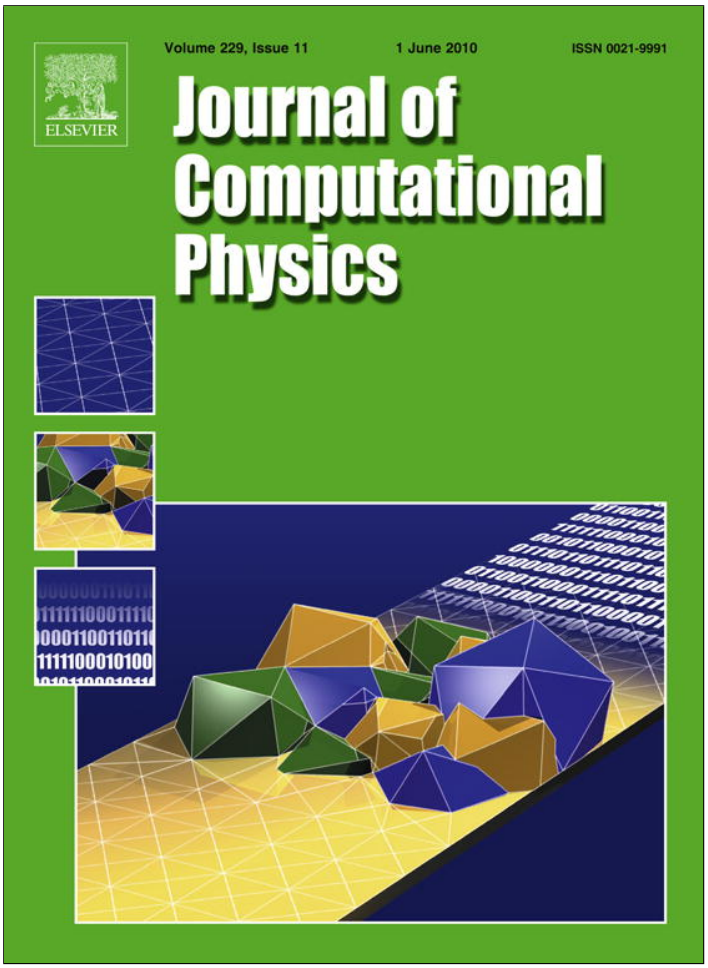


This article appeared in a journal published by Elsevier. The attached copy is furnished to the author for internal non-commercial research and education use, including for instruction at the authors institution and sharing with colleagues.

Other uses, including reproduction and distribution, or selling or licensing copies, or posting to personal, institutional or third party websites are prohibited.

In most cases authors are permitted to post their version of the article (e.g. in Word or Tex form) to their personal website or institutional repository. Authors requiring further information regarding Elsevier's archiving and manuscript policies are encouraged to visit:

http://www.elsevier.com/copyright

Journal of Computational Physics 229 (2010) 4247–4258

Contents lists available at ScienceDirect

Journal of Computational Physics

journal homepage: www.elsevier.com/locate/jcp

Acceleration of the Smith–Waterman algorithm using single and multiple graphics processors

Ali Khajeh-Saeed [a], Stephen Poole [b], J. Blair Perot [a,*]

[a] University of Massachusetts Amherst, Mechanical and Industrial Engineering, Amherst, MA 01003, United States
[b] Computer Science and Mathematics Division Oak Ridge National Laboratory, Oak Ridge, TN 37831, United States

ARTICLE INFO

Article history:
Received 13 August 2009
Accepted 8 February 2010
Available online 20 February 2010

Keywords:
Smith–Waterman
Sequence matching
Parallel scan
GPU
Graphics processor

ABSTRACT

Finding regions of similarity between two very long data streams is a computationally intensive problem referred to as sequence alignment. Alignment algorithms must allow for imperfect sequence matching with different starting locations and some gaps and errors between the two data sequences. Perhaps the most well known application of sequence matching is the testing of DNA or protein sequences against genome databases. The Smith–Waterman algorithm is a method for precisely characterizing how well two sequences can be aligned and for determining the optimal alignment of those two sequences. Like many applications in computational science, the Smith–Waterman algorithm is constrained by the memory access speed and can be accelerated significantly by using graphics processors (GPUs) as the compute engine. In this work we show that effective use of the GPU requires a novel reformulation of the Smith–Waterman algorithm. The performance of this new version of the algorithm is demonstrated using the SSCA#1 (Bioinformatics) benchmark running on one GPU and on up to four GPUs executing in parallel. The results indicate that for large problems a single GPU is up to 45 times faster than a CPU for this application, and the parallel implementation shows linear speed up on up to 4 GPUs.

© 2010 Elsevier Inc. All rights reserved.

1. Introduction

Scientific algorithms tend to operate on long streams of data. This makes modern cached based processors ill-suited for scientific computations. The inability to access large data sets efficiently in these hardware designs means that almost all scientific algorithms are now memory bound and operate at about 5% of the processor rated speed. Parallel processing can compensate for this inefficiency but does not remove it. As a result scientific computing has experienced little in the way of hardware performance gains during the last decade. Fortunately, other inexpensive and commodity-priced electronic devices are now available, such as graphics processors (GPUs) and game processors (IBM Cell), that are better suited to large vector data access. Graphics processors (which are now standard on most PCs) became readily programmable in 2005 and easily programmable via a C-like interface (CUDA 2007/OpenCL 2008) in early 2008. When used as a math accelerator for scientific algorithms GPUs result in roughly an order of magnitude better performance than what can be obtained with a multi-core CPU.

The tradeoff involved in using a graphics processor rather than a general purpose CPU is that a GPU is designed for a specific purpose, and not general purpose computing. While the GPU hardware is well suited for scientific computing, it is still necessary to understand the hardware and its limitations if the sought for performance is to be achieved. In short, algorithms

* Corresponding author. Tel.: +1 413 545 3925; fax: +1 413 545 1027.
E-mail address: perot@ecs.umass.edu (J. Blair Perot).

0021-9991/$ - see front matter © 2010 Elsevier Inc. All rights reserved.
doi:10.1016/j.jcp.2010.02.009

need to structured for the hardware in order to run well on the GPU. Interestingly, the same was true of the original Cray vector supercomputers that initiated the scientific computing era. In fact, the hardware design and algorithm requirements of the GPU are very similar to those of the early vector supercomputers which also had special purpose dedicated scientific computing processors. In some sense scientific computing algorithms have come full circle in two decades with only the price of the hardware changing (10,000× less) due to the economies of commodity production.

The algorithmic changes necessary to efficiently use a GPU are best illustrated by example. In this work we will focus on the problem of sequence matching or sequence alignment. The Smith–Waterman algorithm [1] is a dynamic programming solution to finding the optimal alignment of two sequences. Dynamic programming in this context refers to an optimization strategy, not a programming style. While the algorithm has optimality guarantees, it is also somewhat computationally intensive. For two sequences of length L_1 and L_2 the Smith–Waterman algorithm requires $O(L_1L_2)$ computational work. In biological applications the length of the test sequence, L_1, is often on the order of 10^4 to 10^5 and the size of a genomic database, L_2, can range from 10^7 to 10^{10}. In order to narrow the search space and speed up the search process heuristics are often applied to the classic Smith–Waterman algorithm. BLAST [2] and FASTA [3,4] are very popular implementations which implement heuristics that are appropriate for DNA and protein sequencing. Heuristic accelerators loose the optimality guarantee but they do result in a roughly 60x speedup in the algorithm. Both BLAST and FASTA have well tested parallel implementations and are installed on many production supercomputers.

