Pairwise Sequence Alignment for Very Long Sequences on GPUs

Junjie Li Sanjay Ranka Sartaj Sahni
Department of Computer and Information Science and Engineering
University of Florida
Gainesville, FL 32611
Email: {jl3,ranka,sahni}@cise.ufl.edu

***Abstract*—We develop novel single-GPU parallelizations of the Smith-Waterman algorithm for pairwise sequence alignment. Our algorithms, which are suitable for the alignment of a single pair of very long sequences, can be used to determine the alignment score as well as the actual alignment. Experimental results demonstrate an order of magnitude reduction in run time relative to competing GPU algorithms.**

***Keywords*-Long sequence alignment, local alignment, Smith-Waterman algorithm, CUDA, GPU.**

I. Introduction

Sequence alignment is a fundamental problem in bioinformatics. In its most elementary form, known as *pairwise sequence alignment*, we are given two sequences A and B and are to find their best alignment (either global or local). For DNA sequences, the alphabet for A and B is the four letter set $\{A, C, G, T\}$ and for protein sequences, the alphabet is the 20 letter set $\{A, C-I, K-N, P-T, VWY\}$. The best global and local alignments of the sequences A and B can be found in $O(|A| * |B|)$ time using the Needleman-Wunsch [1] and Smith-Waterman [2] dynamic programming algorithms. In this paper, we consider only the local alignment problem though our methods are readily extendable to the global alignment problem.

When the sequences A and B are long or when the number of sequences in the database D is large, computational efficiency is often achieved by replacing the Smith-Waterman algorithm with a heuristic that trades accuracy for computational time. This is done, for example in the sequence alignment systems BLAST [3], FASTA [4] and Sim2 [5]. However, with the advent of low-cost parallel computers, there is renewed interest in developing computationally practical systems that do not sacrifice accuracy.

Toward this end, several researchers have developed parallel versions of the Smith-Waterman algorithm that are suitable for Graphics Processing Units (GPUs) [6], [7], [8], [9], [10], [11]. Many of these implementations solve a variant of the pairwise sequence alignment problem that asks for the best k, $k > 0$, alignments. In the *database alignment problem*, we are to find the best k alignments of a sequence A with the sequences in a database D. The database alignment problem may be solved by solving $|D|$ pairwise alignment problems with each pair comprised of A and a distinct sequence from D. This requires $|D|$ applications of the Smith-Waterman algorithm. When sequences are small enough to fit the memory of the streaming processor (SM) of the GPU, each of these alignments can be computed independently on different SMs and can generate high performance. However, for large sequences, this cannot be easily achieved as only a portion of the computations can be stored in the SM memory. For example, our experiments indicate that $CUDASW++2.0$ [8] cannot handle strings whose length is more than 70000 on NVIDIA Tesla C2050.

The work of Khajej-Saeed, Poole, and Perot [9] and Sriwardena and Ranasinghe [10] is of particular relevance to us as this work specifically targets the alignment of two very long sequences. As noted by [9], biological applications often have $|A|$ in the range 10^4 to 10^5 and $|B|$ in the range 10^7 to 10^{10}. We refer to instances of this size as very large. Khajej-Saeed et al. [9] modify the Smith-Waterman dynamic programming equations to obtain a set of equations that are more amenable to parallel implementation. However, this modification introduces computational overhead. Despite this overhead, their algorithm is able to achieve a computational rate of up to 0.7 GCUPS (billion cell updates per second) using a single NVIDIA Tesla C2050. The instance sizes they experimented with had $|A| * |B|$ up to 10^{11}. Although Sriwardena and Ranasinghe [10] develop their GPU algorithms for pairwise sequence alignment specifically for the global alignment version, their algorithms are easily adapted to the case of local alignment. While their adaptations do not have the overheads of [9] that result from modifying the recurrence equations so as to increase parallelism, their algorithm is slower than that of [9].

