2009 20th IEEE International Conference on Application-specific Systems, Architectures and Processors

MSA-CUDA: Multiple Sequence Alignment on Graphics Processing Units with CUDA

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

***Abstract*—Progressive alignment is a widely used approach for computing multiple sequence alignments (MSAs). However, aligning several hundred or thousand sequences with popular progressive alignment tools such as ClustalW requires hours or even days on state-of-the-art workstations. This paper presents MSA-CUDA, a parallel MSA program, which parallelizes all three stages of the ClustalW processing pipeline using CUDA and achieves significant speedups compared to the sequential ClustalW for a variety of large protein sequence datasets. Our tests on a GeForce GTX 280 GPU demonstrate average speedups of 36.91 (for long protein sequences), 18.74 (for average-length protein sequences), and 11.27 (for short protein sequences) compared to the sequential ClustalW running on a Pentium 4 3.0 GHz processor. Our MSA-CUDA outperforms ClustalW-MPI running on 32 cores of a high performance workstation cluster.**

Keywords-multiple sequence alignment; CUDA; GPU; ClustalW

I. INTRODUCTION

Multiple Sequence Alignments (MSAs) are one of the primary research areas in bioinformatics, involving aligning three or more biological sequences at the same time. Exhaustive dynamic programming is a straightforward way to compute optimal MSAs. However, the cost of this approach is expensive in terms of both computing time and memory space. This becomes especially evident with the rapid growth of biological sequence databases demanding more powerful high-performance computing solutions. To overcome these constraints, some heuristics such as progressive alignment [1], have been suggested. Progressive alignment is a widely used heuristic for MSA. Many popular MSA software packages have been developed based on this approach, including T-Coffee [2], MUSCLE [3], and ClustalW [4]. ClustalW has more than 26,000 citations in the ISI Web of Science and is considered the most popular MSA tool.

Typically, progressive alignment consists of three stages (see Fig. 1): pairwise distance computation, guided tree generation and profile-profile progressive alignment along the guided tree. Stage 1 computes a distance matrix comprised of the distance value between each pair of sequences using pairwise alignment. Stage 2 generates a guided tree from the distance matrix using distance-based phylogenetic tree reconstruction methods. Stage 3 performs progressive alignment of the various profiles to form the final MSA along the guided tree.

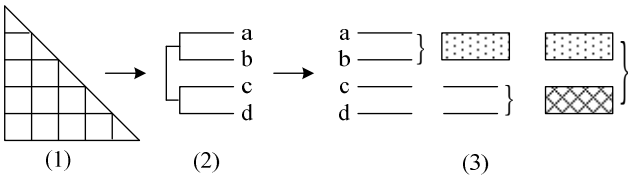

Figure 1. Three stages of progressive alignment: (1) distance matrix; (2) guided tree; (3) profile-profile progressive alignment.

Even though, the progressive alignment is much more efficient than dynamic programming, it still suffers from a high computational complexity. Much research work has been done to accelerate the execution of progressive alignment tools, especially ClustalW, using parallelization. These solutions can be compared from the aspects of granularity, target architecture, ClustalW stages parallelized and parallel programming model used.

Coarse-grained parallel versions of ClustalW have been designed to target shared memory multiprocessors, distributed memory workstation clusters, symmetric multiprocessors (SMPs) and SMP clusters. A commercial parallel version of ClustalW presented by SGI [5] is designed for expensive SGI shared memory multiprocessor machines. ClustalW-MPI [6], Ebedes et al. [7] and pCLUSTAL [8] all target distributed memory workstation clusters using MPI but parallelize only Stages 1 and 3 of ClustalW. ClustalW-SMP [9] is an SMP version of ClustalW designed for SMP machines, which is written using the Pthreads library and parallelized all the three stages of ClustalW. Tan et al. [10] presented a parallel ClustalW running on an SMP cluster by means of a mixed approach using both MPI and OpenMP, but only parallelized Stages 1 and 3 of ClustalW.

Fine-grained parallelization approaches focus on multi-core processors and accelerators such as FPGAs and GPUs. MT-ClustalW [11] is designed to target multi-core processors but merely re-parallelized Stage 2 using the Pthreads library on the basis of ClustalW-SMP. Oliver et al.

1063-6862/09 $25.00 © 2009 IEEE
DOI 10.1109/ASAP.2009.14

[12] [13] constructed a linear systolic array to perform Stage 1 of ClustalW on a standard FPGA using Verilog HDL. GPU-ClustalW [14] reformulated the dynamic programming pairwise alignment algorithm and is implemented using OpenGL on commercial graphics cards but only parallelized Stage 1 of ClustalW.

Besides ClustalW, parallel solutions for other MSA tools have also been developed. Zola et al. [15] presents a parallel implementation of T-Coffee running on workstation clusters using MPI, which parallelizes the library generation stage and the progressive alignment stage. Deng et al. [16] parallelized three modules (FRA, PMC and CT) which take most of the runtime of MUSCLE with OpenMP. Boukerche et al. [17] designed an FPGA-based hardware accelerator to execute the most compute-intensive part of DIALIGN.

