Implementing and Accelerating HMMER3 Protein Sequence Search on CUDA-Enabled GPU

Lin Cheng

A thesis

in

The Department

of

Computer Science and Software Engineering

Presented in Partial Fulfilment of the Requirements

For the Degree of Master of Computer Science

Concordia University

Montréal, Québec, Canada

June 2014

CONCORDIA UNIVERSITY

School of Graduate Studies

This is to certify that the thesis prepared

By : **Lin Cheng**

Entitled : **Implementing and Accelerating HMMER3 Protein Sequence Search on CUDA-Enabled GPU**

and submitted in partial fulfilment of the requirements for the degree of

Master of Computer Science

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee :

_____ Chair
Dr. A.

_____ Examiner
Dr. B.

_____ Examiner
Dr. C.

_____ Supervisor
Dr. Gregory Butler

Approved by _____
Dr. D.
Chair of Department or Graduate Program Director

_____ 2014. _____
Dr. E.
Dean of Faculty
(Computer Science and Software Engineering)

CONCORDIA UNIVERSITY
Computer Science and Software Engineering

Abstract

Implementing and Accelerating HMMER3 Protein Sequence Search on CUDA-Enabled GPU

by Lin Cheng

The recent emergence of multicore CPU and manycore GPU architectures has made parallel computing more and more popular. Hundreds of industrial and research applications have been mapped onto GPUs to further utilize the extra GPUs computing resource. In bioinformatics, the application of Smith-Waterman algorithm and HMMER's hmmsearch are two important applications for finding similarity among protein sequences with dynamic programming method. Both applications are particularly well-suited for many-core architectures due to the parallel nature of sequence database searches.

After studying the existing research on CUDA acceleration in bioinformatics, this thesis investigated the acceleration of the key MSV algorithm in HMMER3. A fully-featureed CUDA-enabled protein database search tool *cudaHmmsearch* was designed, implemented and optimized. We demonstrated a variety of optimization strategies that are useful for general purpose GPU-based applications. Based on our optimization experience in parallel computing, 6 steps were summarized for better performance of CUDA programming.

We made comprehensive tests and analysis for multiple enhancements in our GPU kernels in order to demonstrate the effectiveness of each approaches. The performance analysis showed that GPUs are able to deal with intensive computations, but are very sensitive to random accesses to the global memory. The results show that our implementation achieved 2.5x speedup over the single-threaded HMMER3 CPU SSE2 implementation on average.

Acknowledgements

First, I am truely thankful to my supervisor Dr. Gregory BUTLER for his profound knowledge, flexibility in supervising students, warm-hearted, and selecting this interesting topic for my research.

I also give thanks to my friendly group: Faizah Aplop, Christine Houry Kehyayan and Nada Alhirabi for their communicating and helping me know more about Bioinformatics and life in Montreal, Stuart Thiel for providing Kronos machine as my developing and benchmarking environment.

My deepest gratitude goes to my family in China for their unquestioning love. I miss them so much in Canada. Hope less visa trouble between the two countries I love.

Contents

List of Figures

List of Tables

Abbreviations

CUDA	**C**ompute **U**nified **D**evice **A**rchitecture
DP	**D**ynamic **P**rogramming
GPU	**G**raphics **P**rocessing **U**nit
HMM	**H**idden **M**arkov **M**odel
HMMER	**H**idden **M**arkov **M**odel**ER**
MPI	**M**essage **P**assing **I**nterface
MSV	**M**ultiple **S**egment **V**iterbi
NCBI	**N**ational **C**enter for **B**iotechnology **I**nformation)
NR	**N**on-**R**edundant
OS	**O**perating **S**ystem
Pfam	**P**rotein **fam**ilies
SIMD	**S**ingle-**I**nstruction **M**ultiple-**D**ata
SM	**S**treaming **M**ultiprocessors
SSE	**S**treaming **S**IMD **E**xtensions
SW	**S**mith-**W**aterman

Chapter 1

Introduction

1.1 Problem Statement

HMMER [HMMER, 2014b] is a free and commonly used software package for sequence analysis written by Sean Eddy. It is an open source implementation of HMM algorithms for use with protein databases.

One of its more widely used applications, *hmmsearch* is to identify homologous protein. It does this with Viterbi algorithm described in subsection 2.2.2.2 by comparing a profile HMM to each protein sequence in a large database, evaluating the path that has the maximum probability of the HMM generating the sequence. This search requires a computationally intensive procedure.

There has been a great deal of work on optimizing HMMER for both CPUs and GPUs.

JackHMMer [Wun et al., 2005] uses the Intel IXP 2850 network processor to accelerate Viterbi algorithm. The processor is used as a single-chip cluster with the XScale CPU functioning as the head node. Like a typical cluster, the XScale CPU is responsible for distributing jobs to the individual microengines.

MPI-HMMER [Walters et al., 2006] is a wellknown and commonly used MPI implementation. In their studies, a single master node is used to assign multiple database blocks to worker nodes for computing in parallel. And it is responsible for collecting the results.

HMMER3 [Eddy, 2011] is the most significant acceleration of hmmsearch. The main performance gain is due to a heuristic algorithm called MSV filter, for Multiple (local,

ungapped) Segment Viterbi, as described in Section2.2.2.3. MSV is implemented in SIMD vector parallelization instructions and is about 100-fold faster than HMMER2.

GPUs have been shown to provide very attractive compute resources in addition to CPUs, because of particular manycore parallel computation in GPUs.

[Walters et al., 2009], [Ganesan et al., 2010], [Du et al., 2010] and [Quirem et al., 2011] parallelized Viterbi algorithm on CUDA-enabled GPUs.

[Ahmed et al., 2012][Ahmed et al., 2012] used Intel VTune Analyzer [Intel, 2013] to investigate performance hotspot functions in HMMER3. Based on hotspot analysis, they studied CUDA acceleration for three individual algorithm: Forward, Backward and Viterbi algorithms.

As shown in Figure3.1, the MSV and Viterbi algorithms are implemented in the so-called "acceleration pipeline" at the core of the HMMER3 software package [Eddy, 2011]. And the MSV algorithm is the first filter of "acceleration pipeline" and is the key hotspot of the whole process. Therefore, this thesis concentrate on porting the MSV onto CUDA-enabled GPU to accelerate hmmsearch application.

1.2 Research Contributions

The contribution of this thesis can be classified as follows:

- Analyze the core application *hmmsearch* in HMMER3 and find the key hotspot MSV filter for accelerating hmmsearch.
- Implement the protein sequence search tool *cudaHmmsearch* on CUDA-enabled GPU. Demonstrate many optimization approaches to accelerate cudaHmmsearch.
- Discuss and analyze the advantages and limitations of GPU hardware for CUDA parallel programming.

1.3 Organization of thesis

The rest of this thesis is organized as follows:

Chapter 2 introduces the background necessary for understanding the work in this thesis.

Then Chapter 3 presents the details of our *cudaHmmsearch* implementation and optimization approaches. And 6 steps are summarized for better performance of CUDA programming at the end of this Chapter.

Comprehensive benchmarks were performed and analyzed in Chapter 4.

The conclusion of Chapter 5 summarizes our contributions, points out its limitations, and makes suggestions for future work.

1.4 Typographical Conventions

The following font conventions are used in this thesis:

- Adobe Helvetica font

  Used for code examples.

- *Adobe Helvetica slanted font*

  Used for comments of code.

- Adobe AvantGarde font

  Used for captions of table and figure.

Chapter 2

Background

The background of this thesis is concerned with the algorithms related to our work and how they were accelerated by many studies on CUDA-enabled GPUs.

2.1 Sequence alignment and protein database

2.1.1 Cells, amino acids and proteins

In 1665, Robert Hooke discovered the cell [Wiki Cell, 2014]. The cell theory, first developed in 1839 by Matthias Jakob Schleiden and Theodor Schwann, generalized the view that *all living organisms are composed of cells and of cell products* [Loewy et al., 1991]. As workhorses of the cell, proteins not only constitute the major component in the cell, but they also regulate almost all activities that occurs in living cells.

Proteins are complex chains of small organic molecules known as *amino acids*. In this thesis, *residue* is used to refer to amino acids for protein. The 20 amino acids detailed in Table2.1 have been found within proteins and they convey a vast array of chemical versatility. So proteins can be viewed as sequences of an alphabet of the 20 amino acids {A,C,D,E,F,G,H,I,K,L,M,N,P,Q,R,S,T,V,W,Y}.

Letter	Amino acid	Letter	Amino acid
A	Alanine	C	Cysteine
D	Aspartic acid	E	Glutamic acid
F	Phenylalanine	G	Glycine
H	Histidine	I	Isoleucine
K	Lysine	L	Leucine
M	Methionine	N	Asparagine
P	Proline	Q	Glutamine
R	Arginine	S	Serine
T	Threonine	V	Valine
W	Tryptophan	Y	Tyrosine

TABLE 2.1: The 20 amino acids

2.1.2 Sequence alignment

In bioinformatics, a sequence alignment is a way of arranging the sequences of protein to identify regions of similarity [Wiki Sequence, 2014]. If two amino acid sequences are recognized as similar, there is a chance that they are *homologous*. Homologous sequences share a common functional, structural, or evolutionary relationships between them. Protein sequences evolve with accumulating mutations. The basic mutational process are *substitutions* where one residue is replaced by another, *insertions* where a new residue is inserted into the sequence, and *deletions*, the removal of a residue. Insertions and deletions are together referred to as *gaps*. Table2.2 shows the sequence alignment of human beta globin (the *query* sequence) and myoglobin (the *target* sequence) and an internal gap is indicated by two dashes [Pevsner, 2009].

Query	V T A L W	G K V N V	D - - E V	G G E A L
Target	V L N V W	G K V E A	D I P G H	G Q E V L
match	4 11	6 5 4	6	6 5 4
mismatch	-1-2 1	0 0	-2 -3	-2 0
gap open *gap extend*			-11 -1	

TABLE 2.2: **Sequence alignment**
sum of *match*: +51
sum of *mismatch*: -9
sum of gap penalties: -12
total raw score: 51 - 9 -12 = 30

To establish the degree of homology, the two sequences are aligned: lined up in such a way that the degree of similarity also referred as score is maximized. Table2.2 illustrates how raw scores are calculated. The scores in the table for *match/mismatch* are taken from the scoring matrix BLOSUM62 as shown in Figure2.1. The BLOSUM62 matrix is a substitution matrix used for sequence alignment of proteins. It were first introduced in a paper by S. Henikoff and J.G. Henikoff [Henikoff and Henikoff, 1992]. In a typical scoring scheme there are two gap penalties: one for *gap open* (-11 in the example of Table2.2) and one for *gap extend* (-1 in Table2.2).

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

FIGURE 2.1: BLOSUM62 Scoring Matrix from http://www.ncbi.nlm.nih.gov/Class/FieldGuide/BLOSUM62.txt.

Sorts of sequence alignment algorithms have been studied. Next chapter will introduce two sorts of algorithms: Smith-Waterman algorithm and HMM-based algorithms. They share a very general optimization technique called dynamic programming for finding optimal alignments.

2.1.3 Bioinformatics protein databases

This part is a brief introduction of two protein sequence databases used in this thesis.

NCBI NR databse

The NCBI (National Center for Biotechnology Information) houses a series of databases relevant to Bioinformatics. Major databases include GenBank for DNA sequences and PubMed, a bibliographic database for the biomedical literature. Other databases include the NCBI Epigenomics database. All these databases are updated daily and available online: http://www.ncbi.nlm.nih.gov/guide/all/#databases_.

The NR (Non-Redundant) protein database maintained by NCBI as a target for their BLAST search services is a composite of SwissProt, SwissProt updates, PIR(Protein Information Resource), PDB(Protein Data Bank). Entries with absolutely identical sequences have been merged into NR database. The PIR produces the largest, most comprehensive, annotated protein sequence database in the public domain. The PDB is a repository for the three-dimensional structural data of large biological molecules, such as proteins and nucleic acids and is maintained by Brookhaven National Laboratory, USA.

Release 2014_04 of NCBI NR databse contains 38,442,706 sequence entries, comprising 13,679,143,700 amino acids, more than 24GB in file size [NCBI, 2014].

Swiss-Prot

The Universal Protein Resource (UniProt) is a comprehensive resource for protein sequence and annotation data and is mainly supported by the National Institutes of Health (NIH) [UniProt, 2014]. The UniProt databases are the UniProt Knowledgebase (UniProtKB), the UniProt Reference Clusters (UniRef), and the UniProt Archive (UniParc).

The UniProt Knowledgebase is updated every four weeks on average and consists of two sections:

- UniProtKB/Swiss-Prot

  This section contains manually-annotated records with information extracted from literature and curator-evaluated computational analysis. It is also highly cross-referenced to other databases. Release 2014_05 of 14-May-2014 of UniProtKB/Swiss-Prot contains 545,388 sequence entries, comprising 193,948,795 amino acids abstracted from 228,536 references [UniProtSP, 2014-05].

- UniProtKB/TrEMBL

  This section contains computationally analyzed records that await full manual annotation. Release 2014_05 of 14-May-2014 of UniProtKB/TrEMBL contains 56,010,222 sequence entries, comprising 17,785,675,050 amino acids [UniProtTr, 2014-05].

2.2 Dynamic programming in Bioinformatics

Dynamic Programming (DP) is an optimization technique that recursively breaks down a problem into smaller subproblems, such that the solution to the larger problem can be obtained by piecing together the solutions to the subproblems [Baldi and Brunak, 2001]. This section shows how the Smith-Waterman algorithms and the algorithms in HMMER use DP for sequence alignment and database searches, and then discusses the related work about acceleration on CUDA-enabled GPU.

2.2.1 The Smith-Waterman algorithm

The Smith-Waterman algorithm is designed to find the optimal local alignment between two sequences. It was proposed by Smith and Waterman [Smith and Waterman, 1981] and enhanced by Gotoh [Gotoh, 1982]. The alignment of two sequences is based on dynamic programming approach by computing the similarity score which is given in the form of similarity score matrix H.

Given a query sequence Q with length L_q and a target sequence T with length L_t, let S be the substitution matrix and its element $S[i, j]$ be the similarity score for the

combination of the i^{th} residue in Q and the j^{th} residue in T. Define G_e as the gap extension penalty, and G_o as the gap opening penalty. These similarity scores and G_e, G_o are pre-determined by the life sciences community. The similarity score matrix H for aligning Q and T is calculated as

$$E[i,j] = max \begin{cases} E[i,j-1] - G_e \\ H[i,j-1] - G_o \end{cases}$$

$$F[i,j] = max \begin{cases} F[i-1,j] - G_e \\ H[i-1,j] - G_o \end{cases}$$

$$H[i,j] = max \begin{cases} 0 \\ E[i,j] \\ F[i,j] \\ H[i-1,j-1] + S[i,j] \end{cases}$$

where $1 \leqslant i \leqslant L_q$ and $1 \leqslant j \leqslant L_t$. The values for E, F and H are initialized as $E[i,0] = F[0,j] = H[i,0] = H[0,j]$ when $0 \leqslant i \leqslant L_q$ and $0 \leqslant j \leqslant L_t$.

The maximum value of the matrix H gives the similarity score between Q and T.

2.2.2 HMMER

HMMER [HMMER, 2014b] is a set of applications that create a profile Hidden Markov Model (HMM) of a sequence family which can be utilized as a query against a sequence database to identify (and/or align) additional homologs of the sequence family[Markel and Leon, 2003]. HMMER was developed by Sean Eddy at Washington University and has become one of the most widely used software tools for sequence homology. The main elements of this HMM-based sequence alignment package are *hmmsearch* and *hmmscan*. The former searches a profile HMM against a sequence database, while the latter searches a sequence against a profile HMMs database.

2.2.2.1 HMM and profile HMM

A hidden Markov model (HMM) is a computational structure for linearly analyzing sequences with a probabilistic method [Hancock and Zvelebil, 2004]. HMMs have been widely used in speech signal, handwriting and gesture detection problems. In bioinformatics they have been used for applications such as sequence alignment, prediction of protein structure, analysis of chromosomal copy number changes, and gene-finding algorithm, etc [Pevsner, 2009].

A HMM is a type of a non-deterministic finite state machine with transiting to another state and emitting a symbol under a probabilistic model. According to [Dong and Pei, 2007], a HMM can be defined as a 6-tuple (A, Q, q_0, q_e, tr, e) where

- A is a finite set (the alphabet) of symbols;
- Q is a finite set of *states*;
- q_0 is the *start* state and q_e is the *end* state;
- tr is the *transition* mapping, which is the transition probabilities of state pairs in $Q \times Q$, satisfying the following two conditions:
  (a) $0 \leqslant tr(q, q') \leqslant 1$, $\forall q, q' \in Q$, and
  (b) for any given state q, such that:
  $$\sum_{q' \in Q} tr(q, q') = 1$$
- e is the *emission* mapping, which is the emission probabilities of pairs in $Q \times A$, satisfying the following two conditions:
  (a) $0 \leqslant e(q, x) \leqslant 1$, if it is defined, $\forall q \in Q$, and $x \in A$
  (b) for any given state q, if for any $x \in A$, $e(q, x)$ is defined, then q is an *emitting* state and
  $$\sum_{x \in A} e(q, x) = 1$$
  if $\forall x \in A$, $e(q, x)$ is not defined, then q is a *silent* state.

The dynamics of the system is based on Markov Chain, meaning that only the current state influences the selection of its successor – the system has no 'memory' of its history. Only the succession of characters emitted is visible; the state sequence that generated the characters remains internal to the system, i.e. hidden. By this means, the name is Hidden Markov Model[Lesk, 2008].

Profile HMM is a variant of HMM and can be constructed from an initial multiple sequence alignment to define a set of probabilities. The symbol sequence of an HMM is an observed sequence that resembles a consensus for the multiple sequence alignment. And a protein family can be defined by profile HMMs.

In Figure2.2, the internal structure of the "Plan 7" profile HMM used by HMMER[Eddy, 2011] shows the mechanism for generating sequences. In order to generate sequences, a profile HMM should have a set of three states per alignment column: one *match* state, one *insert* state and one *delete* state.

