2011 IEEE International Parallel & Distributed Processing Symposium

Smith-Waterman Alignment of Huge Sequences with GPU in Linear Space

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

***Abstract*—Cross-species chromosome alignments can reveal ancestral relationships and may be used to identify the peculiarities of the species. It is thus an important problem in Bioinformatics. So far, aligning huge sequences, such as whole chromosomes, with exact methods has been regarded as unfeasible, due to huge computing and memory requirements. However, high performance computing platforms such as GPUs are being able to change this scenario, making it possible to obtain the exact result for huge sequences in reasonable time. In this paper, we propose and evaluate a parallel algorithm that uses GPUs 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 that are able to reduce significantly the amount of data processed and that enforce full parallelism most of the time. Using the GTX 285 Board, our algorithm was able to produce the optimal alignment between sequences composed of 33 Millions of Base Pairs (MBP) and 47 MBP in 18.5 hours.**

I. Introduction

In the last decade, genome projects have produced a huge amount of new biological data. In order to better understand a newly sequenced organism, biologists compare its sequence against millions of other sequences, in order to infer properties. Sequence comparison is, thus, one of the most important mechanisms in Bioinformatics. One of the first exact methods to globally compare two sequences is Needleman-Wunsch (NW) [1]. 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. The NW algorithm was modified by Smith-Waterman (SW) [2] to deal with local alignments. 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, without changing time and space complexity, 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 30 MBP sequences, we would need at least 3.6 PB of memory. This fact was observed by Myers-Miller [4], that proposed the use of Hirschberg's algorithm [5] to compute global alignments in linear space. The algorithm uses a divide and conquer technique that recursively splits the DP matrix to obtain the optimal alignment.

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 some implementations of SW in GPUs [6, 7, 8, 9, 10, 11, 12, 13], that were able to obtain the similarity score with very good speedups. Nevertheless, with the exception of CUDAlign 1.0 [13], all of them define a maximum size for the query sequence. That means that two huge sequences cannot be compared in such implementations.

As far as we know, the only strategies that are able to retrieve the alignment in GPUs are [6] and [12]. Since both of them execute in quadratic space, the sizes of the sequences to be compared is severely restricted.

In this paper, we propose and evaluate CUDAlign 2.0, a new algorithm using GPU that is able to retrieve the alignment of huge sequences with the SW algorithm, using the affine gap model. Our implementation is only bound to the total available global memory in the GPU and the available disk space in the desktop. Our algorithm receives two input sequences and provides the optimal 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 executes SW [2] to retrieve the best score and its position in the DP matrices, as in CUDAlign 1.0 [13]. Also, some special rows are saved to disk. The goal of stages 2, 3 and 4 is to retrieve points where the optimal alignment occurs in special rows/columns, thus creating small sub-problems. In stage 5, the alignments for each sub-problem are obtained and concatenated to generate the full alignment. In stage 6, the alignment can be visualized. The proposed algorithm was implemented in CUDA and C++ and executed in the GTX 285 board. With our algorithm, we were able to retrieve the alignment between the human chromosome 21 and the chimpanzee chromosome 22, with respectively 47 MBP and 33 MBP, in 18.5 hours, using reasonable disk area and GPU memory.

The rest of this paper is organized as follows. In Section II, we present the Smith-Waterman and the Myers and Miller algorithms. In Section III, related work is discussed. Section

1530-2075/11 $26.00 © 2011 IEEE
DOI 10.1109/IPDPS.2011.114

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

IV describes our proposed algorithm. Experimental results are shown in Section V. Finally, Section VI concludes the paper and presents future work.

II. Biological 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 [14]. 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 those sequences; or c) semi-global, composed of prefixes or suffixes of those 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) $+1$, 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. 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 [14]. 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}.

A. Smith-Waterman Algorithm (SW)

The algorithm SW [2] is an exact method based on dynamic programming to obtain the best 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 best local 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, $p(i,j) = ma$ 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 (Equation 2) and F (Equation 3) are needed. Even with this, time complexity remains quadratic [3]. The best 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.

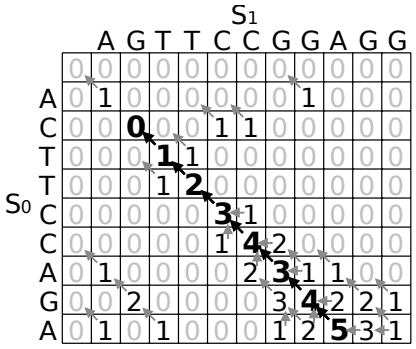


Figure 2. DP matrix for sequences S_0 and S_1. Bold arrows indicate the traceback to obtain the optimal alignment.

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

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

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

Phase 2 - Obtain the best alignment - To retrieve the best local alignment, the algorithm starts from the cell that contains the highest score 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 $S_1[j]$.

B. Myers and Miller's Algorithm (MM)

For long sequences, space is a limiting factor for the optimal alignment computation. Myers and Miller proposed an algorithm [4] that computes optimal global alignments in

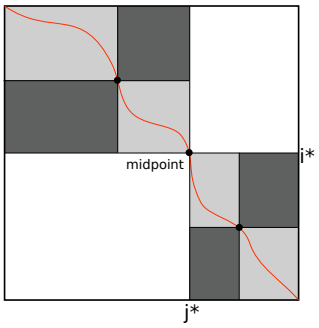


Figure 3. Recursive splitting procedure in MM.

linear space. It is based on Hirschberg [5], 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, in linear space, the midpoint of the LCS using the information obtained from the forward and the reverse directions. Given this midpoint, the problem is divided in two smaller subproblems, that are recursively divided in more midpoints.

