Parallel Reconstruction of Neighbor-Joining Trees for Large Multiple Sequence Alignments using CUDA

Yongchao Liu, Bertil Schmidt, Douglas L. Maskell
School of Computer Engineering, Nanyang Technological University, Singapore 639798
{liuy0039, asbschmidt, asdouglas}@ntu.edu.sg

Abstract

Computing large multiple protein sequence alignments using progressive alignment tools such as ClustalW requires several hours on state-of-the-art workstations. ClustalW uses a three-stage processing pipeline: (i) pairwise distance computation; (ii) phylogenetic tree reconstruction; and (iii) progressive multiple alignment computation. Previous work on accelerating ClustalW was mainly focused on parallelizing the first stage and achieved good speedups for a few hundred input sequences. However, if the input size grows to several thousand sequences, the second stage can dominate the overall runtime. In this paper, we present a new approach to accelerating this second stage using graphics processing units (GPUs). In order to derive an efficient mapping onto the GPU architecture, we present a parallelization of the neighbor-joining tree reconstruction algorithm using CUDA. Our experimental results show speedups of over 26× for large datasets compared to the sequential implementation.

1. Introduction

Multiple Sequence Alignment (MSA) algorithms are used to align three or more biological sequences. Exhaustive dynamic programming is a straightforward way to compute optimal MSAs. However, this approach is prohibitive in terms of both time and space. To overcome these constraints, heuristics such as progressive alignment [1], have been suggested. ClustalW [2] is a widely used and established MSA tool, which is based on the progressive alignment approach.

Even though the progressive MSA method is much more efficient than dynamic programming, it still suffers from a high computational complexity. This is particularly evident with the processing requirements resulting from the rapid growth in the size of biological sequence databases. Recent work reveals that many approaches have been devised to accelerate the execution of progressive sequence alignment. One approach is to improve the algorithms in a heuristic way, which sometimes achieves a better performance but at the expense of accuracy. Other approaches focus on the use of high-performance computing, which often achieves good performance without any degradation in the quality of results. In general, to fully exploit the computational capability of high-performance computing architectures, the algorithms have to be redesigned to conform to the programming models of the architectures. Supercomputers [18], workstation clusters [20] and special purpose hardware accelerators [22] are often used to reduce the runtime of progressive sequence alignment without loss of quality.

The development of general-purpose computing systems, consisting of a powerful CPU and a high performance graphics processing unit (GPU) for gaming applications, has resulted in a powerful tool for high-performance computing. The general-purpose computer with programmable GPU hardware has shown great promise for solving many computationally intensive problems and has opened up a range of possibilities for a variety of application domains ranging from scientific computing [3], computational geometry [4], database operations [5], image processing [6] [7], bioinformatics [8]and so forth. Earlier development on GPUs required that applications use explicit graphics application programming interfaces (APIs) to organize data into streams and invoke kernels, which were usually written using high-level programming languages, such as Cg, GLSL, etc. The main challenge involved in using this methodology is the constraints of the development environment and the limited language support to leverage the huge performance potential. The advent of CUDA [9] enables the GPU to perform computations where hundreds, or even thousands, of streaming processor cores communicate and cooperate with each other to solve complex computing problems, thus transforming the GPU into a massively parallel processor. Speedups ranging from 10× to more than 100× have been achieved using CUDA [26]. Motivated by the power of CUDA-enabled GPUs, we exploit this technology to accelerate the neighbor-joining method [10] used in ClustalW. To

978-1-4244-3750-4/09/$25.00 ©2009 IEEE

the best of our knowledge, it is the first time that the parallel neighbor-joining method has been implemented on GPUs.

The rest of the paper is organized as follows: Section 2 introduces the basic progressive alignment algorithm, analyzes the complexity of ClustalW and then gives a brief summary of previous work to accelerate different stages of this algorithm on different architectures. In Section 3, the CUDA architecture and programming model are described. Section 4 presents a basic neighbor-joining algorithm and an improved compact memory algorithm implemented using CUDA. The performance is evaluated in Section 5. Finally, we conclude this paper and give a roadmap for our future work in Section 6.

2. Multiple Sequence Alignment

2.1. Progressive Sequence Alignment

Progressive sequence alignment typically consists of three stages (see Figure 1). Stage 1 computes a distance matrix comprised of the distance value between each pair of input sequences. Each distance value is generated using a pairwise sequence alignment. In ClustalW, there are two pairwise alignment methods available: one is a faster but approximate method, called fast pairwise alignment and the other is a slower but accurate method, called full pairwise alignment.

