CUDAlign 4.0: Incremental Speculative Traceback for Exact Chromosome-Wide Alignment in GPU Clusters

Edans Flavius de Oliveira Sandes, Guillermo Miranda, Xavier Martorell, Eduard Ayguade, George Teodoro, and Alba Cristina Magalhaes Melo, *Senior Member, IEEE*

Abstract—This paper proposes and evaluates CUDAlign 4.0, a parallel strategy to obtain the optimal alignment of huge DNA sequences in multi-GPU platforms, using the exact Smith–Waterman (SW) algorithm. In the first phase of CUDAlign 4.0, a huge Dynamic Programming (DP) matrix is computed by multiple GPUs, which asynchronously communicate border elements to the right neighbor in order to find the optimal score. After that, the traceback phase of SW is executed. The efficient parallelization of the traceback phase is very challenging because of the high amount of data dependency, which particularly impacts the performance and limits the application scalability. In order to obtain a multi-GPU highly parallel traceback phase, we propose and evaluate a new parallel traceback algorithm called Incremental Speculative Traceback (IST), which pipelines the traceback phase, speculating incrementally over the values calculated so far, producing results in advance. With CUDAlign 4.0, we were able to calculate SW matrices with up to 60 Peta cells, obtaining the optimal local alignments of all Human and Chimpanzee homologous chromosomes, whose sizes range from 26 Millions of Base Pairs (MBP) up to 249 MBP. As far as we know, this is the first time such comparison was made with the SW exact method. We also show that the IST algorithm is able to reduce the traceback time from 2.15× up to 21.03×, when compared with the baseline traceback algorithm. The human×chimpanzee chromosome 5 comparison (180 MBP×183 MBP) attained 10,370.00 GCUPS (Billions of Cells Updated per Second) using 384 GPUs, with a speculation hit ratio of 98.2 percent.

Index Terms—Bioinformatics, sequence alignment, parallel algorithms, GPU

1 Introduction

In comparative genomics, biologists compare the sequences that represent organisms in order to infer functional/structural properties. Sequence comparison is, therefore, one of the most basic operations in Bioinformatics [1], usually solved using heuristic methods due to the excessive computation times of the exact methods.

Smith–Waterman (SW) [2] is an exact algorithm to compute pairwise local comparisons. It is based on Dynamic Programming (DP) and has quadratic time and space complexities. The SW algorithm is divided in two phases, where the first phase is responsible to calculate a DP matrix in order to obtain the **optimal score** and the second phase (traceback) obtains the **optimal alignment**. SW is usually executed to compare (a) two DNA sequences or (b) a protein sequence (query sequence) to a genomic database. In the first case, a single SW matrix is calculated and all the Processing Elements (PEs) cooperate in this calculation, communicating to exchange border elements (fine-grained computation). For Megabase DNA sequences, a huge DP matrix with several Petabytes is computed. In the second case, multiple small SW matrices are calculated usually without communication between the PEs (coarse-grained computation). With the current genomic databases, often hundreds of thousands SW matrices are calculated in a single *query* × *database* comparison.

In the last decades, SW approaches for both cases have been parallelized in the literature, using multiprocessor/multicores [3], [4], Cell Broadband Engines (CellBEs) [5], Field Programmable Gate Arrays (FPGAs) [6], Application Specific Integrated Circuits (ASICs) [7], Intel Xeon Phis [8] and Graphics Processing Units (GPUs) [9], [10], [11], [12]. The SW algorithm is widely used by biologists to compare sequences in many practical applications, such as identification of orthologs [13], and virus integration detection [14]. In this last application, an FPGA-based platform [6] was used to compute millions of SW alignments with small query sequences in short time.

Nowadays, executing SW comparisons with Megabase sequences is still considered unfeasible by most researchers, which currently limits its practical use. We claim that important bioinformatics applications such as whole genome alignment (WGA) [15] could benefit from exact pairwise comparisons of long DNA sequences. WGA applications often construct global genome alignments by using local alignments as building blocks [16], [17]. In [18], the authors state that SW local alignments would be the best choice in this case. However, in order to compare 1 MBP × 1 MBP sequences, the SW tool took more than five days, preventing its use.