The MM algorithm works as follows. 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 $S_1[1..j]$ 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 $S_1[j..n]$ 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 realizes a *matching procedure* between two pairs of vectors: a) vectors CC against RR and b) vectors DD against SS. The midpoint is the coordinate $(i*, j*)$, where $j*$ is the position that satisfies the maximum value in Formula 4.

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

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.

III. Related Work

In this section, some variants of SW are briefly discussed. First, implementations with linear space complexity are presented. Second, GPU implementations are discussed. Finally, an overview of CUDAlign 1.0 [13] is provided.

A. Linear Space Parallel Algorithms

In [15], the MM algorithm 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 [16], a parallel algorithm using prefix computation is presented. The DP matrix is divided into $p-1$ special columns that creates 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 [17], the work in [16] 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 in 4,862s.

FastLSA [18] 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 small problems 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, the run time to obtain the alignment between two 300 KBP sequences was 292.84s.

B. GPU Algorithms

Recently, many GPU implementations of SW have been proposed. Most of them tackle the problem of finding the most similar sequence in a genomic database for a given query sequence. In addition, the GPU often only obtains the score of the most similar sequence. When necessary, the full alignment is obtained in CPU, which is not very challenging when the sizes of the sequences are small.

Table I summarizes the main characteristics of some GPU approaches and the best results presented in the papers. As can be seen in Table I, most of the GPU SW proposals do not provide the alignment in GPU, but provide only the score as output. In [6] and [12] the alignment is also provided,

Paper	Align	Max. Query	GCUPS	GPU
DASW [6]	yes	16,384	0.2	7800 GTX
Weiguo Liu [7]	no	4,095	0.6	7800 GTX
SW-CUDA [8]	no	567	3.4	8800 GTX
CUDASW++ 1.0 [9]	no	5,478	16.1	GTX 295
Ligowski[10]	no	1,000	14.5	9800 GX2
CUDASW++ 2.0[11]	no	5,478	29.7	GTX 295
CUDA-SSCA#1[12]	yes	1,024	1.0	GTX 295
CUDAlign 1.0[13]	no	32,799,110	20.3	GTX 285

Table I
GPU SMITH-WATERMAN PAPERS

but in the experimental results very small query sequences were used (16 KBP and 1 KBP, respectively). Most of the approaches are able to calculate more than 1 Billion cells of the DP matrices per second (GCUPS). The maximum size of the query sequence is restricted in most of the approaches, with the exception of CUDAlign 1.0 [13], that was able to obtain the optimal score for two sequences of 33 MBP and 47 MBP, respectively. As far as we know, there is no proposal using GPU that retrieves the full optimal alignment of two Megabase genomic sequences.

C. CUDAlign 1.0

CUDAlign 1.0 was proposed in [13] and it is able to compare sequences with more than 1 MBP in GPU. It does not execute the traceback phase of SW, so its output is only the best score and its end position.

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. Each block contains T threads and 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 to be emptied and filled at each external diagonal calculation, providing full parallelism during almost all the execution, except in the very beginning and very close to the end. To avoid hazards, each block is processed in two phases: short phase and long phase. The short phase is responsible for the first T cells and the long phase for the remaining cells. An optimized kernel is used in the long phase, achieving faster execution. The 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 access is designed to better fit in the CUDA architecture.

Two main data structures are used in CUDAlign 1.0. The *horizontal bus* is the area of global memory used to store the last row of each block. This area is used to transfer the values of the cells 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 at each thread. This area is used to transfer the cells 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 [13], including the description of the short and long phase kernels and the pseudocodes executed in GPU and CPU.

IV. CUDALIGN 2.0: RETRIEVING ALIGNMENTS IN GPU

CUDAlign 2.0 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 restricted memory.

The CUDAlign 2.0 algorithm is divided in 6 stages, where some of them may be skipped depending on the inputs. Each stage is described in the following sections, but first some notations and considerations will be given.

A. Preliminary Notations and Considerations

The sequences to be aligned are S_0 and S_1 and their sizes are m and n, respectively.

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 II-B) 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 $C_s = (i_s, j_s, score_s, type_s)$ and $C_e = (i_e, j_e, score_e, type_e)$ form a *partition* $P = (C_s, C_e)$ of the matrix. The crosspoints C_s and C_e are the edge coordinates of the partition, where C_s is the start of the partition and C_e is its end. This partition creates an alignment subproblem for subsequences $S_0[i_s + 1..i_e]$ and $S_1[j_s + 1..j_e]$.

Special care must be taken when $type_s \neq 0$ or $type_e \neq 0$. In a sequence of gaps, only the rightmost gap is counted as a gap opening. When a non-zero type happens in the start 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 2.0, many crosspoints are generated in order to reduce the original problem to many

small subproblems. The list of crosspoints obtained at stage k are represented by $L_k = \{C_1, C_2, \ldots, C_{|L_k|}\}$.

The special crosspoints C_1 and $C_{|L_k|}$ are, respectively, the *start point* and the *end point* of the optimal alignment. The start point $C_1 = (i_1, j_1, score_1, type_1)$ always has $score_1 = 0$ and $type_1 = 0$ and the end point $C_{|L_k|} = (i_{|L_k|}, j_{|L_k|}, score_{|L_k|}, type_{|L_k|})$ has $score_{|L_k|} = best$, where $best$ is the score of the optimal alignment, and $type_{|L_k|} = 0$.

