Optimal Alignment of Three Sequences On A GPU

Junjie Li and Sanjay Ranka and Sartaj Sahni
Department of Computer and Information Science and Engineering
University of Florida, Gainesville, FL 32611, USA
(jl3, ranka, sahni)@cise.ufl.edu

Abstract

We develop two algorithms–layered and sloped–to align three sequences on a GPU. Our algorithms can be used to determine the alignment score as well as the actual alignment. Experiments conducted using an NVIDIA C2050 GPU show that our sloped algorithm is 3 times as fast as the layered one. Further, the sloped algorithm delivers a speedup of up to 90 relative to the single core algorithm running on our host CPU when determining the score of the best alignment and a speedup between 21 and 56 when computing the best alignment as well as its score.

1 Introduction

Multiple sequence alignment is a fundamental problem in bioinformatics. In this problem, we are given a set of N sequences $\Psi = \{S_0, S_1, \ldots, S_{N-1}\}$ where $|S_i| = l_i$ and $i \in \{0, 1, \ldots, N-1\}$. For DNA sequences, the alphabet for Ψ is the four letter set $\{A, C, G, T\}$ and for protein sequences, the alphabet is the 20 letter set $\{A, C-I, K-N, P-T, VWY\}$. We are to insert gaps into the sequences so that the resulting sequences are of the same length; in the resulting sequences, each character is said to be aligned with the gap or character in the corresponding position in each of the other sequences; and the alignment score is maximized. On a single core CPU, the best multiple sequence alignment (i.e., the one with maximum score) of Ψ can be found in $O(|S_0| * |S_1| * \ldots * |S_{N-1}|)$ time using dynamic programming [18]. In this paper, we focus on GPU algorithms for the optimal alignment of 3 sequences (i.e., $N = 3$).

Several papers have been written on three-sequence alignment. We briefly mention a few here. Murata et al. in [16] developed algorithms for the optimal alignment of 3 sequences using a constant gap weight. Gotoh [2] developed algorithms to align three sequences using an affine gap penalty model. Huang [5] reduced the memory requirement of these algorithms to be quadratic in the sequence length. Powell et al. [17] have developed a faster three-sequence alignment algorithm using a speed-up technique based on Ukkonen's greedy algorithm [19]. Hung et al. [6] use a position specific gap penalty model for three-sequence alignment. Yue et al. [21] propose a divide-and-conquer algorithm for three-sequence alignment and Lin et al. [10] propose a parallel algorithm for three-sequence alignment.

Besides being of interest in its own right, three-sequence alignment has been proposed as a base step in the alignment of a larger number of sequences. For example, Kruspe and Stadler [7] report improved accuracy when three-sequence alignment is used instead of pairwise sequence alignment in their progressive multiple sequence alignment program *aln3nn*. Three-sequence alignment also helps in the tree alignment problem [20].

In practice, heuristics that trade optimality for computational efficiency are often used when $N > 2$ due to the high computational complexity of multiple sequence alignment, However, with the advent of low-cost parallel computers, there is renewed interest in developing computationally practical algorithms that guarantee optimality of the constructed alignment and several researchers have developed efficient GPU algorithms for the optimal alignment of 2 sequences (e.g., [8], [9]). GPU adaptations for heuristic multiple sequence alignment have also been developed [3] [11] [12] [14] [13]. However, it appears that no algorithms for three-sequence alignment on a GPU have yet been developed.

In this paper, we develop two single-GPU algorithms for the optimal alignment of three sequences. The first of these uses a layering approach while the second uses a sloped approach. Experimental results using an NVIDIA GPU show that the sloped approach results in a faster algorithm and that this algorithm is an order of magnitude faster than a single-core alignment algorithm running on the host computer. The rest of the paper is organized as follows. In section 2, we review the NVIDIA GPU architecture used by us and in Section 3, we describe the optimal single-core algorithm for multiple sequence alignment for the case $N = 3$. In section 4, we describe our GPU adaptation of this algorithm for the case when we want to report only the score of the best alignment and in Section 5, we describe

our adaptation for the case when the best alignment as well as its score are to be reported. Experimental results are presented in Section 6 and we conclude in Section 7.

