CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2015; **27**:3403–3414
Published online 05 August 2014 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/cpe.3355

SPECIAL ISSUE PAPER

GPU-UPGMA: high-performance computing for UPGMA algorithm based on graphics processing units

Yu-Shiang Lin[1], Chun-Yuan Lin[2], Che-Lun Hung[3,*,†], Yeh-Ching Chung[1] and Kual-Zheng Lee[4]

[1]*Department of Computer Science, National Tsing Hua University, No. 101, Section 2, Kuang-Fu Road, Hsinchu City 30013, Taiwan*
[2]*Department of Computer Science and Information Engineering, Chang Gung University, No. 259, Sanmin Rd., Guishan Township, Taoyuan City 33302, Taiwan*
[3]*Department of Computer Science and Communication Engineering, Providence University, No. 200, Sec. 7, Taiwan Boulevard, Shalu Dist., Taichung City 43301, Taiwan*
[4]*Information and Communications Research Laboratories, Industrial Technology Rearch Institute, No. 195, Sec. 4, Chung Hsing Rd., Chutung, Hsinchu City 31040, Taiwan*

SUMMARY

Constructing phylogenetic trees is of priority concern in computational biology, especially for developing biological taxonomies. As a conventional means of constructing phylogenetic trees, unweighted pair group method with arithmetic (UPGMA) is also an extensively adopted heuristic algorithm for constructing ultrametric trees (UT). Although the UT constructed by UPGMA is often not a true tree unless the molecular clock assumption holds, UT is still useful for the clocklike data. Moreover, UT has been successfully adopted in other problems, including orthologous-domain classification and multiple sequence alignment. However, previous implementations of the UPGMA method have a limited ability to handle large taxa sets efficiently. This work describes a novel graphics processing unit (GPU)-UPGMA approach, capable of providing rapid construction of extremely large datasets for biologists. Experimental results indicate that the proposed GPU-UPGMA approach achieves an approximately 95× speedup ratio on NVIDIA Tesla C2050 GPU over the implementation with 2.13 GHz CPU. The developed techniques in GPU-UPGMA also can be applied to solve the classification problem for large data set with more than tens of thousands items in the future.Copyright © 2014 John Wiley & Sons, Ltd.

Received 19 June 2013; Revised 7 April 2014; Accepted 6 July 2014

KEY WORDS: UPGMA; phylogenetic tree construction; graphics processing units; distance matrix; CUDA

1. INTRODUCTION

Constructing phylogenetic trees is of priority concern in areas such as computational biology, viral research, and biomedicine. Phylogenetic tree construction methods are either character-based or distance-based. The character-based method is based on a variety of phylogenetic characters such as DNA or protein sequences, directly aligned characters during tree inference rather than on pairwise distances. It is generally more complex than distance-based method when algorithms used to create phylogenetic trees [1]. The distanced-based method is based on the distances involving the differences between the pairs of sequences. The distance matrix can be constructed using distances

*Correspondence to: Che-Lun Hung, Department of Computer Science and Communication Engineering, Providence University, No. 200, Sec. 7, Taiwan Boulevard, Shalu Dist., Taichung City 43301, Taiwan.
†E-mail: clhung@pu.edu.tw

between the pairs of sequences, and then, a phylogenetic tree can be built using the distance matrix to identify the evolutionary relationship between sequences. The distance-based method uses two algorithms: cluster-based and optimality-based ones. Of those, cluster-based algorithms are used to construct a phylogenetic tree based on a distance matrix starting from the most similar sequence pairs. This work focused on the distanced-based method with cluster-based algorithms.

In the early stage, the conventionally used model assumes that the evolution rate remains constant [2, 3] (called molecular clock hypothesis [4]). Based on this assumption, the phylogenetic tree is an ultrametric trees (UT), which is a rooted, leaf labeled, and edge-weighted binary tree. In a UT, an internal node has the same path length to each leave in its subtree. According to the literature [5], only binary trees need to be considered because a general UT can be easily converted into a binary tree without modifying the distances between leaves. Further examples of UTs can be found in the literatures [6–9]. Because many of these problems are intractable and NP-hard, biologists typically construct the trees by using heuristic algorithms. Unweighted pair group method with arithmetic mean (UPGMA, e.g., Sneath and Sokal [10]) and neighbor-joining (NJ, e.g., Saitou and Nei [11]) are both conventional hierarchical clustering algorithms. UPGMA constructs a phylogenetic tree, which is a UT. Although the UT constructed by UPGMA is often not a true tree unless the molecular clock assumption holds, UT is still useful for the clocklike data and, thus, has been compared with other methods [12, 13]. Moreover, UT has been successfully applied to other problems, including orthologous-domain classification [14] and multiple sequence alignment (ex. CLUSTAL [15], MUSCLE [16], and M-Coffee [17]). For instance, multiple sequence alignment algorithms generally consist of three phases. Phase 1 calculates a distance matrix that consists of the distance values between each pair of input sequences (by using pairwise alignment). Phase 2 generates a guide tree for progressive alignments, which is typically obtained by a simple distance-based method, such as NJ or UPGMA. Phase 3 performs progressive alignments based on a generated guide tree. However, with the increasing number of input sequences, especially for large-scale gene/protein analysis among species, the second stage can dominate the overall execution time. Therefore, some approaches have been devised to accelerate the computational time of phase 2 [16,18]. UPGMA has a naively $O(n^3)$ time complexity, where n denotes the number of operational taxonomic units (OTUs), which can be reduced to an optimal $O(n^2)$ time complexity [16]. NJ provides another phylogenetic tree construction method, which does not impose a molecular clock with an unrooted tree. NJ has a naively $O(n^4)$ time complexity, and it was modified by Studier and Keppler [18], which runs in $O(n^3)$ time complexity.

Many studies have attempted to solve the problem of high-computational complexity by parallelization in recent years. ClustalW-MPI [19] is a distributed and parallel implementation of ClustalW [20]. All three phases including the guide tree generation have been parallelized to reduce the computational time. In the 2006, Du and Feng [21] developed a parallel algorithm, pNJTree, to implement the NJ method by using MPI. Their experimental results indicated that on a large taxa set of 10,000 sequences, pNJTree only require 2888 s on 32 processors, which is faster than ClustalW-MPI; the ClustalW-MPI takes 25,418 s to construct the NJ tree on 32 processors. Yu *et al.* proposed a parallel branch-and-bound algorithm, PBBU, to solving the minimum UT construction problem, which is an NP-hard problem by using MPI in the 2009 [22]. The experimental results show that the PBBU found an optimal solution for 36 sequences on 16 processors within 1 day.