- ***match state*** matches and emits a amino acid from the query. The probability of emitting each of the 20 amino acids is a property of the model.
- ***insert state*** allows the insert of one or more amino acids. The emission probability of this state is computed either from a background distribution of amino acids or from the observed insertions in the alignment.
- ***delete state*** skips the alignment column and emits a blank. Entering this state corresponds to gap opening, and the probabilities of these transitions reflect a position-specific gap penalty.

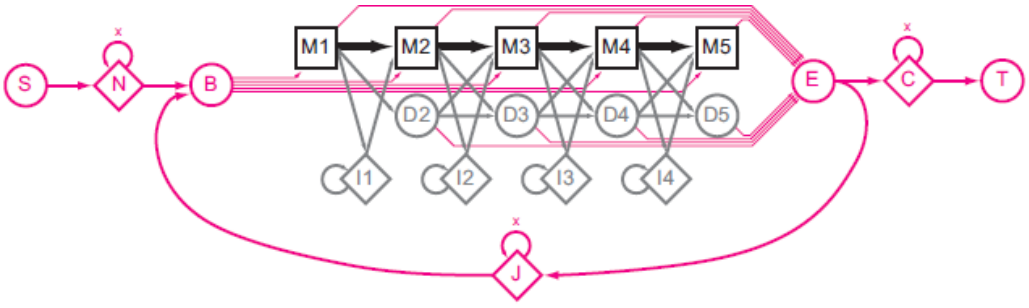

FIGURE 2.2: Profile HMM architecture used by HMMER(Eddy, 2011).

The structure begins at Start(S), and follows some chain of arrows until arriving at Termination(T). Each arrow transits to a state of the system. At each state, an action

can be taken either as (1) emitting a residue, or (2) selecting an arrow to the next state. The action and the selection of successor state are governed by sets of probabilities[Lesk, 2008]. The linear core model has five sets of match (M), insert (I) and delete (D) states. Each M state represents one consensus position and a set of M, I, D states is the main element of the model and is referred to as a "node" in HMMER. Additional flanking states (marked as N, C, and J) emit zero or more residues from the background distribution, modelling nonhomologous regions preceding, following, or joining homologous regions aligned to the core model. Start (S), begin (B), end (E) and termination (T) states are non-emitting states.

A profile HMM for a protein family can be used to compare with target sequences, and classify sequences that are members of the family and those which are not[Eidhammer et al., 2004]. A common application of profile HMMs is used to search a profile HMM against a sequence database. Another application is the query of a single protein sequence of interest against a database of profile HMMs.

2.2.2.2 Viterbi algorithm in HMMER2

In HMMER2, both *hmmsearch* and *hmmpfam* rely on the same core Viterbi algorithm for their scoring function which is named as *P7Viterbi* in codes.

To find whether a sequence is member of the family described by a HMM, we compare the sequence with the HMM. We use an algorithm known as Viterbi to find one path that has the maximum probability of the HMM generating the sequence. Viterbi is a dynamic programming algorithm. Let $V_{i,j}$ be the maximum probability of a path from the start state S_i ending at state S_j and generating the prefix $q_{1\ldots j}$ of the target sequence. $V_{i+1,j}$ is found by the recurrence:

$$V_{i+1,j} = \max_{0 \leqslant k \leqslant j-1} \left(V_{i,k} P(k,j) P(q_{i+1}|j) \right)$$

Define $a[i,j]$ as the transition probability from state i to j and e_i as emission probability in state i. Define $V_j^M(i)$ as the log-odds score of the optimal path matching subsequence $x_{1\ldots i}$ to the submodel up to state j, ending with x_i being emitted by *match* state M_j. Similarly $V_j^I(i)$ is the score of the optimal path ending in x_i being emitted by *insert*

state I_j, and $V_j^D(i)$ for the optimal path ending in *delete* state D_j. q_{x_i} is the probability of x_i. Then we can write the Viterbi general equation[Durbin et al., 1998]:

$$V_j^M(i) = \log \frac{e_{M_j}(x_i)}{q_{x_i}} + max \begin{cases} V_{j-1}^M(i-1) + \log a[M_{j-1}, M_j] \\ V_{j-1}^I(i-1) + \log a[I_{j-1}, M_j] \\ V_{j-1}^D(i-1) + \log a[D_{j-1}, M_j] \end{cases}$$

$$V_j^I(i) = \log \frac{e_{I_j}(x_i)}{q_{x_i}} + max \begin{cases} V_j^M(i-1) + \log a[M_j, I_j] \\ V_j^I(i-1) + \log a[I_j, I_j] \end{cases}$$

$$V_j^D(i) = max \begin{cases} V_{j-1}^M(i) + \log a[M_{j-1}, D_j] \\ V_{j-1}^D(i) + \log a[D_{j-1}, D_j] \end{cases}$$

Based on the above equations, we can write the efficient pseudo code of Viterbi algorithm, as shown in Algorithm2.2.1 [M.Isa et al., 2012]. The inner loop of the code contains three two dimensional matrices (M, I, D), which calculate scores of all node positions involved in the main models for each of the residue. The outer loop consists of flanking and

special states calculated in the one dimensional arrays N, B, C, J, E.

Algorithm 2.2.1: VITERBI()

comment: Initialization

$N[0] \leftarrow 0; \quad B[0] \leftarrow tr(N, B)$

$E[0] \leftarrow C[0] \leftarrow J[0] \leftarrow -\infty$

comment: for every sequence residue i

for $i \leftarrow 1$ **to** L_t

$$\textbf{do} \begin{cases} N[i] \leftarrow N[i-1] + tr(N, N) \\ B[i] \leftarrow max \begin{cases} N[i-1] + tr(N, B) \\ J[i-1] + tr(J, B) \end{cases} \\ M[i, 0] \leftarrow I[i, 0] \leftarrow D[i, 0] \leftarrow -\infty \\ \textbf{comment:} \text{ For every model position j from 1 to } L_q \\ \textbf{for } j \leftarrow 1 \textbf{ to } L_q \\ \quad \textbf{do} \begin{cases} M[0, j] \leftarrow I[0, j] \leftarrow D[0, j] \leftarrow -\infty \\ M[i, j] \leftarrow e(M_j, S[i]) + max \begin{cases} M[i-1, j-1] + tr(M_{j-1}, M_j) \\ I[i-1, j-1] + tr(I_{j-1}, M_j) \\ D[i-1, j-1] + tr(D_{j-1}, M_j) \\ B[i-1] + tr(B, M_j) \end{cases} \\ I[i, j] \leftarrow e(I_j, S[i]) + max \begin{cases} M[i-1, j] + tr(M_j, I_j) \\ I[i-1, j] + tr(I_j, I_j) \end{cases} \\ D[i, j] \leftarrow max \begin{cases} M[i, j-1] + tr(M_{j-1}, D_j) \\ D[i, j-1] + tr(D_{j-1}, D_j) \end{cases} \end{cases} \\ E[i] \leftarrow max\{M[i, j] + tr(M_j, E)\} \quad (j \leftarrow 0 \textbf{ to } L_q) \\ J[i] \leftarrow max \begin{cases} J[i-1] + tr(J, J) \\ E[i-1] + tr(E, J) \end{cases} \\ C[i] \leftarrow max \begin{cases} C[i-1] + tr(C, C) \\ E[i-1] + tr(E, C) \end{cases} \end{cases}$$

comment: Termination:

return $(T(S, M) \leftarrow C[L_t] + tr(C, T))$

From Algorithm2.2.1, we can see the fundamental task of the Viterbi algorithm for Biological Sequence Alignment is to calculate three DP(Dynamic Programming) matrices: $M[]$ for Match state, $I[]$ for Insert state and $D[]$ for Delete state. Each DP matrix consisting of $(L_t + 1) * (L_q + 1)$ blocks, where each value of block is dependent on the value of previous block. As shown in Figure2.3, the Match state $M[i, j]$ depends on the upper-left block $M[i-1, j-1]$, $I[i-1, j-1]$ and $D[i-1, j-1]$; the Insert state $I[i, j]$ depends on the left block $M[i-1, j]$ and $I[i-1, j]$; and the Delete state depends on the upper block $M[i, j-1]$ and $D[i, j-1]$.

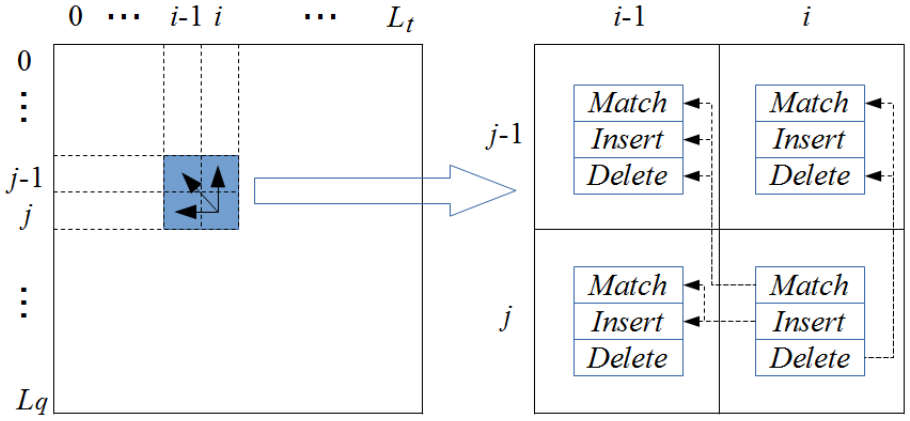

FIGURE 2.3: **The DP matrix calculated in Viterbi algorithm.** The rectangle on the left represents the whole matrix to be calculated by the Viterbi algorithm, and the right rectangle of the figure shows the process of updating a single block of the matrix.

2.2.2.3 MSV algorithm in HMMER3

HMMER3 is near rewrite of the earlier HMMER2 package, with the aim of improving the speed of profile HMM searches. The main performance gain is due to a heuristic algorithm called MSV filter, for Multiple (local, ungapped) Segment Viterbi. MSV is implemented in SIMD(Single-Instruction Multiple-Data) vector parallelization instructions and is about 100-fold faster than HMMER2.

Figure2.4 illustrates the MSV profile architecture. Compared with Figure2.2, the MSV corresponds to the virtual removal of the delete and insert states. All match-match transition probabilities are treated as 1.0. The rest parameters remains unchanged. So this model generates sequences containing one or more ungapped local alignment segments. The pseudo code of MSV score algorithm is simplified and shown in Algorithm2.2.2.

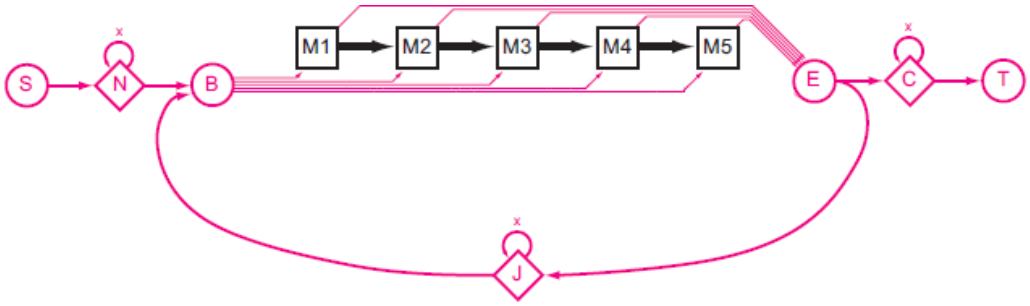

FIGURE 2.4: MSV profile: multiple ungapped local alignment segments (Eddy, 2011).

Figure2.5 illustrates an example of an alignment of a MSV profile HMM model (length $L_q = 14$) to a target sequence (length $L_t = 22$). A path to generate the target sequence with the profile HMM model is shown through a dynamic programming (DP) matrix. The model identifies two high-scoring ungapped alignment segments, as shown in black dots, indicating residues aligned to profile match states. All other residues are assigned to N, J, and C states in the model, as shown in orange dots. Unfilled dot indicates a "mute" non-emitting state or state transition.

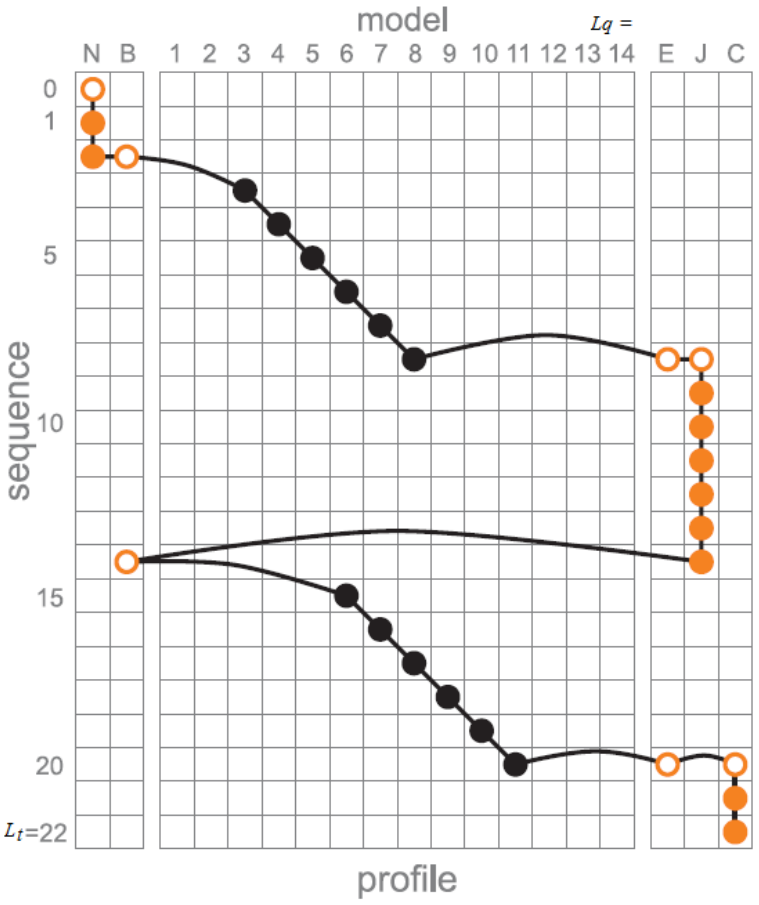

FIGURE 2.5: **Example of an MSV path in DP matrix (Eddy, 2011).** An alignment of a MSV profile HMM model (length $L_q = 14$) to a target sequence (length $L_t = 22$). A path from top to bottom is through a dynamic programming (DP) matrix. The model identifies two high-scoring ungapped alignment segments, as shown in black dots, indicating residues aligned to profile match states. All other residues are assigned to N, J, and C states in the model, as shown in orange dots. Unfilled dot indicates a "mute" non-emitting state or state transition.

Algorithm 2.2.2: MSV()

comment: Initialization

$N[0] \leftarrow 0; \quad B[0] \leftarrow tr(N, B)$

$E[0] \leftarrow C[0] \leftarrow J[0] \leftarrow -\infty$

comment: for every sequence residue i

for $i \leftarrow 1$ **to** L_t

do
$$\begin{cases} N[i] \leftarrow N[i-1] + tr(N,N) \\ B[i] \leftarrow max \begin{cases} N[i-1] + tr(N,B) \\ J[i-1] + tr(J,B) \end{cases} \\ M[i,0] \leftarrow -\infty \\ \textbf{comment:} \text{ For every model position j from 1 to } L_q \\ \textbf{for } j \leftarrow 1 \textbf{ to } L_q \\ \quad \textbf{do} \begin{cases} M[0,j] \leftarrow -\infty \\ M[i,j] \leftarrow e(M_j, S[i]) + max \begin{cases} M[i-1,j-1] \\ B[i-1] + tr(B, M_j) \end{cases} \end{cases} \\ E[i] \leftarrow max\{M[i,j] + tr(M_j, E)\} \quad (j \leftarrow 0 \textbf{ to } L_q) \\ J[i] \leftarrow max \begin{cases} J[i-1] + tr(J,J) \\ E[i-1] + tr(E,J) \end{cases} \\ C[i] \leftarrow max \begin{cases} C[i-1] + tr(C,C) \\ E[i-1] + tr(E,C) \end{cases} \end{cases}$$

comment: Termination:

return $(T(S, M) \leftarrow C[L_t] + tr(C, T))$

2.2.2.4 SIMD vectorized MSV in HMMER3

Single-Instruction Multiple-Data (SIMD) instruction is able to perform the same operation on multiple pieces of data in parallel. The first widely-deployed desktop SIMD was with Intel's MMX extensions to the x86 architecture in 1996. In 1999, Intel introduced Streaming SIMD Extensions (SSE) in Pentium III series processors. The modern SIMD vector instruction sets use 128-bit vector registers to compute up to 16 simultaneous operations. Due to the huge number of iterations in the Smith-Waterman algorithm

calculation, using SIMD instructions to reduce the number of instructions needed to perform one cell calculation has a significant impact on the execution time. Several SIMD vector parallelization methods have been described for accelerating SW dynamic programming.

In 2000, Rognes and Seeberg presented an implementation of the SW algorithm running on the Intel Pentium processor using the MMX SIMD instructions [Rognes and Seeberg, 2000]. They used a query profile parallel to the query sequence for each possible residue. A query profile was pre-calculated in a sequential layout just once before searching database. A six-fold speedup was reported over an optimized non-SIMD implementation.

In 2007, Farrar presented an efficient vector-parallel approach called striped layout for vectorizing SW algorithm [Farrar, 2007]. He designed a striped query profile for SIMD vector computation. He used Intel SSE2 to implement his design. A speedup of 2-8 times was reported over the Rognes and Seeberg SIMD non-stripped implementations.

Inspired by Farrar, in HMMER3[Eddy, 2011], Sean R. Eddy used a remarkably efficient stripped vector-parallel approach to calculate MSV alignment scores. To maximize parallelism, he implemented MSV as a 16-fold parallel calculation with score values stored as 8-bit byte integers. He used SSE2 instructions on Intel-compatible systems and Altivec/VMX instructions on PowerPC systems.

Figure2.6 shows the stripped pattern. The query profile HMM of length L_q is divided into vectors with equal length L_v. The vector length L_v is equal to the number of elements being processed in the 128-bit SIMD register. MSV processes 8-bit byte integer with $L_v = 128/8 = 16$. In a row-vectorized implementation, the query profile HMM is stored in the vectorized dynamic programming matrix dp. The dp stores L_q cells in L_Q vectors which is numbered as $q = 1...L_Q$, where $L_Q = (L_q + L_v - 1)/L_v$. Figure2.6 illustrates L_q cells assigned to L_Q vectors in a non-sequential way. For simple illustration, $L_v = 4, L_q = 14$, and $L_Q = 4$.