This work focuses on the hardware acceleration of the classic Smith–Waterman algorithm (without heuristics). We do this primarily because the heuristics are somewhat specific to the application domain. We have noted that the application domain for sequence matching is growing (beyond bio-informatics) and therefore it makes sense to keep our ideas as general as possible. The techniques developed in the work can still be applied to hardware accelerate BLAST and FASTA or any other Smith–Waterman derivative. In addition, a benchmark (SSCA #1) involving the classic Smith–Waterman algorithm exists.

There are already a number of efficient implementations of the Smith–Waterman algorithm on different CPU architectures [5–10] and GPU architectures [11–16]. The early work of Lia et al. [14] showed a 2–4× speedup over the CPU using a single Nvidia 7800 GTX and demonstrated a maximum of a 241 million cell updates per second (MCUPS). The work of Manavaski and Valle [15] describes an implementation whereby parallelism was achieved by solving a large number of small but separate Smith–Waterman problems. This variation is then trivially parallel and using an Nvidia 8800 GTX, they saw roughly 1850 MCUPS for a single GPU. The work of Akoglu and Striemer [16] describes an implementation of Smith–Waterman algorithm that is somewhat more efficient. They showed a 7–10× speedup over the CPU using a single Nvidia C870 GPU. Both Manavaski and Valle [15] and Akoglu and Striemer [16] employ algorithms that can only compare relatively small test sequences with the database. Our implementation differs from these prior works in that it is designed to solve a single but very large Smith–Waterman problem. Our implementation also uses multiple GPUs to solve this single problem. In our implantation the length of test sequence is not important and the method can handle very large test sequences and very large databases.

The classic Smith–Waterman algorithm is reviewed in Section 2 of this paper. Both the serial and existing parallel solution approaches are discussed. The problems with implementing the existing parallel approaches on GPU hardware are presented in Section 3 along with a new parallel solution approach that maps well to the GPU hardware. In Section 4 it is then shown how this approach can be fairly easily extended to work on multiple GPUs that communicate via MPI running on 100 Mb Ethernet. Section 5 describes the scalable synthetic compact application #1 (SSCA #1) benchmark and presents timing results for a variety of problem sizes and numbers of GPUs (from 1 to 4). Finally, Section 6 discusses the implication of this new algorithm and draws some conclusions about both the power and limitations of GPU acceleration of scientific algorithms of this type.

2. The Smith–Waterman algorithm

Given two possible sequences, such as those shown below in Fig. 1, sequence alignment strives to find the best matching subsequences from the pair. The best match is defined by the formula

$$\sum_{i=1}^{L}[S(A(i),B(i))] - W(G_s,G_e) \tag{1}$$

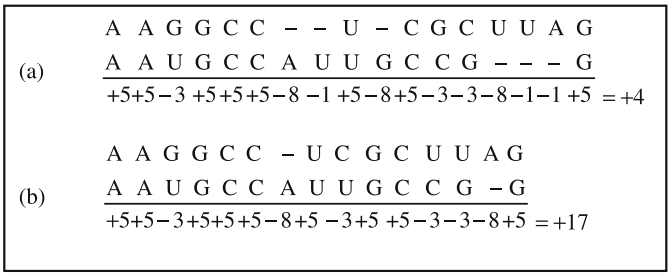


Fig. 1. Possible similarity values for two small sequences: The second alignment possibility is the better one (for the SSCA #1 scoring parameters).

where W is the gap function involving the gap start-penalty G_s and gap extension-penalty G_e, and the similarity score S, is given by the user or particular application. In realistic biological applications the gap penalties and similarity scores can be quite complex. However, the W function itself is often quite simple [17]. In this work we use a very simple scoring system in which matching items get a score of 5 and dissimilar items have a score of -3, the gap start-penalty is 8 and gap extension-penalty is 1. This simple scoring system is dictated by SSCA #1, but our code is completely general and makes no use of this particular information. Using the SSCA #1 scoring system, the possible alignment in Fig. 1(a) has a score of 4 whereas the alignment in Fig. 1(b) (with fewer gaps) has a much higher score of 17.

2.1. Basic algorithm

The Smith–Waterman algorithm finds the optimal alignment by constructing a solution table such as that shown in Fig. 2. The first data sequence (the database) is usually placed along the top row, and the second sequence (the test sequence) is usually placed in the first column. The table values are called $H_{i,j}$ where i is the row index and j is the column index. The algorithm is initiated by placing zeros in the first row and column of the table. The other entries in the table are then set via the equation,

$$H_{i,j} = Max\begin{cases} Max(H_{i-1,j-1} + S_{i,j}, 0) \\ Max_{0<k<i}(H_{i-k,j} - (G_s + kG_e)) \\ Max_{0<k<j}(H_{i,j-k} - (G_s + kG_e)) \end{cases} \tag{2}$$

That is, an entry in the table is the maximum of: the entry diagonally to the left and upwards plus the similarity score for the item in that row and column, the maximum of all the entries above the entry and in that same column minus a gap function, and the maximum of all the entries to the left of the entry and in that same row minus a gap function, and zero. For example, the highlighted (bottom right) entry in Fig. 2 is 4 because that is the maximum of $5 - 3 = 2$ (the upper-left diagonal contribution), -5 (the column contribution, max of $0 - 8 - 1 = -9$, $5 - 8 - 2 = -5$, $0 - 8 - 3 = -11$,...) and 4 (the row contribution, max of $5 - 8 - 1 = -4$, $6 - 8 - 2 = -4$, $15 - 8 - 3 = 4$, $5 - 8 - 4 = -7$, ...).

Note that it is not possible for the table values to be less than zero. In addition, the dependencies of the table values means that the values cannot all be computed in parallel. A value in the table requires all the values to the left and above the value to be computed. Fig. 2, shows the direct dependency of the highlighted table value.

Generating the table values is the computationally intensive part of the Smith–Waterman algorithm. Once these values are obtained the optimal alignment is determined by performing a 'traceback'. Fig. 3 shows the traceback that would occur from the problem shown in Fig. 2. The traceback begins by finding the maximum value in the table. This is the end of the optimal alignment sequence. The algorithm then works backwards (upwards and to the left) from this ending value and determines where the current value came from (which maximum was chosen). Moving backwards is not always easy to determine and is sometimes performed by saving a pointer value $P_{i,j}$ that is set as the table is created. Moving upwards inserts a gap in the first (database) sequence and moving to the left inserts a gap in the second (test) sequence. When a table value of zero is reached the traceback process terminates, and the optimally aligned subsequence has been found. It is possible for the traceback to be non-unique, in this case it is typical to prefer a sequence with no gaps (move diagonally) if they are similar all things being equal or if that is not possible a sequence with gaps in the data base sequence (move up). This later choice is biologically motivated and actually completely arbitrary.

