Retrieving Smith-Waterman Alignments with Optimizations for Megabase Biological Sequences using GPU

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

Abstract—In Genome Projects, biological sequences are aligned thousands of times, in a daily basis. The Smith-Waterman algorithm is able to retrieve the optimal local alignment with quadratic time and space complexity. So far, aligning huge sequences, such as whole chromosomes, with the Smith-Waterman algorithm has been regarded as unfeasible, due to huge computing and memory requirements. However, high performance computing platforms such as GPUs are making it possible to obtain the optimal result for huge sequences in reasonable time. In this paper, we propose and evaluate CUDAlign 2.1, a parallel algorithm that uses GPU to align huge sequences, executing the Smith-Waterman algorithm combined with Myers-Miller, with linear space complexity. In order to achieve that, we propose optimizations which are able to reduce significantly the amount of data processed, while enforcing full parallelism most of the time. Using the NVIDIA GTX 560 Ti board and comparing real DNA sequences that range from 162 KBP (Thousand Base Pairs) to 59 MBP (Million Base Pairs), we show that CUDAlign 2.1 is scalable. Also, we show that CUDAlign 2.1 is able to produce the optimal alignment between the chimpanzee chromosome 22 (33 MBP) and the human chromosome 21 (47 MBP) in 8.4 hours and the optimal alignment between the chimpanzee chromosome Y (24MBP) and the human chromosome Y (59MBP) in 13.1 hours.

1 Introduction

Bioinformatics is an interdisciplinary field that involves computer science, biology, mathematics and statistics [1]. One of its main goals is to analyze biological sequence data and genome content in order to obtain the function/structure of the sequences as well as evolutionary information.

Once a new biological sequence is discovered, its functional/structural characteristics must be established. The first step to achieve this goal is to compare the new sequence with the sequences that compose genomic databases, in search of similarities. This comparison is made thousands of times in a daily basis, all over the world. Sequence comparison is, therefore, one of the most basic operations in Bioinformatics. As output, a sequence comparison operation produces similarity scores and alignments. The score is a measure of similarity between the sequences and the alignment highlights the similarities/differences between the sequences. Both are very useful and often are used as building blocks for more complex problems such as multiple sequence alignment and secondary structure prediction.

Smith and Waterman (SW) [2] proposed an exact algorithm that retrieves the optimal score and local alignment between two sequences. It is based on dynamic programming (DP) and has time and space complexity $O(mn)$, where m and n are the sizes of the sequences. In SW, a linear gap function was used. Nevertheless, in the nature, gaps tend to occur together. For this reason, the affine gap model is often used, where the penalty for opening a gap is higher than the penalty for extending it. Gotoh [3] modified the SW algorithm to include affine gap penalties.

One of the most restrictive characteristics of SW and its variants is the quadratic space needed to store the DP matrices. For instance, in order to compare two 33 MBP (Millions of Base Pairs) sequences, we would need at least 4.3 PB of memory. This fact was observed by Hirschberg [4], who proposed a linear space algorithm to compute the Longest Common Subsequence. Hirschberg's algorithm was later modified by Myers and Miller (MM) [5] to compute global alignments in linear space.

Another restrictive characteristic of the SW algorithm is that it is usually slow due to its quadratic time complexity. In order to accelerate the comparison between long sequences, heuristic tools such as LASTZ [6] and MUMMER [7] were created. They use seeds (LASTZ) and suffix trees (MUMMER) to scan the sequences, providing a big picture of the main differences/similarities between them. On the other hand, Smith-Waterman (SW) provides the optimal local alignment, where the regions of differences/similarities are much more accurate, as well as the gapped regions that represent inclusion/deletion of bases. Therefore, we claim that both kinds of tools should be used in a complementary way: first, MUMMER or LASTZ would be executed and then SW would be run to obtain the optimal local

Digital Object Indentifier 10.1109/TPDS.2012.194

alignment for the cases where a more detailed analysis should be made.

Many efforts have also been made to reduce the execution time of SW and parallel versions were created for clusters [8–11], FPGAs [12–14], Grids [15] and CellBEs [16, 17]. Even though most of these approaches retrieve only the score, some of them [9, 15, 17] are able to retrieve the alignment in linear space.

In the last years, Graphics Processing Units (GPUs) have received a lot of attention because of their TFlops peak performance and their availability in PC desktops. In the Bioinformatics research area, there are many implementations of SW in GPUs [18–27], which were able to achieve impressive speedups. Nevertheless, with the exception of CUDAlign 1.0 [27], all of these implementations impose a maximum size for the query sequence. This means that two huge sequences cannot be compared in such implementations.

CUDAlign 1.0 [27] is able to obtain the score of the optimal alignment, given two huge input sequences. CUDAlign 2.0 [28] extended CUDAlign 1.0 in order to produce both the score and the full optimal alignment. In CUDAlign 2.0, the alignment is obtained with a combined Myers-Miller and Smith-Waterman (MM+SW) algorithm. This implementation is only bound to the total available global memory in the GPU and the available disk space in the desktop.

In this paper, we propose and evaluate CUDAlign 2.1, extending the work in [28] in the following way. First, a new optimization called block pruning is proposed, formally defined and evaluated. Second, we empirically evaluate a wide range of CUDA parameters identifying the values that will provide better performance. Third, we included the human and chimpanzee Y chromosomes in our test set and executed all comparisons in a different GPU (NVIDIA GeForce GTX 560 Ti).

CUDAlign 2.1 receives two input sequences and provides the optimal score and local alignment as output. It runs in 6 stages, where the first three stages are executed in GPU and the last three stages run in CPU. The first stage calculates the SW DP matrices, retrieving the optimal score and its position in the DP matrices. Some special rows are saved to disk and a new optimization called block pruning is used in this stage. The goal of stages 2, 3 and 4 is to retrieve points where the optimal alignment occurs in special rows/columns, thus creating smaller sub-problems. In Stage 5, the alignments for all subproblems are obtained and concatenated to generate the full alignment. In Stage 6, the alignment can be visualized.

The proposed algorithm was implemented in CUDA, C++ and pthreads and executed in the GTX 560 Ti board. With our algorithm, we were able to retrieve the alignment for real sequences whose size range from 150KBP (Thousands Base Pairs) to 59 MBP (Millions of Base Pairs). We show that our algorithm is scalable and it is able to achieve a sustained performance of 28 GCUPS (Billions of Cells Updated per Second) for sequences longer than 3 MBP. We also show the impressive performance gain of our block pruning optimization when sequences with high similarity are compared. In our tests, the block pruning optimization achieved up to 51% of performance gain.

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

$score{=}1$

Figure 1. Alignment and score between sequences S_0 and S_1.

The remainder of this paper is organized as follows. In Section 2, we present the Smith-Waterman and the Myers-Miller algorithms. In Section 3, related work is discussed. Section 4 describes our proposed algorithm. Experimental results are shown in Section 5. Finally, Section 6 concludes the paper and presents future work.

2 Biological Sequence Alignment

To compare two sequences, we need to find the optimal alignment between them, which is to place one sequence above the other making clear the correspondence between similar characters [1]. In an alignment, spaces can be inserted in arbitrary locations along the sequences. Basically, an alignment can be a) global, containing all characters of the sequences; b) local, containing substrings of the sequences; or c) semi-global, composed of prefixes or suffixes of the sequences, where leading/trailing gaps are ignored.

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) $ma = +1$, if both characters are identical (match); b) $mi = -1$, if the characters are not identical (mismatch); and c) $G = -2$, if one of the characters is a space (gap). The score is the sum of all these values. 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 [1]. For this reason, the first 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 difference $G_{first} - G_{ext}$ is the gap opening penalty G_{open}.

2.1 Smith-Waterman Algorithm (SW)

The algorithm SW [2] is an exact method based on dynamic programming to obtain the optimal local alignment between two sequences in quadratic time and space. The SW algorithm was modified by Gotoh [3] in order to calculate affine gap penalties. It is divided in two phases: calculate the Dynamic Programming (DP) matrices and obtain the optimal local alignment.

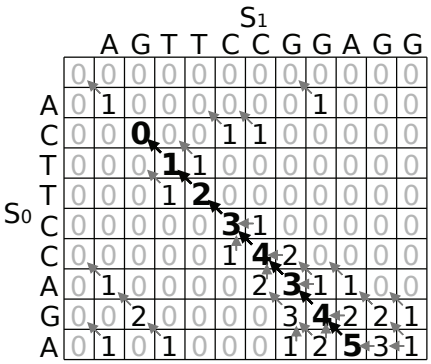


Figure 2. DP matrix for sequences S_0 and S_1. Bold arrows indicate the traceback to obtain the optimal alignment.

Phase 1 – Calculate the DP Matrices - The first phase of the algorithm receives as input sequences S_0 and S_1, with sizes m and n respectively. 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, $p(i,j) = ma$ (match) if $S_0[i] = S_1[j]$ and mi (mismatch) otherwise. In order to calculate gaps according to the affine gap model, two additional matrices E (Equation 2) and F (Equation 3) are needed. Even with this, time complexity remains quadratic [3]. The optimal score between sequences S_0 and S_1 is the highest value in H and the position (i,j) where this value occurs represents the end of alignment.