In this paper, we show how the CUDA programming model can be used to parallelize all three stages of ClustalW in Sections 2 to 5. Section 6 evaluates the performance of the CUDA-based approach to ClustalW 2.0.9 and ClustalW-MPI. Finally, Section 7 concludes the paper.

II. CUDA Programming Model

CUDA (Compute Unified Device Architecture) is an extension of C/C++ which enables users to write scalable multi-threaded programs for CUDA-enabled GPUs [18]. CUDA programs can be executed on GPUs with NVIDIA's Tesla unified computing architecture [19].

CUDA programs contain a sequential part, called a *kernel*. The kernel is written in conventional scalar C-code. It represents the operations to be performed by a single thread and is invoked as a set of concurrently executing threads. These threads are organized in a hierarchy consisting of so-called thread blocks and grids. A *thread block* is a set of concurrent threads and a *grid* is a set of independent thread blocks. The total size of a grid (*dimGrid*) and a thread block (*dimBlock*) is explicitly specified in the kernel function-call:

kernel<<<*dimGrid*, *dimBlock*, … >>> (parameters);

The hierarchical organization into blocks and grids has implications for thread communication and synchronization. Threads within a thread block can communicate through a *per-block shared memory* (PBSM) and may synchronize using barriers. However, threads located in different blocks cannot communicate or synchronize directly. Besides the PBSM, there are four other types of memory: per-thread private local 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.

The Tesla architecture supports CUDA applications using a scalable processor array. The array consists of a number of *streaming multiprocessors* (SMs). Each SM contains eight scalar processors (SPs), which share a PBSM of size 16 KB. All threads of a thread block are executed concurrently on a single SM. The SM executes threads in small groups of 32, called *warps*, in single-instruction multiple-thread (SIMT) fashion. Thus, parallel performance is generally penalized by data-dependent conditional branches and improves if all threads in a warp follow the same execution path.

III. Parallelization of Pairwise Distance Computation

Given two sequences S_a and S_b of lengths l_a and l_b respectively, their distance $d(S_a, S_b)$ is defined as:

$$d(S_a, S_b) = 1 - \frac{nid(S_a, S_b)}{\min\{l_a, l_b\}} \quad (1)$$

where $nid(S_a, S_b)$ denotes the number of exact matches in the optimal local alignment of S_a and S_b. The value $nid(S_a, S_b)$ can be computed in linear space using three passes: a forward score-only pass using Smith-Waterman (SW) algorithm [20] [21], a reverse score-only pass using SW algorithm and a traceback computation pass using Myers-Miller algorithm [22].

We are using the following notations for the SW algorithm: a substitution table *sbt*, a gap opening penalty ρ, and a gap extension penalty σ, the following recurrences for $1 \le i \le l_a$, $1 \le j \le l_b$:

$$\begin{aligned} E(i,j) &= \max\{E(i-1,j)-\sigma, H(i-1,j)-\rho-\sigma\} \\ F(i,j) &= \max\{F(i,j-1)-\sigma, H(i,j-1)-\rho-\sigma\} \\ H(i,j) &= \max\{0, E(i,j), F(i,j), H(i-1,j-1) \\ &\quad + sbt(S_a[i], S_b[j])\} \end{aligned} \quad (2)$$

The recurrences are initialized as $H(i, 0) = H(0, j) = E(0, j) = F(i, 0) = 0$ for $0 \le i \le l_a$ and $0 \le j \le l_b$. The maximum local alignment score *maxScore* is defined as the maximal value in matrix *H*. The three arrows in Fig. 2 show the data dependencies in the alignment matrix: each cell depends on its left, upper, and upper-left neighbors. This dependency implies that all cells on the same minor diagonal in the alignment matrix are independent from each other and can be computed in parallel (also shown in Fig. 2). Thus, the alignment can be computed in minor-diagonal order from the top-left corner to the bottom-right corner in the alignment matrix. Note that, in order to calculate minor diagonal *i* only the results of the minor diagonal *i*–1 and *i*–2 are necessary and therefore *maxScore* can be found in linear space.

The actual optimal alignment path can be found in linear space by computing a traceback with the Myers-Miller

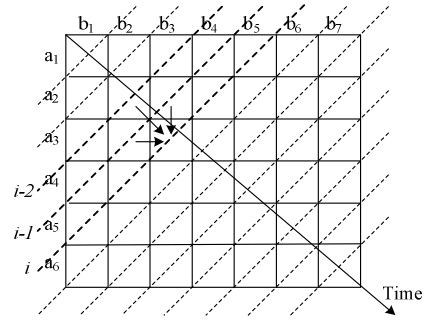

Figure 2. Cells on the same minor diagonal (dashed line) can be computed in parallel and the alignment matrix can be computed in minor diagonal order.

algorithm. The central idea of Myers-Miller is to find the "optimal midpoint" of an optimal alignment using a forward and a reverse pass. By recursively calculating optimal midpoints on both sides of this "optimal midpoint", the complete traceback path can be found. The sequential implementation of this algorithm uses a recursive divide-and-conquer method. However, CUDA does not support recursion. Therefore, we have developed a new stack-based iterative implementation shown in Fig. 3. MSA-CUDA uses this implementation for both pairwise alignments in Stage 1 and profile-profile alignments in Stage 3.