	0	**C**	**A**	**G**	**C**	**C**	**U**	**C**	**G**	**C**	**U**	**U**	**A**	**G**
0	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*
A	*0*	0	5	0	0	0	0	0	0					
A	*0*	0	5	2	0	0	0	0	0					
U	*0*	0	0	2	0	0	5	0	0					
G	*0*	0	0	5	0	0	0	2	5					
C	*0*	5	0	0	10	5	0	5	0					
C	*0*	5	2	0	5	15	6	5	**4**					
A	*0*													
U	*0*													
U	*0*													
G	*0*													
C	*0*													
C	*0*													
G	*0*													
G	*0*													

Fig. 2. Dependency of the values in the Smith–Waterman table.

	0	C	A	G	C	C	U	C	G	C	U	U	A	G
0	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*
A	*0*	0	5	0	0	0	0	0	0	0	0	0	5	0
A	*0*	0	5	2	0	0	0	0	0	0	0	0	5	2
U	*0*	0	0	2	0	0	5	0	0	0	5	5	0	2
G	*0*	0	0	***5***	0	0	0	2	5	0	0	2	2	5
C	*0*	5	0	0	***10***	5	0	5	0	10	1	0	0	0
C	*0*	5	2	0	5	***15***	6	5	4	5	7	1	0	0
A	*0*	0	10	1	0	***6***	12	3	2	1	2	4	6	0
U	*0*	0	1	7	0	5	***11***	9	1	0	6	7	1	3
U	*0*	0	0	0	4	4	10	***8***	6	0	5	11	4	1
G	*0*	0	0	5	0	3	1	7	***13***	4	3	2	8	9
C	*0*	5	0	0	10	5	0	6	4	***18***	9	8	7	6
C	*0*	5	2	0	5	15	6	5	4	9	15	6	5	4
G	*0*	0	2	7	0	6	12	3	10	8	6	12	3	10
G	*0*	0	0	7	4	5	3	9	8	7	5	3	9	8

Fig. 3. Similarity matrix and best matching for two small sequences CAGCCUCGCUUAG (top) and AAUGCCAUUGCCGG (left). The best alignment is: □ GCC-UCGC and GCCAUUGC which adds one gap to the database sequences and which has one dissimilarity (3rd to last unit).

	0	C	A	G	C	C	U	C	G	C	U	U	A	G
0	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*
A	*0*	0	5	0	0	0	0	0	0	0	?			
A	*0*	0	5	2	0	0	0	0	0	?				
U	*0*	0	0	2	0	0	5	0	?					
G	*0*	0	0	5	0	0	0	?						
C	*0*	5	0	0	10	5	?							
C	*0*	5	2	0	5									
A	*0*	0	10	1	?									
U	*0*	0	1	?										
U	*0*	0	?											
G	*0*	?												
C	*0*													
C	*0*													
G	*0*													
G	*0*													

Fig. 4. Anti-diagonal method and dependency of the cells.

2.2. Optimizations

If Eq. (2) is naïvely implemented, then the column maximum and row maximum (second and third items in Eq. (2)) are repetitively calculated for each table entry causing $O(L_1L_2(L_1 + L_2))$ computational work. This is alleviated by retaining the previous row and column sums [18], called $F_{i,j}$ and $E_{i,j}$, respectively. This increases the storage required by the algorithm by a factor of 3 but reduces the work significantly. The Smith–Waterman algorithm is now given by,

$$\begin{aligned} E_{i,j} &= Max(E_{i,j-1}, H_{i,j-1} - G_s) - G_e \\ F_{i,j} &= Max(F_{i-1,j}, H_{i-1,j} - G_s) - G_e \\ H_{i,j} &= Max(H_{i-1,j-1} + S_{i,j}, E_{i,j}, F_{i,j}, 0) \end{aligned} \tag{3}$$

The classic way to parallelize this algorithm is to work along anti-diagonals [19]. Fig. 4 shows an anti-diagonal, and it should be clear from the dependency region shown in Fig. 2 that each diagonal item can be calculated independently of the others. Block diagonal algorithms are also possible as long as each block is processed serially.

3. Parallel scan Smith–Waterman algorithm

The current generation of GPUs (NVIDIA G200 series) requires a minimum of roughly 30 k active threads for 100% occupancy in order to perform efficiently. This means the diagonal algorithm will only perform efficiently on a single GPU if the minimum of the two sequence lengths is at least 30 k. In biological applications test sequence lengths of this size are rare,

		0	C	A	G	C	C	U	C	G	C	U	U	A	G
H	C	0	5	0	0	10	5	0	5	0	10	1	0	0	0
$\widetilde{H}$		0	5	2	0	5	15	2	5	2	5	7	0	0	0
$\widetilde{E}$	C	0	-1	4	3	2	4	14	13	12	11	10	9	8	7
H		0	5	2	0	5	15	6	5	4	5	7	1	0	0

Fig. 5. Graphical representation of row-access for each step in row-parallel scan Smith–Waterman algorithm. This example is calculating the 6th row from the 5th row (of the example problem shown in Fig. 3). The column maximum, F, is not shown but is also stored from the previous row along with H.

but this is not the real problem with the diagonal algorithm. The primary issue with the diagonal approach is how it accesses memory.

Any hardware which can accelerate scientific computations is necessarily effective because it accelerates memory accesses. On the GPU, memory access is enhanced in hardware by using memory banks and memory streaming. On the GPU this means that it is very efficient to access up to 32 consecutive memory locations, and relatively ($5\times$ slower) inefficient to access random memory locations such as those dispersed along the anti-diagonal (and its neighbors).

To circumvent this problem it is shown below how the Smith–Waterman algorithm can be reformulated so that the calculations can be performed in parallel one row (or column) at a time. Row (or column) calculations allow the GPU memory accesses to be consecutive and therefore fast. To create a parallel row-access Smith–Waterman algorithm, the typical algorithm (Eq. (3)) is decomposed into three parts. The first part neglects the influence of the row maximum, $E_{i,j}$ and calculates a temporary variable $\widetilde{H}_{i,j}$.

$$\begin{aligned} F_{i,j} &= Max(F_{i-1,j}, H_{i-1,j} - G_s) - G_e \\ \widetilde{H}_{i,j} &= Max(H_{i-1,j-1} + S_{i,j}, F_{i,j}, 0) \end{aligned} \tag{4a}$$

This calculation depends only on data in the row above, and each item in the current row can therefore be calculated independently and in parallel. It is then necessary to calculate the row maximums.

$$\widetilde{E}_{i,j} = Max_{1<k<j}(\widetilde{H}_{i,j-k} - kG_e) \tag{4b}$$

These row maximums are performed on $\widetilde{H}_{i,j}$ not $H_{i,j}$, since the later is not yet available anywhere on the row. The key to the reformulated algorithm is noting that these row maximums look dependent and inherently serial, but in fact they can be computed rapidly in parallel using a variation of a parallel maximum scan. Finally we compute the true H_{ij} values via the simple formula

$$H_{i,j} = Max(\widetilde{H}_{i,j},\ \widetilde{E}_{i,j} - G_s) \tag{4c}$$

Appendix A shows that this decomposition is mathematically equivalent to Eq. (3). The storage requirements are the same. Each step of this algorithm is completely parallel, and even more importantly for the GPU, has a contiguous memory access pattern. Fig. 5 shows a graphical representation of the row-access parallel Smith–Waterman algorithm.