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

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

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

Phase 2 - Obtain the optimal alignment - To retrieve the optimal local alignment, the algorithm starts from the cell that contains the highest score and follows the arrows until a zero-valued cell is reached (Figure 2). A left arrow in $H_{i,j}$ 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 $S_1[j]$.

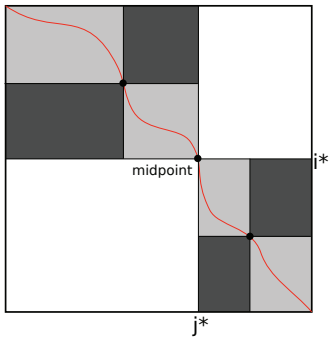


Figure 3. Recursive splitting procedure in MM.

2.2 Myers-Miller Algorithm (MM)

For long sequences, space is a limiting factor for the optimal alignment computation. Myers and Miller proposed an algorithm [5] that computes optimal global alignments in linear space. It is based on Hirschberg [4], but applied over Gotoh [3].

Hirschberg's algorithm uses a recursive divide and conquer procedure to obtain the longest common subsequence (LCS). The idea of this algorithm is to find the midpoint of the LCS using the information obtained from the forward and the reverse directions, maintaining in memory only one row for each direction (thus in linear space). Given this midpoint, the problem is divided in two smaller subproblems, that are recursively divided in more midpoints.

The MM algorithm works as follows [5]. Let S_0 and S_1 be the sequences, with sizes m and n respectively, and $i* = \frac{m}{2}$ the middle row of the DP matrices. In the forward direction, $CC(j)$ is the minimum cost of a conversion of $S_0[1..i*]$ to $S_1[1..j]$ that ends without a gap and $DD(j)$ is the minimum cost of a conversion of $S_0[1..i*]$ to $S_1[1..j]$ that ends with a gap. In the reverse direction, $RR(n-j)$ is the minimum cost of a conversion of $S_0[i*..m]$ to $S_1[j..n]$ that begins without a gap and $SS(n-j)$ is the minimum cost of a conversion of $S_0[i*..m]$ to $S_1[j..n]$ that begins with a gap.

To find the midpoint of the alignment, the algorithm executes a *matching procedure* between: a) vectors CC and RR; b) vectors DD and SS. The midpoint is the coordinate $(i*, j*)$, where $j*$ is the position that satisfies the maximum value in Formula 4 [5].

$$max_{j \in [0..n]} \left\{ max \begin{cases} CC(j) + RR(n-j) \\ DD(j) + SS(n-j) - G_{open} \end{cases} \right. \quad (4)$$

After the midpoint is found, the problem is recursively split into smaller subproblems (Figure 3), until trivial problems are found. The matching of CC with RR represents a junction of the forward alignment with the reverse alignment without a gap and the matching of DD with SS represents the junction with a gap. When there is a gap in both directions, both receive a gap opening penalty, so this duplicated penalty must be adjusted.

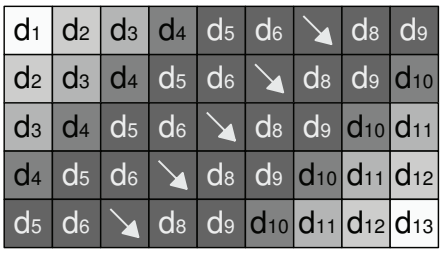


Figure 4. Wavefront Method

Paper	Align	Max. Query	GCUPS	GPU
DASW[18]	yes	16,384	0.2	7800 GTX
Weiguo Liu[19]	no	4,095	0.6	7800 GTX
SW-CUDA[20]	no	567	3.4	8800 GTX
CUDASW++ 1.0[21]	no	5,478	16.1	GTX 295
Ligowski[22]	no	1,000	14.5	9800 GX2
CUDASW++ 2.0[23]	no	5,478	29.7	GTX 295
CUDA-SSCA#1[24]	yes	1,024	1.0	GTX 295
CUDASW++ 1.0i[25]	no	5,478	29.3	C2050
MSA-CUDA[26]	yes	~858	N/A	GTX 280
CUDAlign 1.0[27]	no	32,799,110	20.3	GTX 285

Table 1
GPU Smith-Waterman papers.

2.3 Wavefront Method

In the Smith-Waterman (SW) and the Myers-Miller (MM) algorithms, most of the time is spent calculating the DP matrices and this is the part which is usually parallelized. The access pattern presented by the matrices calculation is non-uniform and the parallelization strategy that is traditionally used in this kind of problem is the wavefront method [29], since the calculations that can be done in parallel evolve as waves on diagonals.

Figure 4 illustrates the wavefront method. Note that in step 1, only one cell is calculated in diagonal d_1. In step 2, diagonal d_2 has 2 cells and two operations can be done in parallel. Assuming that there are T threads, we say that the wavefront is full when there are T elements of the matrices being calculated in parallel. In the example of Figure 4, assuming that there are 5 threads, the wavefront is full only at diagonals d_5 to d_9. Therefore, at the beginning of the computation, the wavefront needs to be filled and, at the end of the computation, the wavefront is emptied.

3 Related Work

In this section, some variants of SW are discussed. First, implementations with linear space complexity are presented. Second, GPU implementations are discussed.

3.1 Linear Space Parallel Algorithms

In [30], the MM algorithm (Section 2.2) is implemented using an additional matrix $CROSS$ where cell $CROSS[i, j]$ contains the column where the optimal alignment between $S_0[1..i]$ and $S_1[1..j]$ crosses the middle row. With this additional matrix, the midpoint is directly accessed at cell $CROSS[m, n]$. An extension of this algorithm was also proposed using k dividing rows. With this technique, the divide and conquer algorithm can divide a problem in k subproblems, giving faster runtimes than MM, with some memory tradeoff.

In [8], an algorithm using parallel prefix computation is presented. The DP matrices are divided into $p - 1$ special columns, creating p partitions, each one being processed by one processor. During the computation of the partition, each matrix element $T[i, j]$ maintains a pointer to a cell, at the closest special column, that belongs to the optimal alignment. At the end, the traceback passing through each special column is done by following the pointers until the beginning of the matrix. Each intersection generates subproblems that are processed recursively. This algorithm can obtain the local alignment in three phases: forward, reverse and alignment matching. In [9], the work in [8] was improved by dividing the matrix in both horizontal and vertical directions, in $p-1$ special columns and $p-1$ special rows. This algorithm runs in $O(\frac{m+n}{p})$ space and in $O(\frac{mn}{p})$ time. It was able to compare 1.1 MBP sequences in a 60-processor cluster.

FastLSA [31] is a variant of the MM algorithm that is based on the divide and conquer method. If the full matrix can be computed in memory, the base case is solved. Otherwise, the algorithm processes the matrix and stores k rows/columns in memory. After, the problem is subdivided in smaller subproblems that are processed recursively until the base case is found. The full alignment is the concatenation of the outputs of all subproblems. In a 32-processor cluster, it was possible to compare two 300 KBP sequences.

3.2 GPU Algorithms

Recently, many GPU implementations of SW have been proposed. Table 1 summarizes the main characteristics of them and the best results presented in the papers.

The works in [18–25] tackle the problem of finding the most similar sequence in a genomic database, given a protein query sequence. Most of these approaches are able to calculate more than 1 Billion cells of the DP matrices per second (GCUPS) and most of them do not compute the alignment in GPU, providing only the score as output. In [18] and [24] the alignment is obtained in quadratic space and the experimental results compared small query sequences (16 KBP and 1 KBP, respectively).

The MSA-CUDA algorithm [26] implements the three phases of the ClustalW [32] multiple sequence alignment algorithm in parallel. In the first phase, MSA-CUDA executes the MM algorithm (Section 2.2) in linear space, retrieving the full alignment in GPU. In the performance evaluation, the protein sequence dataset with long sequences had 1000 sequences with average length of 858 amino acids.

CUDAlign 1.0 [27] handles the pairwise DNA sequence comparison problem for up to 32MBP $\times$ 47MBP.

Only the optimal score is computed and no alignment is produced.

Note that the sequences used in [18–26] are protein sequences, typically composed of hundreds or thousands of amino acids. This problem is slightly different from the problem of aligning huge DNA sequences in the following way. First, differently from DNA sequences, that usually have a unique punctuation for matches and mismatches, substitution matrices of size 20x20 (PAM, BLOSUM) are used to retrieve the match/mismatch punctuation for protein sequences [1]. Second, the amino acid sequences are usually much smaller than the DNA sequences, so the data structures used in the first case are often sufficiently small to fit in the faster memories of the GPU. Third, when comparing protein sequences, most of the approaches execute one comparison per thread (coarse-grained parallelism) or one comparison per block of threads (medium-grained parallelism). Therefore, many different comparisons take place in parallel, with no communication among the blocks or even among the threads. On the other hand, the pairwise comparison of huge DNA sequences demands that all threads from all the blocks execute the same comparison (fine-grained parallelism), which produces a considerable communication among the threads and the blocks.