- *E. Sandes, G. Teodoro, and A. Melo are with the Department of Computer Science, University of Brasília, Brasília, DF, Brazil. E-mail: {edans, teodoro, albamm}@cic.unb.br.*
- *G. Miranda, X. Martorell, and E. Ayguade are with the Barcelona Supercomputing Center, Barcelona, Spain. E-mail: {guillermo.miranda, xavier.martorell, eduard.ayguade}@bsc.es.*

Manuscript received 29 Dec. 2014; revised 10 Dec. 2015; accepted 1 Jan. 2016. Date of publication 7 Jan. 2016; date of current version 14 Sept. 2016. Recommended for acceptance by D. Trystram. For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below. Digital Object Identifier no. 10.1109/TPDS.2016.2515597

In this paper, we focus on GPU solutions since they provide a very good compromise between programmability and performance for SW applications. GPUs are highly parallel architectures that may execute data parallel problems much faster than a general-purpose processor. A number of works have already examined the use of GPUs to accelerate SW computation. Some of them use only one GPU [11], [19], [20], [21], [22], whereas several approaches have been recently proposed to execute SW in multiple GPUs [9], [10], [12].

Some strategies [3], [10], [11], [20] are able to obtain the optimal local alignment of Megabase sequences longer than 1 Million Base Pairs (MBP). These strategies use linear space techniques to obtain the alignment with a reasonable amount of memory. As far as we know, there are no implementations of SW that obtained the optimal alignment between sequences longer than 60 MBP.

To measure the performance of SW strategies, the metric GCUPS (*Billions of Cells Updated per Second*) is often used. GCUPS are calculated using the formula $\frac{mn}{t \times 10^9}$, where m and n are the sequence sizes, and t is the execution time. As far as we know, the best GCUPS (6,020.00) was obtained by the Rivyera FPGA-based platform [6], which calculates SW scores using 128 FPGAs. To our knowledge, the best GCUPS for GPU platforms is 1,782.00, achieved by CUDAlign 3.0 [12]. Therefore, we decided to incorporate our traceback procedure in CUDAlign, since we target GPU platforms.

CUDAlign is a parallel application able to execute the SW algorithm with affine-gap in GPU for huge DNA sequences and it has evolved to many versions with incremental optimizations [11], [12], [19], [20], [23]. CUDAlign 2.1 [11] is able to obtain the optimal alignment of sequences up to 33 MBP in a single GPU. The first phase of SW is executed with a new optimization called block pruning and the second phase of SW executes a modified Myers-Miller (MM) algorithm [24] in four stages. CUDAlign 3.0 [23] was able to obtain the optimal local score comparing sequences up to 228 MBP in multiple GPUs [12].

In this paper, we propose and evaluate CUDAlign 4.0, a new version of CUDAlign that is able to retrieve the optimal SW alignment of huge DNA sequences executing both phases of the SW algorithm using multiple GPUs. In CUDAlign 4.0, we faced the challenge of parallelizing the traceback phase of SW, which is essentially serial and consumes more than 50 percent of the overall execution time in some cases. A straightforward parallelization of this phase would certainly compromise the scalability of the multi-GPU solution. This challenge motivated the development of a new traceback technique called Incremental Speculative Traceback (IST), which takes advantage of otherwise idle times of GPUs to speculate the location of the optimal alignment in order to speedup the traceback phase. As long as the points that actually belong to the optimal alignment are obtained by one GPU, our speculation technique uses them to trigger new speculations in the neighbor GPUs, correcting the former values in an incremental way.

CUDAlign 4.0 was implemented in CUDA, C++ and Pthreads. In order to execute the first phase of the SW algorithm with multiple GPUs, CUDAlign 4.0 extends CUDAlign 3.0 [12] including modifications to allow the execution of the traceback phase of SW. The alignment of the homologous human $\times$ chimpanzee chromosome 5 sequences (180 MBP $\times$ 183 MBP) was obtained in 53 minutes using 384 GPUs, with a processing rate of 10,370.00 GCUPS, surpassing the best GCUPS in the literature. We also show that our IST technique is very effective, executing the traceback phase up to 21.03$\times$ faster than the baseline pipelined traceback (PT). Since IST speculates during the time the GPUs would otherwise be idle, its overhead is negligible.

$$\begin{array}{cccccccc} A & T & A & C & T & C & C & A \\ A & T & A & - & T & C & C & A \\ \hline +1 & +1 & +1 & -2 & +1 & +1 & +1 & +1 \end{array}$$
$$score = 5$$

Fig. 1. Example of alignment and score.