Considering the pairwise distance computation of one pair of sequences as a task, we have investigated two approaches for parallelizing Stage 1 using CUDA.

- *Inter-task parallelization*. Each task is assigned to exactly one thread and *dimBlock* tasks are performed in parallel by different threads within the thread block.
- *Intra-task parallelization*: Each task is assigned to a whole thread block and all *dimBlock* threads in the thread block cooperate to perform the task in parallel, exploiting the parallel characteristics of cells in the minor diagonals as shown in Fig. 2.

In order to achieve high efficiency for inter-task parallelization, the runtime of all threads in a thread block should be roughly identical. We therefore order the input sequences based on their lengths. Hence, for two adjacent threads in a thread block, the difference value between the products of the lengths of the associated sequences is minimized.

```
typedef struct tagNode
{
   int A, B, M, N;
   int tb, te, branch, type;
} Node;
typedef struct tagStack
{
   int  top;
   nodes [MAX_STACK_DEPTH];
}Stack;
#define NONE (-1)    //nothing to do
#define PREFIX  0    //upper-left section of the "optimal midpoint"
#define SUFFIX  1    //lower-right section of the "optimal midpoint"
#define TYPE1   1    //type 1 "optimal midpoint"
#define TYPE2   2    //type2 "optimal midpoint"
Iterative procedure diff ( A, B, M, N, tb, te)
{
   Stack stack;
   Node node;

   stack_init ( &stack);
   node_init ( &node, A, B, M, N, tb, te, NONE,NONE);
   stack_push (&stack, &node);
   while (!stack_empty (&stack)) {
      Node* tmp = stack_pop(&stack);
      A = tmp->A;  B = tmp->B; M = tmp->M;  N = tmp->N; tb = tmp->tb;  te = tmp->te;
      //if the type of the mid point is TYPE2
      if (tmp->type == TYPE2 && tmp->branch = SUFFIX) {
         del (2);  //deleting "a_mini a_mini + 1"
      }

      Handle the boundary cases N=0 and M≤1 by examining all possible optimal alignments;
      //compute and find the "optimal midpoint" (midi, midj)
      midi = M/2;
      Compute HH and DD in a forward phase using the enhanced SW Algorithm;
      Compute RR and SS in a reverse phase using the Enhanced SW Algorithm;
      Find minj∈[0, N] by min_{j ∈ [0, N]} {min (HH[j]+RR[j], DD[j]+SS[j]-_gapOpen)};

      //divide and conquer around the "optimal midpoint" (midi, midj)
      type = the type of the "optimal midpoint" (midi, midj);
      if (type == TYPE1){
         node_init (&node, A+midi, B+midj, M–midi, N–midj, _gapOpen, te, type, SUFFIX);
         stack_push(&stack, &node);
         node_init (&node, A, B, midi, midj, tb, _gapOpen, type, PREFIX);
         stack_push (&stack, &node);
      }else{ //definitely TYPE2
         node_init (&node, A+midi+1, B+midj, M–midj–1, N–midj, 0, te, type, SUFFIX);
         stack_push(&stack, &node);
         node_init (&node, A, B, midi – 1, midj, tb, 0, type, PREFIX);
         stack_push(&stack, &node);
      }
   }
}
```

Figure 3. Pseudocode of the stack-based iterative implementation of the Myers-Miller algorithm.

During the execution of pairwise distance computation, additional memory is required to store intermediate alignment data. The size of this memory is $O(\min\{l_a, l_b\})$ for the two parallelization, given two sequences of length l_a and l_b (e.g. Stage 1 requires about $16 \times \min\{l_a, l_b\}$ bytes for inter-task parallelization and about $40 \times \min\{l_a, l_b\}$ bytes for intra-task parallelization). To support much longer sequences, the global memory is used to store the immediate results. The two approaches work in a multi-pass fashion, where in every pass, a grid consisting of thread blocks whose number is equal to or less than the number of SMs are bound to the corresponding kernel and launched, and the memory allocated for one pass is multiplexed by the successive following passes, reducing the requirements for global memory. For inter-task parallelization, the total amount of required memory for n input sequences of average length l_{ave} can be estimated as:

$$dimBlock \times SM\ number \times O(l_{ave}) \text{ bytes} \quad (3)$$

For intra-task parallelization, the total amount of required memory for the same input sequences can be estimated as:

$$SM\ number \times O(l_{ave}) \text{ bytes} \quad (4)$$

To gain maximum bandwidth and best performance, all threads in a half-warp should access the intermediate results in global memory in a coalesced pattern. A prerequisite for coalescing is that the words accessed by all threads in a half-warp must lie in the same segment. The memory spaces referred to by the same variable names (not referring to same addresses) for all threads in a half-warp have to be allocated in the form of an array to keep them contiguous in address. Fig. 4 presents two global memory allocation patterns of a basic type vector variable of size N for M processing entities (threads or thread blocks, here). Inter-task parallelization exploits the pattern shown in Fig. 4 (a), where a memory slot is allocated to a thread in a thread block and is indexed top-to-bottom, and the access to *MemSlot* using the same index for all threads in a half-warp is coalesced into one or two memory transactions depending on the compute capability of devices. Intra-task parallelization exploits the pattern shown in Fig. 4 (b), where a memory slot is allocated to a thread block and is indexed left-to-right, and the coalesced access is able to be obtained using the common global memory access pattern, i.e. that successive threads access the successive addresses in a memory slot.

