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

Lin Cheng

A thesis

in

The Department

of

Engineering and Computer Science

Presented in Partial Fulfilment of the Requirements

For the Degree of Master of Computer Science

Concordia University

Montréal, Québec, Canada

May 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
(Engineering and Computer Science)

CONCORDIA UNIVERSITY
Engineering and Computer Science

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. However, the manycore architecture of the GPU creates challenges to HMMER implementations. And multiple spaces of memory exposed by CUDA must be carefully managed.

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. 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.

Further, memory lay- out and load balancing are critical performance barriers that must be overcome to maximally utilize the GPU. We examine these issues and demonstrate a variety of optimization strategies that are useful for different classes of GPU-based applications. We make the following contri- butions: • Implement a GPU-based implementation of HM- MER's hmmsearch tool. • Discuss and analyze the advantages and limitations of GPU hardware for general purpose HPC.

our hmmsearch implementation achieves up to 38x speedup.

Acknowledgements

The acknowledgements and the people to thank go here, don't forget to include your project advisor...

Contents

List of Figures

List of Tables

Chapter 1

Background

1.1 Sequence alignment and protein database

1.1.1 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*. The 20 amino acids detailed in Table1.1 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 1.1: The 20 amino acids

1.1.2 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[1] 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*. To establish the degree of homology, the sequences are aligned: lined up in such a way that the degree of similarity is maximized.

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.

1.1.3 Bioinformatics databases

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

[1] In this thesis, *residue* is used to refer to amino acids for protein.

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 available online[2].

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[3], PDB[4]. Entries with absolutely identical sequences have been merged into NR database.

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 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.
- UniProtKB/TrEMBL
  This section contains computationally analyzed records that await full manual annotation.

1.2 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

[2] http://www.ncbi.nlm.nih.gov/guide/all/#databases_

[3] The Protein Information Resource (PIR) produces the largest, most comprehensive, annotated protein sequence database in the public domain.

[4] The Protein Data Bank, maintained by Brookhaven National Laboratory (Long Island, New York, USA)

HMMER use DP for sequence alignment and database searches, and then discusses the related work about acceleration on CUDA-enabled GPU.

1.2.1 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.

1.2.2 HMMER

HMMER [HMMER, 2014] 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 for a profile HMM in a sequence database, while the latter searches for one or more sequences in profile HMMs database.

1.2.2.1 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 or gen family can be defined by profile HMMs.

In Figure1.1, 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 *insertion* state and one *deletion* state.

- ***Match states*** match and emit a amino acid from the query. The probability of emitting each of the 20 amino acids is a property of the model.
- ***Insertion states*** allows the insertion 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.
- ***Deletion states*** skip the alignment column and emit a blank. Entering this state corresponds to gap opening, and the probabilities of these transitions reflect a position-specific gap penalty.

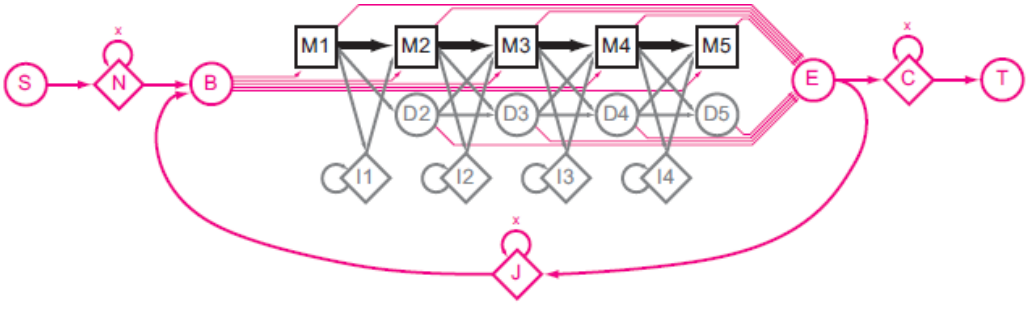

FIGURE 1.1: Profile HMM architecture used by HMMER[Eddy, 2011].

Begin at Start(S), and follow 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), insertion (I) and deletion (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 query 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.

1.2.2.2 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 query. $V_{i+1,j}$ 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 *insertion* state I_j, and $V_j^D(i)$ for the optimal path ending in *deletion* state D_j. q_{x_i} is the probability of x_i. Then we can write the Viterbi general equation[Durbin et al., 1998]:

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

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

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

The efficient DP-based pseudo code of Viterbi algorithm is shown in Algorithm1.2.1 [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 1.2.1: 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** n

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

comment: Termination:

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

1.2.2.3 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.

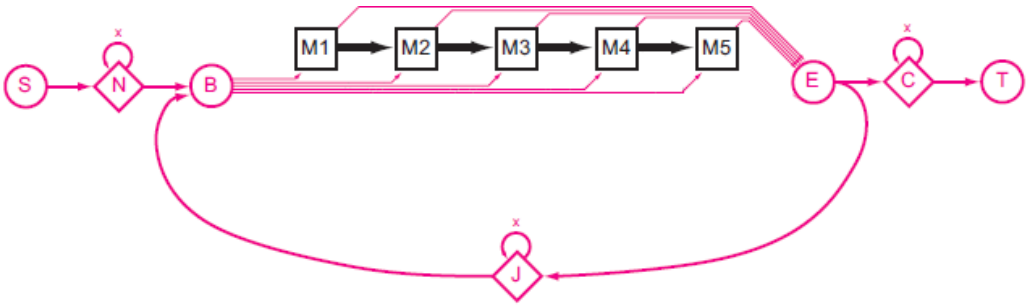

FIGURE 1.2: MSV profile: multiple ungapped local alignment segments[Eddy, 2011].

Figure1.2 illustrates the MSV profile architecture. Compared with Figure1.1, the MSV corresponds to the virtual removal of the deletion and insertion 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 Algorithm1.2.2.

Algorithm 1.2.2: 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** n

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

comment: Termination:

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

SIMD vector parallelization in HMMER3

Single-Instruction Multiple-Data (SIMD) instruction is able to perform the same operation on multiple pieces of data in parallel. The SIMD vector instruction sets use 128-bit vector registers to compute up to 16 simultaneous operations. Several SIMD vector parallelization methods have been described for accelerating Smith-waterman dynamic programming. Rognes and Seeberg [Rognes and Seeberg, 2000] presented an implementation of the Smith–Waterman algorithm running on the Intel Pentium processor using the MMX SIMD instructions. A six-fold speedup was reported over an optimized non-SIMD implementation. Farrar [Farrar, 2007] presented an efficient vector-parallel approach called striped Smith-Waterman using Intel Streaming SIMD Extensions 2 (SSE2).