In Smith-Waterman and Viterbi dynamic programming, the calculation of each cell (i, j) in the dp is dependent on previously calculated cells $(i-1, j)$, $(i, j-1)$ and $(i-1, j-1)$. However, in MSV algorithm, the *delete* and *insert* states have been removed and only ungapped diagonals need calculating, so the calculation of each cell (i, j) requires only

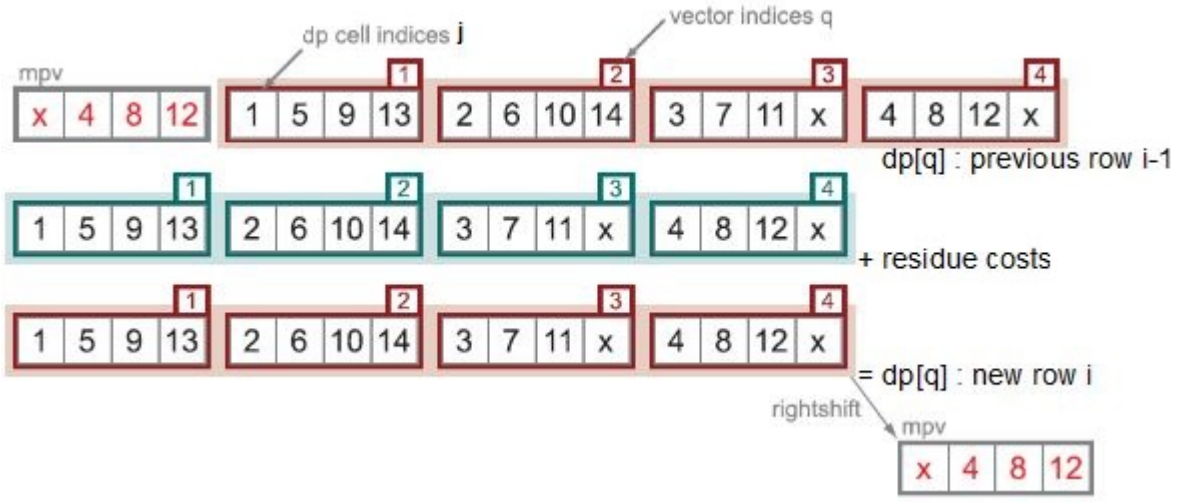

FIGURE 2.6: Illustration of striped indexing for SIMD vector calculations(Eddy, 2011).

previous $(i-1, j-1)$. In Figure2.6, the top red row shows the previous row $i-1$ for the cells $j-1$, which is needed for calculating each new cell j in a new row i.

Striping method can remove the SIMD register data dependencies. As can be seen in the Figure2.6, with striped indexing, vector $q-1$ contains exactly the four $j-1$ cells needed to calculate the four cells j in a new vector q on a new blue row of the dp matrix. For example, when we calculate cells $j = (2, 6, 10, 14)$ in vector $q = 2$, we access the previous row's vector $q - 1 = 1$ which contains the cells we need in the order we need them, $j - 1 = (1, 5, 9, 13)$ (the vector above).

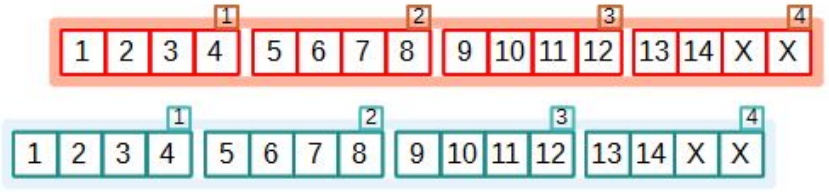

FIGURE 2.7: Illustration of linear indexing for SIMD vector calculations.

Instead, if we indexed cells into vectors in the linear order ($j = 1, 2, 3, 4$ in vector $q = 1$ and so on), as shown in Figure2.7, there is no such correspondence of $(q, q-1)$ with four $(j-1, j)$, and each calculation of a new vector q would require extra expensive operations, such as shifting or rearranging cell values inside the previous row's vectors. By using the striped query access, only one shift operation is needed per row as shown in 2.6. Outside the inner loop(for $q = 1$ to L_Q), the last vector on each finished row is

right-shifted (mpv, in grey with red cell j indices) and used to initialize the next row calculation.

The pseudo code for the implementation is shown in Algorithm2.2.3

Algorithm 2.2.3: MSV-SIMD()

comment: Initialization

$xJ \leftarrow 0$; $dp[q] \leftarrow vec\_splat(0)$ $(q \leftarrow 0$ **to** $L_Q - 1)$

$xB \leftarrow base + tr(N, B)$

$xBv \leftarrow vec\_adds(xB, tr(B, M))$

comment: for every sequence residue i

for $i \leftarrow 1$ **to** L_t

do
- $xEv \leftarrow vec\_splat(0)$
- $mpv \leftarrow vec\_rightshift(dp[L_Q - 1])$
- **for** $q \leftarrow 0$ **to** $L_Q - 1$
  - **do**
    - **comment:** temporary storage of 1 current row value in progress
    - $tmpv \leftarrow vec\_max(mpv, xBv)$
    - $tmpv \leftarrow vec\_adds(tmpv, e(M_j, S[i]))$
    - $xEv \leftarrow vec\_max(xEv, tmpv)$
    - $mpv \leftarrow dp[q]$
    - $dp[q] \leftarrow tmpv$
- $xE \leftarrow vec\_hmax(xEv)$
- $xJ \leftarrow max \begin{cases} xJ \\ xE + tr(E, J) \end{cases}$
- $xB \leftarrow max \begin{cases} base \\ xJ + tr(J, B) \end{cases}$

comment: Termination:

return $(T(S, M) \leftarrow xJ + tr(C, T))$

Five pseudocode vector instructions for operations on 8-bit integers are used in the pseudo code. Either scalars x or vectors v containing 16 8-bit integer elements numbered $v[0]...v[15]$. Each of these operations are either available or easily constructed in Intel SSE2 intrinsics as shown in the following table.

Pseudocode SSE2 intrinsic in C	**Operation**	**Definition**
v = vec_splat(x) v = _mm_set1_epi8(x)	assignment	$v[z] = x$
v = vec_adds(v1, v2) v = _mm_adds_epu8(v1, v2)	saturated addition	$v[z] = min \begin{cases} 2^8 - 1 \\ v1[z] + v2[z] \end{cases}$
v1 = vec_rightshift(v2) v1 = _mm_slli_si128(v2, 1)	right shift	$v1[z] = v2[z-1](z = 15...1);$ $v1[0] = 0;$
v = vec_max(v1, v2) v = _mm_max_epu8(v1, v2)	max	$v[z] = max(v1[z], v2[z])$
x = vec_hmax(v) -	horizontal max	$x = max(v[z]), z = 0...15$

TABLE 2.3: **SSE2 intrinsics for pseudocode in Algorithm2.2.3** The first column is pseudocode and its corresponding SSE2 intrinsic in C language. Because x86 and x86-64 use little endian, **vec_rightshift()** means using a left bit shift intrinsic **_mm_slli_si128()** to do right shift. No SSE2 intrinsic is corresponding to **vec_hmax()**. Shuffle intrinsic **_mm_shuffle_epi32** and **_mm_max_epu8** can be combined to implement **vec_hmax()**.

2.3 CUDA accelerated sequence alignment

In November 2006, NVIDIA®introduced CUDA™(Compute Unified Device Architecture), a general purpose parallel computing platform and programming model that enables users to write scalable multi-threaded programs in NVIDIA GPUs. Nowadays there exist alternatives to CUDA, such as OpenCL [OpenCL, 2014], Microsoft Compute Shader[Microsoft, 2013-11]. These are mostly similar, but as CUDA is the most widely used and more mature, this thesis will focus on that.

This section firstly overviews CUDA programming model, then reviews recent studies on accelerating Smith-waterman algorithm and HMM-based algorithms on CUDA-enabled GPU.

2.3.1 Overview of CUDA programming model

2.3.1.1 Streaming Multiprocessors

A GPU consists of one or more SMs(Streaming Multiprocessors). Quadro K4000 used in our research has 4 SMs. Each SM contains the following specific features [Wilt, 2013]:

- Execution units to perform integer and single- or double-precision floating-point arithmetic, Special function units (SFUs) to compute single-precision floating-point transcendental functions
- Thousands of registers to be partitioned among threads
- Shared memory for fast data interchange between threads
- Several caches, including constant cache, texture cache and L1 cache
- A warp scheduler to coordinate instruction dispatch to the execution units

The SM has been evolving rapidly since the introduction of the first CUDA-enabled GPU device in 2006, with three major Compute Capability 1.x, 2.x, and 3.x, corresponding to Tesla-class, Fermi-class, and Kepler-class hardware respectively. Table2.4 summarizes the features introduced in each generation of the SM hardware [Wilt, 2013].

Compute Capability	**Features introduced**
SM 1.x	Global memory atomics; mapped pinned memory; debuggable; atomic operations on shared memory; Double precision
SM 2.x	64-bit addressing; L1 and L2 cache; concurrent kernel execution; global atomic add for single-precision floating-point values; Function calls and indirect calls in kernels
SM 3.x	SIMD Video Instructions; Increase maximum grid size; warp shuffle; Bindless textures ("texture objects"); read global memory via texture; faster global atomics; 64-bit atomic min, max, AND, OR, and XOR; dynamic parallelism

TABLE 2.4: Features per Compute Capability

2.3.1.2 CUDA thread hierarchy

The execution of a typical CUDA program is illustrated in Figure2.8 The CPU host invokes a GPU kernel in-line with the triple angle-bracket <<< >>> syntax from CUDA C/C++ extension code. The kernel is executed N times in parallel by N different CUDA threads. All the threads that are generated by a kernel during an invocation are collectively called a *grid*. Figure2.8 shows the execution of two grids of threads.

Threads in a grid are organized into a two-level hierarchy, as illustrated in Figure2.9. At the top level, each grid consists of one or more thread blocks. All blocks in a grid have

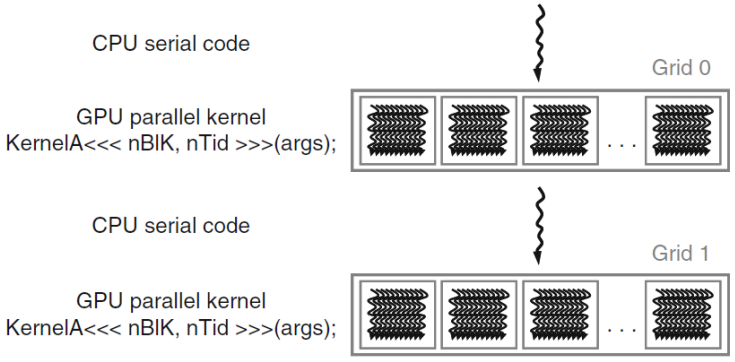

FIGURE 2.8: Execution of a CUDA program(Kirk and Hwu, 2010).

the same number of threads and are organized into a one, two, or three-dimensional *grid* of thread blocks.

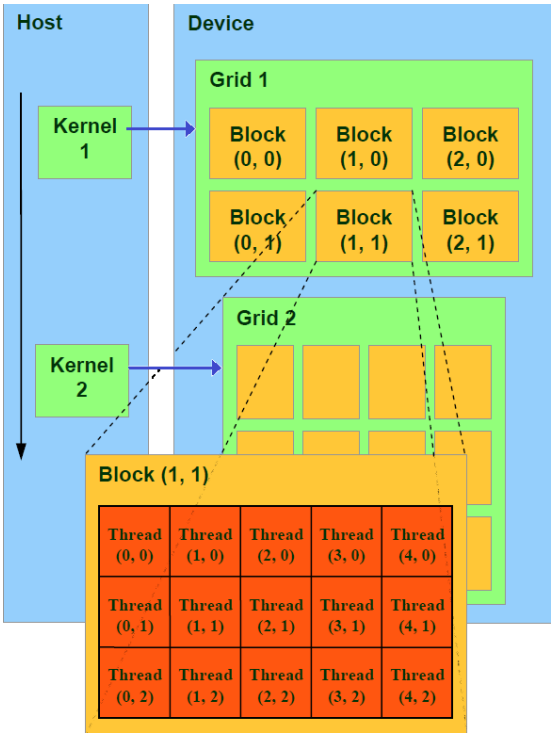

FIGURE 2.9: CUDA thread organization(Zeller, 2008).

Each block can be identified by an index accessible within the kernel through the built-in *blockIdx* variable. The dimension of the thread block is accessible within the kernel through the built-in *blockDim* variable.

The threads in a block are executed by the same multiprocessor within a GPU. They can cooperate by sharing data through some shared memory and by synchronizing their execution to coordinate memory accesses. Each block can be scheduled on any of the available multiprocessors, in any order, concurrently or sequentially, so that a compiled

CUDA program can execute on any number of multiprocessors. On the hardware level, a block's threads are executed in parallel as *warps* which name originate from *weaving loom*. A warp consists of 32 threads.

2.3.1.3 CUDA memory hierarchy

Besides the threading model, another thing that makes CUDA programming different from a general purpose CPU is its memory spaces, including registers, local, shared, global, constant and texture, as shown in Figure2.10

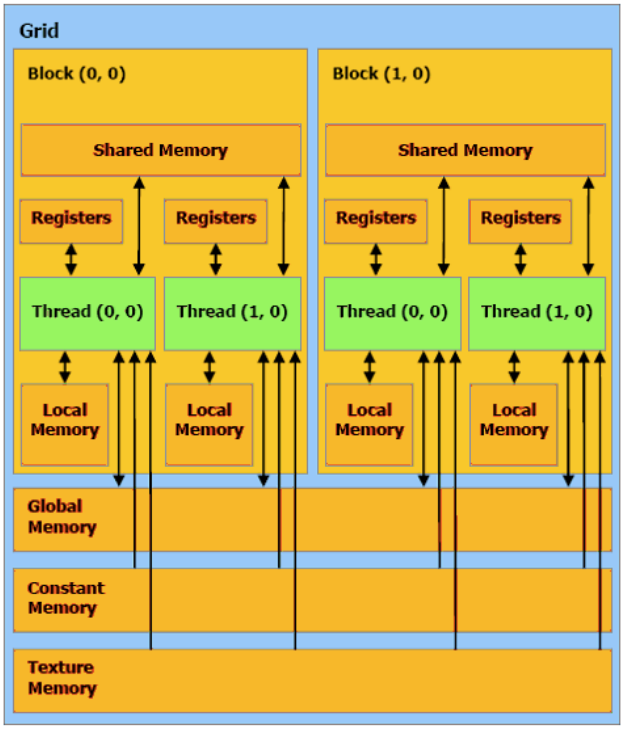

FIGURE 2.10: CUDA memory organization(Zeller, 2008).

CUDA memory spaces have different characteristics that reflect their distinct usages in CUDA applications as summarized in Table2.5 [NVIDIA, 2013-07a].

Memory	Location	Cached	Access	Scope	Speed	Lifetime
Register	On chip	n/a	R/W	1 Thread	1	Thread
Local	Off chip	†	R/W	1 Thread	$\sim 2-16$	Thread
Shared	On chip	n/a	R/W	All threads in block	$\sim 2-16$	Block
Global	Off chip	†	R/W	All threads + host	200+	Host allocation
Constant	Off chip	Yes	R	All threads + host	$2-200$	Host allocation
Texture	Off chip	Yes	R	All threads + host	$2-200$	Host allocation

TABLE 2.5: **Salient Features of GPU Device Memory. Speed** column is the relative speed in number of instructions. † means it is cached only on devices of above compute capability 2.x.

2.3.1.4 CUDA tools

The NVIDIA CUDA Toolkit provides a comprehensive development environment for C/C++ developers building GPU-accelerated applications. The CUDA Toolkit is available at https://developer.nvidia.com/cuda-toolkit, including a compiler *nvcc* for NVIDIA GPUs, math libraries, and tools for debugging and optimizing the performance of CUDA applications.

Nsight Eclipse Edition

NVIDIA Nsight Eclipse Edition is a full-featured IDE powered by the Eclipse platform that provides an all-in-one integrated programming environment for editing, building, debugging and profiling CUDA C/C++ applications. Nsight Eclipse Edition supports a rich set of commercial and free plugins. Nsight Eclipse Edition ships as part of the CUDA Toolkit Installer for Linux and Mac at https://developer.nvidia.com/nsight-eclipse-edition.

NVIDIA Nsight Visual Studio Edition

NVIDIA provides Nsight Visual Studio Edition to integrate seamlessly into Microsoft Visual Studio environment at
https://developer.nvidia.com/nvidia-nsight-visual-studio-edition. It can build, debug, profile and trace heterogeneous compute and graphics applications using CUDA C/C++, OpenCL [OpenCL, 2014], DirectCompute [Microsoft, 2013-11], Direct3D, and OpenGL.

Profiling tools

NVIDIA® provides profiling tools to help execute the kernels in question under the watchful gaze, which are publicly available as a separate download on the CUDA Zone website [NVIDIA, 2014b].

Tools Name	OS	User interface
nvprof	Linux, Mac OS X and Windows	command-line
Visual Profiler	Linux, Mac OS X and Windows	graphical
Nsight™ Ecipse Edition	Linux and Mac OSX	graphical
NVIDIA® Nsight™ Visual Studio Edition	Windows	graphical
Parallel Nsight™	Windows	graphical

TABLE 2.6: CUDA profiling tools

Nvidia-SMI

The NVIDIA System Management Interface (nvidia-smi) is a command line utility that helps managing and monitoring GPU devices. It ships with NVIDIA GPU display drivers on Linux and Windows. This utility allows administrators to query GPU device state and modify GPU device state. It can also report control aspects of GPU execution, such as whether ECC is enabled and how many CUDA contexts can be created on a given GPU.

2.3.2 CUDA accelerated Smith-Waterman

The Smith-Waterman algorithm for sequence alignment uses dynamic programming method for sequence alignment, which is also the characteristic of HMM-based algorithms. In this section, we review the techniques used in parallelizing Smith-Waterman on a CUDA-enabled GPU and these techniques will be evaluated for accelerating MSV algorithm in Chapter 3.

