Parallel Implementation of MAFFT on CUDA-Enabled Graphics Hardware

Xiangyuan Zhu, Kenli Li, Ahmad Salah, Lin Shi, and Keqin Li

Abstract—Multiple sequence alignment (MSA) constitutes an extremely powerful tool for many biological applications including phylogenetic tree estimation, secondary structure prediction, and critical residue identification. However, aligning large biological sequences with popular tools such as MAFFT requires long runtimes on sequential architectures. Due to the ever increasing sizes of sequence databases, there is increasing demand to accelerate this task. In this paper, we demonstrate how graphic processing units (GPUs), powered by the compute unified device architecture (CUDA), can be used as an efficient computational platform to accelerate the MAFFT algorithm. To fully exploit the GPU's capabilities for accelerating MAFFT, we have optimized the sequence data organization to eliminate the bandwidth bottleneck of memory access, designed a memory allocation and reuse strategy to make full use of limited memory of GPUs, proposed a new modified-run-length encoding (MRLE) scheme to reduce memory consumption, and used high-performance shared memory to speed up I/O operations. Our implementation tested in three NVIDIA GPUs achieves speedup up to 11.28 on a Tesla K20m GPU compared to the sequential MAFFT 7.015.

Index Terms—CUDA, graphics hardware, GPGPU, MAFFT, sequence alignment

1 Introduction

MSA is a fundamental tool for phylogeny inference, protein structure and function prediction, and other common tasks in sequence analysis [1]. The typical MSA task consists of seeking an alignment that maximizes the sum of similarities for all pairs of sequences. Since the computational cost increases exponentially with the number of sequences, the dynamic programming (DP)-based algorithm is commonly used for pairwise optimal alignment. With respect to large alignment targets, the progressive method is widely used. Several progressive alignment tools have been introduced, e.g., [2], [3], [4]. Among them, MAFFT is one of the most popular software package with citation of over 1,400 in the web of science. The huge popularity of the MAFFT program comes from the continuing efforts by offering new functions and the excellent service with about seven MAFFT Web servers including European Molecular Biology Laboratory-European Bioinformatics Institute (EMBL-EBI), DNA Data Bank of Japan (DDBJ), and the MPI Bioinformatics Toolkit.

Although MAFFT is a fast alignment program and already has a multi-thread version for some options, there are still great demands for faster solutions. This is due to both the increasing amount of sequence data and the limitations for some options on the maximum number of input sequences for the infeasible runtimes.

In this paper, we present a new approach to accelerating MAFFT on GPUs using the CUDA programming model. Compared with the implementations of other MSA algorithms on GPUs, parallelization of MAFFT is more challenging since the space complexity of most MAFFT options is proportional to L^2, where L is the average length of sequences, significantly conflicting with the limited memory size of GPUs. To gain efficiency, we have optimized the sequence data organization to eliminate the bandwidth bottleneck of memory access, designed a memory allocation and reuse strategy to make full use of limited memory of GPUs, and proposed a new MRLE scheme to compress the scoring matrix, the key space bottleneck of MAFFT. The performance of our implementation is tested in terms of accuracy and runtime for aligning large amount of sequences with various length. Our implementation achieves speedup up to 11.28 on an NVIDIA Tesla K20m GPU while delivering accuracy identical to MAFFT.

The rest of this paper is organized as follows. In Section 2, the features of CUDA and the MAFFT algorithm are reviewed. Section 3 is devoted to the presentation of our CUDA-based MAFFT algorithm. Performance is evaluated in Section 4. Finally, Section 5 outlines the main conclusions.

2 Preliminaries and Related Work

2.1 Parallel Programming with CUDA

In the CUDA programming model, a program consists of two parts: a *host* program running on a host CPU, and one or more *kernels* executing parallel programs on a parallel *device*. A kernel is written in a C-like programming language,

• *X. Zhu is with the College of Information Science and Engineering, Hunan University, Changsha, Hunan 410082, China and the School of Computer Science, Zhaoqing University, Zhaoqing, Guangdong 526061, China. E-mail: hnzxy@hnu.edu.cn.*
• *K. Li is with the College of Information Science and Engineering, Hunan University, Changsha, Hunan 410082, China and the Department of Computer Science, State University of New York, New York, NY 12561. E-mail: lkl@hnu.edu.cn.*
• *A. Salah, L. Shi, and K. Li are with the College of Information Science and Engineering, Hunan University, Changsha, Hunan 410082, China. E-mail: ahmad@hnu.edu.cn, shilin@aimlab.org, lik@newpaltz.edu.*

Manuscript received 26 Jan. 2014; revised 9 July 2014; accepted 18 Aug. 2014. Date of publication 25 Aug. 2014; date of current version 30 Jan. 2015.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TCBB.2014.2351801

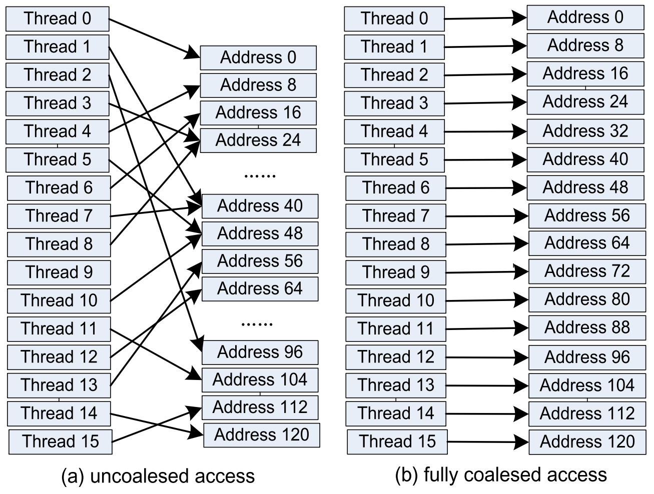


Fig. 1. The effect of coalescing on memory reads.

which performs the computation by a single *thread* and is invoked as a set of concurrently executing threads. These threads are organized in *thread blocks* and many thread blocks may run in a *grid* hierarchically.

A modern NVIDIA GPU is built on a scalable processor array, organized into a number of streaming multiprocessors (SMs). All threads of a thread block are executed concurrently on a single SM. The SM executes threads in small groups of 32 threads, called warps, in a *single-instruction multiple-thread* (SIMT) fashion.

In CUDA, the host CPU and devices have separate memory spaces. Device memory can be categorized in six groups: per-thread registers, per-thread private local memory, per-block shared memory, global memory for data shared by all threads, texture memory, and constant memory. Texture memory and constant memory can be regarded as fast read-only caches. Only threads within a thread block can communicate through shared memory and may directly synchronize using barriers.

Global memory is accessed via 32-, 64-, or 128-byte memory transactions. To maximize memory throughput, the memory accesses of threads within a half-warp (i.e., a group of 16 threads) are coalesced into one or more memory transactions. Taking Fig. 1 as an example, in Fig. 1a, each thread of a half-warp of 16 threads accesses a 8-byte value which stores at unorder address in global memory. In this case, each thread touches a separate address segment, resulting in an uncoalesced access. In Fig. 1b, since the memory requests performed by a half-warp accesses precisely one segment, a fully coalesced access takes place. Compared with the uncoalesced access, the GPU issues a single 128-byte load, thus no bandwidth is wasted and only a single memory transaction is needed.

2.2 Multiple Sequence Alignment and MAFFT Algorithm

MSA is to put protein or DNA residues in the same column according to some selected criteria, being an NP-hard optimization problem. There are many MSA heuristics and the progressive alignment method is one of the most widely used. For a generic MSA approach, a guide tree is created based on all-to-all pairwise comparisons, and an MSA is constructed using a profile-profile alignment algorithm at each node of the guide tree. To improve the alignment accuracy, various iterative refinement methods are usually incorporated in the later stages.

MAFFT is a popular MSA program package for unix-like operating systems. Two novel techniques, i.e., homologous regions identification by the fast Fourier transform (FFT) and a simplified scoring system for improving both accuracy and speed, are incorporated in MAFFT. MAFFT has several options and covers various types of MSA problems including a small alignment consisting of distantly related sequences, large-scale alignment and ncRNA structural alignment. Table 1 shows the major options of MAFFT. More details can be found in [5], [6], [7], [8], [9].

2.3 Previous Work of Sequence Alignment with CUDA

Due to the massively parallel processing power and availability in PC desktops, GPUs have been widely applied to a large number of applications in bioinformatics. Recent efforts on accelerating sequence alignment applications using CUDA includes MUMmerGPU, Smith-Waterman (SW), CUDA-BLAST, and CUDA-BLASTP.

Michael et al. [10] proposed MUMmerGPU 1.0, a parallel pairwise local sequence alignment program that runs on GPUs in common workstations, achieving more than 10-fold speedup over a serial CPU version of the sequence alignment kernel. By featuring a new stackless depth-first-search print kernel, MUMmerGPU 2.0 [11] is 13-fold and 4-fold faster than the serial CPU version and

TABLE 1
Description of the Major Options of MAFFT

Option name	Command	Description
Progressive methods. For a medium-scale alignment ($\sim$200<N<$\sim$10,000).		
FFT-NS-1	mafft −−retree 1 input	Approximately two times faster than the default
FFT-NS-2	mafft −−retree 2 input	Default
Iterative refinement methods. For a small-scale alignment (N<$\sim$200, L<$\sim$10,000).		
FFT-NS-i	mafft −−retree 2 −−maxiterate 16 input	Fastest of the four in this category. Use the weighted sum-of-pairs (WSP) score only
G-INS-i	mafft −−globalpair −−maxiterate 16 input	Uses WSP score and consistency score from global alignments
L-INS-i	mafft −−localpair −−maxiterate 16 input	Uses WSP score and consistency score from local alignments with the affine gap cost
E-INS-i	mafft −−genafpair −−maxiterate 16 input	Uses WSP score and consistency score from local alignments with a generalized affine gap cost

TABLE 2
Description of Smith-Waterman Implementations with CUDA

Paper	Implementation	Align	Max. Query	GCUPS*	GPU
[12]	SW-CUDA	no	567	3.4	8800 GTX
[13]	CUDA-SSCA#1	yes	1,024	1.0	GTX 295
[14]	GSM	no	1,024	N/A	C870
[15]	CUDASW++ 1.0	no	5,478	9.7	GTX 295
[16]	CUDASW++ 2.0	no	5,478	17	GTX 295
[17]	DOPA	no	5,478	21.4	GTX 275
[18]	CUDAlign 1.0	no	32,799,110	20.3	GTX 285
[19]	CUDAlign 2.1	no	59,373,566	30.2	GTX 560 Ti

GCUPS means Billions of Cells Updated per Second.

