CUDAlign: Using GPU to Accelerate the Comparison of Megabase Genomic Sequences

Edans Flavius de O. Sandes Alba Cristina M. A. de Melo
University of Brasilia (UnB), Brazil
{edans,albamm}@cic.unb.br

Abstract

Biological sequence comparison is a very important operation in Bioinformatics. Even though there do exist exact methods to compare biological sequences, these methods are often neglected due to their quadratic time and space complexity. In order to accelerate these methods, many GPU algorithms were proposed in the literature. Nevertheless, all of them restrict the size of the smallest sequence in such a way that Megabase genome comparison is prevented. In this paper, we propose and evaluate CUDAlign, a GPU algorithm that is able to compare Megabase biological sequences with an exact Smith-Waterman affine gap variant. CUDAlign was implemented in CUDA and tested in two GPU boards, separately. For real sequences whose size range from 1MBP (Megabase Pairs) to 47MBP, a close to uniform GCUPS (Giga Cells Updates per Second) was obtained, showing the potential scalability of our approach. Also, CUDAlign was able to compare the human chromosome 21 and the chimpanzee chromosome 22. This operation took 21 hours on GeForce GTX 280, resulting in a peak performance of 20.375 GCUPS. As far as we know, this is the first time such huge chromosomes are compared with an exact method.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming; J.3 [*Life and Medical Sciences*]: Biology and Genetics

General Terms Algorithms

Keywords Biological Sequence Comparison, Smith-Waterman, GPU

1. Introduction

In the last four years, new DNA sequencing technologies have been developed that allow a hundred-fold increase in the throughput over the traditional method. This means that the genomic databases, that have already an exponential growth rate, will experience an unprecedented increase in their sizes. Therefore, a huge amount of new DNA sequences will need to be compared, in order to infer functional/structural characteristics. In this scenario, the time spent in each comparison, as well as the accuracy of the result obtained, will be a fundamental factor to determine the success/failure of the next generation genome projects.

Sequence comparison is, thus, a very basic and important operation in Bioinformatics. As a result of this step, one or more sequence alignments can be produced [1]. A sequence alignment has a similarity score associated to it that is obtained by placing one sequence above the other, making clear the correspondence between the characters and possibly introducing gaps into them [2]. The most common types of sequence alignment are global and local. To solve a global alignment problem is to find the best match between the entire sequences. On the other hand, local alignment algorithms must find the best match between parts of the sequences.

One important issue to be considered is how gaps are treated. A simple solution assigns a constant penalty for gaps. However, it has been observed that keeping gaps together represents better the biological relationships. Hence, the most widely used model among biologists is the affine gap model [3], where the penalty for opening a gap is higher than the penalty for extending it.

Smith-Waterman (SW) [4] is an exact algorithm based on the longest common subsequence (LCS) concept that uses dynamic programming to find local alignments between two sequences of size m and n in $O(mn)$ space and time. In this algorithm, a similarity matrix of size $(m+1) \times (n+1)$ is calculated. SW is very accurate but it needs a lot of computational resources.

In order to reduce execution time, heuristic methods such as BLAST [5] were proposed. These methods combine exact pattern matching with dynamic programming in order to produce good solutions faster. BLAST can align sequences in a very short time, still producing good results. Nevertheless, there is no guarantee that the best result will be produced.

Therefore, many efforts were made to develop methods and techniques that execute the SW algorithm in high performance architectures, allowing the production of exact results in a shorter time. One recent trend in high performance architectures is the Graphics Processing Units (GPUs). In addition to the usual graphics functions, recent GPU architectures are able to execute general purpose algorithms (GPGPUs). These GPUs contain elements that execute massive vector operations in a highly parallel way. Because of its TFlops peak performance and its availability in PC desktops, the utilization of GPUs is rapidly increasing in many scientific areas.

PPoPP'10, January 9–14, 2010, Bangalore, India.

A	C	T	T	C	C	-	-	A	G	A
A	G	T	T	C	C	G	G	A	G	G
+1	-1	+1	+1	+1	+1	-2	-2	+1	+1	-1

$\sum=1$

Figure 1. Example of alignment and score between sequences S_0=ACTTCCAGA and S_1=AGTTCCGGAGG

In the Bioinformatics research area, there are some implementations of SW in GPUs [6, 7, 8, 9, 10], that were able to produce very good speedups. Nevertheless, all of them impose restrictions on the maximum size of the smallest sequence, that range from 2K[8] to 4MB [6]. That means that two huge sequences cannot be compared in such implementations.

In this paper, we propose and evaluate CUDAlign, a new GPU algorithm that is able to compare sequences of unrestricted size with the SW algorithm, using the affine gap model. Here, unrestricted size means that the implementation is only bound by the available global memory, which can permit comparison of sequences with approximately 100MBP (Megabase Pairs) depending on the GPU. As output, the similarity score and its coordinates in the similarity matrix are provided. The proposed algorithm was executed in two NVIDIA boards separately: 8600GT and GTX 280. In the second board, we were able to obtain 20.375 GCUPS (Giga Cell Updates per Second) when comparing Homo sapiens chromosome 21 with chimpanzee chromosome 22, with respectively 47MBP and 33MBP.

The rest of this paper is organized as follows. In Section 2, we present the Smith-Waterman algorithm with affine gap. Section 3 summarizes the CUDA architecture. In Section 4, related work is discussed. Section 5 describes our proposed GPU algorithm. Experimental results are shown in Section 6. Finally, Section 7 concludes the paper and presents future work.

2. Sequence Alignment

To compare two sequences, we need to find the best alignment between them, which is to place one sequence above the other making clear the correspondence between similar characters [2]. In an alignment, spaces can be inserted in arbitrary locations along the sequences.

In order to measure the similarity between two sequences, a score is calculated as follows. Given an alignment between sequences S_0 and S_1, the following values are assigned, for instance, for each column: a) $+1$, if both characters are identical (match); b) -1, if the characters are not identical (mismatch); and c) -2, if one of the characters is a space (gap).

The score is the sum of all these values. The similarity between two sequences is the highest score. Figure 1 presents one possible alignment between two DNA sequences and its associated score.

In Figure 1, a constant value is assigned to gaps. However, keeping gaps together generates more significant results, in a biological perspective [2]. For this reason, the opening of a gap must have a greater penalty than its extension (affine gap model). The penalty for the first gap is G_{first} and for each successive gap, the penalty is G_{ext}.

The algorithm SW [4] is an exact method based on dynamic programming to obtain the best local alignment between two sequences in quadratic time and space. The

		A	G	T	T	C	C	G	G	A	G	G
	0	0	0	0	0	0	0	0	0	0	0	0
A	0	1	0	0	0	0	0	0	1	0	0	0
C	0	0	**0**	0	0	1	1	0	0	0	0	0
T	0	0	0	**1**	1	0	0	0	0	0	0	0
T	0	0	0	1	**2**	0	0	0	0	0	0	0
C	0	0	0	0	0	**3**	1	0	0	0	0	0
C	0	0	0	0	0	1	**4**	2	0	0	0	0
A	0	1	0	0	0	0	2	**3**	1	1	0	0
G	0	0	2	0	0	0	0	3	**4**	2	2	1
A	0	1	0	1	0	0	0	1	2	**5**	3	1

Figure 2. Similarity matrix between sequences S_0=AGTTCCGGAGG and S_1=ACTTCCAGA. The arrows indicate the cell from where the value was obtained. Bold arrows indicate the traceback to obtain the optimal alignment.