A speedup of 2–8 times was reported over the Rognes and Seeberg SIMD implementations.

Similarly, since the MSV model removes deletion and insertion states that interfere with vector parallelism, the striped vector-parallel technique can be remarkably efficient way to calculate MSV alignment scores. To maximize parallelism, in HMMER3[Eddy, 2011], Sean R. Eddy 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.

Figure1.3 shows the stripped pattern. The query profile HMM of length m is divided into vectors with equal length v. The vector length v is equal to the number of elements being processed in the 128-bit SIMD register. MSV processes 8-bit byte integer with $v = 128/8 = 16$. In a row-vectorized implementation, the query profile HMM is stored in the vectorized dynamic programming matrix dp. dp stores m cells in Q vectors which is numbered as $q = 1...Q$, where $Q = (m + v - 1)/v$. Figure1.3 illustrates m cells assigned to Q vectors in a non-sequential way. For simple illustration, $v = 4, m = 14$, and $Q = 4$.

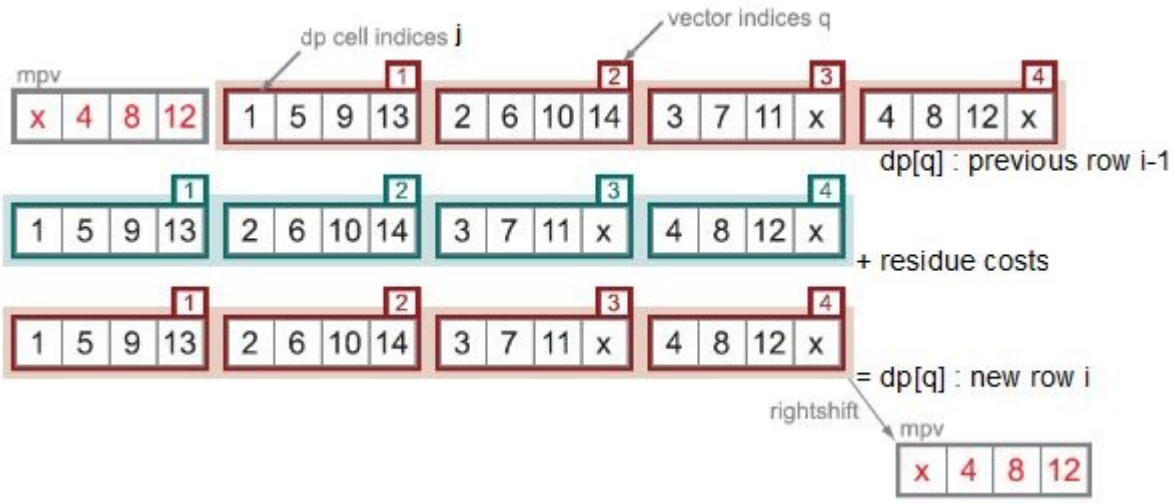

FIGURE 1.3: Illustration of striped indexing for SIMD vector calculations[Eddy, 2011].

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 deletion and insertion states have been removed and only ungapped diagonals need calculating, so the calculation of each cell (i, j) requires only previous $(i-1, j-1)$. In Figure1.3, 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 Figure1.3, 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).

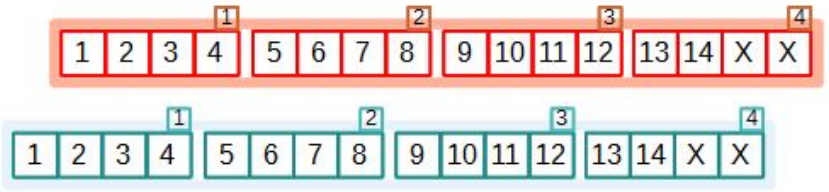

FIGURE 1.4: Illustration of linear indexing for SIMD vector calculations.

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 Figure1.4, 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 1.3. Outside the inner loop(for $q = 1$ to 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 Algorithm1.2.3

Algorithm 1.2.3: MSV-SIMD()

comment: Initialization

$xJ \leftarrow 0; \quad dp[q] \leftarrow vec\_splat(0)\ (q \leftarrow 0\ \textbf{to}\ Q-1)$

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

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

comment: for every sequence residue i

$$\textbf{for}\ i \leftarrow 1\ \textbf{to}\ n$$

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

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)[21]	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) -[22]	horizontal max	$x = max(v[z]), z = 0...15$

TABLE 1.2: SSE2 intrinsics for pseudocode in Algorithm1.2.3

1.3 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.

1.3.1 Overview of CUDA programming model

1.3.1.1 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]:

[21] Because x86 and x86-64 use little endian, this means using a left bit shift intrinsic _mm_slli_si128 to do right shift.

[22] No SSE2 intrinsic is corresponding to vec_hmax. Shuffle intrinsic _mm_shuffle_epi32 and _mm_max_epu8 can be combined to implement vec_hmax.

- 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. Table1.3 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 1.3: Features per Compute Capability

1.3.1.2 CUDA thread hierarchy

The execution of a typical CUDA program is illustrated in Figure1.5 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*. Figure1.5 shows the execution of two grids of threads.

Threads in a grid are organized into a two-level hierarchy, as illustrated in Figure1.6. At the top level, each grid consists of one or more thread blocks. All blocks in a grid have the same number of threads and are organized into a one, two, or three-dimensional *grid* of thread blocks.

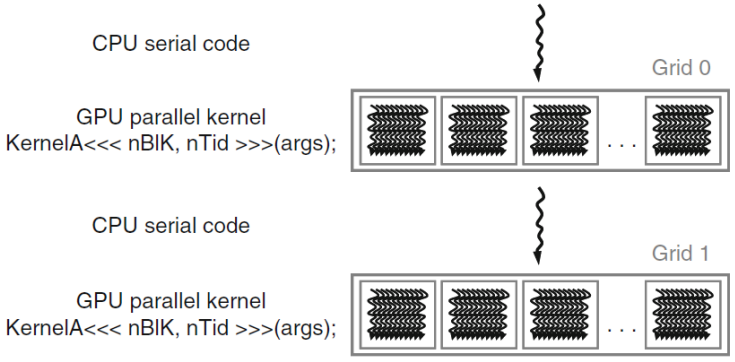

FIGURE 1.5: Execution of a CUDA program[Kirk and Hwu, 2010].