2 GPU Architecture

Our work targets the NVIDIA C2050 GPU. The C2050 comprises 448 processor cores grouped into 14 groups with 32 cores per group. Each group is called a SM (streaming multiprocessor). Each SM has 64KB of shared memory/L1 cache that may be set up as either 48KB of shared memory and 16KB of L1 cache or 16KB of shared memory and 48KB of L1 cache. In addition, each SM has 32K registers. The 14 SMs access a common 3GB of DRAM memory, called device or global memory, via a 768KB L2 cache. A C2050 is capable of performing up to 1.288 TFLOPS of single-precision operations and 515 GFLOPS of double precision operations. A C2050 connects to the host processor via a PCI-Express bus. The master-slave programming model in which one writes a program for the host or master computer and this program invokes kernels that execute on the GPU is supported. The programming language is CUDA, which is an extension of C to include GPU support. The key challenge in deriving high performance on this machine is to be able to effectively minimize the memory traffic between the SMs and the global memory of the GPU. Data that is used repeatedly should go to registers or shared memory while data that is used less frequently but of larger size should go to device memory. This effectively requires design of novel algorithmic and implementation approaches and is the main focus of this paper.

3 Three Sequence Alignment Algorithm

The input to the 3-sequence alignment problem is a set $\Psi = \{S_0, S_1, S_2\}$ of 3 sequences and a substitution matrix *sub* such as BLOSUM [4] or PAM [1], which defines the score for character pairs xy. The output is a $3 \times l$ matrix M where $l \geq \max(|S_0|, |S_1|, |S_2|)$. Row i of M is S_i, $0 \leq i < 3$, with gaps possibly inserted at various positions and such that each column of M has at least one non-gap character. M defines an alignment whose score is

$$score(M) = \sum_{i=0}^{l-1} obj(M[0][i], M[1][i], M[2][i])$$

In this paper, we define *obj* to be the sum-of-pairs function [18]:

$$\begin{aligned} obj(M[0][i], M[1][i], M[2][i]) = & \; sub[M[0][i]][M[1][i]] \\ & + sub[M[0][i]][M[2][i]] \\ & + sub[M[1][i]][M[2][i]] \end{aligned}$$

The alignment M is optimal iff it maximizes $score(M)$. The dynamic programming algorithm to construct M first computes an $(|S_0|+1)\times(|S_1|+1)\times(|S_2|+1)$ matrix H using the recurrences given below [18].

$$H[i][j][k] = \max \begin{cases} H[i-1][j-1][k-1] + obj(S_0[i-1], S_1[j-1], S_2[k-1]) \\ H[i-1][j-1][k] + obj(S_0[i-1], S_1[j-1], -) \\ H[i-1][j][k-1] + obj(S_0[i-1], -, S_2[k-1]) \\ H[i][j-1][k-1] + obj(-, S_1[j-1], S_2[k-1]) \\ H[i-1][j][k] + obj(S_0[i-1], -, -) \\ H[i][j-1][k] + obj(-, S_1[j-1], -) \\ H[i][j][k-1] + obj(-, -, S_2[k-1]) \end{cases}$$

where "$-$" denotes a GAP and β is the GAP penalty. The initial conditions are $(i > 0, j > 0, k > 0)$:

$$\begin{aligned} H[0][0][0] &= 0 \\ H[i][0][0] &= 2 \times i \times \beta \\ H[0][j][0] &= 2 \times j \times \beta \\ H[0][0][k] &= 2 \times k \times \beta \\ H[i][j][0] &= \max \begin{cases} H[i-1][j-1][0] + obj(S_0[i-1], S_1[j-1], -) \\ H[i-1][j][0] + obj(S_0[i-1], -, -) \\ H[i][j-1][0] + obj(-, S_1[j-1], -) \end{cases} \\ H[i][0][k] &= \max \begin{cases} H[i-1][0][k-1] + obj(S_0[i-1], -, S_2[k-1]) \\ H[i-1][0][k] + obj(S_0[i-1], -, -) \\ H[i][0][k-1] + obj(-, -, S_2[k-1]) \end{cases} \\ H[0][j][k] &= \max \begin{cases} H[0][j-1][k-1] + obj(-, S_1[j-1], S_2[k-1]) \\ H[0][j-1][k] + obj(-, S_1[j-1], -) \\ H[0][j][k-1] + obj(-, -, S_2[k-1]) \end{cases} \end{aligned}$$

The score of the optimal alignment is $H[|S_0|][|S_1|][|S_2|]$ and the corresponding alignment matrix M may be constructed using a backtrace process that starts at $H[|S_0|][|S_1|][|S_2|]$. The time required to compute the H values is $O(|S_0||S_1||S_2|)$. An additional $O(l)$ time is required to construct the optimal alignment matrix

4 Computing the Score of the Best Alignment

In this section, we describe two GPU algorithms, $LAYERED$ and $SLOPED$, to compute H.

