
    for $j \leftarrow 1$ to $n$ do
        $S_{i,j} \leftarrow \max\{S_{i-1,j} + v_i, S_{i,j-1} + h_j, S_{i-1,j-1} + D_{ij}\}$
    end for
end for
return $\boldsymbol{S}$
```

Six implementations of profile-sequence alignment have been considered.

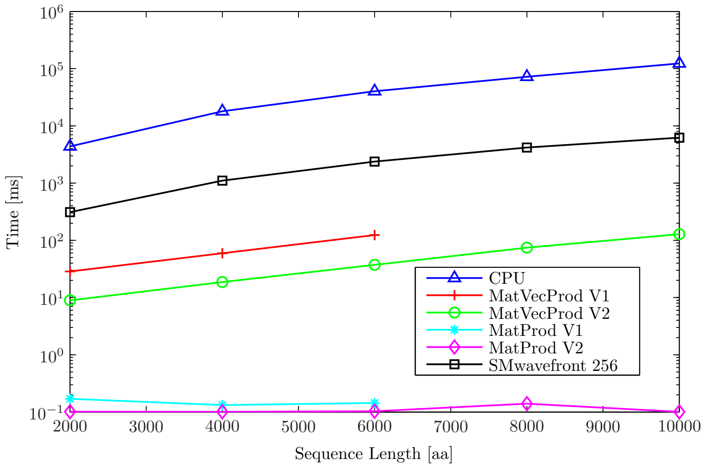
- CPU: serial implementation of PROSEQALIGN on Intel CPU.
- MatVecProd V1: GPU implementation of PROSEQALIGNMATVECPRODV1 with `cublasSgemv`.
- MatVecProd V2: GPU implementation of PROSEQALIGNMATVECPRODV2 with `cublasSgemv`.
- MatProd V1: GPU implementation of PROSEQALIGNMATPRODV1 with `cublasSgemm`.
- MatProd V2: GPU implementation of PROSEQALIGNMATPRODV2 with `cublasSgemm`.
- SMwavefront 256: wavefront approach with shared memory and 256 threads per block.

For MatVecProd V1 and MatProd V1, results are taken up to sequence length 6,000. Sequence length 10,000 require approximately 1145 MB to hold matrices $\boldsymbol{H}$, $\boldsymbol{V}$, and $\boldsymbol{D}$. This is well beyond the capacity of available global memory (1024 MB). So the approaches MatVecProd V2 and MatProd V2 are devised to allow simulations to run on large length which does not require the calculation of the matrices $\boldsymbol{H}$ and $\boldsymbol{V}$.

The performance results presented here compare the GPU based methods with an optimized serial PROSEQALIGN. Not unexpectedly, the parallel implementations outperformed the serial ones. First, these algorithms are evaluated based on their runtime (Fig. 4.12 and 4.13). The time to perform the matrix operation on NVIDIA GPU for MatVecProd V1, MatVecProd V2, MatProd V1, and MatProd V2 is depicted in Figure 4.12. Matrix-matrix product based algorithms are faster than others due to the use of the optimized CUBLAS library routines [79]. The results also show the effect of unnecessary matrix operations in MatVecProd V1 and MatProd V1. Moreover, the



(a) Sequence length < 1000



(b) Sequence length > 1000

Fig. 4.12: Runtime (in milliseconds) of profile-sequence alignment algorithms on NVIDIA and Intel CPU.

performance of `cublasSgemm` over `cublasSgemv` is exhibited. The complexity of both methods is $O(lmn)$. The difference lies in reusing of data by `cublasSgemm`. This result is valid for sequence length > 1000. The performance of SMwavefront depends on computation-to-communication cost. The execution of small number of blocks concurrently means the data transfer time cannot be hidden by executing the program on a highly parallel GPU architecture. As the profile lengths increase, multiple anti-diagonal blocks are executed for SMwavefront k which results in better hardware utilization.

Table 4.4 shows MatProd V1 and MatProd V2 are faster when compared with other implementations by considering only matrix operations performed on GPU. Here, SMwavefront k has worst timing than other GPU implementations because calculation of the entire forward table is performed on GPU.

Moreover, when evaluating the runtime for calculation of the forward table, the performance of GPU based implementations are significantly better than the pure CPU

Table 4.4: Maximum and average runtime (in milliseconds) of profile-sequence alignment algorithms on NVIDIA and Intel CPU.

Runtime	**sequence length < 1000**		**sequence length > 1000**	
	max	**mean**	**max**	**mean**
CPU	1241.6000	424.2226	123432.7000	51585.0600
MatVecProd V1	14.0996	7.1375	123.3552	70.5263
MatVecProd V2	4.8919	2.4844	127.6302	53.3725
MatProd V1	0.2196	0.1276	0.1704	0.1489
MatProd V2	0.1087	0.0842	0.1398	0.1092
SMwavefront 256	97.2716	38.7848	6217.2935	2836.5884

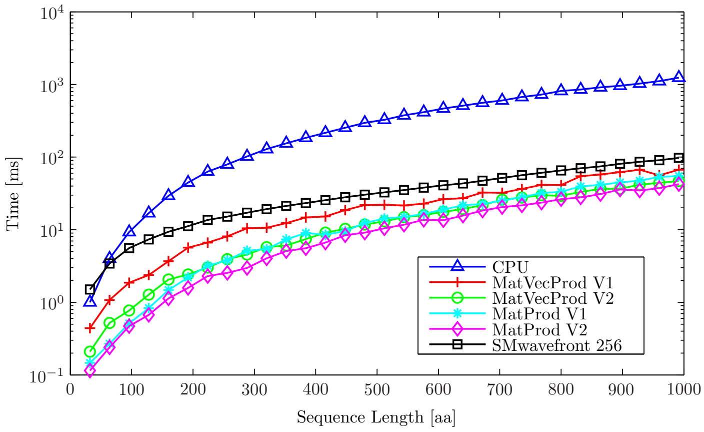
Table 4.5: Maximum and average runtime (in milliseconds) of maximization step for profile-sequence alignment algorithms on CPU.

Runtime	**sequence length < 1000**		**sequence length > 1000**	
	max	**mean**	**max**	**mean**
MatVecProd V1	53.0004	18.7186	1927.7484	1001.79256
MatVecProd V2	41.6634	14.377	4714.3644	2005.03408
MatProd V1	54.6049	18.5096	2078.4036	1109.70057
MatProd V2	42.206	13.9491	4129.8077	1831.43366

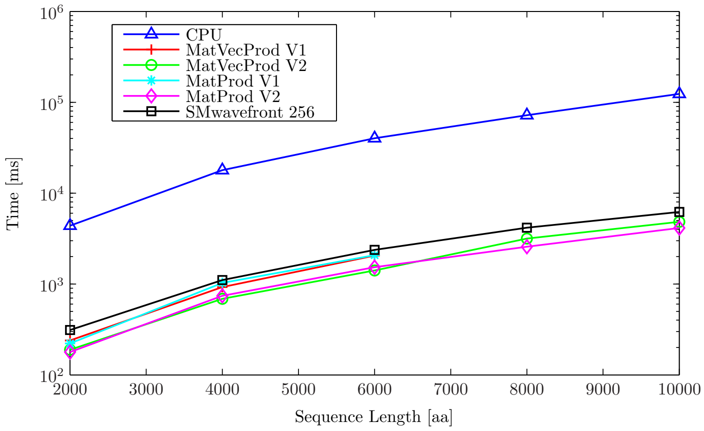
version. The results exhibit that the performance of GPU based implementations is approximately same. The matrix based solution is best if we consider only calculation on the GPU. However, the maximization step of the forward table is performed on CPU which dominates the total computational cost. MatProd V2 performs better than other algorithms because unnecessary matrix operations are not required. Table 4.5 provides more detail. MatVecProd V1 and MatProd V1 take almost the same time for calculation on CPU. For sequence length greater than 1000, the mean value for MatVecProd V1 and MatProd V1 is small because it is averaged for sequence length up to 6,000. The vectors $\boldsymbol{h}$ and $\boldsymbol{v}$ are the main reason behind the less time taken by algorithms MatVecProd V2 and MatProd V2.

Next, the speed-ups attained with the NVIDIA implementations when compared with the Intel CPU implementation have been computed. The speed-up is calculated using the time required by the serial algorithm divided by the time required by the GPU (Fig. 4.14). The results illustrate that NVIDIA implementations achieve speed-up factors of one order of magnitude when compared with the Intel CPU implementation. The matrix approach has better speed-up than the wavefront approach. MatVecProd V2 and MatProd V2 exhibit good speed-up over MatVecProd V1 and MatProd V1. Table 4.6 shows that the maximum average speed-up obtained is a factor of about 28.

Finally, the CUPS with NVIDIA implementations and Intel CPU are calculated (Fig. 4.11) . For each cell to calculate the score, we require 3 times 21 multiplications



(a) Sequence length < 1000

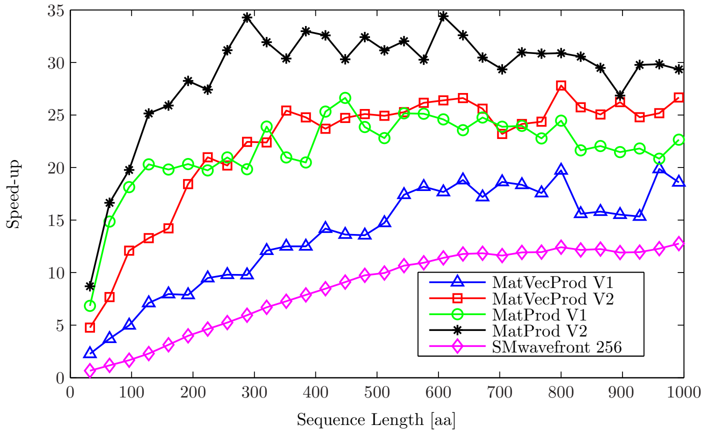


(b) Sequence length > 1000

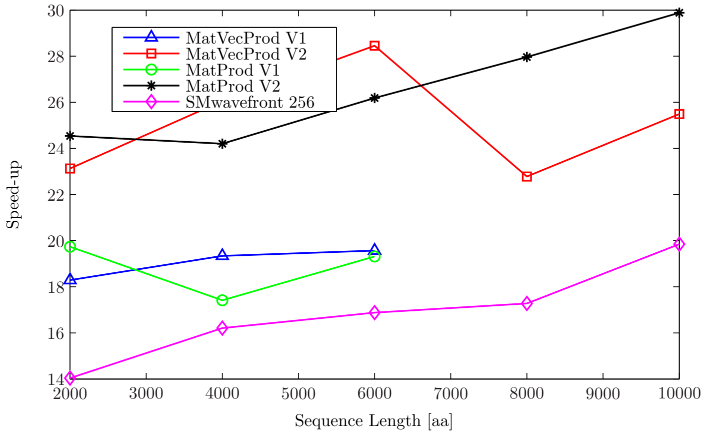
Fig. 4.13: Runtime (in milliseconds) for calculation of the forward table for profile-sequence alignment algorithms.

and 20 additions (20 amino acids plus blank). The processing of the data dependent part on the host CPU contributes to small CUPS values for the matrix approach. Table 4.7 provides more details about CUPS. MatProd V2 achieves about 24 MCUPS on average and clearly outperforms other implementations.

In this section, performance of parallel techniques for profile-sequence alignment has been discussed. A parallel solution of profile-sequence alignment for GPU was proposed by Bassoy et al. [5]. In this study, an improved solution for matrix-vector and matrix-matrix product by avoiding the redundant operations is presented which enables to process larger sequences using profile-sequence alignment. The results exhibits that calculation of data dependent part of the forward table on CPU is a slow operation and it effects the overall performance of the algorithm. The maximum speed-up attained is approximately a factor of 28 using matrix-matrix product.



(a) Sequence length < 1000



(b) Sequence length > 1000

Fig. 4.14: Speed-ups of profile-sequence alignment algorithms.

4.4 ClustalW Algorithm

ClustalW is a widely used progressive alignment method to align multiple sequences. Due to the large computational complexity of the ClustalW algorithm, alignment of multiple sequences can take a huge amount of processing time (Table 2.1). The performance of the ClustalW algorithm can be improved using the parallel architecture of the GPU. In this section, an efficient mapping of the progressive alignment stage of the ClustalW algorithm onto GPU is described.

The guide tree is used as input to the progressive alignment stage. The leaves of the guide tree represent the sequences and are aligned by pairwise sequence alignment. The alignment at the intermediate nodes is obtained by pairwise, profile-sequence, or profile-profile alignment and can only be performed if the left and right sub-trees have aligned sequences. The root of the guide tree corresponds to the overall multiple sequence alignment.

Sections 4.2 and 4.3 provide the techniques to parallelize the profile-profile alignment and profile-sequence alignment. The results show that a mixture of wavefront and matrix-matrix product methods can be useful for the parallelization of the pro-

Table 4.6: Maximum and average speed-ups of profile-sequence alignment algorithms.

Speed-up	**sequence length < 1000**		**sequence length > 1000**	
	max	**mean**	**max**	**mean**
MatVecProd V1	19.8654	13.5625	19.5723	19.0686
MatVecProd V2	27.8019	22.2021	28.4557	25.1844
MatProd V1	26.6299	21.7256	19.7366	18.8212
MatProd V2	34.4063	28.9226	29.8872	26.5587
SMwavefront 256	12.7643	8.5719	19.8531	16.8525

Table 4.7: Maximum and average MCUPS of profile-sequence alignment algorithms.

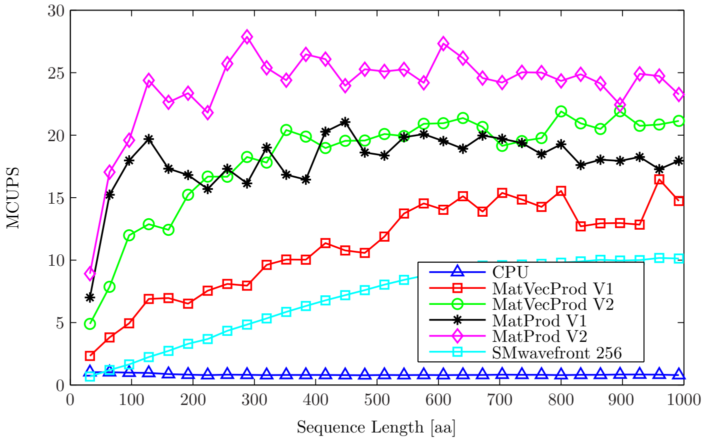
MCUPS	**sequence length < 1000**		**sequence length > 1000**	
	max	**mean**	**max**	**mean**
CPU	1.024	0.8345	0.9143	0.8804
MatVecProd V1	16.4581	11.0688	17.5515	17.1803
MatVecProd V2	21.9147	18.1709	25.5178	22.1639
MatProd V1	21.0505	17.9315	18.0445	16.9695
MatProd V2	27.8728	23.8165	24.8433	23.3179
SMwavefront 256	10.1708	6.9614	16.0842	14.7754

gressive alignment stage of the CLUSTALW algorithm.