4 Design of CUDAlign 2.1

CUDAlign 2.1 is designed to obtain the full alignment of huge DNA sequences in linear space. The main idea is to obtain some coordinates of the optimal alignment and iteratively increase the number of these coordinates until it is possible to retrieve the full alignment using reasonable memory. CUDAlign 2.1 is executed in 6 stages and it uses CUDAlign 1.0 [27] as the base algorithm, combined with other optimizations in order to obtain the full alignment.

In this section, the base algorithm derived from CUDAlign 1.0 is first described. After that, some notations are given. Finally, each stage is explained.

4.1 Base Algorithm - CUDAlign 1.0

CUDAlign 1.0 was proposed in [27] and it is able to compare sequences with more than 10 MBP in GPU.

Given sequences S_0 and S_1 with sizes m and n, respectively, the algorithm is described as follows. First, the DP matrix is divided in a grid with $\frac{n}{\alpha T} \times B$ blocks, where B is the number of CUDA blocks executed in parallel and T is the number of threads in each block. Each thread is responsible to process α rows of the matrix, so each block processes αT rows. Each diagonal of blocks is called external diagonal and each diagonal of threads, inside each block, is called internal diagonal.

Three optimizations were proposed: Cells Delegation, Phase Division and Memory Access Design. Cells delegation avoids the wavefront (Section 2.3) to be emptied and filled at each external diagonal calculation, providing full parallelism during almost all the execution,

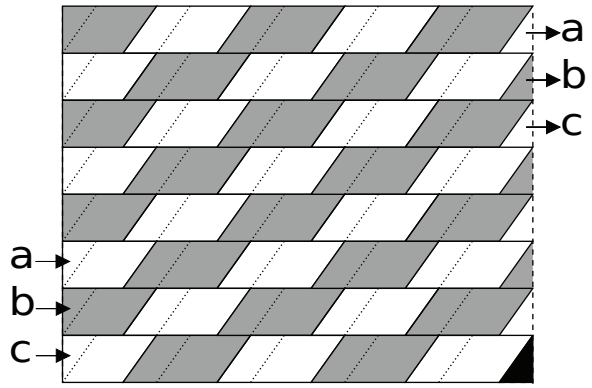


Figure 5. Cells Delegation and phase division.

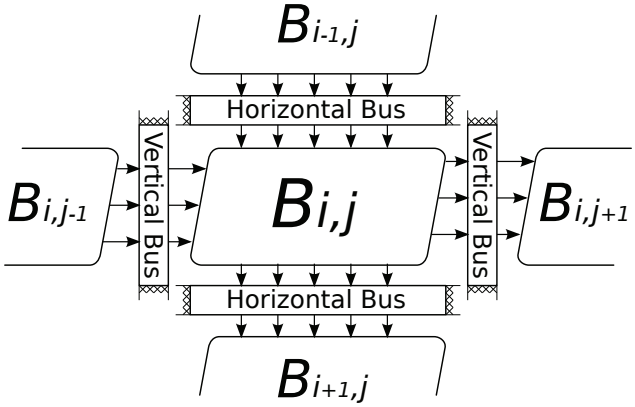


Figure 6. Horizontal Bus and Vertical Bus

except at the very beginning and very close to the end. To avoid hazards, each external diagonal is processed in two phases: short phase and long phase. The short phase calculates the first T internal diagonals of the block and the long phase calculates the remaining internal diagonals of the block. An optimized kernel is used in the long phase, achieving faster execution time [27].

Figure 5 illustrates the cells delegation and phase division. Gray blocks belong to odd external diagonals and white blocks belong to even external diagonals. Each block processes the pending cells of the left block, giving a parallelogram shape to each block. Blocks in the right-most column delegate cells to left-most blocks (a, b and c). The very last block (illustrated in black) requires an extra diagonal to process its pending cells. Each block is divided with a dotted diagonal line, representing the short phase and the long phase.

Phase division requires that the size n of sequence S_1 must be $n \geq 2BT$. This requirement is called *minimum size requirement* and if this is achieved, the blocks of the same external diagonal can concurrently access the shared data structures.

As each thread processes α rows, the memory accesses are designed to better fit in the CUDA architecture. Two main data structures are used. The *horizontal bus* is the

area of global memory used to store the last row of each block (Figure 6). This area is used to transfer values to the block below, that will be processed in the next external diagonal. The *vertical bus* is the area of global memory used to store the last column processed by each thread. This area is used to transfer values to the block on the right, that will also be processed in the next external diagonal.

A detailed description of CUDAlign 1.0 can be found in [27], including the description of the short and long phase kernels and the pseudocodes executed in GPU and CPU.

4.2 Preliminary Notations and Considerations

Let S_0 and S_1 be the sequences being aligned. The sizes of these sequences are $|S_0| = m$ and $|S_1| = n$. The $p(i, j)$ punctuation function (Equation 1) is $p(i, j) = ma$ (match) if $S_0[i] = S_1[j]$ and mi (mismatch) otherwise. A perfect match is an alignment produced when sequences S_0 and S_1 are exactly the same ($S_0 = S_1$).

A *special row* (*special column*) is a row (column) of the DP matrices that was chosen to be saved to disk.

A *crosspoint* is a coordinate of the optimal alignment that crosses some special row or special column. The crosspoints are obtained through the Myers and Miller's matching procedure (Section 2.2) with some adaptations described in the following sections.

A crosspoint is represented by a tuple $(i, j, score, type)$, where: i and j are the coordinates of the optimal alignment in the DP matrices; $score$ is the score of the alignment in position (i, j) and $type$ is the type of the alignment in position (i, j), where 0 represents match or mismatch, 1 is a gap in sequence S_0 and 2 is a gap in sequence S_1.

Two crosspoints $C_x = (i_x, j_x, score_x, type_x)$ and $C_y = (i_y, j_y, score_y, type_y)$ form a *partition* $P = (C_x, C_y)$ of the matrix. Crosspoints C_x and C_y are the edge coordinates of the partition, where C_x is the start of the partition and C_y is its end. This partition creates an alignment subproblem for subsequences $S_0[i_x + 1..i_y]$ and $S_1[j_x + 1..j_y]$.

Special care must be taken when $type_x \neq 0$ or $type_y \neq 0$. In a sequence of gaps, only the leftmost gap is counted as a gap opening. When a non-zero type happens in the beginning of the partition, the algorithm must be adjusted in such a way that it will not compute the gap opening penalty twice. If the partition has a full sequence of gaps joining both edges, the gap opening must not be computed in any of the edges of the partition.

During the execution of CUDAlign 2.1, many crosspoints are generated in order to reduce the original problem to many smaller subproblems. The list of crosspoints obtained at stage k is represented by $L_k = \{C_s, C_1, \ldots, C_{|L_k|-2}, C_e\}$.

The special crosspoints C_s and C_e are, respectively, the *start point* and the *end point* of the optimal alignment. The start point $C_s = (i_s, j_s, score_s, type_s)$ always has $type_s = 0$ and $score_s = 0$ and the end point $C_e = (i_e, j_e, score_e, type_e)$ has $type_e = 0$ and $score_e = opt$, where opt is the score of the optimal alignment. The symbol $*$ is used to indicate that the start point C_s was not been found yet ($L_k = \{*, C_1, \ldots, C_{|L_k|-2}, C_e\}$).

The partition formed by any crosspoints C_x and C_y ($x > y$) will have optimal score $S(C_x, C_y) = score_y - score_x$. Note that if $C_x = C_s$, $S(C_s, C_y) = score_y$, because $score_s = 0$. Particularly, if $C_x = C_s$ and $C_y = C_e$, then $S(C_s, C_e) = score_e = opt$.

The number of *blocks* and *threads* used in stage k is represented as B_k and T_k, respectively.

4.3 Stages Overview

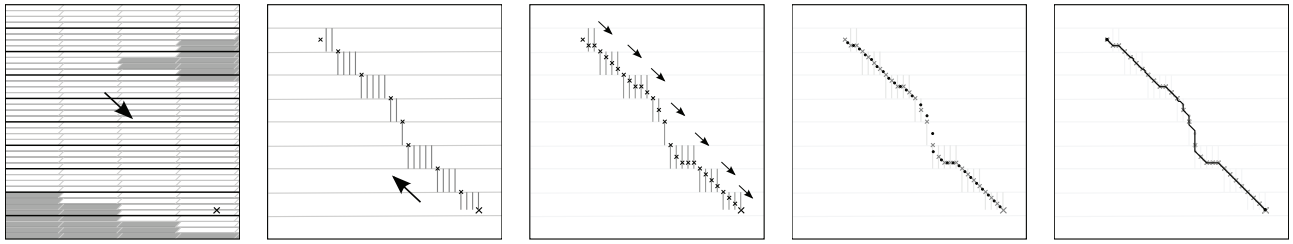
CUDAlign 2.1 is divided in 6 stages (Figure 7), where Stages 1 to 3 are run in GPU and Stages 4 to 6 are run in CPU. We opted to execute the last stages of CUDAlign 2.1 in CPU because these stages do not demand high computing power compared to the first stages.