The rest of this paper is organized as follows. Section 2 presents the biological sequence comparison problem and the SW algorithm. In Section 3 we discuss implementations of SW using multiple Processing Elements. Section 4 briefly reviews the CUDAlign algorithm for a single GPU. Section 5 presents our parallelization of the SW algorithm for multiple GPUs. In Section 6 we present experimental results. Finally, Section 7 concludes the paper and outlines future work.

2 DNA Sequence Comparison

A DNA sequence is represented by an ordered list of nucleotide bases. DNA sequences are treated as strings composed of characters of the alphabet $\sum = \{A, T, G, C\}$. To compare two sequences, we place one sequence above the other, possibly introducing spaces, making clear the correspondence between similar characters [1]. The result of this placement is an alignment.

Given an alignment between sequences S_0 and S_1, a score is associated to it as follows. For each pair of characters, we associate (a) a punctuation *ma*, if both characters are identical (*match*); or (b) a penalty *mi*, if the characters are different (*mismatch*); or (c) a penalty g, if one of the characters is a space (*gap*). The score is the addition of all these values. Fig. 1 presents one possible alignment between two DNA sequences. In this figure, $ma = +1$, $mi = -1$ and $g = -2$.

2.1 Smith-Waterman Algorithm

The algorithm SW [2] is based on DP, obtaining the optimal pairwise local alignment in quadratic time and space. It is divided in 2 phases: calculate the DP matrix and obtain the alignment (traceback).

Phase 1 - This phase receives as input sequences S_0 and S_1, with sizes $|S_0| = m$ and $|S_1| = n$. The DP matrix is denoted $H_{m+1,n+1}$, where $H_{i,j}$ contains the 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).

$$H_{i,j} = \max \begin{cases} H_{i-1,j-1} + (\text{if } S_0[i] = S_1[j] \text{ then } ma \text{ else } mi) \\ H_{i,j-1} + g \\ H_{i-1,j} + g \\ 0. \end{cases} \quad (1)$$

In addition, each cell $H_{i,j}$ contains information about the cell that was used to produce the value. The highest value in $H_{i,j}$ is the optimal score.

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

Fig. 2. DP matrix for sequences S_0 and S_1, with optimal score = 5. The arrows represent the optimal alignment.

Phase 2 (traceback) - The second phase of SW obtains the optimal local alignment, using the outputs of the first phase. The computation starts from the cell that has the highest value in H, following the path that produced the optimal score until the value zero is reached.

Fig. 2 presents a DP matrix with $score = 5$. The arrows indicate the alignment path. In this figure, two DNA sequences with sizes $m = 12$ and $n = 8$ are compared, resulting in a 9×13 DP matrix. In order to compare Megabase sequences of, for instance, 60 MBP a matrix of size 60,000,001 $\times$ 60,000,001 (3.6 Peta cells) is calculated.

The original SW algorithm assigns a constant cost g to each gap. However, gaps tend to occur together rather than individually. For this reason, a higher penalty is usually associated to the first gap and a lower penalty is given to the remaining ones (affine-gap model). Gotoh [25] proposed an algorithm based on SW that implements the affine-gap model by calculating three values for each cell in the DP matrix: H, E and F, where values E and F keep track of gaps in each sequence. As in the original SW algorithm, time and space complexities of the Gotoh algorithm are quadratic.

2.2 Myers–Miller Algorithm

For long sequences, space is a limiting factor for the optimal alignment computation. Myers and Miller proposed an algorithm [24] that computes optimal global alignments in linear space. It is based on Hirschberg [26], but applied over Gotoh (Section 2.1).

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 [24]. 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 2 [24].

$$\max_{j \in [0..n]} \left\{ max \left\{ \begin{array}{l} CC(j) + RR(n-j) \\ DD(j) + SS(n-j) - G_{open}. \end{array} \right. \right. \quad (2)$$

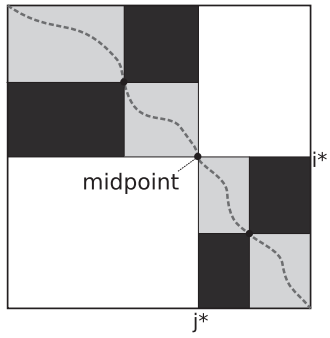


Fig. 3. Recursive splitting procedure in MM.