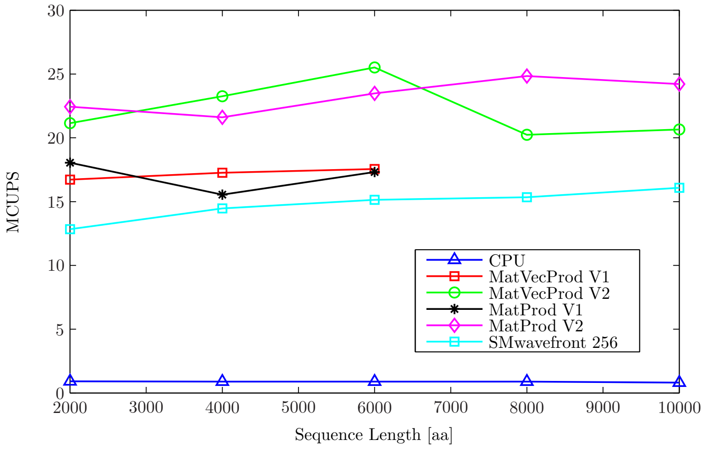
We have used a similar approach mentioned to that in [62]. First, the intermediate nodes of the guide tree are labeled by post-order traversal. Two vectors are used to maintain the right child and left child of the nodes. One flag vector is required to keep track whether the node has been aligned. The flag for the leaf nodes is set to 1 when the alignment is not required. The left and right children indices are 0 for the leaf nodes.

The flag vector is checked to identify the nodes of the guide tree to be aligned. An alignment at an intermediate node can only be performed if the right and left children have been aligned. In first phase, the leaf nodes are aligned using pairwise sequence alignment since the left and right children are assumed to be aligned as indicated in the flag vector. In next phase, the flag vector is again checked to find potential candidates for which alignments can be performed. This process continues until the final multiple sequence alignment is obtained and the flag vector contains 1 for each node.

The frequency based profiles are constructed for the intermediate nodes. A profile and a sequence are aligned by the matrix-matrix product while two profiles are aligned using the wavefront method on the GPU. The traceback is performed on the CPU to find the overall multiple sequence alignment by adding gaps in the aligned sequences.



(a) Sequence length < 1000



(b) Sequence length > 1000

Fig. 4.15: Performance (MCUPS) of profile-sequence alignment algorithms.

4.4.1 Performance

The performance of the parallel progressive alignment stage has been measured by the speed-up. The dataset for the implementation of the ClustalW algorithm is similar to that in [62]. The protein sequence dataset consists of the HIV dataset available at the NCBI database. The dataset has been divided into several subsets:

- 400 sequences of average length 856 and 1000 sequences of average length 858;
- 2000 sequences of average length 266 and 4000 sequences of average length 247;
- 4000 sequences of average length 57 and 8000 sequences of average length 73.

The execution times have been averaged over ten runs for each data subset. The times for memory allocation and data transfer to or from the GPU have been neglected. The input to the progressive alignment stage is a guide tree which has been generated by the ClustalW program from the EMBL-EBI website.

The speed-up for the progressive alignment stage is illustrated in Figure 4.16. Note that profile creation and traceback have been performed on the CPU which

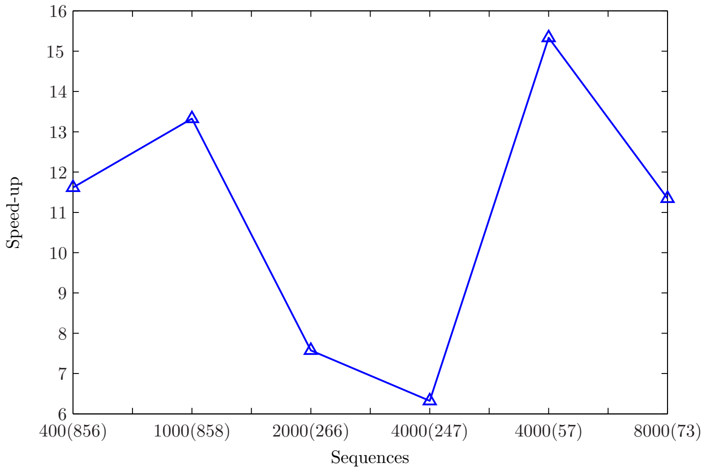


Fig. 4.16: Speed-up for the progressive alignment stage of the ClustalW algorithm.

impacts the performance. The data subsets given by longer sequences have achieved a speed-up of one order of magnitude since the matrices to be multipled utilize the hardware resources rather efficiently. The data subsets given by shorter sequences show a similar behavior due to the possibilty to process multiple sequences in each pass. The computation-to-communication ratio of wavefront approach for sequences and profiles of average length is low which impacts the speed-up exhibited by the data subsets of average length sequences. The maximum speed-up attained is much better than the speed-up of approximately 6 exhibited by progressive alignment stage of MSA-CUDA [62].

This section has provided a parallel algorithm for the progressive alignment stage of the ClustalW algorithm onto the GPU using a mixture of algorithms: matrix-matrix product for profile-sequence alignment and wavefront method for profile-profile alignment. The results have shown a performance increase of more than one order of magnitude for several data sets considered.

Chapter 5

Viterbi Algorithm

The Viterbi algorithm is a popular method used to find the most likely sequence of hidden states which generated the output. In this chapter, we will present a parallel formulation of the Viterbi algorithm in terms of matrix-vector and matrix-matrix product. The resultant matrix-matrix product based algorithm will be implemented on GPU.

5.1 Related Work

The high computational complexity of the hidden Markov model demands improvement in processing time. There have been many efforts to achieve high speed-up for the hidden Markov model applications. A substantial contribution has come forth to speed-up the hidden Markov model based applications by taking advantage of GPUs [8, 16, 20, 44, 47, 99, 120, 127].

CuHMM [59] provides an implementation of discrete hidden Markov model algorithms for GPU. This generic training engine uses single model and multiple observation sequences to achieve 800-fold speed-up for the forward algorithm. In this approach, element-by-element multiplication is performed between the emission probability and the previous states. The resultant matrix is multiplied with the transition probability matrix. Li et al. [56] parallelized the evaluation problem of the hidden Markov model on GPUs. The forward probabilities of an observation sequence were calculated and summed up in parallel to achieve maximum 25-fold speed-up.

ClawHMMER [8, 44] and GPU-HMMER [120] are hidden Markov model based sequence alignment applications that parallelize the general purpose Viterbi algorithm by concurrently executing several sequence alignment tasks. The strategy was to decode the sequence independently using task parallelism. They have used shared and texture memories to store and retrieve probabilities. Ganesan et al. [32] employed a combination of task and data parallel techniques to resolve dependencies in HMMER.

Zhihui et al. [22] proposes a tile-based parallel Viterbi algorithm for biological sequences. The data dependency between different tiles is eliminated by finding the homologous segments. They have also discussed the wavefront and streaming methods for the GPU architecture. Zhang et al. [127] explored the parallel implementation of the Viterbi algorithm for keyword spotting systems and achieved speed-up factor of 3 over serial implementation. They have used multiple parallel threads to compute the transition probability.

Matrix product based solutions for the hidden Markov model problems were discussed by [58, 70, 74, 98]. The Mozes et al. [70] and Lifshits et al. [58] approaches are based on finding the repeated substring and reusing the computation. Nielsen and Sand [74] parallelizes the workload across the observed sequence to achieve a speed-up factor of 7 for the Viterbi algorithm. We have used a similar matrix-matrix product approach for the GPU architecture.

5.2 Parallel Viterbi Algorithm

Improving performance of the Viterbi algorithm is a key task due to its popularity in many scientific applications including biological sequence alignment, speech recognition, and probabilistic inference. By avoiding the dependencies among forward table elements, the matrix-matrix product based solution can be formulated. The recursive part of the algorithm TropViterbi can be reformulated to fit into the parallel architecture of the GPU by separating the data independent and dependent parts. The data independent part is processed on GPU while data dependent is executed on CPU. The values of transition and emission probabilities are known and can be pre-calculated. For this, take n matrices of size $l \times l$ to store these calculated values. For simplicity, these n matrices are stored in a one-dimensional array $\boldsymbol{S}$ where each array element has size $l \times l$. So $n \times l^2$ memory elements are required to store this data.

$$\boldsymbol{S}_k \quad = \quad \begin{pmatrix} t^{'}_{1,1} + e^{'}_{1,\tau_k} & t^{'}_{1,2} + e^{'}_{1,\tau_k} & \cdots & t^{'}_{1,l} + e^{'}_{1,\tau_k} \\ t^{'}_{2,1} + e^{'}_{2,\tau_k} & t^{'}_{2,2} + e^{'}_{2,\tau_k} & \cdots & t^{'}_{2,l} + e^{'}_{2,\tau_k} \\ \vdots & \vdots & \ddots & \vdots \\ t^{'}_{l,1} + e^{'}_{l,\tau_k} & t^{'}_{l,2} + e^{'}_{l,\tau_k} & \cdots & t^{'}_{l,l} + e^{'}_{l,\tau_k} \end{pmatrix}$$

where $1 \leq k \leq n-1$.

By separating the data dependent and independent parts in Eqn. (2.20), we get

$$S_k[\sigma^{'}, \sigma] = t^{'}_{\sigma^{'},\sigma} \odot e^{'}_{\sigma^{'},\tau_k}, \ \sigma^{'}, \ \sigma \in \Sigma, \ 1 \leq k \leq n-1, \tag{5.1}$$

$$M[k, \sigma] = \bigoplus_{\sigma^{'}} (S_k[\sigma^{'}, \sigma] \odot M[k-1, \sigma^{'}]), \ \sigma \in \Sigma, \ 1 \leq k \leq n-1. \tag{5.2}$$

The resultant dynamic programming algorithm is TropScalProdViterbi.

Eqn. (5.2) can easily be written into a matrix-vector product using the tropical matrix-vector multiplication. The k-th row of the matrix $\boldsymbol{M}$ is obtained by matrix-vector multiplication of the $k-1$ row of matrix $\boldsymbol{M}$ with the matrix $\boldsymbol{S}_k$ (Eqn. 5.3). In this way, we get the tropicalized term q_τ which is the probability of the most probable sequence. The resulting algorithm is TropMatVecProdViterbi.

$$M[k, *] = M[k-1, *] \odot S_k, \ 1 \leq k \leq n-1. \tag{5.3}$$

Algorithm 5.1 TropScalProdViterbi($\boldsymbol{\tau}, \boldsymbol{T}^{'}, \boldsymbol{E}^{'}, \boldsymbol{\pi}^{'}$)

```
Require: sequence $\boldsymbol{\tau} \in \Sigma^{'n}$, probabilities $\boldsymbol{T}^{'}$, $\boldsymbol{E}^{'}$, and $\boldsymbol{\pi}^{'}$
Ensure: tropicalized term $q_{\tau}$
$\boldsymbol{M} \leftarrow \text{matrix}[0 \dots n, 1 \dots l]$
for $\sigma \leftarrow 1$ to $l$ do
    $M[0, \sigma] \leftarrow \pi^{'}[\sigma]$
end for
for $k \leftarrow 1$ to $n - 1$ do
    for $\sigma \leftarrow 1$ to $l$ do
        for $\sigma^{'} \leftarrow 1$ to $l$ do
            $S_k[\sigma^{'}, \sigma] \leftarrow T^{'}[\sigma^{'}, \sigma] + E^{'}[\sigma^{'}, \tau_k]$
        end for
    end for
end for
for $k \leftarrow 1$ to $n - 1$ do
    for $\sigma \leftarrow 1$ to $l$ do
        $M[k, \sigma] \leftarrow \infty$
        for $\sigma^{'} \leftarrow 1$ to $l$ do
            $M[k, \sigma] \leftarrow \min\{M[k, \sigma], S_k[\sigma^{'}, \sigma] + M[k - 1, \sigma^{'}]\}$
        end for
    end for
end for
for $\sigma \leftarrow 1$ to $l$ do
    $M[n, \sigma] \leftarrow E^{'}[\sigma, \tau_n] + M[n - 1, \sigma]$
end for
$q_{\tau} \leftarrow \infty$
for $\sigma \leftarrow 1$ to $l$ do
    $q_{\tau} \leftarrow \min\{q_{\tau}, M[n, \sigma]\}$
end for
```

Proposition 5.2.1. *Equations (2.20) and (5.3) generate the same result.*

Proof. k-th row of Eqn. (5.3) is computed as

$$
\begin{aligned}
M[k,*] &= M[k-1,*] \odot \boldsymbol{S}_k \\
&= \begin{pmatrix} m_{k-1,1} & m_{k-1,2} & \dots & m_{k-1,l} \end{pmatrix} \odot \begin{pmatrix} t^{'}_{1,1} + e^{'}_{1,\tau_k} & \dots & t^{'}_{1,l} + e^{'}_{1,\tau_k} \\ t^{'}_{2,1} + e^{'}_{2,\tau_k} & \dots & t^{'}_{2,l} + e^{'}_{2,\tau_k} \\ \vdots & \ddots & \vdots \\ t^{'}_{l,1} + e^{'}_{l,\tau_k} & \dots & t^{'}_{l,l} + e^{'}_{l,\tau_k} \end{pmatrix} \\
&= \Big(\min(t^{'}_{1,1} + e^{'}_{1,\tau_k} + m_{k-1,1}, \dots, t^{'}_{l,1} + e^{'}_{l,\tau_k} + m_{k-1,l}) \dots \min(t^{'}_{1,l} + e^{'}_{1,\tau_k} + \\
&\qquad m_{k-1,1}, \dots, t^{'}_{l,l} + e^{'}_{l,\tau_k} + m_{k-1,l}) \Big) \\
&= \min_{\sigma^{'}} \left(t^{'}_{\sigma^{'},\sigma} + e^{'}_{\sigma^{'},\tau_k} + m_{k-1,\sigma^{'}} \right), \ \sigma^{'}, \ \sigma \in \Sigma, \ 1 \le k \le n-1.
\end{aligned}
$$

Hence, we obtain the k-th row of Eqn. (2.20). ☐

Algorithm 5.2 TropMatVecProdViterbi($\boldsymbol{\tau}, \boldsymbol{T}^{'}, \boldsymbol{E}^{'}, \boldsymbol{\pi}^{'}$)

```
Require: sequence $\boldsymbol{\tau} \in \Sigma^{'n}$, probabilities $\boldsymbol{T}^{'}$, $\boldsymbol{E}^{'}$, and $\boldsymbol{\pi}^{'}$
Ensure: tropicalized term $q_\tau$
$\boldsymbol{M} \leftarrow$ matrix$[0 \dots n, 1 \dots l]$
for $\sigma \leftarrow 1$ to $l$ do
    $M[0,\sigma] \leftarrow \pi^{'}[\sigma]$
end for
for $k \leftarrow 1$ to $n-1$ do
    for $\sigma \leftarrow 1$ to $l$ do
        for $\sigma^{'} \leftarrow 1$ to $l$ do
            $S_k[\sigma^{'},\sigma] \leftarrow T^{'}[\sigma^{'},\sigma] + E^{'}[\sigma^{'},\tau_k]$
        end for
    end for
end for
for $k \leftarrow 1$ to $n-1$ do
    $M[k,*] \leftarrow M[k-1,*] \odot \boldsymbol{S}_k$
end for
for $\sigma \leftarrow 1$ to $l$ do
    $M[n,\sigma] \leftarrow E^{'}[\sigma,\tau_n] + M[n-1,\sigma]$
end for
$q_\tau \leftarrow \infty$
for $\sigma \leftarrow 1$ to $l$ do
    $q_\tau \leftarrow \min\{q_\tau, M[n,\sigma]\}$
end for
```

The calculation part in the algorithm TropMatVecProdViterbi consists of a series of matrix-vector multiplications which can be easily converted into matrix-matrix product. Eqn. (5.3) can also be treated as matrix-matrix multiplication if matrices are multiplied first. The matrix $\boldsymbol{S}$ is populated with the transition and emission probabilities. A series of tropical matrix-matrix multiplications are performed and the result is stored in the $\boldsymbol{S}_1$ matrix. Then tropical matrix-vector multiplication is carried out between the matrix $\boldsymbol{S}_1$ and initial probability vector $\boldsymbol{\pi}^{'}$. Next, the tropical multiplication is performed between the emission probability of the n-th observation and the vector $\boldsymbol{a}$. Finally, we obtain the tropicalized term q_τ by having minimum element in the vector $\boldsymbol{a}$ which is the probability of the most likely sequence of states that generated the observation sequence (Eqns. 5.4–5.8).