Parallelism strategy applied

1) Task-based parallelism

As explained in Section3.1, parallel computing has two types of parallelism: task-based and data-based parallelism. SW algorithm is used for finding similarity among protein sequence database with dynamic programming method. The application is particularly well-suited for many-core architectures due to the parallel nature of sequence database searches.

Liu et al. have been studying accelerating Smith-Waterman sequence database searches for CUDA-enabled GPU since 2009. Here are their 3 articles from 2009 to 2013: [Liu et al., 2009], [Liu et al., 2010] and [Liu et al., 2013]. They present many approaches for optimization. They use task-based parallelism to process each target sequence independently with a single GPU thread. Task-based parallelism removes the need for inter-thread communications or, even worse, inter-multiprocessor communications. This also simplifies implementation and testing.

Due to these reasons, task-based parallelism has been taken by most of studies and more efforts are put on optimizing CUDA kernel execution. Among 8 articles reviewed here, 7 articles apply this approach. Beside 3 articles of Liu et al., other 4 article are [Manavski, 2008], [Akoglu and Striemer, 2009], [Ligowski and Rudnicki, 2009] and [Kentie, 2010].

In [Liu et al., 2013], Liu et al. not only distribute tasks to many threads in GPU kernel, but also balance the workload between CPU and GPU. Their distribution policy calculates a rate R of the number of residues from the database assigned to GPUs, with a formular as

$$R = \frac{N_G f_G}{N_G f_G + N_C f_C / C}$$

where f_C and f_G are the core frequencies of CPUs and GPUs, N_C and N_G are the number of CPU cores and the number of GPU SMs, and C is a constant derived from empirical evaluations.

They find the sequence length deviation generally causes execution imbalance between threads, which in return can not fully utilize GPU compute power. Considering this, they design two CUDA kernels based on two parallelization approaches: static scheduling and dynamic scheduling. These two kernels are launched based on the sequence length deviation of the database.

Since each thread has its own buffers in global memory, the static scheduling launches all thread blocks onto the GPU at the same time. When a thread block finishes its current computation, this thread block will use the dynamic scheduling to obtain an unprocessed profile block. They use the atomic addition function *atomicAdd*() to increment the index of global profile blocks.

[Manavski, 2008] also distributes some tasks to CPU. In GPU, they only launch 450 thread-blocks at a time on a grid. For a database with sequences more than 450, several kernel launches are necessary and this will create additional latency. And since they run 64 threads per block. So, they can only process 450 * 64 = 28,800 alignments per kernel launch. CUDA limits the number of blocks in a grid to 65,535 blocks in a single dimension, which is far above the threshold they are using. This results in low GPU kernel thread occupancy.

Considering Manavski's implementation limitation in thread occupancy, [Akoglu and Striemer, 2009] sets thread block size by the total number of sequences located in the database so that they can to do all alignments with a single kernel launch.

In order to achieve high efficiency for task-based parallelism, the run time of all threads in a thread block should be roughly identical. Therefore many studies often sorted sequences databases by length of sequences. Thus, for two adjacent threads in a thread warp, the difference value between the lengths of the associated sequences is minimized, thereby balancing a similar workload over threads in a warp.

[Manavski, 2008], [Akoglu and Striemer, 2009], [Liu et al., 2009], [Liu et al., 2010] and [Liu et al., 2013] presorted database in ascending order.

[Ligowski and Rudnicki, 2009] presorts database in descending order and organizes data in blocks consisting of 256 sequences, since each kernel has 16 blocks, and a single block consists of 256 threads. The choice of 4096 threads per kernel is dictated partially by limitations of architecture and partially by optimization of performance. There are 16 multiprocessors in each core, each executes a single block of threads. The number of 256 threads per block is limited by the number of registers (8192 per multiprocessor). The SW main routine needs 29 registers, and therefore the highest multiply of 64 that can be concurrently executed is 256. They process the alignment matrix horizontally. The main loop for calculating the matrix is executed for a band of 12 cells columns. Slow

global memory is accessed only at the initialization and termination of the loop. Fast shared memory and registers are used for all operations within the loop. The width of the band is limited by availability of shared memory.

[Kentie, 2010] implements a *dbconv* tool to presort database also in descending order and converted into a special format. The *dbconv* writes the sorted sequences to file in an interlaced fashion: the database is split up into half-warp sized blocks of 16 sequences. The sequences of a block are written in an alternating fashion: one 8-bit symbol of the first sequence is written, then one of the second one, etc. In this way, the first character of all 16 sequences is written, then the second character, etc. All sequences in a rectangular block are padded to the length of the block's longest sequence with blank symbols. By this way, all of a half-warp's threads will load database symbols from neighboring addresses so as to access the global memory in coalesced way.

2) Data-based parallelism

In data-based parallelism, each task is assigned to one or many thread block(s) and all threads in the thread block(s) cooperate to perform the task in parallel.

The main target of [Saeed et al., 2010] is to solve a single but very large Smith-Waterman problem for sequences with very long lengths. Their calculation work along anti-diagonals of the alignment matrix. Each diagonal item can be calculated independently of the others. Block diagonal algorithms are also possible as long as each block is processed serially.

They formulate parallel version of the Smith-Waterman algorithm so that the calculations can be performed in parallel one row (or column) of similarity matrix at a time. Row (or column) calculations allow the GPU global memory accesses to be consecutive and therefore high memory throughput is achieved.

They exploit approach of parallelizing multiple GPUs with using MPI (Message Passing Interface) parallel technique over 100 Mb Ethernet to extend work to multiple GPUs. To use N GPUs, the Smith-Waterman alignment matrix is decomposed into N large, and slightly overlapping blocks.

[Liu et al., 2009] investigate the two parallelism approaches for parallelizing the sequence database searches using CUDA. They find task-based parallelism can achieve better performance although it needs more device memory than data-based. For task-based

parallelism, they sort target sequences and store in an array row by row from the top-left corner to the bottom-right corner, where all symbols of a sequence are restricted to be stored in the same row from left to right. Using these arrangement patterns for the two parallelism strategies, access to the target sequences is coalesced for all threads in a half-warp.

To maximize performance and to reduce the bandwidth demand of global memory, they also apply a cell block division method for the task-based parallelism, where the alignment matrix is divided into cell blocks of equal size.

They apply data-based parallelism to support longest query/target sequences. Each task is assigned to one thread block and all threads in the thread block cooperate to perform the task in parallel, exploiting the parallel characteristics of cells in the minor diagonals of similarity matrix.

Device memory access pattern

As described in Section2.3.1.3, CUDA memory hierarchy includes registers, local, shared, global, constant and texture, as shown in Figure2.10. Memory throughput generally dominates program performance both in the CPU and GPU domains. Here is the reviewing of how these studies applied different device memory access pattern to optimize their implementations.

[Liu et al., 2009] sorts target sequences and arranges in an array like a multi-layer bookcase to store into global memory, so that the reading of the database across multiple threads could be coalesced. Writes to global memory are first batched in shared memory for better coalescing. Due to a reduction in the global memory accesses, they propose a cell block division method for the task-based parallelism, where the alignment matrix is divided into cell blocks of equal size. And they utilize the texture memory on the sorted array of target sequences in order to achieve maximum performance on coalesced access patterns. They use a hash table to index the location coordinate in the array and the length of each sequence, which can provide fast access to any sequence.

They also exploit constant memory to store the gap penalties, scoring matrix and the query sequence. They load the scoring matrix into shared memory, as the performance of constant memory degrades linearly when threads frequently need to access multiple

different addresses in the scoring matrix. They also use the CUDA built-int integer functions max(x, y) and min(x, y) to improve performance further.

[Kentie, 2010] reduces global memory access by making temporary values interleaved and reads/writes score and Ix temporary values in one access. And Kentie exploits constant memory and shared memory for substitution matrix. Constant memory is fast, but only if all threads read the same address. This is not suited to the substitution matrix. And shared memory also has several disadvantages for the matrix. Finally, he finds texture memory has the ability to fetch four values at a time and is well suitable for random access. He gains a total speedup of 25% after switching from shared to texture memory. He also stores gap penalties in constant memory.

[Akoglu and Striemer, 2009] stores database sequences in global memory. The cell calculation blocks are used to store temporary calculation values needed for data dependencies for each column and are also stored in global memory. They use the substitution matrix instead of the query profile to save memory size. In order to index the row and column of the matrix in extremely efficient way, they design a simple function for accessing the matrix, which is as follows:

$$S_{i,j} = (ascii(S_1) - 65, ascii(S_2) - 65)$$

S_1 is a residue from the query sequence and S_2 is a residue from one of the Database sequences. They map query sequence as well as the substitution matrix to the constant memory to make access to these values easy. They track the highest SW score by a single register in the kernel for each thread, and update on each pass of the inner loop. Then the score is written to the global memory, after the alignment is finished.

Among the 7 articles reviewed here, [Manavski, 2008] is the first to study accelerating SW on CUDA-enabled GPU in 2007. They present the query profile as a query-specific substitution matrix computed only once for the entire database. They exploit the cache of texture memory to store query profiles. In this way they replace random accesses to the substitution matrix with sequential ones to the query profile.

[Liu et al., 2010] utilizes texture memory to store query profiles. They use shared memory to store the 4 residues of target sequence.

[Liu et al., 2013] stores both the query profile and its variant in texture memory. They gain more performance from the query profile variant for short queries, because it can

reduce the number of texture fetches by half. However, for longer queries, a query profile becomes superior due to its much smaller memory footprint and less texture cache miss. They apply a query length threshold Q to decide whether to use the query profile or the variant.

[Ligowski and Rudnicki, 2009] reduces global memory access only at the loop initialization and for writing the results at the exit. They performed all operations within the loop in fast shared memory and registers.

Vector programming model

Vector programming model plays important role in operations of array or matrix. On one side, it can reduce greatly the frequency of memory access; on the other side, it can utilize the built-in SIMD vector instructions for parallel computing both on CPU and GPU.

[Manavski, 2008] packs the query profile in texture memory, storing 4 successive values into the 4-byte of a single unsigned integer. To compute a column of alignment matrix, all the H and E values are needed from the previous one. They place them in the two local memory buffer of the thread: one for the previous values and another for the newly computed ones. At the end of each column they swap them. Since local memory is not cached, they use specific access pattern to read 4 H and 4 E values from local memory at a time, which can fully take advantage of the 128 bits memory bandwidth. Thus, each thread can gather all the data needed to compute 4 cells of the alignment matrix with only two read instructions: one from the local buffer and another from the texture.

Manavski pre-computes a query profile parallel to the query sequence for each possible residue and achieves dynamic load balancing between multiple GPUs according to their computational power at run time.

[Akoglu and Striemer, 2009] calculates the Smith-Waterman score from the query sequence and database sequences by means of columns, 4 cells at a time for the residue of database sequence aligned with an 8 residue query sequence. The updated values of cells are placed in a temporary location in the global memory. This cell calculation block is updated each time a new column is computed, and is utilized for dependency purposes in computing columns on each pass.

[Liu et al., 2010] designs a striped query profile for SIMD vector computation and uses a packed data format to store into the CUDA built-in *uchar4* vector data type, instead of the *char* scalar data type. In this way, 4 substitution scores can be accessed with only one texture fetch, thus greatly improving texture memory throughput.

Like the query profile, they also construct each target sequence with a packed data format, where 4 successive residues of each target sequence are packed together and placed in *uchar4*. In this case, they utilize shared memory to store the 4 residues loaded by one texture fetch for the use of the inner loop, when using the cell block division method.

Since CUDA lacks support for saturation addition and subtraction, they use maximum and minimum operation to artificially implement them. The CUDA built-in integer functions max(x, y) and min(x, y) are utilized to avoid divergence. They implement shift operation on vector using shared memory, where all threads comprising a virtualized vector write their original values to a share memory buffer and then read their resulting values from the buffer as per the number of shift elements.

They divide a query sequence into a series of non-overlapping, consecutive small partitions with a specified partition length, and then align the query sequence to a subject sequence partition by partition. They port the SIMD CPU algorithm [Farrar, 2007] to the GPU, viewing collections of processing elements as part of a single vector.

[Kentie, 2010] applies vector data structure to load 4 query characters at a time and processes 8 database characters at a time. He loads query profile values for the 4 current query characters and passes them to the Smith-Waterman function. The 4 query symbols and loaded query profile values are aligned with each loaded database symbol.

And he simplifies substitution matrix lookup by using numeric values instead of letters for sequence symbols.

[Liu et al., 2013] designs a query profile variant data structure and packs the 4 consecutive elements of variant data into the built-in *short4* vector type. In this way, although the variant data need more memory space compared to the query profile, they can reduce the number of texture fetches for the variant by half. On the other side, using the variant can save 6 bitwise operations for generating a substitution score vector. They also utilize

the built-in *uint4* vector data type to store each sequence profile for quad-lane SIMD computing on GPUs.

They use CUDA SIMD Video Instructions in GPU computing and use Intel SSE2 intrinsic in CPU computing. For CPU SIMD computation, their approach is based on the open-source SWIPE [Rognes, 2011]. They compute the SW algorithm by splitting an SSE vector to 16 lanes with 8-bit lane width. Then they re-compute all alignments, whose scores have overflow potential, using 8-lane SSE vectors with 16-bit lane width.

2.3.3 CUDA accelerated HMMER

HMMER includes a MPI (Message Passing Interface) implementation of the searching algorithms, which uses conventional CPU clusters for parallel computing. ClawHMMer [Horn et al., 2005] is the first GPU-enabled *hmmsearch* implementation. Their implementation is based on the BrookGPU stream programming language, not CUDA programming model. Since ClawHMMer, there has been several researches on accelerating HMMER for CUDA-enabled GPU. The following is the summary of techniques applied by 5 research work.

Parallelism strategy applied

As explained in Section3.1, parallel computing has two types of parallelism: task-based and data-based parallelism. SW algorithm is used for finding similarity among protein sequence database with dynamic programming method. The application is particularly well-suited for many-core architectures due to the parallel nature of sequence database searches. Among 5 articles reviewed here, 3 articles, i.e. [Walters et al., 2009], [Quirem et al., 2011] and [Ahmed et al., 2012] used task-based parallelism to process each target sequence independently with a single GPU thread. Task-based parallelism removes the need for inter-thread communications or, even worse, inter-multiprocessor communications. This also simplifies implementation and testing. This approach was taken by most of studies and more efforts were put on optimizing CUDA kernel execution.

In [Walters et al., 2009], Walters et al. port Viterbi function to CUDA-enabled GPU with a variety of optimization approaches. Their implementation operate the GPU kernel on multiple sequences simultaneously, with each thread operating on an independent

sequence. They found the number of threads that can be executed in parallel will be limited by two factors: one is GPU memory will limit the number of sequences that can be placed, and another is the number of registers used by each thread will limit the number of threads that can execute in parallel. And registers are the most important resource in their implementation.

They split the inner loop for computing the dynamic programming matrix into three independent small loops. This approach makes fewer registers be required, resulting in higher GPU utilization. Further, splitting the loop leads an easy mechanism to exploit loop unrolling, which is a classic loop optimization strategy designed to reduce the overhead of inefficient looping. The idea is to replicate the loops inner contents such that the percentage of useful instruction to loop extra bound instruction increases. In their experiment, the performance improvement reaches 80%.

In order to achieve high efficiency for task-based parallelism, the run time of all threads in a thread block should be roughly identical. Therefore many studies often sorted sequences databases by length of sequences. Thus, for two adjacent threads in a thread warp, the difference value between the lengths of the associated sequences is minimized, thereby balancing a similar workload over threads in a warp. Walters et al. presort target sequences database in ascending order. This approach is both effective and quite straightforward way of optimization and they gain a nearly 7x performance improvement over the unsorted database without changing the CUDA kernel in any way.

Walters et al. distribute some tasks to CPU, creating two CPU threads for reading database and post-processing the database hits.

For data-based parallelism, based on the *wave-front* method [Aji et al., 2008], Du et al. apply a new tile-based mechanism to accelerate the Viterbi algorithm on a single GPU in [Du et al., 2010]. The *wave-front* method compute the cells along anti-diagonal of the dynamic matrix in parallel, which is similar to a frontier of a wave to fill a matrix, where each cell's value in the matrix is computed based on the values of the left, upper, and upper-left cells.

They apply streaming method to process very long sequences. In CUDA, a stream is a set of instructions that execute in order and different streams may execute their

instructions asynchronously. This feature enables the execution be overlapped with each other between the host and GPU device.

Since the streaming Viterbi algorithm requires additional storage and communication bandwidth, they design the new tile-based method to simplify the computational model and also handles very long sequences with less memory transfer. The tile-based method handles very long sequence as follows: the large matrix is divided into tiles to ensure that each tile fits in the GPUs memory as a whole and then compute the tiles in parallel.

Unlike the tiling machanism in [Aji et al., 2008], which has data dependencies among each tile, they present the *homological segments* concept into their tile-based mechanism to eliminate the data dependency among different tiles. They apply the k-mer based algorithm to find all homological segments. Then the homological segments are used to divide the full dynamic programming matrix into small tiles.

[Ganesan et al., 2010] parallelizes Viterbi algorithm to accelerate hmmsearch. They present a hybrid parallelization strategy by combining task-based and data-based parallelism. They extend the existing task parallelism upon which data parallelism is built. They reassign the computation of a single sequence across multiple threads to implement the data parallelism. In order to accelerate the computation of the dynamic programming matrix rows, they partition each row into equal sized intervals of contiguous cells and calculate the dependencies between the partitions identically and independently in a data parallel setting.

They process the parallel computation of the rows in three phases as follows:

- In Phase 1, they use independent threads to compute the relationship between the beginning and end of each partition, which enables the fast computation of boundary elements between the partitions.
- In Phase 2, by applying the functions relating the boundary elements from Phase 1, they compute the numeric values for each of the boundary elements. This phase executes consecutively, with N partitions requiring N steps.
- In Phase 3, by using the updated numerical values for each of the boundary elements, each partition independently computes the numeric values for all of the elements within the partition in the data parallelism setting.

Phase 1 is the critical phase that construct data parallelism by building the relationship among the different partitions. Phases 2 and 3 are computation phases for the boundary and internal elements of partitions.