4. Multiple GPUs

It is possible to trivially use the previous algorithm on up to four GPUs which are installed on a single motherboard using SLI (Nvidia) [20] or ATI CrossFire [21] technology. This involves an additional physical connection between the GPUs. However, for large collections of GPUs operating on different motherboards and communicating via the network, a more complex parallelization and implementation strategy is necessary.

The relatively slow speed of network connections relative to memory speeds means that parallelizing multiple GPUs (more than 4) requires as coarse grained a decomposition of the problem as possible. This is in direct contrast to the fine grained row-parallelism which was utilized in the previous section to efficiently program a single GPU. To use N GPUs, the Smith–Waterman table is decomposed into N large, and slightly overlapping blocks. The idea is shown in Fig. 6.

The overlap size is chosen to be larger than the expected size of the alignment subsequence. This size is not known prior to the calculation but it can often be estimated. In addition, this criterion can be confirmed at the end of the calculation and the calculation restarted with a larger overlap if the test fails. Two separate solutions are present in the overlap region. The

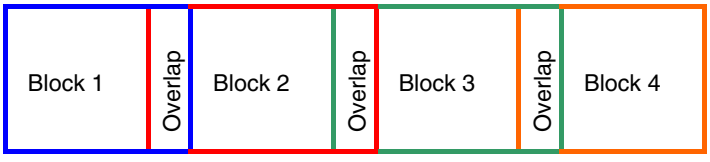


Fig. 6. Schematic diagram of overlapping for using multiple GPUs.

solution in the overlap region associated with the left computation block is the correct value (because the Smith–Waterman algorithm propagates from left to right). The solution in the overlap region associated with the right hand computational block starts with zeros in the left most columns and is incorrect. But it can be seen that after the overlap region the table results will be the same as the standard non-blocked algorithm (as long as the overlap is longer than the longest alignment sequence).

This version of the coarse grained parallel implementation was chosen because the SSCA #1 benchmark tends to produce relatively small alignments (much less than 64 units long). In addition, biological application tends to have one relatively short test sequence and an extremely long database sequence. The alignment sequence cannot be larger than an order one constant times the shortest sequence length (the constant depends on the gap and similarity variables). So in most cases the overlap region is very small relative to the total work that needs to be performed. For 4 GPUs with a 64 long overlap region operating on a 1 M long database, the overlap is less than .02% of the total work. The advantage of this overlapped block approach is that each block is completely independent and no communication between the GPUs is necessary to solve the problem, until the very end. Once every GPU finds the optimal sequence in its block of the problem (ignoring any results from the initial overlap region, but including results from the final overlap region) those potential sequences must be reconciled on a single CPU. Our implementation uses MPI over 100 Mb Ethernet for the final reconciliation operation.

It is not too difficult to implement a blocked version of the Smith–Waterman algorithm that does not use any overlap regions. This algorithm has no restrictions on the size of the overlap region, but does involve more communication between the GPUs. When no overlap is present step 4b requires boundary data to be sent from the neighboring GPU. The time for this data transfer can be entirely hidden by sending the data as soon as it is calculated in the previous step 4c. In contrast, step 4b requires the parallel scan on each GPU to be completed between GPUs. This is only one data item from each GPU, but the data transfer cannot be hidden by any other useful work. The parallel scan completion (between GPUs) must take place once for every row in the Smith–Waterman table. The final step of the row-parallel algorithm is trivially parallel and does not require any communication. This more robust, but also more communication intensive approach is not used in SSCA #1 calculations because it is not necessary.

5. Results

Timings were performed on 3 different GPU architectures and a high-end CPU. The Nvidia 9800 GT has 112 cores and its memory bandwidth is 57.6 GB/s. The Nvidia 9800 GTX has 128 cores and its memory bandwidth is 70.4 GB/s. The newer generation Nvidia 295 GTX comes as two GPU cards that share a PCIe slot. When we refer to a single 295 GTX we refer to one of these two cards, which has 240 cores and a memory bandwidth of 111.9 GB/s. The CPU is an AMD quad-core Phenom II X4, operating at 3.2 GHz, it has 4×512 KB of L2 cache and 6 MB of L3 cache. For the parallel timings a PC that contains 2 Nvidia 295 GTX cards containing a total of 4 GPUS was used. All code was written in C++ with Nvidia's CUDA language extensions for the GPU, and compiled using Microsoft Visual Studio 2005 (VS 8) under Windows XP Professional x64. The bulk of Nvidia SDK examples use this configuration.

The SSCA#1 benchmark was developed as a part of the high productivity computer systems (HPCS) [22] benchmark suite that was designed to evaluate the true performance capabilities of supercomputers. It consists of 5 kernels that are various permutations of the Smith–Waterman algorithm.

In the first kernel the Smith–Waterman table is computed and the largest table values (200 of them) and their locations in the table are saved. These are the end points of well aligned sequences, but the sequences themselves are not constructed or saved. The sequence construction is performed in kernel 2. Because the traceback step is not performed in kernel 1 only a minimal amount of data in the table needs to be stored at any time. For example, in the row-parallel version, only the data from the previous row needs to be retained. This means that kernel 1 is both memory efficient and has a high degree of fine

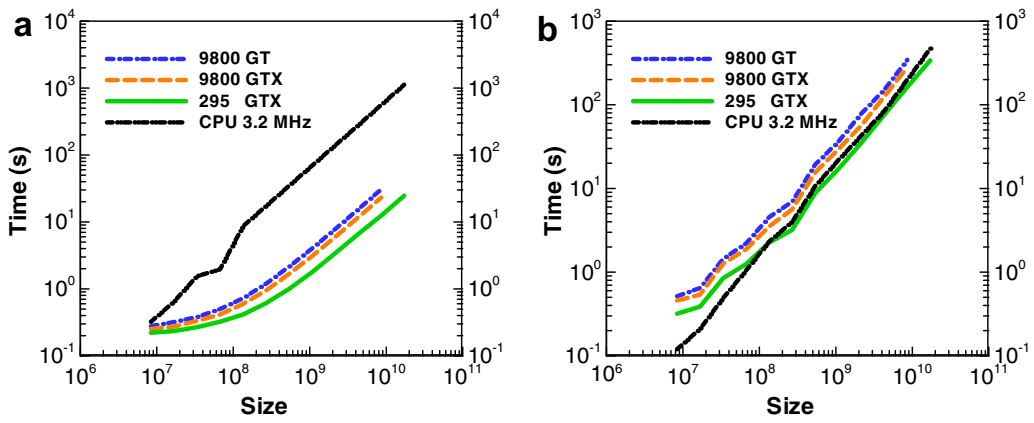


Fig. 7. Time for (a) kernel 1, and (b) kernel 3, for different problem sizes and different processors.

scale parallelism. Single processor timings for kernel 1 are shown in Fig. 7(a) for different table sizes and different processors. The table size is the length of database multiplied by the length of the test sequence (which was always 1024 for the kernel 1 tests). For smaller size problems, the overhead associated with saving and keeping 200 best matches becomes a significant burden for the GPUs and their efficiency decreases.