MUMmerGPU 1.0, respectively. Both MUMmerGPU 1.0 and MUMmerGPU 2.0 were tested in the machine which has a 3.0 GHz dual-core Intel Xeon 5160 with 2 GB of RAM, and a NVIDIA GeForce 8800 GTX.

Many CUDA implementations of SW have been proposed [12], [13], [14], [15], [16], [17], [18], [19]. Table 2 summarizes the main characteristics of them. Column 5 lists the best performance presented in these papers. These works tackle the problem of finding similarities between a query sequence of unknown functionality and a database of known sequences. Except the method in [13] providing the alignment as output, most of the approaches retrieve only the highest score as a measure of the similarity between the sequences. Note that the sequences used in [12], [13], [14], [15], [16], [17] are protein sequences. Only CUDAlign 1.0 [18] and CUDAlign 2.1 [19] compare real DNA sequences that range from 162 KBP (Thousand Base Pairs) to 59 MBP (Million Base Pairs).

Vouzis et al. [20] developed GPU-BLAST to accelerate the Basic Local Alignment Search Tool (BLAST) based on the source code of NCBI-BLAST, maintaining the same input and output interface while producing identical results.

Liu et al. [21] presented a method called CUDA-BLASTP to accelerate BLASTP by using a compressed deterministic finite state automaton as well as a hybrid parallelization scheme with CUDA. CUDA-BLASTP achieves speedup up to 10.0 on a GeForce GTX 295 GPU compared with the sequential NCBI BLASTP 2.2.22.

3 Parallelizing the MAFFT Algorithm with CUDA

There are many options in the MAFFT package. Different options focus on handling different alignment problems. In order to design an efficient MAFFT algorithm for GPUs, profiling should be done to identify the most time-consuming parts of MAFFT, the severe performance bottleneck of the system. It is also a general method and skill to achieve the best performance in parallel programming.

3.1 Profiling of MAFFT

We constructed nine testsets by using PSI-BLAST to search the NCBI non-redundant protein sequence database for hits on nine sequences P02232, P14942, P07327, P01008, P03435, P42357, P21177, Q38941, and P27895 respectively, selecting the highest-scoring 500 sequences. As Table 3 shown, the rows 2 and 3 are the average length and the average distance of these testsets. The remaining rows depict the runtimes of MAFFT options on the testset. From Table 3 we can see that FFT-NS-1 is the fastest option while L-INS-i (abbreviated as LINSi) is the most time-consuming one. It must be noted that the runtimes of MAFFT options increase when the average lengthes of testsets increase.

After continuing profiling LINSi on the testset, we found that the most time-consuming parts of LINSi are pairwise alignment and iterative refinement. In the pairwise alignment stage, a scoring matrix between two amino acid sequences is constructed from the similarity matrix. While in the iterative refinement stage, the LINSi alignment is repeated until the alignment score can no longer be improved. These result in high time complexity and space complexity, which are $O(N^2L^2)$ and at least $O(N^2) + O(L^2) + O(NL)$ respectively, where N and L are the number of sequences and the average length of sequences respectively. As the most accurate option of MAFFT, LINSi is recommended for alignment of less than 200 sequences with the maximum length 5,000 residues because of high computation cost. Hence, it is imperative to accelerate LINSi to handle more sequences in feasible time. In this paper, we design an algorithm to accelerate LINSi with CUDA (named as CUDA-LINSi). It should be noted that this algorithm is suitable for accelerating the other two time-consuming options G-INS-i and E-INS-i, which also improve alignment accuracy by introducing pairwise alignment information and iterative refinement.

3.2 The Flow of LINSi

In order to design an efficient CUDA-LINSi algorithm, we analyze the flow of LINSi firstly. The procedure of LINSi

TABLE 3
Running Time of Different MAFFT Options

Option	P02232	P14942	P07327	P01008	P03435	P42357	P21177	Q38941	P27895
average length	156	224	375	423	563	563	726	1319	1083
average distance	57%	64%	75%	54%	96%	64%	99%	38%	36%
L-INS-i	31	39	105	240	140	222	237	13,557	27,777
E-INS-i	26	29	77	197	83	168	143	10,223	18,803
G-INS-i	32	30	57	155	37	151	56	9,116	12,476
FFT-NS-i	9	8	9	31	6	24	10	1,450	2,050
FFT-NS-2	1	0	1	3	2	2	2	122	242
FFT-NS-1	0	1	1	1	1	2	1	121	181

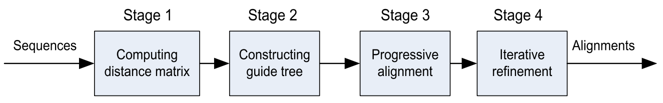


Fig. 2. The LINSi processing procedure.

consists of four stages, as shown in Fig. 2. We briefly describe each step in the following. More details can be found in [5], [6], [7].

Stage 1. This stage makes a distance matrix by calculating all-pairwise alignment. For pairwise alignment, LINSi uses a local pairwise alignment with the affine gap cost.

Stage 2. This stage constructs a guide tree from the distance matrix generated in Stage 1, using the UPGMA method with modified linkage.

Stage 3. This stage outputs an MSA which aligns the sequences according to the branching order of the guide tree constructed in Stage 2.

Stage 4. The initial alignment constructed in Stage 3 is divided into two groups based on a tree-dependent partitioning method, which are then realigned using an approximate group-to-group alignment algorithm. The new alignment replaces the old one if it has a higher score. This process is repeated until no more improvements are made.

Based on the characteristics of the LINSi processing pipeline, we have designed parallel algorithms for Stages 1 and 4. Fig. 3 shows our streaming algorithm framework for CUDA-LINSi. There are two kernels in this framework. In the first kernel, the input sequences are divided into sequence sets according to a sequence data transformation method. All threads then process pairwise alignment of Stage 1 in a coarse-grained parallel fashion. In the second kernel, tree-dependent partitioning by Stage 4 are distributed evenly to all thread blocks. Threads in the same thread block then process individual group-to-group alignment in a fine-grained parallel fashion.

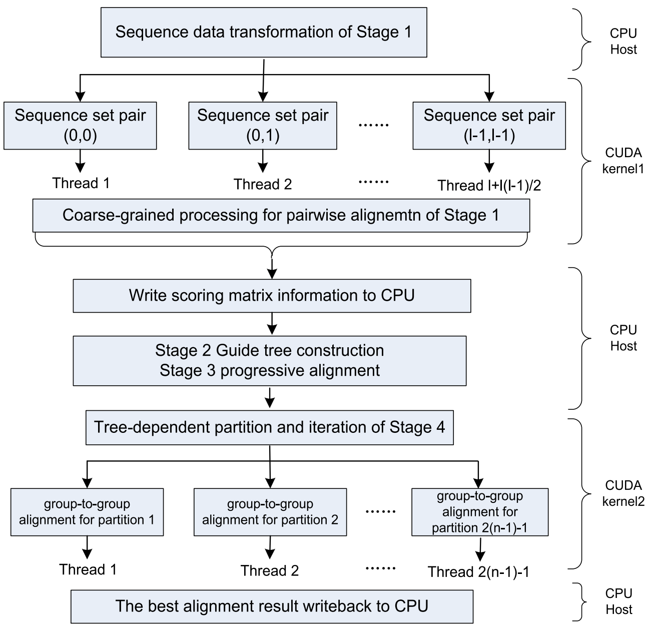

Sequence data transformation of Stage 1
CPU Host
Sequence set pair (0,0)
Sequence set pair (0,1)
......
Sequence set pair (l-1,l-1)
Thread 1
Thread 2
......
Thread l+l(l-1)/2
CUDA kernel1
Coarse-grained processing for pairwise alignemtn of Stage 1
Write scoring matrix information to CPU
Stage 2 Guide tree construction
Stage 3 progressive alignment
CPU Host
Tree-dependent partition and iteration of Stage 4
group-to-group alignment for partition 1
group-to-group alignment for partition 2
......
group-to-group alignment for partition 2(n-1)-1
Thread 1
Thread 2
......
Thread 2(n-1)-1
CUDA kernel2
The best alignment result writeback to CPU
CPU Host

Fig. 3. Our CUDA-LINSi framework.

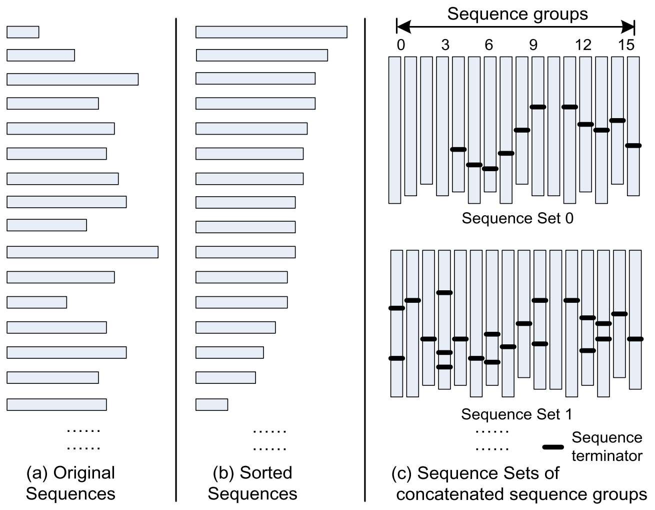


Fig. 4. The sequence data transformation.