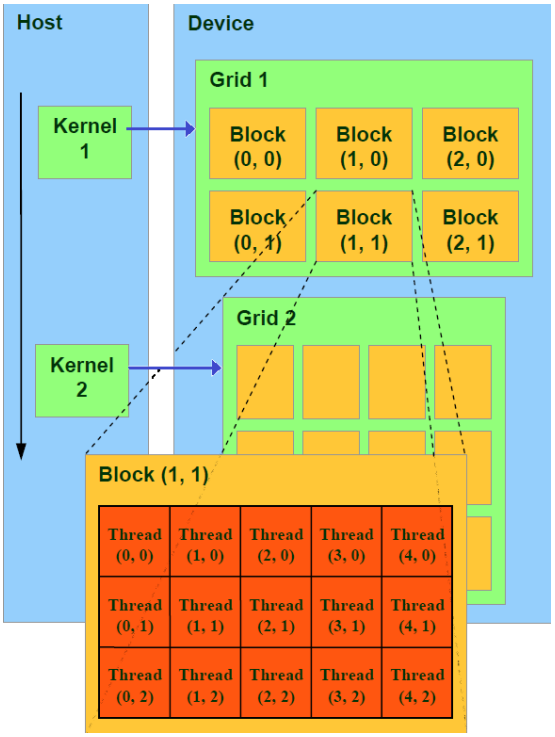

FIGURE 1.6: CUDA thread organization[Zeller, 2008].

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.

1.3.1.3 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 Figure1.7

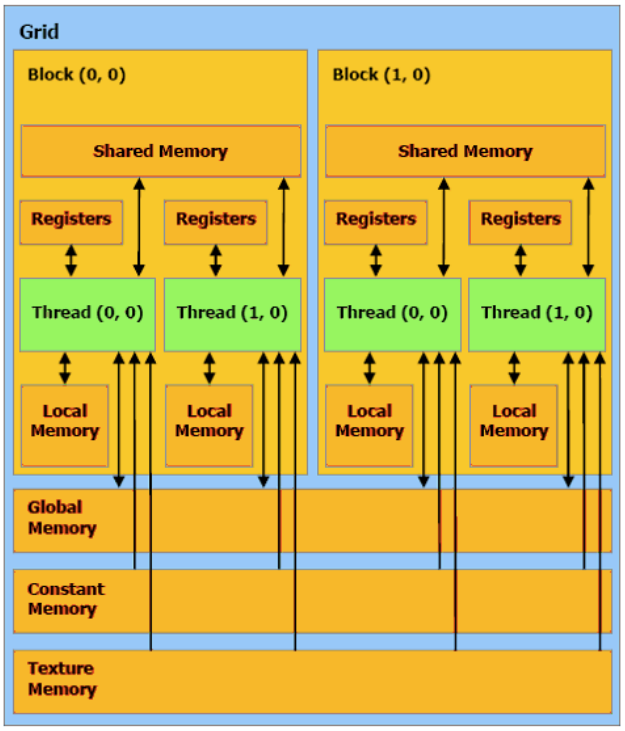

FIGURE 1.7: CUDA memory organization[Zeller, 2008].

CUDA memory spaces have different characteristics that reflect their distinct usages in CUDA applications as summarized in Table1.4 [CUDA Best, 2013].

Memory	Location	Cached	Access	Scope	Speed [31]	Lifetime
Register	On chip	n/a	R/W	1 Thread	1	Thread
Local	Off chip	†[32]	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 1.4: Salient Features of GPU Device Memory

[31] Relative Speed in number of instructions
[32] Cached only on devices of compute capability 2.x

1.3.2 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 2.

Parallelism strategy applied

[Manavski, 2008], [Liu et al., 2009], [Akoglu and Striemer, 2009], [Ligowski and Rudnicki, 2009], [Liu et al., 2010], [Kentie, 2010], [Liu et al., 2013] used task-based parallelism to process each target sequence independently with a single GPU thread.

[Liu et al., 2009] used data-based parallelism to support longest query/target sequences.

[Saeed et al., 2010] formulated parallel version of the Smith-Waterman algorithm and used MPI over 100 Mb Ethernet to extend work to multiple GPUs.

Processing target sequences database

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

[Ligowski and Rudnicki, 2009] presorted database in descending order and organized in blocks consisting of 256 sequences.[Kentie, 2010] presorted database also in descending order and converted into a special format.

Device memory access pattern

[Liu et al., 2009] sorted target sequences and arranged 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 were first batched in shared memory for better coalescing. Due to a reduction in the global memory accesses, they proposed a cell block division method for the task-based parallelization, where the alignment matrix is divided into cell blocks of equal size.

[Kentie, 2010] reduced global memory access by making temporary values interleaved and read/wrote score and Ix temporary values in one access.

[Liu et al., 2009] exploited constant memory to store the gap penalties, scoring matrix and the query sequence. And [Kentie, 2010] stored gap penalties in constant memory. [Akoglu and Striemer, 2009] mapped query sequence as well as the substitution matrix to the constant memory.

[Manavski, 2008], [Liu et al., 2010] used texture memory to store query profiles. And [Kentie, 2010] used texture for substitution matrix.

[Liu et al., 2009] loaded the scoring matrix into shared memory.

[Ligowski and Rudnicki, 2009] reduce 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

[Manavski, 2008] packaged the query profile in texture memory, storing 4 successive values into the 4-byte of a single unsigned integer. And they read at a time 4 H and 4 E values from local memory.

[Akoglu and Striemer, 2009] calculated the Smith-Waterman score from the query sequence and database sequences by means of columns, four cells at a time.

[Liu et al., 2010] designed a striped query profile for SIMD vector computation and used a packed data format to store into the CUDA built-in *uchar4* vector data type. They divided a query sequence into a series of non-overlapping, consecutive small partitions with a specified partition length, and then aligned the query sequence to a subject sequence partition by partition. They ported the SIMD CPU algorithm [Farrar, 2007] to the GPU, viewing collections of processing elements as part of a single vector.

[Kentie, 2010] loaded 4 query characters at a time and processed 8 database characters at a time.