To maximize performance and to reduce the bandwidth demand of global memory, we propose a cell block division method for the forward score-only pass when using inter-task parallelization, where the alignment matrix is divided into cell blocks of equal size. A cell block is a square matrix of size $n \times n$. Assume that the lengths of a pair of sequences

are l_a and l_b, respectively. In this case, l_a and l_b must be multiples of n. If the length is not a multiple of n, the sequence is padded with dummy symbols. To keep *maxScore* unchanged, the dummy symbol is added to the substitution table and the score between the dummy symbol and itself or a real symbol is set to zero. For simplicity, assume that l_a and l_b are multiples of n. Without cell block division, the computation of $H(i, j)$ results in one load operation and one store operation for the intermediate results stored in the global memory. We define the runtime of one load operation to be T_l, the runtime of one store operation to be T_s and the computation time of one cell value to be T_c. Then, without cell block division, the total runtime can be estimated as:

$$l_a \times l_b \times \left(T_l + T_s + T_c\right) \tag{5}$$

However, when using the cell block division method, the computation of n cells in one column (or row) in a cell block only requires one load operation and one store operation on the global memory instead of n load operations and n store operations. In this case, the total runtime can be estimated as:

$$l_a \times l_b \times \left(\frac{1}{n}\left(T_l + T_s\right) + T_c\right) \tag{6}$$

Since one global memory access takes hundreds of clock cycles, the cell block division method leads to a signification reduction of the total runtime due to a reduction in the global memory accesses. However, the size of cell block is limited by the amount of shared memory (or registers) available per thread. Therefore, this leads to the optimal cell block size of 8 × 8 for our implementation.

The multi-pass inter-task parallelization requires a large number of tasks to be efficient. If there are only a few tasks available (e.g. ≤ 100), intra-task parallelization is preferable. The intra-task parallelization executes in a model similar to the execution model of OpenMP. For a loop that can be parallelized between iterations, it is divided into separate iterations and distributes them to threads or warps in a thread block. For Stage 1, one pairwise distance computation is assigned to a whole thread block and computed by all threads in the thread block along the time axes from the top-left corner to the bottom-right corner in the alignment matrix (shown in Fig. 2). In each pass, all cells on a minor diagonal are computed in parallel by all threads, one of which is assigned to compute a separate group of cells. Due to the independent execution of different warps, barrier synchronization is necessary before starting the computation of the next minor diagonal after the completion of the current one.

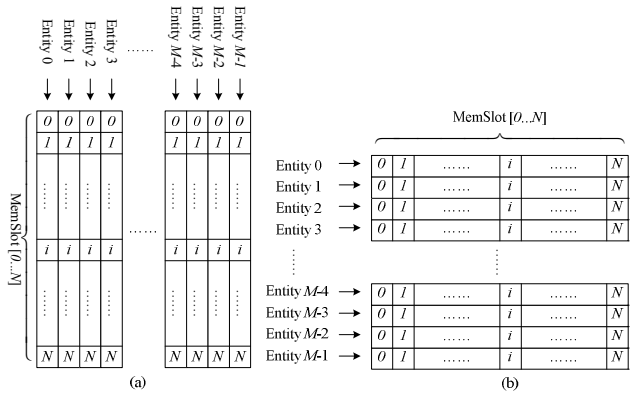

Figure 4. Two global memory allocation patterns of a basic type vector variable of size N for M processing entities (threads or thread blocks).

Obviously, if the sequence length is relatively large and almost no threads in the thread block are idle, a good performance can be gained, albeit the irregular number of cells on different minor diagonals. However, if the sequence length is relatively small, many threads in the thread block will be idle for much of the runtime or even all the time, which results in a poor performance since the access latency of the global memory cannot be offset by overlapping with computing.

Inter-task and intra-task parallelization both use constant memory to store read-only parameters and the substitution stable. The substitution table is loaded into shared memory, as the performance of constant memory degrades linearly if multiple addresses are requested by threads. This is because threads may frequently access different addresses in the substitution table. Texture memory is used to store the ordered input sequences. The symbols of a sequence are restricted to be stored in the same row of the texture array. All sequences are sequentially stored in the array row by row from top-left to bottom-right. A hash table records the location coordinate in the texture array and length of each sequence and provides the fast access to any sequence.

IV. Parallelization of Neighbor-Joining Tree

In ClustalW, the guided tree is reconstructed using the neighbor-joining (NJ) method [23] [24]. Stage 2 can be further divided into two sub-stages:

- *Stage 2a*: Reconstruction of the unrooted NJ tree (*NJ tree reconstruction*);
- *Stage 2b*: Rooting the NJ tree and computing sequence weights (*NJ tree rerooting*).