They implement the partitioning scheme with storing index data of model positions at regular intervals consecutively to achieve coalesced global memory access. They benchmark the implementation on Tesla C1060, showing a speed-up of 5x-8x compared to [Walters et al., 2009] using unsorted database.

[Ahmed et al., 2012] uses Intel VTune Analyzer [Intel, 2013] to investigate performance hotspot functions in HMMER3. Based on hotspot analysis, they study CUDA acceleration for three individual algorithm: Forward, Backward and Viterbi algorithms. And they found data transfer overhead between heterogeneous processors could be a performance bottleneck.

Based on their experiments, they show that Forward, Backward and Viterbi function take 37%, 31% and 20% respectively for the percentage of the total CPU clock. They tested their CUDA implementation of these three functions. Their results show that about 2.27x, 1.58x and 1.50x speedup over the original CPU-only implementation for Forward, Backward and Viterbi function respectively. And they analyze the reason is coming from hotspot analysis. Since Forward function has highest clock time among the three function, it is the most dominant module and CUDA implementation of Forward shows large impact on speedup. Backward is the second one and Viterbi shows least speedup according to hotspot analysis based partial acceleration. They also combine the three modules and show about 2.10x speedup over the original CPU-only implementation.

However, they did not present how they exactly implement their CUDA programs for the three functions and how they accelerate their programs in [Ahmed et al., 2012]

Device memory access pattern

As described in Section2.3.1.3, CUDA memory hierarchy includes registers, local, shared, global, constant and texture, as shown in Figure2.10. Memory throughput generally dominates program performance both in the CPU and GPU domains. Here is the

reviewing of how these studies applied different device memory access pattern to optimize their implementations.

[Walters et al., 2009] found the most effective optimization for Viterbi algorithm is to optimize CUDA memory layout and usage patterns within the implementation.

Since Viterbi algorithm requires only the current and previous rows of the dynamic programming matrices, they reduce the memory requirements of the Viterbi scoring calculation from $3 * M * L + 5 * L$ to $6 * M$ integer array elements, where M and L are the length of the sequence and HMM, respectively.

They utilize high speed texture memory to store the target sequence batch. Because the seqence data are static and read-only through computing. They note that global memory coalesced access can significantly improve hmmsearch overall speedup. Their benchmark results show that this approach contributes an improvement of more than 9x for larger HMMs.

They use constant memory and texture memory to store the query profile HMM depending on its size. They place the normal size HMM in constant memory. In cases where the very large HMM exceeds the capacity of constant memory, they switch over to texture memory for the remaining portion of the HMM after exploiting the full constant memory. They utilize shared memory to temporarily store the index into each threads digitized sequence.

[Du et al., 2010] reorganizes the computational kernel of the Viterbi algorithm, and divides the basic computing unit into two parts: independent and dependent parts. All of the independent parts are executed with a parallel and balanced load in an optimized coalesced global memory access manner, which significantly improves the Viterbi algorithms performance on GPU.

They implement the *wave-front* pattern using the data skewing strategy. This makes cells in the same parallelizing group be adjacent to each other. In this way, because data accessed by neighbor threads are organized adjacent to each other, threads could access memory in a more efficient manner.

[Quirem et al., 2011] implements Viterbi algorithm being performed on Tesla C1060.

They utilize pinned memory to reduce the latency induced by transferring memory from device to host and back. And they test two different versions of the memory allocation mechanism: one is pageable memory allocation with the standard *malloc* function, another is pinned memory with the *cudaHostAlloc* function. Pinned memory is host memory that can not be paged (swapped) out to disk by the virtual memory management of the OS, and thus reduces the latency induced by transferring memory from device to host and back. Based on their experiments, utilizing pinned memory, the execution of the kernels spends more GPU time relative to the less memory transfer time. Kernel execution time is basically the same for both pageable and pinned memory, because the two versions only relate to the speed of the memory transfer. In their experiments, the performance of pinned memory is improved roughly 20% over that of pinned memory.

Their implementation gains 10x-15x speedup over the original implementation of the Viterbi function according to the number of queries. They tested up to 16,384 queries because of the limitations of the Tesla C1060 and the speedup increased exponentially as the number of launched threads i.e. number of queries doubled.

Chapter 3

A CUDA accelerated HMMER3 protein sequence search tool

3.1 Requirements and design decisions

The following are the requirements for a CUDA accelerated HMMER3 protein sequence search tool:

- A HMMER3 protein sequence search tool, named cudaHmmsearch, will be implemented to run on CUDA-enabled GPU and several optimization will be taken to accelerate the computation. The cudaHmmsearch will be tested and compared with other CPU and GPU implementations.
- The cudaHmmsearch will be based on HMMER3 algorithm so that the result will be same as hmmsearch of HMMER3.
- The cudaHmmsearch will be completely usable under various GPU devices and sequence database size. This means that cudaHmmsearch is not just for research purpose or just a proof of concept.

Implementation toolkit and language

NVIDIA CUDA [NVIDIA, 2014b] was chosen as the toolkit to be used in the implementation phase. Since its introduction in 2006, CUDA has been widely deployed through

thousands of applications and published research papers, and supported by an installed base of over 500 million CUDA-enabled GPUs in notebooks, workstations, compute clusters and supercomputers [NVIDIA, 2014c]. As of writing, CUDA is the most mature and popular GPU programming toolkit.

HMMER3 is implemented in C programming language. CUDA provides a comprehensive development environment for C and C++ developers. However, some advanced features of CUDA, such as texture, are only supported in C++ template programming. So C++ has to be used, and some compatible problems when compiling and programming between C and C++ have also to be dealt with accordingly.

Implementation methods

The following two approaches have been explored for parallelizing the protein sequence database search using CUDA. A target sequence in the database is processed as one task.

- **Task-based parallelism** Each task is assigned to exactly one thread, and *blockDim* tasks are performed in parallel by different threads in a thread block.
- **Data-based parallelism** Each task is assigned to one or many thread block(s) and all threads in the thread block(s) cooperate to perform the task in parallel.

Task-based parallelism has some advantages over data-based parallelism. On one side, it removes the need for inter-thread communications or, even worse, inter-multiprocessor communications. As described before, the thread or processing elements of one CUDA multiprocessor can communicate by using shared memory, while slow global memory must be used to transfer data between multiprocessors. At the same time, data-based also needs to take time on synchronizing and cooperating among threads. On the other side, to task-based, performing one sequence on each thread results in a kernel where each processing element is doing the exact same thing independently. This also simplifies implementation and testing. Although task-based parallelism needs more device memory than data-based, it can achieve better performance [Liu et al., 2009]. Thus, the approach of task-based parallelism is taken and more efforts are put on optimizing CUDA kernel execution.

Since data-based parallelism occupies significantly less device memory, [Liu et al., 2009] uses it to support longest query/subject sequences. However, different strategy here is applied to work around this problem and is discussed in details in subsection3.3.8 on workload distribution.

3.2 A straightforward implementation

This section describes a straightforward, mostly un-optimized implementation of the protein database search tool. First, a simple serial CPU implementation of hmmsearch is presented, with no GPU specific traits. Next, the MSV filter is ported to the GPU. This implementation is then optimized in the next section.

3.2.1 CPU serial version of hmmsearch

The CPU serial version of hmmsearch in HMMER3 is shown in Figure3.1. The MSV and Viterbi algorithms described in subsection 2.2.2.2 and 2.2.2.3 are implemented in the so-called "acceleration pipeline" at the core of the HMMER3 software package [Eddy, 2011]. One call to the acceleration pipeline is executed for the comparison of each query model and target sequence.

After each filter step, the pipeline either accepts or rejects the entire comparison, based on the P-value of the score calculated in each filter. For example, as can be seen in Figure3.1, by default a target sequence can pass the MSV filter if its comparison gets a P-value of less than 0.02. By this way, in practice about 2% of the top-scoring target sequences are expected to pass the filter. So, much fewer target sequence can pass one filter and need further computing. In consequence, the comparison is accelerated. Thus, the first MSV filter is typically the run time bottleneck for hmmsearch. Therefore, the key to parallelizing hmmsearch tool is to offload the MSV filter function to multiple computing elements on GPU, while ensuring that the code shown in Figure2.2.3 is as efficient as possible.

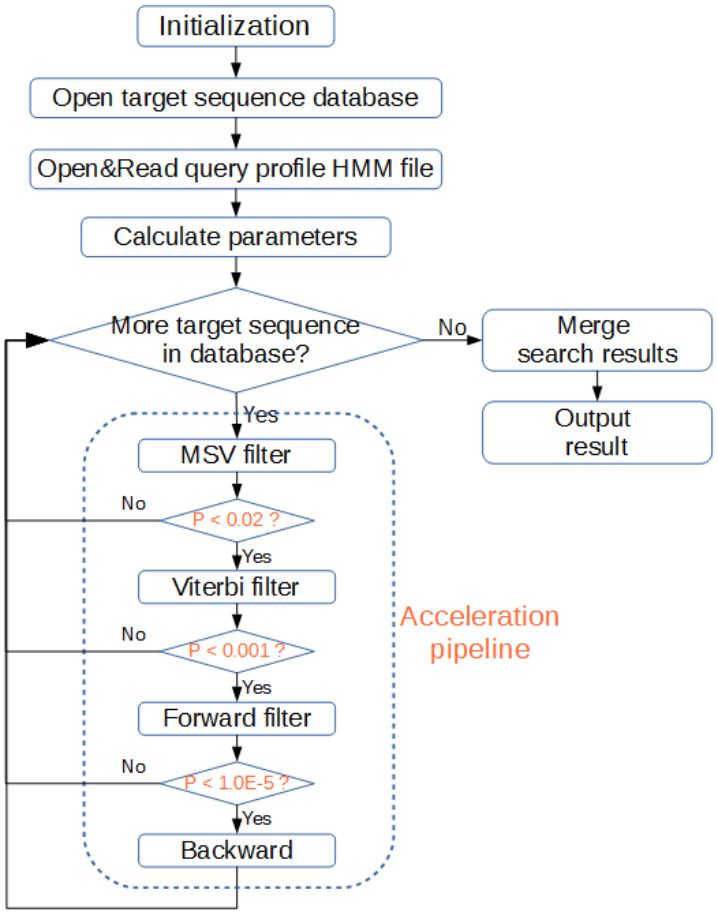

FIGURE 3.1: The CPU serial version of hmmsearch

3.2.2 GPU implementation of MSV filter

A basic flow of the GPU implementation for MSV filter is shown in Figure3.2. As can be seen, the code must be split up into two parts, with the left *host* part running on CPU and the right *device* part running on GPU. Some redundancy as data needed by GPU to compute will be copied around the memories in host and device.

The CPU code mainly concerns about allocating data structures on GPU, loading data, copying them to GPU, launching the GPU kernel and copying back the result for further steps.

The GPU kernel code is corresponding to the MSV filter Algorithm 2.2.3 and some more explanation are worth for it. First, the thread's current database sequence is set to the thread id. By this way each thread begins processing a different neighbouring sequence. This thread id is a unique numeric identifier for each thread and the id numbers of threads in a warp are consecutive. Next, the location where each thread can store and

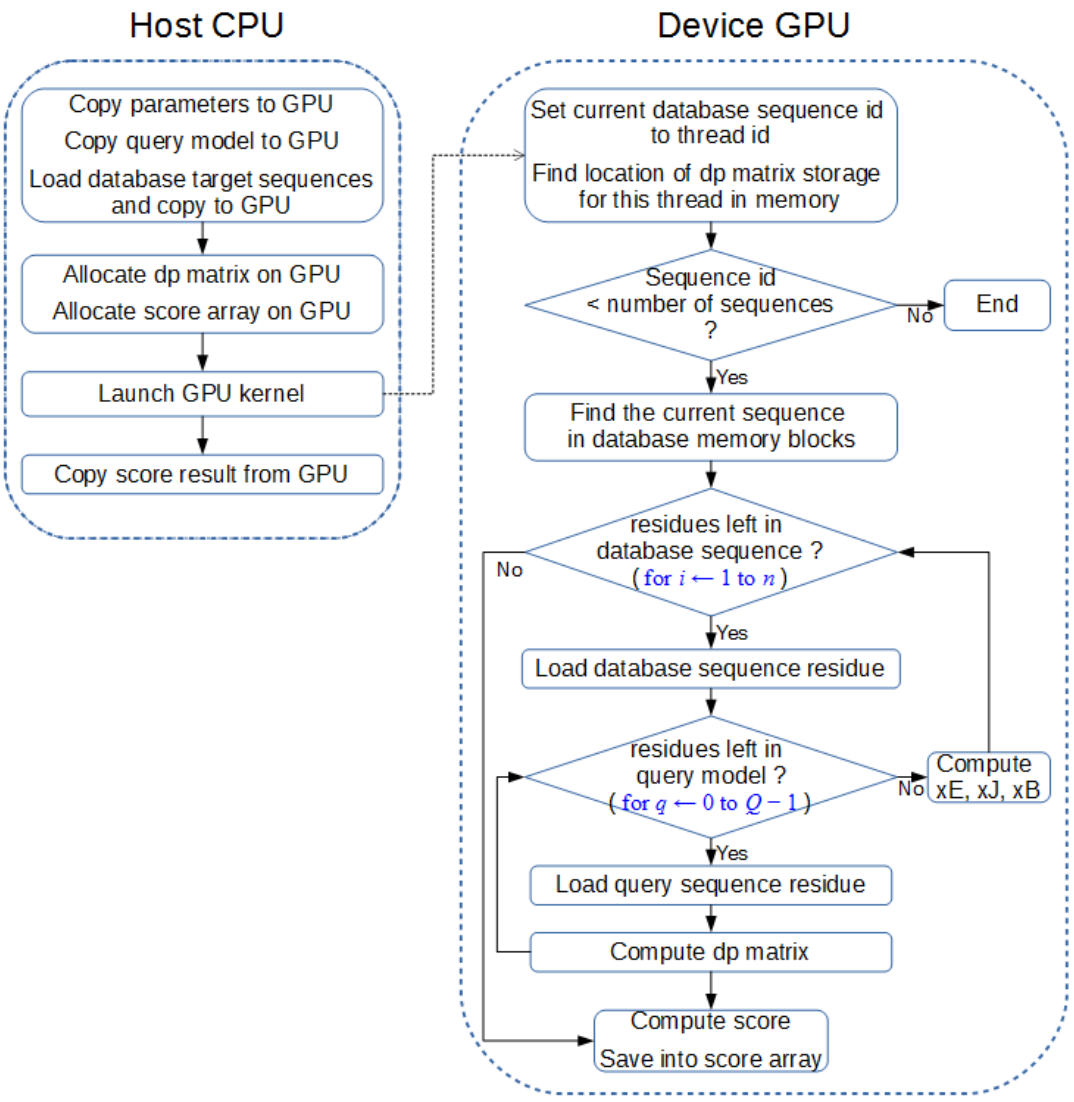

FIGURE 3.2: The GPU porting of MSV filter

compute its dp matrix is determined in the global memory. This is calculated also using the thread id for each thread. When processing the sequence, successive threads access the successive addresses in the global memory for the sequence data and dp matrix, i.e. using a coalesced access pattern. Execution on GPU kernel is halted when every thread finishes its sequence.

3.3 Optimizing the implementation

Although a fully functioning GPU MSV filter has been presented, its simple implementation is quite slow: more than 227 seconds to search the test database Swiss-Prot with 540,958 query sequences, as shown in Tabletab.opt.

This section discusses the optimization steps taken to eventually reach a benchmark database search time of 1.65 seconds: an almost 137 times speedup.

3.3.1 Global Memory Accesses

The global memory is used to store most of data on GPU. A primary in the optimization is to improve the efficiency of accessing global memory as much as possible. One way of course is to reduce the frequency of access. Another way is coalescing access.

Access frequency

The elements of the *dp* matrix and the query profile matrix are 8-bit value. The *uint4* and *ulong2* (see the code below) are 128-bit CUDA built-in vector types. So the access frequency would be decreased 16 times by using *uint4* or *ulong2* to fetch the 8-bit value residing in global memory, compared with using 8-bit *char* type.

```
struct __device_builtin__ uint4
{
unsigned int x, y, z, w;
}
struct __device_builtin__ ulong2
{
unsigned long int x, y;
};
```

This approach resulted in increased register pressure and gained a huge speed boost of almost 8 times in total.

coalescing access

Coalescing access is the single most important performance consideration in programming for CUDA-enabled GPU architectures. Coalescing is a technique applied to combine non-contiguous and small reads/writes of global memory, into the single and more

efficient contiguous and large memory reads. A prerequisite for coalescing is that the words accessed by all threads in a warp must lie in the same segment. As can be seen in Figure3.3, the memory spaces referred to by the same variable names (not referring to same addresses) for all threads in a warp have to be allocated in the form of an array to keep them contiguous in address.

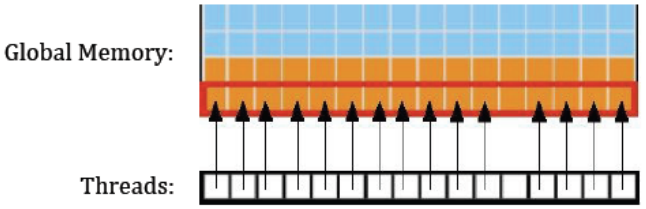

FIGURE 3.3: Coalescing Global Memory Accesses(Waters, 2011).

For coalescing access, the target sequences are arranged in an array like an upside-down bookcase shown in Figure3.4, where all residues of a sequence are restricted to be stored in the same column from top to bottom. And all sequences are arranged in decreasing length order from left to right in the array, which is explained in section3.3.7.

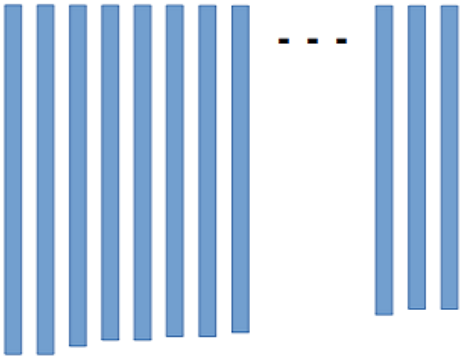

FIGURE 3.4: Alignment of target sequences.