$$\begin{aligned}
S_k[\sigma^{'},\sigma] &= t^{'}_{\sigma^{'},\sigma} \odot e^{'}_{\sigma^{'},\tau_k},\ \sigma^{'},\ \sigma \in \Sigma,\ 1 \leq k \leq n-1, && (5.4)\\
\boldsymbol{S}_1 &= \boldsymbol{S}_1 \odot \boldsymbol{S}_k, 2 \leq k \leq n-1, && (5.5)\\
\boldsymbol{a} &= \boldsymbol{\pi}^{'} \odot \boldsymbol{S}_1, && (5.6)\\
\boldsymbol{a} &= e^{'}_{\sigma,\tau_n} \odot \boldsymbol{a},\ \sigma \in \Sigma, && (5.7)\\
q_\tau &= \bigoplus_{\sigma_n} a[\sigma_n]. && (5.8)
\end{aligned}$$

The corresponding algorithm is TropMatProdViterbi. We can formulate the parallel algorithm for evaluation problem of hidden Markov model in a similar manner by replacing minimum with sum operation.

Algorithm 5.3 TropMatProdViterbi($\boldsymbol{\tau}, \boldsymbol{T}^{'}, \boldsymbol{E}^{'}, \boldsymbol{\pi}^{'}$)

```
Require: sequence $\boldsymbol{\tau} \in \Sigma'^{n}$, probabilities $\boldsymbol{T}^{'}$, $\boldsymbol{E}^{'}$, $\boldsymbol{\pi}^{'}$
Ensure: tropicalized term $q_\tau$
for $k \leftarrow 1$ to $n-1$ do
    for $\sigma \leftarrow 1$ to $l$ do
        for $\sigma^{'} \leftarrow 1$ to $l$ do
            $S_k[\sigma^{'},\sigma] \leftarrow T^{'}[\sigma^{'},\sigma] + E^{'}[\sigma^{'},\tau_k]$
        end for
    end for
end for
for $k \leftarrow 2$ to $n-1$ do
    $\boldsymbol{S}_1 \leftarrow \boldsymbol{S}_1 \odot \boldsymbol{S}_k$
end for
$\boldsymbol{a} \leftarrow \boldsymbol{\pi}^{'} \odot \boldsymbol{S}_1$
for $\sigma \leftarrow 1$ to $l$ do
    $a[\sigma] \leftarrow E^{'}[\sigma,\tau_n] + a[\sigma]$
end for
$q_\tau \leftarrow \infty$
for $\sigma \leftarrow 1$ to $l$ do
    $q_\tau \leftarrow \min\{q_\tau, a[\sigma]\}$
end for
```

5.3 Performance

We have implemented the serial and parallel version of the Viterbi algorithm. These implementations are compared by means of execution time and speed-up. For our experiments, we have assumed single precision arithmetic and the generalized Viterbi algorithm. The hidden Markov model can be further optimized for known problems such as speech recognition [91] and the biological sequence alignment [22]. However, these optimizations are out of scope for this work.

In order to evaluate performance, the test data are generated randomly by using MATLAB [68]. The execution times have been averaged over ten runs. Moreover, the time for memory allocation and data transfer to or from the GPU are ignored. Three implementations of the Viterbi algorithm have been considered:

- Viterbi: serial implementation of TropViterbi on Intel CPU.
- SMemViterbi: TropMatProdViterbi on GPU using shared memory.
- TMemViterbi: TropMatProdViterbi on GPU using texture memory.

The number of threads per block is 256. The performance for the above mentioned approaches is evaluated by fixing the state space and varied sequence length and vice versa. We have considered the state space up to 256, maximum 32,768 sequence length. The length of observation alphabet is 8.

TMemViterbi stores the matrices into texture memory and then the sub-matrix is transferred into the shared memory for the matrix multiplication. The MAGMA library [72] provides optimized dense linear algebra operations for the heterogeneous and the GPU architectures. We have modified the MAGMA BLAS routines to accommodate the tropical matrix multiplication for TropMatProdViterbi.

The algorithm TropMatProdViterbi can be implemented using several approaches. One approach is to pre-calculate the transition and emission probabilities and store in the matrix $\boldsymbol{S}$ for the entire sequence length. The matrix $\boldsymbol{S}$ will take $O(nl^2)$ memory space. For the computation on GPU, this matrix should be copied into the GPU memory which limits the algorithm effectiveness. The maximum sequence length and the number of states which can be processed is also limited due to the small size of GPU memory. Therefore, this approach is not considered.

Another approach is to pre-calculate the $\boldsymbol{S}$ matrix and transfer only part of the matrix required for the matrix multiplication. However, there is a limitation of maximum array size which restricts the sequence length to be processed. To accommodate large sequence lengths, the matrices are created at runtime for each sequence using the transition and emission probabilities. In this way, only $O(2l^2)$ memory space is required on GPU for the matrix multiplication.

First, the matrix-matrix product based solution is compared with the optimized serial Viterbi algorithm based on execution time (Fig. 5.1 and 5.2). Both X- and Y-axes are plotted using the logarithm for better display of results. The time to perform the matrix operation using the shared and texture memories is depicted in Figure 5.1. These results are taken by fixing the number of states (8, 16, 32, 64, 128, and 256) and varying sequence length. The execution time of the TMemViterbi include time for the texture binding, unbinding, and movement of the data from texture memory to the shared memory. This result in performance degradation for the TMemViterbi approach. The computational complexity of the serial Viterbi algorithm is $O(nl^2)$ while the matrix-matrix product based algorithm has $O(nl^3)$. There are several methods to perform faster matrix-matrix product [11, 17, 108]. However, such methods for small matrix dimensions are not considered.

The computational complexity and processing overhead for a small number of states results into better or almost the same runtime of Viterbi when compared with SMemViterbi. However, SMemViterbi outperforms the serial version by an order of magnitude for the state space greater than or equal to 32. This is because the number of states (l) are squared for each observation sequence and the arithmetic intensity increases by using a large number of states. At this point, we take advantage of parallel processing and use the parallel matrix-matrix product to attain better running time.

For small state space, SMemViterbi is not a suitable option because the multiplying matrices having small size results in under utilization of the GPU resources. The reason is most of the threads within the thread blocks are idle or do not perform useful work for small matrix dimensions. Moreover, there is also some processing overhead for switching between the CPU and GPU.

The results also show that the increase in the number of matrices to be multiplied leads to an increase in runtime. This means the matrix multiplication takes approximately constant time for each sequence length. Moreover, the average runtime for SMemViterbi is almost same for all states (Table 5.1).

Next, the performance by fixing the sequence length and varying the state space is considered (Fig. 5.2). SMemViterbi performs much better than the serial version for the states greater than or equal to 32. The runtime using SMemViterbi for sequence lengths 2048, 4096, 8192, 16384, and 32768 are almost constant for all states. The runtime for SMemViterbi increases by increasing the number of matrices to be multiplied. TMemViterbi has almost the same time for states up to 32 and it grows linearly as the number of states increases. The runtime is lower for 64 states because the implementation takes two sub-matrices of size 64×16 and 16×64 from the texture memory to the shared memory for multiplication. Table 5.2 provides more detail. Note that increasing the sequence length is directly proportional to the average runtime. This observation is valid for all approaches.

The speed-ups attained with SMemViterbi when compared with the serial Intel CPU implementation have been calculated (Figs. 5.3 and 5.4). In these figures, the X-axis is plotted using the logarithm. First, the speed-up by fixing the number of states and varying sequence length is illustrated (Fig. 5.3). The small number of states i.e., less than or equal to 32, exhibit little speed-up. The maximum average speed-up is approximately 173 (Table 5.3).

Finally, the speed-up by fixing the sequence length and varying state space is presented (Fig. 5.4). For the large states space i.e., greater than or equal to 64, SMemViterbi performs much better because large matrices utilize the GPU resources more efficiently. The maximum average speed-up obtained is approximately 40. The small values are because states less than 64 also contribute to the average speed-up (Table 5.4).

Moreover, the performance of the Viterbi algorithm can be further enhanced by using the associative property of matrix multiplication. In this way, matrices can be multiplied independently of others in a parallel reduction manner. Fermi based GPUs do not support the kernel invocation within a kernel. However, new Kepler based GPUs support this behavior but they were not available at the time of this study.

The Viterbi algorithm is an important algorithm used in many applications and the high computational complexity of the algorithm affects the overall performance of the

Table 5.1: Maximum and average runtime (in milliseconds) of the Viterbi algorithm by fixing number of states.

Number of states	**Viterbi**		**SMemViterbi 256**		**TMemViterbi 256**	
	max	**mean**	**max**	**mean**	**max**	**mean**
8	55.5000	9.9200	178.5012	35.9452	7984.043	1588.5672
16	181.9000	34.9100	182.3307	36.2711	8001.2554	1595.6052
32	656.3000	129.4900	186.5624	37.2636	8139.3359	1616.0280
64	2481.2000	492.3700	193.9977	38.4717	3967.0183	786.0438
128	9708.5000	1929.4900	205.6519	40.7710	15146.2549	3019.8150
256	40078.6016	7995.2602	231.1033	46.0453	63194.5234	12616.0909

Table 5.2: Maximum and average runtime (in milliseconds) of the Viterbi algorithm by fixing sequence length.

Sequence length	**Viterbi**		**SMemViterbi 256**		**TMemViterbi 256**	
	max	**mean**	**max**	**mean**	**max**	**mean**
2048	2494.6001	549.3834	14.3987	11.9195	3937.4878	1097.0655
4096	5003.7002	1095.8834	28.6527	24.0002	7880.8345	2203.3498
8192	10017.5000	2207.4500	57.7612	47.7676	15777.2441	4412.3845
16384	19975.0996	4404.6333	114.6411	97.6949	31580.3711	8843.8451
32768	40078.6016	8860.3336	231.1033	196.3579	63194.5234	17738.7385

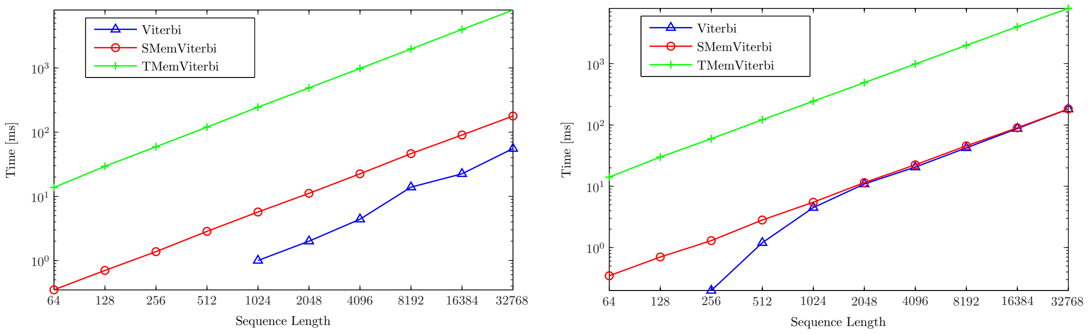
applications. In this chapter, a parallel formulation of the Viterbi algorithm employing matrix-matrix product has been designed. The results depicts matrix-matrix product is not a viable option for small number of states. However, matrix-matrix product solution using shared memory for large number of states gains good performance when compared with the serial version. This is because the hardware resources are better utilized. Note that the speed-up attained is improved by increasing the number of states. The speed-up for states greater than 256 should be investigated on a GPU with large memory space. This approach should be studied to solve evaluation and learning problems of hidden Markov model.

Table 5.3: Maximum and average speed-up of the Viterbi algorithm by fixing number of states.

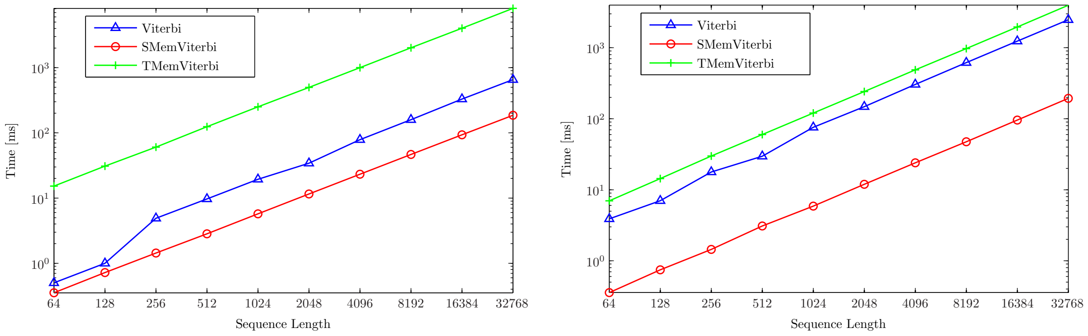
Number of states	**SMemViterbi 256**	
	max	**mean**
8	0.3109	0.1411
16	0.9976	0.6158
32	3.5343	2.9842
64	13.0111	11.8907
128	48.0068	46.0042
256	176.2889	172.4315

Table 5.4: Maximum and average speed-up of the Viterbi algorithm by fixing sequence length.