The improved compact memory algorithm, which we present in detail in [25], is used for the NJ tree reconstruction sub-stage. After the reconstruction of NJ tree, *Stage 2b* starts to reroot the unrooted NJ tree, to recalculate the weights of sequences and to traverse the rooted tree to identify the alignment steps for Stage 3. The unrooted NJ tree is rerooted using a "mid-point" method [26]. The root is placed at the position where the means of branch lengths on either side of the root are identical. The position is determined using the following algorithm:

- Every node of the NJ tree is iteratively selected as the reference node;
- Determine which leaf node is on the left or on the right of the selected node. If a leaf node is not a descendent of the selected node, it is positioned on the left; otherwise, on the right. The distance

between each leaf node and the selected node is then computed. If the selected node is an ancestor of a leaf node, the distance between them can be computed directly by summing the length of each branch along the path from the leaf node to the selected node; otherwise, the distance is the sum of the branch lengths from either of them to their common ancestor;

- Compute the difference value between the means of branch lengths on the left and on the right of the selected node;
- Exhaustively compute the difference values between the means of branch lengths on the left and on the right for all nodes and then select the node that gives the minimum positive difference value (including 0) and produces the shallowest tree;
- Insert a new root as the ancestor of the node selected in the previous step.

For the CUDA implementation, all the tree nodes are stored in a vector and the relationship between nodes is maintained through vector indices instead of pointers. Each node object stores the indices of itself, its parent and its left and right children, and accesses them using vector index. One thread block is assigned to compute the difference value of the means of branch lengths on the left and on the right of one node, which is selected as the reference. Every thread in the thread block is assigned to perform the computation on a separate sub-set of leaf nodes. For each leaf node in a sub-set, the corresponding thread identifies on which side of the selected node this leaf node lies and computes the distance between this leaf node and the selected node. Shared memory is exploited to store the results of all threads in a thread block and texture memory is used to store the tree structure.

V. Parallelization of Progressive Alignment

The final stage performs profile-profile alignments following the rooted guided tree from the leaves up to the root. Every leaf node of the guided tree corresponds to a sequence and each internal node corresponds to a profile-profile alignment produced from the aligned sequences in the left sub-tree and in the right sub-tree. The alignment corresponding to an internal node can be launched if and only if the alignments corresponding to the roots of its left and right sub-trees have been performed. Obviously, the alignments at the same level of the guided tree can be performed in parallel but even alignments that are not at the same level could also be parallelized. For example, in Fig. 5, all alignments with the same patterns can be performed in parallel.

Initially, the rooted guided tree is depth-first traversed in post-order to number all the internal nodes and build the dependency relationship with their left and right sub-trees. All internal tree nodes are stored in a vector in traversal-order. For all tree nodes, three auxiliary vectors are used to record the indices of their children, the indices of their right children and a flag indicating whether the corresponding alignments has been performed. For a leaf node, the indices

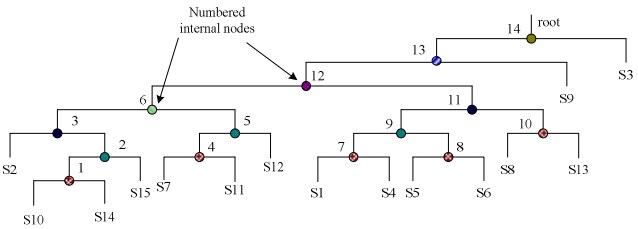

Figure 5. Example of a rooted guided tree produced by the NJ method.

of its left and right children are set to 0. For an internal node, if one child is a leaf, then the index of this child is also set to 0. The dummy sub-tree numbered as 0 is always defined aligned since it corresponds to an input sequence for an alignment. Fig. 6 presents the three initial auxiliary vectors for the rooted guided tree shown in Fig. 5.

In MSA-CUDA, the progressive alignment is conducted iteratively in a multi-pass way. For each pass, firstly, all undone alignments that are able to be performed in this pass are identified by checking the flag words of their left and right children stored in the flag-vector. If both of its left and right children have been aligned, this alignment is added to the ready alignment list managing all the alignments to be performed in this pass; otherwise, this alignment has to wait until both of its children have been aligned. After the completion of the ready alignment list, the pairs of profiles corresponding to those alignments are constructed. Secondly, the pairwise alignments of all pairs of profiles are performed on the GPU in parallel. Thirdly, gaps are added to the sequences corresponding to each pair of profiles by tracing back its optimal alignment. Finally, all the alignments performed in this pass will set their flag words in the flag-vector to indicate that they are aligned.