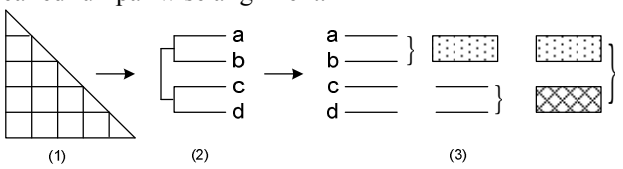

Figure 1. **Three stages of progressive sequence alignment: (1) distance matrix; (2) guided tree and (3) progressive alignment along the guided tree**

Stage 2 generates a guided tree from the distance matrix produced by Stage 1 using distance-based phylogenetic tree reconstruction methods. In ClustalW, the neighbor-joining method is used to reconstruct the guided tree. The neighbor-joining method is a distance-based method for reconstructing phylogenetic trees from evolutionary distance data [10]. It provides not only the topology but also the branch lengths of the final tree. The original version by Saito and Nei [10] was modified by Studier and Keppler [11] so that it runs in time $O(n^3)$, where n is the number of operational taxonomic units. This stage can be divided into two sub-stages: the neighbor-joining tree reconstruction (*NJ Tree*) sub-stage and the getting weights and steps (*W&S*) sub-stage, which reroots the unrooted neighbor-joining tree and gets the sequence weights and the alignment steps for Stage 3.

Stage 3 performs progressive alignment of the various profiles to form the final MSA along the guided tree. In ClustalW, the final progressive alignment consists of two stages: the closely related sequences, which are the leaves of a same sub-tree in the guided tree, are aligned first and then more divergent sequences are aligned to the closest aligned sequences.

2.1 Complexity Analysis of ClustalW

To identify ClustalW bottlenecks, we consider the complexities of all the three stages. Previously, the complexity of ClustalW version 1.82 has been analyzed by Edgar [12]. However, ClustalW version 2.0 contains some new features [13]. In this paper, the latest ClustalW source code version 2.0.9 is chosen for analysis. Given a DNA or protein sequence dataset $S=\{S_1, S_2, \ldots, S_i, \ldots, S_n\}$, which consists of n sequences, define l_i to denote the length of the sequence S_i and l_{ave} to denote the average length of all the sequences in dataset S.

As mentioned above, there are two pairwise alignment methods in ClustalW. In this paper, we consider the complexity analysis of *full pairwise alignment* as it is more widely used in practice. The *full pairwise alignment* calculates more accurate scores from full dynamic programming alignments, which mainly consists of three steps for each pair of sequences. Firstly, a forward optimal local alignment is performed, using the Smith-Waterman alignment algorithm [14][15] from the beginning of both sequences to their respective ends in order to identify the position (se_1, se_2) of the maximum score. Secondly, a reverse optimal global alignment is performed from the position (se_1, se_2) to the beginning of both sequences, respectively, to find another position (sb_1, sb_2) with a maximum score that is equal to the previous one; finally, an affine gap penalty global alignment using the Myers and Miller algorithm [17] from position (sb_1, sb_2) to (se_1, se_2) is performed. Let $T_{dist}(S)$ denote the runtime of *full pairwise alignment*, $T_{sw}(l_i, l_j)$ denote the runtime of the Smith-Waterman Algorithm dealing with a pair of sequences whose length are l_i and l_j respectively and $T_{mm}(lm_i, lm_j)$ denote the runtime of the Myers and Miller algorithm to align two profiles whose length are lm_i and lm_j respectively. In general, the runtime of the forward alignment is slightly different from that of the reverse alignment. Here for simplicity, we assume that both have the same runtime. Therefore, $T_{dist}(S)$ can be denoted by the following equation:

$$T_{dist}(S) = \sum_{i=1}^{n} \sum_{j=i+1}^{n} \left(2 * T_{sw}(l_i, l_j) + T_{mm}(lm_i, lm_j)\right) + O(n^2) \quad (1)$$

It is well-known that the complexity of the Smith-Waterman Algorithm is $O(l_i \times l_j)$ [16] and the complexity of the Myers and Miller algorithm is $O(lm_i \times lm_j)$ [12]. In

the *full pairwise alignment*, lm_i and lm_j are never greater than l_i and l_j, respectively. From the equation (1), it can be easily shown that the complexity of *full pairwise alignment* is $O(n^2 l_{ave}^2)$.

For each internal node in the neighbor-joining tree, the method has to select a pair of valid nodes ready to be combined into the tree. Let C_{tree} denote the runtime of the basic operations to calculate the branch length of the guided tree when one node pair is combined into a new node, where C_{tree} is a constant value. Then the runtime of the neighbor-joining tree can be determined as:

$$T_{tree}(S) = \sum_{k=1}^{n-3} \left(\sum_{i=1}^{n-k+1} \sum_{j=i+1}^{n-k+1} C_{tree} \right) + O(n^2) \qquad (2)$$

thus, the complexity of neighbor-joining tree is $O(n^3)$.

The final progressive alignment consists of two stages: the first is to align the closely related sequences following the neighbor-joining tree; the second is to align more divergent sequences to its closest pair among the aligned sequences. The runtime of a single iteration of progressive alignment is mainly subject to two operations: the construction time of profiles from sequences and the runtime of aligning both profiles using the Myers and Miller algorithm. Let s and t denote the lengths of both profiles respectively. The complexity of operations relating to profile construction is $O(n \times (s+t))$. Since the complexity of the Myers and Miller algorithm is $O(s \times t)$, the complexity of a single iterative progressive alignment can be considered as $O(n \times (s+t)+s \times t)$. For the convenience of analysis, without loss of generality, we assume that s and t are both equal to l_{ave}, so the complexity of a single iterative progressive alignment is $O(nl_{ave} + l_{ave}^2)$.