Current high-end graphics processing units (GPUs), which contain up to hundreds of cores per-chip, are widely used in the high-performance computing community. As a massively multithreaded processor, GPU expects the thousands of concurrent threads to fully utilize its computational power. The ease of access GPUs by using general-purpose computing on GPUs such as Open Computing Language [23] and compute unified device architecture (CUDA) [24], as opposed to graphic APIs, has made supercomputing available widely. In our work, we choose CUDA to use a new computing architecture referred to as single instruction multiple threads, which differs from the Flynn's classification [25]. Importantly, the computational power and memory bandwidth for modern GPUs have made porting applications possible. In computational biology, several algorithms or approaches have been ported on GPUs with CUDA, including MSA-CUDA [26], CUDA-MEME [27], CUDA-BLASTP [28, 29], phylogenetic algorithm [30, 31], and Smith–Waterman algorithm [32–39]. For constructing a phylogenetic tree, Liu *et al.* [30] implemented the GPU version of NJ algorithm.

DOI: 10.1002/cpe

Their experimental results indicate that their implementation can achieve a speedup ratio of 26× for datasets with more than 10,000 sequences. Liu *et al*. proposed the Bioinformatics Cell, BiCell, which is a programmable and scalable architectural platform, to accelerate the maximum likelihood method in the phylogenetic tree construction problem [31]. To our knowledge, no GPU version of parallel UPGMA algorithm has been developed. One algorithm proposed by Sreesa and Davis has improved the computational time of UPGMA, which is implemented on an FPGA-based platform [40].

Therefore, this work designs and implements a GPU-based UPGMA approach for the phylogenetic tree construction problem with CUDA, referred to herein as GPU-UPGMA v1.0. According to the computational time of each step in sequential UPGMA (SUPGMA) (an implementation of sequential UPGMA algorithm; Table I in Section 5), the find minimum value is the most time-consuming step (i.e., $>98\%$ overall computational time of SUPGMA). The find minimum value step identifies the shortest distance in a distance matrix. Because the computations of find minimum value step are independent; this step is feasible for parallel computing. Hence, a GPU implementation of the parallel tree reduction (PTR) method for finding the minimum value (GPTR-M) is designed in GPU-UPGMA v1.0 approach at first. GPTR-M in GPU-UPGMA v1.0 achieves a speedup ratio of 97× for finding the minimum values over the SUPGMA algorithm. Then, a simple implementation of update distance matrix step on GPU also is presented in GPU-UPGMA v1.0 approach to speed the computational time of step 2 and overlap the computation time of build phylogenetic tree step by CPU. Finally, the GPU-UPGMA v1.0 achieves a speedup ratio of 95× for the overall computational time over the SUPGMA algorithm.

The rest of this paper is organized as follows. Section 2 briefly describes the preliminary concepts for CUDA programming model and UPGMA algorithm. Section 3 then introduces the method of solving tree reduction problem on GPU and implements the GPU-UPGMA v1.0 approach. Next, Section 4 summarizes the experimental results. Conclusions are finally drawn in Section 5, along with recommendations for future research.

2. PRELIMINARY CONCEPTS

2.1. CUDA programming model (CUDA 3.2)

The CUDA is an extension of C/C++, in which users can write scalable multithreaded programs for GPU computing field [24]. The CUDA program is implemented in two parts: host and device. The host is executed by CPU, and the device is executed by GPU. The function executed on the device called a *kernel* (called *KF* for short). The *KF* can be invoked as a set of concurrently executing threads, and it is executed by *threads* (called *Td* for short). These *Tds* are in a hierarchical organization that can be combined into *thread blocks* (called *Tb* for short) and *grids* (called *Gd* for short). A *Gd* is a set of independent *Tbs*, and a *Tb* contains many *Tds*. The size of *Gd* is the number of *Tbs* per-grid, and the size of *Tb* is the number of *Tds* per-block. *Tds* in a *Tb* can communicate and synchronize with each other. *Tds* within a *Tb* can communicate through a per-block *shared*

Table I. Computational time for each stage of SUPGMA algorithm based on the naively UPGMA method, where the number of OTUs is *N*, which ranges from 1000 to 10,000. The unit is expressed in microsecond (ms).

	Stage 1: Find minimum	Stage 2: Update	Stage 3: Build tree	Total
$N=1000$	4592.29	54.52	0.06	4646.87
$N=2000$	36,419.66	279.62	0.12	36,699.40
$N=3000$	122,655.30	706.63	0.16	123,362.09
$N=4000$	288,351.31	1338.24	0.22	289,689.77
$N=5000$	565,168.87	2133.56	0.27	567,302.70
$N=6000$	975,664.62	3238.02	0.33	978,902.97
$N=7000$	1,549,718.62	4741.40	0.38	1,554,460.40
$N=8000$	2,314,399.25	6756.07	0.44	2,321,155.76
$N=9000$	3,305,549.50	9056.85	0.50	3,314,606.85
$N=10{,}000$	4,532,068.00	11,962.72	0.55	4,544,031.27