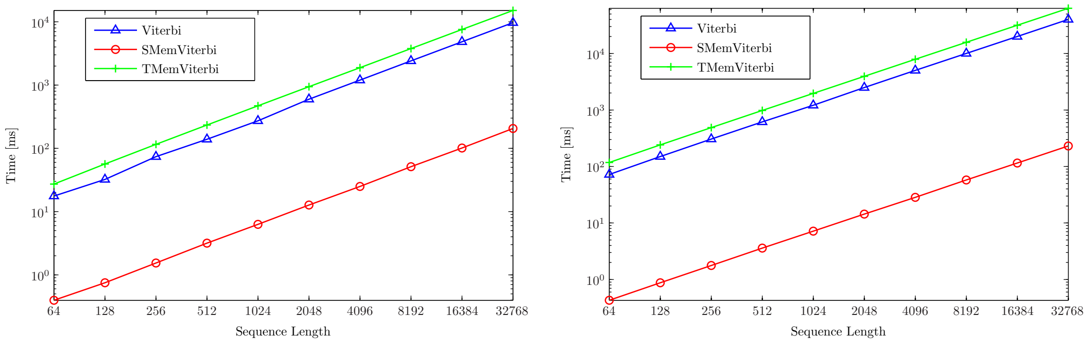
Sequence length	SMemViterbi 256	
	max	**mean**
2048	173.2518	39.6647
4096	174.6328	39.7842
8192	173.4296	40.6272
16384	174.2403	39.7932
32768	173.4229	39.7079



(a) Number of states = 8 (b) Number of states = 16

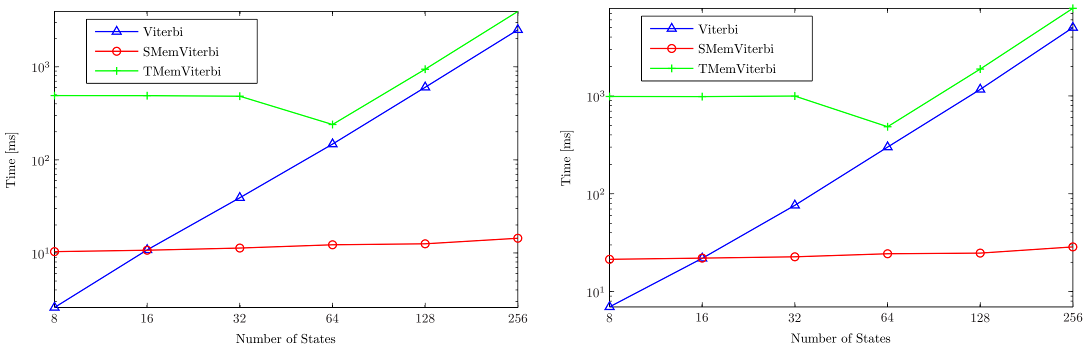


(c) Number of states = 32 (d) Number of states = 64

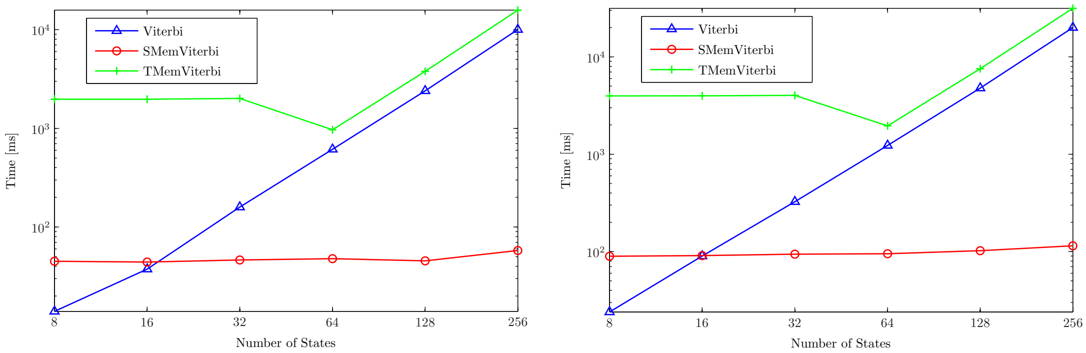


(e) Number of states = 128 (f) Number of states = 256

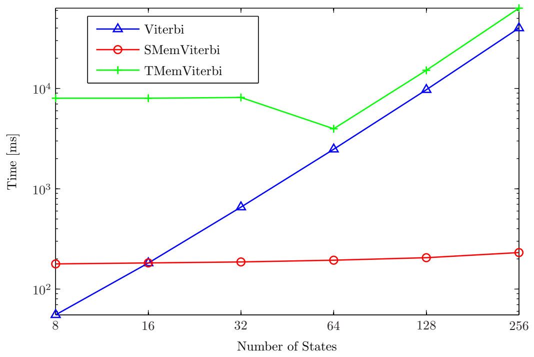
Fig. 5.1: Runtime (in milliseconds) of the Viterbi algorithm using fixed number of states and variable sequence length.



(a) Sequence length = 2048 (b) Sequence length = 4096

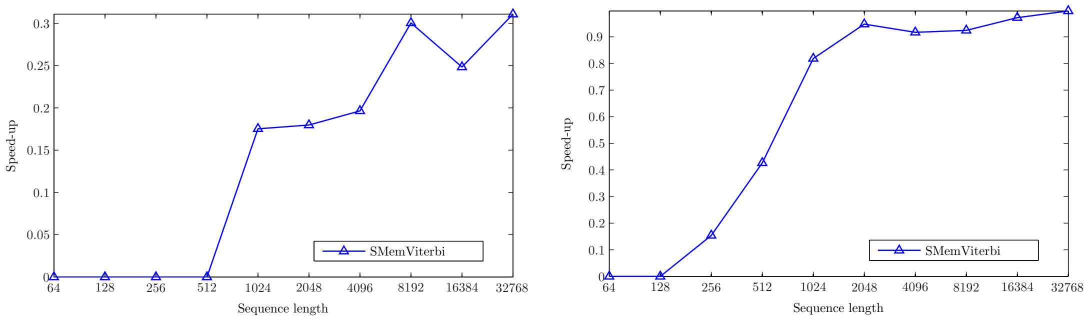


(c) Sequence length = 8192 (d) Sequence length = 16384

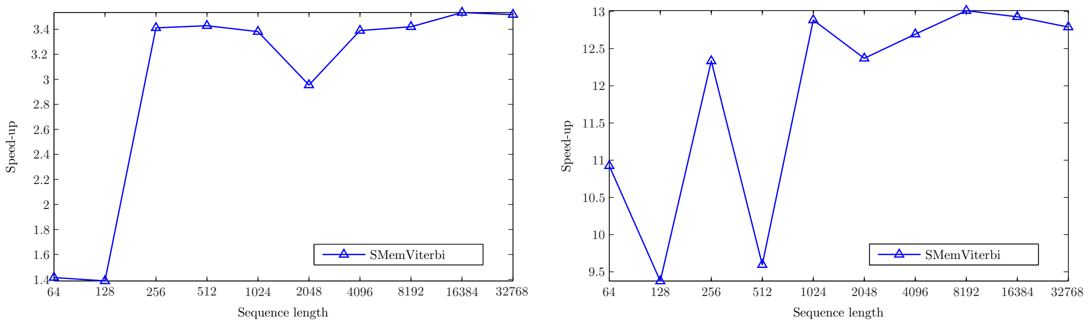


(e) Sequence length = 32768

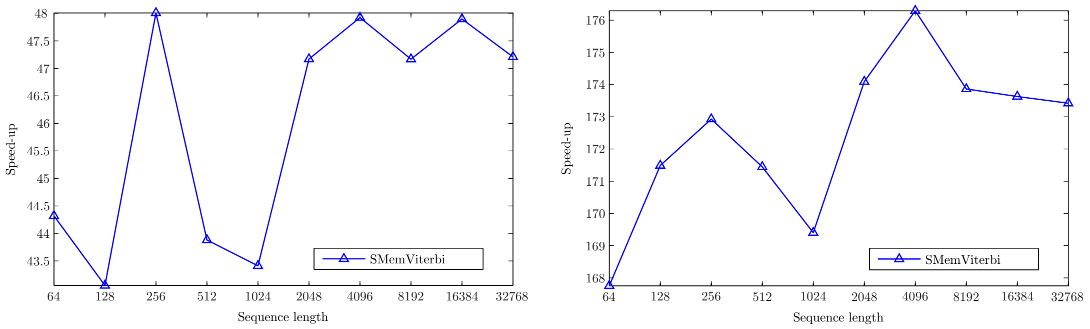
Fig. 5.2: Runtime (in milliseconds) of the Viterbi algorithm using fixed sequence length and variable number of states.



(a) Number of states = 8 (b) Number of states = 16

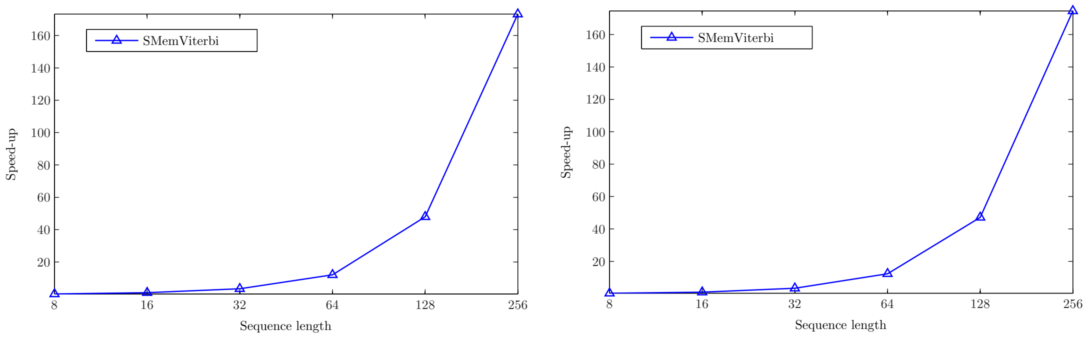


(c) Number of states = 32 (d) Number of states = 64

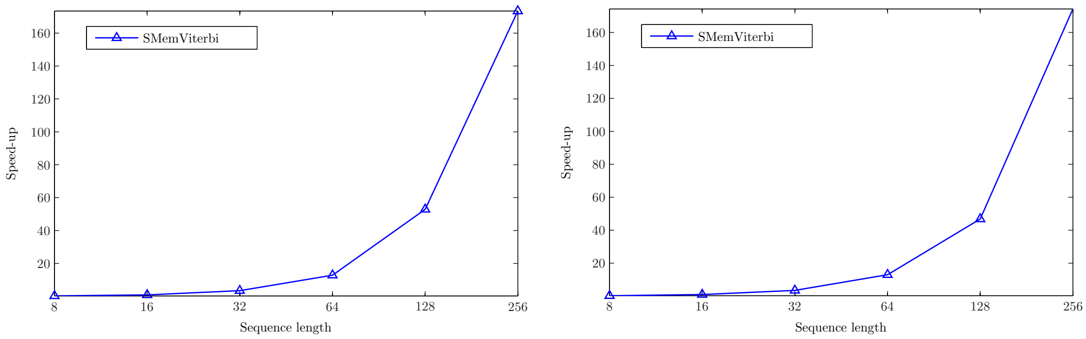


(e) Number of states = 128 (f) Number of states = 256

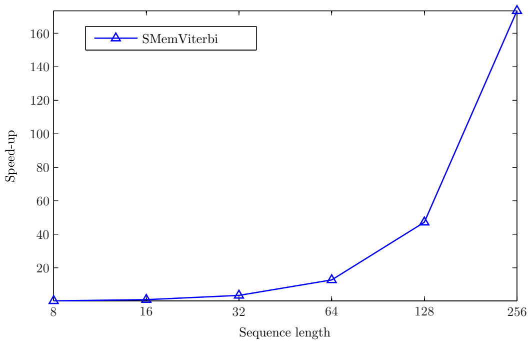
Fig. 5.3: Speed-up for shared memory matrix product version of the Viterbi algorithm using fixed number of states and variable sequence length.



(a) Sequence length = 2048 (b) Sequence length = 4096



(c) Sequence length = 8192 (d) Sequence length = 16384



(e) Sequence length = 32768

Fig. 5.4: Speed-up for shared memory matrix product version of the Viterbi algorithm using fixed sequence length and variable number of states.

Chapter 6

Dynamic Programming

The dynamic programming approach is widely used to solve many optimization problems. The large computational requirements and strong data dependencies limit their usage in applications such as sequence alignment. Many efforts have been made to improve the performance either using distributed systems or devising parallel algorithms for modern multicore processors. In this chapter, we will formulate the parallel solution for the different dynamic programming problems and discuss the techniques to map on the GPU. We will conclude this chapter by presenting some general guidelines for GPU programming.

6.1 Mapping Dynamic Programming Problems on GPU

Grama et al. [34] classified the dynamic programming problems into four categories based on the data dependencies and cost function criteria: SMDP, SPDP, NMDP, and NPDP. In recent years, the GPU has evolved from the traditional graphics processor to a general purpose computing device. The mapping of dynamic programming algorithms to the vector-processing architecture of the GPU can give huge performance boost. We will investigate the parallel dynamic programming algorithm formulation for each category using the matrix-matrix product or the wavefront method.

It is a difficult task to develop a generic parallel solution for each dynamic programming category. However, the problems in each class have certain similarities [34]. Note that all dynamic programming problems cannot be parallelized employing the matrix-matrix product or wavefront methods.

6.1.1 Serial Monadic Dynamic Programming

Many problems can be solved using SMDP. In this section, the Viterbi algorithm is discussed as representative of this class. The Viterbi algorithm is used to solve the decoding problem of hidden Markov model. The recursive step of the Viterbi algorithm having l states (Eqn. 2.20) is shown in Figure 6.1. Each state at level k is connected to every state at level $k+1$. This problem finds the maximum likelihood of the states that generated the observation sequence. The probability of moving to the next state depends on the current state, the transition probability, and the emission probability.

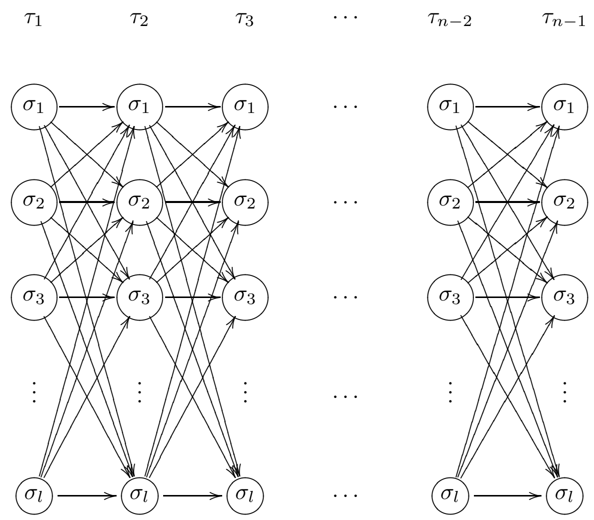


Fig. 6.1: The recursive step of the Viterbi algorithm.

The optimal state sequence that generated the observation sequence is obtained since the sub-problems calculate the optimal solution. This problem is monadic since Eqn. (2.20) has only one recursive term in its right-hand side. Moreover, it is serial because the solution to a sub-problem requires solutions to the sub-problems at the immediate previous level.

Due to the recursive term, it is hard to derive a parallel matrix-matrix product based formulation. However, separating the data dependent and independent parts can be helpful. The transition and emission probabilities are known and can be pre-calculated and stored in matrix for each sequence character as shown in Eqn. (6.1). The matrix S requires $O(nl^2)$ memory space. We have discussed the technique to reduce the memory requirements in Section 5.3.

$$S_k[\sigma^{'},\sigma] = t^{'}_{\sigma^{'},\sigma} \odot e^{'}_{\sigma^{'},\tau_k},\ \sigma^{'},\ \sigma\epsilon\Sigma,\ 1 \leq k \leq n-1. \tag{6.1}$$

Next, these $n-1$ matrices are multiplied from left-to-right on GPU. Furthermore, the performance of this process can be improved using the associativity property of matrix multiplication and parallel reduction.

$$\boldsymbol{S}_1 = \boldsymbol{S}_1 \odot \boldsymbol{S}_2 \odot \boldsymbol{S}_3 \odot \boldsymbol{S}_4 \odot \cdots \odot \boldsymbol{S}_{n-1}. \tag{6.2}$$

The detailed matrix-matrix product based algorithm is presented in Section 5.2. We cannot use CUBLAS matrix operations as our algorithm uses tropical algebra. For this, we have written a kernel function which uses shared memory. The code from the MAGMA library BLAS routines [72] has been modified to accommodate tropical matrix multiplication. The maximum average speed-up attained is approximately 173 (Table 5.3).

6.1.2 Non-serial Monadic Dynamic Programming

The dynamic programming algorithms for profile-sequence and profile-profile alignments fall into this category.