4.1 Layered Algorithm

GPU Computational Strategy

In this algorithm, which is called $LAYERED$, the three-dimensional matrix H is partitioned into $s \times s \times$

$(|S_2|+1)$ chunks (cuboids) as shown in Figure 1, where s is an algorithm design parameter.

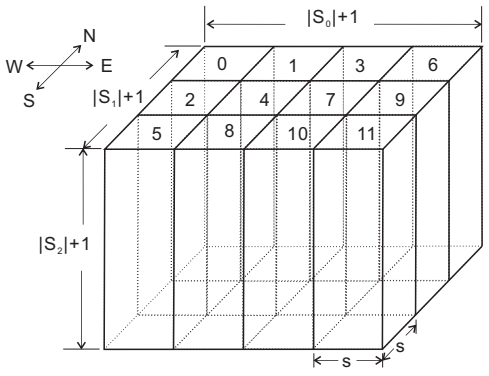

Figure 1: The partitioning of the three-dimensional matrix H

Let d be the number of chunks in the partitioning of H. These chunks form an $(|S_1|+1)/s \times (|S_0|+1)/s$ matrix whose elements may be numbered from 0 through $d-1$ in antidiagonal (i.e., top-right to bottom-left) order as in Figure 1. Let p be the number of SMs in the GPU (for the C2050, $p = 14$). SM i of the GPU will compute the H values for all chunks j such that $j \bmod p = i$, $0 \le j < c$, $0 \le i < p$.

Each SM works on its assigned chunks serially. For example, when $p = 3$ and $d = 12$, SM 0 is assigned chunks 0, 3, 6, and 9. This SM will first compute the H values for chunk 0, then for chunk 3, then for chunk 6, and finally for chunk 9. In general, SM i computes chunk i, followed by $i+p$ and so on.

The H values of a chunk are computed by the SM to which the chunk is assigned one layer at a time, where a layer is comprised of all H values in an $s \times s$ horizontal slice of the chunk. As can be seen, the total number of layers is $|S_2|+1$. Within a layer, the computations are done in antidiagonal order beginning with the antidiagonal that is comprised of the top left element of the layer (or slice). All values on the same antidiagonal of the layer are computed in parallel in a SIMD (singe instruction multiple data) fashion. When one layer has been computed, the SM moves to the next layer. When all layers have been computed, the SM moves to the next chunk assigned to it. In order to achieve high performance, the computed H values for a layer are stored in the shared memory of the SM until the computation of the next layer is complete as the current layer's H values are needed to compute those of the next layer. At any time during the computation, only two layers of H values are kept in shared memory and the memory space for these two layers is used in a round-robin fashion to save shared memory.

From the dynamic programming recurrence for H, we see that the H values in a chunk depend only on those on the shared boundary (east vertical face) of the chunks to its west and northwest and those on the shared boundary (south vertical face) of the chunks to its north and northwest. With respect to the $(|S_1|+1)/s \times (|S_0|+1)/s$ matrix of chunks, it is necessary for the SM that computes the H values for chunk (i,j) to communicate the computed H values on its east vertical face to the SM that will compute the chunks $(i,j+1)$ and $(i+1,j+1)$ (the top left corner of this matrix is indexed $(0,0)$ and the values on its south vertical face to the SM that will compute the chunks $(i+1,j)$ and $(i+1,j+1)$). The $s \times (|S_2|+1)$ H values on each of these vertical faces are communicated to the appropriate SMs via the GPU's global memory. Each SM writes the H values on its east and south faces to global memory. When an SM has completed the computation for its current layer, it polls the global memory to determine whether the boundary values need for the next layer are ready. If so, it proceeds to the next layer. If not, it idles. In case there is no next layer in the current chunk, the SM proceeds to its next assigned chunk.