[Liu et al., 2013] designed a query profile variant data structure and used the built-in *uint4* vector data type to store each sequence profile for quad-lane SIMD computing on GPUs. They used CUDA SIMD Video Instructions in GPU computing and used Intel SSE2 intrinsics in CPU computing.

Miscellaneous techniques

[Manavski, 2008] pre-computed a query profile parallel to the query sequence for each possible residue and achieved dynamic load balancing between multiple GPUs according to their computational power at run time.

[Kentie, 2010] simplified substitution matrix lookup by using numeric values instead of letters for sequence symbols.

[Liu et al., 2013] distributed workload between CPU and GPU.

1.3.3 CUDA accelerated HMMER

HMMER includes a MPI (Message Passing Interface) implementation of the searching algorithms, which uses conventional CPU clusters for parallel computing. Since [Horn et al., 2005] [5], the first GPU-enabled *hmmsearch* implementation, 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

[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.

[Du et al., 2010] used data-based parallelism by dividing the basic computational kernel of the Viterbi algorithm of each tile into two parts, independent and dependent parts.

[Ganesan et al., 2010] presented a hybrid parallelization strategy by combining task-based and data-based parallelism. They parallelized evaluations of recurrence equations by partitioning the chain of dependencies in a uniform and regular fashion.

HMM-based algorithm researched

[Walters et al., 2009], [Ganesan et al., 2010], [Du et al., 2010] and [Quirem et al., 2011] parallelized Viterbi algorithm.

[Ahmed et al., 2012] used Intel VTune Analyzer [6] to investigate performance hotspot functions in HMMER3. Based on hotspot analysis, they studied CUDA acceleration for

[5] ClawHMMer is based on the BrookGPU stream programming language, not CUDA programming model.

[6] https://software.intel.com/en-us/articles/analyzing-and-optimizing-performance-of-windows-store-apps-using-intel-vtune-analyzer-and

three individual algorithm: Forward, Backward and Viterbi algorithms. And they also found data transfer overhead between heterogeneous processors could be a performance bottleneck.

However, our research focus on the MSV algorithm in HMMER3.

Device memory access pattern

[Walters et al., 2009] made use of high speed texture memory to store both the target sequence batch as well as the query profile HMM. Their benchmark result showed that global memory coalescing contributed an improvement of more than 9x for larger HMMs. They also used constant memory to store the HMM and used shared memory to temporarily store the index into each threads digitized sequence.

[Ganesan et al., 2010] adopted the partitioning scheme to coalesce memory access by storing lookup data of model positions at regular intervals contiguously.

[Du et al., 2010] reorganized the computational kernel of the Viterbi algorithm, and divided 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.

[Quirem et al., 2011] utilized pinned memory to reduce the latency induced by transferring memory from device to host and back.

Miscellaneous techniques

- [Walters et al., 2009] created two CPU threads for reading database and post-processing the database hits.
- [Walters et al., 2009] presorted target sequences database in ascending order.
- [Walters et al., 2009] applied loop unrolling which is a classic loop optimization strategy designed to reduce the overhead of inefficient looping.

Chapter 2

A CUDA accelerated HMMER3 protein sequence search tool

2.1 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 [CUDA zone, 2014] 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 [CUDA What, 2014]. 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 *dimBlock* tasks are performed in parallel by different threads in a thread block.
- **Data-based parallelism** Each task is assigned to one thread block and all *dimBlock* threads in the thread block 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 subsection2.3.8 on workload distribution.

2.2 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.

2.2.1 CPU serial version of hmmsearch

The CPU serial version of hmmsearch in HMMER3 is shown in Figure2.1. The MSV and Viterbi algorithms described in subsection 1.2.2.2 and 1.2.2.3 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.

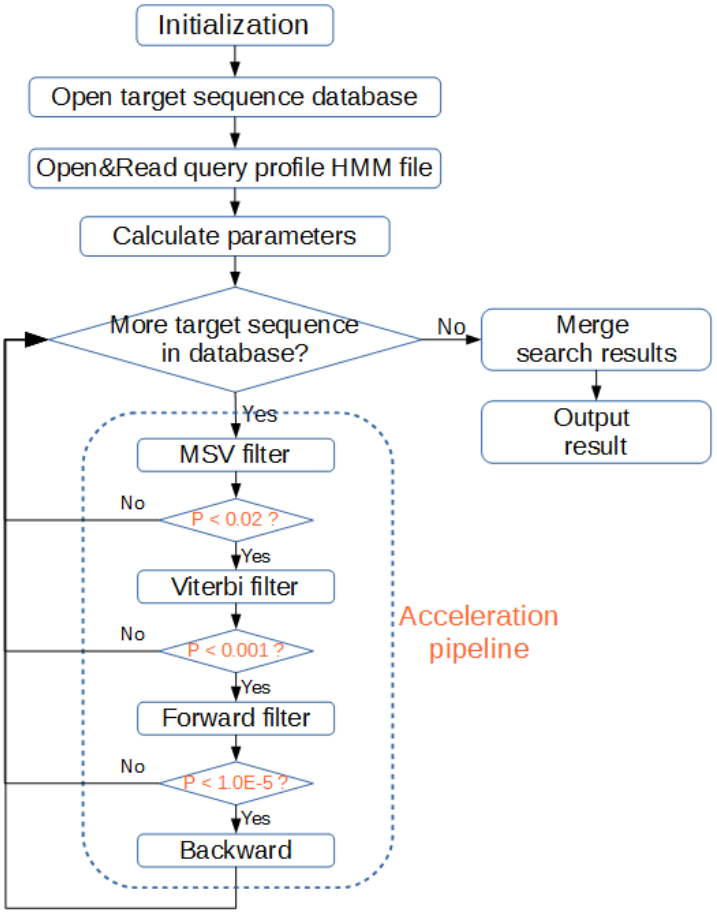

FIGURE 2.1: The CPU serial version of hmmsearch

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

Figure2.1, by default a target sequence can pass the MSV filter if its comparison gets a P-value of less than 0.02, which means the top-scoring 2% of target sequences are expected to pass the filter. So, much fewer target sequence can pass one filter and need further computing. By this way, 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 Figure1.2.3 is as efficient as possible.

2.2.2 GPU implementation of MSV filter

A basic flow of the GPU implementation for MSV filter is shown in Figure2.2. 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 1.2.3. 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 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 each thread finishes its sequences.