DOI: 10.1002/cpe

memory (called *SM* for short), whereas *Tds* in different *Tbs* fail to communicate or synchronize directly. Besides *SM*, four memory types are per-thread private *local memor*y (called *LM* for short), *global memory* (called *GM* for short) for data shared by all *Tds*, *texture memory* (called *TM* for short), and *constant memory* (called *CM* for short). Of these memory types, *CM* and *TM* can be regarded as fast read-only caches; the fastest memories are the *registers* (called *RG* for short) and *SM*. The *GM*, *LM*, *TM*, and *CM* are located on the GPU's memory. Besides *SM* accessed by single *Tb* and *RG* only accessed by a single *Td*, the other memory can be used by all of the *Tds*. The caches of *TM* and *CM* are limited to 8 KB per-*streaming multiprocessor* (called *STM* for short). The optimum access strategy for *CM* is all *Tds* reading the same memory address. The cache of *TM* is designed for *Tds* to read between the proximity of the address in order to achieve an improved reading efficiency. The basic processing unit in NVIDIA's GPU architecture is called the *streaming processor* (called *STP* for short). Many *STPs* perform the computation on GPU. Several *STPs* can be integrated into a *STM*. While the program runs the *KF*, the GPU device schedules *Tbs* for execution on the *STM*. The single instruction multiple thread scheme refers to *Tds* running on the *STM* in a small group of 32, called a *warp* (called *WP* for short). The *WP* scheduler simultaneously schedules and dispatches instructions. For instance, NVIDIA GeForce GTX 260, each *STM* with 16,384 32-bit *RG*s has 16 KB of *SM*. The *RGs* and *SM* used in a *Tb* affect the number of *Tbs* assigned to the *STM*. *STM* can be assigned up to eight *Tbs*. In a relatively new GPUs with higher compute capability versions, there would be more hardware resource support, such as more *STPs*, more memory space, more *WP* schedulers, and have real configurable L1 and unified L2 caches.

2.2. *Naively UPGMA algorithm*

As a heuristic algorithm for constructing a phylogenetic tree by a distance matrix, UPGMA is a progressive clustering method [41]. UPGMA first identifies the shortest distance between two OTUs, these two OTUs are then combined into a new composite OTU, and finally, the closet distances between the composite OTU with the other OTUs are calculated. This process is repeated until a single (composite) OTU remains. By tracing the content of composite OUT, the phylogenetic tree can be constructed. UPGMA must assume that the variability generation and evolution time are a positive correlation, that is, the molecular clock exists. According to the calculation principles of UPGMA, the internal node in the UT has the same distance to each leave in its subtree. In this work, the distances between the OTUs C_i, C_j from the individual distances d_{pq} are maintained as

$$d_{ij} = \frac{1}{|C_i||C_j|} \sum_{p \in C_i} \sum_{q \in C_j} d_{pq} \tag{1}$$

where $|C_i|$ and $|C_j|$ denote the number of sequences in the two OTUs C_i and C_j, respectively. The pseudo code involved in implementing the naively UPGMA algorithm is written as follows:

```
//initialization:
//Assign each sequence its own OTU C_x.
//Each sequence is placed at the UT of height zero, where defines one leaf.
  While (the_number_of_remain_ OTUs == 2)
  {
      step 1. Find two clusters C_i, C_j, in which d_ij is minimal. If several of the same values are
              chosen, select one randomly.
      step 2. Define a new OTU C_k = C_i U C_j , and, then, calculate the distance d_kl between
              OTU C_k and other OTUs C_l by d_kl = (d_il|C_i| + d_jl|C_j|) / (|C_i| + |C_j|)
      step 3. Place an internal node k with leaves i and j, and, then, assign d_ij /2 to the height of
              internal node k.
      step 4. Add C_k to the current OTU and remove OTUs C_i and C_j.
  }
```


DOI: 10.1002/cpe

3. SUPGMA METHOD

3.1. Implementation of sequential UPGMA

The SUPGMA algorithm is implemented based on the naively UPGMA method (Section 2.2) and largely complies with the UPGMA algorithm processes. The SUPGMA algorithm is divided into three stages (with an example shown in Figure 1):

Stage 1: Find minimum value. Find the minimum value from the distance matrix and, then, assign a corresponding index represented by (row i, column j) from the input distance matrix, where i and j range from integer zero to n-1, respectively, and n refers to the number of OTUs. Each array element value of input distance matrix is generated by using a pairwise sequence alignment for a pair of sequences i and j.

Stage 2: Update distance matrix. Update the input distance matrix by re-calculating the values of array elements in original row i and column j, after the minimum value d_{ij} with the index (i, j) found by Stage 1. These values of array elements in original row i and column j are recalculated to the new distance values by the naively UPGMA algorithm, and then, the values of array elements in column i are replaced by those in column j. Finally, the values of array elements in original row j and column i are set to zero to perform the matrix reduction (these array elements are omitted in the following repeats). According to the previous method, a part of these computations earlier may be unnecessary under the symmetry of distance matrix. Despite the many methods available to perform this stage in order to reduce the computation, this work selects this method to simplify the data structure implementation and avoid the complex conversion of data structure. In this work, the SUPGMA algorithm is not an optimal UPGMA algorithm, and the goal of this work is not to compare the GPU-UPGMA v1.0 with the optimal UPGMA algorithm. The contribution of this work is to present an efficient GPU implementation of UPGMA and prove the benefit of GPU-UPGMA v1.0 on GPUs with CUDA.

Stage 3: Build phylogenetic tree. Create a branch in a phylogenetic tree by combining node i and node j. These two nodes are the leaves of the branch, and their root is a newly created node (new OTU) k. Indices i and j are pushed to a stack A, and index k is pushed to another stack B. To repeat the previous three stages about n-1 times until the phylogenetic tree is completed, the phylogenetic tree is represented by tracing the stack A and stack B.

3.2. Time complexity of the SUPGMA method

Assume that a distance matrix with n sequences implies that the size of the distance matrix is $n \times n$. In Stage 1, the fact that the distance matrix is a symmetric matrix implies that the minimum value can be obtained from the lower-triangular or upper-triangular matrix. Therefore, a time complexity $O((n^2 - n)/2) = O(n^2)$ is required to find the minimum value from the distance matrix in the first iteration, where n refers to the number of OTUs. Stage 2 runs in a time complexity $O(4n)$ in the first iteration, because two rows and two columns must be recalculated and replaced. Given that the total iteration times are n-1, for SUPGMA algorithm, the total time complexity of Stage 1 is $O(n^3)$, Stage 2

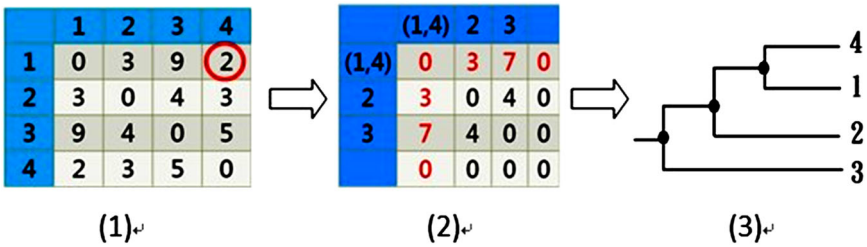

Figure 1. Three stages of SUPGMA: (1) find minimum value, (2) update distance matrix, and (3) build phylogenetic tree.