3.3 Sequence Data Transformation

Instead of directly loading sequences, our GPU implementation transforms them to a format to better match the device capabilities. As shown in Fig. 4, the transformation process consists of the following steps.

3.3.1 Sorting

In the CUDA Architecture, a warp refers to a collection of 32 threads that execute in locksteps. It means that the threads in a half-warp will have to wait for each other to finish their workload instead of continuing on independently. To reduce this waiting time, the sequences are sorted by length to minimize length differences between neighboring threads, as shown in Fig. 4b.

3.3.2 Concatenation

After sorting, groups of 16 sequences are taken and managed in sequence sets that will be handled by a half-warp of threads, as shown in Fig. 4c. Even though sorting by length has somewhat balanced workload within each sequence set, various sequence sets still have different sizes. To overcome this, sequences within a sequence set are concatenated with leftover sequences to form sequence groups. The lengthes of sequence groups within a sequence set are nearly equal or equivalent to the length of the longest sequence in that set. This results in a workload balancing for each thread in a half-warp processing of a sequence set. Sequence terminators are inserted between the concatenated sequences. They are labels to initiate a new pairwise alignment.

3.3.3 Interleaving

Once all sequences have been managed into 16-wide sets of sequence groups, they are ready to store in a character matrix. The sequence sets are stored in an interleaved fashion, as shown in Fig. 5. Each interleaved subset consists of eight bytes of characters from each sequence group. Eight characters of the set's first sequence group are stored, then eight characters of the set's second group, and so on. As there are 16 sequence groups in each sequence set, each thread in a half-warp is now able to load 8 bytes of sequence

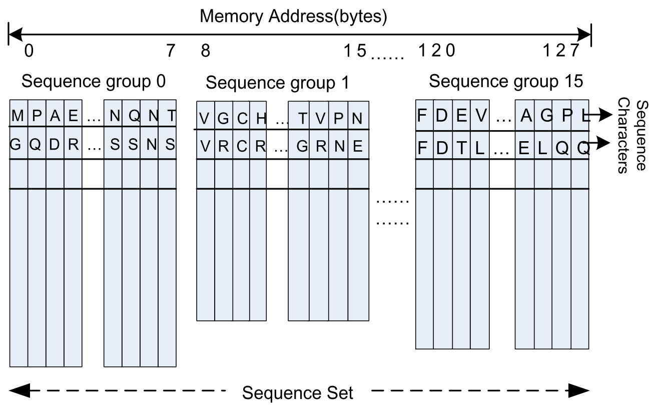


Fig. 5. The sequence data interlacing.

characters from neighboring addresses. As a result, 128-byte coalesced loading takes place.

The pipeline of sequence transformation consists of three steps: sorting by sequence length, whose time complexity is $O(N^2)$, concatenation and interleaving, whose time complexity is $O(N^3)$, where N is the number of sequences. Then the time complexity of sequence data transformation is $O(N^3)$. The performance of sequence data transformation will be discussed in Section 4.1.

3.4 Coarse-Grained Parallel Algorithm for Pairwise Alignment

The first stage of LINSi conducts a local pairwise alignment with the affine gap cost. We take advantage of the inherent parallelism of pairwise alignment and design a coarse-grained algorithm to accelerate it. In the CUDA kernel 1 shown in Fig. 3, a thread processes the pairwise alignment on a sequence set pair, i.e., greater than or equal to 16×16 sequence pairs with concatenation, instead of a sequence pair. The main reason we choose this method is to reuse memory between threads. The memory allocation and reuse strategy will be detailed in Section 3.5.

After sequence data transformation, m sequences are managed to l sequence sets. The sequence set pairwise alignment can be represented as the following $l \times l$ lower-left triangular matrix:

$$\begin{bmatrix} (l-1,l-1) & & & & \\ (l-2,l-2) & (l-2,l-1) & & & \\ (l-3,l-3) & (l-3,l-2) & (l-3,l-1) & & \\ \cdots & \cdots & \cdots & & \\ (0,0) & (0,1) & (0,2) & \cdots & (0,l-1) \end{bmatrix}. \tag{1}$$

In order to guarantee that a thread only handles a sequence set pair at any time, it is natural that the sequence set pairs in matrix (1) can be represented as $threadIDs$, as shown in matrix (2) below:

$$\begin{bmatrix} \mathbf{1} & & & & & \\ 2 & \mathbf{3} & & & & \\ 4 & 5 & \mathbf{6} & & & \\ 7 & 8 & 9 & \mathbf{10} & & \\ 11 & 12 & 13 & 14 & \mathbf{15} & \\ \cdots & \cdots & \cdots & \cdots & \cdots & \\ \cdots & \cdots & \cdots & \cdots & \cdots & \mathbf{l+\frac{l(l-1)}{2}} \end{bmatrix}. \tag{2}$$

Theorems 1, 2, and 3 characterize the relation between sequence set pairs and $threadIDs$.

Theorem 1. *Let n be a positive integer, i.e., $n \in N^+$. The positive integer root of equation $x^2 + x = 2s_n$ is n, where $s_n = \sum_{k=1}^{n} k$.*

Proof. As $s_n = \sum_{k=1}^{n} k = \frac{n(n+1)}{2}$, we have

$$x^2 + x = 2s_n = 2 \times \frac{n(n+1)}{2} = n(n+1). \tag{3}$$

The solutions to equation (3) are

$$x = \frac{-1 \pm \sqrt{1 + 4n(n+1)}}{2}$$

i.e.,

$$x_1 = n, \quad x_2 = -(n+1).$$

Hence, the positive integer root of equation $x^2 + x = 2s_n$ is n. □

Theorem 2. *Let $S = \{s_i = \sum_{k=1}^{i} k \,|\, i \in N^+\}$. The positive roots of all equations represented as $x^2 + x = 2s_i$ are the consecutive positive integer values, i.e., $X = \{i \,|\, i \in N^+\}$ for set S.*

Proof. By Theorem 1, for $i \in N^+$, the positive root of equation $x^2 + x = 2s_i$ is i, where $s_i = \sum_{k=1}^{i} k$. Then, for all s_i, where $s_i \in S$, the positive roots of all equations represented as $x^2 + x = 2s_i$ are the consecutive positive integer values, i.e., $X = \{i \,|\, i \in N^+\}$ for set S. □

Theorem 3. *For each positive integer y which does not belong to $S = \{s_i = \sum_{k=1}^{i} k \,|\, i \in N^+\}$, the positive root of equation $x^2 + x = 2y$ is a real number in the range $(j, j+1)$, where $\sum_{k=1}^{j} k < y < \sum_{k=1}^{j+1} k$.*

Proof. Since $\sum_{k=1}^{j+1} k - \sum_{k=1}^{j} k = j + 1$, set S contains non-consecutive integer values. By Theorem 2, the root solutions for set S are consecutive integer values. Then, for $\sum_{k=1}^{j} k < y < \sum_{k=1}^{j+1} k$, the root should be a real number in the range $(j, j+1)$. □

To get the one-to-one relationship between the thread $threadID$ and the sequence set pair ($Seqset1ID$, $Seqset2ID$), it is natural to get the row number $RowID$ and column number $ColID$, where the $Seqset1$ and $Seqset2$ locate in matrix (1). With the help of Theorems 1, 2, and 3, we design a constant time algorithm Algorithm 1 for sequence set pair retrieval. As shown in steps 12-13, when $threadID \in S = \{\sum_{k=1}^{1} k, \sum_{k=1}^{2} k, \ldots, \sum_{k=1}^{i} k\} = \{1, 3, 6, 10, \ldots, l + \frac{l(l-1)}{2}\}$ (i.e., the bold elements in matrix (2)), according to Theorem 2, the $RowID$ and $ColID$ both equal to the positive integer root of $x^2 + x = 2\,threadID$. As listed in steps 7-9, when $threadID \notin S$, using Theorem 3, the $RowID$ is the ceil of the obtained non-integer root of $x^2 + x = 2\,threadID$. The fraction of the root and the $RowID$ are used to get the $ColID$. As shown in steps 16 and 17, after transforming by the intermediate variables $RowID$ and $ColID$, the corresponding sequence set pair ($Seqset1ID, Seqset2ID$) is achieved and assigned to thread $threadID$.