Figure3.5 presents the similar global memory allocation pattern of dp matrix for M processing target sequences. Each thread processes independent dp array with the same length Q. A memory slot is allocated to a thread and is indexed top-to-bottom, and the access to dp arrays is coalesced by using the same index for all threads in a warp.

An alignment requirement is needed to fulfill for fully coalescing, which means any access to data residing in global memory is compiled to a single global memory instruction. The alignment requirement is automatically fulfilled for the built-in types like *uint4* [NVIDIA, 2013-05].

The move to vertical alignment of dp matrix resulted in an improvement of about 44%.

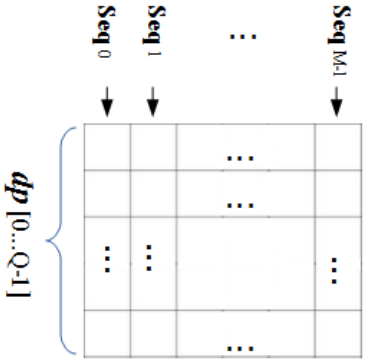

FIGURE 3.5: The dp matrix in global memory.

Note on coding global memory coalescing access

At the beginning, since *uint4* is 16-bytes data block, the traditional C/C++ memory block copy function *memcpy*() was used to copy data between global memory and register memory, as shown in the following code. The *dp* is the pointer to the address of global memory. The *mpv* and *sv* are *uint4* data type residing in register memory.

```
memcpy(&mpv, dp, sizeof(uint4));
memcpy(dp, &sv, sizeof(uint4));
```

However, in practice of CUDA kernel execution, the above *memcpy* involves $16 = sizeof(uint4)$ reads/writes from/to global memory respectively, not 1 read/write. Switching to the following direct assignment instruction will be 1 read/write and fully coalesce access global memory, with 81% improvement over the above *memcpy*().

```
mpv = *(dp);
*(dp) = sv;
```

3.3.2 texture memory

The read-only texture memory space is a cached window into global memory that offers much lower latency and does not require coalescing for best performance. Therefore, a texture fetch costs one device memory read only on a cache miss; otherwise, it just costs one read from the texture cache. The texture cache is optimized for 2D spatial locality, so threads of the same warp that read texture addresses that are close together will achieve best performance [NVIDIA, 2013-05].

Texture memory is well suited to random access. CUDA has optimized the operation fetching 4 values (RGB colors and alpha component, a typical graphics usage) at a time in texture memory. And this mechanism is applied to fetch 4 read-only values from the query profile matrix *texOMrbv* with the *uint4* built-in type. Since the data of target sequences is read-only, it can also use texture for better performance.

Switching to texture memory for the query profile texOMrbv resulted in about 22% performance improvement.

Restrictions using texture memory

Texture memory come from the GPU graphics and therefore are less flexible than the CUDA standard types. It must be declared at compile time as a fixed type, for example *uint4* for the query profile in our case:

```
texture<uint4, cudaTextureType2D, cudaReadModeElementType> texOMrbv;
```

How the values are interpreted is specified at run time. Texture memory is read-only to CUDA kernel and must be explicitly accessed via a special texture API (e.g. tex2D(), tex1Dfetch(), etc) and arrays bound to textures.

```
uint4 rsc4 = tex2D(texOMrbv, x, y);
```

However, on the CUDA next-generation architecture Kepler, the texture cache gets a special compute path, removing the complexity associated with programming it [NVIDIA, 2013-07b].

3.3.3 Virtualized SIMD vector programming model

Inspired by the fact that CUDA has optimized the operation fetching a four component RGBA colour in texture memory, the target sequences is re-organized using a packed data format, where four consecutive residues of each sequence are packed together and represented using the *uchar4* vector data type, instead of the *char* scalar data type, as can be seen in Figure3.6(a). In this way, four residues are loaded using only one texture fetch, thus significantly improving texture memory throughput.

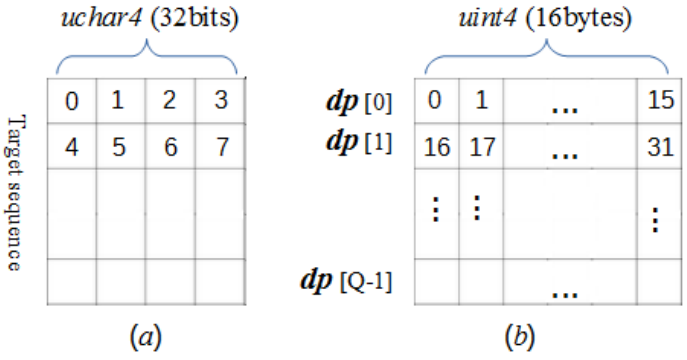

FIGURE 3.6: SIMD vector alignment of data: (*a*) target sequence; (*b*) dp array.

Similarly, the dp array and the query profile also use the virtualized SIMD vector allocation pattern, as can be seen in Figure3.6(*b*).

3.3.4 SIMD Video Instructions

Like Intel SSE2 described in subsection2.2.2.4, CUDA also provides the scalar SIMD (Single Instruction, Multiple Data) video instructions. These are available on devices of compute capability 3.0. The SIMD video instructions enable efficient operations on pairs of 16-bit values and quads of 8-bit values needed for video processing.

The SIMD video instructions can be included in CUDA programs by way of the assembler, *asm*(), statement.

The basic syntax of an *asm*() statement is:

```
asm("template-string" : "constraint"(output) : "constraint"(input));
```

The following three instructions are used in the implementation. Every instruction operates on quads of 8-bit signed values. The source operands("op1" and "op2") and destination operand("rv") are all unsigned 32-bit registers("u32"), which is different from 128-bit in SSE2. For additions and subtractions, saturation instructions("sat") have been used to clamp the values to their appropriate unsigned ranges.

```
/* rv[z] = op1[z] + op2[z] (z = 0,1,2,3) */
asm("vadd4.u32.u32.u32.sat %0, %1, %2, %3;" : "=r"(rv) : "r"(op1), "r"(op2), "r"(0));
/* rv = op1 + op2 */
```

```
asm("vsub4.u32.u32.u32.sat %0, %1, %2, %3;" : "=r"(rv) : "r"(op1), "r"(op2), "r"(0));
/* rv = max(op1,op2) */
asm("vmax4.u32.u32.u32 %0, %1, %2, %3;" : "=r"(rv) : "r"(op1), "r"(op2), "r"(0));
```

Switching to the SIMD video instructions also achieved a large speedup of nearly 2 times.

From here and the acceleration of HMMER3, we can see the parallel vector instructions can greatly improve computing for array or matrix. We can also see the limitation of GPU computing: GPU can only support 32-bit operations, much lower than SSE2 128-bit operations.

3.3.5 Pinned (non-pageable) Memory

It is necessary to transfer data to the GPU over the PCI-E data bus. Compared to access to CPU host memory, this bus is very slow. Pinned memory is memory that cannot be paged (swapped) out to disk by the virtual memory management of the OS. In fact, PCI-E transfer can only be done using pinned memory, and if the application does not allocate pinned memory, the CUDA driver does this in the background for us. Unfortunately, this results in a needless copy operation from the regular (paged) memory to or from pinned memory. We can of course eliminate this by allocating pinned memory ourselves.

In the application, we simply replace *malloc/free* when allocating/freeing memory in the host application with *cudaHostAlloc/cudaFreeHost*.

```
cudaHostAlloc (void** host_pointer, size_t size, unsigned int flags)
```

3.3.6 Asynchronous memory copy and Streams

Asynchronous memory copy

By default, any memory copy involving host memory is synchronous: the function does not return until after the operation has been completed. This is because the hardware cannot directly access host memory unless it has been page-locked or pinned and mapped for the GPU. An asynchronous memory copy for pageable memory could be implemented

by spawning another CPU thread, but so far, CUDA has chosen to avoid that additional complexity.

Even when operating on pinned memory, such as memory allocated with *cudaMallocHost*(), synchronous memory copy must wait until the operation is finished because the application may rely on that behavior. When pinned memory is specified to a synchronous memory copy routine, the driver does take advantage by having the hardware use DMA, which is generally faster [Wilt, 2013].

When possible, synchronous memory copy should be avoided for performance reasons. Keeping all operations asynchronous improves performance by enabling the CPU and GPU to run concurrently. Asynchronous memory copy functions have the suffix *Async*(). For example, the CUDA runtime function for asynchronous host to device memory copy is *cudaMemcpyAsync*().

It works well only where either the input or output of the GPU workload is small in comparison to one another and the total transfer time is less than the kernel execution time. By this means we have the opportunity to hide the input transfer time and only suffer the output transfer time.

Multiple streams

A CUDA stream represents a queue of GPU operations that get executed in a specific order. We can add operations such as kernel launches, memory copies, and event starts and stops into a stream. The order in which operations are added to the stream specifies the order in which they will be executed. CUDA streams enable CPU/GPU and memory copy/kernel processing concurrency. For GPUs that have one or more copy engines, host $\longleftrightarrow$ device memory copy can be performed while the SMs are processing kernels. Within a given stream, operations are performed in sequential order, but operations in different streams may be performed in parallel [Sanders, 2011].

To take advantage of CPU/GPU concurrency as depicted in Figure3.7, when performing memory copies as well as kernel launches, asynchronous memory copy must be used. And Mapped pinned memory can be used to overlap PCI Express transfers and kernel processing.

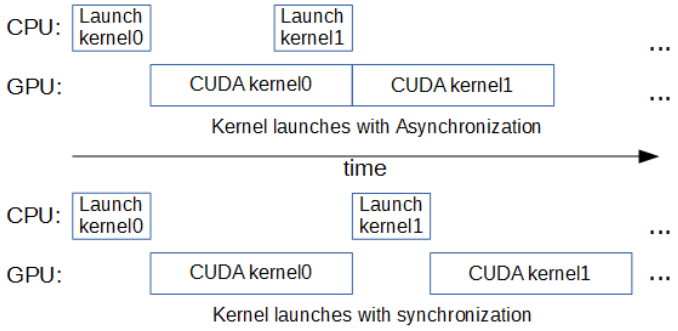

FIGURE 3.7: CPU/GPU concurrency.

CUDA compute capabilities above 2.0 are capable of concurrently running multiple kernels, provided they are launched in different streams and have block sizes that are small enough so a single kernel will not fill the whole GPU.

By using multiple streams, we broke the kernel computation into chunks and overlap the memory copies with kernel execution. The new improved implementation might have the execution timeline as shown in Figure3.8 in which empty boxes represent time when one stream is waiting to execute an operation that it cannot overlap with the other stream's operation.

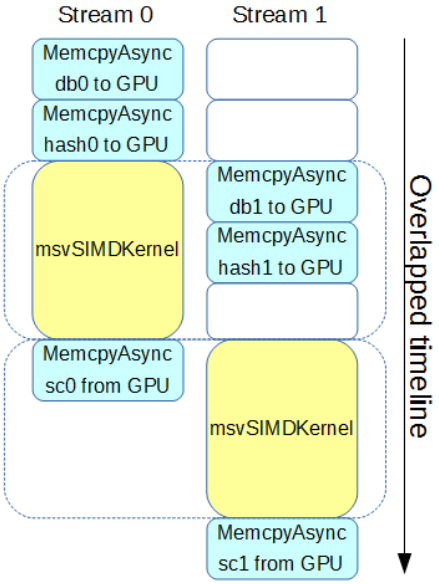

FIGURE 3.8: Timeline of intended application execution using two independent streams.

Running the improved program using pinned memory, asynchronous memory copy and two streams reveals the time drops from 16.45s to just 9.46s, a quite significant drop of over 73.8% in execution time.

3.3.7 Sorting database

As described in Section2.2.2.3, MSV filter function is sensitive to the length of a target sequence, which determines the execution times of main *for* loop in Algorithm2.2.3.

Target sequence database could contain many sequences with different lengths. The NCBI NR database used in this thesis consists of over 38 million sequences with sequence lengths varying from 6 $\sim$ 41,943 amino acids [NCBI, 2014].

This brings a problem for parallel processing of threads on GPU: one thread could be processing a sequence of several thousands of residues while another might be working on a sequence of just a few. As a result, the thread that finishes first might be idle while the long sequence is being handled. Furthermore, unless care is taken when assigning sequences to threads, this effect might be compounded by the heavily unbalanced workload among threads.

In order to achieve high efficiency for task-based parallelism, the run time of all threads in a thread block should be roughly identical. Therefore the database is converted with sequences being sorted by length. Thus, for two adjacent threads in a thread warp, the difference value between the lengths of the associated sequences is minimized, thereby balancing a similar workload over threads in a warp.

Block reading

Many research implementation were not concerned with practical matters. They just loaded the whole database data into memories of CPU host and GPU device. Large database, like NCBI NR database, is more than 24GB in size and is still increasing, being too large to load into memories of most machines.

Given GPU global memory is much less than CPU host, we use the size of GPU global memory as the basis to decide the size of sequence block while reading the database.

Descending order

The memory pools for database sequences both in CPU host and GPU device are dynamically allocated at run time. The pools may be required to reallocated due to more

space needed for the current data block than the last one. If the pool allocated at the first time is the largest one during execution, then the overhead of reallocation will be saved. Hence, the descending order is used for sorting the database.

Performance improved

The CUDA profiling tool nvprof [NVIDIA, 2014d] was used to understand and optimize the performance of the MSV GPU application cudaHmmsearch. The nvprof command was used as follows:

```
# nvprof ./cudaHmmsearch globins4.hmm uniprot_sprot.fasta
```

The table3.1 and 3.2 list the profiling results of before and after the target database was sorted.

Time(%)	Time	Calls	Avg	Min	Max	Name
91.61%	4.27108s	134	31.874ms	5.9307ms	137.67ms	msvSIMDKernel
8.37%	390.01ms	271	1.4391ms	704ns	23.027ms	[CUDA memcpy HtoD]
0.01%	556.21us	134	4.1500us	1.7280us	5.9840us	[CUDA memcpy DtoH]
0.01%	491.23us	134	3.6650us	3.4880us	4.0640us	[CUDA memset]

TABLE 3.1: **Profiling result of before sorting database.** Each row is the statistics of profiling result for the function named in the '**Name**'. The statistics includes the percentage of running time, the running time, the number of called times, as well as the average, minimum, and maximum time.

Time(%)	Time	Calls	Avg	Min	Max	Name
97.41%	2.07263s	134	15.467ms	29.056us	194.91ms	msvSIMDKernel
2.54%	54.115ms	271	199.69us	704ns	23.013ms	[CUDA memcpy HtoD]
0.03%	550.53us	134	4.1080us	1.6640us	4.6720us	[CUDA memcpy DtoH]
0.02%	474.90us	134	3.5440us	672ns	4.0320us	[CUDA memset]

TABLE 3.2: **Profiling result of after sorting database.** The meaning of each column is same as Table3.1

From the result of profiling, we can see the performance has been increased, which is clearly shown in two ways:

1. the ratio of the msvSIMDKernel run-time to the total increased from 91.61% to 97.41%;

2. the msvSIMDKernel run-time decreased from 4.27108s to 2.07263s and the time of the memory copy from Host to Device (CUDA memcpy HtoD) decreased from 390.01ms to 54.115ms.

This approach has the advantage of being both effective and quite straightforward as a large 129% performance improvement can be gained over the unsorted database without changing the GPU kernel in any way (see Table4.2 and Figure4.1). For the 24GB NCBI NR database used in these experiments, only 6 minutes were taken for sorting. Further, the sorted database can still be usable for other applications, making the one-time cost of sorting it negligible.

3.3.8 Distributing workload

After launching GPU kernel, CPU must wait for the GPU to finish before copying back the result. This is accomplished by calling *cudaStreamSynchronize*(*stream*). We can get further improvement by distribute some work from GPU to CPU while CPU is waiting. In protein database, the sequences with the longest or the shortest length are very few. According to Swiss-Prot database statistics [UniProtSP, 2014-05], the percentage of sequences with length > 2500 is only 0.2%. Considering the length distribution of database sequences and based on the descending sorted database discussed in section3.3.7, we assigned the first part of data with longer lengths to CPU. By this way, we can save both the GPU global memory allocated for sequences and the overheads of memory transfer.

The compute power of CPU and GPU should be taken into consideration in order to balance the workload distribution between CPU and GPU. The distribution policy calculates a ratio R of the number of database sequences assigned to GPU, which is calculated as

$$R = \frac{N_G f_G}{N_G f_G + f_C}$$

where f_G and f_C are the core frequencies of GPU and CPU, N_G is the number of GPU Multiprocessors.

3.3.9 Miscellaneous consideration

This sections discusses various small-scale optimization and explains some techniques not suited to the MSV implementation.

Data type for register memory

In order to reduce the register pressure in CUDA kernel, we may consider using unsigned 8-bit char type (u8) instead of 32-bit int type (u32). Declaring the registers as u8 results in sections of code to shift and mask data. The extract data macros are deliberately written to mask off the bits that are not used, so this is entirely unnecessary. In fact, around four times the amount of code will be generated if using an u8 type instead of an u32 type.

Changing the u8 definition to an u32 definition benefits from eliminating huge numbers of instructions. It seems potentially wasting some register space. In practice, CUDA implements u8 registers as u32 registers, so this does not actually cost anything extra in terms of register space [Cook, 2013].

Branch divergence

A warp executes one common instruction at a time, so full efficiency is realized when all 32 threads of a warp agree on their execution path. If threads of a warp diverge via a data-dependent conditional branch, the warp serially executes each branch path taken, disabling threads that are not on that path, and when all paths complete, the threads converge back to the same execution path. Branch divergence occurs only within a warp; different warps execute independently regardless of whether they are executing common or disjoint code paths. So the divergence results in some slowdown.

Since the implementation has changed u8 type in register memory to u32 type, the test for the overflow condition is not needed any more. This not only saves several instructions, but also avoids the issue of branch divergence.

Constant memory

Constant memory is as fast as reading from a register as long as all threads in a warp read the same 4-byte address. Constant memory does not support, or benefit from, coalescing, as this involves threads reading from different addresses. Thus, parameters used by all threads, such as $base$, t_{jb}, t_{ec}, are stored into constant memory.

Shared memory

In terms of speed, shared memory is perhaps 10x slower than register accesses but 10x faster than accesses to global memory. However, some disadvantages apply to shared memory.