2.3 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 a test database with 540,958 query sequences.

This section discusses the optimization steps taken to eventually reach a benchmark database search time of 1.55 seconds: an almost 147 times speedup.

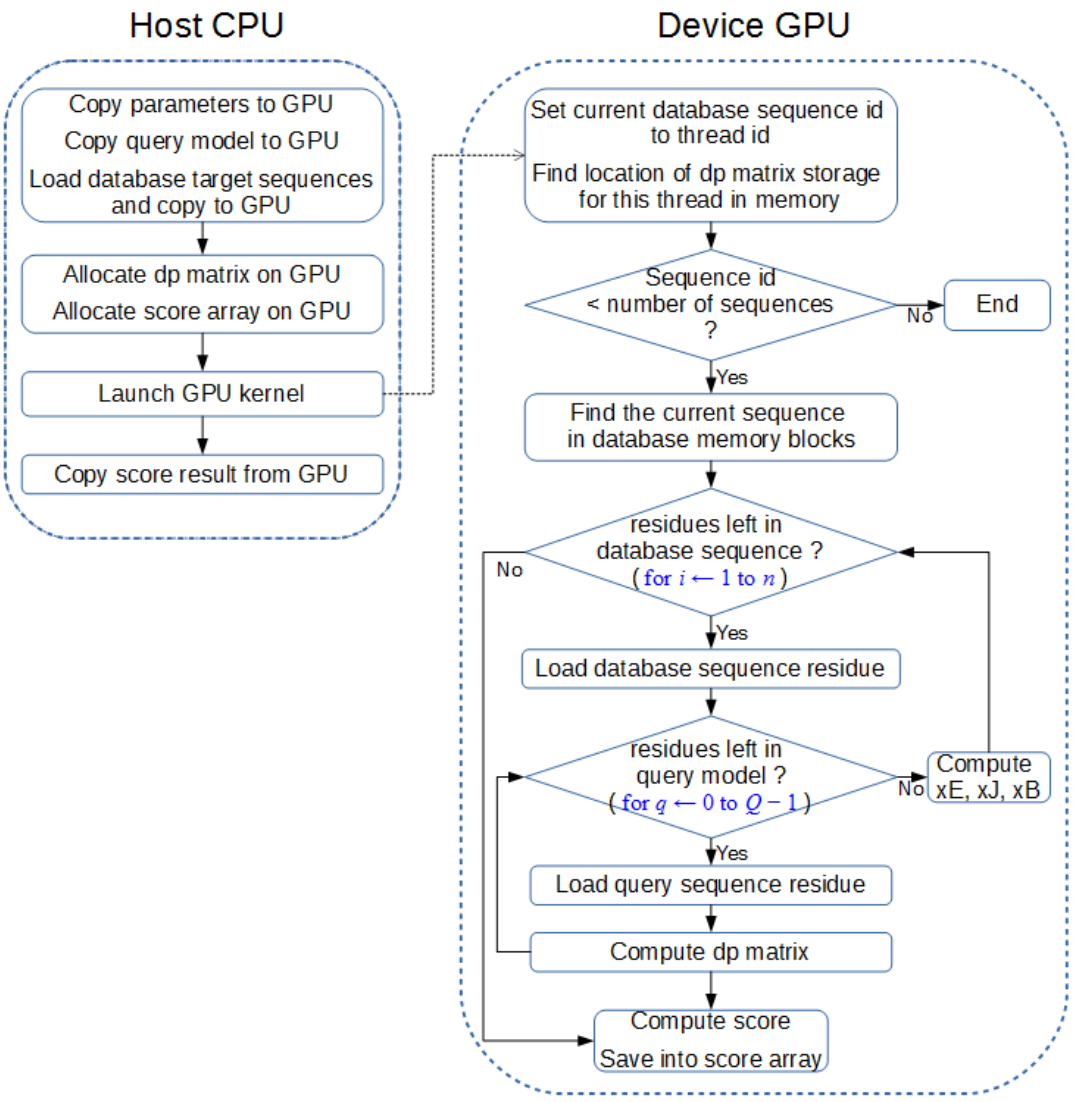

FIGURE 2.2: The GPU porting of MSV filter

2.3.1 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 2D 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.