If the number of times that progressive alignment is performed were known, it would be very easy to work out the complexity of progressive alignment. However, the guided tree differs for different sequence datasets, making it difficult to exactly evaluate how many times a single iterative progressive alignment is executed. Instead, let us examine the extreme cases. Obviously, for n sequences, the number of internal nodes of the guided tree is n-1. On one hand, if all of sequences were close enough to be aligned, a single iterative progressive alignment will be executed n-1 times. On the other hand, if the sequences were so divergent that none is aligned in the first stage, a single iterative progressive alignment would also be executed n-1 times in this stage. That is to say, in all cases, the total number of times that a single iterative progressive alignment is executed can not be greater than $2 \times (n\text{-}1)$. Thus, we can conclude that the complexity of progressive alignment is $O(nl_{ave}^2 + n^2 l_{ave})$. Table 1 presents the complexities of different stages of ClustalW (version 2.0.9).

Table 1. **Big-O asymptotic complexities of the different stages of ClustalW version 2.0.9**

Stage	***O*(Time)**
Distance matrix	$O(n^2 l_{ave}^2)$
Guided Tree	$O(n^3)$
Progressive Alignment (each iteration)	$O(nl_{ave} + l_{ave}^2)$
Progressive Alignment (total)	$O(nl_{ave}^2 + n^2 l_{ave})$
Total	$O(n^2 l_{ave}^2 + n^3)$

2.2 Profiling of Sequential ClustalW

From the complexities shown in Table 1, we can make the following conclusions about the general runtime behavior of ClustalW:

- Stage 1 always has a longer runtime than Stage 3;
- If $n > l_{ave}$ then Stage 2 has a longer runtime than Stage 3;
- if $n > l_{ave}^2$ then Stage 2 has a longer runtime than Stage 1.

To verify the above conclusion with actual runtimes, we have used several large datasets of protein HIV sequences with small average lengths and profiled the runtime of different stages of ClustalW 2.0.9 on a P4 3GHZ with 1GB RAM running Linux. The results are shown in Table 2.

Table 2. **Runtimes (in second) of the different stages of ClustalW (version 2.0.9) for Protein HIV datasets**

Sequence Number	**Average Length**	**Distance Matrix**	**Guided Tree**		**Progressive Alignment**
			NJ Tree	***W&S***	
4000	57	963	258	95	226
5866	54	1969	810	276	564
8000	73	5165	2062	616	884
8309	57	3628	2397	695	323
10083	49	3999	4266	1258	603

When dealing with the dataset of 10083 sequences, the runtime of the neighbor-joining tree reconstruction sub-stage and the overall runtime of Stage 2 is larger than that of Stage 1. Hence, it is imperative to speed up the neighbor-joining method for large sequence datasets so that multiple sequence alignment using ClustalW could be finished in a reasonable time.

2.3 Previous Work on Parallelizing MSA

Previous work on parallelizing ClustalW has been performed on coarse-grained (e.g.[19-22]) as well as fine-grained parallel architectures (e.g.[22-23]).

Li [19] implemented ClustalW-MPI running on workstation clusters with a distributed memory architecture, which parallelizes the first and the third stages of ClustalW to reduce the execution time. Ebedes

and Datta [20] also parallelized the first stage and the third stage of ClustalW using the MPI method that is similar to Li [19]. Du and Lin [21] pointed out that when handling larger datasets, such as when the sequence number is over 5000, the reconstruction of the neighbor-joining tree occupies more than 30% of the runtime of ClustalW. To overcome this, they proposed a parallel method for the reconstruction of the neighboring-joining tree using MPI running on a workstation cluster [21].

A number of fine-grained parallel architectures such as FPGAs and GPUs have also been used for speeding up ClustalW. Oliver et al [22] [23]constructed a linear systolic array to perform pairwise sequence distance computations using dynamic programming on a standard FPGA. The pairwise alignment algorithm was efficiently mapped to a linear array of PEs, where each PE is assigned to each character in the query string and a subject sequence is shifted through the linear chain of PEs in a systolic way. Liu et al [24] reformulated the dynamic-programming-based alignment algorithm into streaming algorithm using OpenGL and implemented the pairwise alignment on a GPU.

Besides ClustalW, some other multiple sequence alignment tools including T-Coffee [28], MUSCLE [12] and DIALIGN [29], have also been parallelized. Zola et al [30] presented a parallel implementation of T-Coffee, which parallelizes the library generation stage and the progressive alignment stage using MPI. Deng et al [31] parallelized the three modules (FRA, PMC and CT), which take most of the runtime of MUSCLE, with OpenMP paradigms. Boukerche et al [32] implemented a hardware accelerator on an FPGA to execute the most compute intensive part of DIALIGN.