- Unlike the L1 cache, the shared memory has a per-block visibility, which would mean having to duplicate the data for every resident block on the SM.
- Data must be loaded from global to shared memory in GPU kernel and can not be uploaded to shared memory directly from the host memory.
- Shared memory is well suited to exchange data between CUDA threads within a block. As described in subsection3.1, task-based parallelism is applied without the need for inter-thread communications, which also saves the cost of synchronization $\_\_syncthreads()$ among threads.

Because of these disadvantages, the MSV implementation does not use shared memory.

Kernel launch configuration

Since the MSV implementation does not use shared memory explained above, the following dynamic kernel launch configuration is used to prefer larger L1 cache and smaller shared memory so as to further improve memory throughput.

```
cudaFuncSetCacheConfig(msvSIMDKernel, cudaFuncCachePreferL1);
```

3.4 Conclusion of optimization

This section briefly reviews the all the optimization approaches discussed in this chapter thus far, and summarizes the steps to gain better performance for CUDA programming.

Six steps to better performance

1. Assessing the application

   In order to benefit from any modern processor architecture, including GPUs, the first steps are to assess the application to identify the hotspots [MSV filter in Section3.2.1], which type of parallelism [Task-based parallelism in Section3.1] is better suited to the application.

2. Application Profiling

   NVIDIA provides profiling tools to help identify hotspots and compile a list of candidates for parallelization or optimization on CUDA-enabled GPUs [nvprof in Section3.3.7], as detailed in Section2.3.1.4. Intel provides VTune Amplifier XE to collect a rich set of data to tune CPU & GPU compute performance at `https://software.intel.com/en-us/intel-vtune-amplifier-xe`.

3. Optimizing memory usage

   Optimizing memory usage starts with minimizing data transfers both in size [Database sorted in Section3.3.7, workload distribution in Section3.3.8] and time [Pinned Memory in Section3.3.5] between the host and the device [Asynchronous memory copy in Section3.3.6]. Be careful with CUDA memory hierarchy: register memory [Section3.3.9], local memory, shared memory [Section3.3.9], global memory, constant memory [Section3.3.9] and texture memory [Section3.3.2], and combine these memories to best suit the application [Kernel launch configuration in Section3.3.9]. Sometimes, the best optimization might even be to avoid any data transfer in the first place by simply recomputing the data whenever it is needed.

   The next step in optimizing memory usage is to organize memory accesses according to the optimal memory access patterns. This optimization is especially important for coalescing global memory accesses [Section3.3.1].

4. Optimizing instruction usage

   This suggests using SIMD Video Instructions [Section3.3.4] and trading precision

for speed when it does not affect the end result, such as using intrinsic instead of regular functions or single precision instead of double precision [HMMER3 in Section2.2.2.4]. Particular attention should be paid to control flow instructions [Branch divergence in Section3.3.9].

5. Maximizing parallel execution

   The application should maximize parallel execution at a higher level by explicitly exposing concurrent execution on the device through streams [Section3.3.6], as well as maximizing concurrent execution between the CPU host [Database sorted in Section3.3.7] and the GPU device [Workload distribution in Section3.3.8 and SIMD Video Instructions in Section3.3.4].

6. Considering the existing libraries

   Many existing GPU-optimized libraries [NVIDIA, 2014e] such as cuBLAS [NVIDIA, 2014f], MAGMA [MAGMA, 2014], ArrayFire [ArrayFire, 2014], or Thrust [NVIDIA, 2014g], are available to make the expression of parallel code as simple as possible.

Chapter 4

Benchmark results and discussion

Chapter 3 described several approaches to optimize the cudaHmmsearch implementation. This chapter presents the performance measurements when experimenting these approaches on a GPU and on a multicore CPU.

4.1 Benchmarking environment

The benchmarking environment were set up in Kronos machine as follows:

- CPU host

  Intel Core i7-3960X with 6 cores, 3.3GHz clock speed, 64GB RAM

- GPU device

  NVIDIA Quadro K4000 graphics card with 3 GB global memory, 768 Parallel-Processing Cores, 811 MHz GPU Clock rate, CUDA Compute Capability 3.0.

- Software system

  The operating system used was Ubuntu 64 bit Linux v12.10; the CUDA toolkit used was version 5.5.

- Target sequences database

  One was Swiss-Prot database in fasta format released in September 2013 [UniProt, 2013-09] containing 540,958 sequences with length varying from 2 $\sim$ 35,213 amino acids, comprising 192,206,270 amino acids in total, more than 258MB in file size.

Another was much larger NCBI NR database in fasta format released in April 2014 [NCBI, 2014] containing 38,442,706 sequences with length varying from 6 $\sim$ 41,943 amino acids, comprising 13,679,143,700 amino acids in total, more than 24GB in file size.

- Query profile HMMs

  We tested 5 profile HMMs of length 149, 255, 414, 708 and 1111 states, detailed in Table4.1. Globin4 with length of 149 states was distributed with the HMMER source [HMMER, 2014a]. Other 4 HMMs were taken directly from the Pfam database [Pfam, 2013] that vary in length from 255 to 1111 states.

Name	Globin4	120_Rick_ant	2HCT	ACC_central	AAA_27
Accession number	-	PF12574.3	PF03390.10	PF08326.7	PF13514.1
Length	149	255	414	708	1111

TABLE 4.1: Profile HMMs used in benchmarking. Globin4 has no Accession number.

- Measuring method

  The execution time of the application was timed using the C clock() instruction. The performance was measured in unit GCUPS(Giga Cell Units Per Second) which is calculated as follows:

$$GCUPS = \frac{L_q * L_t}{T * 1.0e09}$$

  where L_q is the length of query profile HMM, i.e. the number of the HMM states, L_t is the total residues of target sequences in the database, T is the execution time in second.

  All programs were compiled using GNU g++ with the -O3 option and executed independently in a 100% idle system.

4.2 Performance Results

4.2.1 Comparison with less optimized approaches

To show the performance impact of several selected optimization approaches, the performance of the implementation was compared with that of previous approach.

Table4.2 shows the approaches taken in optimizing performance. All tests are taken against the Swiss-Prot database. The query HMM used was globin4. The fourth column 'Improvement' is measured in percentage compared with the previous approach.

Description of approach	**Execution time (s)**	**Performance** (GCUPS)	**Improvement** (%)
Initial implementation	227.178	0.126	-
SIMD Video Instruction	125.482	0.228	81
Minimizing global memory access	16.449	1.741	664
Async memcpy & Multi streams	9.463	3.026	74
Coalescing of global memory	6.565	4.362	44
Texture memory	5.370	5.333	22
Sorting Database	2.346	12.207	129
Distributing workload	1.650	17.357	42

TABLE 4.2: **Performance of optimization approaches.** The fourth column **Improvement** is measured in percentage compared with the previous approach. The row 'Coalescing of global memory' is benchmarked only for the *dp* matrix. The row 'Texture memory' is benchmarked only for the query profile texOMrbv 2D texture.

The graphic view corresponding to Table is shown in Figure4.1.

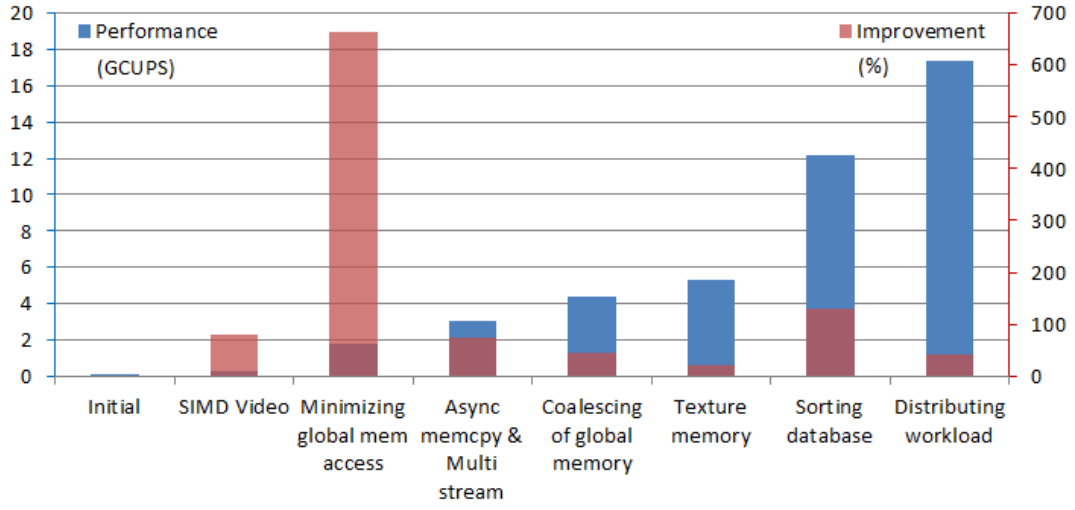

FIGURE 4.1: **Performance of optimization approaches.** The data of this chart come from Table4.2. The blue bar is Performance (in GCUPS) of each approach, corresponding to the left Y axis. The red bar is Improvement in % corresponding to the right Y axis.

From the chart, it can be seen that several factors are related to global memory accesses, including the highest 663% minimizing global memory access, coalescing of global memory and texture memory. So the global memory optimizations are the most important area for performance. To make all threads in a warp execute similar tasks, the auxiliary sorting database also plays important role in optimizations.

4.2.2 Practical benchmark

The final cudaHmmsearch implementation, with the optimization discussed in Chapter3, was benchmarked to determine its real-world performance. This was done by searching the much large NCBI NR database for the 5 profile HMMs with various lengths, as detailed in Table4.1. As comparison, the same searches were executed by hmmsearch of HMMER3 on 1 CPU core. The result of this benchmark is shown in the following table.

Profile HMM (length)	**globins4** (149)	**120_Rick_ant** (255)	**2HCT** (414)	**ACC_central** (708)	**AAA_27** (1111)
Performance of hmmsearch (GCUPS)	9.37	11.72	11.68	11.96	6.90
Performance of cudaHmmsearch (GCUPS)	23.00	32.17	30.01	32.83	14.68
Speedup (times)	2.45	2.74	2.57	2.75	2.13

TABLE 4.3: **Result of Practical benchmark.** Speedup is measured in times of cudaHmmsearch performance over that of hmmsearch.

The results of the benchmarks are shown in graphical form in Figure4.2. The GPU cudaHmmsearch performance hovers just above 25 GCUPS, while the CPU hmmsearch only around 10 GCUPS. The whole performance of cudaHmmsearch is stable with various lengths of query HMMs. On average, cudaHmmsearch has a speedup of 2.5x than hmmsearch.

Figure4.2 shows that the performance of both GPU and CPU searching for AAA_27 dropped greatly. The reason can be seen from the table4.4 listing the internal pipeline statistics summary for searching globin4 and AAA_27. For every filter, the count of passed sequences for searching AAA_27 with 1111 states is much more than that of globin4 with 149 states. This means that for searching AAA_27, much more target sequences than searching globin4 are needed calculating in each filter after MSV filter.

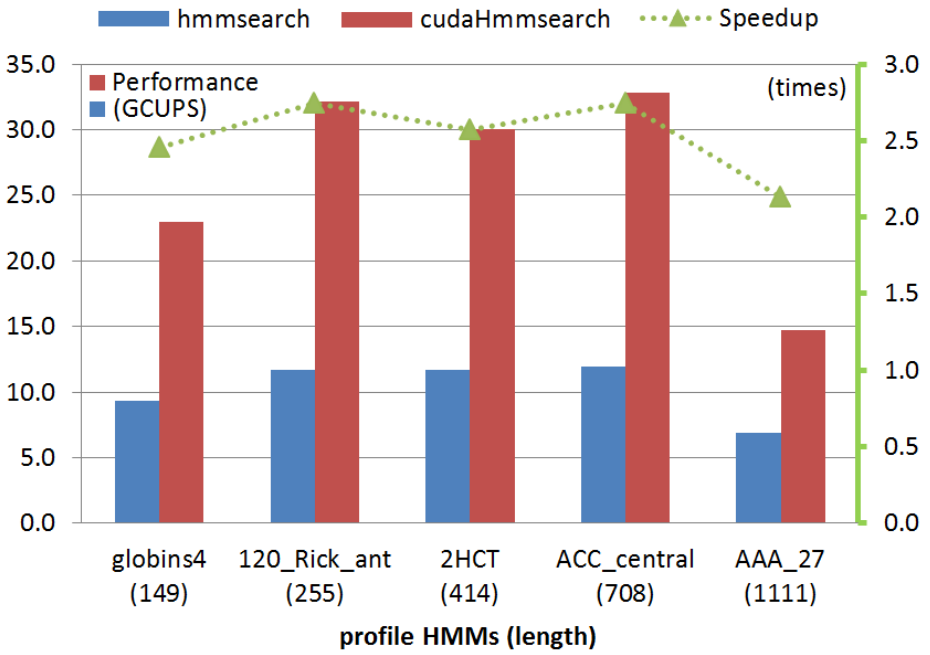

FIGURE 4.2: **Practical benchmarks.** The data of this chart come from Table4.3. The blue and red bar is Performance (in GCUPS) of hmmsearch and cudaHmmsearch respectively, corresponding to the left Y axis. The green dot line is Speedup (in times) of cudaHmmsearch performance over that of hmmsearch, corresponding to the right Y axis.

And at the same time, each filter is more time-consuming than MSV filter. All of these result in the AAA_27 performance dropping greatly.

Query profile HMM(length)	**globin4 (149 states)**	**AAA_27 (1111 states)**
Target sequences	38442706	38442706
Passed MSV filter	1195043	4305846
Passed bias filter	973354	1671084
Passed Viterbi filter	70564	322206
Passed Forward filter	7145	17719

TABLE 4.4: Internal pipeline statistics summary

4.2.3 Comparison with multicore CPU

Since multicore processors were developed in the early 2000s by Intel, AMD and others, nowadays CPU has become multicore with two cores, four cores, six cores and more. The Kronos 4.1 experiment system has CPU with six cores. This section presents the benchmarks of cudaHmmsearch running with multiple CPU cores.

The experiment was done by executing cudaHmmsearch and hmmsearch with 1, 2...6 CPU cores, searching the NCBI NR database for the HMM with 255-state length.

The benchmark has not been used in the articles cited in this thesis. The purpose of this benchmark is to show how the performance increase or decrease with more CPU cores involved in computing. The result of this benchmark is shown in the following table.

Performance (GCUPS)	1 CPU core	2 CPU cores	3 CPU cores	4 CPU cores	5 CPU cores	6 CPU cores
cudaHmmsearch	32.17	50.22	57.70	59.14	59.39	59.29
hmmsearch	11.72	23.28	29.22	44.15	46.19	44.69

TABLE 4.5: Result of Comparison with multicore CPU.

The graphic view of the benchmark is shown in Figure4.3. The number above each bar is the Performance in GCUPS. As can be seen, from 1 CPU core to 4 CPU cores, both cudaHmmsearch performance and hmmsearch performance go up almost linearly. From then on, due to complex schedule among CPU cores, the extra CPU core will not contribute much to both cudaHmmsearch and hmmsearch execution. Even worse, it will have negative effect as shown clearly in the '6 CPU' case.

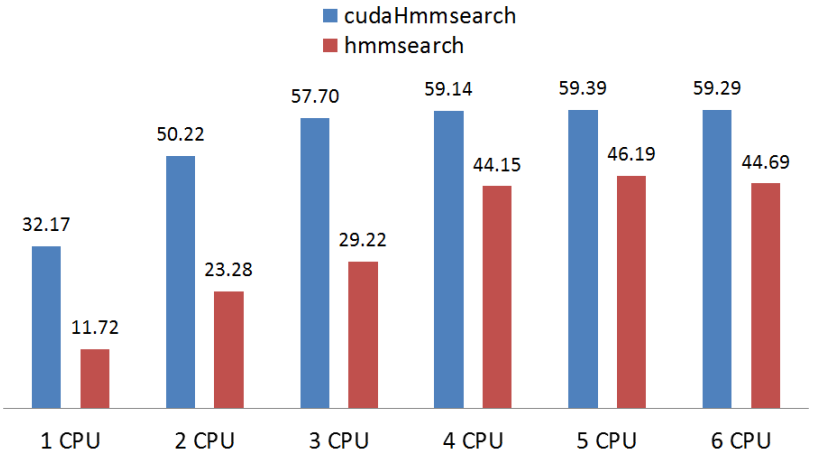

FIGURE 4.3: **Comparison with multicore CPU.** The data of this chart come from Table4.5. The number above each bar is the Performance in GCUPS.

4.2.4 Comparison with other implementations

The performance of cudaHmmsearch was also compared to the previous HMMER solutions: HMMER2.3.2 [HMMER2.3.2, 2003], GPU-HMMER2.3.2 [Walters et al., 2009] and HMMER3 [HMMER, 2014a].

All tests are taken searching against the Swiss-Prot database for the globin4 profile HMM. The result of this benchmark is shown in the following table.

Application (Device)	HMMER2.3.2 (CPU)	GPU-HMMER2.3.2 (GPU)	HMMER3 (CPU)	cudaHmmsearch (GPU)
Performance (GCUPS)	0.14	0.95	8.47	17.36

TABLE 4.6: Result of Comparison with other implementations.

As seen from the Figure4.4, since the release of HMMER2.3.2 in Oct 2003, accelerating hmmsearch researches on both CPU and GPU have achieved excellent improvement.

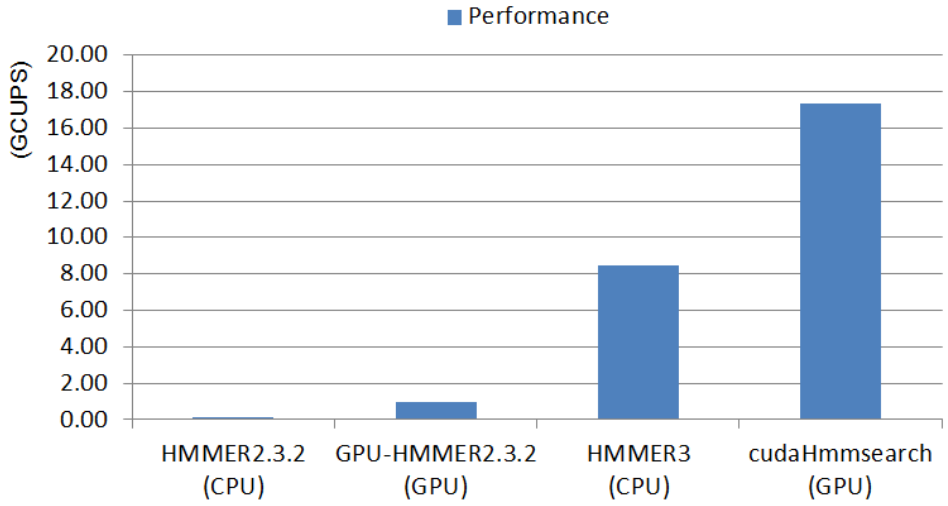

FIGURE 4.4: Comparison with other implementations.