For example, if there are six sequence set (i.e., $l = 6$), it means 21 sequence set pairs (i.e., $l + \frac{l(l-1)}{2}$) need to be handled. For thread 15 which belongs to set S, the $RowID$ and $ColID$ both equal to the positive integer root of $x^2 + x = 2 \times 15$ minus 1, i.e., 4, and then the $Seqset1ID$ and $Seqset2ID$ are 1 and 5 respectively. It should be noted that since both $Seqset1ID$ and $Seqset2ID$ begin with 0, both $RowID$ and $ColID$ begin with 0. For thread 17 which does not belong to set S, according to Theorem 3, its $RowID$ equals to that of thread 21, i.e., 5, and then $ColID$, $Seqset1ID$, and $Seqset2ID$ are 1, 0, and 1 respectively. The following matrix (4) shows the mapping of a $threadID$ onto a sequence set pair ($Seqset1ID$, $Seqset2ID$):

$$\begin{bmatrix} \mathbf{1} & & & & & \\ (5,5) & & & & & \\ \mathbf{2} & \mathbf{3} & & & & \\ (4,4) & (4,5) & & & & \\ \mathbf{4} & \mathbf{5} & \mathbf{6} & & & \\ (3,3) & (3,4) & (3,5) & & & \\ \mathbf{7} & \mathbf{8} & \mathbf{9} & \mathbf{10} & & \\ (2,2) & (2,3) & (2,4) & (2,5) & & \\ \mathbf{11} & \mathbf{12} & \mathbf{13} & \mathbf{14} & \mathbf{15} & \\ (1,1) & (1,2) & (1,3) & (1,4) & (1,5) & \\ \mathbf{16} & \mathbf{17} & \mathbf{18} & \mathbf{19} & \mathbf{20} & \mathbf{21} \\ (0,0) & (0,1) & (0,2) & (0,3) & (0,4) & (0,5) \end{bmatrix}. \tag{4}$$

Algorithm 1. Constant Time Algorithm for Sequence Set Pair Retrieving

```
Input: The number of sequence sets $l$ and thread index $threadID$.
Output: Sequence set pair ($Seqset1ID$, $Seqset2ID$) aligned on thread $threadID$.
1:  int $a$, $b$, $c$, $RowID$, $ColID$
2:  double $X1$
3:  $a \leftarrow 1, b \leftarrow 1, c \leftarrow threadID$
4:  $c \leftarrow -2c$
5:  $X1 \leftarrow \frac{-b+\sqrt{b^2-4ac}}{2a}$
6:  /* For $threadID \notin S$, using Theorem 3 to get $RowID$ and $ColID$ where $Seqset1$ and $Seqset2$ locate */
7:  if $X1 \bmod 1 > 0$ then
8:      $RowID \leftarrow \lfloor X1 + 1 \rfloor - 1$
9:      $ColID \leftarrow \lfloor (X1 \bmod 1)/(1/\lfloor X1 + 1 \rfloor) \rfloor - 1$
10:     /* For $threadID \in S$, using Theorem 2 to get $RowID$ and $ColID$ where $Seqset1$ and $Seqset2$ locate */
11: else
12:     $RowID \leftarrow \lfloor X1 \rfloor - 1$
13:     $ColID \leftarrow \lfloor X1 \rfloor - 1$
14: end if
15: /* Get $Seqset1ID$ and $Seqset2ID$ by using $RowID$ and $ColID$ */
16: $Seqset1ID \leftarrow l - RowID - 1$
17: $Seqset2ID \leftarrow Seqset1ID + ColID$
18: Output $threadID$ ($Seqset1ID$, $Seqset2ID$)
```

3.5 Similarity Matrix Stores and Accesses

Protein alignment requires the use of a similarity matrix, which is accessed every time two amino acids are aligned, making its access time critical to the alignment performance. In order to increase the efficiency of alignment, MAFFT adopted a normalized similarity matrix that has both positive and negative values. Instead of a 20×20 regular matrix, MAFFT stored the similarity matrix as a 128×128 spare matrix which uses ASCII values of the capital characters to index rows and columns. Although it dramatically simplifies the access, the sparse matrix consumes more than 40 times memory space. Taking the limited memory size of GPU into consideration, we compressed the spare similarity matrix to three vectors using the traditional compressed sparse row (CSR) algorithm. In addition to a constant data access time, our compressed matrix saves 94.7 percent memory space compared with the sparse similarity matrix.

Similarity matrix accesses are random and are completely dependent on the sequences, complicating the choice of memory used. Global memory is not a good choice for such a frequent usage due to its high access time. Also the random nature of similarity matrix accesses makes coalescing very difficult. As an alternative, texture memory and constant memory are the good choice for their unique features. Texture memory is a cached window into global memory that offers lower latency and does not require coalescing for best performance. Like texture memory, constant memory is another variation of read-only memory and also cached. It is capable of broadcasting a single read to a half-warp threads, effectively saving up to 15 reads. Both of them are thus well suited for random access. We store the compressed similarity matrix in texture memory and constant memory respectively. The Tesla C2050 GPU used for our implementation has 8 KB of constant cache per multiprocessor and 12KB of texture cache per multiprocessor, respectively. Compared with constant memory implementation in Tesla C2050, aligning 800 sequences with average length 430 residues resulted in 3.5 percent performance improvement with texture memory implementation.

3.6 Scoring Matrix Computes and Compressed Stores

In order to increase the efficiency of alignment, MAFFT adopted a modified scoring system (similarity matrix and gap penalties). The optimal alignment between two groups of sequences is given by Equation (5):

$$P(i,j) = H(i,j) + max \begin{cases} P(i-1, j-1) & \\ P(x, j-1) - G_1(i,x) & 1 \leq x < i-1 \\ P(i-1, y) - G_2(j,y) & 1 \leq y < j-1. \end{cases} \tag{5}$$

$P(i,j)$ is the accumulated score for the optimal path from $(1,1)$ to (i,j), $G_1(i,x)$ and $G_2(j,y)$ are gap penalties, and $H(i,j)$ is the homology matrix constructed from the similarity matrix. More details about the scoring system can be found in [5].

To get the maximum score of a sequence pair, it involves reading and writing four temporary values (homology distance and gap penalties), for eight accesses in total. In LINSi, all these values are stored in vectors which are allocated dynamically according to the lengths of the sequence pair.

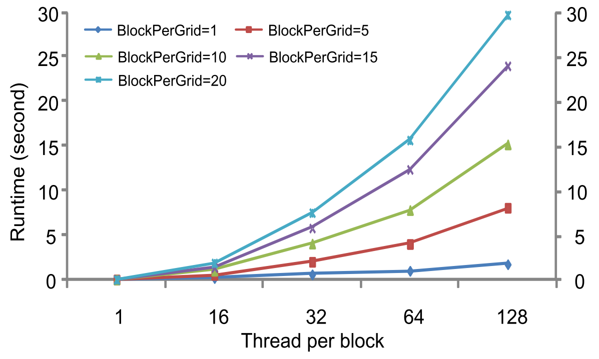


Fig. 6. Comparison of runtimes of allocation memory at runtime.

However, there is a severe performance bottleneck to allocate and deallocate memory in runtime in CUDA because of a requirement of a form of global synchronization when systems become more and more parallel [22]. We allocated a 100×100 integer matrix. As depicts in Fig. 6, the runtimes of allocation at runtime (using malloc() function) increases exponentially with the number of blocks per grid and the number of threads per block. However, per-allocation an equivalent matrix (using cudamalloc() function) saves 24.38 percent runtime. In order to increase the efficiency of CUDA-LINSi, we propose a memory pre-allocation and reuse strategy.

A sequence set pair is assigned to a thread explicitly by using Algorithm 1. We pre-allocate a global memory pool to all sequence set pairs according to the length of their two longest sequences. Each thread uses a segment of the memory pool. It means that all sequence pairs of the sequence set pair share the same memory segment whose size is enough to reuse. In addition to the efficiency improved by memory pre-allocation instead of allocation at runtime in CUDA, this strategy improves more than 256-fold memory use rate because more than 16×16 sequence pairs share the same memory segment.

To track the path of the optimal score for recording the number of gaps that start or end at each site of the sequence pair results in high space complexity, which brings huge challenges to CUDA-LINSi implementation for GPU's limited memory. The scoring matrix has three features: 1) all elements are initialized to zeros; 2) there are many consecutively identical values called as characteristic value *localstop* here which is an indicator used to stop inserting gaps to the sequences of a sequence pair; and 3) there are many consecutively decreasing values to indicate the number of gaps that start or end at each site. To save memory consumption, we propose a new scheme that combines compressed row storage (CRS) with run-length encoding (RLE), namely MRLE, to compress the scoring matrix, the most space-consuming part of LINSi.

CRS is a popular algorithm for compressing a sparse matrix. Instead of storing as a two dimensional array, CRS format uses three vectors val, col_ind, and row_ptr to store nonzero elements. val stores the values of the nonzero elements in a row-wise fashion, col_ind stores the corresponding column indices of the elements in the val, and row_ptr stores the locations in the val and col_ind that start a row. RLE is a well-known method for compressing strings. It simply represents the consecutive, identical symbols of a string with a run, usually denoted by σ^i, where σ is an alphabet symbol and i is its repetition times. For example, a string $aaaddbbbbbbcccc$ is encoded as $a^3d^2b^6c^4$ in the RLE format.

The data compression and decompression processes of MRLE are depicted in Algorithms 2 and 3. The objective of MRLE is to reduce (or compact) the number of consecutive, identical or decreasing values into a smaller number. MRLE scheme first considers the type of runs of consecutively identical values (i.e., zeros and characteristic values), so the total number of runs will decrease, which results in better compression effect. Then, it further explores the consecutively decreasing values which decreases by one in each run, and further improves the compression effect.

Algorithm 2. Modified Run Length Encoding Algorithm for Scoring Matrix Compression

```
Input: A scoring matrix $M$ with $m$ rows and $n$ columns.
Output: Two vectors $Av$ and $Ar$.
1:  $index \leftarrow 0$
2:  $Ar[0] \leftarrow 0$
3:  /* Compress $M$ to $Av$ and $Ar$ row by row */
4:  for $i \leftarrow 0$ to $m$ do
5:    /* Calculate $count$ which is the times of consecutively
      identical values or consecutive, decreasing values */
6:    $count \leftarrow 0$
7:    for $j \leftarrow 1$ to $n$ do
8:      Read the value $M[i][j]$
9:      while $M[i][j]$ is a consecutively identical values or
        consecutive, decreasing values do
10:       $j \leftarrow j+1$
11:       /* At the end of the row */
12:       if $j \geqslant n$ then
13:         break;
14:       end if
15:       $count \leftarrow count + 1$
16:   end while
17:   /* For $count > 0$, store $M[i][j]$ in $Av[index]$, and then store
      its identical/decreasing times $count$ plus $2 \times localstop$ in
      $Av[index+1]$ */
18:   if $count > 0$ then
19:     $Av[index] \leftarrow M[i][j]$
20:     $index \leftarrow index + 1$
21:     $Av[index] \leftarrow count + 2 \times localstop$
22:     $index \leftarrow index + 1$
23:   else
24:     /* For $count = 0$, only store $M[i][j]$ in $Av[index]$ */
25:     $Av[index] \leftarrow M[i][j]$
26:     $index \leftarrow index + 1$
27:     end if
28:   end for
29:   /* Complete the compression of the $i$th row of $M$. Store
      the current $index$ of $Av$ in $Ar[i+1]$ to indicate the
      position which the first element of the $(i+1)$th row of
      $M$ locates in $Av$ */
30:   $Ar[i+1] \leftarrow index$
31: end for
```

As shown in Algorithm 2, MRLE compresses the scoring matrix M to two vectors Av and Ar. Av stores the first values of consecutively identical or decreasing elements of M and the times of them (called as $count$) in an interleaved fashion. Ar points to beginning of each row in Av. In order to distinguish the value of consecutively identical or decreasing elements of M from its consecutively repeating or decreasing times in the process of decompression, when storing $count$ to Av in the process of compression, Av stores the times of consecutively identical or decreasing elements of M with an extra value $2localstop$, where $localstop$ is the value of consecutively identical value in M. In Av, those whose values are greater than $2localstop$ are the times of consecutively identical or decreasing elements of M. Because the consecutively identical elements and the first value of decreasing elements of M are less than $2localstop$ absolutely. As shown in Algorithm 3, with the help of Ar, it is convenient to retrieve an element from Av corresponding to the location of matrix M.