When $L_k = \emptyset$, no crosspoint has been found until that moment. If only one crosspoint has been found, this crosspoint is the end point of the alignment and the list L_k is represented by $L_k = \{*, C_1\}$, where the symbol $*$ represents an unknown start point. The partition formed by two any crosspoints C_i and C_j with $j > i$ will have optimal score $S(C_i, C_j) = score_j - score_i$. Note that if $i = 0$, $S(C_0, C_j) = score_j$, because $score_i = 0$. Particularly, if $i = 0$ and $j = |L_k| - 1$, then $S(C_0, C_{|L_k|-1}) = score_{|L_k|-1} = best$.

The number of blocks used in stage k is represented as B_k and the number of threads in each block are represented as T_k.

B. Stage 1 - Obtain the best score

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 $L_1 = \{*, C_1\}$, where C_1 is the end point of the optimal alignment and the start point is unknown. Stage 1 computes the best score with the affine gap model (Section II) using the GPU algorithm proposed in [13] with one modification: some special rows are saved to disk. These special rows are flushed from the horizontal bus (Section III-C) at a certain interval of blocks. Because 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.

Note that the horizontal bus is shared between the blocks in the same external diagonal, so the bus contains data from different rows. Figure 5 shows the horizontal bus (in bold) scattered across the blocks of the same external diagonal (in gray). It can be seen that many iterations of external diagonals may be executed until a full special row is flushed. The short phase of the last block must also be processed in order to complete a full 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 II). So, each special row has $8n$ bytes and the maximum number of special rows that can be saved is at most $\frac{|SRA|}{8n}$. 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.

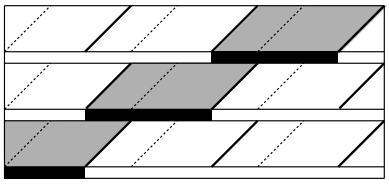


Figure 5. Shifted Bus

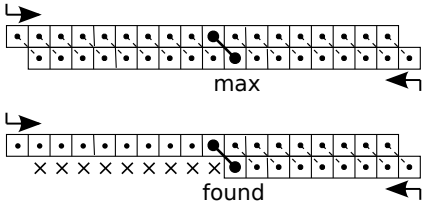


Figure 6. Matching Procedures. The MM matching procedure (top) matches all cells and finds out where the maximum score occurs. In the goal-based matching procedure (bottom), the maximum score is already known, so matching can stop after the maximum score is found. The '×' symbol represents the cells that do not need to be matched.

Figure 4(a) shows the outputs of Stage 1. The flush interval was 4 in this example, so after calculating 4 rows, a special row is flushed to disk. The special rows are illustrated as a darker horizontal line in the figure. The position of the best score is illustrated as a cross.

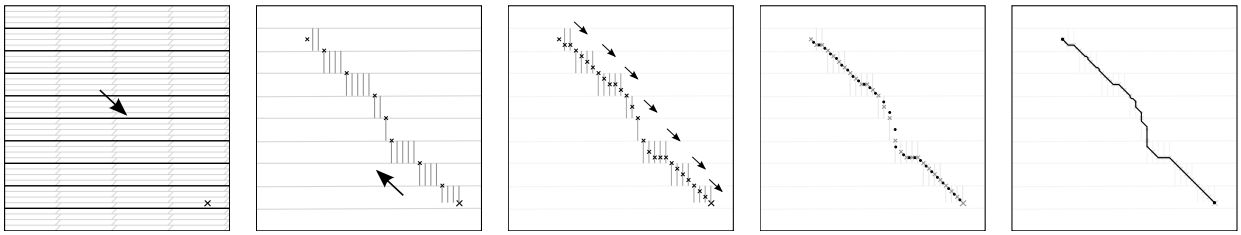
C. Stage 2 - Partial Traceback

Given the outputs of Stage 1, Stage 2 executes a semi-global alignment in the reverse direction starting from the end point of the alignment. The goal of Stage 2 is to find all the crosspoints ($L_2 = \{C_1, \ldots, C_{|L_2|}\}$) over the special rows intercepting the optimal alignment, including the start point C_1 of the alignment. Stage 2 is run in GPU and it is very similar to Stage 1.

In Stage 2, we propose the following optimizations: goal-based matching procedure and orthogonal execution.

1) Goal-Based Matching Procedure: In order to find a crosspoint, an adapted version of the MM's matching procedure is done between the special row and the reverse alignment. The original MM algorithm (Section II-B) matches all cells of the middle row against the reverse alignment, so the crosspoint is the position where the maximum score occurs. In Stage 2, differently than the original MM algorithm, the maximum score is already known and it is called the *goal score* (initially this is the best score obtained in Stage 1). This observation allows the procedure to stop as soon as the goal score is found in the matching procedure. Figure 6 illustrates the difference of the original MM's matching procedure and the goal-based matching procedure.

2) Orthogonal Execution: To take advantage of the Goal-Based Matching Procedure, instead of executing the threads



(a) Stage 1 finds the best score and its position. Special rows are stored on disk.

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

(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 4. Execution of each stage until obtaining the full alignment

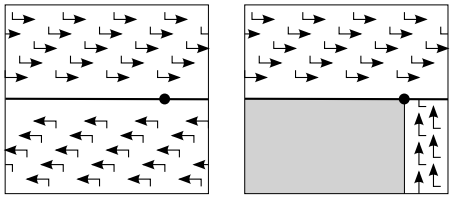


Figure 7. Orthogonal Execution. The conventional thread execution (on left) fully processes both halves of the matrix. Using orthogonal thread execution (on right), the threads can stop execution as soon as the matching goal is found. The gray area is not processed, reducing the execution time. Note that orthogonal execution is only possible if the matching goal is known.