As illustrated in Fig. 5, the guided tree is seldom well-balanced and the numbers of alignments that can be performed in one pass decreases as the alignments move up to the root of the tree. Therefore, MSA-CUDA uses the following parallelization strategy. When the number of alignments to be performed in one pass is relatively large, an inter-task parallelization method is utilized; and when it is relatively small, an intra-task parallelization method is superior. Thus, a combination of inter-task and intra-task parallelization is used to compute all the alignments to be performed in one pass. A *threshold* determines the branches of the program flow. If the total number of alignments or the remaining number of alignments after one or more passes is still greater than or equal to *threshold*, the inter-task parallelization method is used; otherwise, the intra-task parallelization method is used to compute those remaining

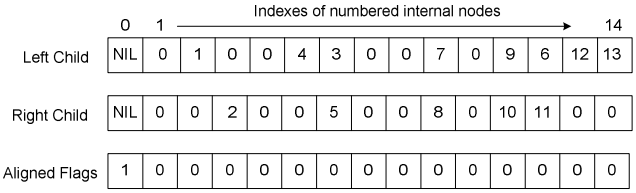

Figure 6. Three initial auxiliary vectors storing the dependency relationship with their left and right sub-trees and the aligned flags.

alignments.

Constant memory is exploited to store all read-only parameters. Since any profile-profile alignment has a different substitution table, texture memory is used to store the substitution tables. Substitution tables are then loaded into shared memory from texture memory during the kernel runtime. Texture memory is also used to store the 2D profiles of each alignment. A hash table records the location coordinate in the texture array, the width and height of each profile, and provides fast access to any profile.

VI. Performance Evaluation

MSA-CUDA is benchmarked on an nVIDIA GeForce GTX 280 graphics card, with 30 SMs comprising 240 SPs and 1 GB RAM, installed in a PC with an AMD Opteron 248 2.2 GHz processor running the Linux OS. The sequential ClustalW (version 2.0.9) program is profiled on a desktop PC with a Pentium 4 3.0 GHz processor and 1 GB RAM running the Linux OS. ClustalW-MPI [27] is benchmarked on a workstation cluster with 16 nodes connected through a fast 10 Gb/s InfiniBand switch. Each node is equipped with a dual-core Intel Xeon 3.0 GHz processor and 4 GB RAM running the Linux OS.

Three kinds of protein sequence datasets are used to evaluate the performance of MSA-CUDA. They are further subdivided into two representative datasets with different numbers of sequences. The datasets consist of sequences selected from the *Human immunodeficiency virus* dataset downloaded from NCBI [28], as given below:

- *Case 1: Small number of long sequences.* 400 sequences of average length 856 and 1,000 sequences of average length 858;
- *Case 2: Medium number of average-length sequences*. 2,000 sequences of average length 266 and 4,000 sequences of average length 247;
- *Case 3: Large number of short sequences*. 4,000 sequences of average length 57 and 8,000 sequences of average length 73.

Fig. 7 shows the performance comparison of the inter-task and intra-task parallelization for Stage 1. The graph clearly shows that the inter-task parallelization outperforms the intra-task parallelization for all datasets. Using the inter-task parallelization, the highest and the lowest speedups are 47.13 and 23.82. The average speedups are 46.15, 25.94, and 24.98 for Case 1, 2, and 3, respectively. Thus, if there are sufficient tasks and available large device memory capacity on the GPU, MSA-CUDA chooses inter-task parallelization for Stage 1. In general, the highest speedup is achieved for Case 1 datasets. This can be explained by the larger amount of computation performed compared to Cases 2 and 3.

Speedups for the NJ tree reconstruction sub-stage generally increase with the number of input sequences, but grow more slowly for the larger datasets (see Fig. 8). Consequently, the highest speedup of 23.20 is achieved using the dataset of 8,000 sequences. As expected, the sequence length has little impact on the runtime. Speedups for the NJ tree rerooting sub-stage are relatively low, the highest and the lowest speedups are 4.03 and 2.49. The

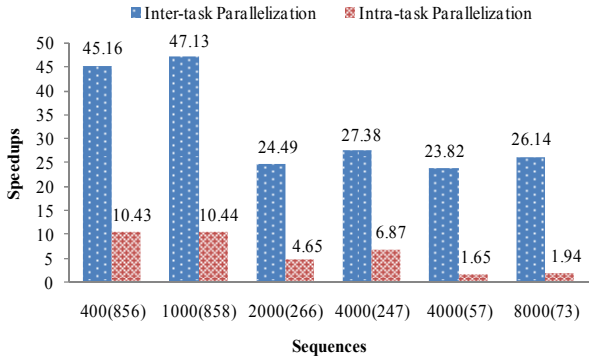

Figure 7. Speedup comparison of inter-task and intra-task parallelization for Stage 1.

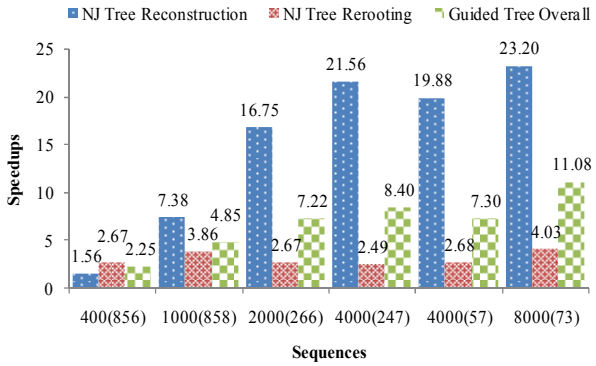

Figure 8. Speedups for Stage 2.

speedup of this sub-stage depends on the number of sequences as well as the tree topology. The overall speedup of Stage 2 is mainly subject to the speedup of the NJ tree reconstruction sub-stage since it dominates the total runtime.