Timings for kernel 3 of the SSCA #1 benchmark are shown in Fig. 7(b). Kernel 3 is similar to kernel 1 in that it is another Smith–Waterman calculation. However in this case the sequences are required to be computed at the same time (in a single pass) as the table evaluation. The table is, of course, too large to store in its entirety as it is calculated, so only 'enough' of the table to perform a traceback is kept. Kernel 3 therefore requires the maximum possible subsequence length to be known beforehand. In SSCA #1 this is possible because the test sequences in kernel 3 are the 100 most interesting subsequences (100 highest scoring sequences) found in kernel 2. In kernel 3, the 100 best matches for each of the 100 test sequences (10,000 matches) must be computed. By assigning groups of threads into a single-instruction, multiple-thread (SIMT) block, a single block is assigned to find 100 best matches for one test sequence. The results of kernel 3 are 100×100 subsequences. Kernel 3 therefore has coarse grained parallelism that might be used to make it faster than kernel 1 on general purpose hardware. However, the kernel 3 algorithm requires different threads and different thread blocks to be doing different things at the same time, which is ill-suited to the GPU (essentially SIMT) architecture. The performance of the GPUs for this kernel is therefore roughly the same as one core of a (large cache) CPU.

The speedup of the GPUs over the CPU is shown Fig. 8 for kernel 1 and kernel 3. For a finely parallel (mostly SIMT) algorithm (kernel 1) the latest GPU (295 GTX) is over 45 times faster than a single core of the latest CPU. This performance benefit might be reduced to roughly $10\times$ faster if all four cores of the (Phenom II) CPU were put to use. But the factor of 45x can also be regained (see below) by putting 4 GPUs on the motherboard. Many motherboards now have two (or even four) PCIe 16x slots.

Fig. 8 shows that the performance of the GPU tends to be strongly problem size dependent. Like a vector supercomputer from the 1990s the startup costs are high for processing a vector. On a GPU the highest efficiencies are found when computing 10^5 or more elements at a time. Because the test sequence is 1024 a problem size of 10^9 translates into an entire row of (10^6) database elements being processed during each pass of the algorithm.

This figure also shows that the relative performance of the GPU cards on computally intensive problems is proportional to their hardware capabilities (number of cores and memory bandwidth) irrespective of whether the algorithm works efficiently on the GPU. It is probably the memory bandwidth that dictates the performance for the memory bound Smith–Waterman problem, but this is difficult to prove since the GPU architectures tend to scale up the number of cores and bandwidth simultaneously. Of course, the constant of proportionality is a strong function of the algorithm structure and the problem size.

The other three kernels in the SSCA #1 benchmark are very fast compared to kernels 1 and 3. In the second kernel, subsequences are constructed from the (kernel 1) 200 endpoint locations. Since the table has not been stored during the first kernel this means small parts of the table need to be recomputed. This reconstruction happens by applying the Smith–Waterman 'backwards', since the algorithm works just as well either direction. Starting at the end point location the corresponding row and column are zeroed. The algorithm is now applied moving upwards and to the left (rather than downwards and to the right). When the endpoint value is obtained in the table, this is the starting point of the sequence. The traceback process then occurs by tracing to the right and down. In our GPU implementation groups of 64 threads work on each of these small sub-tables, and the sub-tables are all computed in parallel.

As with the kernel 3 implementation, the traceback procedure is an inherently serial process and difficult to implement well on the GPU. A GPU operates by assigning threads into a SIMT group called a block. In kernel 2 a single block is assigned to compute the small sub-table for each endpoint. 64 threads are assigned to each block (32 is the minimum possible on the GPU), so the sub-table is constructed in 64×64 chunks. In SSCA#1, a single 64×64 block is almost always sufficient to

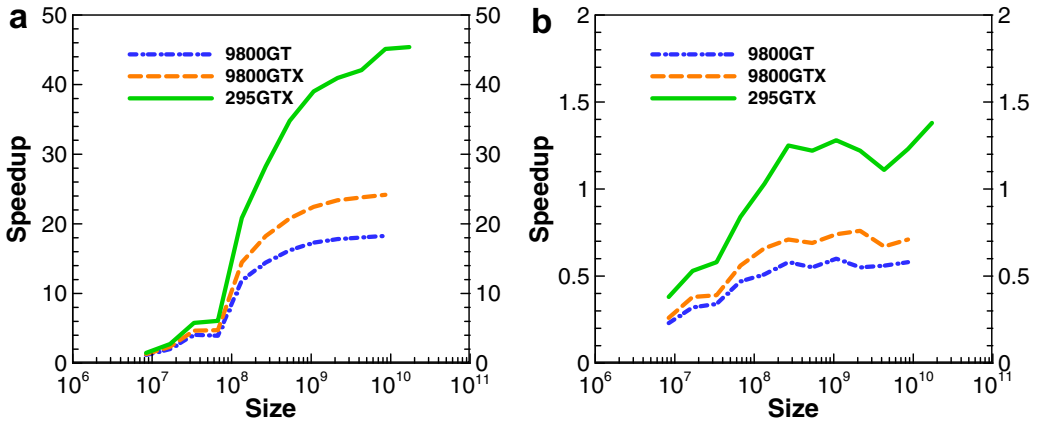


Fig. 8. Speedup for (a) kernel 1, and (b) kernel 3, for different problem sizes and different GPUs.

capture the alignment sequence, but it is almost always wasteful as well. The 64 threads assigned to a sub-table use the anti-diagonal method to construct each 64 × 64 chunk. Timing for this kernel is shown in Fig. 9(a) for different problem sizes and GPUs. The performance is similar to kernel 3 although the times themselves are much smaller. As expected the performance is not a function of the problem size. Kernel 2 always works on 200 sequences. It does not matter how large the table is that they originally came from.

The fourth kernel goes through each set of matches (100 × 100 best matches) found by the third kernel, and performs a global pairwise alignment for each pair. The output of this kernel is 100 × 100 × (100 − 1)/2 global alignment scores. The global alignment of two (same length) subsequences is given by Eq. (5) below.

$$H_{i,j} = Min \begin{cases} H_{i-1,j-1} + 0 & A_j = B_i \\ H_{i-1,j-1} + 1 & A_j \neq B_i \\ H_{i-1,j} + 2 \\ H_{i,j-1} + 2 \end{cases} \tag{5}$$

The global similarity, dissimilarity and gap penalty (given in Eq. (5)) are dictated by SSCA #1 but could be arbitrary. Global alignment is also performed by creating a table. The global pairwise alignment score of two subsequences is the number in the right bottom corner of table. The table is started with $H_{i,0} = i \times 2, H_{0,j} = j \times 2$. On the GPU, each thread performs a single global pairwise alignment for each pair. Every table construction is completely independent and so this kernel is highly parallel and very fast. Fig. 9(b) shows the timing for this kernel.

The fifth kernel performs multiple sequence alignment on each of the sets of alignments generated by the third kernel, using the simple, approximate center star algorithm [18]. The center sequence is chosen based on maximum score obtained in the fourth kernel. Each block performs single multiple sequence alignment per sets of alignments generated by the third kernel. Timing for this kernel is shown in Fig. 9(c) for different problem sizes. This kernel does not map well to the GPU but is very fast nonetheless. It involves 10,000 independent tasks. Kernel 4 and 5 are scale on the log of the problem size because the sequence lengths get slightly longer as the database gets larger (due to the probability of extreme events increasing).