Analysis

To run the layered algorithm, each SM requires $O(s^2)$ shared memory to store the values associated with a layer and $O(|S_0||S_1||S_2|/s)$ global memory to communicate the H values on its east and south faces. Since shared memory is very small on current GPUs, the available shared memory constrains the chunk size s. Because of the high cost of data transfer between SMs and global memory (relative to the cost of arithmetic), the run time performance of a GPU algorithm is often correlated to the volume of data transferred between the SMs and global memory. The layered algorithm transfers a total of $O(|S_0||S_1||S_2|/s)$ data between the GPU's global memory and the SMs.

The computational time (excluding time taken by the global memory I/O traffic) is computed by first noting that the c cores of an SM can do the computation for one layer in $O(s^2/c)$ time computing the H values on each antidiagonal of a layer in parallel. So, the time to do the computation for one chunk is $O(s^2|S_2|/c)$. Since the computation for the ith chunk cannot begin until the first layers of its west, north, and northwest neighbor chunks have been computed, each SM, other than SM 0, experiences a startup delay. As the antidiagonal from which the first chunk assigned to the last SM, SM $p-1$, is $O(\sqrt{p})$, the startup delay for SM $p-1$ is $O(\sqrt{p} * (s^2/c))$. When $|S_2|$ is sufficiently large (larger than $\lceil\sqrt{p}\rceil$) SMs experience (almost) no further delay in working on their assigned SMs. The number of chunks assigned to an SM is $O(|S_0||S_1|/(ps^2))$. So,

the total computation time (exclusive of global memory I/O time) is $O(|S_0||S_1||S_2|/(pc) + \sqrt{p} * (s^2/c))$.

While computation time exclusive of global I/O time increases as s increases (because the startup delay for SMs increases), global I/O time decreases as s increases. Our experiments show that for large $|S_0|$, $|S_1|$ and $|S_2|$, the reduction in global I/O memory traffic that comes from increasing s is substantially higher than the increase in time spent on computational tasks. Thus, within limits, choosing a large s will reduce the overall time requirements. As noted earlier, the value of s, is however, upper bounded by the amount of shared memory per SM.

4.2 Sloped Algorithm

GPU Computational Strategy

In this algorithm, which is called $SLOPED$, we partition H into $s \times s \times (|S_2| + 1)$ chunks and assign these chunks to SMs as in the layered algorithm. However, instead of computing the H values in a chunk by horizontal layers and within a layer by antidiagonals, the H values are computed by "sloped" planes comprised of $H[i][j][k]$s for which $q = i + j + k$ is the same. The first sloped plane has $q = 0$, the next has $q = 1$, and the last has $q = 2s + |S_2| - 2$. The computation for the plane $q + 1$ begins after that for the plane q completes. Within a plane q, the H values for all i, j, and k for which $i + j + k = q$ can be done in parallel. The number of parallel steps in the computation of a chunk is therefore $2s + |S_2| - 1$. In contrast, the layered algorithm cannot compute all H values in a chunk's layer in parallel. The computation of a layer is done by antidiagonals in $2s - 1$ steps with each step computing the values on one antidiagonal in parallel. The total number of parallel steps employed by the layered algorithm in the computation of a chunk is $(2s-1)*(|S_2|+1)$. Hence, when both the layered and sloped algorithms use the same s, the sloped algorithm uses fewer steps with each step comprised of more work that can be done in parallel. Under these conditions, the sloped algorithm is expected to perform better than the layered algorithm. However, as noted earlier, the value of s is constrained by the amount of local SM memory available. As we shall see below, the sloped algorithm requires more SM memory and so must use a smaller s.

Analysis

To compute the H values on a sloped plane q, we need the H values from the planes $q - 1$, $q - 2$, and $q - 3$. For efficient computation, we must therefore have adequate SM memory for 4 sloped planes. Since a sloped plane may have up to $(s+1)^2$ H values, each SM must have sufficient memory to accommodate $4(s+1)^2$ H values, which is twice the number of H values that the layered algorithm stores in the shared memory of an SM. The sloped algorithm generates the same amount of I/O traffic between the SMs and global memory as does the layered algorithm. The two algorithms also require the same amount of global memory.

Proceeding as for $LAYERED$, we see that the delay time for SM $p - 1$ is $O(\sqrt{p}(s^3/c))$ where $O(s^3/c)$ is the time taken to compute the small pyramid. The time to compute all the chunks assigned to an SM is $O(|S_0||S_1||S2|/(pc)$. So, the total time (exclusive of global I/O traffic time) is $O(|S_0||S_1||S_2|/(pc) + \sqrt{p} * (s^3/c))$. Although the analysis shows the total time for $SLOPED$ is larger than that for $LAYERED$ because of the larger delay to start SM $p-1$, when S_0, S_1, and S_2 are large, the greater parallelism afforded by $SLOPED$ coupled with the need for fewer synchronization steps dominates and the measured run time is smaller for $SLOPED$.