The speedups for Stage 3 vary largely (see Fig. 9), ranging from 1.35 to 5.94. There are several reasons for this. Firstly, the building of the profiles of each alignment is performed sequentially on the CPU, reducing the speedups achieved in the parallelized parts. Secondly, the speedup heavily depends on the topology of the guided tree. The topology greatly influences the number of alignments that can be processed in parallel. Thirdly, the lengths of the profiles of an alignment also have impact on performance. Generally, larger datasets and longer sequences mean better performances.

Fig. 10 presents the speedups of MSA-CUDA and ClustalW-MPI compared to the sequential ClustalW. MSA-CUDA achieves average overall speedups of 36.91, 18.74 and 11.27, respectively for Case 1, 2 and 3, and outperforms ClustalW-MPI for all test cases. The speedup for ClustalW-MPI is particularly poor for Case 3 since it exploits an older NJ tree reconstruction algorithm and does not parallelize it. However, even for Case 1 datasets, for which Stage 2 has a negligible runtime, MSA-CUDA on a single GPU is able to outperform ClustalW-MPI on 32 CPU cores by a small margin.

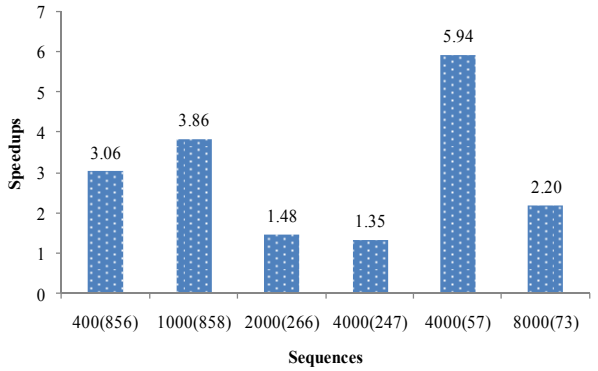

Figure 9. Speedups for Stage 3.

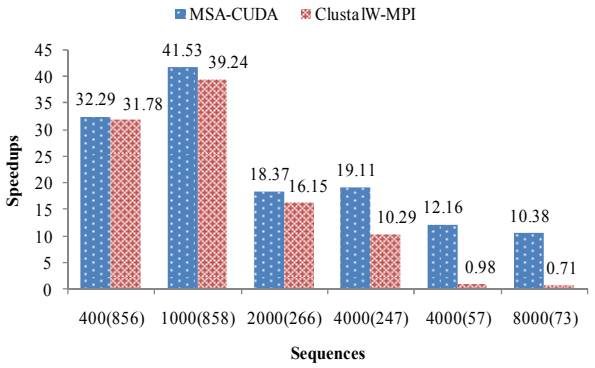

Figure 10. Speedup comparison between MSA-CUDA and ClustalW-MPI.

VII. CONCLUSIONS

MSA-CUDA demonstrates that CUDA-compatible graphics hardware provides a cost-effective high-speed solution to MSA. Through parallelization of all three stages of ClustalW, we have achieved average speedups of 36.91 (for long protein sequences), 18.74 (for average-length protein sequences), and 11.27 (for short protein sequences) on a single GPU, which is available for less than US$500 at any local computer outlet. These speedups also compare favorably to ClustalW-MPI on a high-performance compute cluster with 32 CPU cores. A comparison of these two parallelization approaches shows that GPU acceleration is clearly superior in terms of price/performance. The very rapid growth of biological sequence databases demands even more powerful high-performance solutions in the near future. Hence, our results are especially encouraging since GPU performance grows faster than Moore's law as it applies to CPUs.

ACKNOWLEDGMENT

We would like to thank Dr. Liu Weiguo and Dr. Shi Haixiang for helping to provide the experimental environments for conducting the tests.

REFERENCES

[1] D. Feng and R. Doolittle, "Progressive sequence alignment as a prerequisite to a correct phylogenetic Trees," J. Molecular Evolution, vol. 25, Aug. 1987, pp. 351 – 360, doi: 10.1007/BF02603120.

[2] C. Notredame, D.G. Higgins and J. Heringa, "T-Coffee: a novel method for fast and accurate multiple sequence alignment," J. Mol. Biol., vol. 302, Sep. 2000, pp. 205 – 217, doi: 10.1006/jmbi.2000.4042.

[3] R.C. Edgar, "MUSCLE: a multiple sequence alignment method with reduced time and space complexity," BMC Bioinformatics, vol. 5, Aug. 2004, doi: 10.1186/1471-2105-5-113.

[4] J.D. Thompson, D.G. Higgins and T.J. Gibson, "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, Nov. 1994, pp. 4673 – 4680.

[5] Silicon Graphics, Inc., http://www.sgi.com.