In this paper, we develop single-GPU parallelizations of the unmodified Smith-Waterman algorithm and obtain a speedup of up to 17 relative to the single-GPU algorithm of [9] and a computational rate of 7.1 GCUPS. Our high-level parallelization strategy is similar to that used by Melo et al. [12] and Futamura et al. [13] to arrive at parallel algorithms for local alignment and syntenic alignment on a cluster of workstations, respectively. Both divide the scoring matrix into as many strips as there are processors and each processor computes the scoring matrix for its strip row wise. Melo et al. [12] do the traceback needed to determine the actual alignment serially using a single processor while Futamura et al.'s [13] do the traceback in parallel using

a strategy similar to the one used by us. The essential differences between our work and that of [12] and [13] are (a) our algorithms are optimized for a GPU rather than for a cluster, (b) we divide the scoring matrix into many more strips than the number of streaming multiprocessors in a GPU, and (c) the computation of a strip is done in parallel using many threads and the CUDA cores of a streaming multiprocessor rather than serially.

The rest of the paper is organized as follows. In section II, we review the NVIDIA GPU architecture used by us and in Section III, we describe the Smith-Waterman algorithm for pairwise sequence alignment. In section IV, we describe our GPU adaptation of the Smith-Waterman algorithm for the case when we want to report only the score of the best alignment and in Section V, we describe our adaptation for the case when the best alignment as well as its score are to be reported. Experimental results comparing the performance of our GPU adaptations with those of [9] and [10] are presented in Section VI and we conclude in section VII.

II. GPU Architecture

Our work targets the NVIDIA C2050 GPU. The C2050 comprises 448 processor cores grouped into 14 streaming multiprocessors (SM) with 32 cores per SM. 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 PCIexpress 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. This effectively requires design of novel algorithmic and implementation approaches and is the main focus of this paper.

III. Smith-Waterman Algorithm

Let $A = a_1a_2...a_m$ and $B = b_1b_2...b_n$ be the two sequences that are to be locally aligned. Let $c(a_i, b_j)$ be the score for matching or aligning a_i and b_j and let α be the gap opening penalty, and β the gap extension penalty. So, the penalty for a gap of length k is $\alpha + k\beta$. Gotoh's [14] variant of the Smith-Waterman dynamic programming algorithm with an affine penalty function uses the following three recurrences.

$$
\begin{aligned}
H(i,j) &= \max \begin{cases} H(i-1,j-1) + c(a_i, b_j) \\ E(i,j) \\ F(i,j) \\ 0 \end{cases} \\
E(i,j) &= \max \begin{cases} E(i-1,j) - \beta \\ H(i-1,j) - \alpha - \beta \end{cases} \\
F(i,j) &= \max \begin{cases} F(i,j-1) - \beta \\ H(i,j-1) - \alpha - \beta \end{cases} \\
& \quad where\ 1 \le i \le m,\ 1 \le j \le n
\end{aligned}
$$

Where the score matrices H, E, and F have the following meaning:

1) $H(i,j)$ is the score of the best local alignment for $(a_1...a_i)$ and $(b_1...b_j)$.
2) $E(i,j)$ is the score of the best local alignment for $(a_1...a_i)$ and $(b_1...b_j)$ under the constraint that a_i is aligned to a gap.
3) $F(i,j)$ is the score of the best local alignment for $(a_1...a_i)$ and $(b_1...b_j)$ under the constraint that b_j is aligned to a gap.

The initial conditions are: $H(0,0) = H(i,0) = H(0,j) = 0$; $E(0,0) = -\infty$; $E(i,0) = -\alpha - i\beta$; $E(0,j) = -\infty$; $F(0,0) = -\infty$; $F(i,0) = -\infty$; $F(0,j) = -\alpha - j\beta$; $1 \le i \le m,\ 1 \le j \le n$.