5 Computing the Best Alignment

The layered and sloped algorithms may be extended to compute not only the score of the best alignment but also the best alignment. We describe three possible extensions for the layered algorithm. These extensions represent a tradeoff among conceptual and implementation simplicity, computational requirements for the traceback done to compute the best alignment from the scores, and the amount of parallelism. Identical extensions may be made to the sloped algorithm.

5.1 $LAYERED - BT1$

To compute the best alignment, we need to maintain additional information that can be used during the traceback. In particular, for each position (i, j, k) of H, we associate the coordinates (i_s, j_s, k_s) of the local start point of the optimal path to (i, j, k). This local start point is a position on the boundary of north, northwest, or west neighbor chunk of the chunk that contains the position (i, j, k). $LAYERED - BT1$ is a 3-phase algorithm:

(1) Phase 1: This is an extension of $LAYERED$ in which each chunk stores, in global memory, not only the H values needed by its neighboring chunks but also the local start point of the optimal path to each boundary cell.

(2) Phase 2: For each chunk, we sequentially deter-

mine, the start point and end point of the subpath of the best alignment that goes through this chunk. This is done by tracing the best path backwards from position $(|S_0|, |S_1|, |S_2|)$ to position $(0, 0, 0)$ using the local start points of boundary cells computed in Phase 1. This traceback goes from the boundary of one chunk to the boundary of a neighbor chunk without actually entering any chunk.

(3) Phase 3: The subpath of the best path within each chunk is computed by recomputing the H values for the chunks through which the best alignment path traverses. Note that Phase 2 determines which chunks the best path goes through. Using the saved boundary H values, it is possible to compute the subpaths for all chunks in parallel.

The global memory required by $LAYERED - BT1$ is more than that required by $LAYERED$ as Phase 1 stores a 3D position with H value saved. Assuming 4 bytes for each coordinate of a 3D position and 4 bytes for an H value, $LAYERED - BT1$ requires 16 bytes per boundary cell while $LAYERED$ requires only 4. Additionally, in the Phase 3 computation, we need to store with every position in a chunk which of the 7 options on the right hand side of the dynamic programming recurrence for H resulted in the max value. This can be done using 3 bits (or more realistically, 1 byte) per position in a chunk.

5.2 $LAYERED - BT2$

Although $LAYERED - BT2$ is also a three phase algorithm, $LAYERED - BT2$ partitions each chunk into subchunks of height h (Figure 2). The three phases are as follows:

(1) Phase 1: Chunks are assigned to SMs as in $LAYERED$. In addition to the H and local start points stored in global memory by $LAYERED - BT1$, we store the H values of the positions on the bottom face of each subchunk.

(2) Phase 2: Use the local start point data stored in global memory by the Phase 1 computation to determine the start and end points of the subpaths of the best alignment path within each chunk. This phase computes the same start and end points as computed in Phase 2 of $LAYERED - BT1$.

(3) Phase 3: The subpath of the best path within each chunk is computed by recomputing the H values for the chunks through which the best alignment path traverses. For each chunk through which this path passes, the computation begins at the first layer of the topmost subchunk through which the path passes (see Figure 2, shaded subchunks are the subchunks through which the best alignment path passes). The computation for the different chunks through which the best alignment path passes can be done in parallel as in $LAYERED - BT1$.

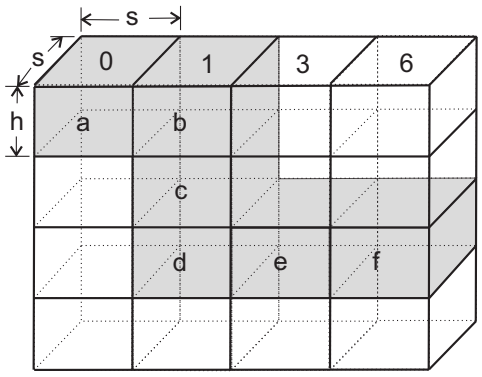

Figure 2: Some of the chunks traversed by the optimal path

The major differences between $LAYERED - BT1$ and $LAYERED - BT2$ are:

(1) Since $LAYERED - BT2$ saves H values on the bottom face of each subchunk in addition to data saved by $LAYERED - BT1$, it generates more I/O traffic than $LAYERED - BT1$ and also requires more global memory. The amount of additional I/O traffic and global memory required is $O(S_1||S_2||S_3|/h)$.