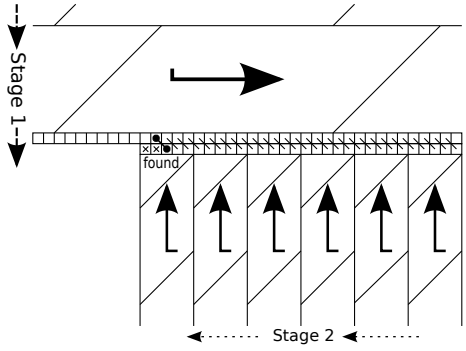


Figure 8. Detail on the orthogonal matching procedure between the forward alignment (stage 1) and the reverse alignment (stage 2).

in the same horizontal direction of Stage 1, Stage 2 calculates vertically, as illustrated in Figure 7. This technique is called *orthogonal execution* and its goal is to reduce the area processed until the matching occurs. Note that the matching procedure is executed for each block, so it does not wait for the full computation of the bottom area. Figure 8 shows in detail the matching region.

The vectors that are compared in the matching procedure are the nearest special row (forward) and the vertical bus of the last block of Stage 2 (reverse). Nevertheless, the vertical bus stores the last internal diagonal of the block (Section III-C). So the vertical bus had to be rectified in order to the matching procedure be correctly executed. This correction is made in the short phase, where the last cells of the last block are saved before starting a new line. The vector used to store the last cells values is called *rectified vertical bus*.

Whenever a match occurs, a new crosspoint is registered and the reverse matrix computation restarts from that point. So, the next special row is loaded and a new iteration of Stage 2 is done until it founds a new crosspoint again. At each iteration, the goal score must be updated to the value found in the special row. If there is a gap in this crosspoint, the F matrix value is taken, or else the H value is taken. This procedure is repeated until the goal score is zero.

Stage 2 does not compute the full DP matrix, but only the cells near the optimal alignment. Approximately, the area processed is the size of the flush interval multiplied by the size n of the sequence S_1. So, the processing area is inversely proportional to the number of special rows saved in Stage 1.

3) Differences from Stage 1: Unlike Stage 1, that searches the value of the best score, Stage 2 just needs to find out where the best score happens in the reverse DP matrix. That position is the start point of the alignment. As an optimization, this check is made only when the reverse alignment is near to its end, and that happens when the goal score can be found in a position below the next special row. When the goal score is found in the reverse matrix, this position is the start point of the forward alignment and the Stage 2 finishes execution.

Another modification in Stage 2 compared to Stage 1 is that the DP computation is based on the global recurrence equation, instead of the local recurrence that is executed in Stage 1. This requires a proper initialization before each iteration of Stage 2. When the previous crosspoint is a gap, the initialization must be adjusted not to consider a duplicated gap open penalty.

All the Stage 1 optimizations are also applied in Stage 2. The Cells Delegation, the Phase Division and the Memory Access Design are executed in the same way, but the

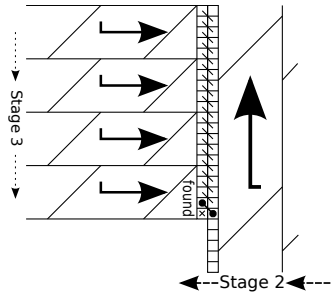


Figure 9. Detail on the orthogonal matching procedure between the reverse alignment (stage 2) and the forward alignment (stage 3).

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 [13]. 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 each special row.

Figure 4(b) shows the outputs of Stage 2. Starting from the position of the best score, the execution is made in reverse direction. Each thread executes α columns from that point up to the next special row, so the horizontal bus represents a column instead of a row at Stage 1. Special columns are saved as in Stage 1 and the special columns are illustrated as dark vertical lines in the figure. The rectified vertical bus is then matched with the special row of Stage 1. Whenever the goal score is found in the matching procedure, the process is reinitiated from that point, up to the next special row. This is done until the start of the alignment is found. Each interception of the alignment is marked as a cross in Figure 4(b).

D. 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, unlike Stage 2, that only knows the end point of the alignment and the start point is unknown. 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, so the threads executes horizontally, as in Stage 1. Figure 9 shows the execution directions and the matching procedure of Stage 3.

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 becomes 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.

Figure 4(c) shows the outputs of Stage 3. Each partition is split in many crosspoints, one for each special column saved in Stage 2.

E. Stage 4 - MM with balanced splitting and orthogonal execution

This stage executes at the CPU the MM Algorithm (Section II-B) 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 4(d) shows one iteration of Stage 4.

An optimization, called *Balanced Splitting*, is 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.

Figure 10 compares, using an example, the unbalanced and the balanced splitting. After the second split, the unbalanced splitting generates a partition larger than the maximum partition size (max), requiring a third splitting. The balanced splitting generates partitions with balanced sizes, so they may fit the maximum partition size earlier, at the second splitting. Note that, in the balanced splitting, the splitting is always done against the largest dimension of the partition. Depending on the sizes of the dimensions, the partition will be split in the middle rows or in the middle columns.

Orthogonal execution is also applied to speedup Stage 4. The forward computation will compute all the top half of the matrix and the reverse computation will process the bottom half, but only until it finds the crosspoint. In the average case, the new crosspoint generated resides in the center position, so, in average, only 50% of the bottom half needs to be processed. Considering both the top and the bottom computations, 75% of the matrix of that partition will be processed in the average case, so a speedup of 25% is expected in Stage 4 using the orthogonal execution.