DOI: 10.1002/cpe

is O(n^2), and Stage 3 is O(n). The time complexity of each stage reveals that the bottleneck of SUPGMA algorithm for large-scale data occurs on Stage 1, which attempts to find the minimum value from the distance matrix repeatedly.

4. GPU-UPGMA V1.0

According to Section 3.2, Stage 1 is the most time-consuming stage in SUPGMA algorithm to find the minimum value from the input distance matrix. Many methods are available to obtain the minimum value from an array sequentially or in parallel. The tree reduction method (or matrix reduction method) by using GPU has received considerable attention in recent years. For instance, Roger *et al.* [42] developed a reduction primitive method on GPU by using OpenGL shading language. Harris [43] also developed a GPU version of sum-reduction primitive under CUDA architecture. Meanwhile, Wang *et al.* [44] designed and implemented a reduction primitive of GPU-based Prim's algorithm, in which a min-reduction method was proposed as well. This work extends upon the min-reduction method to improve Stage 1 in SUPGMA algorithm.

This work attempts to operate an array by using a parallel approach, called the PTR method. The PTR process aims to halve the number of working threads from the previous reduction iteration, which operates two values of a distance matrix to be one per-thread. This process is repeated recursively until the operating results of distance matrix are derived. Notably, N array elements in the distance matrix take log$N$ steps and require N/2 threads. In this work, a GPU implementation of the PTR method for finding the minimum value (GPTR-M) has been designed in GPU-UPGMA v1.0 approach.

4.1. Stage 1-GPTR-M in GPU-UPGMA v1.0

In this work, two *KFs* are launched (shown in Figure 2) because no efficient method can perform the synchronization among the *Tbs* in the same *Gd*. This is despite the fact that output data from a *KF* can be stored in the *GM* to perform the synchronization among multiple *Tbs* in different *Gds*. Here, a variable *NTG* denotes the number of *Tds* per-grid; in addition, the values of lower-triangular or upper-triangular distance matrix from OTUs data are placed in an array *A* stored in *GM*. The array with a feasible minimum result, called *FMINR*, and its corresponding index array (called *FMINRi*)

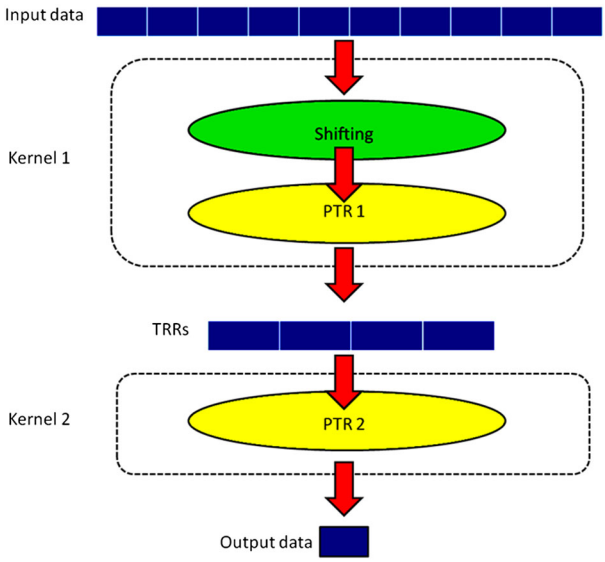

Figure 2. The GPTR-M module, in which two *KFs* are launched; *Kernel* 1 executes the shifting process and parallel tree reduction 1 (PTR 1) method and, then, produces the temporary reduction results (TRRs) as the input for *Kernel* 2, which executes the PTR 2 method.

DOI: 10.1002/cpe

are both stored in the *GM*. The length of array *A* is l_n, and the lengths of *FMINR* and *FMINRi* are *NTG*. In the GPTR-M, finding the minimum value of distance matrix consists of three steps.

Step 1. If $l_n > NTG$, then perform the shifting processes in an array *A*, which takes the values of address from $NTG * p$ to $NTG(1+p)$ in *FMINR* and *FMINRi* arrays, where *p* denotes a zero or positive integer. Each *Td* per-grid loads the recent value and its index from an array *A* at first, then loads the previously stored value and its index from arrays *FMINR* and *FMINRi*, and finally, compares the recent value with previously stored value and stores the smaller value and its index into arrays *FMINR* and *FMINRi*. All *Tds* accessing data are coalesced in the *GM* of GPU. If the loading range of an array *A* exceeds l_n, then the remaining *Tds* do not pick up any value and index from an array *A*. Figure 3 shows the shifting processes with total $\lceil \frac{l_n}{NTG} \rceil$-1 times, and each shifting process is used to compare with the values of a segment of an array *A* repeatedly until all values of an array *A* over scan. Because each shifting process for comparing the values by each *Td* per-grid in parallel, the final *FMINR* and *FMINRi* take only $\lceil \frac{l_n}{NTG} \rceil$-1 times by the effort of parallel comparison. The shifting processes reduce the input data, which are the size of an array *A* and index array to that of arrays *FMINR* and *FMINRi*. Moreover, the length is reduced from l_n to *NTG* in *Kernel* 1, as shown in Figure 2.

Step 2. Each *Td* per-block loads the value and index of arrays *FMINR* and *FMINRi* from the *GM* to *SM*. Following the synchronization of all data loaded from the *GM* to *SM* by each *Td*, all *Tds* of each *Tb* perform the PTR1 process and obtain the results, which are a local minimum value and its index in *SM*. Because the life time of *SM* is in a *Tb* and the life time of *GM* is in the whole program, these values and their indices in the *SM* for all *Tbs* must be stored in the *GM*. The *GM* stores the values and indices into corresponding addresses according to the *Tb* ID, to avoid losing values after completing the *KF*, *Kernel* 1. The proposed GPU-UPGMA v1.0 approach uses the cudaThreadSynchronize(), an API function of CUDA runtime library (see the NVIDIA programming guide), after the statement of *Kernel* 1 function call, to ensure that *Kernel* 1 completes its copy from the *SM* to *GM*, which produces the temporary reduction results (TRRs) among all *Tbs*.