SW algorithm was modified by [3] in order to calculate affine gap penalties. It is divided in two phases: calculate the dynamic programming matrices and obtain the best local alignment.

Phase 1 – Calculate the Dynamic Programming (DP) Matrices - The first phase of the algorithm receives as input sequences S_0 and S_1, with sizes $|S_0| = m$ and $|S_1| = n$. For sequences S_0 and S_1, there are $m + 1$ and $n + 1$ possible prefixes, respectively, including the empty sequence. The notation used to represent the n-th character of a sequence seq is $seq[n]$ and, to represent a prefix with n characters, we use $seq[1..n]$. The similarity matrix is denoted H, where $H_{i,j}$ contains the similarity score between prefixes $S_0[1..i]$ and $S_1[1..j]$. At the beginning, the first row and column are filled with zeroes. The remaining elements of H are obtained from equation 1. In equation 1, $p(i, j) = ma$ if $S_0[i] = S_1[j]$ (match) and mi otherwise (mismatch). In order to calculate gaps according to the affine gap model, two additional matrices E and F are needed. Even with this, time complexity remains quadratic [3]. Equations 2 and 3 are used to calculate matrices E and F respectively. The similarity score between sequences S_0 and S_1 is the highest value in H and the position (i, j) of occurence of this value represents the end of alignment. Note that there can be many positions with the highest value, resulting in many optimal alignments.

$$H_{i,j} = \max \begin{cases} 0 \\ E_{i,j} \\ F_{i,j} \\ H_{i-1,j-1} - p(i,j) \end{cases} \quad (1)$$

$$E_{i,j} = \max \begin{cases} E_{i,j-1} - G_{ext} \\ H_{i,j-1} - G_{first} \end{cases} \quad (2)$$

$$F_{i,j} = \max \begin{cases} E_{i-1,j} - G_{ext} \\ H_{i-1,j} - G_{first} \end{cases} \quad (3)$$

Phase 2 – Obtain the best alignment - In order to retrieve the best local alignment, the algorithm starts from the cell that contains the highest score value and follows the arrows until a zero-valued cell is reached. A left arrow in $H_{i,j}$ (Figure 2) indicates the alignment of $S_0[i]$

with a gap in S_1. An up arrow represents the alignment of $S_1[j]$ with a gap in S_0. Finally, an arrow on the diagonal indicates that $S_0[i]$ is aligned with $S_1[j]$.

3. CUDA architecture

CUDA (Compute Unified Device Architecture) [11] is a general purpose parallel computing architecture introduced by NVIDIA. A GPU with CUDA support is able to run computation intensive algorithms with high speedup compared to CPU executions. CUDA is based on a many-core architecture capable of running a high number of threads in parallel, executing arithmetics operations over data stored in a memory hierarchy.

In CUDA, threads are grouped in blocks with 1, 2 or 3 dimensions, and blocks are also grouped in grids of 1, 2 or 3 dimensions. Threads inside the same block can share data through a fast shared memory and execution can be synchronized by barriers. Threads from different blocks exchange data by a slow global memory and are not usually synchronized.

CUDA extends the C language, allowing the programming of functions that can be executed in GPU. These functions are named kernels and are invoked with an execution configuration that defines at least the number of blocks in the grid (B) and the number of threads in each block (T). This execution configuration is described as the pair $\lll B, T \ggg$. After the invocation, the kernel is executed many times in parallel, each thread running the same kernel code. Threads are distinguished by the thread and block identifiers, accessed by the built-in variables `threadIdx` and `blockIdx`, respectively.

When programming with CUDA, special care must be taken with its memory model. The correct placement of data in the hierarchy is crucial to achieve good speedups. The memory model in CUDA has 5 types of memory: shared memory, global memory, local memory, constant memory and texture memory. The shared memory is a read-write per-block memory which is very fast, if there are no bank conflits. The global memory is a slow read-write per-grid memory, where accesses are not cached and many MB are available. The local memory is in fact the global memory, only allocated per thread. For this reason, it is a slow read-write memory that is usually used to store large data structures. The texture and the constant memory are both read-only per-grid memories, with cached accesses. The main difference between them is that, while the texture memory is slow and big (many MB), the constant memory is fast and small (up to 64KB).

4. Related Work

In the Smith-Waterman (SW) algorithm (Section 2), most of the time is spent calculating the similarity matrix H and this is the part which is usually parallelized. The access pattern presented by the matrix calculation is non-uniform and the parallelization strategy that is traditionally used in this kind of problem is the wavefront method [12], since the calculations that can be done in parallel evolve as waves on diagonals. More specifically, in order to calculate each cell $H[i, j]$, we need to access $H[i-1, j]$, $H[i-1, j-1]$ and $H[i, j-1]$.

Figure 3 illustrates the wavefront method. At the beginning of the computation, only the top-left cell can be calculated. After this computation, two cells in the following antidiagonal can be calculated in parallel. The maximum parallelism is attained at the matrix main antidiagonal and the bottom-right cell is the last to be calculated.

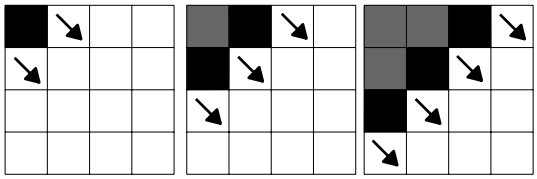

Figure 3. Wavefront execution. Three steps are shown, where each step calculates a diagonal. Black cells represent the cells being processed in the current diagonal, gray cells were already processed and white cells will be processed in the next steps.

Many implementations of the SW algorithm that use Single Instruction Multiple Data (SIMD) optimizations were proposed. Vector instructions, like MMX, SSE, SSE2 and Altivec, were used for reducing the execution time of SW, as shown in [13, 14, 15, 16]. Of those, only [13] uses the wavefront method. The other ones [14, 15, 16] use variations of the SWAT optimization [17]. Nevertheless, in all these SIMD proposals, parallelism is obtained by using SIMD instructions to process several elements simultaneously in each column, row or diagonal. In order to improve the speedup, operations are usually executed with 8-bit operands. However, if the score reaches 255, a new run is made, with larger operands [14, 15].

Several SW implementations were also developed for clusters of computers [18, 19, 20], usually with variants of the wavefront method. These algorithms usually divide the DP matrices into a set of columns or rows, which are assigned in a per node basis. Speedups close to linear, or even superlinear [21, 22], were reported for most implementations. In order to further reduce the execution time, Field Programmable Gate Arrays (FPGAs) have also been used to implement SW [23, 24, 25]. It is also a good alternative, presenting impressive speedups over the software-only implementations, but it is still not commodity hardware. Also, its programming interface is rather complex, since hardware description languages are generally used to describe the circuits.