(2) For each chunk through which the best alignment path passes, $LAYERED - BT1$ begins the Phase 3 computations at layer 1 of that chunk while $LAYERED - BT2$ begins this computation at the first layer of the topmost subchunk through which this path passes.

5.3 $LAYERED - BT3$

For $LAYERED - BT3$, the local start points are defined to be positions on neighboring subchunks of the subchunk that contains the position (i, j, k) (rather than points on neighboring chunks of the chunk that contains (i, j, k)). The three phases of $LAYERED - BT3$ are:

(1) Phase 1: Chunks are assigned to SMs as in $LAYERED$. For each subchunk, we store in global memory, the H values on its south, east, and bottom faces as well as the local start point of

the optimal path to each of the positions on these faces.

(2) Phase 2: For each subchunk, we sequentially determine, the start point and end point (if any) of the subpath of the best alignment path that goes through this subchunk.

(3) Phase 3: The subpath (if any) of the best path within each subchunk is computed by recomputing the H values for those subchunks through which the best alignment path traverses. Note that Phase 2 determines the subchunks through which the best path goes. Using the saved boundary H values, it is possible to compute the subpaths for all subchunks in parallel.

$LAYERED - BT3$ has more global I/O traffic than $LAYERED - BT2$ in Phase 1 and also requires more global memory. Phase 2 of both algorithms do the same amount of work. We expect Phase 3 of $LAYERED - BT3$ to be faster than that of $LAYERED - BT2$ because of better load balancing resulting from the finer granularity of the per-subchunk work and the fact that there are more subchunks than chunks, which leads to more parallelism. For example, suppose one chunk has 50 subchunks through which the best alignment path passes and another chunk has only 1 such subchunk. Phase 3 of $LAYERED - BT2$ can use at most 2 SMs with 1 working on all 50 subchunks of the first chunk and the other working on the single subchunk of the second chunk. The workload over the two assigned SMs is unbalanced and the degree of parallelism is only 2. Phase 3 of $LAYERED - BT3$, however, is able to use up to 51 SMs assigning 1 subchunk to each SM resulting in better workload balancing and a higher degree of parallelism.

6 Experimental Results

In this section, we present experimental results for our scoring and alignment algorithms. All of these experiments were conducted on an NVIDIA Tesla C2050 GPU. The host machine has an Intel i7-x980 3.33GHz CPU and 12GB DDR3 RAM.

6.1 Computing the score of the best alignment

We fixed $s = 67$ for $LAYERED$ and $s = 47$ for $SLOPED$ and experimented with real instances of different sizes. These two s values were determined from our experimental results which produced the least running time. The sequences used were retrieved from NCBI Entrez Gene [15] and their sizes are represented as a tuple $(|S_0|, |S_1|, |S_2|)$. The running time is shown in Table 1. The running time of the single-core scoring method and the single-core traceback method running on our host CPU is referred to as *Scoring* and *Traceback* in Table 1, respectively, for comparison purpose. As can be seen, $SLOPED$ is about three times as fast as $LAYERED$ for large instances and is up to 90 times as fast as the single-core CPU algorithm! In our tests speedup increases with instance size.

Table 1: Running time (seconds) for different instances

	(113,166,364)	(347,349,365)	(267,439,452)	(764,771,773)	(1399,1404,1406)
$LAYERED$	0.034	0.165	0.162	1.250	6.643
$SLOPED$	0.044	0.087	0.090	0.416	1.921
Scoring	0.570	3.610	4.270	37.500	-
Traceback	0.630	4.010	4.760	41.600	-

6.2 Computing the alignment

Because of memory limitations, our scoring algorithms can handle sequences whose size is up to approximately $(2500, 2500, 2500)$ while our alignment algorithms can handle sequences of size up to approximately $(1500, 1500, 1500)$. We used $s = 47$, $h = 40$ for layered algorithms and $s = 31$, $h = 100$ for the sloped algorithms which were determined by experiments to be the optimal values.