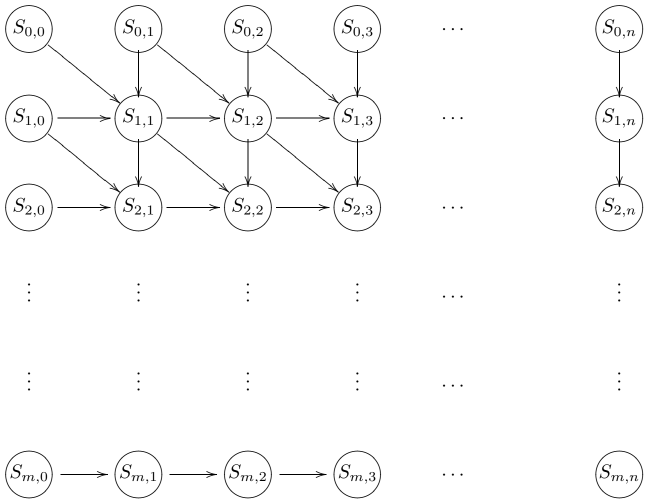


Fig. 6.2: The data dependencies in profile-sequence alignment.

Profile-Sequence Alignment

Profile-sequence alignment combines a sequence with a profile which is statistical representation of an alignment. Given a profile $\boldsymbol{O}$ and a sequence $\boldsymbol{x}$, the objective is to determine the optimal value $S_{m,n}$ [128]. Eqn. (6.3) shows the dynamic programming formulation for this problem, where $\sigma(a, b)$ is the similarity score. There are different scoring matrices such as BLOSUM and PAM which can be used to calculate scores for matches, mismatches, and gaps. The sequential implementation of profile-sequence alignment computes the forward table in row-major order. The calculation of the entry $S_{i,j}$, for $1 \leq i \leq m$ and $1 \leq j \leq n$, requires the $S_{i-1,j-1}$, $S_{i-1,j}$, and $S_{i,j-1}$ values. For example, $S_{1,1}$ depends on cells $S_{0,0}$, $S_{0,1}$, and $S_{1,0}$. These dependencies should be avoided to devise a parallel solution (Fig. 6.2).

$$S_{i,j} = \begin{cases} 0, & \text{if } i = 0 \text{ and } j = 0, \\ \sum_{k=1}^{i} \sigma(\boldsymbol{o}_k, -), & \text{if } 1 \leq i \leq m \text{ and } j = 0, \\ \sum_{k=1}^{j} \sigma(-, x_k), & \text{if } 1 \leq j \leq n \text{ and } i = 0, \\ \max\{S_{i-1,j-1} + \sigma(\boldsymbol{o}_i, x_j), S_{i-1,j} + \\ \sigma(\boldsymbol{o}_i, -), S_{i,j-1} + \sigma(-_o, x_j)\}, & \text{if } 1 \leq i \leq m \text{ and } 1 \leq j \leq n. \end{cases} \tag{6.3}$$

The anti-diagonal entries of the forward table can be processed independently (Fig. 6.3). To compute the elements at the k-th anti-diagonal, we require $k-1$ and $k-2$ anti-diagonals. The formulation is monadic since a solution to any sub-problem at k-th level is a function of only one of the solutions at the previous levels and non-serial because each table entry depends on two sub-problems ($S_{i,j-1}$ and $S_{i-1,j}$) at the $k-1$ level and one sub-problem $S_{i-1,j-1}$ at the $k-2$ level.

In order to parallelize the profile-sequence alignment, the non-serial dynamic programming formulation should be transformed into the serial one. Profile-sequence alignment can be parallelized using two techniques: matrix-matrix product and wavefront. In the matrix-matrix product method, the data dependent and data independent parts are separated [5]. Using the wavefront method, anti-diagonal elements are parallelized [39, 90]. Section 4.3 discusses both methods in detail. The results exhibit that

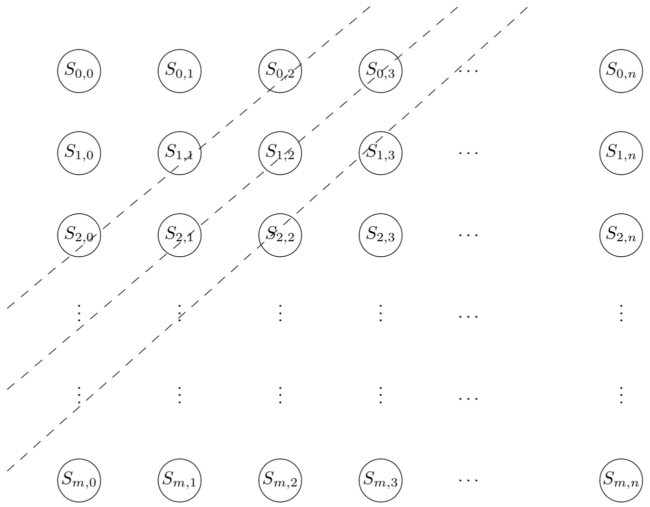


Fig. 6.3: The anti-diagonal data independence for profile-sequence alignment.

the matrix-matrix product method has better speed-up than the wavefront method by using the optimized BLAS routines.

Profile-Profile Alignment

Profile-profile alignment is a technique to align two alignments. Consider two profiles $\mathbf{P}$ and $\boldsymbol{Q}$, the objective is to find the optimal value $S_{m,n}$. The score between the profiles is calculated using the Euclidean distance [128]. Many other scoring schemes exist [84]. Eqn. 6.4 provides the dynamic programming formulation of the problem. Profile-profile alignment exhibit a similar structure to profile-sequence alignment and belongs to the NMDP class.

$$S_{i,j} = \begin{cases} 0, & \text{if } i = 0 \text{ and } j = 0, \\ \sum_{k=1}^{i} d(\boldsymbol{p}_k, -_p), & \text{if } 1 \leq i \leq m \text{ and } j = 0, \\ \sum_{k=1}^{j} d(-_p, \boldsymbol{q}_k), & \text{if } 1 \leq j \leq n \text{ and } i = 0, \\ \begin{array}{l} \min\{S_{i-1,j-1} + d(\boldsymbol{p}_i, \boldsymbol{q}_j), S_{i-1,j} + \\ d(\boldsymbol{p}_i, -_p), S_{i,j-1} + d(-_p, \boldsymbol{q}_j)\}, \end{array} & \text{if } 1 \leq i \leq m \text{ and } 1 \leq j \leq n. \end{cases} \tag{6.4}$$

Section 4.2 provides more details to parallelize this problem using the matrix-matrix product and wavefront techniques. The matrix-matrix product based solution for profile-profile alignment is expensive in terms of computation and memory, since only the diagonal entries are of interest. The results show that the wavefront method using shared memory with block size 256 is a very good candidate for the implementation of profile-profile alignment on a GPU and attains a performance increase of more than one order of magnitude.

6.1.3 Serial Polyadic Dynamic Programming

This section discusses algorithms for finding the shortest paths between all pairs of vertices in general and bipartite graphs.

Floyd-Warshall Algorithm

An algorithm to find all-pairs shortest paths for a weighted graph was introduced by Floyd [30] and extended by Warshall [121]. Let $\mathbf{G} = (\boldsymbol{V}, \boldsymbol{E})$ be a graph with vertex set $\boldsymbol{V}$ and edge set $\boldsymbol{E}$. The graph $\mathbf{G}$ can be stored in an adjacency matrix or an adjacency list. Normally, the adjacency matrix is used to store dense graphs while the adjacency list is used for sparse graphs. The edges between the vertices can have weights and these weights can be stored in a weight matrix $\boldsymbol{M} = (m_{ij})$. The weight matrix entry m_{ij} is

$$m_{i,j} = \begin{cases} 0, & \text{if } i = j, \\ \infty, & \text{if there is no edge between } v_i \text{ and } v_j \text{ and } i \neq j, \\ w_{ij}, & \text{otherwise.} \end{cases} \tag{6.5}$$

The recurrence for the APSP problem is

$$d^{(k)}_{i,j} = \begin{cases} w(v_i, v_j), & \text{if } k = 0 \\ \min(d^{(k-1)}_{i,j}, d^{(k-1)}_{i,k} + d^{(k-1)}_{k,j}), & 1 \leq k \leq n. \end{cases} \tag{6.6}$$

The cost of overall shortest path from node i to j using all n nodes is given by $d^{(n)}_{i,j}$. This problem belongs to the serial polyadic class of dynamic programming (Eqn. 6.6). The formulation is serial since the nodes at level $k + 1$ depend only on the nodes at level k and polyadic since the solution to $d^{(k)}_{i,j}$ requires a composition of solution of two sub-problems $d^{(k-1)}_{i,k}$ and $d^{(k-1)}_{k,j}$. Each level requires $O(n^2)$ time. However, only three results are required from the previous level for the computation of each element $d^{(k)}_{i,j}$ [34].

The algorithm FloydWarshall solves the APSP problem. The dynamic programming matrix $\boldsymbol{D}$ is initialized with the weight matrix $\boldsymbol{M}$. At every step k, the element $d_{i,j}$ has the shortest path from the node v_i to v_j using the intermediate nodes from v_1 to v_k. The time complexity of the algorithm is $O(n^3)$.

Algorithm 6.1 FloydWarshall($\boldsymbol{M}$)

```
Require: weight matrix: $M$
Ensure: APSP matrix $D$
$D \leftarrow M$                                  // initialization
for $k \leftarrow 1$ to $n$ do
    for $i \leftarrow 1$ to $n$ do
        for $j \leftarrow 1$ to $n$ do
            $d^{(k)}_{i,j} \leftarrow min(d^{(k-1)}_{i,j}, d^{(k-1)}_{i,k} + d^{(k-1)}_{k,j})$
        end for
    end for
end for
return $D$
```

Many efforts have been made to improve the complexity of the APSP problem [11, 21, 45, 94, 101, 109, 110]. Several researchers have used GPU to accelerate the performance of the Floyd-Warshall algorithm [40, 49, 66, 85]. Romani [95] presented the

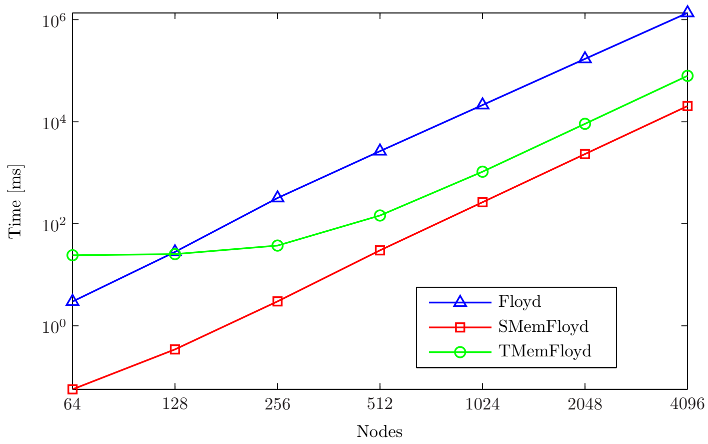


Fig. 6.4: Runtime (in milliseconds) of FloydWarshall algorithm on NVIDIA and Intel CPU.

matrix multiplication solution for the APSP problem. In this work, we have used matrix multiplication based solution for the APSP problem on GPU. The tropical matrix multiplication of weight matrix with itself $n-1$ times gives the solution of APSP problem. However, repeated squaring technique reduces the time complexity of the APSP problem to $O(n^3 \log_2(n))$.

$$\boldsymbol{D} = \underbrace{\boldsymbol{M} \odot \boldsymbol{M} \odot \boldsymbol{M} \odot \cdots \odot \boldsymbol{M}}_{n-1 \text{ times}} = \boldsymbol{M}^{\odot n-1} \tag{6.7}$$

Three versions of FloydWarshall algorithm have been considered.

- Floyd: serial implementation of FloydWarshall on Intel CPU.
- SMemFloyd: APSP using shared memory on GPU.
- TMemFloyd: APSP using texture memory on GPU.

The performance is measured using runtime and speed-up. The weighted directed graphs are generated at random. Execution times have been averaged over ten runs for each graph. X- and Y-axis are plotted using the logarithm. The shared memory solution SMemFloyd performs an order of magnitude better than the serial FloydWarshall (Fig. 6.4). This is just a simple mapping of the matrix-matrix product on GPU. Efficient matrix multiplication algorithms can be used to further enhance the performance [11, 17, 108].

APSP for Bipartite Graphs

A bipartite graph is a graph whose vertex set can be partitioned into two non-empty subsets such that every edge in one subset connects to a vertex in the other (or vice versa for directed graphs). There is no edge between vertices of a subset. The bipartite graphs are commonly used in computational biology and computer science for modeling of complex networks.

Consider the bipartite graph $\mathbf{G} = (\boldsymbol{V}, \boldsymbol{E})$ with the vertex set $\boldsymbol{V} = \boldsymbol{V}_1 \cup \boldsymbol{V}_2$. Let $n_1 = |\boldsymbol{V}_1|$ and $n_2 = |\boldsymbol{V}_2|$. The structure of the weight matrix for the bipartite graph is

$$\boldsymbol{M} = \begin{pmatrix} \boldsymbol{U}_1 & \boldsymbol{M}_1 \\ \boldsymbol{M}_2 & \boldsymbol{U}_2 \end{pmatrix}.$$

$\boldsymbol{M}_1$ and $\boldsymbol{M}_2$ are $n_1 \times n_2$ and $n_2 \times n_1$ matrices, respectively. $\boldsymbol{U}_1$ and $\boldsymbol{U}_2$ are tropical identity matrices of size $n_1 \times n_1$ and $n_2 \times n_2$, respectively.

$$\boldsymbol{U}_1 = \begin{pmatrix} 0 & \infty & \cdots & \infty \\ \infty & 0 & \ddots & \vdots \\ \vdots & \ddots & \ddots & \infty \\ \infty & \cdots & \infty & 0 \end{pmatrix}.$$

Torgasin and Zimmermann [116] have designed an APSP algorithm for the bipartite graphs using tropical matrix multiplication. This problem also belongs to the serial polyadic class of dynamic programming since the recurrence equation is similar to the Floyd-Warshall algorithm. The algorithm BIPARTITEAPSP takes two blocks of the weight matrix as input. The dynamic programming matrix $\boldsymbol{D}$ is initialized with $\boldsymbol{M}_1 \odot \boldsymbol{M}_2 \oplus \boldsymbol{U}_1$. This matrix is multiplied with itself $n_1 - 1$ times and is used to calculate the four blocks of the weight matrix.