As mentioned in the introduction, the GPU adaptations of Khajej-Saeed, Poole, and Perot [9] and Sriwardena and Ranasinghe [10] are most suited for the pairwise alignment of very long sequences. Khajej-Saeed, Poole, and Perot [9] enhance parallelism by rewriting the recurrence equations. This rewrite eliminates the E terms and so their algorithm initially computes H values that differ from those computed by the original set of equations. Let H' be the computed H values. In a follow up step, modified E values, E', are computed. The correct H values are then computed in a final step from H' and E'. Although the resulting 3-step computation increases parallelism, it also increases I/O traffic between device memory and the SMs.

Sriwardena and Ranasinghe [10] propose two GPU algorithms for global alignment using the Needleman-Wunsch dynamic programming algorithm [1]. These strategies can be readily deployed for local alignment using Gotoh's variant of the Smith-Waterman algorithm. Both of the strategies of Sriwardena and Ranasinghe [10] are based on the observation that for any (i,j), the H, E, and F values depend only on values in the positions immediately to the north, northwest, and west of (i,j) (see Figure 1 (a)). Consequently, it is possible to compute all H, E, and F values on the same antidiagonal, in parallel, once these values have been computed for the preceding two antidiagonals. The first algorithm, *Antidiagonal*, of Sriwardena and Ranasinghe [10] does precisely this. The GPU kernel computes H, E, and F values on a single antidiagonal using values stored in device/global memory for the preceding two antidiagonals. The host program sends the two strings A and B to device

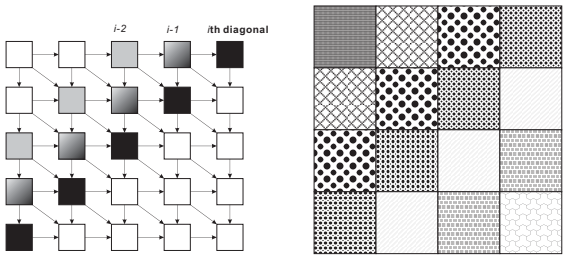

Figure 1. (a) Data dependency of Smith-Waterman algorithm; (b) Illustration of $BlockedAntidiagonal$

memory and then invokes the GPU kernel once for each of the $m+n-1$ antidiagonals. Additional (but minor) speedup can be attained by recognizing that the computation for the first and last few antidiagonals can be done faster on the host CPU and invoking the GPU kernel only for sufficiently large antidiagonals. When we desire to determine only the score of the best alignment, the device memory needed by $Antidiagonal$ is $O(\min\{m,n\})$. However, when the best alignment also is to be reported, the algorithm saves for each (i,j) the direction (north, northwest, west) that yielded the H value for this position. $O(mn)$ memory is required to save this information. Following the computation of the H, E, and F values a serial traceback is done to determine the best alignment.

The second GPU algorithm, $BlockedAntidiagonal$, of Sriwardena and Ranasinghe [10] partitions the H, E, and F values into $s \times s$ square blocks (see Figure 1 (b)) and employs a GPU kernel to compute the values for a block. The host program allocates blocks to SMs and each SM computes the H, E, and F values for its assigned block using values computed earlier and stored in device memory for the blocks immediately to its north, northwest, and west. Hence, $BlockedAntidiagonal$ attempts to enhance performance by utilizing both block-level parallelism and parallelism within an antidiagonal of a block. Experimental results reported in [10] demonstrate that $BlockedAntidiagonal$ is roughly two times faster than $Antidiagonal$. The $BlockedAntidiagonal$ strategy of Figure 1 (b) may be enhanced for the case when we are interested only in the score of the best alignemnt. In this enhancement, we write to global memory only the computed values for the bottom and right boundaries of each block. This reduces the global memory I/O traffic to $O(mn/s)$.