[6] K.B. Li, "Clustal-MPI: ClustalW analysis using parallel and distributed computing," Bioinformatics, vol. 19, Aug. 2003, pp. 1585-1586.

[7] J. Ebedes and A. Datta, "Multiple sequence alignment in parallel on a workstation cluster," Bioinformatics, vol. 20, Feb. 2004, pp. 1193-1195.

[8] J. Cheetham, et al., "Parallel ClustalW for PC clusters", International Conference on Computational Science and Its Applications (ICCSA 2003), LNCS, Jan. 2003, pp. 300–309, doi: 10.1007/3-540-44843-8.

[9] O. Duzlevski, "SMP version of ClustalW 1.82," unpublished.

[10] G. Tan, S. Feng and N. Sun, "Parallel multiple sequences alignment in SMP cluster," International Conference on High Performance Computing in Asia Region (HPC Asia 2005), IEEE Press, Jul. 2005, pp. 425 – 431, doi: 10.1109/HPCASIA.2005.70.

[11] K. Chaichoompu, S Kittitornkun and S. Tongsima, "MT-ClustalW: multithreading multiple sequence alignment," IEEE International Parallel and Distributed Processing Symposium (IPDPS 2006), IEEE Press, Apr. 2006, doi: 10.1109/IPDPS.2006.1639537.

[12] T. Oliver, B. Schmidt, D. Nathan, R. Clemens and D. Maskell, "Using reconfigurable hardware to accelerate multiple sequence alignment with ClustalW," Bioinformatics, vol. 21, May 2005, pp. 3431-3432, doi: 10.1093/bioinformatics/bti508.

[13] T. Oliver, B. Schmidt and D.L. Maskell, "Reconfigurable architectures for bio-sequence database scanning on FPGAs," IEEE Trans. Circuits Syst. II, vol. 52, Dec. 2005, pp. 851-855, doi: 10.1109/TCSII.2005.853340.

[14] W. Liu, B. Schmidt, G. Voss and W. Muller, "Streaming algorithms for biological sequence alignment on GPUs," IEEE Transactions on Parallel and Distributed Systems, vol. 18, Sep. 2007, pp. 1270-1281, doi: 10.1109/TPDS.2007.1069.

[15] J. Zola, X. Yang, S. Rospondek and S. Aluru, "Parallel T-Coffee: a parallel multiple sequence aligner," International Society of Computers and their Applications (ISCA 2007), pp. 248 – 253.

[16] X. Deng, E. Li, J. Shan, W. Chen, "Parallel implementation and performance characterization of MUSCLE,", IEEE International Parallel and Distributed Processing Symposium (IPDPS 2006), IEEE Press, April. 2006, doi: 10.1109/IPDPS.2006.1639616.

[17] A. Boukerche, J.M. Correa, A.C.M.A. Melo, R.P. Jacobi and A.F. Rocha, "An FPGA-based accelerator for multiple biological sequence alignment with DIALIGN," International Conference on High Performance Computing (HiPC 2007), LNCS, Jan. 2008, pp. 71-82, doi: 10.1007/978-3-540-77220-0_11.

[18] J. Nickolls J., I. Buck, M. Garland and K. Skadron, "Scalable parallel programming with CUDA," ACM Queue, vol. 6, Mar./Apri. 2008, pp. 40-53, doi: 10.1145/1365490.1365500.

[19] E. Lindholm, J. Nickolls, S. Oberman and J. Montrym, "NVIDIA Tesla: a unified graphics and computing architecture," IEEE Micro., vol. 28, Mar./Apri. 2008, pp. 39-55, doi: 10.1109/MM.2008.31.

[20] T. Smith and M. Waterman, "Identification of common molecular subsequences," J. Mol. Biol., vol. 147, Mar. 1981, pp. 195-197.

[21] O. Gotoh, "An improved algorithm for matching biological sequences," J. Mol. Biol., vol. 162, Dec. 1982, pp. 707-708.

[22] E.W. Myers and W. Miller, "Optimal alignments in linear space," Comput. Appl. Biosci., vol. 4, Mar. 1988, pp. 11-17.

[23] M. Saitou and N. Nei, "The neighbor-joining method: a new method for reconstructing phylogenetic trees," Mol. Biol. Evol., vol. 4, Jul. 1987, pp. 406-425.

[24] J.A. Studier and K.J. Keppler, "A note on the neighbor-joining algorithm of Saitou and Nei," Mol. Biol. Evol., vol. 5, Nov. 1988, pp. 729-731.

[25] Y. Liu, B. Schmidt and D.L. Maskell, "Parallel reconstruction of neighbor-joining trees for large multiple sequence alignments using CUDA," IEEE International Workshop on High Performance Computational Biology (HiCOMB 2009), in press.

[26] J.D. Thompson, D. Higgins and T.J. Gibson, "Improved sensitivity of profile searches through the use of sequence weights and gap excision", Comput. Appl. Biosci., vol. 10, Feb. 1994, pp. 19-29.

[27] ClustalW-MPI, http://www.bii.a-star.edu.sg/achievements/applications/clustalw/download.php.

[28] NCBI home page, http://www.ncbi.nlm.nih.gov.