Step 3. If the number of *Tbs* is larger than one, *Kernel* 2 must be used to find the global minimum value. Here, the TRRs generated from *Kernel* 1 are used as the input of *Kernel* 2, and the PTR 2 process is implemented. In the experimental tests, the number of *Tbs* per-grid (*NBG*) and the number of *Tds* per-block (*NTB*) are the same. In the TRRs, which include *FMINR* and *FMINRi*, their sizes are the same as the *NTB*. Therefore, in *Kernel* 2, all *Tds* of one *Tb* load values and indices of arrays *FMINR* and *FMINRi* from the *GM* can be placed in the *SM*. Next, the PTR 2 process is performed to find the (global) minimum value and its corresponding index value in the *SM*. Finally, the minimum value and its index from the *SM* are placed in the *GM*, followed by transferring them to the host memory.

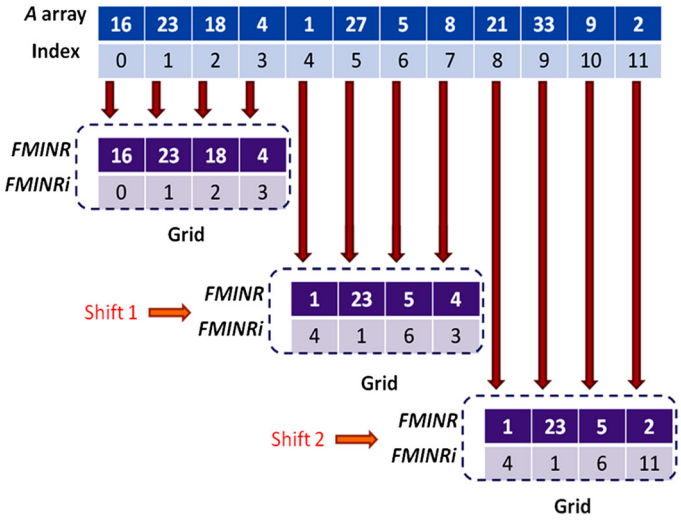

Figure 3. Shifting process to obtain the feasible minimum results and its corresponding index.

DOI: 10.1002/cpe

4.2. Stages 2 and 3-update distance matrix and build phylogenetic tree in GPU-UPGMA v1.0

After the minimum value and its corresponding index are derived by GPTR-M, the distance matrix is updated in Stage 2 and the updated row i and column j are set based on the corresponding index of minimum value. In GPU-UPGMA v1.0, Stage 2 attempts to recalculate and replace the values of array elements in rows (i and j) and columns (i and j) by using $4n$ *Tds* to update them in a *KF*, where n refers to the number of OTUs. This *KF* requires proper quantity *Tbs* to reduce the useless synchronization until out of the kernel launch. Therefore, this work designs an adjustable method for setting the variable *NBG*. When $n \leq 1024$, where the number 1024 denotes the maximum *NTB* set in the experimental GPU device, only four *Tbs* can supply $4n$ *Tds* to update distance matrix. Additionally, if $n > 1024$, then the used *NTG* is larger than $4 * 1024$, and the *NBG* is set up to $\left\lceil \frac{NTG}{1024} \right\rceil$. Similarly, after the minimum value and its corresponding index are derived by GPTR-M, in GPU-UPGMA v1.0, Stage 3 is used to build the phylogenetic tree on CPU by tracing the corresponding index of minimum value transferred from the *GM* on GPU to the host memory. Stage 3 in GPU-UPGMA v1.0 is similar to Stage 3 of SUPGMA algorithm. However, the computational times of Stage 2 and Stage 3 in GPU-UPGMA v1.0 can be overlapped with each other.

5. EXPERIMENTAL RESULTS

In this work, GPU-UPGMA v1.0 is implemented on a single NVIDIA Tesla C2050, with 448 *STP* cores and 3 GB GDDR3 RAM; the setting ranges from 64 to 512 *Tbs* (*NBG*), each with 64–512 *Tds* (*NTB*). The host (CPU) is Intel Xeon E5506 2.13 GHz with 8 GB RAM running the Linux operation system (Ubuntu 9.04 64-bit). The input distance matrices are randomly generated according to the number of OTUs. Testing datasets range from 1000 to 10,000 sequences, and the values of testing datasets range from 1 to 999.9. In the following experiment results, for SUPGMA algorithm and GPU-UPGMA v1.0, the computational times of find minimum value stage, update distance matrix stage, and build phylogenetic tree stage are estimated by sum of each iterative loop.

5.1. The computational time of SUPGMA algorithm

The computational time for each stage of SUPGMA algorithm is observed in Table I. From Table I, the computational time of find minimum value stage (Stage 1) is the most time-consuming step, especially for the large number of OTUs. It occupied about 98.86–99.7% of overall computational time of SUPGMA algorithm. Hence, this work is mainly focused on accelerating the computational time of find minimum value stage in GPU-UPGMA v1.0. Besides, the computational time of update distance matrix stage (Stage 2) increases greatly when the number of OTUs increases. Therefore, it still is worth to implement the update distance matrix stage on GPU in GPU-UPGMA v1.0. The build phylogenetic tree stage in GPU-UPGMA v1.0 is implemented on CPU according to the results (Stage 3) of Table I.

5.2. SUPGMA algorithm vs. GPU-UPGMA v1.0

In the find minimum value stage, Figure 4 shows the speedup ratios by comparing the computational time in GPU-UPGMA v1.0 with that in the SUPGMA algorithm for different *NTG* cases. From Figure 4, the speedup ratios are influenced by the variable *NTG*, and the speedup ratio increases when the size of *NTG* increases. The best result is occurred when the size of *NTG* is set to $512 * 512$, and GPTR-M in GPU-UPGMA v1.0 achieves a speedup ratio of 97× over the SUPGMA algorithm. These results in Figure 4 show that the GPTR-M is an efficient method to find the minimum value in GPU-UPGMA v1.0.