Algorithm 3. Data Retrieve Algorithm from Compressed Vectors

```
Input: $Av$ and $Ar$ are the vectors compressed by Algorithm 2,
    $localstop$ is the characteristic value.
Output: The value $M[i][j]$, where $i$ and $j$ is the row and
    column index of scoring matrix $M$.
1:  /* $value$ is the current element of $Av$*/
2:  $value \leftarrow 0$
3:  /* $col\_value$ is the column position where $value$ locates
    in $M$ */
4:  $col\_value \leftarrow 0$
5:  /* $k$ is the range of $Ar$ corresponding to the $i$th row of
    $M$ */
6:  for $k \leftarrow Ar[i]$ to $Ar[i+1]$ do
7:    if $col\_value == j$ then
8:      $M[i][j] \leftarrow value$; break
9:    end if
10:   /* The current element of $Av$ is a value of $M$ */
11:   if $Av[k] \leqslant localstop$ then
12:     $value \leftarrow Av[j]$
13:     $col\_value \leftarrow col\_value + 1$
14:   else
15:     /*The current element of $Av$ is the times of $value$, i.e.,
        there are $Av[k] - 2 \times localstop$ consecutively identical
        or decreasing $value$s. Judge whether $M[i][j]$ belongs to
        these consecutive values */
16:     if $(col\_value + Av[k] - 2 \times localstop) > j$ then
17:       /* $M[i][j]$ is a consecutively decreasing value */
18:       if $((value \neq 0)\ and\ (value \neq localstop))$ then
19:         $M[i][j] = value - (j - col\_value)$; break;
20:       else
21:         /* $M[i][j]$ is a consecutively identical value */
22:         $M[i][j] \leftarrow value$; break
23:       end if
24:     else
25:       /* Calculate the column range of $value$ */
26:       $col\_value \leftarrow col\_value + Av[k] - 2 \times localstop - 1$
27:     end if
28:   end if
29: end for
```

The following is an example to demonstrate the MRLE algorithm. Matrix (6) below has 10 rows and 10 columns:

$$\begin{bmatrix} 21 & 21 & 21 & 21 & 21 & 21 & 21 & 21 & 21 & 21 \\ 21 & 21 & 0 & 21 & 21 & 0 & 21 & 21 & 0 & 21 \\ 21 & 0 & 21 & 21 & 0 & 21 & 0 & 0 & 21 & 21 \\ 0 & 21 & 21 & 21 & 21 & 21 & 0 & -2 & -3 & 0 \\ 21 & 0 & 0 & 0 & 21 & 0 & 0 & 2 & 0 & -2 \\ 0 & 0 & 0 & 21 & 0 & 21 & 21 & 0 & 3 & 2 \\ 0 & 0 & 21 & 0 & 0 & 21 & 21 & 21 & 21 & 0 \\ 0 & 0 & 0 & 0 & 0 & 21 & 0 & 21 & 21 & 21 \\ 21 & 0 & 0 & 0 & 21 & 0 & 0 & 21 & 21 & 21 \\ 21 & 0 & 0 & 0 & 0 & 21 & 0 & 0 & 0 & 21 \end{bmatrix}. \qquad (6)$$

The value of $localstop$ is 21. After using MRLE, Ar and Av are matrixes (7) and (8) respectively:

$$[0 \quad 2 \quad 12 \quad 22 \quad 29 \quad 38 \quad 47 \quad 55 \quad 61 \quad 69 \quad 76] \qquad (7)$$

$$\begin{bmatrix} 21 & 52 & & & & & & & & \\ 21 & 44 & 0 & 21 & 44 & 0 & 21 & 44 & 0 & 21 \\ 21 & 0 & 21 & 44 & 0 & 21 & 0 & 44 & 21 & 44 \\ 0 & 21 & 47 & 0 & -2 & 44 & 0 & & & \\ 21 & 0 & 45 & 21 & 0 & 44 & 2 & 0 & -2 & \\ 0 & 45 & 21 & 0 & 21 & 44 & 0 & 3 & 44 & \\ 0 & 44 & 21 & 0 & 44 & 21 & 46 & 0 & & \\ 0 & 47 & 21 & 0 & 21 & 45 & & & & \\ 21 & 0 & 45 & 21 & 0 & 44 & 21 & 45 & & \\ 21 & 0 & 46 & 21 & 0 & 45 & 21 & & & \end{bmatrix}. \qquad (8)$$

To clarify the MRLE more clearly, elements of Av are listed in matrix fashion rather than vector fashion. In the first row of matrix (6), there are 10 consecutively identical values 21. So $Av[0]$ stores 21. $Av[1]$ stores 52 which equals to the repeated times 10 plus $2 \times localstop$. These 10 elements are compressed to two elements which are stored in $Av[0]$ and $Av[1]$. The compression processes of other rows of matrix (6) are omitted because of the same method used. $Ar = [0, 2, 12, 22, 29, 38, 47, 55, 61, 69, 76]$. It means that from $Av[0]$ to $Av[1]$, the two elements are the compressed values of the first row of matrix (6). From $Av[2]$ to $Av[11]$, the 10 elements are the compressed values of the second row of matrix (6), and so on. Ar points to beginning of each row in Av. Using MRLE, the compression rate of matrix (6) is 14 percent. The compression effect of MRLE will be shown in Section 4.4.

The other optimizations to improve the performance are as follows.

- Smaller, 16-bit data type (short integer) for scoring matrix values cuts the theoretically required bandwidth in half as well as saves memory space in half.
- CudaMallocPitch( ) is used to allocate linear memory for 2D array as it makes sure that the allocation is appropriately padded to meet the alignment requirements.

3.7 Fine-Grained Parallel Algorithm for Iterative Refinement

In the iterative refinement stage of LINSi, a binary tree is generated according to the initial alignment constructed from progressive alignment in Stage 3. All the sequences are represented as leaf nodes of the binary tree. Using the tree-dependent restricted partition technique, the binary

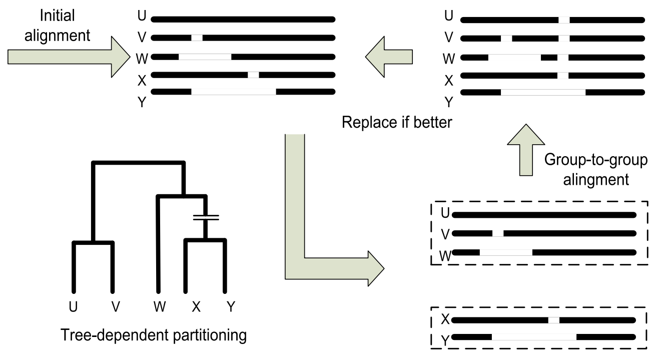


Fig. 7. Illustration of stage 4 of the LINSi algorithm.

tree is then divided into two groups. An approximate group-to-group alignment is implemented to get the alignment score. The process is repeated until no more alignment score improvements are made. Fig. 7 gives an illustration of this stage.

In CDUA-LINSi, we design and implement a CUDA-based fine-grained parallel algorithm to carry out the iterative refinement stage. Our method takes advantage of the fact that the tree-dependent partition and group-to-group alignment can be done independent of each other in parallel. Thus, the basic idea is to spawn enough threads to implement these operations in parallel. A thread is assigned to calculate a group-to-group alignment for a corresponding tree-dependent partition independently. The total number of threads needed is the number of branches of the binary tree. For example, n sequences need to be aligned. It means that the number of nodes of the corresponding binary tree is $2n-2$. Except the root node, the number of branches, i.e., the times the binary tree needs to be partitioned, is $2n-3$.

The goal of the CUDA-LINSi algorithm is to seek the highest scoring alignment in the given maximum iteration. To get the highest scoring alignment, we need to get the highest score of each iteration. For an iteration, the highest score is obtained from all thread blocks. It is well known that shared memory is expected to be much faster than global memory. In order to speed up the I/O operations, the alignment scores are stored in high-performance shared memory arrays. We exploit an opportunity to replace global memory accesses by shared memory accesses.

We have declared a buffer of shared memory named cache. This buffer is used to store each thread's running alignment score. We declare an array of size of the number of threads per block, so that each thread in the block has a place to store its temporary result. Each thread computes the alignment of the corresponding tree-dependent partition and stores its temporary score into the shared buffer. One simple way to accomplish the highest score reduction would be having one thread iterate over the shared memory and calculate a running score. However, since we have thousands of threads available to do our work, we do this comparison reduction in parallel and take time that is proportional to the logarithm of the length of the array. Fig. 8 illustrates one step of the comparison reduction. The size of each shared memory array is the number of threads per block. In the kernel program there are the number of branches working in the fine-grained concurrent way to

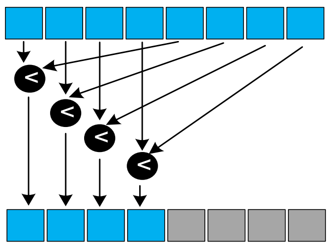


Fig. 8. One step of a comparison reduction.

calculate the group-to-group alignment score, and the number of such threads is the number of branches.