Algorithm 6.2 BIPARTITEAPSP($\boldsymbol{M}_1, \boldsymbol{M}_2$)

```
Require: two blocks of weight matrix: $\boldsymbol{M}_1, \boldsymbol{M}_2$
Ensure: four blocks of $\boldsymbol{M}^{\odot 2n_1}$: $\boldsymbol{M}_1^{(2n_1)}, \boldsymbol{M}_2^{(2n_1)}, \boldsymbol{L}_1^{(2n_1)}, \boldsymbol{L}_2^{(2n_1)}$
$\boldsymbol{D} \leftarrow \boldsymbol{U}_1$
$\boldsymbol{D}_1 \leftarrow (\boldsymbol{M}_1 \odot \boldsymbol{M}_2 \oplus \boldsymbol{U}_1)$
for $k \leftarrow 1$ to $n_1 - 1$ do
    $\boldsymbol{D} \leftarrow \boldsymbol{D} \odot \boldsymbol{D}_1$
end for
$\boldsymbol{L}_1^{(2n_1)} \leftarrow \boldsymbol{D} \odot \boldsymbol{D}_1$
$\boldsymbol{L}_2^{(2n_1)} \leftarrow \boldsymbol{M}_2 \odot \boldsymbol{D} \odot \boldsymbol{M}_1 \oplus \boldsymbol{U}_2$
$\boldsymbol{M}_1^{(2n_1)} \leftarrow \boldsymbol{D} \odot \boldsymbol{M}_1$
$\boldsymbol{M}_2^{(2n_1)} \leftarrow \boldsymbol{M}_2 \odot \boldsymbol{L}_1^{(2n_1)}$
return $\boldsymbol{L}_1^{(2n_1)}, \boldsymbol{L}_2^{(2n_1)}, \boldsymbol{M}_1^{(2n_1)}, \boldsymbol{M}_2^{(2n_1)}$
```

The time complexity of the algorithm BIPARTITEAPSP is $O(n_1^3 \log_2(n_1))$. In the worst case, this method is at least eight times faster and requires four times less space than the APSP method.

We have considered three versions of the BIPARTITEAPSP algorithm for implementation.

- Bipartite: serial implementation of BIPARTITEAPSP on Intel CPU.
- SMemBipartie: BIPARTITEAPSP using shared memory on GPU.
- TMemBipartie: BIPARTITEAPSP using texture memory on GPU.

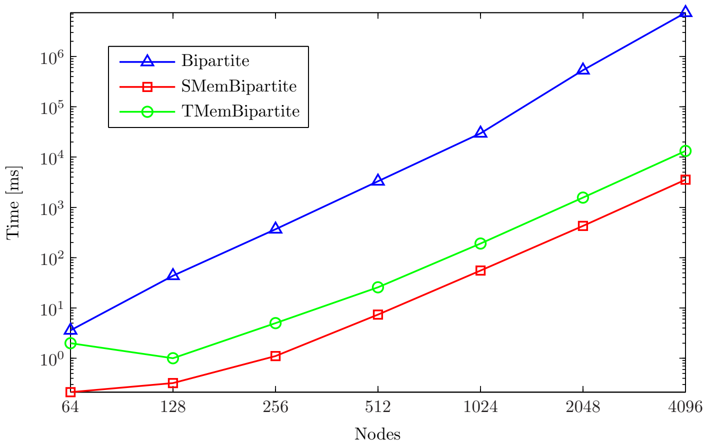


Fig. 6.5: Runtime (in milliseconds) of BIPARTITEAPSP algorithm on NVIDIA and Intel CPU.

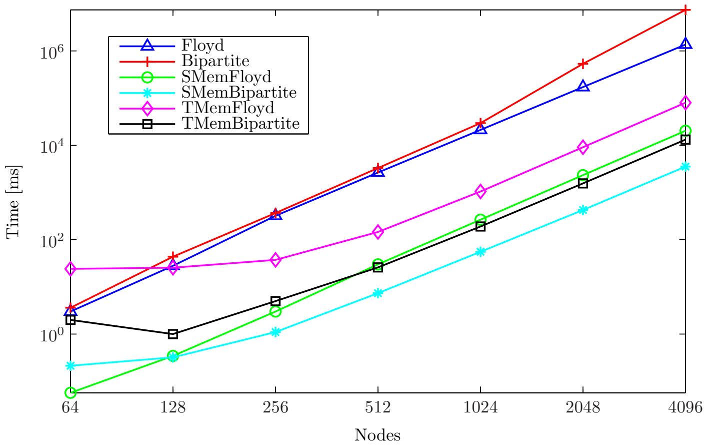


Fig. 6.6: Runtime comparison of FLOYDWARSHALL and BIPARTITEAPSP algorithms for bipartite graphs on NVIDIA and Intel CPU.

The graphs are generated randomly and the runtime is averaged over ten executions. We have considered the worst case for the experiment i.e., $n_1 = n_2$ and the weight matrix is generated accordingly.

First, the runtime of the BIPARTITEAPSP algorithm on NVIDIA and Intel CPU have been computed (Fig. 6.5). Both X- and Y-axis are plotted using the logarithm for better display. SMemBipartite performs much better than the serial Bipartite. Next, the performance of FLOYDWARSHALL and BIPARTITEAPSP for bipartite graphs is compared (Fig. 6.6). On average, the performance of SMemBipartite is almost six times faster than SMemFloyd. Theoretically, it should be at least eight times faster but the additional matrix multiplication to compute blocks of the distance matrix contribute to the overall running time. The results hold for texture memory based matrix multiplication. The performance of Bipartite is worst than Floyd. Although, the size of dynamic programming matrix for Bipartite is four times smaller i.e., $n^2/4$, in worst case, but the additional factor of $\log_2(n_1)$ impacts the performance (Table 6.1).

Table 6.1: Maximum and average runtime (in milliseconds) of BIPARTITEAPSP and FLOYDWARSHALL algorithms.

Runtime	**max**	**mean**
Floyd	1363207.6250	222588.6027
Bipartite	7436817.0000	1143393.3300
SMemFloyd	20360.4434	3284.7355
SMemBipartite	3544.9109	576.6045
TMemFloyd	79665.6016	12863.0573
TMemBipartite	13234.9004	2145.8429

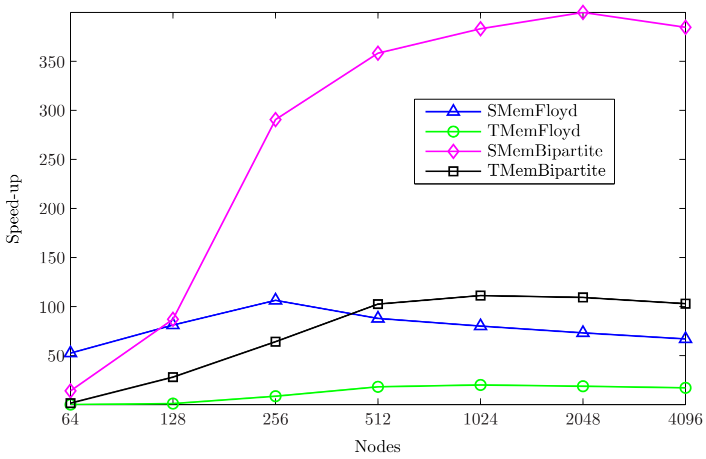


Fig. 6.7: Speed-up of BIPARTITEAPSP over FLOYDWARSHALL algorithm for bipartite graphs.

Finally, the speed-up attained with the GPU based APSP implementations for bipartite graphs when compared with the serial Floyd have been calculated (Fig. 6.7). SMemBipartite and TMemBipartite performs better than SMemFloyd and TMemFloyd. The time complexity for the matrix-matrix product based APSP method is $O(n^3 \log_2(n))$ which is at least eight times slower than BIPARTITEAPSP. Table 6.2 provides more details. Maximum average speed-up attained for SMemBipartite is approximately 274.

6.1.4 Non-serial Polyadic Dynamic Programming

This is the most complex dynamic programming class. In this class, the data dependencies are non-uniform. The data dependencies between states are dynamic and not restricted to immediate previous level but may exist between non-consecutive levels [119]. For most of the NPDP applications, the dynamic programming matrix is triangular which results in difficult memory optimization and load balancing [112]. The structure and dependencies of the dynamic programming matrix for the Zuker algorithm are shown in Figure 6.8.

Table 6.2: Maximum and average speed-up parallel APSP algorithms over serial FLOYDWARSHALL algorithm for bipartite graphs.

Speed-up	**max**	**mean**
SMemFloyd	106.3428	78.2893
TMemFloyd	20.1548	11.9984
SMemBipartite	399.8470	273.9073
TMemBipartite	111.1152	74.2239

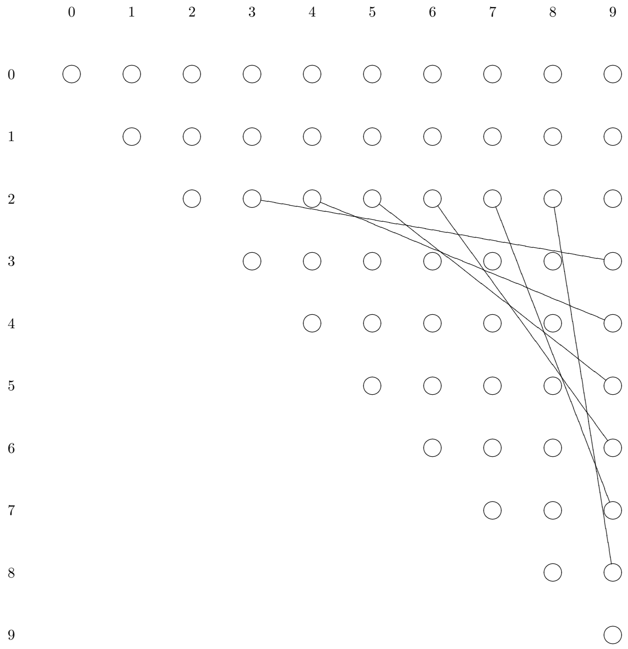


Fig. 6.8: An example of the NPDP class [60].

This class of dynamic programming is hard to parallelize. Many efforts have been made recently for different NPDP problems [93, 107, 112, 125, 126]. Some applications of NPDP are Zuker algorithm for RNA secondary structure prediction, optimal matrix chain problem, and Smith-Waterman alignment algorithm with affine gap penalty. Due to its non-uniformity, it is difficult to design a matrix-matrix product based algorithm for the Smith-Waterman algorithm with affine gap penalty. However, Xiao et al. [126] already implemented the Smith-Waterman algorithm onto GPU using the wavefront method. For efficient memory access, they have employed matrix re-alignment, coalesced memory access, and tiling to increase computational granularity.

6.2 General guidelines for GPU programming

In this section, we present some general guidelines for GPU programming which were observed during this work. Detailed optimization strategies for GPU programming can be found in [83].

Matrix Storage Layout

If a problem can be transformed into the matrix-matrix product, then it is highly recommended to use the existing optimized CUBLAS library for the matrix operations instead of writing unoptimized code whenever possible. However, for tropical matrix operations, no BLAS library exists. The CUBLAS library uses column-major storage and 1-based indexing. `C` and `C++` use row-major storage [79]. Therefore, it is advisable to change the matrix storage layout to fully utilize the functionality.

For some applications, changing the data storage layout can improve the performance and some unnecessary operations can be avoided. Xiao et al. [126] used matrix re-alignment for the Smith-Waterman algorithm for efficient memory access. They have stored the forward matrix in diagonal-major order instead of row-major order so that the threads within a block access adjacent memory locations.

Number of Blocks and Threads

The choice of the number of blocks and threads plays an important role to keep the GPU busy and to utilize the hardware resources in efficient manner. The Fermi architecture based GPU used for this study has compute compatibility 2.1. The compute compatibility determine the device architecture. The devices with the same compute compatibility have similar architecture [80]. The GPU has 8 multiprocessors and each multiprocessor has 48 CUDA cores. The Fermi architecture supports at most 1536 threads per SM, eight blocks can be scheduled simultaneously per SM, and 1024 threads per block. The aim of the block size choice is to maximize occupancy. Each thread block is entirely scheduled to a single SM.

- If a block has 64 threads, then we require 24 blocks ($1536/64 = 24$) to fully utilize the resources of an SM. However, the limit of eight blocks per SM restricts the maximum occupancy to 33%. This is because only eight blocks with each block having 64 threads can be scheduled to an SM. So 512 threads can be executed in parallel per SM.
- If a block has 192 threads then 8 blocks ($1536/192 = 8$) can be scheduled per SM. So maximum occupancy can be achieved.
- If a block has 256 threads then 6 blocks ($1536/256 = 6$) can be scheduled per SM to obtain maximum occupancy.

The advantage of multiple blocks per SM is to keep the SM busy while some blocks are in waiting state. However, multiple blocks per SM require more registers and shared memory resources. The number of threads per block should be a multiple of 32 since processors execute simultaneously 32 threads in a warp. The maximum warps per SM are 48 since $48 \cdot 32 = 1576$. There should be at least one block scheduled to execute for each SM [83].

Dynamic Parallelism

Dynamic parallelism is an extension to the CUDA programming model that enables to call a child kernel from a parent kernel function. There is no requirement to involve the

CPU which is a costly procedure as it incurs delay. Previously, CUDA library routines cannot be called within a kernel or a kernel cannot call another kernel function. This restricts the usage of already developed optimized procedures. With the introduction of the new Kepler architecture which supports dynamic parallelism, this restriction is eliminated [82].

For example, to multiply $\boldsymbol{N}$ matrices using the CUBLAS library function `gemm`, $N-1$ times the library function is executed. By using the associative property of matrix multiplication, we can parallelize the process in a parallel reduction manner. However, the Fermi architecture does not allow to execute BLAS library routines within parallel reduction kernel.

Memory Optimization

The data transfer between host and device should be minimized due to low bandwidth. Sometimes, it is beneficial to re-calculate the data instead of transferring from the host. The techniques to achieve higher bandwidth for the data transfer between host and device is discussed in [83].

Due to higher bandwidth and lower latency, the use of shared memory is highly recommended. However, memory bank conflicts can impact the performance. In this case, memory access is serialized and hardware splits memory requests into separate conflict-free requests which results in decrease of effective bandwidth [83].

Constant memory is read-only memory and it is cached. When all the threads in a warp read the same memory location, the speed of constant memory is like that of registers. However, warp costs more if threads read from different locations of the constant memory [83].

Texture memory is read-only and bound to global memory before kernel launch. The threads of the same warp accessing memory that are close together will achieve good performance. The global memory locations written by current kernel executions will return undefined data for a texture fetch. However, this is not valid if the memory location is updated by previous kernel execution [83].

Warp Divergence

Threads are executed in warps of 32. All threads within a warp execute the same instruction at the same time. However, if the threads within warps follow different execution paths, the SM serializes the warp execution by keeping the unused threads within the warp idle. The warp divergence is usually due to `if` or `case` statements. It can have impact on the overall performance because resources are wasted. In the worst case, the performance can be degraded to a factor of 32 if one thread is diverged and others are idle [83].

Tiling

The applications that use shared memory achieve significant performance improvement. The problem with the GPU shared memory is its limited size. In order to maximize performance, data can be partitioned into tiles so that each tile can be placed into fast shared memory. The tiles can be processed by kernel functions independent of other tiles. However, some data structures cannot be partitioned into tiles for a given kernel function. In this thesis, we have used tile-based matrix multiplication solution for the applications.

Coalesced Memory Access

Coalesced memory access is very important to reduce memory transactions. Significant performance reductions can result if memory access is not coalesced, causing extra memory transactions. Memory reads or writes of multiple data items into few transactions help to improve performance of applications. Memory coalescing techniques are discussed in [83].

Built-in Functions

Use the math functions provided by CUDA. For example, taking minimum of two numbers can be defined by a macro in `C`. However, it can result in thread divergence when used in a kernel function. CUDA supports two types of math operations. The functions whose names are prefixed with underscore are faster but less accurate. The functions whose names does not start with underscore are slower but more accurate [83].