The parallel performance (for kernels 1 and 3) of using up to four GPUs is shown in Fig. 10. These four GPUs happen to be located on a single motherboard, but the code uses MPI and four separate processes to execute on 4 GPUs so it can also be

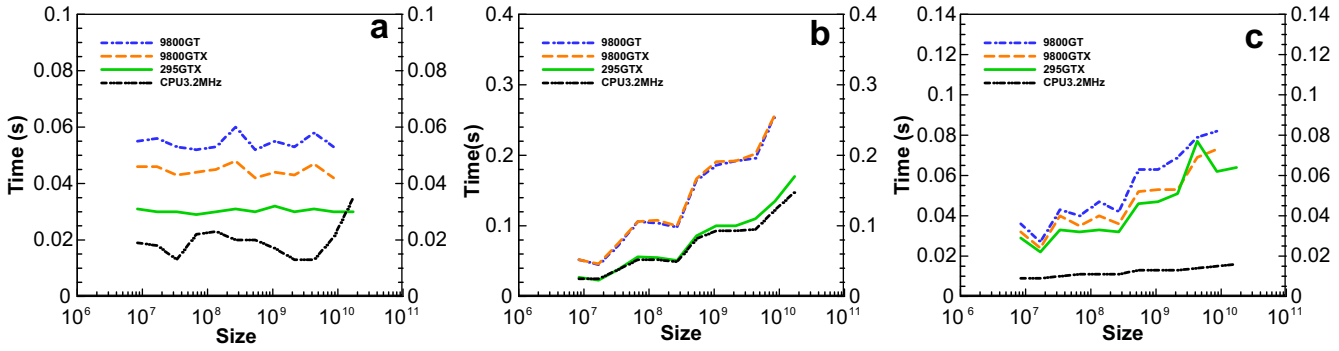


Fig. 9. Time for (a) kernel 2, (b) kernel 4 and, (c) kernel 5 for different problem sizes and different processors.

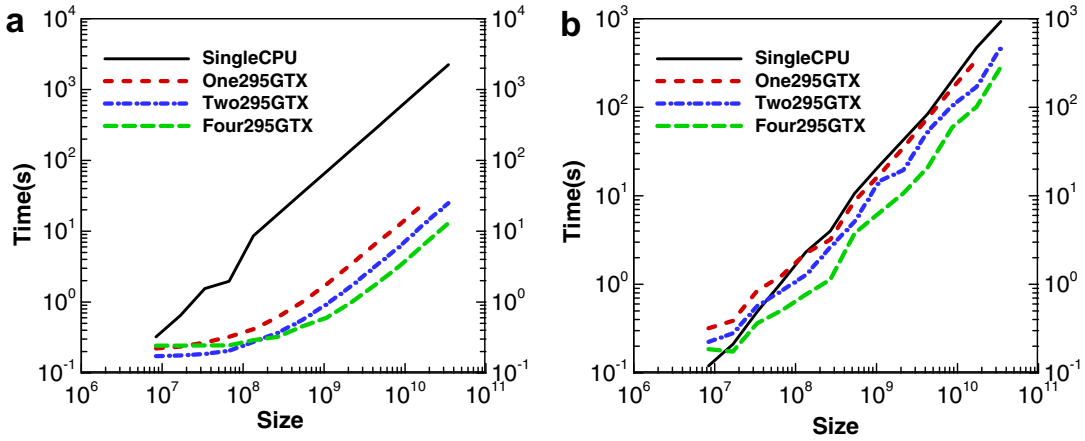


Fig. 10. Time for (a) kernel 1 and, (b) kernel 3, for different problem size and different numbers of GPUs.

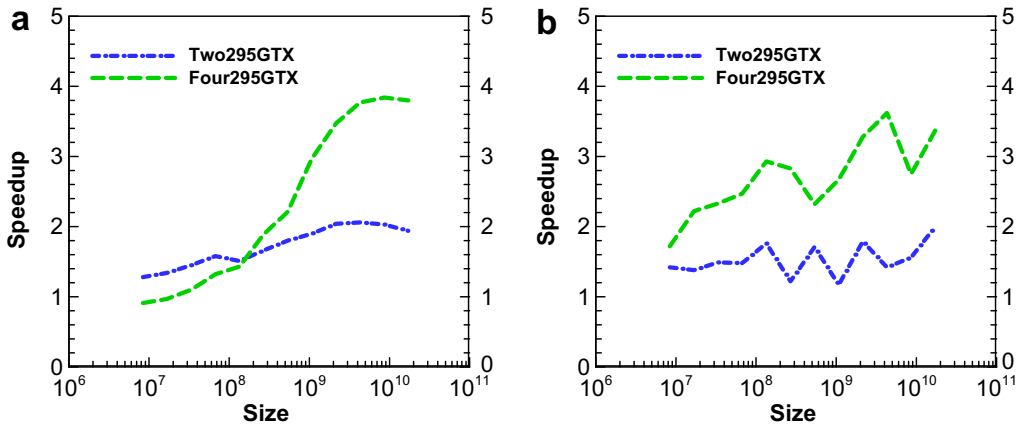


Fig. 11. Speedup versus a single 295 GTX . (a) kernel 1 and, (b) kernel 3 , for different problem size and different numbers of GPUs.

applied to four GPUs on four different motherboards which have a network connection. Kernel 1 and Kernel 3 (the core Smith–Waterman algorithm) take by far the most time (and memory), so these are by far the most interesting kernels to parallelize on multiple GPUs. In addition to being fast already, kernels 2, 4 and 5 have an inherent coarse grained parallelism to them that should make them very easy to parallelize if one desired to. Kernel 2 has 200 completely independent tasks, kernel 4 has close to half a million independent tasks, and kernel 5 has ten thousand.

The speed up of multiple GPUs versus a single GPU (for kernels 1 and 3) is shown in Fig. 11. For kernel 1, the speed up is poor for small problem sizes because of the cost associated with maintaining the list of the current best 200 matches is larger than the Smith–Waterman calculation itself.

6. Conclusion

Performance increases of an order of magnitude over a conventional high-end CPU are possible on the Smith–Waterman algorithm when graphics processors are used as the compute engine. Like most scientific algorithms, the Smith–Waterman algorithm is a memory bound computation when the problem sizes become large. The increased performance of the GPUs in this context is due almost entirely due to the GPU's different memory subsystem. The GPU uses memory banks rather than caches. This fundamental difference in the memory architecture means that these results are not unique to this problem, and superior GPU performance (of roughly an order of magnitude) is to be expected from a great number of scientific computing problems (most of which are memory bound like the Smith–Waterman algorithm).

The particularly high-performance of kernel 1 of the SSCA #1 benchmark was obtained because it was possible to reformulate the Smith–Waterman algorithm to use sequential memory accesses. Since each sequential memory access can be guaranteed to come from a different memory bank they occur simultaneously. The parallel scan was a critical component of this reformulation. The parallel scan operation may currently be under-utilized and might be effectively used in many other scientific algorithms as well. In order to accomplish sequential memory accesses a novel row (or column) parallel version of the Smith–Waterman algorithm was formulated.