3. CUDA Programming Model

NVIDIA's CUDA development environment enables developers to utilize the tremendous computational capability and high memory bandwidth of GPUs. A CUDA-enabled GPU is composed of a scalable array of multithreaded streaming multiprocessors, with each multiprocessor consisting of eight processor cores. When a CUDA program running on the host CPU invokes a kernel grid which can run on a GPU with available execution capability, the thread blocks of the grid will be distributed to the multiprocessors. The threads of a thread block are grouped into warps, which execute concurrently on one of the multi-processors. CUDA makes three refinements to the concept of running kernel functions across thousands of parallel threads: hierarchical thread blocks, shared memory and barrier synchronization [26].

Thread blocks may contain up to 512 threads. All the threads in a thread block execute the same kernel and are scheduled independently. A thread block can be arranged as a 1-, 2- or 3-D array. Threads in a block can communicate and synchronize with each other, whereas threads in different blocks can not. The on-chip shared memory is very small but extremely fast. As long as there is no bank conflict, it can be accessed as fast as accessing registers. Shared memory is visible to all the threads in a block. Therefore, the threads in a block can share data through the fast on-chip shared memory. Sometimes, the extent to which it is possible to efficiently utilize shared memory has a significant impact on the performance. When threads are accessing the same memory in parallel, a mechanism to guarantee that one thread is ready to read an address will not conflict with another thread ready to write to the same address. In CUDA, a simple and light-weight intrinsic function __syncthreads () is provided to act as a barrier synchronization at which all the threads in a block must wait until the last thread finishes.

Any thread can access device memory from global memory, constant memory or texture memory. The global memory space is not cached. Hence, the optimization of global memory is especially important as the global memory bandwidth is low and its latency is hundreds of clock cycles. The constant memory is cached and only readable by the threads. For all the threads of a half-warp, the cost of reading from constant memory scales linearly with the number of different addresses read. As long as all threads read the same address, reading from constant memory is as fast as reading from a register. The texture memory is cached and is also only readable by threads. Generally, reading device memory through texture fetching can achieve better performance than reading device memory directly from global or constant memory [9].

4. Parallelization of Neighbor-Joining Tree Reconstruction

4.1 Sequential Neighbor-Joining Algorithm

In ClustalW, the guided tree is used to guide the final multiple sequence alignment process. It is calculated using the neighbor-joining method from the distance matrix produced by pairwise sequence alignment. Figure 2 shows the descriptive procedure of the neighbor-joining method in ClustalW version 2.0.9.

The inner loop (from lines 7 - 18) is the most time consuming step with time complexity $O(n^2)$. If the inner loop could be completed in constant or nearly linear time by using parallelization techniques, the runtime of the reconstruction of the neighbor-joining tree would be significantly reduced. In this paper, two parallel methods using CUDA are devised to implement the inner loop.

```
1  /*Here, all the sequences are defined as leaf nodes of the guided tree. Any node that is ready be
2  combined with another node and added into the tree is called a valid node.*/
3  Initialization;
4  For the construction of each internal tree node numbered from 1 to n−3,  loop
5  /*Calculate and select the node pair (i , j) from valid nodes, whose combination makes
6   the guided tree have the smallest branch length than that of all other pairs;*/
7  For valid node j, loop
8   For valid node i and i<j, loop
9      Calculate S_ij = ( n - 2)* D_ij − ( R_i + R_j ), where D_ij is the distance between node i and j
10     stored in the distance matrix, R_i is the sum of values in the ith row (or column) of the
11     distance matrix, and R_j is the sum of values in the jth row ( or column) of the distance
12     matrix, and n is the number of valid nodes.
13
14     If S_ij < S_min , then
15        Store this pair ( i , j) and store the value S_ij to S_min.
16     End if;
17   End loop;
18  End loop;
19  Create a new node whose children are node i and j and then add it to the neighbor-joining tree;
20  Update distance matrix and other relevant variables;
21  End loop;
22  Finish the reconstruction of the neighbor-joining tree.
```

Figure 2. **Pseudocode of the sequential neighbor-joining method**

4.2 Basic Parallel Algorithm Using CUDA

From Figure 2, it is easily seen that for each pair of nodes i and j, the calculation of $S_{i,j}$ is independent from the other pairs. That is to say, there is no data dependency between any pair of nodes, which is the most ideal case for parallel application design.