Stages 1 to 4 increasingly obtain more crosspoints, until all partitions are smaller than a constant called *maximum partition size*. Then, Stage 5 aligns and concatenates all the partitions and Stage 6 is optionally used for visualization of the alignment.

The goal of Stage 1 is to find the score and the end point of the optimal alignment. The list of crosspoints obtained in this phase is $L_1 = \{*, C_e\}$, where C_e is the end point of the optimal alignment and the start point $*$ is unknown. Stage 1 computes the optimal score with the affine gap model (Section 2) using CUDAlign 1.0 (Section 4.1) with two main modifications: a) some special rows are saved to disk and b) blocks that cannot lead to a score higher than the current best score are pruned. Figure 7(a) shows the outputs of Stage 1. The flush interval was set to 4 blocks in this example, so after calculating 4 blocks, a special row is flushed to disk. The special rows are illustrated as a darker horizontal line in the figure. The position of the optimal score is illustrated as a cross and pruned blocks are marked in gray.

Given the outputs of Stage 1, Stage 2 executes a semi-global alignment in the reverse direction starting from the end point C_e of the alignment. The goal of Stage 2 is to find all the crosspoints ($L_2 = \{C_s, C_1, \ldots, C_{|L_2|-2}, C_e\}$) over the special rows intercepting the optimal alignment, including the start point C_s of the alignment. Stage 2 is run in GPU and it is very similar to Stage 1. Figure 7(b) shows the outputs of Stage 2. The computation starts at the optimal score position (bottom-most cross). Then, in the reverse direction, it finds the crosspoint in the next special row. During the computation, some special columns (vertical lines) are also saved to disk. When the start point C_s of the alignment is found (top-most cross), Stage 2 is completed.

Stage 3 is almost the same as Stage 2, where the main difference is that Stage 3 has partitions with defined start and end points, unlike Stage 2, that only has the end point of the alignment and the start point is unknown.



(a) Stage 1 finds the optimal score and its position. Special rows are stored on disk and some blocks are pruned (gray blocks).

(b) Stage 2 finds crosspoints between optimal alignment and special rows. Special columns are flushed.

(c) Stage 3 finds more crosspoints over special rows stored from previous stages.

(d) Stage 4 executes Myers and Miller's algorithm between each successive crosspoints.

(e) Stage 5 obtains the complete alignment between each successive crosspoints.

Figure 7. General overview of the CUDAlign 2.1 execution

The goal of Stage 3 is then to obtain more crosspoints inside each partition. Figure 7(c) shows the outputs of Stage 3.

Stage 4 executes in CPU the MM Algorithm (Section 2.2) between each successive pair of crosspoints found on Stage 3, using multiple threads. The goal of this stage is to increase the number of crosspoints until the distance between any successive pair of crosspoints is smaller or equal than a given limit, called *maximum partition size*. Figure 7(d) shows more crosspoints generated inside each partition during Stage 4.

Stage 5 aligns in CPU each partition formed by the crosspoints found in Stage 4. Then it concatenates all the results, giving as output the full optimal alignment, as can be seen in Figure 7(e).

Stage 6 is an optional stage used only for visualization of the alignment.

In the following sections, further details will be given to each stage.

4.4 Stage 1 - Obtain the optimal score

In stage 1, the algorithm described in Section 4.1 is executed with two modifications: a) some special rows are saved to disk and b) a new optimization called *block pruning* is used.

4.4.1 Saving Special Rows to Disk

The goal of saving special rows to disk is to accelerate the execution of the further stages, saving computing time that would otherwise be spent calculating the original MM algorithm. This approach is similar to FastLSA (Section 3.1), but we save rows to disk instead of memory because there is usually more available space. Also, saving rows to disk allows the recovery of the computation in case of interruption (power failure or user intervention, among others).

The special rows that are flushed to disk are taken from the horizontal bus (Section 4.1) at a certain interval of blocks. Since the horizontal bus contains the cells of the last row of a block, only rows that are multiple of the block height (αT) are candidates be considered a special row.

The area reserved for storing the special rows in disk is called *special rows area* (SRA) and it has a limited constant space $|SRA|$. Each cell of a special row contains two 4-byte values, representing the elements of matrices H and F (Section 2). So, each special row has $8n$ bytes and the maximum number of special rows that can be saved is $\frac{|SRA|}{8n}$. The *flush interval* is the number of blocks between each special row saved, and this must be at least $\lceil \frac{8mn}{\alpha T.|SRA|} \rceil$. It must be ensured that the size of SRA is at least the size of one special row.

4.4.2 Block Pruning

The goal of the block pruning optimization is to eliminate the calculation of blocks of cells that surely do not belong to the optimal alignment. These blocks have such a small score that it is not mathematically possible to lead to a score higher than a score that has already been produced.

The optimization described in [33] uses a similar idea, but it is only applied to the second phase of the SW algorithm, where the optimal score is already known. In CUDAlign 2.1, the pruning optimization is applied already in the first stage, where the optimal score is not known yet. Another difference is that, instead of analyzing the possibility of pruning each cell, the analysis in CUDAlign 2.1 is made per-block and using wavefront.

In the following paragraphs we first give some definitions. Then, the pruning procedure is formally defined. Finally, we present some theorems that identify the areas where the pruning optimization can be effective.

Definitions: Consider the definitions of S_0, S_1 and $p(i,j)$ given in the first paragraph of Section 4.2. For a given cell (i,j) of the DP matrix, the *i-distance* (Δ_i) is the distance from row i to the last row of the matrix ($\Delta_i = m - i$) and the *j-distance* (Δ_j) is the distance from column j to the last column of the matrix ($\Delta_j = n - j$). A Δ_i-cell is a cell (i,j) where $\Delta_i < \Delta_j$ and a Δ_j-cell is a cell (i,j) where $\Delta_j < \Delta_i$.

Suppose that cell (i,j) of the DP matrix has score $H(i,j)$. The maximum score of any alignment that passes through cell (i,j) is $H_{max}(i,j) = H(i,j) + min(\Delta_i, \Delta_j).ma$, where ma is the punctuation

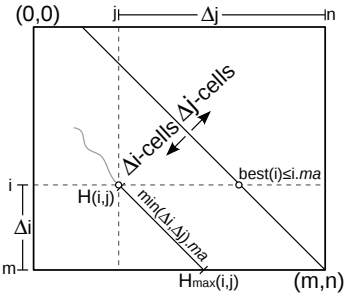


Figure 8. Geometrical representation of some definitions.

for matches. This situation happens in a perfect substring match $S_0[i..(i+min(\Delta_i,\Delta_j))] = S_1[j..(j+min(\Delta_i,\Delta_j))]$.

The maximum possible score that can be found in row i happens in the perfect substring match $S_0[1..i] = S_1[1..i]$, and the maximum possible score in this row i is $best(i) = i.ma$. A *prunable cell* is a cell that cannot generate an alignment with score greater than the maximum score found until then, i.e. cell (i,j) will be prunable if $H_{max}(i,j) \leq best(i)$. Therefore, all prunable cells could be ignored with no impact on the final optimal score. Figure 8 shows some of these definitions in geometrical representation.

Pruning Procedure: In order to reduce the number of processed cells, we must identify the prunable cells before their computation. This can be done with the observation that if cells $(i-1,j)$, $(i,j-1)$ and $(i-1,j-1)$ are all prunable, then the cell (i,j) will also be prunable.

So, let the *non-prunable window* be the interval $[k_s..k_e]$ of columns that must necessarily be computed. Initially, let $[k_s..k_e] = [0..n]$. For each row i, the algorithm computes cells $(i, j \in [k_s..k_e])$ and then it updates the *non-prunable window* to values $[k'_s..k'_e]$, where values k'_s and k'_e are, respectively, the first and the last non-prunable cell of the row.

Assuming that all cells $(i-1, j \in [0..k_s])$ are prunable, then all cells $(i, j \in [0..k_s])$ will also be prunable. So we can conclude that $k'_s \geq k_s$. Similarly, if all cells $(i-1, j \in [k_e..n])$ are prunable and cell $(i, x \in [k_e..n])$ is prunable, then all cells $(i, j \in [x+1..m])$ will also be prunable. If cell (i, k_e+1) is prunable, then the new k'_e is simply the last non-prunable cell in $(i, j \in [k_s..k_e])$, with $k'_e \leq k_e$. Otherwise, we must compute the remaining cells of row i until we find a prunable cell $(i, x \in [k_e+2..n])$, then the new k'_e is $x-1$. The cells $(i, j \in [0..k'_s])$ are called *left-prunable cells* and the cells in the interval $(i, j \in [k'_e+1..n])$ are called *right-prunable cells*.

The *block pruning* optimization is the pruning procedure applied to blocks of cells. It is faster because the pruning verification is done only once for each block, and not for every cell. Nevertheless, if the blocks are too big, the pruning procedure will be so coarse-grained that it will be difficult to find many *prunable blocks*.