LISTING 2.1: struct of uint4 and ulong2

```
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 Figure2.3, 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.

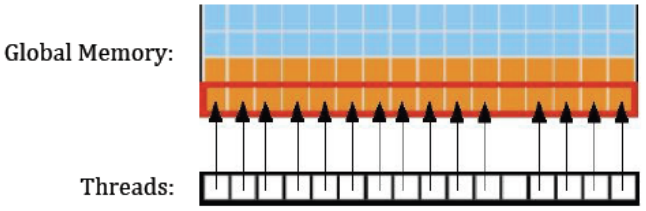

FIGURE 2.3: Coalescing Global Memory Accesses[Waters, 2011].

For coalescing access, the target sequences are arranged in an array like an upside-down bookcase shown in Figure2.4, 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 section x.

Figure2.5 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.

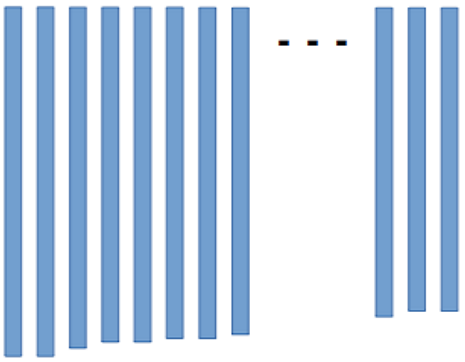

FIGURE 2.4: Alignment of target sequences.

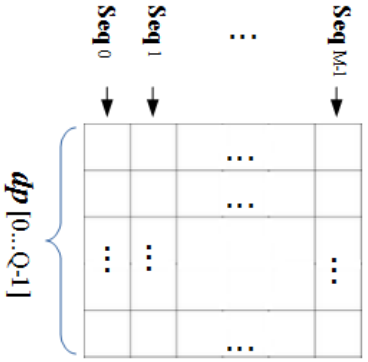

FIGURE 2.5: dp matrix in global memory.

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* [CUDA C, 2013].

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

Note on coding global memory coalescing access

At the beginning, the traditional C/C++ memory block copy function *memcpy*() was used, since uint4 is 16-bytes block. *dp* is the pointer to the address of global memory. *mpv* and *sv* are *uint4* data 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 and writes respectively. This doesn't fully coalescing access global memory. Switching to the following direct assignment resulted in 81% improvement.

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

2.3.2 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 [CUDA C, 2013].

Texture memory is well suited to random access. CUDA has optimized the operation fetching four values (RGB colors and alpha component, a typical graphics usage) at a time in texture memory. And this mechanism is also applied to fetch four values from the query profile 2D matrix 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, *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 [CUDA Kepler, 2013].

2.3.3 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 Figure2.6(a). In this way, four residues are loaded using only one texture fetch, thus significantly improving texture memory throughput.

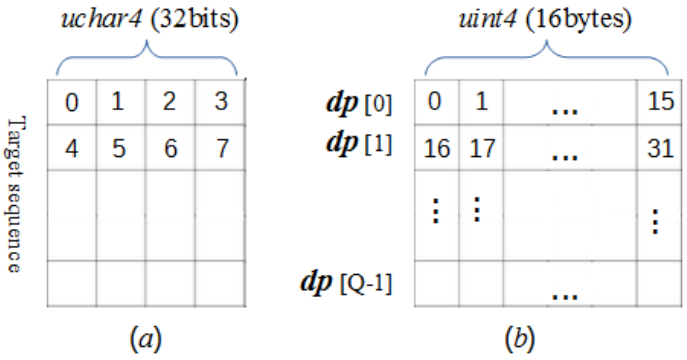

FIGURE 2.6: SIMD vector alignment of data: (a) target sequence; (b) dp array.

Similarly, the dp array and the query profile also use the virtualized SIMD vector allocation pattern, as can be seen in Figure2.6(b).

2.3.4 SIMD Video Instructions

Like Interl SSE2 described in subsection1.2.2.3, 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.

2.3.5 Pinned (non-pageable) Memory

It's 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)

2.3.6 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[1] concurrency. Within a given stream, operations are performed in sequential order, but operations in different streams may be performed in parallel [CUDAintro, 2011].

To take advantage of CPU/GPU concurrency as depicted in Figure2.7, 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.

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 Figure2.8 in which empty boxes represent time when

[1]For GPUs that have one or more copy engines, host $\longleftrightarrow$ device memory copy can be performed while the SMs are processing kernels.

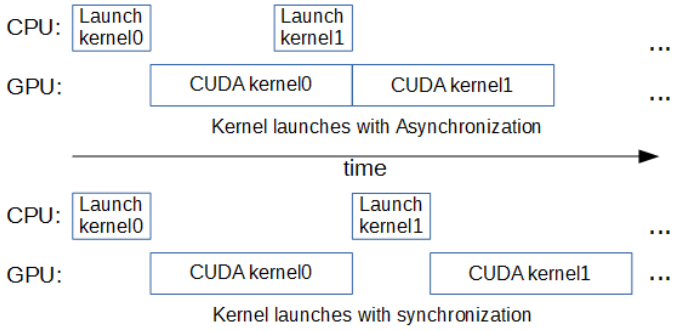

FIGURE 2.7: CPU/GPU concurrency.

one stream is waiting to execute an operation that it cannot overlap with the other stream's operation.

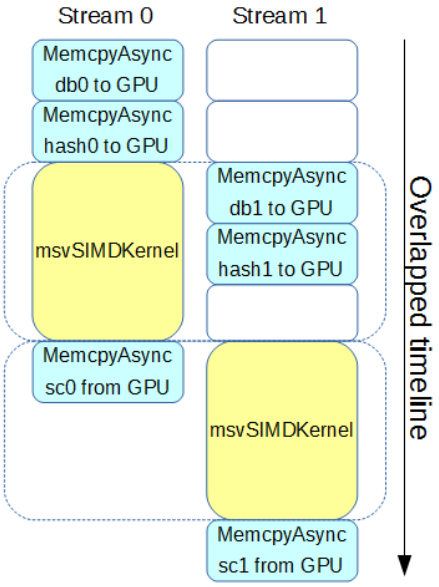

FIGURE 2.8: Timeline of intended application execution using two independent streams.

LISTING 2.2: Combined asynchronous memory copy and multiple streams

```
// enqueue copies of dbQuad in stream0 and stream1
cudaMemcpyAsync(cudaPtr.dbQuad0, hostPtr.dbQuad0,
                num0 * sizeof(uint),
                cudaMemcpyHostToDevice,
                stream[0]);
cudaMemcpyAsync(cudaPtr.dbQuad1, hostPtr.dbQuad1,
                num2 * sizeof(uint),
                cudaMemcpyHostToDevice,
                stream[1]);

// enqueue copies of hashQuad in stream0 and stream1
cudaMemcpyAsync(cudaPtr.hashQuad0, hostPtr.hashQuad0, num0 * sizeof(uint), ↩
    cudaMemcpyHostToDevice, stream[0]);
cudaMemcpyAsync(cudaPtr.hashQuad1, hostPtr.hashQuad1, num1 * sizeof(uint), ↩
    cudaMemcpyHostToDevice, stream[1]);
```

```
// enqueue kernels in stream0 and stream1
msvSIMDKernel0<<<numBlocks, KERNEL_BLOCKSIZE, 0, stream[0]>>>();
msvSIMDKernel0<<<numBlocks, KERNEL_BLOCKSIZE, 0, stream[1]>>>();

// enqueue copies of result from device to locked memory
cudaMemcpyAsync(hostSC0, cudaSC0, num0 * sizeof(float), cudaMemcpyDeviceToHost, ↩
    stream[0]);
cudaMemcpyAsync(hostSC1, cudaSC1, num1 * sizeof(float), cudaMemcpyDeviceToHost, ↩
    stream[1]);
```

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.

2.3.7 Sorting database

As described in Section1.2.2.3, MSV filter function is sensitive to the length of a target sequence, which determines the execution times of main *for* loop in Algorithm1.2.3.

Target sequence database could contain many sequences with different lengths. The 24GB NCBI NR database consists of over 38 million sequences with sequence lengths varying from 6–37,000+ 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 weren't 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 [Profiler, 2014] 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 table2.1 and 2.2 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 2.1: profiling result of before sorting database

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 2.2: profiling result of after sorting database

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 Table3.1 and Figure3.1). 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.

2.3.8 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. Considering the length distribution of database sequences[2] and based on the descending sorted database discussed in section2.3.7, 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.

[2] According to Swiss-Prot database statistics [Swiss-Prot, 2014-04], the percentage of sequences with length > 2500 is only 0.2%.

2.4 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 doesn't 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 subsection2.1, 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 doesn't use shared memory.

Kernel launch configuration

Since the MSV implementation doesn't 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);
```

2.5 Summary of optimization steps taken

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 Section2.2.1], which type of parallelism [Task-based parallelism in Section2.1] is better suited to the application.