In the iterative refinement stage of LINSi, there are many complex data structures such as binary tree and linked list. To simplify GPU programming and data migration, we use unified memory which is first introduced in CUDA 6.0. Unified memory defines a new managed memory space in which both CPUs and GPUs of any type and architecture see a single coherent memory image with a common address space. In CUDA-LINSi, the program allocates managed memory for the topology structure of the binary tree via the new cudaMallocManaged() routine. The underlying system manages data access and locality of the binary tree within the CUDA program without need for explicit memory copy calls. This benefits CUDA-LINSi in two primary ways.

- GPU programming is simplified by unifying memory spaces coherently across all GPUs and CPUs in the CUDA-LINSi system.
- Data access speed is maximized by transparently migrating data.

Fig. 9 shows the pseudo-code of our implementation of the fine-grained parallel algorithm for Stage 4.

4 Performance Evaluation

The process of performance evaluation has been divided into three parts. The first part deals with traditional HPC metrics such as speedup and scalability. The second part deals with the overall performance such as quality assessment and computational complexity. The third part deals with the efficiency of the proposed compressed algorithm. CUDA-LINSi was implemented in CUDA 6.0 and tested in three NVIDIA GPUs, i.e., Tesla C2050, Tesla M2090, and Tesla K20m. The three boards were connected to three servers, i.e., 2 AMD Opteron Octa Core Processors 6134 with

```
/*Host program executed on CPU*/
1. According to the distance matrix of pairwise sequences and a predefined tree
   method, create a binary tree for all sequences.
2. Load all sequences and the binary tree into GPU memory.
3. For each iteration, launch the kernel
      /*Kernel program executed on each thread block*/
      4. Define and initialize shared memory arrays to store the alignment score
      5. Divide the binary tree into two groups according to branches
      6. Do fine-grained parallel group-to-group alignment
      7. Calculate the highest alignment score of all thread blocks and write it back
         to CPU
8. End for
9. Calculate the highest score of all iterations and output the best alignment results
```

Fig. 9. The pseudo-code of Stage 4 of our CUDA implementation for the fine-grained parallel algorithm.

TABLE 4
GPU Hardware Specifications

Detailed Specifications	Tesla C2050	Tesla M2090	Tesla K20m
Double-precision peak performance (Gflops)	515	665	1,170
Number of stream processors	448	512	2,496
Stream processors clock (GHz)	1.15	1.30	0.71
Memory size (GB)	3	6	5
Memory clock (GHz)	1.50	1.85	2.6
Memory bandwidth (GB/s)	144	177	208
Global memory size (MB)	2,687	5,375	4,800
Constant memory size (MB)	64	64	64
Shared memory size per block (MB)	48	48	48
Registers available per block (MB)	32	32	64
CUDA compute capability	2.0	2.0	3.5

8 GB RAM, Intel(R) Xeon(R) CPU E5-2620 with 32 GB RAM, and Intel(R) Xeon(R) CPU E5-2650 with 32 GB RAM. Table 4 presents detailed information about the GPUs. The operating system are all Red Hat nash version 5.1.19.6.

4.1 Speedup Assessment

In order to assess the efficiency of CUDA-LINSi, we developed twenty-eight testsets to evaluate the speedup of CUDA-LINSi over sequential LINSi7.015. These testsets were constructed in two ways. One way is that sixteen testsets were constructed by randomly searching the NCBI non-redundant protein sequence database, selecting the sequences with length ranging from 30 to 1,000, and the number of sequences ranging from 100 to 800. These testsets were used to evaluate the speedup of Stage 1 of CUDA-LINSi, as shown in Fig. 10. The other way is that twelve testsets were generated by using PSI-BLAST to search the NCBI non-redundant protein sequence database for hits on Cadherin-related tumor suppressor (NCBI: P33450) and Zinc finger protein 2 (NCBI: P28167), selecting the highest-scoring 200 sequences, respectively. These sets of sequences have average length 4,896 and 2,907 residues, maximum length 5,277 and 3,907 residues, and average pair-wise identity 47.68 and 37.35 percent, respectively. We aligned randomly chosen subsets with 100 to 200 sequences and noted the speedup of Stage 4 of CDUA-LINSi, as shown in Fig. 11.

From Fig. 10, we can see that Stage 1 of CUDA-LINSi achieves speedup up to 56.7 on the testset with the average length 30. It should be noted that when the average length of sequences increases, the speedup decreases. This is because the space complexity of LINSi is at least $O(N^2) + O(L^2) + O(NL)$ but greatly depends on the similarity level [5]. When the sequence lengths increase, more memory are consumed, resulting in a severe performance bottleneck of CUDA-LINSi implementation.

From Fig. 11, we can see that Stage 4 of CUDA-LINSi achieves speedup up to 15.96 on the testset with the average length 4,896. The speedup of Stage 4 of CUDA-LINSi in fact increases with increasing number of sequences, while Stage 4 of multi-thread LINSi7.015 maintains a stable speedup. It

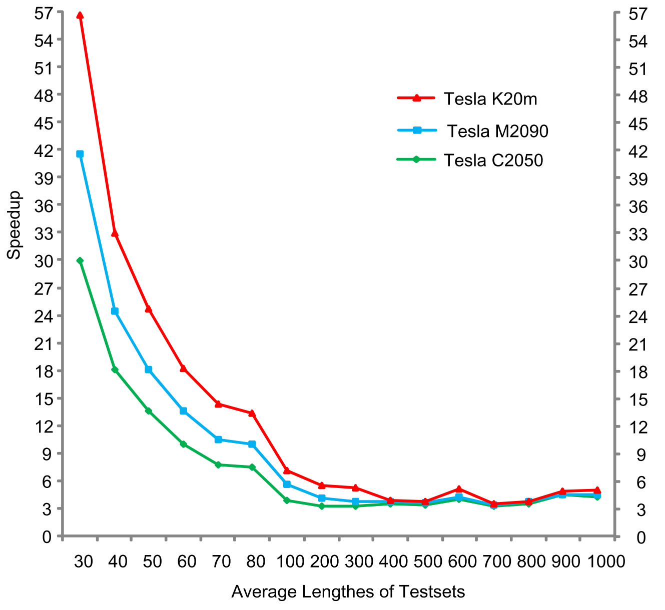


Fig. 10. Comparison of speedups (over the LINSi 7.015) of Stage 1 of CDUA-LINSi.

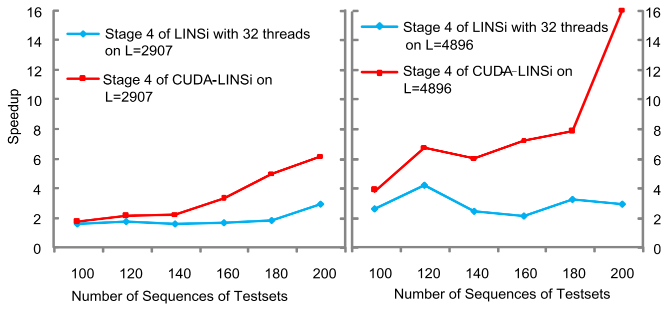


Fig. 11. Comparison of speedups (over the LINSi7.015) of Stage 4 of CDUA-LINSi and multi-thread LINSi7.015.

TABLE 5
Runtime Profiling (in Seconds) of Each Stage Of CUDA-LINSi

		Stage 1				Stage 4			
The Number of Sequences	Data Transformation	Kernel1 Runtime	Kernel1 Overhead	Total	Stages 2 and 3	Kernel2 Runtime	Kernel2 Overhead	Total	Speedup
100	1.23	58.92	0.56	59.48	19.60	115.96	3.24	119.20	5.53
120	1.41	75.78	0.88	76.66	22.93	140.53	4.46	144.99	6.97
140	1.60	98.06	1.04	99.10	27.76	208.53	5.87	214.40	6.47
160	1.92	132.45	1.39	133.84	32.96	238.07	7.69	245.76	7.18
180	2.11	169.85	1.81	171.66	35.64	256.53	10.02	266.55	7.40
200	2.38	211.25	2.82	214.07	44.02	279.06	13.16	292.22	11.28

TABLE 6
The Average Speedup of Multi-Thread LINSi7.015 over Sequential LINSi7.015

Average Lengthes of Testsets	Tesla C2050	Tesla M2090	Tesla K20m
30	2.25	2.29	2.17
40	2.94	3.10	2.78
50	2.30	2.29	2.17
60	2.67	2.72	2.76
70	3.22	3.30	3.26
80	2.58	2.73	2.73
100	2.61	2.78	3.13
200	2.97	3.12	2.72
300	2.93	2.61	2.50
400	2.87	2.67	2.53
500	2.60	2.32	2.68
600	3.08	2.98	3.02
700	2.49	2.20	2.32
800	2.42	2.73	2.68
900	3.53	3.47	3.90
1,000	3.47	3.51	3.95

effectively verifies the validity of GPU as an efficient computational platform to accelerate the MAFFT algorithm. It must be noted that we show the speedup for up to 200 sequences, because currently LINSi is recommended for alignment of less than 200 sequences with the maximum length 5,000.

In practice, data transfer between the CPU and GPU is a known bottleneck for many GPGPU applications. Data transfer, together with the kernel program initialization, kernel launch and release constitute the kernel overhead [21]. In addition, in CUDA-LINSi sequence data transformation needs to be done before data transfer from the CPU to GPU. Table 5 shows the runtime profiling of each stage in CUDA-LINSi and its overall speedup over sequential LINSi7.015. From Table 5, we can see that CUDA-LINSi achieves overall speedup up to 11.28 on the testset with the average length and the number of sequences 4,896 and 200, respectively. The runtime of sequence data transformation and the kernel overhead in CUDA-LINSi are very small and they are not a bottleneck for our implementation.