Figure 5 shows the speedup ratios by comparing the computational time of update distance matrix stage in GPU-UPGMA v1.0 with that in the SUPGMA algorithm for different number of sequences. From Figure 5, *NBG* and *NTG* are 64, the speedup ratio increases when the number of sequences increases, and this is close to linear growth in the experimental test. The update distance matrix stage in GPU-UPGMA v1.0 achieves a speedup ratio of 23× over the SUPGMA algorithm with 10,000 input sequences. These results are helpful to improve the overall computational time of GPU-UPGMA v1.0 when the number of sequences is large.

DOI: 10.1002/cpe

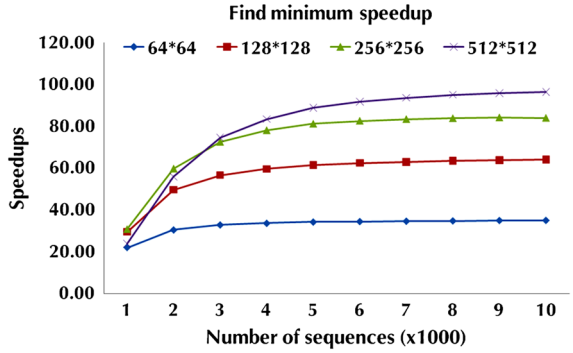

Figure 4. Speedup ratios of finding the minimum value by SUPGMA algorithm and GPU-UPGMA v1.0. The different *NTG* cases are represented by *NBG* * *NTB*. Four *NTG* cases are 64 * 64, 128 * 128, 256 * 256, and 512 * 512.

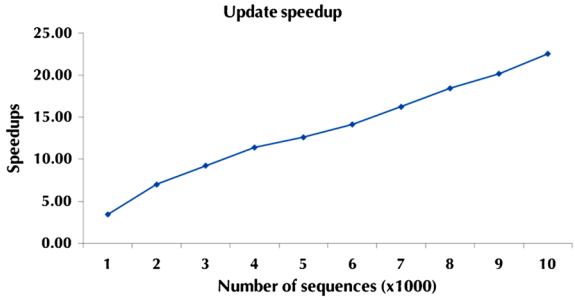

Figure 5. Speedup ratios of update distance matrix stage by SUPGMA algorithm and GPU-UPGMA v1.0. *NBG* * *NTB* is 64 * 64.

Figure 6 shows the speedup ratios by comparing the overall computational time in GPU-UPGMA v1.0 with that in the SUPGMA algorithm for different *NTG* cases. As observed in Figure 4, the speedup ratios are influenced by the variable *NTG*, and the speedup ratio increases when the size of *NTG* increases. The best result is occurred when the size of *NTG* is set to 512 * 512, and GPU-UPGMA v1.0 achieves a speedup ratio of 95× for the overall computational time over the SUPGMA algorithm. These results demonstrate that GPU-UPGMA v1.0 is an efficient GPU implementation of UPGMA.

Table II summarizes the overall computational time in GPU-UPGMA v1.0. Table II reveals that GPU-UPGMA v1.0 performs much better than SUPGMA algorithm shown in Table I. In the case with 10,000 sequences, only 47 s is necessary to construct the UT tree by using the size of *NTG*, 512 * 512. These results show the benefit by using GPU with CUDA to construct a phylogenetic tree with large taxa sets.

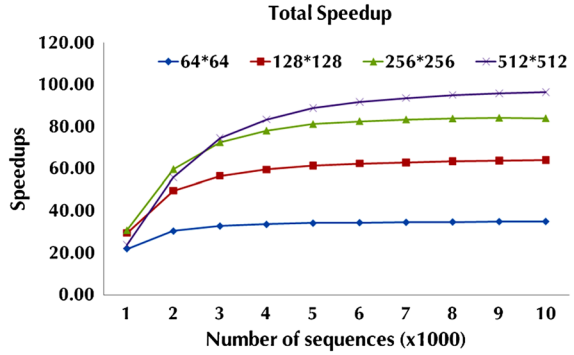

Figure 6. Overall speedup ratios by SUPGMA algorithm and GPU-UPGMA v1.0. The different *NTG* cases are represented by *NBG* * *NTB*. Four *NTG* cases are 64 * 64, 128 * 128, 256 * 256, and 512 * 512.

DOI: 10.1002/cpe

Table II. Overall computational time (unit in ms) for GPU-UPGMA v1.0 with different *NTG* cases, which are represented by *NBG* * *NTB*. Four *NTG* cases are 64 * 64, 128 * 128, 256 * 256, and 512 * 512.

	64 * 64 threads	128 * 128 threads	256 * 256 threads	512 * 512 threads
$N = 1000$	210.87	156.87	149.84	193.58
$N = 2000$	1203.02	737.92	612.87	656.05
$N = 3000$	3746.52	2174.94	1698.38	1652.67
$N = 4000$	8592.71	4855.26	3707.59	3467.88
$N = 5000$	16,537.06	9206.01	6983.07	6371.44
$N = 6000$	28,318.65	15,646.28	11,843.37	10,647.22
$N = 7000$	44,763.32	24,667.72	18,631.87	16,580.02
$N = 8000$	66,560.14	36,552.32	27,670.86	24,382.57
$N = 9000$	94,465.78	51,856.49	39,354.41	34,523.23
$N = 10{,}000$	129,290.98	70,841.28	54,056.93	47,107.70

5.3. *User interface design for GPU-UPGMA v1.0*

In order to provide the biologists to construct a phylogenetic tree for real applications by using GPU-UPGMA v1.0 on GPU, a friendly UI is also designed using QT [45] to access GPU-UPGMA

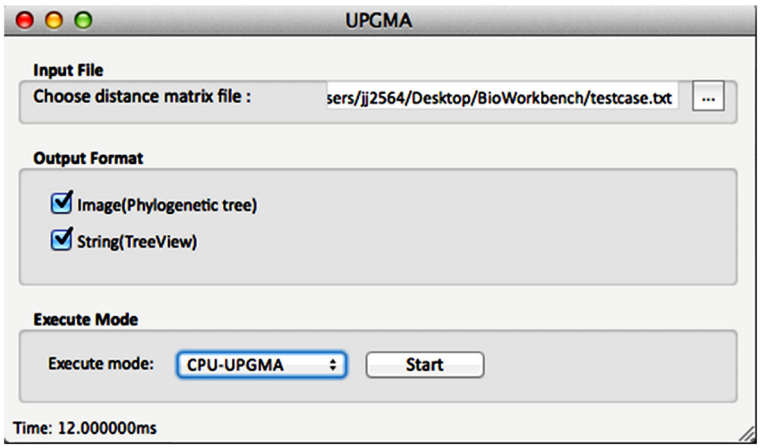

Figure 7. A user interface of GPU-UPGMA v1.0.