2. Application Profiling

   NVIDIA® provides profiling tools to help execute the kernels in question under the watchful gaze [nvprof in Section2.3.7], which are publicly available as a separate download on the CUDA Zone website [CUDA zone, 2014].

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 2.3: CUDA profiling tools

3. Optimizing memory usage

   Optimizing memory usage starts with minimizing data transfers both in size [Database sorted in Section2.3.7, workload distribution in Section2.3.8] and time [Pinned Memory in Section2.3.5] between the host and the device [Asynchronous memory copy in Section2.3.6]. Be careful with CUDA memory hierarchy: register memory [Section2.4], local memory, shared memory [Section2.4], global memory, constant memory [Section2.4] and texture memory [Section2.3.2], and combine these memories to best suit the application [Kernel launch configuration in Section2.4]. 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 [Section2.3.1].

4. Optimizing instruction usage

   This suggests using SIMD Video Instructions [Section2.3.4] 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 Section1.2.2.3]. Particular attention should be paid to control flow instructions [Branch divergence in Section2.4].

5. Maximizing parallel execution

   The application should maximize parallel execution at a higher level by explicitly exposing concurrent execution on the device through streams [Section2.3.6], as well as maximizing concurrent execution between the CPU host [Database sorted

in Section2.3.7] and the GPU device [Workload distribution in Section2.3.8 and SIMD Video Instructions in Section2.3.4].

6. Considering the existing libraries

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

Chapter 3

Benchmark results and discussion

Chapter2 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.

3.1 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 Capability version 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 fasta release September 2013 [Swiss-Prot, 2013-09] with 540,958 sequences and another was much larger NCBI NR fasta release April 2014 [NCBI, 2014] with 38,442,706 sequences.
- Query profile HMMs
  A profile HMM named globin4 with length of 149 states was distributed with the HMMER source [HMMER source, 2014]. 4 HMMs were taken directly from the Pfam database [Pfam, 2013] that vary in length from 255 to 1111 states.

- 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.

3.2 Performance Results

3.2.1 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.

Table3.1 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[51]	6.565	4.362	44
Texture memory[52]	5.370	5.333	22
Sorting Database	2.346	12.207	129
Distributing workload	1.650	17.357	42

TABLE 3.1: Performance of optimization approaches

The graphic view corresponding to Table is shown in Figure3.1.

[51] benchmarked only for *dp* matrix

[52] benchmarked only for the query profile texOMrbv 2D texture

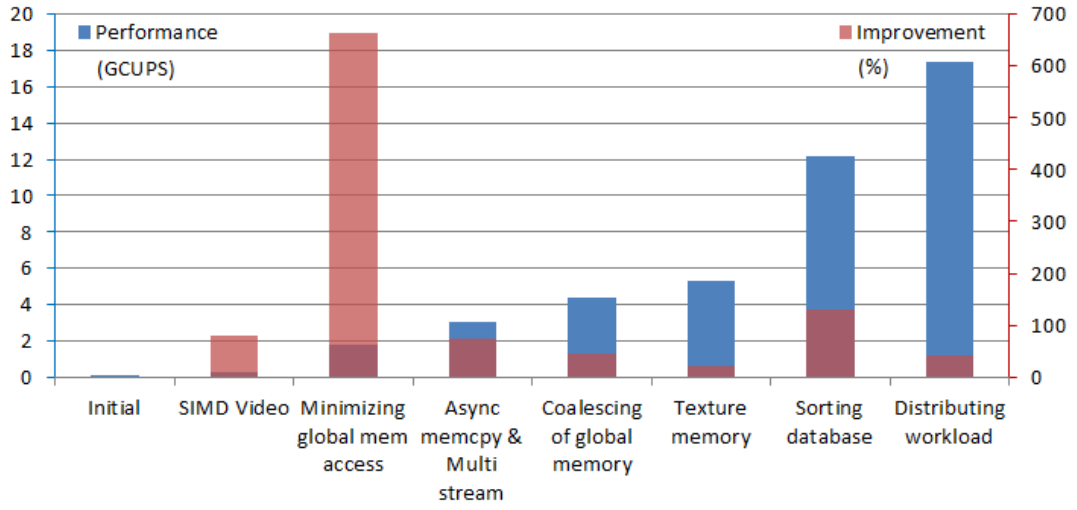

FIGURE 3.1: Performance of optimization approaches.

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.

3.2.2 Practical benchmarks

The final cudaHmmsearch implementation, with the optimization discussed in Chapter2, was benchmarked to determine its real-world performance. This was done by searching the much large NCBI NR database for several profile HMMs with various lengths, which were selected from Pfam database. As comparison, the same searches were executed by hmmsearch of HMMER3 on 1 CPU core.

The results of the benchmarks are shown in graphical form in Figure3.2. 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.

Figure3.2 shows that the performance of both GPU and CPU searching for AAA_27 dropped greatly. The reason can be seen from the table3.2 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.