Recently, it seems to be a trend to use GPUs to implement SW, taking advantage of its massively parallel architecture. With GPUs, impressive speedups can be achieved with commodity hardware, with the advantage of using a programming model that is simpler than the one required to program FPGAs. In the following paragraphs, we discuss some GPU implementations of SW. Unless otherwise stated, all of them execute SW with affine gap, compare a query sequence with a genomic database, and output the score as the result of the computation.

In [6], SW with double affine-gap function is implemented in GPU. In this case, besides the penalty for gap opening, there is a separate penalty for each gap that extends the alignment. Since 16-bit floats are used to store the elements of the matrices, the maximum value for the score is 64K. Also, the size of the query sequence is restricted to 4M. Target sequences are concatenated and between each sequence a special character is inserted. During the DP computation, when a special character is found, the corresponding values are set to zero. This optimization reduces reinitialization overhead of the wavefront. Two modes of operation are available: ASM and ATM. The ASM mode outputs the score and the ATM mode outputs the alignment. In the latter mode, the entire DP matri-

Paper	Algorithm	Output	Max. query	MCUPS	GPU
[6]	SW (double affine gap)	score/alignment	4M	241.12	GeForce 7800 GTX
[7]	SW (affine gap)	score	4K	650	GeForce 7800 GTX
[8]	SW (affine gap)	score	2050	3,612	2×GeForce 8800 GTX
[9]	SW (affine gap)	score	59K	16,087	2×GeForce GTX 295
[10]	SW (affine gap)	score	1K	14,500	GeForce 9800 GX2

Table 1. Comparison of GPU Smith-Waterman papers. All of them compare a query sequence against a database

ces are stored in the texture memory. A total of 241.12 MCUPS and 178.41 MCUPS were achieved for ASM and ATM, respectively.

In [7], a GPU implementation is proposed where, as a first step, the protein sequences and the substitution matrix are stored in the texture memory. Diagonals k, $k-1$ and $k-2$ are stored in separate circular buffers. As an optimization, cells of each diagonal are stored as columns. Query protein sequences of up to 4,095 aminoacids were compared to the SwissProt genomic database.

A Multi GPU-accelerated version of SW is proposed in [8]. In order to optimize the access to the substitution matrix, the authors use a technique called query profile [14], where a specific substitution matrix is pre-computed, replacing the random accesses by sequential ones. Scores are restricted to 16-bit values. Each GPU thread computes the whole alignment between the query sequence and one target sequence (coarse-grained parallelism). Query sequences of up to 2050 aminoacids are supported. Results of 1.889 GCUPS and 3.612 GCUPS were obtained for one and two GPUs, respectively.

In [9], single GPU and multi-GPU versions are provided. Two levels of parallelism exist: inter-task and intra-task parallelization. The first one is coarse-grained and assigns each query × target sequence comparison to a unique thread. The second one assigns several threads to each query × target comparison (fine-grained mode). If the query sequence length is below a threshold, the first mode is used. Otherwise, the second mode is chosen. Subject sequences are pre-ordered by their lengths. Optimized access patterns and block-based accesses are used in order to reduce access times for the texture and global memories. Query sequences of up to 59K are supported. Results of 9.660 GCUPS and 16.087 GCUPS were obtained for one and two GPUs, respectively.

In [10], the proposed implementation compares a query sequence against a database, returning the best score obtained by each alignment. For a single alignment, the DP Matix is divided in groups of 12 rows. Each step processes 12 cells with 2 global memory transactions, giving additional speedup. Results of 7.5 GCUPS and 14.5 GCUPS were obtained for one and two GPUs respectively, when comparing sequences of up to 1000 bases.

Table 1 summarizes the main characteristics of the GPU approaches discussed in the previous paragraphs.

As can be seen in Table 1, all GPU proposals discussed in this paper execute SW with simple affine gap, with the exception of [6], that calculates a double affine gap function. All algorithms compare a query sequence with a protein database. With the exception of [6], that calculates also the alignment on GPU, all proposals provide the similarity score as output. The maximum size allowed for the query sequence varies from 2,050 BP [8] to 4 MBP [6]. The maximum GCUPS obtained was 16.087 [9], with a dual-GPU. As far as we know, there is no proposal using GPU that aligns two Megabase genomic sequences with more than 4MBP.

5. Design of CUDAlign

5.1 General Overview

The GPU implementations of SW discussed in Section 4 do not compare two huge sequences. For these proposals (Table 1), high GCUPS are obtained when comparing small sequences against a large database. Aligning small sequences allows better speedup because the accesses to slower memories can be reduced and also the instructions can manipulate operands with reduced size, like 8-bit. In [10], very high GCUPS were obtained when comparing small sequences, using 16-bit variables. In Bioinformatics, however, there is a need to compare large DNA sequences, to analyze how similar two species are.

The goal of CUDAlign is to align two huge DNA sequences with an exact variant of SW. As output, it provides the similarity score and also the end coordinates of the optimal alignment, allowing future works to backtrace the full alignment.

Given sequences S_0 and S_1 with lengths m and n respectively, the SW algorithm computes a $(m+1) \times (n+1)$ matrix (Section 2). In our proposal, the computation is made with *affine gap*, so we need, in fact, to compute three matrices H, E and F. These three matrices are logically grouped into a single matrix M, where each cell (i, j) contains the three values H_{ij}, E_{ij} and F_{ij}.

In our solution, the cells of matrix M are grouped into blocks with $R \times C$ cells, resulting in a grid G with $\frac{m}{R} \times \frac{n}{C}$ blocks. If the division n/C is not exact, there must be a redistribution of the remaining columns between the blocks, then the blocks may have different number of columns. In contrast, each block must contain exactly R rows. If m is not divisible by R, the last blocks will be padded with rows that will not be processed. Given these observations, we will further consider that n is divisible by C and m is divisible by R.

The values C and R are chosen according to the execution configuration $\lll B, T \ggg$, where B is the number of concurrent blocks running in the grid and T is the number of threads per block. The execution configuration is chosen considering the GPU specifications and empirical results for each board. Given the values B and T, we set $C = \frac{n}{B}$ and $R = \alpha.T$, where α is an integer constant representing the number of rows that each thread is responsible to process.

For example, supose $m = 36$, $n = 36$, $B = 3$, $T = 3$ and $\alpha = 2$. Considering these values, the grid G contains 6×3 blocks, and each block G_{ij} contains 6×12 cells. Grid G is illustrated by Figure 4.

After the cells are grouped into blocks, the blocks are grouped into a set of antidiagonals. Antidiagonal D_k contains the blocks defined by equation $D_k = \{G_{ij} | i + j = k\}$.