In order to implement the block pruning optimizations in CUDAlign 2.1, the kernels verify if the block is inside the non-prunable window $[k_s..k_e]$. If it is outside the window, the block is not calculated and its associated areas of the vertical and horizontal buses (Section 4.1) are set to zeroes in order to prevent data inconsistency. After each external diagonal calculation, the non-prunable window is updated considering the maximum score that each block can achieve.

Theorems: Without loss of generality, the following theorems assume that the computation of the DP matrix is made row by row and that each block contains one cell.

Theorem 4.1: A Δ_i-cell (i,j) can only be prunable if $i \geq m/2$.

Proof: A Δ_i-cell is a cell (i,j) where $\Delta_i < \Delta_j$ and the maximum possible score in row i is $best(i) = i.ma$. So, we have inequalities (5):

$$\begin{aligned} H_{max}(i,j) &\leq best(i) \leq i.ma \\ H(i,j) + min(\Delta_i,\Delta_j).ma &\leq i.ma \\ H(i,j) + \Delta_i.ma &\leq i.ma \\ H(i,j) + (m-i).ma &\leq i.ma \\ H(i,j) + m.ma &< 2.i.ma \\ \frac{H(i,j)}{2.ma} + \frac{m}{2} &\leq i \\ \frac{m}{2} &\leq i \text{ , since } H(i,j) \geq 0 \end{aligned} \tag{5}$$

□

Theorem 4.2: A Δ_j-cell (i,j) can only be prunable if $i \geq n-j$.

Proof: A Δ_j-cell is a cell (i,j) where $\Delta_j < \Delta_i$ and the maximum possible score in row i is $best(i) = i.ma$. So, we have inequalities (6):

$$\begin{aligned} H_{max}(i,j) &\leq best(i) \leq i.ma \\ H(i,j) + min(\Delta_i,\Delta_j).ma &\leq i.ma \\ H(i,j) + \Delta_j.ma &\leq i.ma \\ H(i,j) + (n-j).ma &\leq i.ma \\ \frac{H(i,j)}{ma} + (n-j) &\leq i \\ (n-j) &\leq i \text{ , since } H(i,j) \geq 0 \end{aligned} \tag{6}$$

□

Corollary 4.3: A cell (i,j) cannot be prunable if $i \geq n-j$ and $i \geq m/2$.

Proof: This is the negation of Theorems 4.1 and 4.2.

□

Figure 9 shows the areas where we can find prunable cells. Also, the area where we cannot find any prunable cell (Corollary 4.3) is presented in white. It contains approximately $\frac{mn}{2} - \frac{m^2}{8}$ cells when $m \leq 2n$, or $\frac{n^2}{2}$ cells when $m > 2n$. This gives a superior limit for the prunable area of $\frac{mn}{2} + \frac{m^2}{8}$ cells when $m \leq 2n$, or $mn - \frac{n^2}{2}$ cells when $m > 2n$. Dividing this by the total number of cells (mn) we have a superior limit for the pruning ratio of $\frac{1}{2} + \frac{1}{8}\frac{m}{n}$ when $m \leq 2n$, or $1 - \frac{1}{2}\frac{n}{m}$ when $m > 2n$. So, for example, if there are two sequences with the same size ($m = n$), then the superior pruning ratio is 62.5%. In the worst case, block pruning will not discard any cell. Therefore, Stage 1 with the block pruning optimization also runs in $O(mn)$ time.

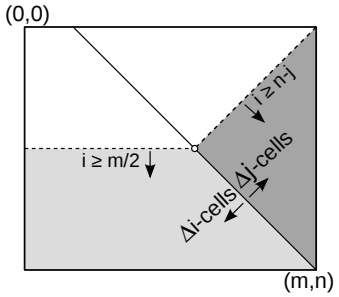


Figure 9. Prunable area. Gray areas represent where a prunable Δ_i-cell (light gray area) and a prunable Δ_j-cell (dark gray area) can be found. The white area cannot contain any prunable cell.

4.5 Stage 2 - Partial Traceback

Starting from the end point of the optimal alignment found in Stage 1, Stage 2 executes a semi-global alignment in GPU, in reverse direction. The goal of this stage is to find all the crosspoints over the special rows intercepting the optimal alignment, including the start point of the alignment (Figure 7(b)).

Stage 2 only computes the cells near the optimal alignment, without using the block pruning optimization. On the other hand, the cells delegation and the Phase Division are executed as in Stage 1 (Section 4.1). The constants B_2 and T_2 may be different from the constants B_1 and T_1 used in Stage 1.

The B_2 and T_2 values are chosen precisely in order to maintain the minimum size requirement [27]. Because Stage 2 computes a subproblem of the full alignment problem, the size considered for the minimum size requirement in Stage 2 is not the size of the sequence S_1, but the distance between two special rows.

In Stage 2, two optimizations are used: a) goal-based matching procedure and b) orthogonal execution. The idea of these optimizations is to avoid the calculation of unnecessary cells of the matrix, stopping the calculation as soon as a new crosspoint is found. Further details of these optimizations can be found in [28].

4.6 Stage 3 - Splitting Partitions

The goal of this stage is to obtain more crosspoints. Stage 3 is almost the same as Stage 2. The main difference is that Stage 3 has partitions with defined start and end points. Whenever the last special column of each partition is intercepted, no more computation needs to be done in that partition, because the next crosspoint is the already known end point of that partition. Note that the order of execution of the partitions is irrelevant, so they can be processed in parallel.

The thread execution of Stage 3 is made in the orthogonal direction of the Stage 2 (Figure 7(c)), so that threads execute horizontally, as in Stage 1.

The number of blocks and the number of threads executing each block are represented by the constants B_3 and T_3 respectively. Like stages 1 and 2, the width of the partition cannot be smaller than the minimum size requirement ($2B_3T_3$). Because the partitions in Stage 3 are smaller than the ones in Stages 1 and 2, the minimum size requirement can become a strong limitation. So, if this requirement cannot be fulfilled, B_3 may be reduced. When B_3 is reduced, the performance also decreases, so all the B_k and T_k constants must be carefully chosen in order to achieve good execution times.

4.7 Stage 4 - MM with balanced splitting and orthogonal execution

This stage executes in CPU the MM Algorithm (Section 2.2) between each successive pair of crosspoints, using multiple threads. The goal of this stage is to increase the number of crosspoints until the distance between any successive pair of crosspoints is smaller or equal than a given limit, called *maximum partition size*. Each iteration of Stage 4 may increase up to twice the number of crosspoints, so many iterations may be necessary until the sizes of the partitions are reduced to less than the maximum partition size. As in Stage 3, the order of execution of the partitions is irrelevant, so they can be processed in parallel. Figure 7(d) shows one iteration of Stage 4.

An optimization, called *Balanced Splitting*, was proposed in order to reduce the number of iterations in Stage 4. The main idea is to halve the largest dimension of the partition between the crosspoints, instead of halve always at the middle row. This ensures that the largest dimension is reduced at each iteration, preventing narrow partitions to keep their disproportional dimensions during many iterations [28].

4.8 Stage 5 - Obtaining the Full Alignment

This stage aligns each partition in CPU and concatenates all the results, giving as output the full optimal alignment, as can be seen in Figure 7(e).

After executing Stage 4, it is guaranteed that the sizes of the partitions are smaller or equal than the maximum partition size. If the maximum partition size is small enough (e.g. 16×16), the execution time of each partition will be very fast. Because the maximum partition size is constant, the memory complexity of aligning each partition is also constant and the full alignment can be obtained in linear space.

In order to reduce the size of the output that represents the full alignment, the following data are output from Stage 5: the start position (i_0, j_0) and end position (i_1, j_1); the optimal score; two lists, GAP_1 and GAP_2, of tuples $(i_{gap}, j_{gap}, length)$ where i_{gap} and j_{gap} are the position of a gap open, $length$ is the number of successive gaps. There must be two lists of tuples because GAP_1 is for type 1 gaps and the GAP_2 for type 2 gaps. So, Stage 5 stores to disk a binary file representing these data, without storing the characters of the sequences. The size of this binary representation is much smaller than the textual representation, reducing the I/O needed to store the alignment on disk or to transfer it over the network.

The full alignment is reconstructed from this binary file in Stage 6.

4.9 Stage 6 - Visualization

This is an optional stage for visualization of the binary representation of the alignment. Given the input sequences S_0 and S_1, the start position (i_0, j_0) and end position (i_1, j_1) and the lists GAP_1 and GAP_2, the reconstruction of the alignment is made by joining the gaps. Starting with $(i, j) = (i_0, j_0)$, the nearest gap is taken from GAP_1 or GAP_2 and the next (i, j) will be the end of the selected gaps sequence. This is done iteratively until the end position (i_1, j_1) is reached.

This procedure allows to generate textual or graphical representations. As the binary format is very compact when compared to the textual representation of the alignment, Stage 6 may only be executed when a detailed analysis of the alignment is necessary.

5 Experimental Results

CUDAlign 2.1 was implemented in CUDA 4.0, C++ and pthreads and tested in an NVIDIA GeForce GTX 560 Ti. This board has 1GB of memory, 8 multiprocessors and 384 cores. It was connected to an Intel Pentium Dual-Core 3GHz, 3GB RAM, 250GB HD, running Ubuntu 10.04, Linux kernel 2.6.32. The Smith-Waterman parameters were set to: match: $+1$; mismatch -3; first gap: -5; extension gap: -2.