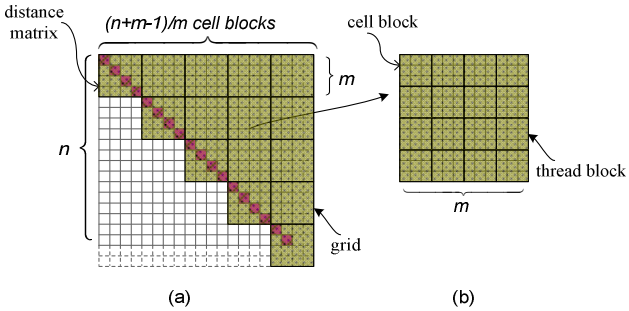

Figure 3. **Simple and straightforward grid mapping method in the basic algorithm: (a) the distance matrix is mapped to a grid of thread blocks; (b) one cell block corresponds to one thread block**

From the perspective of CUDA architecture, the distance matrix can be mapped onto a 1- or 2-D grid of thread blocks, each of which processes different cells in the distance matrix. The threads in a thread block concurrently conduct the computational work on those cells distributed to the thread block. Since the distance matrix is a symmetric square matrix, the data of the cells in the upper triangle or lower triangle is sufficient for the reconstruction of NJ tree. Therefore, a basic and straightforward algorithm (as shown in Figure 3) is to map the upper triangle (or the lower triangle) of the distance matrix to a 1-D grid of thread blocks and then group the cells in the upper triangle (or the lower triangle) of the distance matrix into many equally-shaped cell blocks, so that all the cells can be tackled in a coherent way. One thread block in the grid is designed to logically correspond to one cell block in the distance matrix and for simplicity, both the thread blocks and the cell blocks are all designed as 2-D square matrixes. According to Figure 3, for a distance matrix of size $n \times n$, if one cell block is of size $m \times m$ in the distance matrix, the total number of cell blocks can be denoted as follows:

$$totalBlocks = \frac{blocks \times (blocks+1)}{2}, \qquad blocks = (n+m-1)/m \tag{3}$$

Since there is one-to-one correspondence relationship between cell blocks and thread blocks, the total number of thread blocks in the grid is equal to the total number of cell blocks in the distance matrix.

According to [9], the number of threads per thread block should be a multiple of the warp (consisting of 32 threads) size to avoid wasting computing resources and at least 192 active threads per multiprocessor to hide the delays incurred by read-and-write dependencies of registers. Thus, in this algorithm, 256 threads per thread block is a better choice considering the available number of registers per block. To keep the total number of thread blocks within a reasonable range and achieve a better load balancing for all threads in a thread block, the size of a cell block must be carefully selected. The assignment of the same number of cells to each thread in a thread block leads to a good load balancing. For example, for a cell block of size 64 × 64, each thread in a thread block may be assigned to process a cell sub-block of size 4 × 4. Figure 4 shows the pseudocode of the CUDA implementation of the inner loop (from line 7 to line 18) in Figure 2 using the basic algorithm.

At the initialization stage, the distance matrix is loaded into GPU device memory from host memory. After each node pair is selected, the data values of the relevant cells in the distance matrix have to be updated for the successive iteration. If the whole matrix is entirely re-loaded, the time overhead would be fairly high due to the relatively narrow memory bandwidth between the host and the GPU. In order to significantly reduce the amount of data transferred, only the changed valid cells are updated. In this way, only the data values of one-column and one-row cells in the distance matrix need to be transferred from host to GPU every iteration, which makes the data transfer overhead negligible.

Shared memory is used to store the temporary results of each block. Each thread in one thread block compares and selects the node pair whose combination into a new node gives the smallest branch length among the node pairs allocated to it and then stores the selected node pair and its value S_{min} into the storage space allocated to it in shared memory. Texture memory is exploited to store the distance matrix.

```
/*****************CUDA KERNEL OF THE BASIC METHOD*********************/
Step 1: Calculate the start row blk_row and start column blk_col of the cell block
        in the distance matrix corresponding to the thread block that this thread belongs to.

Step 2: Calculate the start row start_row and start column start_col of the cell square matrix that
        this thread processes in the above cell block.
        start_row = blk_row + threadIdx.y * cells_y;
        start_col = blk_col + threadIdx.x * cells_x;

Step 3: Calculate the end row end_row and end column end_col of the cell square matrix that this
        thread processes in the above cell block
        end_row = max(start_row + cells_y, size);
        end_col = max(start_col + cells_x, size);

Step 4: Initialize the relevant variables
    for (j = start_col;  j < end_col; j ++){
        if (((int)tex2D(sumCols, j, 0)) == NJ_INVALIDNODE){
            continue;
        }
        for (i= start_row; i < end_row; i++){
            if (i < j){
                if (((int)tex2D(sumCols, i, 0)) == NJ_INVALIDNODE){
                    continue;
                }

                total = tex2D(distMat, j, i);
                total *= fnseqs2;
                total += sumd;
                total -= tex2D(sumRows,i,0) + tex2D(sumCols,i,0);
                total -= tex2D(sumRows,j,0) + tex2D(sumCols,j,0);
                if ( total < tmin){
                    tmin=total;
                    mini=i;
                    minj=j;
                }
            }
        }
    }
    blk_tmins[tid]= tmin;
    blk_minis[tid] = mini;
    blk_minjs[tid] = minj;
    __syncthreads();

Step 5: Let thread 0 in this thread block select the node pair (mini, minj) with the minimum value
        among all the node pairs processed by the threads in this thread block and then save the
        node pair (mini, minj) and its corresponding value tmin into the storage space allocated
        for this thread block in the global memory.
```

Figure 4. **Pseudocode of the CUDA kernel for the neighbor-joining method using the basic algorithm**

4.3 Improved Compact Memory Parallel Algorithm Using CUDA

If the entire distance matrix is loaded into GPU device memory without any modification, half of the GPU memory space is wasted due to the symmetry of the distance matrix. Since the GPU device memory is relatively small (typically less than 1GB), it is advantageous to fully utilize the memory so that larger datasets can be accommodated.