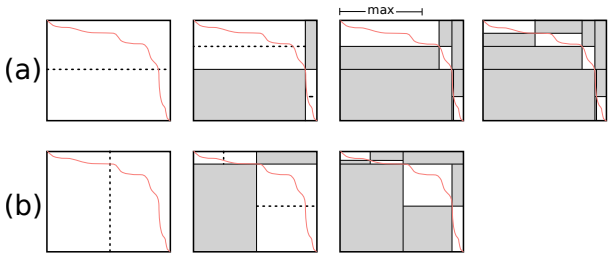


Figure 10. Using the original MM Algorithm (a), the splitting is always done in the middle row, so 3 splitting steps are necessary to obtain partitions smaller than the maximum partition size (max). Using the balanced splitting (b), the splitting is made in the largest dimension, so 2 splitting steps are sufficient.

F. 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 4(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, 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 best 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 at Stage 6.

G. 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.

H. Complexity Analysis

In this section, a worst case complexity analysis will be done for each of the stages of CUDAlign 2.0.

As shown in [13], CUDAlign 1.0 runs in time $O(mn)$ and space $O(m+n)$. With the proposed modification, a disk area of size $|SRA|$ is reserved with at least the size of one special row, so we need disk space in $O(max(n, |SRA|)) = O(n)$. Considering both the disk area in $O(n)$ and the volatile memory in $O(m + n)$, the total space complexity of Stage 1 is also $O(m + n)$. Note that saving the special rows does not change the time complexity, so Stage 1 is also $O(mn)$.

In Stage 2, the execution time depends on the length of the optimal alignment. The longest alignment possible happens when the end point is at the end of the matrix and the start point is at the beginning. In this case, there will be as many crosspoints as the number of special rows ($k = \frac{|SRA|}{8n}$ and $k \geq 1$). The distance $y = \frac{m}{k+1}$ between special rows limits the size of each partition, so the height of each partition is limited to the size of y. Nevertheless, SRA needs to store at least one special row ($k \geq 1$), so y is $O(m)$. Using the orthogonal execution explained in Section IV-C, the time complexity is $O(yn) = O(mn)$. If Stage 3 needs to be executed, at least one special column must be saved per partition and each special column has size y. So the disk area for special columns is $O(max(y, |SRA|)) = O(m)$. The RAM memory used is $O(y + n) = O(m + n)$ because the execution method is essentially the same as Stage 1. The total space complexity is then $O(m + n)$ and the execution time is $O(mn)$.

The complexity analysis of Stage 3 is analogous to the one made for Stage 2. Although the computation of each partition is not done after the last special column, this does not reduce the time complexity of this stage. So, the time complexity is $O(mn)$ and the space complexity is $O(m+n)$. There are no special rows stored in SRA in this stage.

The MM algorithm is known to be $O(mn)$ in time and $O(m + n)$ in space [4]. Therefore, Stage 4 has also the same complexities. As we guarantee that the partitions will be smaller than the constant *minimum partition size*, so Stage 5 needs to align $O(m + n)$ partitions of size $O(1)$. Thus, the execution time and memory utilization of Stage 5 is $O(m + n)$. Stage 6 is used just to visualize the alignment and it is also linear in space and memory.

Analyzing all stages, the overall implementation has time complexity of $O(mn)$ and space complexity of $O(m + n)$ (considering both the RAM memory and the disk space). Note that, although the disk space has linear complexity $O(m+n)$, the disk space needed to save one single special row and one special column for each partition is usually

much smaller then the available disk space used for the constant $|SRA|$.

V. Experimental Results

CUDAlign 2.0 was implemented in CUDA 3.1 and C++ and tested in an NVIDIA GeForce GTX 285. This board has 1GB of memory, 30 multiprocessors and 240 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. The execution configurations used for GTX 285 were $\alpha = 4$, $B_1 = 240$, $T_1 = 2^6$, $B_2 = B_3 = 60$ and $T_2 = T_3 = 2^7$. The number of blocks may be reduced during runtime in order to satisfy the minimum size requirement in each stage. Note that the number of blocks must be preferably a multiple of the number of multiprocessors (30 for GTX 285). By doing this, a better performance can be achieved because the multiprocessors do not become idle when they reach the end of an external diagonal.

A. Execution Times and GCUPS

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 II. The sizes of these real sequences range from 162 KBP to 47 MBP and the sequences are the same used in [13].

For all pairs of sequences shown in Table II, Table III lists the best score, its end and start positions, the length of the optimal alignment, the number of gaps found, as well as the number of cells processed in Stage 1. Note that the score for the 32,799KBP $\times$ 46,944KBP comparison is 27,206,434, which indicates that the optimal alignment is huge.

An usual performance metric for Smith-Waterman implementations is the number of cell updates per second (CUPS), 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. Table IV presents the execution times and MCUPS of Stage 1, with and without flushing special rows to disk. Note that, for long sequences, the overhead is about 1% of the execution time, which we consider very low. In these tests, the size of the SRA was chosen empirically. The results without flushing rows are basically the ones presented in [13], but an increased overall performance was noted due to the higher version of the CUDA Platform SDK (3.1 compared to 2.2).

Table V shows the execution time for all the sequences and all the stages on GeForce GTX 285. Note that when the length of the optimal alignment is small, the runtime of Stages 2, 3, 4 and 5 are negligible, as can be easily seen in comparisons 162K$\times$172K, 543K$\times$536K and 7146K$\times$5227K.

Figure 11 shows the wallclock execution times for CUDAlign 2.0 vs. DP matrix size (number of cells calculated) in logarithmic scale. This shows the scalability

Comparison	No Flush		Flush		
	Time	MCUPS	SRA	Time	MCUPS
162K×172K	1.4	19769	5M	1.5	18678
543K×536K	12.9	22545	50M	13.6	21419
1044K×1073K	48.3	23205	250M	51.6	21706
3147K×3283K	436	23706	1G	448	23035
5227K×5229K	1147	23822	3G	1185	23068
7146K×5227K	1568	23816	3G	1604	23282
23012K×24544K	23620	23911	10G	23750	23780
32799K×46944K	64507	23869	50G	65153	23632

Table IV
Runtimes (in seconds) of Stage 1 of CUDAlign 2.0 flushing special rows to disk. The overhead of saving special rows depends on the sizes of the SRA and the sequences.