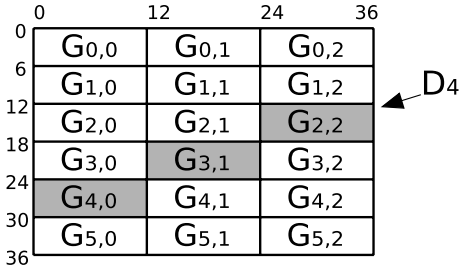

Figure 4. Blocks Grouping when $m = n = 36$, $B = 3$, $T = 3$ and $\alpha = 2$. Blocks in gray represent the external diagonal D_4

These diagonals are called *external diagonals*. The number of external diagonals is $|D| = B + \frac{m}{\alpha T} - 1$ and the number of blocks in the diagonal D_k ranges from 1 to B. Figure 4 shows in gray the blocks that compose external diagonal D_4.

For each diagonal D_k, the CPU invokes a kernel execution over all the blocks of that diagonal in parallel. When the GPU completes the execution of all blocks of that external diagonal, the CPU reinvokes the kernel for the next diagonal, successively until the end of the grid.

Inside the block execution, each block also contains parallelism, that is achieved by many threads processing the diagonals internal to the block. Similar to the grouping of external diagonals, these diagonals are called *internal diagonals*, and the internal diagonal d_k is expressed as the set of cells $d_k = \{(i,j)|\lfloor\frac{i}{\alpha}\rfloor + j = k\}$, considering that values i and j are relative to the top-left cell of the block (i_0, j_0). Each block contains T threads and thread T_k processes rows αk to $\alpha k + \alpha - 1$ of the block, from left to right. All threads calculate in parallel the same internal diagonal. In this way, thread T_i processes cells from column j in the same time as thread T_{i+1} processes cells from column $j - 1$. Figure 5 illustrates the execution of a single block with three threads. Note the diagonal d_4, whose cells can be computed in a parallel way by each thread.

Unlike to the external diagonals, the number of internal diagonals is simply $|d| = \frac{n}{B}$. When the first thread (T_0) hits the last column of the block, all other threads finish their execution, leaving some cells unprocessed. More precisely, thread T_k leaves αk cells unprocessed and the total block execution finishes with $\sum_{k=0}^{T-1} \alpha k = \frac{\alpha}{2} T(T-1)$ pending cells. Figure 5 shows the pending cells as hatched squares.

5.2 Optimizations

5.2.1 Cells Delegation

In order to treat the pending cells, we propose an optimization called *cells delegation*, that is a variation of the sequences aggregation proposed in [6]. With cells delegation, pending cells are processed by other block in the next external diagonal, enabling the first internal diagonal of the next block to start processing with full parallelism, i.e. with all T threads computing values simultaneously. Without this optimization, we would need to restart the wavefront procedure for each block, starting with one thread in the first diagonal, until we achieve the maximum parallelism again, only at diagonal d_{T-1}.

The cells delegation optimization is ilustrated in Figure 6. Delegations are only made between consecutive external diagonals because block $G_{i,j}$ (diagonal $i + j$) only delegates cells to block $G_{i,j+1}$ (diagonal $i + j + 1$). This optimization is also applied to the right-most blocks of the grid. In these boundary situations, the delegation will be made from blocks $G_{i,B-1}$ (last column of the grid) to blocks $G_{i+B,0}$ (first column of the grid) as can be seen in Figure 6 (b). The last external diagonals also delegates cells to the next diagonal (Figure 6 (c)), but the last block does not have next diagonal to delegate. So we need to create one extra diagonal, with only one block (Figure 6 (d)), in order to process the last pending cells. The number of external diagonals is then increased by one, totalizing $|D| = B + \frac{m}{T}$.

Note that in [6], the delegation is made between different target sequences and there is a need to separate sequences by a special character. In the Cells Delegation, the delegation is more complex because it happens between blocks inside the two-dimensional DP matrix, with an extra level of parallelism inside the external diagonal. The boundary test is made with the sizes of the matrix, without the need for a special delimiter character. Observe that

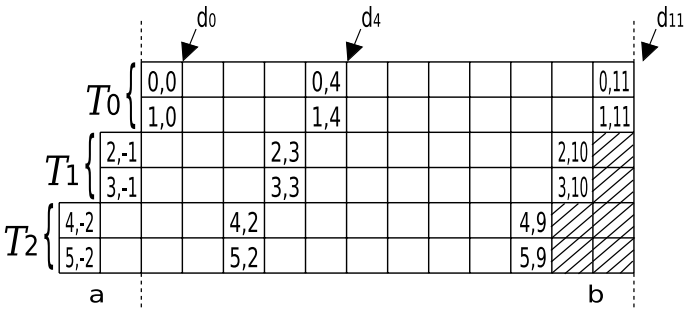

Figure 5. Each thread is responsible for performing two rows ($\alpha = 2$) of the block and threads execute simultaneously in the same internal diagonal. Diagonals range from d_0 to d_{11} in this example. Cell numbers are relative to the first cell of the block, so negative columns (a) represents delegated cells from the left block. Hatched cells (b) are delegated to the right block.

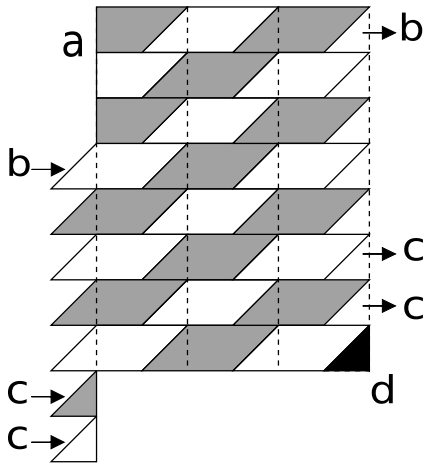

Figure 6. Cells Delegation Details. Gray blocks are from odd external diagonals and white blocks are from even external diagonals. First Blocks (a) does not receive delegated cells. Blocks in the right-most column delegate cells to left-most blocks (b). The last blocks also delegate to the next external diagonal (c), except the very last block, that requires an extra diagonal to process its pending cells. The extra diagonal are ilustrated in black color (d)

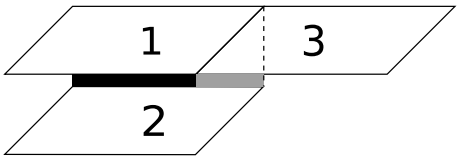

Figure 7. Delegation Hazard

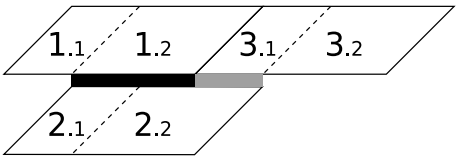

Figure 8. Block Execution divided in two phases. Short and long phases are labeled with .1 and .2 suffixes, respectively

both methods reduce the overhead of reinitialization compared with the conventional wavefront method. This happens because, once the wavefront is filled after processing the first diagonals, the algorithm explores the maximum parallelism, until the very last diagonals.