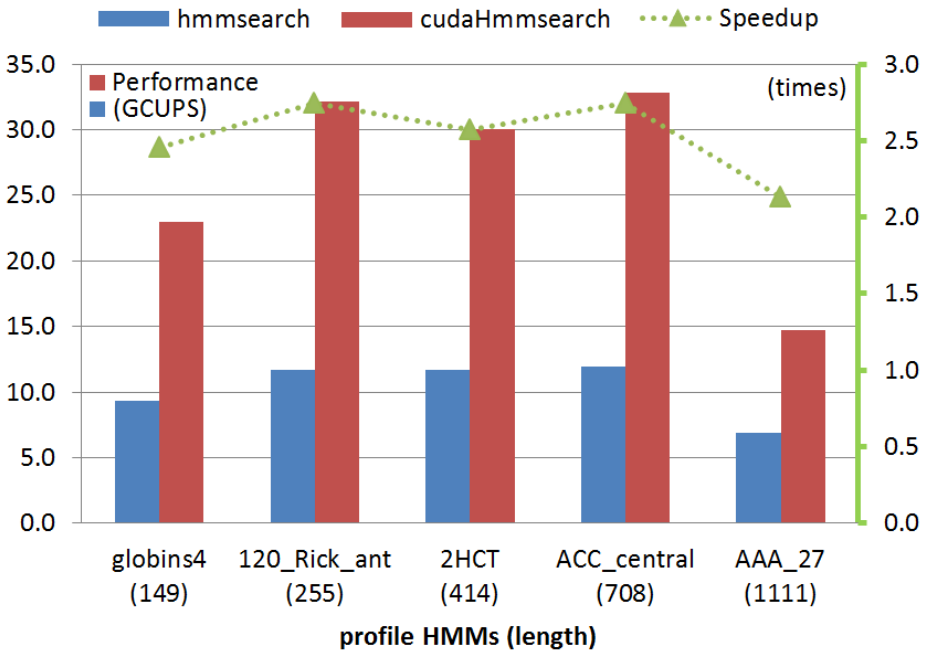

FIGURE 3.2: Practical benchmarks.

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 3.2: Internal pipeline statistics summary

3.2.3 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 3.1 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.

Figure3.3 shows the benchmark result. The number performance results is measured in GCUPS. As can be seen, from 1 CPU core to 4 CPU cores, both cudaHmmsearch

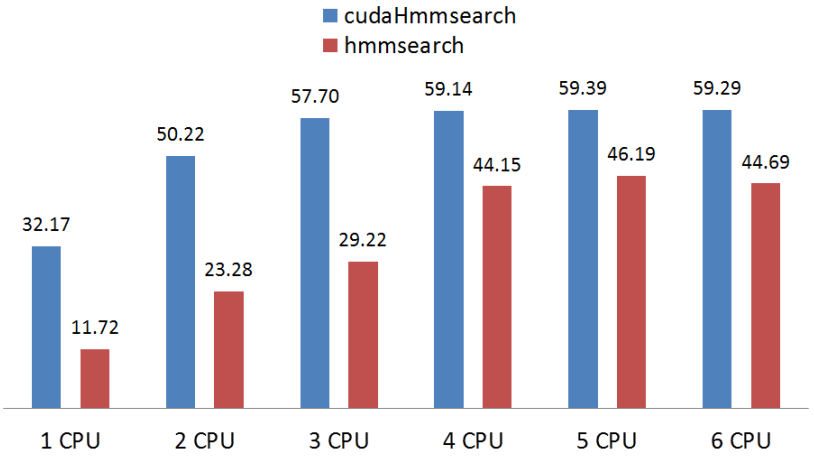

FIGURE 3.3: Comparison with multicore CPU.

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.

3.2.4 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 source, 2014].

All tests are taken searching against the Swiss-Prot database for the globin4 profile HMM.

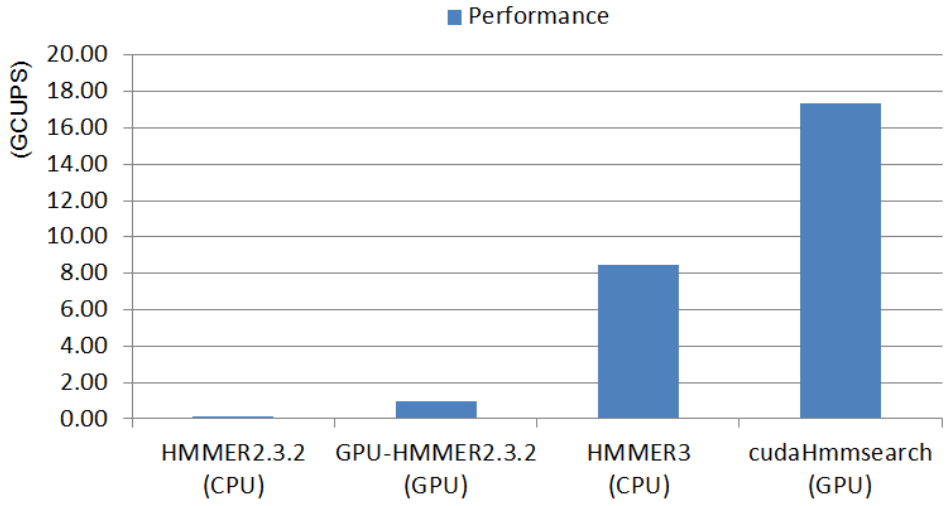

FIGURE 3.4: Comparison with other implementations.

As seen from the Figure3.4, since the release of HMMER2.3.2 in Oct 2003, accelerating hmmsearch researches on both CPU and GPU have achieved excellent improvement.

Chapter 4

Conclusions and recommendations

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.

4.1 Conclusions of current work

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.

At the beginning of Section 1.3, we specially summarized the technologies of accelerating Swith-Waterman algorithm on CUDA-enabled GPU, on which has been widely researched. Then we briefly presented GPU acceleration work related to Viterbi algorithm.

Then Chapter 2 presented the details of our *cudaHmmsearch* optimization approaches, which is based upon the open source of MSV algorithm from HMMER3's *hmmsearch* database searching tool, one of the widely used Bioinformatics application. At the end of Chapter 2, 6 steps were summarized for better performance of CUDA programming.

Comprehensive benchmarks were performed in Chapter 3. The results were analyzed and the efficiency of the *cudaHmmsearch* implementations on the GPUs is proved. 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 [1] is the complete list of CUDA-enabled GPUs.

4.2 Recommendations for further research

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 Section3.2.2, 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.

[1] https://developer.nvidia.com/cuda-gpus

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

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

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

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.

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

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

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.

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.

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.

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.

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.

Ali Khajeh-Saeed, Stephen Poole and J. Blair Perot *Acceleration of the SmithWaterman algorithm using single and multiple graphics processors*. Journal of Computational Physics 229 (2010) 42474258.

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.

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.

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.

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

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.

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.

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.

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

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.

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.

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.

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

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

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

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

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

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

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

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

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

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

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

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.

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.

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.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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/.

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

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

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

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.

SIB Bioinformatics Resource Portal. UniProtKB/Swiss-Prot protein knowledgebase release 2014-04 statistics. Website, 2014-04. URL http://web.expasy.org/docs/relnotes/relstat.html.