Table 6 reports the speedup of multi-thread LINSi7.015 over sequential LINSi7.015 running on the three servers mentioned above. From this table we can see that the speedup is nearly independent to the platforms which have different sizes of CPU and memory. Compared with multi-thread LINSi7.015, CUDA-LINSi is scalable when the number of sequences increases.

4.2 Quality Assessment

We used four benchmarks BAliBASE3.0, OXBench1.3, IRMBASE2.0, and PREFAB4.0 to assess the alignment accuracy of CUDA-LINSi in comparison with LINSi7.015, and other well-known sequential MSA algorithms: MSAProbs0.9.4, MUSCLE3.8.31, PicXAA with different options('-PF', '-PHMM', and '-SPHMM'), ProbCons1.12, ProbAlign1.1, MUMMALS1.01 with option HMM_1_3_1, T-Coffee6.00, CLUSTALW 2.0.10 , DIALIGN-TX, and CDAM. The scores of LINSi7.015, MSAProbs0.9.4, MUSCLE3.8.31, and CDAM were tested by us, while the scores of the others have been derived from [23]. Two accuracy measures were used to score the alignment, i.e., the quality score (Q), which is the number of correctly aligned residue pairs divided by the number of residue pairs in the reference alignment, and total column score (TC), which is the number of correctly aligned columns divided by the number of columns in the reference alignment [3].

Tables 7 shows the accuracy of different categories of BAliBASE3.0. Table 8 depicts the average accuracy of OXBench1.3, IRMBASE2.0, and PREFAB4.0. From these data we can see that CUDA-LINSi maintains identical accuracy to LINSi7.015, which effectively verifies the validity of LINISi implementation on GPUs. Further analysis of the alignment results indicates that MSAProbs yields the highest average Q and TC scores on BAliBASE3.0. This is because the design of MSAProbs is based on a combination of pair hidden Markov models and partition functions to calculate posterior probabilities [24]. PicXAA exhibits the best performance on OXBench1.3 in term of the average Q and TC scores. This suggests that construction of alignment from confidently alignable regions with high local similarities leads to more accurate results [25]. MUMMALS

TABLE 7
Comparison of Q/TC Scores of LINSi7.015 and CUDA-LINSi on BAliBASE3.0

Method	BAliBASE3.0						
	RV11 Q/TC	RV12 Q/TC	RV20 Q/TC	RV30 Q/TC	RV40 Q/TC	RV50 Q/TC	average Q/TC
CUDA-LINSi	0.653/0.431	0.936/0.843	0.925/0.451	0.859/0.585	0.915/0.578	0.901/0.599	0.859/0.579
MAFFT-LINSi	0.653/0.431	0.936/0.843	0.925/0.451	0.859/0.585	0.915/0.578	0.901/0.599	0.859/0.579
MSAProbs	0.682/0.444	0.946/**0.870**	**0.928**/**0.469**	**0.865**/**0.612**	0.923/0.610	**0.908**/**0.612**	0.878/**0.608**
PicXAA-PF	0.689/**0.462**	0.946/0.862	0.925/0.415	0.861/0.578	**0.932**/**0.633**	0.892/0.530	**0.879**/0.593
PicXAA-PHMM	0.663/0.420	0.942/0.858	0.917/0.388	0.850/0.530	0.909/0.562	0.902/0.602	0.866/0.563
PicXAA-SPHMM	**0.695**/0.447	**0.948**/0.864	0.917/0.403	0.841/0.530	0.891/0.523	0.898/0.584	0.867/0.561
ProbAlign	0.694/0.445	0.947/0.863	0.926/0.439	0.853/0.566	0.922/0.604	0.890/0.549	0.876/0.588
ProbCons	0.669/0.416	0.941/0.855	0.917/0.406	0.846/0.544	0.906/0.546	0.890/0.559	0.864/0.560
MUMMALS	0.670/0.416	0.943/0.840	0.910/0.428	0.848/0.494	0.872/0.486	0.879/0.529	0.855/0.539
MUSCLE	0.572/0.321	0.915/0.809	0.889/0.353	0.814/0.412	0.863/0.453	0.835/0.464	0.819/0.478
T-Coffee	0.660/0.414	0.941/0.853	0.915/0.387	0.837/0.495	0.897/0.551	0.894/0.581	0.859/0.552
DIALIGN-TX	0.515/0.265	0.892/0.752	0.879/0.305	0.762/0.385	0.836/0.448	0.823/0.466	0.788/0.443
CLUSTALW	0.494/0.240	0.871/0.719	0.862/0.235	0.720/0.269	0.786/0.400	0.734/0.304	0.754/0.380
CDAM	0.497/0.222	0.857/0.654	0.844/0.191	0.693/0.137	0.738/0.337	0.724/0.265	0.732/0.321

TABLE 8
Comparison of Q/TC Scores of LINSi7.015 and CUDA-LINSi on IRMBASE2.0, PREFAB4.0 and OXBench1.3

Method	IRMBASE 2.0	PREFAB4.0	OXBench1.3		
			master	full	extended
	Q/TC	Q	Q/TC	Q/TC	Q/TC
CUDA-LINSi	0.894/0.460	0.692	0.882/0.827	0.830/0.749	0.928/0.884
MAFFT-LINSi	0.894/0.460	0.692	0.882/0.827	0.830/0.749	0.928/0.884
MSAProbs	0.204/0.088	0.704	0.898/0.849	0.291/0.191	0.931/0.889
PicXAA-PF	0.890/0.501	0.713	0.897/0.847	**0.842/0.765**	0.924/0.881
PicXAA-PHMM	0.908/0.545	0.712	0.893/0.841	0.832/0.752	0.921/0.877
PicXAA-SPHMM	0.728/0.330	0.724	**0.906/0.857**	0.838/0.760	**0.930/0.889**
ProbAlign	0.817/0.367	0.719	0.898/0.849	0.841/0.764	0.926/0.884
ProbCons	0.853/0.425	0.716	0.893/0.841	0.833/0.752	0.924/0.882
MUMMALS	0.684/0.246	**0.727**	0.902/0.852	0.828/0.751	0.925/0.878
MUSCLE	0.114/0.038	0.650	0.892/0.840	0.288/0.188	0.916/0.865
T-Coffee	0.878/0.463	0.708	0.818/0.750	0.734/0.638	0.913/0.866
DIALIGN-TX	**0.929/0.710**	0.625	0.860/0.797	0.808/0.723	0.888/0.829
CLUSTALW	0.263/0.024	0.618	0.893/0.838	0.816/0.727	0.894/0.839
CDAM	0.098/0.028	0.584	0.885/0.826	0.287/0.188	0.880/0.816

outperforms other methods on PREFAB4.0 because it incorporates structural information into the process of alignment probabilities computation. DIALIGN-TX shows the best performance on IRMBASE2.0 which was originally developed to assess the performance on aligning sequences with local similarities.

4.3 Computational Complexity

To assess the computational complexity, we calculated the runtimes of CUDA-LINSi in comparison with MUSCLE3.8.31, CDAM, MSAProbs0.9.4, PicXAA with '-PHMM' option, Probcons, DIALIGN-TX, and MUMMALS. Using Rose sequence generator, we constructed ten testsets with each 200 sequences and the average distance between sequences ranging from 150 to 1,050. Fig. 12 depicts the runtimes of these algorithms on the ten testsets on the server the NVDIA Tesla C2050 locates. As demonstrated in the results, although PicXAA-PHMM, MSAProbs, and DIALIGN-TX achieve high accuracy on OXBench1.3, BAliBASE3.0, and IRMBASE2.0 respectively, they consume more time to calculate probabilities or similarities information in the process of alignment. MUMMALS with option HMM_1_3_1 is the most time-consuming method whose average runtime is 4281.00 seconds on the ten testsets. CDAM is the fastest algorithm while losing some accuracy. CUDA-LINSi achieves better balance between the accuracy and the runtime.

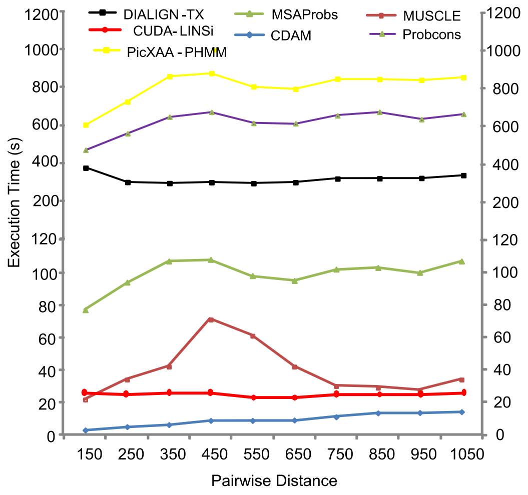


Fig. 12. Runtime comparison on different alignment algorithms.

4.4 Compression Rate of MRLE Assessment

We evaluate the performance of MRLE on the testsets used in Section 4.3. Table 9 shows the compression rate of MRLE. Column 2 is the lengthes of the longest two sequences of the testset. Noted that the compression rate of MRLE is dependent on the lengthes of the aligned sequences. From Table 9 we can see that the average compression rate of MRLE is around 38.00 percent.