IV. Computing the Score of the Best Local Alignment

In our GPU adaptation, $StripedScore$, of the Smith-Waterman algorithm, we assume that $m \leq n$ (in case this is not so, simply swap the roles of A and B) and partition the scoring matrices H, E, and F into $\lceil n/s \rceil$ $m \times s$ strips (Figure 2). Here, s is the strip width. Let sm be the number of SMs in the GPU (for the C2050, $sm = 14$). The GPU kernel is written so that SM i computes the H, E, and F values for all strips j such that $j \mod sm = i$, $0 \leq j < \lceil n/s \rceil$, $0 \leq i < sm$. Each SM works on its assigned strips serially from left to right. That is, if SM 0 is assigned strips 0, 14, 28, and 42 (this is the case, for example when $sm = 14$, $s = 8$, and $n = 440$), SM 0 first computes all H, E, and F values for strip 0, then for strip 14, then for strip 28, and finally for strip 42. When computing the values for a strip, the SM computes by antidiagonals confined to the strip with values along the same antidiagonal computed in parallel. The computed values for each antidiagonal are stored in shared memory. Each SM uses three one-dimensional arrays (preceding two antidiagonals and current antidiagonal) residing in shared memory and one for each of E and F and one for swapping purpose. The size of each of these arrays is $O(\min\{m,s\})$. Additionally, each strip needs to communicate m H values and m F values to the strip immediately to its right. This communication is done via global memory. First each strip accumulates, in a buffer, a threshold number, T, of H and F values needed by its right adjacent strip in shared memory. When this threshold is reached, the accumulated H and F values are written to global memory. Each SM polls global memory to determine whether the next batch of H and F values it needs from its left adjacent strip are ready in global memory. If so, it reads this batch and computes the next T antidiagonals for its strip. If not, it waits in an idle state.

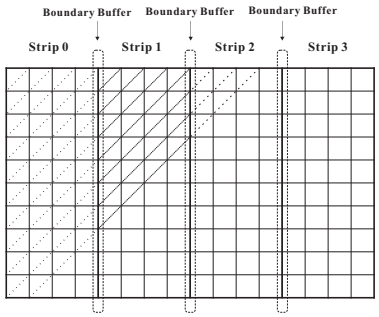

Figure 2. Striped Smith-Waterman algorithm

Our striped algorithm therefore requires $O(\min\{m,s\})$ shared memory per SM and $O(mn/s)$ global memory. The I/O traffic between global memory and the SMs is $O(mn/s)$. To derive the computational time requirements (exclusive of the time taken by the global memory I/O traffic), we assume that the threshold value T is $O(1)$. We note that the computation for the kth strip cannot begin until the top right value of strip $k-1$ has been computed. An SM with c processors takes $T_a = O(s^2/c)$ to compute the top right value of the strip assigned to it and $O(ms/c)$ time to complete the computation for the entire strip. So, SM $p-1$ cannot start working on the first strip assigned to it until time $(p-1)T_a$. When an SM can go form the computation of one strip to the computation of the next strip with no delay, the completion time of SM $p-1$, and hence the time taken by the GPU to do all its assigned work (exclusive of the time taken

by global memory I/O traffic), is $O((p-1)T_a + \frac{ms}{c} * \frac{n}{ps}) = O(\frac{ps^2}{c} + \frac{mn}{pc})$. When an SM takes less time to complete the computation for a strip than it takes to compute the data needed to commence on the next strip assigned to the SM (approximately, $\frac{ms}{c} < pT_a$), an SM must wait $O(pT_a - \frac{ms}{c})$ time between the computation of successive strips assigned to it. So, the time at which SM p finishes is $O((\frac{n}{s} - 1)T_a + \frac{ms}{c}) = O(\frac{(m+n)s}{c})$. We see that while computation time exclusive of global I/O time increases as s increases, global I/O time decreases as s increases. Our experiments of Section VI show that for large m and n, the reduction in global I/O memory traffic that comes from increasing the strip size s more than compensates for the increase in time spent on computational tasks. Although using a larger strip size s reduces overall time, the size of the available shared memory per SM limits the value of s that may be used in practice.