We tested our alignment algorithms, with parameters set as above, on the real instances used earlier for the scoring experiments. The measured run times are reported in Table 2 and Figure 3 ($L-$ for $LAYERED$, $S-$ for $SLOPED$). We do not report the time for Phase 2 as this is negligible compared to that for the other phases. Although $BT2$ and $BT3$ reduce the run time of the third phase, the overall time is dominated by that for Phase 1. Again, the $SLOPED$ algorithms are about three times as fast as the $LAYERED$ algorithms. The $SLOPED$ algorithms provide a speedup between 21 to 56 relative to the single core algorithm running on our host CPU.

7 Conclusion

We have developed two GPU algorithms to optimally align three sequences. The sloped scoring algorithm, which is 3 times as fast as the layered algorithm, provides a speedup of up to 90 relative to a single-core algorithm running on our host CPU. The sloped alignment algorithm is also 3 times as fast as the layered one and provides a speedup between 21 and 56 relative to the single core algorithm. The strategies we used in this paper can also be extended to affine gap model though the computation will require more cubic

Table 2: Running time (seconds) of traceback methods for real instances

	(113,166,364)			(347,349,365)			(267,439,452)			(764,771,773)			(1399,1404,1406)		
	Phase 1	Phase 3	Total	Phase 1	Phase 3	Total	Phase 1	Phase 3	Total	Phase 1	Phase 3	Total	Phase 1	Phase 3	Total
$L-BT1$	0.062	0.012	0.076	0.298	0.042	0.342	0.348	0.035	0.386	2.814	0.121	2.941	14.994	0.277	15.300
$L-BT2$	0.062	0.005	0.069	0.298	0.010	0.310	0.347	0.010	0.359	2.813	0.014	2.834	14.982	0.021	15.036
$L-BT3$	0.062	0.005	0.069	0.298	0.010	0.310	0.347	0.009	0.360	2.812	0.016	2.836	14.977	0.023	15.040
$S-BT1$	0.013	0.002	0.030	0.064	0.005	0.111	0.077	0.006	0.125	0.583	0.021	0.736	3.513	0.058	3.850
$S-BT2$	0.014	0.002	0.032	0.067	0.002	0.114	0.080	0.002	0.127	0.606	0.003	0.748	3.648	0.005	3.946
$S-BT3$	0.014	0.002	0.032	0.067	0.002	0.114	0.080	0.002	0.127	0.604	0.004	0.747	3.636	0.006	3.939

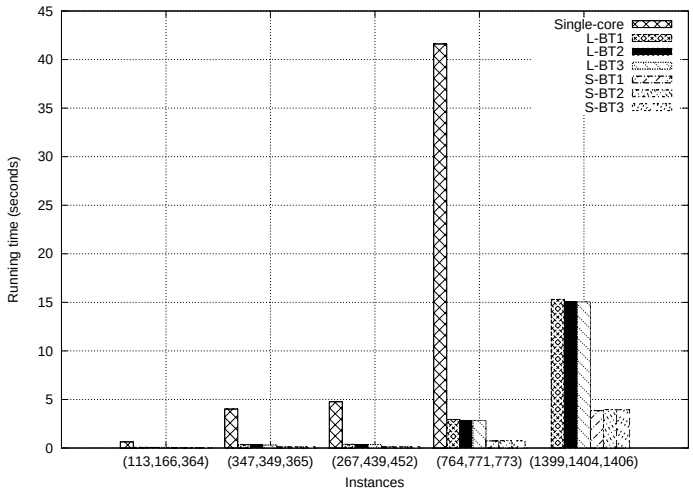

Figure 3: Plot of running time (seconds) of traceback methods for real instances

matrices to store temporary values.

8 Acknowledgment

This work was supported, in part, by the National Science Foundation under grants CNS0963812, CNS1115184, and the National Institutes of Health under grant R01-LM010101.

References

[1] M. O. Dayhoff, R. M. Schwartz, and B. C. Orcutt. A model of evolutionary change in proteins. *Atlas of protein sequence and structure*, 5(suppl 3):345–351, 1978.

[2] Osamu Gotoh. Alignment of three biological sequences with an efficient traceback procedure. *Journal of Theoretical Biology*, 121(3):327–337, 1986.