To improve the GPU device memory utilization, we propose an improved algorithm based on memory compaction. In this algorithm, the distance matrix is considered as a 2-D coordinate space. Through coordinate mapping, up to half of the memory size can be saved compared with the basic algorithm. For n sequences, in the basic algorithm, the size of the memory occupied by the distance matrix is $(n + 1) \times (n + 1)$, whereas in the compact memory algorithm, the size is reduced to $(n + 1) \times (MidPoint + 1)$, where $MidPoint$ is equal to $(n + 1)/2$. Figure 5 illustrates an example for both algorithms.

As can be seen from Figure 5, this algorithm maps the bottom half of the upper triangle to the lower triangle of the top half. The distance matrix is considered as a 2-D coordinate space on the Cartesian plane, where the origin is located on the left-top corner and the horizontal (x) coordinates increase from the left to the right and the vertical (y) coordinates increase from the top to bottom. The coordinate mapping rules are as follows. For each cell (x, y) in the upper triangle of the distance matrix,

- if y is less than or equal to $MidPoint$, the coordinate is not modified;
- if y is greater than $MidPoint$, the coordinate is mapped to $(n + 1 - x, n + 1 - y)$.

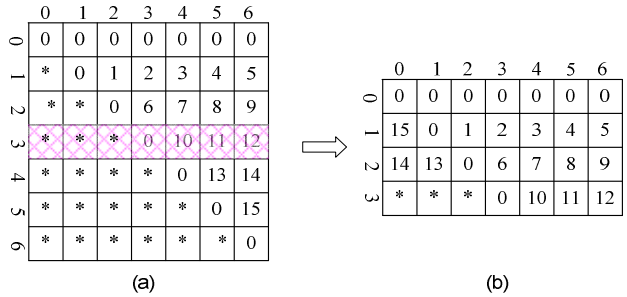

Figure 5. **Examples of the distance matrices in both algorithms: (a) the original distance matrix used in the basic algorithm; (b) the new compact memory distance matrix in the improved algorithm**

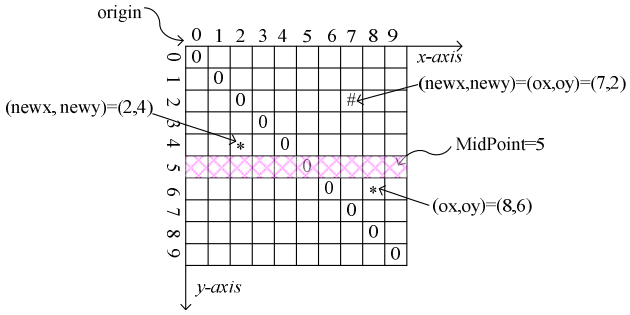

Figure 6**. Coordinate mapping rules used in the compact memory algorithm.**

Figure 6 shows an example of the mapping of the new compact memory distance matrix for 9 sequences (n = 9). For the coordinate (8, 6), because the y coordinate (i.e. 6) is greater than $MidPoint$, it is mapped to coordinate (n+1-8, n+1-6) = (2, 4). Coordinate (7, 2), it is not modified because the y coordinate (i.e. 2) is less than $MidPoint$. Therefore, this algorithm requires coordinate transformations for the cells whose y coordinates are greater than $MidPoint$ when accessing data in the distance matrix. In this algorithm, the cells that are in the same row (or column) in the original distance matrix are still in the same row (or column) in the mapped distance matrix.

5. Performance Evaluation

We have implemented the basic algorithm and the improved compact memory algorithm using CUDA and evaluated both of the algorithms on an NVIDIA GeForce GTX 280 graphics card. This card has 30 streaming multiprocessors (a total of 240 scalar processing cores)

operating at a clock rate of 1296MHz with 1GB of device memory. It is installed in a PC with an AMD Opteron 248 2.2 GHz processor and 2GB RAM running the Linux operating system. The original ClustalW (version 2.0.9) program is executed on a HP xw4100 workstation with a P4 3.0 GHZ processor and 1GB RAM running the Linux operating system.

The topology of the phylogenetic tree depends on the sequences, whereas from the neighbor-joining method, it can be seen that the runtime of the neighbor-joining method mainly depends on the number of sequences. Hence, the actual sequences have little impact on the runtime. Without loss of generality, all the data values of the cells in the distance matrix are positive, normalized simulated data, which can be generated from a random number generator for simplicity.