In our GPU implementation of $StripedScore$, the substitution matrix is stored in the shared memory of each SM using $23 \times 23 \times sizeof(int)$ bytes. Additionally, each SM has an output buffer of length 32 for writing values on the boundary of each strip to global memory. This buffer takes $32 \times sizeof(int)$ bytes. We also use six arrays of length $\min\{s, m\} + 2$ each to hold the H values on three adjacent antidiagonals, E values and F values, and new E or F values to be swapped with old values. Another 1200 bytes are reserved by the CUDA compiler to store built-in variables and pass function parameters. The shared memory cache was configured as 48KB shared memory and 16KB L1 cache. So, $min(s, m)$ should be less than 1902. Since we are aligning very large sequences, we assume $s < m$. Hence, $s < 1902$ for our implementation.

The followings are important differences between $BlockedAntidiagonal$ and $StripedScore$:

1) $BlockedAntidiagonal$ requires many kernel invocations from the host while $StripedScore$ requires just one kernel invocation. In other words, the synchronization of $BlockedAntidiagonal$ is done on the host side while in $StripedScore$, the synchronization is done on the device side, which significantly reduces the overhead.
2) In $BlockedAntidiagonal$ the assignment of blocks that are ready for computation to SMs is done by the GPU block scheduler while in $StripedScore$ the assignment of strips to SMs is programmed into the kernel code.
3) The I/O traffic of $StripedScore$ is $O(mn/s)$ while that of $BlockedAntidiagonal$ is $O(mn)$.
4) While for $BlockedAntidiagonal$ near-optimal performance is achieved when $s = 8$, we envision much larger s values for $StripedScore$ which can be up to 1900. Consequently, there is greater opportunity for parallelism within a strip than within a block.

The above steps can lead to significant improvement in the overall performance.

V. Computing the Best Local Alignment

In this section, we describe three GPU algorithms for the case when we wish to determine both the best alignment and its score.

A. $StripedAlignment$

This is a 3-phase algorithm. The first phase is an extension of $StripedScore$ in which each strip stores, in global memory, not only the H and F values needed by the strip to its right but also the coordinates of the local start point of the optimal path to each boundary cell. This local start point is either a position in the current strip or a position on the right boundary of the strip immediately to the left of the current strip. So, for the 4 strips of Figure 3, the boundary cells store the local start points of subpaths that end at the boundary cells $(*, 4)$, $(*, 8)$, $(*, 12)$, and $(*, 16)$, Additionally, we need to store the local start point and the end point for the overall best alignment. Suppose the optimal local alignment path is as the one in Figure 3. This path is made of subpaths that go through strips 0, 1, and 2. The boundary buffer for $(3, 4)$ stores $(2, 3)$ since the best path passing $(3, 4)$ starts at $(2, 3)$, which is a location within the strip. The buffer for $(7, 8)$ stores $(3, 4)$, as the optimal path through $(7, 8)$ enters this strip from $(3, 4)$ in the left adjacent strip. The highest score at $(8, 9)$ and the local start point for the path to $(8, 9)$ is $(7, 8)$. So, $(7, 8)$ is initially stored in registers and finally written along with $(8, 9)$ to global memory. In phase 1, the local start points for the optimal paths to all boundary buffers (not just the boundary buffers through which the overall alignment path traverses) are written to global memory.

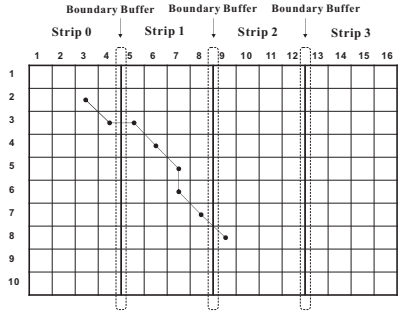

Figure 3. Example for $StripedAlignment$

In phase 2, we serially determine, for each strip, the start and end point of the optimal alignment subpath that goes through this strip. So, for our example of Figure 3, we determine that the optimal alignment path is comprised of a subpath from $(7, 8)$ to $(8, 9)$, another subpath from $(3, 4)$ to $(7, 8)$ and one from $(2, 3)$ to $(3, 4)$.