5 Conclusion and Future Work

In this paper, we have presented a parallel CUDA-based algorithm for accelerating the LINSi option of MAFFT on a commodity GPU. In order to exploit the GPU's capabilities for accelerating the LINSi option, we have optimized the sequence data organization to eliminate the bandwidth bottlenecks of memory access, designed a memory allocation and reuse strategy to make full use of limited memory of GPUs, designed a constant time algorithm to schedule

TABLE 9
Compression Rate of MRLE

Ave. Distance	Max. Lengthes	Compression Rate		
		maximum	minimum	average
150	184×245	60.71%	28.94%	37.11%
250	198×210	55.31%	30.15%	38.05%
350	231×250	62.95%	31.42%	38.89%
450	228×310	58.44%	32.06%	39.41%
550	240×211	53.06%	29.68%	38.60%
650	215×277	59.76%	33.75%	36.97%
750	264×197	57.65%	28.70%	38.68%
850	311×268	50.08%	28.72%	39.83%
950	189×234	60.93%	29.55%	38.69%
1050	266×187	61.42%	27.59%	38.24%

sequence pairs, proposed a new MRLE scheme to reduce memory consumption, and used high-performance shared memory to speed up the I/O operations. Our implementation tested in three NVIDIA GPUs and achieves speedup up to 11.28 on a Tesla K20m GPU compared to the sequential MAFFT 7.015. To the best of our knowledge, this work embodies the first attempt to accelerate MAFFT application using GPU. Hence, our results are especially encouraging, sine performance of many-core architectures grows faster than the performance of standard multicore CPUs.

As future work, we intend to accelerate the other two time-consuming options of MAFFT, i.e., G-INS-i and E-INS-i, and extend tests to multiple graphics processors.

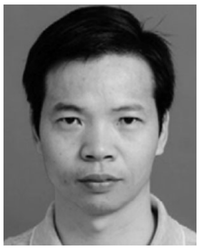
ACKNOWLEDGMENTS

A preliminary version of the paper [26] was presented on BIBM2013. The authors would like to thank Dr. Greg Ruthenbeck for his guidance on compress algorithm and Guanghui Li for helping to provide the experimental environments for conducting the tests of Stage 4. The authors would like to thank the anonymous reviewers for their helpful remarks. This research was partially funded by the Key Program of National Natural Science Foundation of China (Grant No. 61133005), the National Natural Science Foundation of China (Grant Nos. 90715029, 61402400, 61070057, 61370095, 61202109, 61300038), the National Science Foundation for Distinguished Young Scholars of Hunan (Grant No. 12JJ1011), and Hunan Provincial Natural Science Foundation of China (Grant No. 13JJB006).

REFERENCES

[1] R. C. Edgar and S. Batzoglou, "Multiple sequence alignment," *Current Opin. Struct. Biol.*, vol. 16, no. 3, pp. 368–373, 2006.
[2] J. D. Thompson, D. G. Higgins, and T. J. Gibson, "Clustal w: Improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-specific gap penalties and weight matrix choice," *Nucleic Acids Res.*, vol. 22, no. 22, pp. 4673–4680, 1994.
[3] R. C. Edgar, "Muscle: multiple sequence alignment with high accuracy and high throughput," *Nucleic Acids Res.*, vol. 32, no. 5, pp. 1792–1797, 2004.
[4] M. Roytberg, A. Gambin, L. Noé, S. Lasota, E. Furletova, E. Szczurek, and G. Kucherov, "On subset seeds for protein alignment," *IEEE/ACM Trans. Comput. Biol. Bioinform.*, vol. 6, no. 3, pp. 483–494, Jul. 2009.
[5] K. Katoh, K. Misawa, K.-I. Kuma, and T. Miyata, "Mafft: A novel method for rapid multiple sequence alignment based on fast fourier transform," *Nucleic Acids Res.*, vol. 30, no. 14, pp. 3059–3066, 2002.
[6] K. Katoh, K.-i. Kuma, H. Toh, and T. Miyata, "Mafft version 5: Improvement in accuracy of multiple sequence alignment," *Nucleic Acids Res.*, vol. 33, no. 2, pp. 511–518, 2005.
[7] K. Katoh and H. Toh, "Recent developments in the mafft multiple sequence alignment program," *Briefings Bioinformat.*, vol. 9, no. 4, pp. 286–298, 2008.
[8] K. Kazutaka and T. Hiroyuki, "Parallelization of the mafft multiple sequence alignment program," *Bioinformatics*, vol. 26, no. 15, pp. 1899–1900, 2010.
[9] K. Katoh and C. M. Frith, "Adding unaligned sequences into an existing alignment using mafft and last," *Bioinformatics Appl. Note*, vol. 28, no. 23, pp. 3144–3146, 2012.
[10] M. C. Schatz, C. Trapnell, A. L. Delcher, and A. Varshney, "High-throughput sequence alignment using graphics processing units," *BMC Bioinformat.*, vol. 8, no. 1, p. 474, 2007.
[11] C. Trapnell and M. C. Schatz, "Optimizing data intensive gpgpu computations for dna sequence alignment," *Parallel Comput.*, vol. 35, no. 8, pp. 429–440, 2009.
[12] S. A. Manavski and G. Valle, "Cuda compatible gpu cards as efficient hardware accelerators for smith-waterman sequence alignment," *BMC Bioinformat.*, vol. 9, no. suppl. 2, p. S10, 2008.
[13] A. Khajeh-Saeed, S. Poole, and J. Blair Perot, "Acceleration of the smith–waterman algorithm using single and multiple graphics processors," *J. Comput. Phys.*, vol. 229, no. 11, pp. 4247–4258, 2010.
[14] A. Akoglu and G. M. Striemer, "Scalable and highly parallel implementation of smith-waterman on graphics processing unit using cuda," *Cluster Comput.*, vol. 12, no. 3, pp. 341–352, 2009.
[15] Y. Liu, D. L. Maskell, and B. Schmidt, "Cudasw++: Optimizing smith-waterman sequence database searches for cuda-enabled graphics processing units," *BMC Res. Notes*, vol. 2, no. 1, p. 73, 2009.
[16] Y. Liu, B. Schmidt, and D. L. Maskell, "Cudasw++ 2.0: Enhanced smith-waterman protein database search on cuda-enabled gpus based on simt and virtualized simd abstractions," *BMC Res. Notes*, vol. 3, no. 1, p. 93, 2010.
[17] L. Hasan, M. Kentie, and Z. Al-Ars, "Dopa: GPU-based protein alignment using database and memory access optimizations," *BMC Res. Notes*, vol. 4, no. 1, p. 261, 2011.
[18] E. F. O. Sandes and A. C. de Melo, "Cudalign: Using gpu to accelerate the comparison of megabase genomic sequences," in *Proc. 15th ACM SIGPLAN Symp. Principles Pract. Parallel Program.*, vol. 45, no. 5, pp. 137–146, 2010.
[19] E. Sandes and A. de Melo, "Retrieving smith-waterman alignments with optimizations for megabase biological sequences using gpu," *IEEE Trans. Parallel Distrib. Syst.*, vol. 24, no. 5, pp. 1009–1021, Jul. 2013.
[20] P. D. Vouzis and N. V. Sahinidis, "Gpu-blast: Using graphics processors to accelerate protein sequence alignment," *Bioinformatics*, vol. 27, no. 2, pp. 182–188, 2011.
[21] W. Liu, B. Schmidt, and W. Muller-Wittig, "Cuda-blastp: Accelerating blastp on cuda-enabled graphics hardware," *IEEE/ACM Trans. Comput. Biol. Bioinform.*, vol. 8, no. 6, pp. 1678–1684, Nov. 2011.
[22] S. Widmer, D. Wodniok, N. Weber, and M. Goesele, "Fast dynamic memory allocator for massively parallel architectures," in *Proc. 6th Workshop Gen. Purpose Process. Using Graph. Process. Units*, 2013, pp. 120–126.
[23] S. M. E. Sahraeian and B.-J. Yoon, "Picxaa: Greedy probabilistic construction of maximum expected accuracy alignment of multiple sequences," *Nucleic Acids Res.*, vol. 38, no. 15, pp. 4917–4928, 2010.
[24] Y. Liu, B. Schmidt, and D. L. Maskell, "Msaprobs: Multiple sequence alignment based on pair hidden markov models and partition function posterior probabilities," *Bioinformatics*, vol. 26, no. 16, pp. 1958–1964, 2010.
[25] X. Zhu, K. Li, and A. Salah, "A data parallel strategy for aligning multiple biological sequences on multi-core computers," *Comput. Biol. Med.*, vol. 43, pp. 350–361, 2013.
[26] X. Zhu and K. Li, "Cuda-mafft: Accelerating mafft on cuda-enabled graphics hardware," in *Proc. IEEE Int. Conf. Bioinformat. Biomed.*, 2013, pp. 486–489.

Xiangyuan Zhu received the MS and PhD degrees in computer science from the College of Information Science and Engineering at Hunan University, Changsha, China, in 2006 and 2014, respectively. She is currently an assistant professor of computer science and technology at Zhaoqing University, Zhaoqing, China. Her major research interests includes parallel computing, parallel scheduling, and algorithms designing in bioinformatics.



Kenli Li received the PhD degree in computer science from Huazhong University of Science and Technology, Wuhan, China, in 2003. He is currently a full professor of computer science and technology at Hunan University, Changsha, China, and the deputy director of National Supercomputing Center in Changsha. His major research interests include parallel computing, grid and cloud computing, and DNA computing. He has published more than 90 papers in international conferences and journals. He is a member of the IEEE.

Ahmad Salah received the MS degree in computer science from Ain Sahmas University, Cairo, Egypt, in 2009. Since 2011, he has been working toward the PhD degree at Hunan University, Changsha, China. His current research interests include parallel algorithms, computational biology and mathematics.

Lin Shi received the PhD degree in computer science from College of Information Science and Engineering at Hunan University, Changsha, China, in 2012. His current research interests include virtual machines and GPGPU computing. He is a member of the IEEE.

Keqin Li is currently a SUNY distinguished professor of computer science and an Intellectual Ventures endowed visiting chair professor at Tsinghua University, Tsinghua, China. His current research interests include the design and analysis of algorithms, parallel and distributed computing, and computer networking. He has nearly 300 research publications. He is currently on the editorial boards of the *IEEE Transactions on Computers* and the *IEEE Transactions on Cloud Computing*. He is a senior member of the IEEE.