Kernel 3 of the SSCA #1 benchmark accomplishes the same task as kernel 1. However, the implementation is restricted and essentially forces a MIMD solution to the problem, with different cores working on different parts of the problem. The GPU does not require a SIMT solution approach, but it does tend to work better with that style algorithm. This is because on the GPU groups of 16 threads do work in a SIMT fashion. MIMD style code can therefore be up to 16 times slower. Not too surprisingly our kernel 3 implementation is roughly 14 times slower than kernel 1.

Similarly, the speedup for the already very fast kernels 2, 4 and 5 was not impressive. The naïve algorithms for these tasks are parallel but also MIMD. Because these kernels execute so quickly already there was little motivation to find SIMT analogs and pursue better GPU performance. Fortunately, every GPU is hosted by a computer with a CPU, so there is no need to use a GPU if the task is not well suited to that hardware. The GPU is a supplement not a substitute for the CPU. This paper is intended to give the reader insight into which types of tasks will do well by being ported to the GPU and which will not.

The issue of programming multiple GPUs is interesting because it requires a completely different type of parallelism to be exploited. A single GPU functions well with massive fine grained (at least 30 k threads) nearly SIMT parallelism. With multiple GPUs which are potentially on different computers connected via MPI and Ethernet cards, very course grained MIMD parallelism is desired. For this reason, all our multi-GPU implementations partition the problem coarsely (into 2 or 4 subsets) for the GPUs, and then use fined grained parallelism within each GPU.

By running our kernel 1 implementation on a variety of Nvidia GPU architectures we have shown that the performance of these devices (on large scientific problems) is strongly correlated with their memory bandwidth (and/or their number of cores). The performance of this new Smith–Waterman algorithm on future GPU architectures can therefore be directly inferred from those future devices memory bandwidth.

Acknowledgments

This work was supported by the Department of Defense and used resources from the Extreme Scale Systems Center at Oak Ridge National Laboratory. Some of the development work occurred on the NSF Teragrid supercomputer, Lincoln, located at NCSA.

Appendix A. Equivalence of the row-parallel algorithm

The row-parallel algorithm has a mathematical form very close to the original Smith–Waterman algorithm. However, the equivalence of these two forms is not trivial. Using Eqs. (3) and (4c) one can write

$$H_n = Max(\widetilde{H}_n, E_n) \tag{A1}$$

From the definition of E_n in Eq. (3) this becomes,

$$H_n = Max\left(\widetilde{H}_n, Max\begin{Bmatrix} H_{n-1} - G_s - G_e \\ H_{n-2} - G_s - 2G_e \\ \vdots \quad \vdots \quad \vdots \\ H_0 - G_s - (n)G_e \end{Bmatrix}\right) \tag{A2}$$

Expanding H_{n-1} in a similar fashion gives,

$$H_{n-1} = Max\left(\widetilde{H}_{n-1}, Max\begin{Bmatrix} H_{n-2} - G_s - G_e \\ H_{n-3} - G_s - 2G_e \\ \vdots \quad \vdots \quad \vdots \\ H_0 - G_s - (n-1)G_e \end{Bmatrix}\right) \tag{A3}$$

After placing Eq. (A3) into Eq. (A2) we have

$$H_n = Max\left(\widetilde{H}_n, Max\begin{Bmatrix} (\widetilde{H}_{n-1}, H_{n-2} - G_s - G_e, H_{n-3} - G_s - 2G_e, \ldots, H_0 - G_s - (n-1)G_e) - G_s - G_e \\ H_{n-2} - G_s - 2G_e \\ \vdots \quad \vdots \quad \vdots \\ H_0 - G_s - nG_e \end{Bmatrix}\right) \tag{A4}$$

Comparing the top row items with the column of items, it is clear that all the items in the top row (except the first one) are smaller than those in the column if $G_s \geqslant 0$. So we can write this as,

$$H_n = Max\left(\widetilde{H}_n, Max\begin{Bmatrix} \widetilde{H}_{n-1} - G_s - G_e \\ H_{n-2} - G_s - 2G_e \\ \vdots \quad \vdots \quad \vdots \\ H_0 - G_s - nG_e \end{Bmatrix}\right) \tag{A5}$$

This process can be repeated. So the next item is,

$$H_{n-2} = Max\left(\widetilde{H}_{n-2}, Max\begin{Bmatrix} H_{n-3} - G_s - G_e \\ H_{n-4} - G_s - 2G_e \\ \vdots \quad \vdots \quad \vdots \\ H_0 - G_s - (n-2)G_e \end{Bmatrix}\right) \tag{A6}$$

And inserting this into (A5) gives

$$H_n = Max\left(\widetilde{H}_n, Max\begin{Bmatrix} \widetilde{H}_{n-1} - G_s - G_e \\ (\widetilde{H}_{n-2}, H_{n-3} - G_s - G_e, H_{n-4} - G_s - 2G_e, \ldots, H_0 - G_s - (n-2)G_e) - G_s - 2G_e \\ H_{n-3} - G_s - 3G_e \\ \vdots \quad \vdots \quad \vdots \\ H_0 - G_s - nG_e \end{Bmatrix}\right) \tag{A7}$$

Again the row items are smaller except the first, so

$$H_n = Max\left(\widetilde{H}_n, Max\begin{Bmatrix}\widetilde{H}_{n-1} - G_s - G_e\\ \widetilde{H}_{n-2} - G_s - 2G_e\\ H_{n-3} - G_s - 3G_e\\ \vdots \quad \vdots \quad \vdots\\ H_0 - G_s - nG_e\end{Bmatrix}\right) \tag{A8}$$

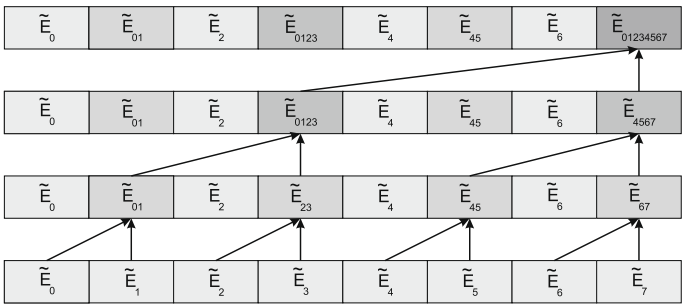
After repeating this substitution for all the items, one obtains,

$$H_n = Max\left(\widetilde{H}_n, Max\begin{Bmatrix}\widetilde{H}_{n-1} - G_s - G_e\\ \widetilde{H}_{n-2} - G_s - 2G_e\\ \vdots \quad \vdots \quad \vdots\\ \widetilde{H}_0 - G_s - nG_e\end{Bmatrix}\right) \tag{A9}$$

And with definition of $\widetilde{E}$ (Eq. (4b)), this shows that,

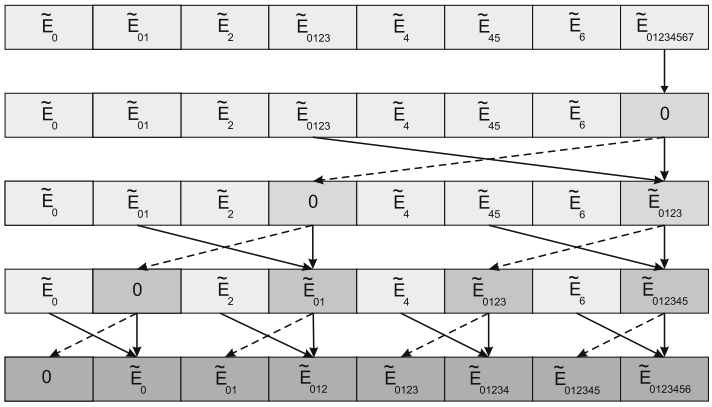
$$H_n = Max\left(\widetilde{H}_n, \widetilde{E}_n - G_s\right) \tag{A10}$$

Which is also Eq. (4c).