Finally, in phase 3, the optimal subpath for each strip the optimal path goes through is computed by recomputing the H, E, and F values for the strips the optimal alignment path

traverses. Using the saved boundary H and F values, it is possible to compute the subpaths for all strips in parallel.

B. $ChunkedAlignment$

$ChunkedAlignment$, like $StripedAlignment$, is a 3-phase algorithm. In $ChunkedAlignment$, each strip is partitioned into chunks of height h (Figure 4). For each $h \times s$ chunk we store, in global memory, the H, F, and local start points for positions on right vertical chunk boundaries (i.e., vertical buffers, which are the same as boundary buffers in $StripedAlignment$) and the H and E values for horizontal buffers. The assignment of strips to SMs is the same as in $StripedScore$ (and $StripedAlignment$).

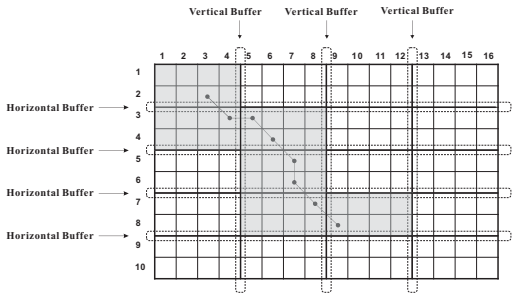

Figure 4. Example for $ChunkedAlignment$

In phase 2, we use the data stored in global memory by the phase 1 computation to determine the start and end points of the subpaths of the optimal alignment path within each strip. Finally, in phase 3, the optimal subpaths are constructed by a computation within each strip through which the optimal alignment traverses. However, the computation with a strip can be limited to essential chunks as shown by the shaded chunks in Figure 4. The computation for these (sub)-strips can be done in parallel.

There are two major differences between $StripedAlignment$ and $ChunkedAlignment$: 1) $ChunkedAlignment$ generates more I/O traffic than does $StripedAlignment$ and also requires more global memory on account of storing horizontal buffer data. Assuming that the height and width of the chunk are nearly equal, The I/O traffic and the global memory requirement are roughly twice the amount for $StripedAlignment$ for the same strip size as the width of the chunk, and 2) Unlike $StripedAlignment$, the computation begins at the start point of a chunk rather than at the first row of the strip. In practice, this should reduce the amount of computation significantly. Our experimental results support the above observations. The major advantage is that by performing more I/O in Phase 1, we are able to reduce the time for Phase 3.

VI. Experimental Results

$lenQuery$	5103	10206	20412	30618	51030
$lenDB$	7168	14336	28672	43008	71680
$s = 64$	67.2	260.2	1031.7	2316.7	6427.1
$s = 256$	23.1	72.6	269.8	597.5	1645.2
$s = 1024$	-	59.4	135.5	261.7	664.4
$s = 1900$	-	-	175.0	279.9	625.7

Figure 5. Running time (ms) of $StripedScore$ for different s values

$StripedScore$: First, we evaluated the running time of $StripedScore$ as a function of strip width s (Figure 5). $lenQuery$ is the length of the query sequence and $lenDB$ is the length of the subject sequence. As predicted by the analysis of Section IV, for sufficiently large sequences, the running time decreases as s increases. However if sequences are relatively small, when s increases, the running time decreases first and then increases.

Next, we compared the relative performance of $StripedScore$ with $s = 1900$, $PerotRecurrence$ (the code of [9] modified to report the best score rather than the best 200 scores), $BlockedAntidiagonal$ [10], $EnhancedBA$ (our enhancement of $BlockedAntidiagonal$ in which only the values on the right and bottom boundaries of each block are stored in global memory thus reducing global memory usage significantly), and CUDASW++2.0 [8]. Figure 6 gives the run time for these algorithms. As can be seen, $PerotRecurrence$ takes 13 to 17 times the time taken by $StripedScore$. The speedup ranges of $StripedScore$ relative to $BlockedAntidiagonal$, $EnhancedBA$, and $CUDASW++2.0$ are, respectively, 20 to 33, 2.8 to 9.3, and 7.7 to 22.8. $BlockedAntidiagonal$ and $CUDASW++2.0$ were unable to solve large instances because of the excessive memory required by them.