Comparison	Stages					Total
	1	2	3	4	5+6	
162K×172K	1.5	<0.1	<0.1	<0.1	<0.1	1.8
543K×536K	13.6	<0.1	<0.1	<0.1	<0.1	13.9
1044K×1073K	51.6	3.1	1.0	5.4	0.1	61.6
3147K×3283K	448	0.1	<0.1	0.3	<0.1	449
5227K×5229K	1185	65.9	20.3	47.6	1.9	1321
7146K×5227K	1604	<0.1	<0.1	<0.1	<0.1	1605
23012K×24544K	23750	0.3	<0.1	0.7	<0.1	23755
32799K×46944K	65153	805	236	376	9	66579

Table V
Runtimes of each stage of CUDAlign 2.0 on GTX 285 varying the comparison size. The total time includes all the stages and the I/O of reading the sequences.

of our approach with almost constant GCUPS for megabase sequences. Note that, for sequences longer than 3 MBP, CUDAlign 2.0 is able to achieve a sustained performance of 23 GCUPS when running at the GTX 285 board.

The execution times obtained by CUDAlign 2.0 were compared to the Z-align cluster solution [19], 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 or equal than 3 MBP each). Table VI presents the execution times for Z-align (1 core), Z-align (64 cores) and CUDAlign 2.0 (GPU). The same organisms were aligned by Z-align and CUDAlign. In this table, we also show the impressive speedups achieved by CUDAlign 2.0 over Z-align. Due to high execution times, the 5 MBP and 23 MBP alignments were not executed with Z-align using one core. Also, Z-Align was not able to align the human chromosome 21 with the chimpanzee chromosome 22.

Using only one core, Z-align aligned two 3 MBP sequences in 294,000 seconds (Table 3 in [19]) and using 64 cores, it aligned sequences of approximately 23 MBP in 400,863 seconds (Table 3 in [19]). In the first case, CUDAlign 2.0 obtained the alignment in 449 seconds (Table V), resulting in a speedup of 654.79 over an one-core CPU machine and, in the second case, the alignment was obtained in 23,755 seconds, with a speedup of 16.87 over a 64-core cluster. The maximum speedups obtained were 702.22 (1 core) and 19.52 (64 cores) when aligning sequences with, respectively, 3 MBP and 500 KBP. These results clearly

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
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 II
REAL SEQUENCES DETAILS. SIZES RANGE FROM 162 KBP TO 47 MBP.

Comparison	Cells	Score	End Position	Start Position	Length	Gaps
162K×172K	2.79E+10	18	(41058 , 44353)	(41040 , 44335)	18	0
543K×536K	2.91E+11	48	(308558 , 455134)	(308466 , 455042)	92	0
1044K×1073K	1.12E+12	88353	(1072950 , 722725)	(606895 , 259025)	471858	14021
3147K×3283K	1.03E+13	4226	(2991493 , 2689488)	(2977390 , 2675374)	14554	891
5227K×5229K	2.73E+13	5220960	(5227292 , 5228663)	(0 , 1)	5229192	2430
7146K×5227K	3.74E+13	172	(4655867 , 5077642)	(4655314 , 5077083)	565	18
23012K×24544K	5.65E+14	9063	(14651731 , 11501313)	(14642625 , 11492211)	9107	6
32799K×46944K	1.54E+15	27206434	(32718231 , 46919080)	(0 , 13841680)	33583457	1371283

Table III
RESULTS FOR THE REAL SEQUENCES USED IN TESTS (STAGE 1)

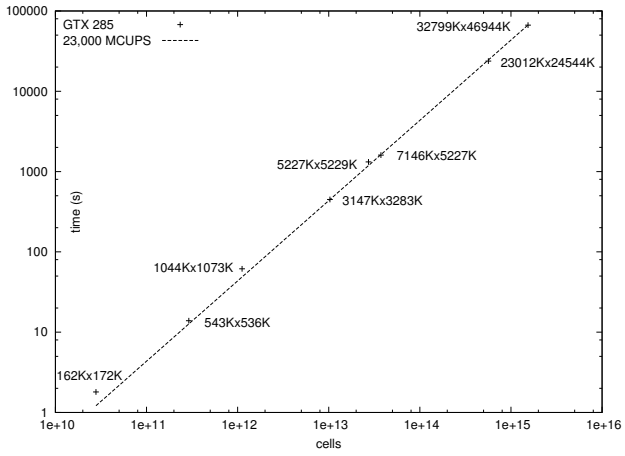


Figure 11. Runtimes (seconds) × DP matrix size (cells) in logarithm scale. Results show scalability with approximately 23,000 MCUPS for several sequence sizes.

show the great advantage of using our GPU solution.

Size	Times (s)			CUDAlign speedup	
	Z-align		CUDAlign 2.0 (GTX 285)	1 Core	64 Cores
	1 Core	64 Cores			
150K	1118	22.6	1.8	620.89	12.56
500K	9761	176	13.9	702.22	12.66
1M	32094	1044	61.6	521.01	16.95
3M	294000	8765	449	654.79	19.52
5M	-	23235	1321	-	17.59
23M	-	400863	23755	-	16.87
46M	-	-	66579	-	-

Table VI
CUDALIGN SPEEDUPS COMPARED TO THE Z-ALIGN CLUSTER SOLUTION.