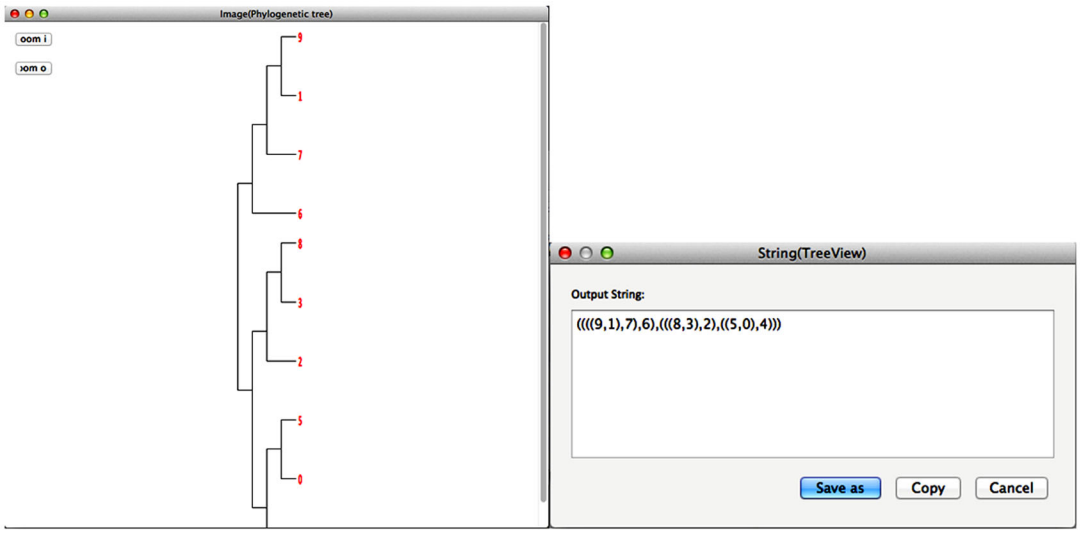

Figure 8. An example of outputs for GPU-UPGMA v1.0.

DOI: 10.1002/cpe

v1.0. As a cross-platform application framework, QT is used to design the same UI for different operating systems. Figure 7 shows the UI of GPU-UPGMA v1.0. In Figure 7, the input data are a distance matrix stored in a text file. The biologists can choose the executing program, either CPU-UPGMA (SUPGMA algorithm) or GPU-UPGMA v1.0, to construct the phylogenetic tree. The biologists also can choose the output formats including an image and/or a string in a Newick format as shown in Figure 8. The output string can be saved in a text file. The Newick format can be used as an input for other free or commercial software, such as TREEVIEW or MEGA5 [46], applied in the phylogenetic tree analysis.

6. CONCLUSIONS

This work designs and implements a GPU-UPGMA v1.0 for constructing phylogenetic tree problem with CUDA. In GPU-UPGMA v1.0, particular focus is the parallel reduction of finding the minimum value on GPU. To reduce the time of finding the minimum value, in GPU-UPGMA v1.0, the PTR algorithm is used to derive the minimum value and its corresponding index of the distance matrix. Experimental results indicate that the GPTR-M of GPU-UPGMA v1.0 achieves a speedup ratio of 97× over the SUPGMA algorithm, with 512 *Tbs* and 512 *Tds* per-block used. Moreover, the GPTR-M reduces the time of finding the minimum value in the SUPGMA algorithm. Therefore, the overall computational performance achieves a speedup ratio of 95×. Efforts are underway in our laboratory to further improve the results of our collection procedure and construct a multi-GPU system of GPU-UPGMA in order to support biology research.

ACKNOWLEDGEMENTS

The authors would like to thank the National Science Council of the Republic of China, Taiwan (contract no. NSC-100-2221-E-126-007-MY3) and the Industrial Technology Research Institute (contract no. SCRPD2B0081) for partially supporting this research. Ted Knoy is appreciated for his editorial assistance.

REFERENCES

1. Felsenstein J. Phylogenies from molecular sequences: inference and reliability. *Annual Review of Genetics* 1988; **22**:521–565.
2. Gusfield D. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press: New York, NY, USA, 1997.
3. Li WH, Graur D. *Fundamentals of Molecular Evolution*. Sinauer Associates: Sunderland, MA, USA, 1991.
4. Zuckerkandl E, Pauling L. Evolutionary divergence and convergence in proteins. In *Evolving Genes and Proteins*, Bryson V, Vogel H (eds). Academic Press: New York, NY, 1965; 97–166.
5. Hendy MD, Penny D. Branch and bound algorithms to determine minimal evolutionary trees. *Mathematical Biosciences* 1982; **59**(2):277–290.
6. Bandelt HJ. Recognition of tree metrics. *SIAM Journal on Discrete Mathematics* 1990; **3**(1):1–6.
7. Dahlhaus E. Fast parallel recognition of ultrametrics and tree metrics. *SIAM Journal on Discrete Mathematics* 1993; **6**(4):523–532.
8. Farach M, Kannan S, Warnow T. A robust model for finding optimal evolutionary trees. *Algorithmica* 1995; **13**(1):155–179.
9. Krivanek M. The complexity of ultrametric partitions on graphs. *Information Processing Letters* 1988; **27**(5):265–270.
10. Sneath PHA, Sokal RR. *Numerical Taxonomy*. W. H. Freeman: San Francisco, CA, 1973.
11. Saitou N, Nei M. The neighbor-joining method: a new method for reconstructing phylogenetic trees. *Molecular Biology and Evolution* 1987; **4**:406–425.
12. Drummond A, Rodrigo AG. Reconstructing genealogies of serial samples under the assumption of a molecular clock using serial-sample UPGMA. *Molecular Biology* 2000; **17**(12):1807–1815.
13. Pybus OG, Rambaut A, Harvey PH. An integrated framework for the inference of viral population history from reconstructed genealogies. *Genetics* 2000; **155**:1429–1437.
14. Uchiyama I. Hierarchical clustering algorithm for comprehensive orthologous-domain classification in multiple genomes. *Nucleic Acids Research* 2006; **34**(2):647–658.
15. Higgins DG, Sharp PM. CLUSTAL: a package for performing multiple sequence alignment on a microcomputer. *Gene* 1988; **73**(1):237–244.
16. Edgar RC. MUSCLE: multiple sequence alignment with high accuracy and high throughput. *Nucleic Acids Research* 2004; **32**(5):1792–1797.
17. Wallace IM, O'Sullivan O, Higgins DG, Notredame C. M-Coffee: combining multiple sequence alignment methods with T-Coffee. *Nucleic Acids Research* 2006; **34**(6):1692–1699.