Our tests 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 2. The sizes of these real sequences range from 162 KBP to 59 MBP and the sequences are the same used in [28] with the addition of the chromosomes Y.

5.1 CUDA Parameters

The performance of CUDAlign 2.1 depends on a good choice of the CUDA parameters B (number of blocks) and T (number of threads). In order to determine these parameters for the GTX 560 Ti board and for Stage 1, we executed CUDAlign 2.1 for the $1044K \times 1073K$ comparison (Table 2) varying the parameters B and T. In this execution, pruning blocks and flushing rows were disabled. Firstly, we varied B from 128 to 512, with $T = 128$ (Figure 10). The lower runtimes occur for every B that is multiple of 8. This is explained by the fact that the GTX 560 Ti has 8 multiprocessors, and the time needed to compute each block is almost the same. So, when the number of blocks is a multiple of 8, the multiprocessors do not become idle when they reach the end of an external diagonal, resulting in a better performance. For the further analysis, we will only consider a number of blocks B that is multiple of 8.

Then, CUDAlign 2.1 was tested with $T = 128$, $T = 192$ and $T = 256$. Figure 11 shows the average runtime of the $1044K \times 1073K$ comparison for each value of T and

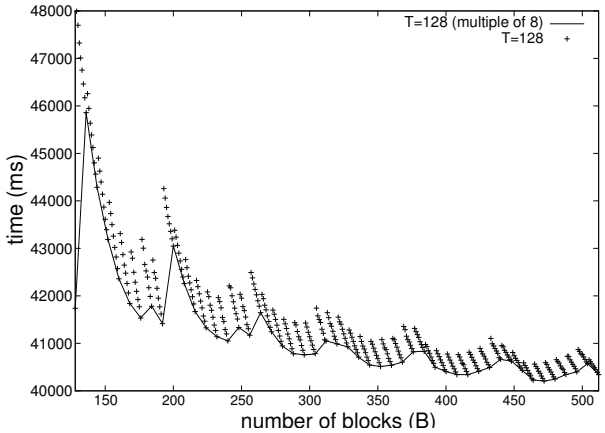


Figure 10. Runtimes for the $1044K \times 1073K$ comparison varying the number of blocks (B) with $T = 128$.

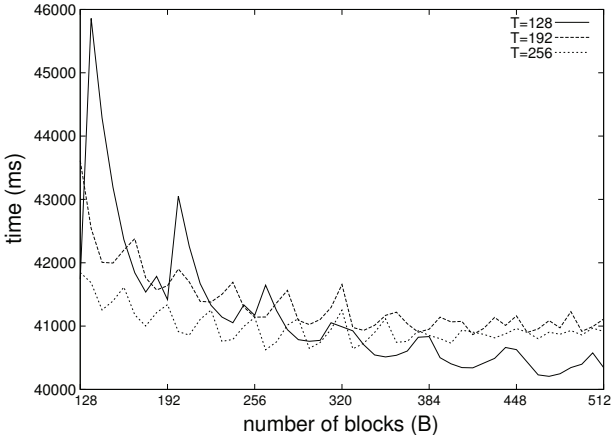


Figure 11. Runtimes for the $1044K \times 1073K$ comparison varying the number of blocks (B) and the number of threads (T).

varying B from 128 to 512, in steps of 8. The tests were run 5 times and the maximum standard deviation was 6.6 ms, showing very precise and repeatable results. The best average runtime was 40.21 seconds, obtained when $T = 128$ and $B = 472$. When $B = 512$ and $T = 128$ the runtime was 40.35 seconds (only 0.35% higher than the best runtime). Thus we chose this number of blocks as it is the nearest power of 2 from 472. Also, choosing a higher B may produce a slightly better pruning optimization, as it permits a finer grained check in block pruning.

Additionally, the same tests were run for the $3147K \times 3283K$ comparison, presenting a very similar pattern.

Since stages 2 and 3 usually spend much less time than stage 1, the detailed analysis of the choice of B and T for these stages will not be presented in this paper.

Therefore, the execution configurations used for GTX 560 Ti were $B_1 = 512$, $B_2 = B_3 = 32$ and $T_1 = T_2 = T_3 = 128$, with $\alpha = 4$. The number of blocks may be reduced during runtime in order to satisfy the minimum size requirement (Section 4.1) in each stage.

Comparison	Real size	Accession Number	Name
162K×172K	162,114 BP	NC_000898.1	*Human herpesvirus* 6B
	171,823 BP	NC_007605.1	*Human herpesvirus* 4
543K×536K	542,868 BP	NC_003064.2	*Agrobacterium tumefaciens*
	536,165 BP	NC_000914.1	*Rhizobium sp.*
1044K×1073K	1,044,459 BP	CP000051.1	*Chlamydia trachomatis*
	1,072,950 BP	AE002160.2	*Chlamydia muridarum*
3147K×3283K	3,147,090 BP	BA000035.2	*Corynebacterium efficiens*
	3,282,708 BP	BX927147.1	*Corynebacterium glutamicum*
5227K×5229K	5,227,293 BP	AE016879.1	*Bacillus anthracis* str. Ames
	5,228,663 BP	AE017225.1	*Bacillus anthracis* str. Sterne
7146K×5227K	7,145,576 BP	NC_005027.1	*Rhodopirellula baltica* SH 1
	5,227,293 BP	NC_003997.3	*Bacillus anthracis* str. Ames
23012K×24544K	23,011,544 BP	NT_033779.4	*Drosophila melanog.* chromosome 2L
	24,543,557 BP	NT_037436.3	*Drosophila melanog.* chromosome 3L
59374K×23953K	59,373,566 BP	NC_000024.9	*Homo sapiens* chromosome Y
	23,952,694 BP	NC_006492.2	*Pan troglodytes* chromosome Y
32799K×46944K	32,799,110 BP	BA000046.3	*Pan troglodytes DNA*, chromosome 22
	46,944,323 BP	NC_000021.7	*Homo sapiens* chromosome 21

Table 2
Real sequences details. Sizes range from 162 KBP to 59 MBP.

5.2 Execution Times and GCUPS

An usual performance metric for Smith-Waterman implementations is the number of billions of cell updates per second (GCUPS), that 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 and t is the runtime in seconds.

Table 3 presents the execution times and GCUPS for Stage 1 with and without the block pruning optimization (without flushing rows to disk). The GCUPS$_P$ for the block pruning represents the performance considering the formula $\frac{mn-p}{t \times 10^9}$, where p is the number of pruned cells. The GCUPS metric is the traditional GCUPS, calculated with the formula $\frac{mn}{t \times 10^9}$, that assumes that all cells were processed. The percentage of left-prunable and right-prunable blocks increases when the similarity score is higher, as can be seen in the "%*Pruned*" columns.

Table 3 shows that block pruning can lead to an impressive reduction in execution time when the sequences have a high similarity. For instance, in the $32799K \times 46944K$ comparison (optimal score = 27,206,434), there were 48.1% of pruned blocks, resulting in a runtime gain of 45.9%. In the $5227K \times 5229K$ comparison, which contains the most similar result from the test set (optimal score = 5,220,960), there were 53.7% of pruned blocks, resulting in a runtime gain of 51.2%. The optimal scores for all comparisons are presented and discussed in Section 5.3.

Note that the $59374K \times 23953K$ comparison has a higher execution time than the $32799K \times 46944K$ comparison, even though the first comparison produces a DP matrix with 8% less cells than the second one. This is explained by the fact that the $59374K \times 23953K$ comparison produces a not very expressive alignment, and consequently a smaller runtime gain (5.6%) with the block pruning optimization (Table 3).

Table 4 shows the execution time for all the comparisons and all the stages on GeForce GTX 560 Ti, executed with all the proposed optimizations. The size of the special rows area (SRA) was chosen empirically for each comparison. Comparing Table 4 with Table 3, it can be observed that for each Gigabyte in the SRA there is an overhead of approximately 3 seconds. In the tests, a low overhead (approximately 2%) for flushing rows was observed.

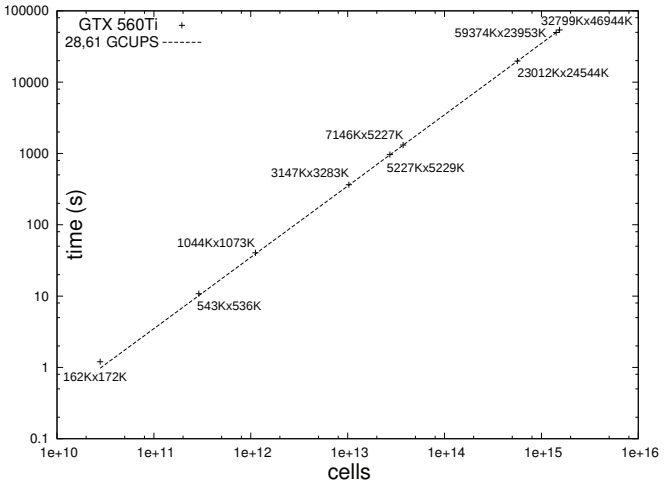


Figure 12. Runtimes (seconds) × DP matrix size (cells) in logarithm scale of Stage 1 without flushing rows and without block pruning. Results show scalability with approximately 28,61 GCUPS for several sequence sizes .