B. Detailed Analysis

In order to analyze in detail the stages of CUDAlign 2.0, an extensive test was run for the human-chimpanzee chromosome comparison (32,799KBP × 46,944KBP). Table VII shows the runtimes of all stages on GTX 285, when varying the size of SRA. Some statistics about the execution are shown on Table VIII.

As we increase the size of SRA on Stage 1, its runtime is increased because more memory copies and disk writes

$\|SRA\|$	Stages						Sum
	1	2	3	4	5	6	
0GB	64507	-	-	-	-	-	-
10GB	64634	1721	126	8211	5.23	5.17	74702
20GB	64773	1015	111	2098	5.37	5.23	68008
30GB	64887	851	144	974	5.18	5.00	66866
40GB	65039	818	187	525	5.36	5.52	66580
50GB	65153	805	236	376	4.35	5.02	66579

Table VII
CHROMOSOMES COMPARISON - RUNTIMES (IN SECONDS) OF EACH STAGE USING DIFFERENT SPECIAL ROWS AREA SIZES.

SRA	10GB	20GB	30GB	40GB	50GB
B_1	240	240	240	240	240
B_2	60	60	60	60	60
B_3	60	30	26	14	10
$Cells_1$	1.54e+15	1.54e+15	1.54e+15	1.54e+15	1.54e+15
$Cells_2$	3.83e+13	1.95e+13	1.31e+13	1.00e+13	8.10e+12
$Cells_3$	2.23e+12	7.19e+11	4.44e+11	2.48e+11	1.82e+11
$\|L_1\|$	1	1	1	1	1
$\|L_2\|$	30	58	87	115	144
$\|L_3\|$	603	2338	5014	9283	12986
H_{max}	74956	28347	20675	17607	16583
W_{max}	56320	14336	6656	3684	2624
$VRAM_1(MB)$	435	435	435	435	435
$VRAM_2(MB)$	686	686	686	686	686
$VRAM_3(MB)$	685	685	685	685	685

Table VIII
STATISTICS OF EXECUTION: B_k IS THE MAXIMUM NUMBER OF CUDA BLOCKS USED IN STAGE k; $Cells_k$ IS THE NUMBER OF CELLS PROCESSED DURING THE STAGE k; $|L_k|$ IS THE NUMBER OF CROSSPOINTS FOUND AFTER THE EXECUTION OF STAGE k; H_{max} AND W_{max} ARE THE MAXIMUM HEIGHT AND MAXIMUM WIDTH OF THE PARTITIONS GENERATED IN THE END OF STAGE 3; $VRAM_k$ IS THE GPU MEMORY USED IN STAGE k.

It.	H_{max}	W_{max}	crosspoints	Time$_1$(s)	Time$_2$(s)
1	16583	2624	12986	250	188
2	8292	2539	25971	126	93.9
3	4146	2455	51941	62.7	47.0
4	2073	1904	103881	31.5	23.6
5	1037	1035	207761	15.9	12.0
6	519	519	415422	8.13	6.05
7	260	260	830157	4.21	3.15
8	130	130	1657901	2.37	1.81
9	65	65	3306027	0.26	0.26
10	33	33	3319746	0.25	0.25
11	17	17	3324354	0.29	0.25
12	16	16	3324553	-	-
Total	-	-	-	**501**	**376**

Table IX
RUNTIMES OF STAGE 4 FOR THE CHROMOSOMES ALIGNMENT USING MAXIMUM PARTITION SIZE OF 16. THE TABLE SHOWS, FOR EACH ITERATION (IT.), THE LARGEST HEIGHT (H_{max}) AND WIDTH (W_{max}) OF THE PARTITIONS, THE NUMBER OF CROSSPOINTS AND THE RUNTIMES OF THE ITERATION. TIME$_1$ SHOWS THE RUNTIME USING MM ALGORITHM AND TIME$_2$ SHOWS THE RUNTIME USING ORTHOGONAL EXECUTION.

	occurrences	%	score
Matches:	31696101	94.4%	31696101
Mismatches:	516073	1.5%	-1548219
Gap Openings:	66294	0.2%	-331470
Gap Extensions:	1304989	3.9%	-2609978
Total:	33583457	100%	27206434

Table X
NUMERICAL DETAILS OF THE HUMAN-CHIMPANZEE CHROMOSOMES ALIGNMENT

are made. In our platform, we observed that approximately 13 seconds were added for each additional GB stored in disk. Nevertheless, as the SRA increases, Stage 2 runs faster (Table VII) because the processed area in Stage 2 becomes smaller.

The runtime of Stage 3 tends to be smaller when the size of SRA is increased but, when this stage receives very small partitions, the minimum size requirement becomes a hard constraint and the number of blocks B_3 must be decreased. When B_3 becomes smaller, the runtime of Stage 3 tends to be higher. In Table VII, it is possible to see how the runtimes of Stage 3 increase when the size of SRA increases above a threshold. This is caused by the decrease of B_3, as can be seen in Table VIII.

When the SRA size is reduced, the runtime of Stage 4 becomes considerable. So, a proper SRA size must be chosen in order to maintain the scalability. In our tests, an SRA of 20GB is sufficient for a feasible Stage 4 runtime (Table VII) and this amount of disk is easily available in conventional workstations.

Using an $|SRA|$ of 50GB and maximum partition size of 16, Table IX shows the execution time per iteration of Stage 4 for both non-orthogonal execution ($Time_1$) and orthogonal execution ($Time_2$). Note that the performance gain using the orthogonal execution was 25%, which was the expected value in theoretical average case.