Chapter 7

Conclusion

Tremendous amount of speed-up can be achieved by mapping applications on a low-cost GPU. In this work, huge processing power of GPUs has been used to accelerate the dynamic programming algorithms which exhibit massive data parallelism. The performance of the dynamic programming algorithms can be enhanced by redesigning the algorithms that are suitable for the vector-processing architecture of the GPU. For this, matrix-matrix product and wavefront methods have been studied. Matrix-matrix product method works by separating the data independent and dependent parts while wavefront method calculates all entries in each anti-diagonal at once. The optimized CUBLAS library routines for the matrix operations are used to achieve better efficiency whenever possible.

The conversion of dynamic programming algorithms into matrix-matrix product is not always straightforward due to data dependencies. Other factors such as memory requirement also limit the opportunities for matrix-matrix product based solution. For this, we have discussed the wavefont method. However, the physical GPU memory is a limiting factor to handle long sequences.

Dynamic programming applications have been categorized into four classes corresponding to data dependence and recurrence to find optimal solutions for the subproblems [34]. We have presented parallel formulations for each class of the dynamic programming using matrix multiplication. The applications which exhibit non-serial data dependencies need to be transformed into serial data dependencies. However, it is hard to devise a matrix-matrix product solution for the NPDP problems due to non-serial data dependencies. The development of a general framework for all the problems belonging to a class is not possible. For example, we have designed the matrix-matrix product solution for profile-profile alignment but the memory requirement limits its efficiency.

At first, we have used the GPU to improve the performance of progressive sequence alignment methods. The profile-sequence and profile-profile alignments are members of the NMDP class. Profile-profile alignment algorithm is redesigned to map onto GPU architecture. One approach used is to design an algorithm by separating the data dependent and independent parts. The data independent part is processed using matrix-matrix product and result of these matrix operations are used in the data dependent part to calculate the forward table. The wavefront approach exploits the data independence along the anti-diagonals of the forward table for profile-profile alignment and processes the anti-diagonal elements in parallel. The results exhibit that modern graphics cards can be utilized as efficient hardware accelerators for profile-profile alignment. The wavefront approach has better hardware utilization and speed-up compared to other methods for the implementation of profile-profile alignment on a GPU.

Bassoy et al. [5] already proposed the matrix-matrix product based solution for profile-sequence alignment on the GPU. We have improved the memory space requirement and reduced the number of matrix operations for this method. As a result, our modified algorithm can handle long sequences. The design of the wavefront technique for profile-sequence alignment is similar to profile-profile alignment due to the identical structure. The matrix-matrix product based solution shows better speed-up over other methods and attains approximately 28-fold speed-up over the serial implementation. The progressive alignment stage of the ClustalW algorithm is mapped onto GPU using a combination of the matrix-matrix product and the wavefront methods. The results exhibit speed-up of more than one order of magnitude for several data sets considered.

Next, we have devised the matrix-matrix product solution for the Viterbi algorithm which is a member of the SMDP class. The Viterbi algorithm is used to find the most probable sequence of hidden states that has generated the observation which is a compute intensive task. The matrix-matrix product based algorithm is designed by separating the data independent and dependent parts. The emission and probabilities are pre-calculated and stored in a matrix for each sequence character. Then the sub-matrices are multiplied to obtain the maximum likelihood probability. We have achieved approximately 173-fold speed-up for SMemViterbi. The performance is not effected by changing the number of states for a particular sequence length using SMemViterbi.

The all-pairs shortest path problem for the general and bipartite graphs is a member of the SPDP class. We have implemented the matrix-matrix product solution on GPU. For bipartite graphs, the GPU implementation achieved approximately 274-fold speed-up over the serial implementation of the Floyd-Warshall algorithm.

In this work, we have discussed the techniques to calculate large scale multiple alignments which performs significantly faster on a low-cost GPU than on the CPU. However, many problems are not suitable to GPU programming because they cannot be vectorized. Due to the latency issues, transferring data from CPU to GPU and back can be a bottleneck for problems that require much GPU-CPU interaction.

Future Directions

We have shown the performance of different dynamic programming algorithms on a single GPU. The performance enhancement in a multiple GPU environment should be investigated. We can also take benefit of the dynamic parallelism present in Kepler architecture to enhance the efficiency of matrix-matrix product technique.

The OpenACC is a parallel programming model which is designed to enhance the performance and portability of applications across many types of platforms. It is a directive-based programming model and requires fewer changes in code compared with CUDA and OpenCL [9, 123]. The speed-up for each of dynamic programming class should be studied using OpenACC.

The profile-hidden Markov model is used for modelling DNA and protein sequence families based on multiple sequence alignment. The matrix-matrix product based profile-hidden Markov model should be designed to utilize the huge computation power of the GPU.

The parallel formulation of all-pairs shortest paths problem for bipartite graphs results in better performance. This problem should be explored for sparse bipartite graphs using tropical matrix-matrix product.

Bibliography

[1] A. M. Aji, W. Feng, F. Blagojevic, and D. S. Nikolopoulos. Cell-SWat: modeling and scheduling wavefront computations on the cell broadband engine. In *Proceedings of the 5th Conference on Computing Frontiers*, CF'08, pages 13–22. ACM, 2008.

[2] S. Altschul, W. Gish, W. Miller, E. Myers, and D. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215:403–410, 1990.

[3] L. A. Anbarasu, P. Narayanasamy, and V. Sundararajan. Multiple sequence alignment using parallel genetic algorithms. In *Simulated Evolution and Learning*, volume 1585 of *Lecture Notes in Computer Science*, pages 130–137. Springer, 1999.

[4] D. Bashford, C. Chothia, and A. M. Lesk. Determinants of a protein fold. Unique features of the globin amino acid sequences. *Journal of Molecular Biology*, 196(1):199–216, 1987.

[5] C. S. Bassoy, S. Torgasin, M. Yang, and K.-H. Zimmermann. Accelerating scalar-product based sequence alignment using graphics processor units. *Signal Processing Systems*, 61(2):117–125, 2010.

[6] R. Bellman. *Dynamic programming.* Princeton University Press, 1st edition, 1957.

[7] A. P. Bird. CpG islands as gene markers in the vertebrate nucleus. *Trends in Genetics*, 3:342–347, 1987.

[8] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: stream computing on graphics hardware. *ACM Transactions on Graphics*, 23(3):777–786, 2004.

[9] CAPS Enterprise, Cray Inc., NVIDIA, and the Portland Group. *The OpenACC application programming interface*, 2011.

[10] K. Chaichoompu, S. Kittitornkun, and S. Tongsima. MT-ClustalW: multithreading multiple sequence alignment. In *Proceedings of the 20th International Conference on Parallel and Distributed Processing*, IPDPS'06, pages 254–254. IEEE Computer Society, 2006.

[11] T. M. Chan. More algorithms for all-pairs shortest paths in weighted graphs. In *Proceedings of the Thirty-ninth Annual ACM Symposium on Theory of Computing*, STOC'07, pages 590–598. ACM, 2007.

[12] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, and K. Skadron. A performance study of general-purpose applications on graphics processors using CUDA. *Journal of Parallel and Distributed Computing*, 68(10):1370–1380, 2008.

[13] S. Che, J. Li, J. W. Sheaffer, K. Skadron, and J. Lach. Accelerating compute-intensive applications with GPUs and FPGAs. In *Proceedings of the 2008 Symposium on Application Specific Processors*, SASP'08, pages 101–107. IEEE Computer Society, 2008.

[14] J. Cheetham, F. Dehne, S. Pitre, A. Rau-Chaplin, and P. J. Taillon. Parallel Clustal W for PC clusters. In *Proceedings of the International Conference on Computational Science and Its Applications: PartII, ICCSA 2003*, volume 2668 of *Lecture Notes in Computer Science*, pages 300–309. Springer, 2003.

[15] R. Chenna, H. Sugawara, T. Koike, R. Lopez, T. J. Gibson, D. G. Higgins, and J. D. Thompson. Multiple sequence alignment with the Clustal series of programs. *Nucleic Acids Research*, 31:3497–3500, 2003.

[16] J. Chong, K. You, Y. Yi, E. Gonina, C. Hughes, W. Sung, and K. Keutzer. Scalable HMM based inference engine in large vocabulary continuous speech recognition. In *Proceedings of the 2009 IEEE International Conference on Multimedia and Expo, ICME 2009*, pages 1797–1800. IEEE, 2009.

[17] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. *Journal of Symbolic Computation*, 9(3):251–280, 1990.

[18] M. O. Dayhoff and R. M. Schwartz. A model of evolutionary change in proteins. In *Atlas of Protein Sequence and Structure*, pages 345–352, 1978.

[19] X. Deng, E. Li, J. Shan, and W. Chen. Parallel implementation and performance characterization of MUSCLE. In *Proceedings of the 20th International Conference on Parallel and Distributed Processing, IPDPS'06*, pages 311–311. IEEE Computer Society, 2006.

[20] P. R. Dixon, T. Oonishi, and S. Furui. Fast acoustic computations using graphics processors. In *Proceedings of the 2009 IEEE International Conference on Acoustics, Speech and Signal Processing*, ICASSP'09, pages 4321–4324. IEEE Computer Society, 2009.

[21] F. F. Dragan. Estimating all pairs shortest paths in restricted graph families: a unified approach. *Journal of Algorithms*, 57(1):1–21, 2005.

[22] Z. Du, Z. Yin, and D. A. Bader. A tile-based parallel Viterbi algorithm for biological sequence alignment on GPU with CUDA. In *Proceedings of the 24th IEEE International Symposium on Parallel and Distributed Processing, IPDPS'10*, pages 1–8. IEEE, 2010.

[23] R. Durbin, S. R. Eddy, A. Krogh, and G. J. Mitchison. *Biological sequence analysis: probabilistic models of proteins and nucleic acids.* Cambridge University Press, 1998.

[24] J. Ebedes and A. Datta. Multiple sequence alignment in parallel on a workstation cluster. *Bioinformatics*, 20(7):1193–1195, 2004.

[25] S. R. Eddy. Multiple alignment using hidden Markov models. In *Proceeding of International Conference on Intelligent Systems for Molecular Biology*, pages 114–120, 1995.

[26] R. C. Edgar and K. Sjölander. COACH: profile-profile alignment of protein families using hidden Markov models. *Bioinformatics*, 20(8):1309–1318, 2004.

[27] R. C. Edgar and K. Sjölander. A comparison of scoring functions for protein sequence profile alignment. *Bioinformatics*, 20(8):1301–1308, 2004.

[28] D.-F. Feng and R. Doolittle. Progressive sequence alignment as a prerequisite to correct phylogenetic trees. *Journal of Molecular Evolution*, 25(4):351–360, 1987.

[29] G. A. Fink. *Markov models for pattern recognition: from theory to applications.* Springer, 2008.

[30] R. W. Floyd. Algorithm 97: Shortest path. *Communications of the ACM*, 5(6):345–345, 1962.

[31] G. D. Forney. The Viterbi algorithm. *Proceedings of the IEEE*, 61:268–278, 1973.

[32] N. Ganesan, R. D. Chamberlain, J. Buhler, and M. Taufer. Accelerating HMMER on GPUs by implementing hybrid data and task parallelism. In *Proceedings of the First ACM International Conference on Bioinformatics and Computational Biology*, pages 418–421, 2010.

[33] M. Goujon, H. McWilliam, W. Li, F. Valentin, S. Squizzato, J. Paern, and R. Lopez. A new bioinformatics analysis tools framework at EMBL-EBI. *Nucleic Acids Research*, 38:695–699, 2010.

[34] A. Grama, G. Karypis, V. Kumar, and A. Gupta. *Introduction to parallel computing.* Addison Wesley, 2nd edition, 2003.

[35] M. Gribskov, M. Homyak, J. Edenfield, and D. Eisenberg. Profile scanning for three-dimensional structural patterns in protein sequences. *Computer Applications in the Biosciences*, 4(1):61–66, 1988.

[36] M. Gribskov, A. D. McLachlan, and D. Eisenberg. Profile analysis: detection of distantly related proteins. *Proceedings of the National Academy of Sciences of the United States of America*, 84(13):4355–4358, 1987.

[37] S. K. Gupta, J. D. Kececioglu, and A. A. Schäffer. Improving the practical space and time efficiency of the shortest-paths approach to sum-of-pairs multiple sequence alignment. *Journal of Computational Biology*, 2(3):459–472, 1995.

[38] D. Gusfield. *Algorithms on strings, trees, and sequences: computer science and computational biology.* Cambridge University Press, 1997.

[39] M. K. Hanif and K.-H. Zimmermann. Graphics card processing: accelerating profile-profile alignment. *Central European Journal of Computer Science*, 2:367–388, 2012.

[40] P. Harish and P. J. Narayanan. Accelerating large graph algorithms on the GPU using CUDA. In *Proceedings of the 14th International Conference on High Performance Computing*, volume 4873 of *HiPC'07*, pages 197–208. Springer, 2007.

[41] S. Henikoff and J. Henikoff. Amino acid substitution matrices from protein blocks. *Proceedings of the National Academy of Sciences of the United States of America (PNAS)*, 89:10915–10919, 1992.

[42] D. G. Higgins and P. M. Sharp. CLUSTAL: a package for performing multiple sequence alignment on a microcomputer. *Gene*, 73(1):237–244, 1988.

[43] D. G. Higgins, J. D. Thompson, and T. J. Gibson. Using CLUSTAL for multiple sequence alignments. In *Computer Methods for Macromolecular Sequence Analysis*, volume 266 of *Methods in Enzymology*, pages 383–402. Academic Press, 1996.

[44] D. R. Horn, M. Houston, and P. Hanrahan. ClawHMMER: a streaming HMMer-search implementation. In *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, SC'05. IEEE Computer Society, 2005.

[45] S. Hougardy. The Floyd–Warshall algorithm on graphs with negative cycles. *Information Processing Letters*, 110(8-9):279–281, 2010.

[46] C.-L. Hung, C.-Y. Lin, Y.-C. Chung, and C. Y. Tang. A parallel algorithm for three-profile alignment method. In *International Joint Conference on Bioinformatics, Systems Biology and Intelligent Computing. IJCBS'09*, pages 153–159. IEEE Computer Society, 2009.

[47] S. R. Hymel. Massively parallel hidden Markov models for wireless applications. Master's thesis, Virginia Polytechnic Institute and State University, 2011.

[48] K. Katoh, K. Misawa, K. I. Kuma, and T. Miyata. MAFFT: a novel method for rapid multiple sequence alignment based on fast Fourier transform. *Nucleic Acids Research*, 30(14):3059–3066, 2002.

[49] G. J. Katz and J. T. Kider, Jr. All-pairs shortest-paths for large graphs on the gpu. In *Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware*, GH'08, pages 47–55, 2008.

[50] J. Kendrew, K. Davis, E. Barnard, and E. Lawrence. *Encyclopedia of molecular biology.* Blackwell Publishing Limited, 1994.

[51] D. B. Kirk and W. W. Hwu. *Programming massively parallel processors: a hands-on approach.* Morgan Kaufmann Publishers Inc., 1st edition, 2010.

[52] A. Krogh. *An introduction to hidden Markov models for biological sequences*, pages 45–63. Elsevier, 1998.

[53] A. Krogh, B. Larsson, G. von Heijne, and E. L. L. Sonnhammer. Predicting transmembrane protein topology with a hidden Markov model: application to complete genomes. *Journal of Molecular Biology*, 305:567–580, 2001.

[54] S. Y. Kung. *VLSI array processors.* Prentice-Hall, Inc., 1987.