$for\ d = 0\ to\ Log_2 n - 1$
  $in\ parallel\ \ for\ k = 0\ to\ n-1\ by\ 2^{d+1}$
    $\tilde{E}[k + 2^{d+1} - 1] = Max\left(\tilde{E}[k + 2^d - 1],\ \tilde{E}[k + 2^{d+1} - 1] + 2^d \times G_e\right)$

Fig. B1. Up-sweep for modified parallel scan for calculating the $\widetilde{E}$ for Smith–Waterman algorithm.



$\tilde{E}[n-1] = 0$
$for\ d = Log_2 n - 1\ \ to\ 0$
  $in\ parallel\ \ for\ k = 0\ to\ n-1\ by\ 2^{d+1}$
    $Temp = \tilde{E}[k + 2^d - 1]$
    $\tilde{E}[k + 2^d - 1] = \tilde{E}[k + 2^{d+1} - 1]$
    $\tilde{E}[k + 2^{d+1} - 1] = Max\left(Temp,\ \tilde{E}[k + 2^{d+1} - 1]\right) - 2^d \times G_e$

Fig. B2. Down-sweep for modified parallel scan for calculating the $\widetilde{E}$ for Smith–Waterman algorithm.

Appendix B. Modified parallel scan

The modified parallel maximum scan algorithm for calculating $\widetilde{E}$ in the row-parallel Smith–Waterman Algorithm is described. The implementation basically uses the work-efficient formulation of Blelloch [23] and GPU implementation of Sengupta et al. [24]. This efficient scan needs two steps to scan the array, called up-sweep and down-sweep. The algorithms for these two steps is shown in Figs. B1 and B2, respectively. Each of these two steps requires *Log n* parallel steps. Since the amount of work becomes half at each step. The overall work is $O(n)$.

Each thread processes two elements and if the number of elements is more than the size of a single block, the array is divided into many blocks and the partial modified scan results are used as an input to the next level of recursive scan. The dark gray cells in Figs. B1 and B2 are the values of $\widetilde{E}$ in these cells that are updated in each step. The dash lines in Fig. B2 means the data is copied to that cell.

References

[1] T.F. Smith, M.S. Waterman, Identification of common molecular subsequences, J. Mol. Biol. 147 (1981) 195–197.
[2] S.F. Altschul, W. Gish, W. Miller, E.W. Myers, D.J. Lipmanl, Basic local alignment search tool, J. Mol. Biol. 215 (1990) 403–410.
[3] D.J. Lipman, W.R. Pearson, Rapid and sensitive protein similarity search, Science 227 (1985) 1435–1441.
[4] W.R. Pearson, D.J. Lipman, Improved tools for biological sequence comparison, Proc. Natl. Acad. Sci. USA 85 (1988) 2444–2448.
[5] A. Wozniak, Using video-oriented instructions to speed up sequence comparison, Comput. Appl. Biosci. 13 (2) (1997) 145–150.
[6] T. Rognes, E. Seeberg, Six-fold speed-up of Smith–Waterman sequence database searches using parallel processing on common microprocessors, Bioinformatics 16 (8) (2000) 699–706.
[7] M. Farrar, Striped Smith–Waterman speeds database searches six times over other SIMD implementations, Bioinformatics 23 (2) (2007) 156–161.
[8] A. Wirawan, C.K. Kwoh, N.T. Hieu, B. Schmidt, CBESW: sequence alignment on the Playstation 3, BMC Bioinformatics 9 (2008) 377–387.
[9] B. Alpern, L. Carter, K.S. Gatlin, Microparallelism and high-performance protein matching, in: ACM/IEEE Supercomputing Conference, San Diego, CA, 1995.
[10] W.R. Rudnicki, A. Jankowski, A. Modzelewski, A. Piotrowski, A. Zadrożny, The new SIMD implementation of the Smith–Waterman algorithm on Cell microprocessor, Fundamenta Informaticae 96 (1/2) (2009) 181–194.
[11] W. Liu, B. Schmidt, G. Voss, A. Schroder, W. Muller-Wittig, Bio-sequence database scanning on GPU, in: Proceeding of the 20th IEEE International Parallel and Distributed Processing Symposium, 2006.
[12] Ł. Ligowski, W. Rudnicki, An efficient implementation of Smith–Waterman algorithm on GPU using CUDA, for massively parallel scanning of sequence databasesm, in: 23rd IEEE International Parallel and Distributed Processing Symposium, May 25–29, Aurelia Convention Centre and Expo Rome, Italy, 2009.
[13] G.M. Striemer, A. Akoglu, Sequence alignment with GPU: performance and design challenges, in: 23rd IEEE International Parallel and Distributed Processing Symposium, May 25–29, Aurelia Convention Centre and Expo Rome, Italy, 2009.
[14] Y. Liu, W. Huang, J. Johnson, Sh. Vaidya, GPU Accelerated Smith–Waterman, ICCS 2006, Part IV, LNCS 3994, pp. 188–195.
[15] S.A. Manavski, G. Valle, CUDA compatible GPU cards as efficient hardware accelerator for Smith–Waterman sequence alignment, BMC Bioinformatics 9 (2) (2008) 10–19.
[16] A. Akoglu, G.M. Striemer, Scalable and highly parallel implementation of Smith–Waterman on graphics processing unit using CUDA, Cluster Comput. 12 (2009) 341–352.
[17] D. Gusfield, Algorithms on Strings, Trees, and Sequences, Cambridge University Press, 1997.
[18] O. Gotoh, An improved algorithm for matching biological sequences, J. Mol. Biol. 162 (1982) 705–708.
[19] Storaasli, Olaf, Accelerating science applications up to 100X with FPGAs, in: Proceedings of Ninth International Workshop on State-of-the-Art in Scientific and Parallel Computing, Trondheim, Norway, May 13–16, 2008.
[20] http://www.slizone.com/page/home.html.
[21] http://ati.amd.com/technology/crossfire/features.html.
[22] http://www.highproductivity.org/SSCABmks.htm.
[23] G.E. Blelloch, Prefix sums and their applications, in: John H. Reif (Ed.), Synthesis of Parallel Algorithms, Morgan Kaufmann, 1990.
[24] S. Sengupta, M. Harris, Y. Zhang, J.D. Owens, Scan primitives for GPU computing, Graphics Hardware (2007) 97–106.