[3] Adam Gudy and Sebastian Deorowicz. A parallel gpu-designed algorithm for the constrained multiple sequence alignment problem. In Tadeusz Czachrski, Stanislaw Kozielski, and Urszula Stanczyk, editors, *Man-Machine Interactions 2*, volume 103 of *Advances in Intelligent and Soft Computing*, pages 361–368. Springer Berlin / Heidelberg, 2011.

[4] S. Henikoff and J. G. Henikoff. Amino acid substitution matrices from protein blocks. *Proceedings of the National Academy of Sciences of the United States of America*, 89(22):10915–10919, November 1992.

[5] Xiaoqiu Huang. Alignment of three sequences in quadratic space. *ACM SIGAPP Applied Computing Review*, 1(2):7–11, 1993.

[6] Che-Lun Hung, Chun-Yuan Lin, Yeh-Ching Chung, and Chuan Yi Tang. Introducing variable gap penalties into three-sequence alignment for protein sequences. In *Advanced Information Networking and Applications-Workshops, 2008. AINAW 2008. 22nd International Conference on*, pages 726–731. IEEE, 2008.

[7] Matthias Kruspe and Peter F Stadler. Progressive multiple sequence alignments from triplets. *BMC bioinformatics*, 8(1):254, 2007.

[8] Junjie Li, Sanjay Ranka, and Sartaj Sahni. Pairwise sequence alignment for very long sequences on gpus. *Computational Advances in Bio and Medical Sciences, IEEE International Conference on*, 0:1–6, 2012.

[9] Junjie Li, Sanjay Ranka, and Sartaj Sahni. Parallel syntenic alignment on gpus. In *Proceedings of the 3rd ACM Conference on Bioinformatics, Computational Biology and Biomedicine*, BCB '12. ACM, 2012.

[10] Chun Yuan Lin, Chen Tai Huang, Yeh-Ching Chung, and Chuan Yi Tang. Efficient parallel algorithm for optimal three-sequences alignment. In *Parallel Processing, 2007. ICPP 2007. International Conference on*, pages 14–14. IEEE, 2007.

[11] Cheng Ling, K. Benkrid, and A.T. Erdogan. High performance intra-task parallelization of multiple sequence alignments on cuda-compatible gpus.

In *Adaptive Hardware and Systems (AHS), 2011 NASA/ESA Conference on*, pages 360 –366, june 2011.

[12] W. Liu, B. Schmidt, G. Voss, and W. Muller-Wittig. Gpu-clustalw: Using graphics hardware to accelerate multiple sequence alignment. *High Performance Computing-HiPC 2006*, page 363374, 2006.

[13] W. Liu, B. Schmidt, G. Voss, and W. Muller-Wittig. Streaming algorithms for biological sequence alignment on gpus. *Parallel and Distributed Systems, IEEE Transactions on*, 18(9):1270 –1281, sept. 2007.

[14] Yongchao Liu, B. Schmidt, and D.L. Maskell. Msa-cuda: Multiple sequence alignment on graphics processing units with cuda. In *Application-specific Systems, Architectures and Processors, 2009. ASAP 2009. 20th IEEE International Conference on*, pages 121 –128, july 2009.

[15] Donna Maglott, Jim Ostell, Kim D. Pruitt, and Tatiana Tatusova. Entrez gene: gene-centered information at ncbi. *Nucleic Acids Research*, 33(suppl 1):D54–D58, 2005.

[16] M Murata, JS Richardson, and Joel L Sussman. Simultaneous comparison of three protein sequences. *Proceedings of the National Academy of Sciences*, 82(10):3073–3077, 1985.

[17] David R Powell, Lloyd Allison, and Trevor I Dix. Fast, optimal alignment of three sequences using linear gap costs. *Journal of Theoretical Biology*, 207(3):325–336, 2000.

[18] Bertil Schmidt. *Bioinformatics: High Performance Parallel Computer Architectures*. CRC Press, Inc., Boca Raton, FL, USA, 1st edition, 2010.

[19] Esko Ukkonen. On approximate string matching. In *Foundations of Computation Theory*, pages 487–495. Springer, 1983.

[20] Andrés Varón, Ward C Wheeler, et al. The tree alignment problem. *BMC bioinformatics*, 13:293, 2012.

[21] Feng Yue and Jijun Tang. A divide-and-conquer implementation of three sequence alignment and ancestor inference. In *Bioinformatics and Biomedicine, 2007. BIBM 2007. IEEE International Conference on*, pages 143–150. IEEE, 2007.