5.2.2 Phase Division

An implementation detail that must be considered in cells delegation is that, in CUDA, thread blocks can be executed in any order, in parallel or in series [11]. So, the scheduling of blocks of the same external diagonal is unpredictable, which can create a hazard during the delegations cells processing. Figure 7 shows this hazard during the execution of blocks 1, 2 and 3. Block 1 is executed first, because it is in the previous diagonal of blocks 2 and 3. The black bar represents the values calculated when block 1 finishes to be processed and block 2 can read them correctly. Suppose that the second external diagonal is being executed, but the scheduler is not executing the threads of block 3. So, block 2 will start reading all values from top blocks, until it reaches an unspecified area, represented in Figure 7 as a gray bar.

In order to remove this hazard, we must synchronize blocks to ensure that all top cells are ready to be read. In CUDA, there is no primitive to synchronize blocks, so our solution consists on dividing the external diagonal computation in two phases: the short phase and the long phase. The short phase will process T internal diagonals, from all blocks, and then return control to CPU to force block synchronization. After this, the CPU reinvokes the kernel to complete the remaining $\frac{n}{B} - T$ diagonals (long phase). Figure 8 shows the blocks execution split into these two phases. Note that when the short phase terminates, the gray area on Figure 8 was already processed and the long phase will read all values correctly from that area. Although Figure 8 visually presents both phases with almost the same length, note that the short phase is exactly T diagonals wide, but the long phase can contain up to millions diagonals, depending of the size m and the number of blocks B.

The only requirement still needed is that the number of cells that can be read in the short phase is bigger than T. As already explained, the last row $T-1$ of the upper block lefts $T-1$ pending cells. So, the bottom block can read up to $\frac{n}{B} - (T-1)$ cells in the short phase without hazard. Then, the delegation method requires that $n \geq 2BT$ (we round up because we need to compensate when n is not divisable by B). We will call this *minimum size requirement*. As we are dealing with huge sequences, the minimum size requirement will normally hold with the ideal B. Nevertheless, when small sequences are compared, the minimum size requirement is achieved by reducing the number of blocks B per invocation.

Besides removing the hazard, the short/long phase division presents one performance benefit. As shown in Figure 6 (b-d), the last column of blocks delegates cells to the first column of blocks. This requires a condition to test if the current cell belongs to a previous row or if it belongs to the current row. When the condition test detects that the row has changed, some variables must be reinitialized in order to represent the new row of the matrix. Note that this condition is only relevant in the short phase, so we removed this conditional test in the long phase, saving unnecessary computations.

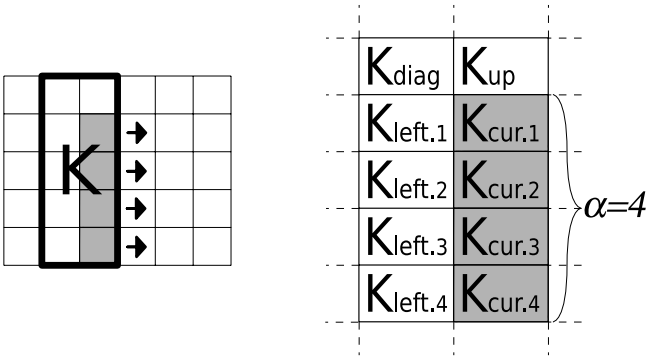

Figure 9. K-neighborhood

5.2.3 Memory Accesses Design

CUDAlign executes one thread for each group of α rows of the block. The computation in the thread is made in the horizontal direction, iterating for each vertical group of α cells. Consider that K^{α}_{cur} represents the cells $(K_{cur.1}, K_{cur.2}, \ldots, K_{cur.\alpha})$ that are being processing in the current iteration and K^{α}_{left} represents the α cells processed in the last iteration. The cells K_{up} and K_{diag} represents the cell above $K_{cur.1}$ and above $K_{left.1}$, respectively. The aggregation of all these cells is represented as $K = (K^{\alpha}_{cur}, K^{\alpha}_{left}, K_{diag}, K_{up})$ and is called K-neighborhood. Figure 9 shows the positions of each element in the K-neighborhood.

During the thread execution, successive groups of α cells are calculated in each iteration, and each iteration presents a new K-neighborhood. For each new iteration, the new K^{α}_{left} is set from the previous K^{α}_{cur} and the new K_{diag} is set from the previous K_{up}.

The values of K-neighborhood are intensively used and, thus, are stored in registers. Nevertheless, the values of K_{up} must be read from the upper thread. If the upper thread resides in the same block, values are loaded from shared memory. If the upper thread resides in a different block, the values of K_{up} are loaded from the global memory, because blocks cannot share data in shared memory.

Similarly, the bottom values of the K-neighborhood (i.e. $K_{cur.\alpha}$) are stored in memory in order to be read by the bottom thread. If the bottom thread resides in the same block, values are stored in shared memory. Otherwise, global memory is used.

The global area used for exchange values of K_{up} and $K_{cur.\alpha}$ is called *horizontal bus* and it has a size that is proportional to the size $|S_1| = n$. In fact, the only values that must be stored in this bus are the 32-bit values of matrices H and F. So, the size of horizontal bus is equal to $8n$ bytes. Because the top row of the matrix must be set to zero, the horizontal bus is firstly initialized with zeroes.

The load from the horizontal bus is made through texture memory because of its cache capability. We are storing and loading elements over the same vector without any hazard, because reads and writes to/from the same position are always made in different moments.

At the begining of the block execution, the values of K^{α}_{left} and of K_{diag} must also be initialized. These values are obtained from the last cells K^{α}_{cur} and K_{up} of the previous block. This is a direct consequence of cells delegation, because the left block delegates cells to the right block. The delegation happens in global memory. The area used to exchange these values is called *vertical bus* and it has a size that is proportional to the total number of threads from all blocks, i.e. $B.T$. In fact, the only values that must be delegated through the vertical bus are the 32-bit values of matrices H and E for each K^{α}_{cur} cells. Additionally, each thread must also save the H component of the K_{up} cell and the F component of the $K_{cur.\alpha}$ cell, so two extra 32-bit words must be included for each thread. So, the size of vertical bus is equals to $(8.\alpha + 2)BT$ bytes.

Because we are dealing with huge sequences, the size of the horizontal bus is the main restriction of CUDAlign. Considering that the memory used by each sequence is 1 byte per base, we can roughly estimate that the total memory usage in CUDAlign is $9n + m$ bytes. So, with a GPU with 1GB of memory, we would be capable to compare two sequences of approximately 100 Megabases.

5.3 Pseudocode

In Algorithm 1, a pseudocode is presented for the kernel explained in Section 5.1 and 5.2. This code is executed by T threads in parallel for each of the B blocks.

The procedure Kernel is invoked with the parameter D_k (line 1), which represents the external diagonals being processed ($0 \leq D_k \leq (\frac{m}{\alpha T} + B - 1)$).