Also, note that when the length of the optimal alignment is small, the runtimes of Stages 2, 3, 4 and 5 are negligible, as can be easily seen in comparisons $162K \times 172K$, $543K \times 536K$, $7146K \times 5227K$ and $23012K \times 24544K$.

Figure 12 shows the wallclock execution times for Stage 1 of CUDAlign 2.1 vs. DP matrix size (number of cells calculated) in logarithmic scale. The block pruning optimization was disabled in this test. The results shows the scalability of our approach with almost constant GCUPS for megabase sequences. Note that, for sequences longer than 3 MBP, CUDAlign 2.1 is able

Comparison	Original			Block Pruning					% Pruned		
	Time(s)	Cells	GCUPS	Time(s)	Cells	$GCUPS_P$	GCUPS	Gain	Left	Right	Total
162K×172K	1.2	2.79E+10	23.28	1.2	2.78E+10	23.37	23.45	0.7%	0.3%	0.0%	0.3%
543K×536K	10.8	2.91E+11	26.97	10.8	2.91E+11	26.98	27.00	0.1%	<0.1%	0.0%	<0.1%
1044K×1073K	40.3	1.12E+12	27.81	36.2	9.97E+11	27.54	30.95	10.1%	7.4%	3.6%	11.0%
3147K×3283K	363.6	1.03E+13	28.41	363.2	1.03E+13	28.40	28.44	0.1%	0.1%	0.0%	0.1%
5227K×5229K	962.4	2.73E+13	28.40	469.5	1.27E+13	26.95	58.21	51.2%	32.9%	20.8%	53.7%
7146K×5227K	1309	3.74E+13	28.53	1309	3.73E+13	28.53	28.54	<0.1%	<0.1%	0.0%	<0.1%
23012K×24544K	19701	5.65E+14	28.67	19694	5.65E+14	28.67	28.68	<0.1%	<0.1%	0.0%	<0.1%
59374K×23953K	49634	1.42E+15	28.65	46869	1.34E+15	28.51	30.34	5.6%	1.2%	4.8%	6.0%
32799K×46944K	53869	1.54E+15	28.58	29133	7.99E+14	27.44	52.85	45.9%	35.5%	12.6%	48.1%

Table 3

Runtimes (in seconds) of Stage 1 of CUDAlign 2.1 without flushing rows. The times are presented with and without block pruning. The original GCUPS considers the number of cells of all the DP matrix and the pruned GCUPS ($GCUPS_P$) considers only the non-pruned cells.

Comparison	SRA	Stages					Total	
		1	2	3	4	5+6	Time	GCUPS
162K×172K	5M	1.2	<0.1	<0.1	<0.1	<0.1	2.1	13.26
543K×536K	50M	11.0	<0.1	<0.1	<0.1	<0.1	11.8	24.67
1044K×1073K	250M	37.1	1.9	0.6	2.8	0.2	43.4	25.82
3147K×3283K	1G	366	<0.1	<0.1	0.2	<0.1	367	28.15
5227K×5229K	3G	480	38.6	7.3	26.1	4.7	558	48.98
7146K×5227K	3G	1320	<0.1	<0.1	<0.1	<0.1	1321	28.28
23012K×24544K	10G	19755	0.2	<0.1	0.4	<0.1	19757	28.59
59374K×23953K	50G	47039	52.9	5.5	11.0	0.9	47123	30.18
32799K×46944K	50G	29333	592	175	251	7.5	30369	50.70

Table 4

Runtimes (in seconds) of each stage of CUDAlign 2.1 on GTX 560 Ti varying the comparison size. The total time includes all the stages and the I/O of reading the sequences.

Size	Z-align (64 cores)	CUDAlign 2.1 (GTX 560 Ti)	
	Time(s)	Time(s)	Speedup
150KBP	22.6	2.1	10.72
500KBP	176	11.8	14.96
1MBP	1044	43.4	24.08
3MBP	8765	367	23.86
5MBP	23235	558	41.64
23MBP	400863	19757	20.29

Table 5

CUDAlign speedups compared to the Z-align cluster solution.

to achieve a performance higher than 28 GCUPS when running at the GTX 560 Ti board.

The execution times obtained by CUDAlign 2.1 were compared to the Z-align cluster solution [34], which is, as far as we know, the only CPU cluster solution that is able to produce optimal pairwise alignments of huge sequences (sizes greater than 10 MBP each). Table 5 presents the execution times for Z-align running on a 64-core cluster (Table 3 in [34]) and CUDAlign 2.1 running on a GTX 560 Ti GPU. Z-Align was not able to align the human and chimpanzee chromosomes and the speedup comparison was made only between the organisms aligned by both Z-align and CUDAlign 2.1.

CUDAlign 2.1 obtained speedups from 10.72x (150KBP) up to 41.64x (5MBP). The maximum speedup of 41.64x was partially due the block pruning optimization, as we can see in Table 3. This result clearly show the great advantage of using CUDAlign 2.1 with all the proposed optimizations.

5.3 Alignment Results

For all pairs of sequences shown in Table 2, Table 6 lists data about the optimal alignments found, including the number of matches, mismatches and gaps, the optimal score and the total length of the alignment.

For the $32799K \times 46944K$ comparison, the number of matches in the optimal alignment was 96.6% of the size of the chimpanzee chromosome 22 (32,799,110 BP), showing an impressive similarity between both chromosomes. In [35], this comparison was done with a heuristic method. As far as we know, there is no other implementation, either in clusters, CEllBEs, FPGAs or GPUs, that aligned those chromosomes with Smith-Waterman.

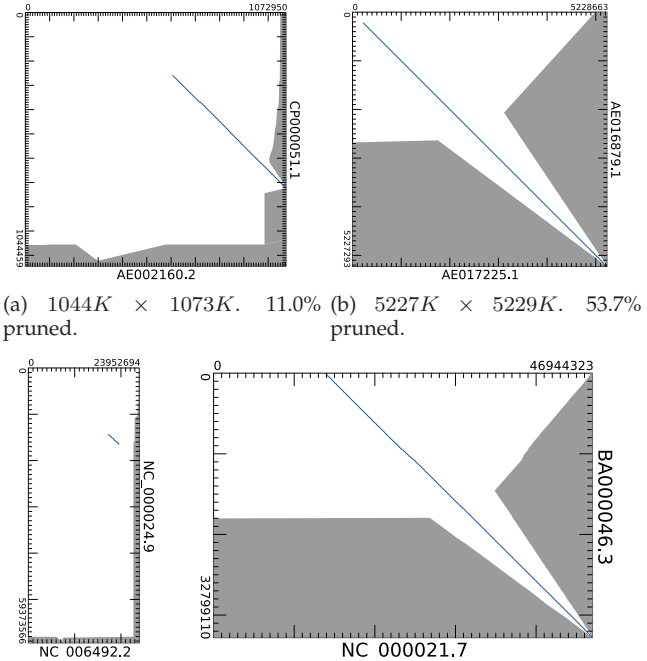
Figure 13 shows a plotting of the most similar alignments, provided as output of the visualization stage (Stage 6). The gray area represents the pruned blocks. Note some diagonal edges in the bottom of the $1044K \times 1073K$ matrix (Figure 13(a)). This is caused by the wavefront strategy used by CUDAlign 2.1, which is most visible in the pruned area of the the smaller comparisons.

The $59374K \times 23953K$ comparison (Figure 13(c)) also produces a big alignment, but not as significant as the one obtained with the $32799K \times 46944K$ comparison (Figure 13(d)). The number of matches in the optimal alignment for $59374K \times 23953K$ comparison was 8.3% of the size of the chimpanzee chromosome Y (23,952,694 BP).

The number of matches in the optimal alignment for the $5227K \times 5229K$ comparison was 99.987% of the *Bacillus anthracis* str. Ames (5,227,293 BP), showing that

Comparison	Match	Mismatch	Gaps		Score	Length
			First	Ext.		
162K×172K	18	0	0	0	18	18
543K×536K	81	11	0	0	48	92
1044K×1073K	376200	81719	4952	8965	88353	471888
3147K×3283K	12089	1568	451	452	4226	14554
5227K×5229K	5226597	165	94	2336	5220960	5229192
7146K×5227K	475	72	17	1	172	565
23012K×24544K	9096	5	2	4	9063	9107
59374K×23953K	1987064	71479	7156	214653	1307541	2280389
32799K×46944K	31696109	516090	66313	1304920	27206434	33583457

Table 6
Information of the optimal local alignments found.



(a) $1044K \times 1073K$. 11.0% pruned. (b) $5227K \times 5229K$. 53.7% pruned.

(c) $59374K \times 23953K$. 6.0% pruned. (d) $32799K \times 46944K$. 48.1% pruned.