Chapter 5

Conclusions

A fully-featured and accelerated HMMER3 protein search tool *cudaHmmsearch* was implemented on CUDA-enabled GPU. It can search the protein sequence database for profile HMMs.

5.1 Summary of Contributions

Our research work started with dynamic programming, the common characteristic of Swith-Waterman algorithm, Viterbi algorithm and MSV algorithm, which are famous protein sequence alignment algorithms in Bioinformatics. We specially summarized the technologies of accelerating Swith-Waterman algorithm on CUDA-enabled GPU, on which has been widely researched. We also briefly presented GPU acceleration work related to Viterbi algorithm.

After analyzing the core application *hmmsearch* in HMMER3, we found the key hotspot MSV filter for accelerating hmmsearch. We presented the details of our *cudaHmmsearch* implementation and optimization approaches. At the same time, we also discussed and analyzed the advantages and limitations of GPU hardware for CUDA parallel programming. Then we summarized 6 steps for better performance of CUDA programming.

We performed comprehensive benchmarks. The results were analyzed and the efficiency of the *cudaHmmsearch* implementations on the GPUs is proved. We achieved 2.5x

speedup over the single-threaded HMMER3 CPU SSE2 implementation. The performance analysis showed that GPUs are able to deal with intensive computations, but are very sensitive to random accesses to the global memory.

The solutions in this thesis were designed and customized for current GPUs, but we believe that the principles studied here will also apply to future manycore GPU processors, as long as the GPU is CUDA-enabled. Here is the complete list of CUDA-enabled GPUs: https://developer.nvidia.com/cuda-gpus.

5.2 Limitations of Work

There are some weak points in our work summarized as follows:

Although our *cudaHmmsearch* can search against unsorted protein sequence database, it can gain 129% improvement searching against sorted database according to benchmark Table4.2. And although the extra sorting database program is provided, user may be unaware of this and run *cudaHmmsearch* against an unsorted database. It is better for the program to evaluate the database automatically and prompt user to sort if necessary.

We use block reading method to process very large database. However, the number of sequences for each block reading is fixed. So the number of threads launched in GPU kernel is also fixed. For those sequences with shorter lengths, it is better to use dynamic block reading to get more sequences, so as to increase the occupancy of GPU threads and achieve better performance.

5.3 Recommendations for Future Research

The limitations noted in the last section call attention to several areas that we deem worthy of further improvement and investigation. The suggested topics are placed under the following headings.

Forward filter for no threshold

By default, the top-scoring of target sequences are expected to pass each filter. Alternatively, the *--max* option is available for those who want to make a search more sensitive to get maximum expected accuracy alignment. The option causes all filters except Forward/Backward algorithm to be bypassed. And according to practical benchmarking in Section4.2.2, the performance decreased greatly due to much more calculation in Forward algorithm. So our next research should be focused on accelerating Forward algorithm on CUDA-enabled GPU.

Multiple GPUs approach

Since currently we don't have multiple GPUs within a single workstation, we didn't research on multiple GPUs approach. However, CUDA already provides specific facilities for multi-GPU programming, including threading models, peer-to-peer, dynamic parallelism and inter-GPU synchronization, etc. Almost all PCs support at least two PCI-E slots, allowing at least two GPU cards to insert almost any PC. Looking forward, we should also investigate multi-GPU solutions.

Appendix A

Resource of this thesis

The code of *cudaHmmsearch* and this thesis has submitted to Google subversion server powered by Google Project Hosting and can be accessed in a web browser or subversion client with the address: http://cudahmmsearch.googlecode.com/svn/trunk/.

Bibliography

[Ahmed et al., 2012] Fahian Ahmed, Saddam Quirem, Gak Min and Byeong Kil Lee *Hotspot Analysis Based Partial CUDA Acceleration of HMMER 3.0 on GPGPUs*. International Journal of Soft Computing and Engineering (IJSCE) ISSN: 2231-2307, Volume-2, Issue-4, September 2012.

[Aji et al., 2008] S. Aji, F. Blagojevic, W. Feng, D.S. Nikolopoulos. *Cell-Swat: Modeling and Scheduling Wavefront Computations on the Cell Broadband Engine*. Proceedings of the 2008 ACM International Conference on Computing Frontiers 13-22.

[Akoglu and Striemer, 2009] Ali Akoglu and Gregory M. Striemer *Scalable and highly parallel implementation of Smith-Waterman on graphics processing unit using CUDA*. Springer Science+Business Media, LLC 2009.

[ArrayFire, 2014] AccelerEyes. Comprehensive GPU function library. Website, 2014. URL http://www.accelereyes.com/.

[Cook, 2013] Shane Cook. *CUDA Programming : A Developer's Guide to Parallel Computing with GPUs*. Elsevier Inc. USA, 2013.

[Baldi and Brunak, 2001] Pierre Baldi and Soren Brunak. *Bioinformatics: The Machine Learning Approach*. The MIT Press, England, 2nd edition, 2001.

[Dong and Pei, 2007] Guozhu Dong and Jian Pei. *Sequence Data Mining*. Springer Science+Business Media, LLC, New York, USA, 2007.

[Du et al., 2010] Zhihui Du, Zhaoming Yin, and David A. Bader *A Tile-based Parallel Viterbi Algorithm for Biological Sequence Alignment on GPU with CUDA*. Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on , vol., no., pp.1,8, 19-23 April 2010.

[Durbin et al., 1998] Richard Durbin, Sean Eddy, Anders Krogh, and Graeme Mitchison. *Biological sequence analysis: Probabilistic models of proteins and nucleic acids.* Cambridge University Press, New York, USA, 1998.

[Eddy, 2011] Sean R. Eddy. *Accelerated profile HMM searches.* PLoS Comput Biol 7(10): e1002195. doi:10.1371/journal.pcbi.1002195, October 2011. URL http://www.ploscompbiol.org.

[Eidhammer et al., 2004] Ingvar Eidhammer, Inge Jonassen, and William R. Taylor. *Protein Bioinformatics: An Algorithmic Approach to Sequence and Structure Analysis.* John Wiley & Sons, New Jersey, USA, 2004.

[Farrar, 2007] Farrar M. *Striped Smith-Waterman speeds database searches six times over other SIMD implementations.* Bioinformatics 23: 156–161 2007.

[Ganesan et al., 2010] Narayan Ganesan, Roger D. Chamberlain, Jeremy Buhler, and Michela Taufer *Accelerating HMMER on GPUs by Implementing Hybrid Data and Task Parallelism.* Proc. of ACM International Conference on Bioinformatics and Computational Biology, August 2010, pp. 418-421.

[Gotoh, 1982] Gotoh O. *An improved algorithm for matching biological sequences.* J. Mol. Biol 1982, 162:705-708.

[Hancock and Zvelebil, 2004] John M. Hancock and Marketa J. Zvelebil, editors. *Dictionary of Bioinformatics and Computational Biology.* John Wiley & Sons, New Jersey, USA, 2004.

[Henikoff and Henikoff, 1992] S. Henikoff and J.G. Henikoff *Amino Acid Substitution Matrices from Protein Blocks.* PNAS 89 (22): 10915-10919. doi:10.1073/pnas.89.22.10915. PMC 50453. PMID 1438297, 1992.

[HMMER, 2014a] Howard Hughes Medical Institute. HMMER source code release archives. Website, 2014. URL http://hmmer.janelia.org/software/archive.

[HMMER, 2014b] Howard Hughes Medical Institute. HMMER. Website, 2014. URL http://hmmer.janelia.org/.

[HMMER2.3.2, 2003] Howard Hughes Medical Institute. HMMER2.3.2 source code release. Website, 2003. URL http://selab.janelia.org/software/hmmer/2.3.2/hmmer-2.3.2.tar.gz.

[Horn et al., 2005] Daniel Horn, Mike Houston, and Pt Hanrahan *ClawHMMER: A Streaming HMMer-Search Implementation.* presented at Supercomputing 2005, Washington, D.C., 2005.

[Intel, 2013] Intel. Intel VTune Amplifier XE 2013. Website, 2013. URL https://software.intel.com/en-us/intel-vtune-amplifier-xe/.

[Kentie, 2010] M.A. Kentie *Biological Sequence Alignment Using Graphics Processing Units.* Master thesis (2010), Delft University of Technology.

[Kirk and Hwu, 2010] David B. Kirk and Wen-mei W. Hwu *Programming Massively Parallel Processors : A Hands-on Approach.* Published by Elsevier Inc. 2010.

[Lesk, 2008] Arthur M. Lesk. *Introduction to Bioinformatics.* Oxford University Press, New York, USA, 3rd edition, 2008.

[Ligowski and Rudnicki, 2009] Lukasz Ligowski and Witold Rudnicki *An Efficient Implementation Of Smith Waterman Algorithm On GPU Using CUDA, For Massively Parallel Scanning Of Sequence Databases.* IEEE 2009.

[Liu et al., 2009] Yongchao Liu, Douglas Maskell and Bertil Schmidt *CUDASW++: optimizing smith-waterman sequence database searches for cuda-enabled graphics processing units.* BMC Research Notes 2, no. 1, 73. 2009. URL http://www.biomedcentral.com/1756-0500/2/73.

[Liu et al., 2010] Yongchao Liu, Bertil Schmidt, and Douglas Maskell *cudasw++2.0: enhanced smith-waterman protein database search on cuda-enabled gpus based on simt and virtualized simd abstractions.* BMC Research Notes 2010. URL http://www.biomedcentral.com/1756-0500/3/93.

[Liu et al., 2013] Yongchao Liu, Adrianto Wirawan and Bertil Schmidt *CUDASW++ 3.0: accelerating Smith-Waterman protein database search by coupling CPU and GPU SIMD instructions.* BMC Bioinformatics 2013. URL http://www.biomedcentral.com/1471-2105/14/117.

[Loewy et al., 1991] Ariel G. Loewy, Philip Siekevitz, John R. Menninger and Jonathan A. N. Gallant *Cell Structure & Function : An Integrated Approach.* Saunders College Publishing, 3rd edition, 1991.

[M.Isa et al., 2012] M.Nazrin M.Isa, Khaled Benkrid and Thomas Clayton *A Novel Efficient FPGA Architecture for HMMER Acceleration*. IEEE, 2012.

[MAGMA, 2014] The University of Tennesse. A collection of next generation linear algebra (LA) GPU accelerated libraries MAGMA. Website, 2014. URL http://icl.cs.utk.edu/magma/software/.

[Manavski, 2008] Svetlin A Manavski and Giorgio Valle *CUDA compatible GPU cards as efficient hardware accelerators for Smith-Waterman sequence alignment*. BMC Bioinformatics 2008. URL http://www.biomedcentral.com/1471-2105/9/S2/S10.

[Markel and Leon, 2003] Scott Markel and Darry Leon. *Sequence Analysis in a Nutshell*. O'Reilly & Associates, Inc., USA, 2003.

[Microsoft, 2013-11] Microsoft. Compute shader overview. Website, 2013-11. URL http://msdn.microsoft.com/en-us/library/windows/desktop/ff476331%28v=vs.85%29.aspx.

[Microsoft, 2013-12] Microsoft. Programming guide for HLSL. Website, 2013-12. URL http://msdn.microsoft.com/en-us/library/windows/desktop/bb509635%28v=vs.85%29.aspx.

[NCBI, 2014] National Center for Biotechnology Information, U.S. National Library of Medicine. NCBI NR (non-redundant) Protein Database. Website, 2014-4-7. URL ftp://ftp.ncbi.nih.gov/blast/db/FASTA/nr.gz.

[NVIDIA, 2013-05] NVIDIA. CUDA C PROGRAMMING GUIDE. Website, PG-02829-001_v5.5 May 2013. URL http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[NVIDIA, 2013-07a] NVIDIA. CUDA C Best Practices Guide. Website, DG-05603-001_v5.5 July 2013. URL http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html.

[NVIDIA, 2013-07b] NVIDIA. Kepler Tuning Guide. Website, July 19, 2013. URL http://docs.nvidia.com/cuda/kepler-tuning-guide/index.html.

[NVIDIA, 2014a] NVIDIA. Directcompute NVIDIA developer zone. Website, 2014. URL https://developer.nvidia.com/directcompute.

[NVIDIA, 2014b] NVIDIA. NVIDIA CUDA zone. Website, 2014. URL https://developer.nvidia.com/cuda-zone.

[NVIDIA, 2014c] NVIDIA. NVIDIA What is CUDA?. Website, 2014. URL https://http://www.nvidia.com/object/cuda_home_new.html.

[NVIDIA, 2014d] NVIDIA. Profiler User's Guide. Website, 2014. URL http://docs.nvidia.com/cuda/profiler-users-guide/.

[NVIDIA, 2014e] NVIDIA. GPU-accelerated libraries. Website, 2014. URL https://developer.nvidia.com/gpu-accelerated-libraries.

[NVIDIA, 2014f] NVIDIA. CUDA Basic Linear Algebra Subroutines (cuBLAS) library. Website, 2014. URL https://developer.nvidia.com/cublas.

[NVIDIA, 2014g] NVIDIA. A powerful library of parallel algorithms and data structures. Website, 2014. URL https://developer.nvidia.com/thrust.

[OpenCL, 2014] Khronos Group. The open standard for parallel programming of heterogeneous systems. Website, 2014. URL https://www.khronos.org/opencl/.

[Pevsner, 2009] Jonathan Pevsner. *Bioinformatics and Functional Genomics*. John Wiley & Sons, New Jersey, USA, 2nd edition, 2009.

[Pfam, 2013] Wellcome Trust Sanger Institute and Howard Hughes Janelia Farm Research Campus. Pfam database. Website, March 2013. URL ftp://ftp.sanger.ac.uk/pub/databases/Pfam/releases/Pfam27.0/Pfam-A.hmm.gz.

[Quirem et al., 2011] Saddam Quirem, Fahian Ahmed, and Byeong Kil Lee *CUDA Acceleration of P7Viterbi Algorithm in HMMER 3.0*. 30th IEEE Internationa Performance Computing and Communications Conference (IPCCC), 2011.

[Rognes and Seeberg, 2000] Rognes T and Seeberg E. *Six-fold speed-up of Smith-Waterman sequence database searches using parallel processing on common microprocessors*. Bioinformatics 16: 699-706 2000.

[Rognes, 2011] Rognes T. *Faster Smith-Waterman database searches with inter-sequence SIMD parallelization*. BMC Bioinformatics 2011, 12:221.

[Saeed et al., 2010] Ali Khajeh-Saeed, Stephen Poole and J. Blair Perot *Acceleration of the Smith-Waterman algorithm using single and multiple graphics processors.* Journal of Computational Physics 229 (2010) 4247-4258.

[Sanders, 2011] Jason Sanders and Edward Kandrot. *CUDA by example : an introduction to general-purpose GPU programming.* NVIDIA Corporation 2011.

[Smith and Waterman, 1981] Smith TF. and Waterman MS. *Identification of common molecular subsequences.* J Mol Biol 1981, 147:195-197.

[UniProtSP, 2014-05] SIB Bioinformatics Resource Portal. UniProtKB/Swiss-Prot protein knowledgebase release 2014-05 statistics. Website, 2014-05. URL http://web.expasy.org/docs/relnotes/relstat.html.

[UniProtTr, 2014-05] EMBL-EBI, Wellcome Trust Genome Campus. UniProtKB/TrEMBL PROTEIN DATABASE RELEASE 2014_05 STATISTICS. Website, 2014-05. URL http://www.ebi.ac.uk/uniprot/TrEMBLstats.

[UniProt, 2013-09] Universal Protein Resource. UniProt Release. Website, 2013-09. URL ftp://ftp.uniprot.org/pub/databases/uniprot/previous_releases/release-2013_09/.

[UniProt, 2014] Universal Protein Resource. About UniProt. Website, 2014. URL http://www.uniprot.org/help/about.

[Walters et al., 2006] J. P. Walters, B. Qudah, and V. Chaudhary *Accelerating the HMMER Sequence Analysis Suite Using Conventional Processors.* In AINA'6: Proceedings of the 20th International Conference on Advanced Information Networking and Applications, pages 289-294, Washington, DC, USA, 2006. IEEE Computer Society.

[Walters et al., 2009] J. P. Walters, V. Balu, S. Kompalli, and V. Chaudhary *Evaluating the use of GPUs in Liver Image Segmentation and HMMER Database Searches.* International Parallel and Distributed Processing Symposium (IPDPS), 2009.

[Waters, 2011] Blue Waters. *Taking CUDA to Ludicrous Speed.* Undergraduate Petascale Education Program, Website, June 2011. URL http://iclcs.uiuc.edu/index.php/iclcs-news/51-blue-waters-undergraduate-petascale-internship-program.html.

[Wiki Cell, 2014] Wikipedia. Cell (biology). Website, 2014. URL http://en.wikipedia.org/wiki/Cell_(biology).

[Wiki Sequence, 2014] Wikipedia. Sequence Alignment. Website, 2014. URL http://en.wikipedia.org/wiki/Sequence_alignment.

[Wilt, 2013] Nicholas Wilt. *The CUDA Handbook : A Comprehensive Guide to GPU Programming*. Pearson Education, Inc. 2013.

[Wun et al., 2005] Wun, B., Buhler, J., and Crowley, P. *Exploiting coarsegrained parallelism to accelerate protein motif finding with a network processor*. In PACT '05: Proceedings of the 2005 International Conference on Parallel Architectures and Compilation Techniques, 2005.

[Zeller, 2008] Cyril Zeller. *Tutorial CUDA*. NVIDIA Developer Technology, 2008.