The execution time for Stage 5 depends on the maximum partition size of Stage 4. As the maximum partition size is a constant, the runtimes of Stages 5 and 6 are almost constant, independently of the chosen SRA size (Table VII). For the human-chimpanzee chromosome alignment, both stages took together approximately 10 seconds. The output of Stage 5 is the alignment in a 519KB binary file. The output of the Stage 6 is a 142MB text file, 279 times bigger than the binary representation.

The details of the optimal alignment between the chromosomes are shown in Table X. Note that the number of matches in the optimal alignment was 96.6% of the size of the chimpanzee 22 chromosome (32,799,110 BP), showing an impressive similarity between both chromosomes. In [20], this comparison was done with a heuristic method. As far as we know, there is no implementation, either in clusters, FPGAs or GPUs, that aligned those chromosomes with exact methods. Figure 12 shows a plotting of the chromosome alignment, provided as output of the visualization stage (Stage 6).

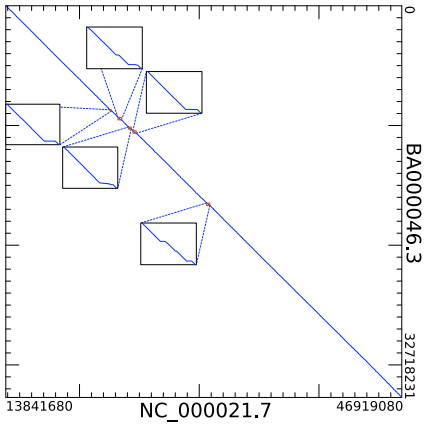


Figure 12. The alignment of the human-chimpanzee chromosomes. Five relevant sections of the alignment are zoomed in.

The sum of the runtime of all stages for the chromosome alignment, using a $|SRA|$ of 50GB, was 66579 seconds, approximately 18 hours and 30 minutes. The stages spent time in the following proportion: 97.86% in Stage 1, 1.21% in Stage 2, 0.35% in Stage 3, 0.56% in Stage 4, 0.007% in Stage 5 and 0.008% in Stage 6. Note that the GPU stages (Stages 1, 2 and 3) spent together 99.4% of the execution time.

VI. Conclusion and Future Work

In this paper we presented CUDAlign 2.0, that 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 the work in [13], but some special rows are saved in an area called Special Rows Area (SRA). 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.

Stages 1, 2 and 3 consume the most time of the algorithm, so these phases were implemented in GPU. The other stages were implemented in CPU.

The experimental results were obtained with a GTX 285 GPU with 1GB of memory. CUDAlign 2.0 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 18 hours and 30 minutes, where 99.4% of this time were spent on the GPU stages. CUDAlign 2.0 obtained maximum speedups of 702.22 and 19.52 when compared to the Z-align cluster solution with, respectively, 1 core and 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. If only one thread block processes each partition, the minimum size requirement would not exist, because there would not be any hazard inside each block.

Currently, Stage 4 is implemented in CPU. Using a sufficient SRA size, the execution time of Stage 4 is insignificant when compared to the previous stages, so the migration of this stage to GPU is considered as a future work. Nevertheless, the optimizations proposed in this stage were developed for a future GPU implementations. For example, the original MM Algorithm is recursively executed, but Stage 4 was implemented iteratively, that is easier to be implemented in GPU. Also, the balanced splitting procedure reduces the discrepancy of sizes between all partitions and, the more uniform is the execution time of each partition, the greater performance gains may be achieved in GPU. Because, at the end of Stage 4, each partition is limited by a constant size, this allows the migration of Stage 5 to GPU, potentially reducing the execution time.

Also, we intend to align other chromosome sequences and extend the tests to even more powerful GPUs, including systems with dual cards. Finally, we plan to migrate our work to the OpenCL framework to test our algorithm in other platforms.

References

[1] S. B. Needleman and C. D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *J Mol Biol*, 48(3):443–453, March 1970.

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

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

[6] John Johnson Yang Liu, Wayne Huang and Sheila Vaidya. Gpu accelerated smith-waterman. In *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] Yongchao Liu, Bertil Schmidt, and Douglas 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.

[12] Ali Khajeh-Saeed, Stephen 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.

[13] Edans Flavius de Oliveira Sandes and Alba Cristina Magalhaes Alves de Melo. CUDAlign: using GPU to accelerate the comparison of megabase genomic sequences. In R. Govindarajan, David A. Padua, and Mary W. Hall, editors, *PPOPP*, pages 137–146. ACM, 2010.

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

[15] 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.

[16] Srinivas Aluru, Natsuhiko Futamura, and Kishan Mehrotra. Parallel biological sequence comparison using prefix computations. *J. Parallel Distrib. Comput.*, 63(3):264–272, 2003.

[17] Stjepan Rajko and Srinivas Aluru. Space and time optimal parallel sequence alignments. In *Proceedings of the 2003 International Conference on Parallel Processing (32th ICPP'03)*, pages 39–47, Kaohsiung, Taiwan, October 2003. IEEE Computer Society.

[18] Driga, Lu, Schaeffer, Szafron, Charter, and Parsons. FastLSA: A fast, linear-space, parallel and sequential algorithm for sequence alignment. In *ICPP: 32th International Conference on Parallel Processing*, 2003.

[19] Azzedine Boukerche, Rodolfo Bezerra Batista, and Alba Cristina Magalhaes Alves de Melo. Exact pairwise alignment of megabase genome biological sequences using a novel z-align parallel strategy. In *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing - Workshop NIDISC*, pages 1–8, Washington, DC, USA, 2009. IEEE Computer Society.

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