[55] M. Larkin, G. Blackshields, N. Brown, R. Chenna, P. McGettigan, H. McWilliam, F. Valentin, I. Wallace, A. Wilm, R. Lopez, J. Thompson, T. Gibson, and D. Higgins. Clustal W and Clustal X version 2.0. *Bioinformatics*, 23(21):2947–2948, 2007.

[56] J. Li, S. Chen, and Y. Li. The fast evaluation of hidden Markov models on GPU. In *IEEE International Conference on Intelligent Computing and Intelligent Systems, ICIS'09*, volume 4, pages 426–430, 2009.

[57] K.-B. Li. ClustalW-MPI: ClustalW analysis using distributed and parallel computing. *Bioinformatics*, 19(12):1585–1586, 2003.

[58] Y. Lifshits, S. Mozes, O. Weimann, and M. Ziv-Ukelson. Speeding up HMM decoding and training by exploiting sequence repetitions. *Algorithmica*, 54(3):379–399, 2009.

[59] C. Liu. CuHMM: a CUDA implementation of hidden Markov model training and classification. Technical report, Johns Hopkins University, 2009.

[60] L. Liu, M. Wang, J. Jiang, R. Li, and G. Yang. Efficient nonserial polyadic dynamic programming on the cell processor. In *25th IEEE International Symposium on Parallel and Distributed Processing, IPDPS'11*, pages 460–471. IEEE Computer Society, 2011.

[61] W. Liu, B. Schmidt, G. Voss, and W. Müller-Wittig. GPU-ClustalW: using graphics hardware to accelerate multiple sequence alignment. In *Proceedings of the 13th International Conference on High Performance Computing, HiPC'06*, volume 4297 of *Lecture Notes in Computer Science*, pages 363–374. Springer, 2006.

[62] Y. Liu, B. Schmidt, and D. L. Maskell. MSA-CUDA: multiple sequence alignment on graphics processing units with CUDA. In *Proceedings of the 20th IEEE International Conference on Application-specific Systems, Architectures and Processors*, ASAP'09, pages 121–128. IEEE Computer Society, 2009.

[63] Y. Liu, B. Schmidt, and D. L. Maskell. Parallel reconstruction of neighbor-joining trees for large multiple sequence alignments using CUDA. In *Proceedings of the 23rd IEEE International Symposium on Parallel & Distributed Processing*, IPDPS'09, pages 1–8. IEEE Computer Society, 2009.

[64] H. Lopes and G. Moritz. A distributed approach for a multiple sequence alignment algorithm using a parallel virtual machine. *27th Annual International Conference of the Engineering in Medicine and Biology Society, IEEE-EMBS*, 3:2843–2846, 2005.

[65] A. V. Lukashin and M. Borodovsky. GeneMark.hmm: new solutions for gene finding. *Nucleic Acids Research*, 26:1107–1115, 1998.

[66] B. D. Lund and J. W. Smith. A multi-stage CUDA kernel for Floyd–Warshall. *CoRR*, abs/1001.4108, 2010.

[67] S. Manavski and G. Valle. CUDA compatible GPU cards as efficient hardware accelerators for Smith-Waterman sequence alignment. *BMC Bioinformatics*, 9(S-2), 2008.

[68] MATLAB. *version 7.10.0 (R2010a).* The MathWorks Inc., Natick, Massachusetts, 2010.

[69] D. Mikhailov, H. Cofer, and R. Gomperts. Performance optimization of Clustal W: parallel Clustal W, HT Clustal, and MULTICLUSTAL. Technical report, SGI ChemBio, 2001.

[70] S. Mozes, O. Weimann, and M. Ziv-Ukelson. Speeding up HMM decoding and training by exploiting sequence repetitions. In *Combinatorial Pattern Matching, 18th Annual Symposium, CPM 2007*, volume 4580 of *Lecture Notes in Computer Science*, pages 4–15. Springer, 2007.

[71] Y. Munekawa, F. Ino, and K. Hagihara. Design and implementation of the Smith-Waterman algorithm on the CUDA-compatible GPU. In *Proceedings of the 8th IEEE International Conference on BioInformatics and BioEngineering, BIBE'08*, pages 1–6. IEEE, 2008.

[72] R. Nath, S. Tomov, and J. Dongarra. An improved Magma Gemm for Fermi graphics processing units. *International Journal of High Performance Computing Applications*, 24(4):511–515, 2010.

[73] S. B. Needleman and C. D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48(3):443–453, 1970.

[74] J. Nielsen and A. Sand. Algorithms for a parallel implementation of hidden Markov models with a small state space. In *Proceedings of the 25th IEEE International Symposium on Parallel and Distributed Processing, IPDPS'11*, pages 452–459. IEEE Computer Society, 2011.

[75] C. Notredame. Recent progress in multiple sequence alignment: a survey. *Pharmacogenomics*, 3(1):131–144, 2002.

[76] C. Notredame, D. G. Higgins, and J. Heringa. T-coffee: a novel method for fast and accurate multiple sequence alignment. *Journal of Molecular Biology*, 302(1):205–217, 2000.

[77] NVIDIA Corporation. *NVIDIA's Next Generation CUDA Compute Architecture: Fermi*, 2009.

[78] NVIDIA Corporation. *CUDA C Programming Guide Version 4.0*, 2011.

[79] NVIDIA Corporation. *CUDA Toolkit 4.0 CUBLAS Library*, 2011.

[80] NVIDIA Corporation. *CUDA C Programming Guide Version 5.0*, 2012.

[81] NVIDIA Corporation. *CUDA Toolkit 5.0 CUBLAS Library*, 2012.

[82] NVIDIA Corporation. *NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110*, 2012.

[83] NVIDIA Corporation. *CUDA C BEST PRACTICES GUIDE Version 5.5*, 2013.

[84] T. Ohlson, B. Wallner, and A. Elofsson. Profile-profile methods provide improved fold-recognition: a study of different profile-profile alignment methods. *Proteins*, 57(1):188–197, 2004.

[85] T. Okuyama, F. Ino, and K. Hagihara. A task parallel algorithm for finding all-pairs shortest paths using the GPU. *International Journal of High Performance Computing and Networking*, 7(2):87–98, 2012.

[86] T. Oliver, B. Schmidt, D. Nathan, R. Clemens, and D. Maskell. Using reconfigurable hardware to accelerate multiple sequence alignment with ClustalW. *Bioinformatics*, 21(16):3431–3432, 2005.

[87] L. Pachter, M. Alexandersson, and S. Cawley. Applications of generalized pair hidden Markov models to alignment and gene finding problems. *Journal of Computational Biology*, 9(2):389–399, 2002.

[88] L. Pachter and B. Sturmfels. *Algebraic statistics for computational biology*. Cambridge University Press, 2005.

[89] W. R. Pearson and D. J. Lipman. Improved tools for biological sequence comparison. *Proceedings of the National Academy of Sciences of the United States of America*, 85(8):2444–2448, 1988.

[90] D. PS. Sequence alignment using graphics processor units. Master's thesis, The University of Western Australia, 2008.

[91] L. R. Rabiner. A tutorial on hidden Markov models and selected applications in speech recognition. In *Proceedings of the IEEE*, pages 257–286, 1989.

[92] L. R. Rabiner and B. H. Juang. An introduction to hidden Markov models. *IEEE Transactions on Acoustics, Speech, and Signal Processing Magazine*, 3:4–16, 1986.

[93] G. Rizk and D. Lavenier. GPU accelerated RNA folding algorithm. In *Computational Science, ICCS'09*, volume 5544 of *Lecture Notes in Computer Science*, pages 1004–1013. Springer, 2009.

[94] L. Roditty and A. Shapira. All-pairs shortest paths with a sublinear additive error. *ACM Transactions on Algorithms*, 7(4):45:1–45:12, 2011.

[95] F. Romani. Shortest-path problem is not harder than matrix multiplication. *Information Processing Letters*, 11(3):134–136, 1980.

[96] L. Rychlewski, L. Jaroszewski, W. Li, and A. Godzik. Comparison of sequence profiles. Strategies for structural predictions using sequence information. *Protein Science*, 9:232–241, 2000.

[97] N. Saitou and M. Nei. The neighbor-joining method: a new method for reconstructing phylogenetic trees. *Molecular Biology and Evolution*, 4:406–425, 1987.

[98] A. Sand, M. Kristiansen, C. N. S. Pedersen, and T. Mailund. zipHMMlib: a highly optimised HMM library exploiting repetitions in the input to speed up the forward algorithm. *BMC Bioinformatics*, 14:339, 2013.

[99] S. Scanzio, S. Cumani, R. Gemello, F. Mana, and P. Laface. Parallel implementation of artificial neural network training for speech recognition. *Pattern Recognition Letters*, 31(11):1302–1309, 2010.

[100] M. Schatz, C. Trapnell, A. Delcher, and A. Varshney. High-throughput sequence alignment using graphics processing units. *BMC Bioinformatics*, 8, 2007.

[101] R. Seidel. On the all-pairs-shortest-path problem. In *Proceedings of the Twenty-fourth Annual ACM Symposium on Theory of Computing*, STOC'92, pages 745–749. ACM, 1992.

[102] C. Shaffer. Next-generation sequencing outpaces expectations. *Nature Biotechnology*, 25(2):149–149, 2007.

[103] A. Siepel and D. Haussler. Phylogenetic hidden Markov models. In *Statistical Methods in Molecular Evolution*, pages 325–351. Springer, 2005.

[104] A. C. Siepel and D. Haussler. Combining phylogenetic and hidden Markov models in biosequence analysis. In *Proceedings of the Seventh Annual International Conference on Research in Computational Molecular Biology*, RECOMB'03, pages 277–286, 2003.

[105] I. Simon. Recognizable sets with multiplicities in the tropical semiring. In *Mathematical Foundations of Computer Science*, volume 324 of *Lecture Notes in Computer Science*, pages 107–120. Springer, 1988.

[106] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147:195–197, 1981.

[107] S. Solomon and P. Thulasiraman. Performance study of mapping irregular computations on GPUs. In *IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW)*, pages 1–8. IEEE, 2010.

[108] V. Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13:354–356, 1969.

[109] T. Takaoka. A simplified algorithm for the all pairs shortest path problem with o(n2logn) expected time. *Journal of Combinatorial Optimization*, 25(2):326–337, 2013.

[110] T. Takaoka and M. Hashim. A simpler algorithm for the all pairs shortest path problem with o(n2logn) expected time. In *Proceedings of the 4th International Conference on Combinatorial Optimization and Applications*, COCOA'10, pages 195–206. Springer-Verlag, 2010.

[111] G. Tan, S. Feng, and N. Sun. Parallel multiple sequences alignment in SMP cluster. In *Proceedings of the Eighth International Conference on High-Performance Computing in Asia-Pacific Region, HPCASIA'05*, pages 425–431. IEEE Computer Society, 2005.

[112] G. Tan, N. Sun, and G. R. Gao. Improving performance of dynamic programming via parallelism and locality on multicore architectures. *IEEE Transactions on Parallel and Distributed Systems*, 20(2):261–274, 2009.

[113] K. Terkelsen and A. Krogh. Automatic generation of gene finders for eukaryotic species. *BMC Bioinformatics*, 7(263), 2006.

[114] J. Thompson, D. Higgins, and T. Gibson. CLUSTAL W: improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-specific gap penalties and weight matrix choice. *Nucleic Acids Research*, 22:4673–4680, 1994.

[115] J. D. Thompson, F. Plewniak, and O. Poch. A comprehensive comparison of multiple sequence alignment programs. *Nucleic Acids Research*, 27(13):2682–2690, 1999.

[116] S. Torgasin and K.-H. Zimmermann. An all-pairs shortest path algorithm for bipartite graphs. *Central European Journal of Computer Science*, 3(4):149–157, 2013.

[117] A. Viterbi. Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *IEEE Transactions on Information Theory*, 13(2):260–269, 1967.

[118] N. von Öhsen, I. Sommer, and R. Zimmer. Profile-profile alignment: a powerful tool for protein structure prediction. In *Pacific Symposium on Biocomputing*, pages 252–263, 2003.

[119] B. W. Wah and G.-J. Li. Systolic processing for dynamic programming problems. *Circuits, Systems and Signal Processing*, 7(2):119–149, 1988.

[120] J. P. Walters, V. Balu, S. Kompalli, and V. Chaudhary. Evaluating the use of GPUs in liver image segmentation and HMMER database searches. In *Proceedings of the 23rd IEEE International Symposium on Parallel & Distributed Processing*, IPDPS'09, pages 1–12. IEEE Computer Society, 2009.

[121] S. Warshall. A theorem on boolean matrices. *Journal of the ACM*, 9(1):11–12, 1962.

[122] M. S. Waterman. *Introduction to computational biology - maps, sequences, and genomes: interdisciplinary statistics.* CRC Press, 1995.

[123] S. Wienke, P. Springer, C. Terboven, and D. Mey. OpenACC: first experiences with real-world applications. In *Proceedings of the 18th International Conference on Parallel Processing*, Euro-Par'12, pages 859–870. Springer-Verlag, 2012.

[124] K.-J. Won, T. Hamelryck, A. Prügel-Bennett, and A. Krogh. An evolutionary method for learning HMM structure: prediction of protein secondary structure. *BMC Bioinformatics*, 8:357, 2007.

[125] C.-C. Wu, J.-Y. Ke, H. Lin, and W. Feng. Optimizing dynamic programming on graphics processing units via adaptive thread-level parallelism. In *Proceedings of IEEE 17th International Conference on Parallel and Distributed Systems*, ICPADS'11, pages 96–103. IEEE Computer Society, 2011.

[126] S. Xiao, A. M. Aji, and W. Feng. On the robust mapping of dynamic programming onto a graphics processing unit. In *Proceedings of the 15th International Conference on Parallel and Distributed Systems*, ICPADS'09, pages 26–33. IEEE Computer Society, 2009.

[127] D. Zhang, R. Zhao, L. Han, T. Wang, and J. Qu. An Implementation of Viterbi Algorithm on GPU. In *Proceedings of the First IEEE International Conference on Information Science and Engineering*, ICISE'09, pages 121–124, 2009.

[128] K.-H. Zimmermann. *An introduction to protein informatics.* Kluwer Academic Publishers, 2003.

[129] K.-H. Zimmermann. Algebraic statistics. Manuscript, Institute of Computer Technology, Hamburg University of Technology, 2009.

List of Publications

1. M. K. Hanif and K.-H. Zimmermann. Graphics card processing: accelerating profile-profile alignment. Central European Journal of Computer Science, 2(4), pages 367-388, 2012.

Curicculum Vitæ

Last name	Hanif
First name	Muhammad Kashif
Date of birth	May 1st, 1983
Place of birth	Faisalabad, Pakistan

Studies

Sept. 2000 – Feb. 2005	Bachelor in Science (Hons) in Computer Science. University of Punjab, Lahore, Pakistan.
Sept. 2005 – Jun. 2008	Master of Science, area: Computer Science. University of Agriculture, Faisalabad, Pakistan.
Oct. 2009 – Jul. 2014	HEC-DAAD Scholarship holder towards a Ph.D. Hamburg University of Technology, Germany.

Work

Sept. 2006 – Jun. 2007	Management Trainee. Crescent Textile Mills, Faisalabad, Pakistan.
Jun. 2007 – Aug. 2009	Assistant Manager-IT. Crescent Textile Mills, Faisalabad, Pakistan.