In both implementations, every thread block consists of 256 threads and each thread processes 16 different cells in the corresponding cell block. The experiments are conducted on datasets where the number of sequences ranges from 1,000 to 12,000. Figure 7 demonstrates the speedups of both algorithms.

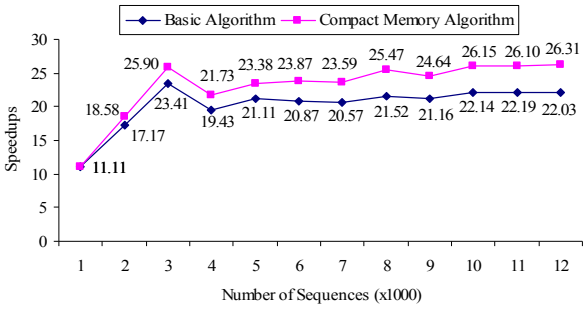

Figure 7. **Speedups of the basic and the improved compact memory algorithms**

The experimental results indicate that our CUDA implementations have achieved significant performance speedups over the original neighbor-joining method. Furthermore, the speedups increase with the size of datasets in both algorithms, albeit growing more slowly. For the datasets with more than 10,000 sequences, a speedup of more than 22 is achieved for the basic algorithm and more than 26 for the improved compact memory algorithm. An average speedup of 20.23 is achieved for the basic algorithm and 23.07 for the improved compact memory algorithm.

From Figure 7, it can be seen that for all tested datasets, the improved compact memory algorithm outperforms the basic algorithm. This might be due to the following reasons. For a CUDA application, the issue order of the thread blocks within the grid is undefined. When one of the thread blocks assigned to a streaming multiprocessor finishes, one of the thread blocks in the waiting lists will be activated and scheduled to this streaming multiprocessor. Thus, the cells in the distance matrix, which are accessed through texture memory by all the threads at any time during the execution of both algorithms, may be widely distributed all through the distance matrix. This distribution is a challenge to the texture caching. With the increase of the sizes of datasets, the basic algorithm distance matrix occupies much more memory, resulting in a worse cache hit rate compared to the improved compact memory algorithm. In this case, the basic algorithm spends more time on external memory accesses, so that the time spent on the coordinate mapping become negligible and the improved compact memory algorithm runs faster than the basic algorithm. Both implementations use 20 registers for their kernels and less than 4KB shared memory, respectively. Table 3 gives the GPU occupation data calculated using CUDA Occupancy Calculator [27].

Table 3. **GPU occupation data for both algorithms**

Active Threads per Multiprocessor	768
Active Warps per Multiprocessor	24
Active Thread Blocks per Multiprocessor	3
Occupancy of each Multiprocessor	75%

6. Conclusion and Future work

In this paper, we have introduced two CUDA algorithms for the neighbor-joining tree reconstruction for large multiple sequence alignments. Our implementation on an NVIDIA GeForce GTX 280 graphics card results in a speedup up to 26 for datasets with more than 10,000 sequences. Even though the reconstruction of neighbor-joining tree can dominate the runtime of ClustalW for large datasets, the runtime of the first stage is still significant. Therefore, to further speed up ClustalW, the first stage must also be parallelized. Our experimental results have shown the high computational power of CUDA-enabled GPUs. Therefore, parallelizing the first stage using CUDA is part of our future work. Furthermore, we are planning to accelerate all the stages of ClustalW using a dedicated hybrid computing system, which is equipped with a multi-core CPU, a CUDA-compatible GPU and a high-density FPGA.

7. Acknowledgement

We would like to thank Dr. Liu Weiguo for helping provide the experimental platform.

8. References

[1] Feng D., Doolittle R.: Progressive sequence alignment as a prerequisite to a correct phylogenetic Trees. J. Molecular