In line 2, the coordinates (i, j) of the top-left cell to be processed by this thread are obtained. If j has a negative value (line 3), values must be adjusted so that the pending cells from αBT rows above can be processed (lines 4-5). In line 7, the first cell delegated by the previous block is obtained from the vertical bus. The α bases associated with the α rows processed by this thread are loaded in variable b^{α} in line 8.

The internal diagonals are calculated in the main loop (lines 9-26). The condition in line 10 ensures that cells out of the matrix will not be processed. In line 11, K_{up} is loaded from the upper thread, that stored its $K_{cur.\alpha}$ value in the previous iteration (line 13). As already explained, K_{up} is read from the global memory (horizontal bus) if the thread is the first of the block (i.e. `threadIdx.x` is zero), or from the shared memory, otherwise.

The Smith-Waterman (SW) computation is made in line 12 for all K^{α}_{cur} values. After the K^{α}_{cur} values are calculated, the maximum local score is verified in line 14. If any K^{α}_{cur} values contains the best score, the maximum value and the best position are both updated in global memory. In lines 15-16 the initialization of the next iteration is made. In line 18, the column index j is incremented and,

Algorithm 1 Process Block (GPU)

```
 1: procedure KERNEL(D_k)
 2:     (i, j) := GETCOORD(D_k, blockIdx.x, threadIdx.x)
 3:     if j < 0 then
 4:         i := i − αBT
 5:         j := |S_1| − j
 6:     end if
 7:     K := GETDELEGATEDCELL(i)
 8:     b^α := S_0[i..i + α − 1]
 9:     for d_k = 0..|d| − 1 do
10:         if i ≥ 0 or i < m then
11:             K_up := LOADUP(threadIdx.x)
12:             K^α_cur := SW(b^α, S_1[j], K^α_left, K_diag, K_up)
13:             STOREDOWN(threadIdx.x + 1, K_cur.α)
14:             UPDATEMAXSCORE(K^α_cur, i, j)
15:             K^α_left := K^α_cur
16:             K_diag := K_up
17:         end if
18:         j := j + 1
19:         if j = |S_1| then
20:             j := 0
21:             i := i + B.αT
22:             K := STARTNEWLINE
23:             b^α := S_0[i..i + α − 1]
24:         end if
25:         __syncthreads()
26:     end for
27:     DELEGATECELL(K, i)
28: end procedure
```

if it exceeds the boundary of the matrix, the thread moves to the first column (line 20) and to αBT rows below (line 21). Also, the algorithm reinitializes the K values (line 22) and the loaded bases b^{α} (lines 23).

We synchronize all threads of the block in line 25, in order to mantain shared memory consistency.

After all internal diagonals be processed, we delegate, in line 27, the K values in order to be processed by the next block. These values are stored in global memory, so that the execution of the next block can continue the pending cells computation.

In Algorithm 2, the pseudocode of the CPU implementation is shown. In lines 2-3, the grid and thread dimensions are initialized. Because we are using one-dimension organization, we set to 1 the other dimensions. The loop in lines 4-7 iterates through each external diagonal $0 \leq D_k \leq |D| + B - 1$. Each iteration executes the short (line 5) and long (line 6) phases of kernel. Although Algorithm 1 does not show the division in short and long phases, in the implementation this division is made, considering that d_k iterates from 0 to $T - 1$ in the short phase and from T to $|d| - 1$ in the long phase. Moreover, in Algorithm 1 the two conditional statements in lines 3-6 and 19-24 are removed from the long phase. As a result, Algorithm 2 returns the best score and its corresponding position (lines 8-9), previously stored in line 14 of Algorithm 1.

6. Results

CUDAlign was implemented in CUDA 2.2 and tested in two NVIDIA GPUs separately: GeForce 8600GT and GTX 280. These boards were connected to the following desktops: AMD Athlon 64 X2 Dual Core Processor 6000+, with 2GB RAM (GeForce 8600GT) and Intel Pentium D CPU

Algorithm 2 Process Grid (CPU)

```
1: function ProcessGrid
2:     grid := (B, 1, 1)
3:     threads := (T, 1, 1)
4:     for D_k := 0..|D| + B − 1 do
5:         KERNEL.1<<< grid, threads >>>(D_k);
6:         KERNEL.2<<< grid, threads >>>(D_k);
7:     end for
8:     (max, max_pos) := GetMaxScore
9:     return (max, max_pos)
10: end function
```

Geforce Name:	8600 GT	GTX 280
Revision	1.1	1.3
Memory	512MB	1GB
# of multiprocessors	4	30
# of cores	32	240
Theoretical GFlops:	113	1008
Clock Rate:	1.19GHz	1.40GHz*

Table 2. Used GPUs. *GTX 280 is 8% overclocked

Comparison	GeForce 8600GT		GeForce GTX280	
	Time	MCUPS	Time	MCUPS
*128K×128K	8.6s	1895	1.1s	15277
162K×172K	14.5s	1915	1.7s	16421
*256K×256K	34.1s	1923	3.7s	17837
*512K×512K	135s	1939	13.7s	19147
543K×536K	150s	1941	15.2s	19212
*1000K×1000K	514s	1947	50.6s	19773
1044K×1073K	569s	1968	56.6s	19813
*2000K×2000K	2050s	1951	199s	20106
*3000K×3000K	4611s	1952	446s	20189
3147K×3283K	5291s	1953	512s	20196
5227K×5229K	13982s	1955	1348s	20278
7146K×5227K	19120s	1954	1841s	20289
23012K×24544K	-	-	27730s	20367
32799K×46944K	-	-	75571s	20375

Table 4. Runtimes of CUDAlign. Random sequences are marked with '*'

3.40GHz, with 4GB RAM (GTX280). Table 2 presents detailed information about both GPUs.

In our tests, we used real DNA sequences retrieved from the NCBI site (www.ncbi.nlm.nih.gov). The names of the sequences compared, as well as their sizes, are shown in Table 3. As can be seen in Table 3, the sizes of the real sequences range from 162 KBP to 47 MBP. In some tests, random generated DNA sequences were also used.

The best scores and their ending positions obtained for the comparison of real sequences are listed on Table 5, including the longest comparison, between *Homo sapiens* chromosome 21 and *Pan troglodytes* chromosome 22. With the exception of this comparison, the sequences were the same chosen in [19]. The Smith-Waterman score parameters used in the tests were: match: $+1$; mismatch -3; first gap: -5; extention gap: -2. The execution configurations used for 8600GT was $B = 2^4$ and $T = 2^6$ and for GTX280 was $B = 2^9$ and $T = 2^6$. For both boards, we set $\alpha = 4$ rows per thread. Execution times and MCUPS on each board are presented on Table 4.

An usual performance metric for SmithWaterman implementation is the number of cell updates per second (CUPS) and recent GPU implementations have shown performances up to 16 GCUPS (billions of cell updates per second) with dual GPUs. The CUPS metrics is calculated with the formula $\frac{mn}{t \times 10^9}$ where m and n are the sizes of both sequences S_0 and S_1 respectively. When the implementations compare a query sequence Q against a database D, the m and n values in the CUPS formula are replaced by the size of the query sequence $|Q|$ and the size of the database $|D|$.