$StripedAlignment$: We tested $StripedAlignment$ with different s values and the results are shown in Figure 7. The maximum strip size, which is limited by the amount of shared memory per SM, is 900. The time for phase 2 is negligible and is not reported separately. The time for all three phases generally decrease as s increases. For really large s, the number of strips is comparable to or smaller than the number of streaming processors leading to idle time on some processors. Generally, choosing $s = 256$ or 512 gives the best overall performance for sequences of size 36000 or larger. Choosing a larger s allows for larger sequences to be aligned (as I/O is inversely proportional to s).

We do not compare $StripedAlignment$ with the algorithms of [10], [8], [9] for the following reasons (a) in [10], the traceback is done serially in the host CPU, (b) CUDASW++2.0 [8] does not have a traceback capability, and (c) the traceback of [9] is specifically designed for the benchmark suite SSCA#1 [15] and so only aligns multiple but small subsequences of length less than 128.

$ChunkedAlignment$: There are two parameters in $ChunkedAlignment$ - s representing for the width of one strip, and h representing for the height of one chunk. Effectively, the scoring matrix is divided into blocks of size $h \times s$. Experimental results in Figures 8 (we only present one case due to space limitations) show that increasing

$lenQuery$	1×10^4	2×10^4	3×10^4	5×10^4	1×10^5
$lenDB$	1×10^4	2×10^4	3×10^4	5×10^4	1×10^5
$PerotRecurrence$	815.3	1917.7	3061.7	7014.9	20437.3
$StripedScore$	48.4	113.0	216.3	449.8	1543.1
$BlockedAntidiagonal$	957.2	3719.9	-	-	-
$EnhancedBA$	137.4	527.0	1185.9	3438.6	14327.4
$CUDASW++2.0$	374.5	1530.4	3404.0	10259.1	-

Figure 6. Running time (ms) of scoring algorithms

$lenQuery$	10206			20412		
$lenDB$	14336			28672		
	Phase 1	Phase 3	Total	Phase 1	Phase 3	Total
$s = 128$	180.5	176.5	362.1	710.9	684.2	1408.9
$s = 256$	99.1	97.6	199.9	370.0	366.6	744.4
$s = 512$	71.1	91.2	165.1	229.7	383.8	617.4
$s = 900$	85.1	142.4	230.1	227.6	536.9	767.8

Figure 7. Running time (ms) of $StripedAlignment$ for different s values

s results in better performance and has similar behavior to $StripedAlignment$. Large values of h and s have the potential to reduce the amount of parallelism in Phase 3. A good choice from our experimental results is $s = 512$ and $h = 256$.

As expected, the Phase 3 time for $ChunkedAlignment$ is significantly less than for $StripedAlignment$. Although this reduction comes with an additional computational and I/O cost in Phase 1, the overall time for $ChunkedAlignment$ is much less than for $StripedAlignment$. For sequences of size 20412 and 28672, the best time for $StripedAlignment$ is 617.4ms while that for $ChunkedAlignment$ is 302.4ms $(s = 512, h = 256)$; the ratio is slightly more than 2.

Since $StripedScore$ is an order of magnitude faster than $PerotRecurrence$ and $ChunkedAlignment$ is not an order of magnitude slower than $StripedScore$, we conclude, without experiment, that $ChunkedAlignment$ is faster than the code of [9] modified to find the best alignment.

VII. Conclusion

In this paper, we have developed single-GPU parallelizations of the unmodified Smith-Waterman algorithm for sequence alignment. Our scoring algorithm $StripedScore$ achieves a speedup of 13 to 17 relative to the single-GPU algorithm of [9]. The speedup ranges relative to $BlockedAntidiagonal$ [10] and $CUDASW++2.0$ [8] are, respectively, 20 to 33 and 7.7 to 22.8. Our algorithms achieve a computational rate of 7.1 GCUPS on a single GPU.