DOI: 10.1002/cpe

18. Studier JA, Keppler KJ. A note on the neighbor-joining algorithm of Saitou and Nei. *Molecular Biology and Evolution* 1988; **5**(6):729–731.
19. Li KB. ClustalW-MPI: ClustalW analysis using distributed and parallel computing. *Bioinformatics* 2003; **19**:1585–1586.
20. Thompson JD, Higgins DG, Gibson TJ. CLUSTALW: improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-specific gap penalties and weight matrix choice. *Nucleic Acids Research* 1994; **22**(22):4673–4680.
21. Du ZH, Feng B. pNJTree: a parallel program for reconstruction of neighbor-joining tree and its application in ClustalW. *Parallel Computing* 2006; **32**:441–446.
22. Yu KM, Zhou J, Lin C-Y, Tang CY. Efficient parallel branch-and-bound algorithm for constructing minimum ultrametric trees. *Journal of Parallel and Distributed Computing* 2009; **69**:905–914.
23. OpenCL: https://www.khronos.org/opencl/ [accessed on 2013].
24. Nickolls J, Buck I, Garland M, Skadron K. Scalable parallel programming with CUDA. *ACM Queue* 2008; **6**:40–53. accessed, 2013.
25. Flynn M. Some computer organizations and their effectiveness. *IEEE Transactions on Computers* 1972; **C-21**:948–960.
26. Liu Y, Schmidt B, Maskell DL. MSA-CUDA: multiple sequence alignment on graphics processing units with CUDA. *Proceedings of ASAP*, 2009; 121–128.
27. Liu Y, Schmidt B, Liu W, Maskell DL. CUDA–MEME: accelerating motif discovery in biological sequences using CUDA-enabled graphics processing units. *Pattern Recognition Letters* 2010; **31**:2170–2177.
28. Liu W, Schmidt B, Mu¨ller-Wittig W. CUDABLASTP: accelerating BLASTP on CUDA-enabled graphics hardware. *IEEE/ACM Transactions on Computational Biology and Bioinformatics* 2011; **8**(6):1678–1684.
29. Liu W, Schmidt B, Liu Y, Voss G, Mu¨ller-Wittig W. Mapping of BLASTP algorithm onto GPU clusters. *Proceedings of ICPADS*, 2011; 236–243.
30. Liu Y, Schmidt B, Maskell DL. Parallel reconstruction of neighbor-joining trees for large multiple sequence alignments using CUDA. *Proceedings of IPDPS*, 2009; 23–29.
31. Liu P, Paul K. A coarse-grained reconfigurable processor for sequencing and phylogenetic algorithms in bioinformatics. *Proceedings of ReConFig*, 2011; 190–197.
32. Manavski SA, Valle G. CUDA compatible GPU cards as efficient hardware accelerators for Smith–Waterman sequence alignment. *BMC Bioinformatics* 2008; **9**(S2):S10.
33. Striemer GM, Akoglu A. Sequence alignment with GPU: performance and design challenges. *Proceedings of IPDPS*, 2009; 1–10.
34. Liu Y, Maskell DL, Schmidt B. CUDASW++: optimizing Smith–Waterman sequence database searches for CUDA-enabled graphics processing units. *BMC Research Notes* 2009; **2**:73.
35. Liu Y, Schmidt B, Maskell DL. CUDASW++2.0: enhanced Smith–Waterman protein database search on CUDA-enabled GPUs based on SIMT and virtualized SIMD abstractions. *BMC Research Notes* 2010; **3**:93.
36. Sandes FDO, Melo ACMAD. CUDAlign: using GPU to accelerate the comparison of megabase genomic sequences. *Proceedings of PPOPP*, 2010; 137–146.
37. Sandes FDO, Melo ACMAD. Smith–Waterman alignment of huge sequences with GPU in linear space. *Proceedings of IPDPS*, 2011; 1199–1211.
38. Khajeh-Saeed A, Poole S, Perot JB. Acceleration of the Smith–Waterman algorithm using single and multiple graphics processors. *Journal of Computational Physics* 2010; **229**(11):4247–4258.
39. Lee S-T, Lin C-Y, Hung C-L. GPU-based cloud service for Smith–Waterman algorithm using frequency distance filtration scheme. *International Journal of Biomedical Research* 2013. doi:10.1155/2013/721738.
40. Sreesa A, Davis JP. Custom computing for phylogenetics: exploring the solution space for UPGMA. *Proceedings of FCCM*, 2004.
41. Sokal R, Michener C. A statistical method for evaluating systematic relationships. *University of Kansas Science Bulletin* 1958; **38**:1409–1438.
42. Roger D, Assarsson U, Holzschuch N. Efficient stream reduction on the GPU. *Proceedings of GPGPU*, 2007.
43. Harris M. Optimizing parallel reduction in CUDA. Nvidia Tech Rep, 2007.
44. Wang W, Huang Y, Guo S. Design and implementation of GPU-based Prim's algorithm. *I.J.Modern Education and Computer Science* 2011; **4**:55–62.
45. QT: http://qt-project.org [accessed on 2013].
46. Tamura K, Peterson D, Peterson N, Stecher G, Nei M, Kumar S. MEGA5: molecular evolutionary genetics analysis using maximum likelihood, evolutionary distance, and maximum parsimony methods. *Molecular Biology and Evolution* 2011; **28**:2731–2739.

DOI: 10.1002/cpe