In Table 4, it can be noted that the GCUPS rates of both GPUs for sequences longer than 1MBP range from 1.946 to 1.968 GCUPS (8600GT) and 19.773 to 20.375 GCUPS (GTX 280). As we increase the sizes of the megabase sequences, the GCUPS values tend to increase. This happens because the overhead of block reinitialization is proportionally reduced as n grows. Note that the difference from the peak and the lower GCUPS is 3% in the GTX280. This shows the potential for scalability of our design. The GCUPS obtained by our algorithm when running in GeForce GTX 280 are higher than all the GCUPS obtained by the other works discussed in Section 4 (Table 1). Nevertheless, we cannot make a direct comparison basically for two reasons. First, CUDAlign was tested against DNA sequences and no substitution matrix was fetched from memory, as other works do. Second, the proposals discussed in Section 4 impose a restriction on the size of the smallest sequence and, therefore, can use some memory access optimizations that are not possible in CUDAlign.

As stated in [26], human-chimpanzee comparative genomics is fundamental to determine the genetic changes that led to the acquisition of unique human features. In this scenario, one comparison of extraordinary interest is the comparison between the human chromosome 21 and the chimpanzee (*Pan troglodytes*) chromosome 22. In [26], this comparison was done with BLAST, since the huge sizes of the chromosomes (47MBP and 33MBP, respectively) prevented the use of exact methods. As far as we know, there is no Smith-Waterman based exact method, implemented either in clusters, FPGAs or GPUs, that compared sequences as huge as those chromosomes.

In Tables 5 and 4, we present the results obtained when comparing these two chromosomes with CUDAlign. The huge score obtained (27,206,434) reveals the great similarity between these genomic sequences and can be an important indication to establish our evolutionary history. Also, as can be seen in Table 4, this comparison took less than 21 hours (75,571s) on the GTX 280 GPU. This achievement was only possible because of the careful design of the CUDAlign and the optimized placement over the memory hierarchy.

The graphic in Figure 10 presents the results in a logarithmic scale. The MCUPS dotted lines in Figure 10 show the peak performance with 1,968 MCUPS and 20,375 MCUPS, for Geforce 8600GT and Geforce GTX 280 respectively.

Table 6 shows the BLAST results and execution times for the real sequences, executed on an Athlon 64 X2 Dual Core Processor 6000+, with 2GB DDR2 RAM. As BLAST is a heuristic method, its runtime is extremely low when compared to exact methods such as CUDAlign. Nevertheless, BLAST results usually diverge significantly when the sequences are long and the optimal score is high. For instance, for the 5227K $\times$ 5229K comparison, CUDAlign obtained a score of 5,220,960 (Table 5), while the best score obtained by BLAST was 36,159 (Table 6). In the human-

Aprox. Size	Real size	Accession Number	Name
162KBP	162,114 BP	NC_000898.1	*Human Herpesvirus* 6B
172KBP	171,823 BP	NC_007605.1	*Human Herpesvirus* 4
543KBP	542,868 BP	NC_003064.2	*Agrobacterium tumefaciens*
536KBP	536,165 BP	NC_000914.1	*Rhizobium sp.*
1MBP	1,044,459 BP	CP000051.1	*Chlamydia trachomatis*
1MBP	1,072,950 BP	AE002160.2	*Chlamydia muridarum*
3MBP	3,147,090 BP	BA000035.2	*Corynebacterium efficiens*
3MBP	3,282,708 BP	BX927147.1	*Corynebacterium glutamicum*
5MBP	5,227,293 BP	AE016879.1	*Bacillus anthracis* str. Ames
5MBP	5,227,293 BP	NC_003997.3	*Bacillus anthracis* str. Ames
5MBP	5,228,663 BP	AE017225.1	*Bacillus anthracis* str. Sterne
7MBP	7,145,576 BP	NC_005027.1	*Rhodopirellula baltica* SH 1
23MBP	23,011,544 BP	NT_033779.4	*Drosophila melanog.* chromosome 2L
25MBP	24,543,557 BP	NT_037436.3	*Drosophila melanog.* chromosome 3L
33MBP	32,799,110 BP	BA000046.3	*Pan troglodytes DNA*, chromosome 22
47MBP	46,944,323 BP	NC_000021.7	*Homo sapiens* chromosome 21

Table 3. Real sequences details. Sizes range from 162KBP to 47MBP.

Comparison	Cells	S_0	S_1	Score	Position
162K×172K	2.79E+10	NC_000898.1	NC_007605	18	(41058 , 44353)
543K×536K	2.91E+11	NC_003064.2	NC_000914.1	48	(308558 , 455134)
1044K×1073K	1.12E+12	CP000051.1	AE002160.2	88353	(1072950 , 722725)
3147K×3283K	1.03E+13	BA000035.2	BX927147.1	4226	(2991493 , 2689488)
5227K×5229K	2.73E+13	AE016879.1	AE017225.1	5220960	(5227292 , 5228663)
7146K×5227K	3.74E+13	NC_005027.1	NC_003997.3	172	(4655867 , 5077642)
23012K×24544K	5.65E+14	NT_033779.4	NT_037436.3	9063	(14651731 , 11501313)
32799K×46944K	1.54E+15	BA000046.3	NC_000021.7	27206434	(32718231 , 46919080)

Table 5. Comparison for the real sequences used in tests. The best local score and the end position are presented.

Comparison	BLAST	
	Time	Score
162K×172K	0.4s	18
543K×536K	0.6s	48
1044K×1073K	2.4s	6973
3147K×3283K	6.7s	3888
5227K×5229K	17.4s	36159
7146K×5227K	7.7s	157
23012K×24544K	110s	7085
32799K×46944K	-	-

Table 6. BLAST Results.

chimpanzee chromosome comparison, BLAST finished its execution with a segmentation fault, due to an out-of-memory error.

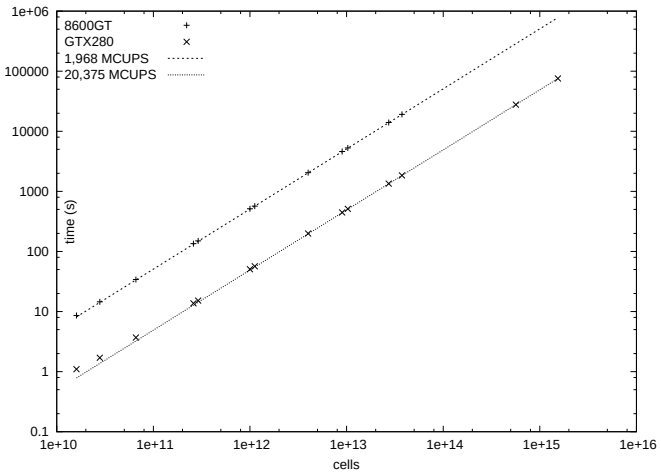

Figure 10. Runtimes (seconds) × DP matrix size (cells) in logarithm scale. Results show scalability and almost constant MCUPS ratio for Megabase sequences (cells $\geq 1e+12$).