Evolution, vol. 25, pp. 351-360, 1987
[2] Thompson J.D., Higgins D.G., Gibson T.J.: CLUSTALW: 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] Stone J.E., Phillips J.C., Freddolino P.L., Hardy D.J., Trabuco L.G., Schulten K.: Accelerating molecular modeling applications with graphics processors. Journal of Computational Chemistry, vol.28, No. 16 , pp. 2618 – 2640, 2007
[4] Agarwal P.K., Krishan S., Mustafa N.H., Venkatasubramanian S.: Streaming Geometric Optimization Using Graphics Hardware. Proc. 11th European Symp. Algorithms, 2003
[5] Govindaraju N., Lloyd B., Wang W., Lin M., Manocha D.: Fast Computation of Database Operations Using Graphics Processors, Proc. ACM Int'l Conf. Management of Data(SIGMOD'04), pp. 215-226, 2005
[6] Sharp G.C., Kandasamy N., Singh H., Folkert M.: GPU-based streaming architectures for fast cone-beam CT image reconstruction and demons deformable registration. Physics in Medicine and Biology, vol.52, No. 19, pp.5771-5783, 2007
[7] Muyan-Ozcelik P., Owens J.D., Xia J., Samant S.S.: Fast Deformable Registration on the GPU: A CUDA Implementation of Demons. International Conference on Computational Sciences and Its Applications. pp. 223-233, 2008
[8] Bader D.A.: Computational Biology and High-Performance Computing. Communications of the ACM, vol. 47, No. 11, pp. 34-41, 2004
[9] Nvidia Corporation. NVIDIA CUDA Compute Unified Device Architecture Programming Guide version 2.0, 2008
[10] Saitou N., Nei M.: The Neighbor-Joining Method: a new method for reconstructing phylogenetic trees. Molecular Biology and Evolution, vol 4, pp.406-425, 1987
[11] Studier J.A., Keppler K.J.: A note on the neighbor-joining method of Saitou and Nei. Mol.Biol.Evlo.5(6): 729-731, 1988
[12] Edgar R.C.: MUSCLE: a multiple sequence alignment method with reduced time and space complexity. BMC Bioinformatics, vol. 5, No. 113, 2004
[13] Larkin M.A., Blackshields G., Brown N.P., Chenna R., McGettigan P.A., McWilliam H., Valentin F., Wallace1 I.M., Wilm A., Lopez R., Thompson J.D., Gibson T.J., Higgins D.G.: Clustal W and Clustal X version 2.0. Bioinformatics Applications Note, vol. 23, No.21, pp. 2947-2948, 2007
[14] Smith T., Waterman M.: Identification of Common Molecular Subsequences. J. Molecular Biology, vol.147, pp. 195-197, 1981
[15] Gotoh O.: An improved algorithm for matching biological sequences. J. Mol.Biol. Vol.162, No. 3, pp. 707-708, 1982
[16] Yu C.W., Kwrong K.H., Lee K.H., Leong P.H.: A Smith-Waterman Systolic Cell". 13th International Conference on Field-Programmable Logic and Applications. Springer-Verlag LNCS, Lisbon, Portugal, 2778: 375-384, 2003
[17] Myers E.W., Miller W.: Optimal alignments in linear space. Comput Appl Biosci, vol.4, No. 1 pp. 11-17, 1988
[18] Kleinjung, J., Douglas, N., Heringa, J.: Parallelized Multiple Sequence Alignment. Bioinformatics, Vol. 18, No. 9, pp.1270–1281, 2002
[19] Li K.B.: Clustal-MPI: ClustalW analysis using parallel and distributed computing. Bioinformatics, vol.19, No.12, pp.1585-1586, 2003
[20] Ebedes J., Datta A.: Mulitple sequence alignment in parallel on a workstation cluster. Bioinformatics, 20(7) pp.1193-1195, 2004
[21] Du Z.H., Feng B.: pNJTree: A parallel program for reconstruction of neighbor-joining tree and its application in ClustalW. Parallel Computing, vol.32, pp. 441-446, 2006
[22] Oliver T., Schmidt B., Nathan D., Clemens R., Maskell D.: Using reconfigurable hardware to accelerate multiple sequence alignment with ClustalW. Bioinformatics, vol. 21, No. 16, pp. 3431-3432, 2005
[23] Oliver T., Schmidt B. and Maskell D.L., "Reconfigurable Architectures for Bio-sequence Database Scanning on FPGAs", IEEE Trans. Circuits Syst. II, Vol. 52, pp. 851-855, Dec, 2005.
[24] Liu W., Schmidt B.,Voss G., Muller W.: Streaming Algorithms for Biological Sequence Alignment on GPUs. IEEE transactions on parallel and distributed systems, vol.18, No. 10, 2007
[25] Liu Y., Huang W., Johnson J., Vaidya S.: GPU Accelearted Smith-Waterman. Proc. ICCS'06, pp. 188-195, 2006
[26] Luebke D.: CUDA: Scalable parallel programming for high-performance scientific computing. ISBI 2008
[27] CUDA Zone (CUDA programming tools): http://www.nvidia.com/object/cuda_programming_tools.html
[28] Notredame C., Higgins D.G., Heringa, J.: T-Coffee: a novel method for fast and accurate Multiple Sequence Alignment, Journal of Molecular Biology, Vol. 302, No. 1, pp.205–217, 2000
[29] Morgenstern B., Werner T.: Dialign 1.0: Multiple alignment by Segment rather than by Position comparison. German Conference on Bioinformatics, pp. 69-71, 1997
[30] Zola J., Yang X., Rospondek S., Aluru S.: Parallel T-Coffee: A Parallel Multiple Sequence Aligner. Proc. of ISCA PDCS-2007, pp. 248-253, 2007
[31] Deng X., Li E., Shan J., Chen W.: Parallel implementation and performance characterization of MUSCLE. Parallel and Distributed Processing Symposium, 2006.
[32] Boukerche A., Correa J.M., A.C.M.A. Melo, Jacobi R.P., Rocha A.F: An FPGA-Based Accelerator for Multiple Biological Sequence Alignment with DIALIGN. Lecture Notes Computer Science. Vol. 4873, pp. 71-82, 2007