Figure 13. Plot of some alignments with pruned blocks in gray.

both Ames and Sterne strains of *Bacillus anthracis* have very similar DNA sequences (Figure 13(b)).

6 Conclusion and Future Work

In this paper we presented CUDAlign 2.1, which is a version of Smith-Waterman (SW) able to fully align two huge sequences in linear space. The algorithm is divided in 6 stages. The first stage processes the full DP matrix as in [27], but some special rows are saved in an area called Special Rows Area (SRA) and some blocks are pruned. The second stage processes the DP matrix in the reverse direction starting from the endpoint of the optimal alignment and also saves special columns in disk. Using an optimization called orthogonal execution, the area calculated in Stage 2 is reduced. Stage 3 increases the number of crosspoints with an execution similar to Stage 2 but in the forward direction.

Stage 4 uses the MM algorithm with orthogonal execution to decrease the size of the partitions. As soon as all the partitions are smaller than the maximum partition size, Stage 5 finds the alignment of each partition and concatenates the results in the full alignment. Stage 6 is optional and it presents the full alignment in textual or graphical representation.

The experimental results were obtained with a GTX 560 Ti GPU with 1GB of memory. CUDAlign 2.1 was able to obtain the alignment of two whole chromosomes of size 47 MBP and 33 MBP in feasible time and using reasonable memory space. Using an SRA of 50GB, the full alignment of these genomic sequences was obtained in 8 hours and 26 minutes, where 99.1% of this time was spent on the GPU stages. CUDAlign 2.1 obtained maximum speedups of 41.64x when compared to the Z-align cluster solution with 64 cores.

As future work, we intend to further optimize some stages of the algorithm. In Stage 3, the parallelism is currently exploited intensively inside each partition, but in future works many partitions may also be processed in parallel, reducing the execution time of Stage 4. Also, we intend to implement the block pruning optimization on Stages 2 and 3. We will also extend the tests to even more powerful GPUs, including systems with dual cards and from other vendors. Finally, we will investigate the possibility of solving the multiple sequence alignment problem with the optimizations proposed in this paper.

References

[1] D. W. Mount. *Bioinformatics: sequence and genome analysis*. Cold Spring Harbor Laboratory Press, 2004.

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

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

[4] D. S. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Commun. ACM*, 18(6):341–343, 1975.

[5] E. W. Myers and W. Miller. Optimal alignments in linear space. *Computer Applications in the Biosciences*, 4(1):11–17, 1988.

[6] R.S. Harris. *Improved pairwise alignment of genomic DNA. Ph.D. Thesis.* The Pennsylvania State University, 2007.

[7] S. Kurtz, A. Phillippy, A.L. Delcher, M. Smoot, M. Shumway, C. Antonescu, and S.L. Salzberg. Versatile

and open software for comparing large genomes. *Genome biology*, 5(2):R12, 2004.

[8] S. Aluru, N. Futamura, and K. Mehrotra. Parallel biological sequence comparison using prefix computations. *J. Parallel Distrib. Comput.*, 63(3):264–272, 2003.

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

[10] R. B. Batista, A. Boukerche, and A. C. M. A. de Melo. A parallel strategy for biological sequence alignment in restricted memory space. *J. Parallel Distrib. Comput.*, 68(4):548–561, 2008.

[11] C. Chen and B. Schmidt. Computing Large-Scale Alignments on a Multi-Cluster. *Cluster Computing, IEEE International Conference on*, page 38, 2003.

[12] P. Zhang, G. Tan, and G. R. Gao. Implementation of the Smith-Waterman algorithm on a reconfigurable supercomputing platform. In *HPRCTA '07*, pages 39–48, 2007.

[13] X. Liu L. Xu P. Zhang N. Sun X. Jiang. "A Reconfigurable Accelerator for Smith-Waterman Algorithm". *"IEEE Transactions on Circuits and Systems II"*, 54(12):1077–1081, December 2007.

[14] A. Boukerche, J. M. Correa, A. C. M. A. de Melo, R. P. Jacobi, and A. F. Rocha. Reconfigurable Architecture for Biological Sequence Comparison in Reduced Memory Space. In *IPDPS*, pages 1–8, 2007.

[15] C. Chen and B. Schmidt. An adaptive grid implementation of DNA sequence alignment. *Future Generation Computer Systems*, 21(7):988–1003, 2005.

[16] F. Sánchez, F. Cabarcas, A. Ramirez, and M. Valero. Long DNA sequence comparison on multicore architectures. In *Euro-Par*, pages 247–259, Berlin, Heidelberg, 2010. Springer-Verlag.

[17] A. Sarje and S. Aluru. Parallel Genomic Alignments on the Cell Broadband Engine. *IEEE Trans. Parallel Distrib. Syst.*, 20(11):1600–1610, 2009.

[18] Y. Liu, W. Huang, J. Johnson, and S. Vaidya. GPU Accelerated Smith-Waterman. In *ICCS 2006*, volume 3994 of *LNCS*, pages 188–195. Springer, 2006.

[19] W. Liu, B. Schmidt, G. Voss, A. Schroder, and W. Muller-Wittig. Bio-sequence database scanning on a GPU. *IPDPS*, 2006.

[20] S. Manavski and G. Valle. CUDA compatible GPU cards as efficient hardware accelerators for Smith-Waterman sequence alignment. *BMC Bioinformatics*, 9(Suppl 2), 2008.

[21] Y. Liu, D. Maskell, and B. Schmidt. CUDASW++: optimizing Smith-Waterman sequence database searches for CUDA-enabled graphics processing units. *BMC Research Notes*, 2(1):73, 2009.

[22] L. Ligowski and W. Rudnicki. An Efficient Implementation of Smith-Waterman Algorithm on GPU using CUDA, for Massively Parallel Scanning of Sequence Databases. In *IPDPS-HiCOMB*, 2009.

[23] Y. Liu, B. Schmidt, and D. Maskell. CUDASW++2.0: enhanced Smith-Waterman protein database search on CUDA-enabled GPUs based on SIMT and virtualized SIMD abstractions. *BMC Research Notes*, 3(1):93, 2010.

[24] A. Khajeh-Saeed, S. Poole, and J. Blair Perot. Acceleration of the Smith-Waterman algorithm using single and multiple graphics processors. *J. Comput. Phys.*, 229(11):4247–4258, 2010.

[25] D. Hains, Z. Cashero, M. Ottenberg, W. Bohm, and S. Rajopadhye. Improving CUDASW++, a Parallelization of Smith-Waterman for CUDA Enabled Devices. *Parallel and Distributed Processing Workshops and PhD Forum, 2011 IEEE International Symposium on*, 0:490–501, 2011.

[26] Y. Liu, B. Schmidt, and D. L. Maskell. MSA-CUDA: Multiple Sequence Alignment on Graphics Processing Units with CUDA. In *ASAP*, pages 121–128. IEEE, 2009.

[27] E. F. de O. Sandes and A. C. M. A. de Melo. CUDAlign: using GPU to accelerate the comparison of megabase genomic sequences. In *PPOPP*, pages 137–146. ACM, 2010.

[28] E. F. de O. Sandes and A. C. M. A. de Melo. Smith-Waterman Alignment of Huge Sequences with GPU in Linear Space. In *IPDPS*, pages 1199–1211, 2011.

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

[30] X. Guan and E. C. Uberbacher. A multiple divide-and-conquer(MDC) algorithm for optimal alignments in linear space. *Tech. Rep. ORNL/TM-12764, Oak RidgeNational Lab.*, 1994.

[31] A. Driga, P. Lu, J. Schaeffer, D. Szafron, K. Charter, and I. Parsons. FastLSA: A Fast, Linear-Space, Parallel and Sequential Algorithm for Sequence Alignment. *Algorithmica*, 45(3):337–375, 2006.

[32] JD Thomopson, D.G. Higgins, and TJ Gibson. ClustalW. *Nucleic Acids Res*, 22:4673–4680, 1994.

[33] A. Boukerche, A. C. Melo, E. F. Sandes, and M. Ayala-Rincon. An exact parallel algorithm to compare very long biological sequences in clusters of workstations. *Cluster Computing*, 10(2):187–202, 2007.

[34] A. Boukerche, R. B. Batista, and de A. C. Melo. Exact pairwise alignment of megabase genome biological sequences using a novel z-align parallel strategy. In *IPDPS-NIDISC*, pages 1–8, 2009.

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

Edans F. O. Sandes received his B.Sc. in Computer Science in 2006 and his M.Sc. degree in Informatics in June 2011, both from the University of Brasilia, Brazil. He is currently a PhD student at the Department of Computer Science at the University of Brasilia. His research interests include computational biology, GPGPUs and load balancing.

Alba C.M.A. Melo received her Ph.D. in Computer Science from the Institut National Polytechnique de Grenoble (INPG), France, in 1996. She is currently an associate professor in the Computer Science Department at the University of Brasilia, Brazil. Her research interests include GPGPUs, FPGAs, cluster computing, grid comput and computational biology. She is a Senior Member of IEEE Society and the IEEE Computer Society.