7. Conclusion and Future Work

In this paper, we proposed and evaluated CUDAlign, a GPU-accelerated version of Smith-Waterman (SW) that compares two Megabase genomic sequences. Differently from the previous GPU Smith-Waterman (SW) proposals in the literature, our proposal does not impose severe restrictions on the size of the smallest sequence and that allows, for instance, the comparison of two entire chromosomes. CUDAlign is able to heavily exploit the parallelism present in the wavefront SW computations in such a way that the only moment that there are idle threads is at the very beginning and at the very end of the matrix calculations. Also, memory accesses were carefully designed in order to exploit the characteristics of the GPU memory hierarchy.

The experimental results obtained with real and synthetic DNA sequences in two GPU boards show that an almost uniform GCUPS is obtained when the sizes of the sequences increase. This indicates that our algorithm is scalable for Megabase sequence comparisons. In order to compare DNAsequences of size 47MBP and 33MBP, CU-

DAlign took less than 21 hours, with a sustained 20.375 GCUPS. This is an impressive result, since a shorter comparison of 23012K $\times$ 24544K in [19] took more than four days in a 64-processor cluster.

By now, our algorithm outputs the similarity score and the coordinates that correspond to the end of the optimal alignment.

As future work, we intend to extend CUDAlign to return the full alignment. The alignment retrieval for Megabase sequences is a very challenging problem, since we obviously cannot store the whole DP matrices in this case. For instance, the human-chimpanzee chromosome comparison would require more than 16PB for the whole DP matrices. Therefore, several memory usage reduction techniques must be combined to obtain the full alignment.

Also, we intend to align other relevant sequences and extend the tests to more powerful GPUs. Finally, we plan to migrate our work to the OpenCL framework, taking advantage on its heterogeneity capability for testing the algorithm in other platforms.

Acknowledgments

We would like to thank Prof. Rafael Morgado Silva (UnB) for letting us use the NVIDIA GTX 280 board. This work is partially supported by FAPDF/Brazil, CNPq/Brazil and FINEP/Brazil.

References

[1] S. Batzoglou. The many faces of sequence alignment. *Brief Bioinform*, 6(1):6–22, March 2005.

[2] Richard Durbin, Sean R. Eddy, Anders Krogh, and Graeme Mitchison. *Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids*. Cambridge University Press, 1999.

[3] O. Gotoh. An improved algorithm for matching biological sequences. *J Mol Biol*, 162(3):705–708, December 1982.

[4] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *J Mol Biol*, 147(1):195–197, March 1981.

[5] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic local alignment search tool. *J Mol Biol*, 215(3):403–410, October 1990.

[6] John Johnson Yang Liu, Wayne Huang and Sheila Vaidya. Gpu accelerated smith-waterman. In *Computational Science – ICCS 2006*, volume 3994 of *LNCS*, pages 188–195. Springer, 2006.

[7] Weiguo Liu, B. Schmidt, G. Voss, A. Schroder, and W. Muller-Wittig. Bio-sequence database scanning on a gpu. *IPDPS*, page 274, 2006.

[8] Svetlin Manavski and Giorgio Valle. Cuda compatible gpu cards as efficient hardware accelerators for smith-waterman sequence alignment. *BMC Bioinformatics*, 9(Suppl 2), 2008.

[9] Yongchao Liu, Douglas Maskell, and Bertil Schmidt. Cudasw++: optimizing smith-waterman sequence database searches for cuda-enabled graphics processing units. *BMC Research Notes*, 2(1):73, 2009.

[10] Lukasz Ligowski and Witold Rudnicki. An efficient implementation of Smith-Waterman algorithm on GPU using CUDA, for massively parallel scanning of sequence databases. In *IEEE International Workshop on High Performance Computational Biology (HiCOMB 2009)*, 2009.

[11] NVIDIA. *NVIDIA CUDA Programming Guide 2.1*. 2008.

[12] Gregory F. Pfister. *In search of clusters: the coming battle in lowly parallel computing*. Prentice-Hall, Inc., 1995.

[13] A. Wozniak. Using video-oriented instructions to speed up sequence comparison. *Computer Applications in the Biosciences*, 13(2):145–150, 1997.

[14] T. Rognes and E. Seeberg. Six-fold speed-up of smith-waterman sequence database searches using parallel processing on common microprocessors. *Bioinformatics*, 16(8):699–706, August 2000.

[15] M. Farrar. Striped smith-waterman speeds database searches six times over other simd implementations. *Bioinformatics*, 23(2):156–161, January 2007.

[16] Adrianto Wirawan, Chee K. Kwoh, Tri H. Nim, and Bertil Schmidt. Cbesw: Sequence alignment on the playstation 3. *BMC Bioinformatics*, 9, September 2008.

[17] P. Green. Url: http://www.phrap.org/phredphrap/swat.html.

[18] Stjepan Rajko and Srinivas Aluru. Space and time optimal parallel sequence alignments. *IEEE Trans. Parallel Distrib. Syst.*, 15(12):1070–1081, 2004.

[19] Rodolfo Bezerra Batista, Azzedine Boukerche, and Alba Cristina Magalhaes Alves de Melo. A parallel strategy for biological sequence alignment in restricted memory space. *J. Parallel Distrib. Comput.*, 68(4):548–561, 2008.

[20] Chunxi Chen and Bertil Schmidt. Computing large-scale alignments on a multi-cluster. *Cluster Computing, IEEE International Conference on*, page 38, 2003.

[21] Azzedine Boukerche, Alba Cristina Melo, Edans Flavius Sandes, and Mauricio Ayala-Rincon. An exact parallel algorithm to compare very long biological sequences in clusters of workstations. *Cluster Computing*, 10(2):187–202, 2007.

[22] Xiandong Meng and Vipin Chaudhary. Optimised fine and coarse parallelism for sequence homology search. *Int. J. Bioinformatics Res. Appl.*, 2(4):430–441, 2006.

[23] Peiheng Zhang, Guangming Tan, and Guang R. Gao. Implementation of the smith-waterman algorithm on a reconfigurable supercomputing platform. In *HPRCTA '07: Proc. of the 1st int. workshop on High-performance reconfigurable computing technology and applications*, pages 39–48, 2007.

[24] L. Xu P. Zhang N. Sun X. Jiang, X. Liu. "a reconfigurable accelerator for smith-waterman algorithm". *"IEEE Transactions on Circuits and Systems II"*, 54(12):1077–1081, December 2007.

[25] Azzedine Boukerche, Jan Mendonca Correa, Alba Cristina Magalhaes Alves de Melo, Ricardo P. Jacobi, and Adson Ferreira Rocha. Reconfigurable architecture for biological sequence comparison in reduced memory space. In *IPDPS*, pages 1–8, 2007.

[26] The international chimpanzee chromosome 22 consortium. Dna sequence and comparative analysis of chimpanzee chromosome 22. *Nature*, 429(6990):382–388, May 2004.