Acknowledgment

This work was supported, in part, by the National Science Foundation under grants CNS0829916, CNS0905308 and CCF0903430, and the National Institutes of Health under grant R01-LM010101.

References

[1] S. B. Needleman and C. D. Wunsch, "A general method applicable to the search for similarities in the amino acid sequence of two proteins," *Molecular Biology, 48, 443-453*, 1970.

[2] T. F. Smith and M. S. Waterman, "Identification of common molecular subsequences," *Molecular Biology, 147, 195-197*, 1981.

[3] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman, "Basic local alignment search tool," *Molecular Biology, 215, 403-410*, 1990.

[4] D. Lipman and W. Pearson, "Rapid and sensitive protein similarity searches," *Science, 227, 1435-1441*, 1985.

[5] K. mao Chao, J. Zhang, J. Ostell, and W. Miller, "A local alignment tool for very long DNA sequences," *Comput. Appl. Biosci, 11, 147-153*, 1995.

[6] A. Khalafallah, H. Elbabb, O. Mahmoud, and A. Elshamy, "Optimizing Smith-Waterman algorithm on Graphics Processing Unit," in *ICCTD 2010, 650-654*, 2010.

[7] S. Manavski and G. Valle, "CUDA compatible GPU cards as efficient hardware accelerators for Smith-Waterman sequence alignment," *BMC Bioinformatics, 9, S10*, 2008.

[8] Y. Liu, B. Schmidt, and D. Maskell, "CUDASW++2.0: enhanced Smith-Waterman protein database search on CUDA-enabled GPUs based on SIMT and virtualized SIMD abstractions," *BMC Research Notes, 3, 93*, 2010.

[9] A. Khajeh-Saeed, S. Poole, and J. Blair Perot, "Acceleration of the Smith-Waterman algorithm using single and multiple Graphics Processors," *Computational Physics*, 2010.

[10] T. Siriwardena and D. Ranasinghe, "Accelerating global sequence alignment using CUDA compatible multi-core GPU," in *ICIAfS 2010, 201-206*, 2010.

[11] L. Ligowski and W. Rudnicki, "An efficient implementation of Smith-Waterman algorithm on GPU using CUDA, for massively parallel scanning of sequence databases," *IPDPS 2009, 0, 1-8*, 2009.

[12] A. Melo, M. Walter, R. Melo, M. Santana, and R. Batista, "Local DNA sequence alignment in a cluster of workstations: algorithms and tools," *Journal of the Brazilian Computer Society 2004, 10, 81-88*, 2004.

[13] N. Futamura, S. Aluru, and X. Huang, "Parallel Syntenic Alignments," in *HiPC 2002, 2552, 420-430*, Springer Berlin / Heidelberg, 2002.

[14] O. Gotoh, "An improved algorithm for matching biological sequences," *Molecular Biology, 162, 705-708*, 1982.

[15] D. A. Bader, K. Madduri, J. R. Gilbert, V. Shah, J. Kepner, T. Meuse, and A. Krishnamurthy, "Designing Scalable Synthetic Compact Applications for Benchmarking High Productivity Computing Systems," *CTWatch Quarterly, 2(4B):41-51*, 2006.

$s \rightarrow$	128			512			900		
$h \downarrow$	Phase 1	Phase 3	Total	Phase 1	Phase 3	Total	Phase 1	Phase 3	Total
128	925.3	14.5	945.0	289.9	15.0	308.5	322.3	29.0	354.7
256	881.2	17.8	903.8	282.0	17.2	302.4	315.3	32.9	351.3
512	859.9	24.0	888.8	277.6	22.2	303.0	305.6	41.6	350.2
900	850.2	30.9	886.0	275.7	24.4	303.1	287.8	49.2	340.0

Figure 8. Running Time (ms) of $ChunkedAlignment$ ($lenQuery = 20412$ $lenDB = 28672$)