After the midpoint is found, the problem is recursively split into smaller subproblems, until trivial problems are found. Fig. 3 illustrates the execution of two steps of the MM algorithm. In the first step, the whole DP matrix is processed and the midpoint is found. In the second step, the top left and the bottom right rectangles are processed, producing two new alignment points. Both points are used to split the rectangles, generating four light gray areas which will be processed in the third step.

2.3 Parallel Smith–Waterman

In SW, most of the time is spent calculating the DP matrices and this is the part which is usually parallelized. In Equation (1), we can notice that cell $H_{i,j}$ depends on three other cells: $H_{i-1,j}$, $H_{i-1,j-1}$ and $H_{i,j-1}$. This kind of dependency is well suited to be parallelized using the wavefront method [27]. Using this method, the DP matrix is calculated by diagonals, and all cells in each diagonal can be computed in parallel.

Fig. 4 illustrates the wavefront method. In step 1, only one cell is calculated in diagonal d_1. In step 2, diagonal d_2 has two cells, that can be calculated in parallel. In the further steps, the number of cells that can be calculated in parallel increases until it reaches the maximum parallelism in diagonals d_5 to d_9, where five cells are calculated in parallel. In diagonals d_{10} to d_{12}, the parallelism decreases until only one cell is calculated in diagonal d_{13}. The wavefront strategy suffers from reduced parallelism during the beginning of the calculation (filling the wavefront) and the end of the computation (emptying the wavefront).

3 Related Work

There are two main types of SW computations: fine-grained or coarse-grained. Fine-grained proposals are the ones that

Fig. 4. Wavefront method.

TABLE 1
Multi-PE Smith–Waterman Fine-Grained Proposals

Ref.	Year	Output	Number of PEs	GCUPS	Max. Size
[3]	2004	align.	60 × CPUs	0.25	1,100,000
[4]	2013	score	6,144 × CPUs	15.50	24,894,269
[10]	2013	align.	2 × GPUs	65.20	32,799,110
[8]	2014	score	4 × Xeon Phis	114.40	50,000,000
[12]	2014	score	64 × GPUs	1,762.00	228,333,871

use more than one PE to compute the same DP matrix; otherwise, the proposal is classified as coarse-grained. Even though there are several multi-PE coarse-grained approaches for SW in the literature ([6], [9], [22], [28]), we will provide in this section a discussion of fine-grained approaches, which are more closely related to our work.

Table 1 lists fine-grained SW implementations for platforms composed of multiple PEs. The maximum number of PEs used by each paper is presented in column 4. As can be seen, the maximum number of PEs employed is 6,144 (cluster of multicores), 64 (GPU) and 4 (Xeon Phi).

GCUPS (column 5) range from 0.25 (2004) to 1,762.00 (2014), showing an spectacular increase in performance in 10 years. These GCUPS values, however, cannot be compared directly because they were obtained in different platforms, with different sequences. Nevertheless, they provide an indication of the potential of each platform. For fine-grained comparisons, the best GCUPS was obtained by CUDAlign 3.0 (1,762.00 GCUPS) [12] with 64 GPUs. If we also consider the coarse-grained approaches, the best GCUPS achieved so far is 6,020.00, obtained by the Rivyera platform [6] with 128 FPGAs. In this case, query sequences of exactly 100 characters are compared with the SW algorithm (linear gap function). The maximum size of the sequences compared in the papers is also presented in Table 1.

In Table 1 (column 3), two approaches were able to retrieve the optimal alignment of long sequences with multiple PEs. The first approach [3] uses Parallel Prefix (PP) to compute SW alignments. PP breaks the dependency among the DP cells, making it possible to compute a whole row/column in parallel, with a communication step at the end of each row/column computation. PP was also implemented for SW executions in the CellBE [5]. In [29], the wavefront method is compared to PP for SW computations. Using their implementations, the authors conclude that PP is faster than wavefront parallelization for sequences up to 4 K; for sequences of length 8 K or higher, wavefront is faster. Also, in [7], a hardware platform composed of a network-on-chip and ASICs is proposed that implements PP and wavefront parallelization (AD) to execute SW for small sequences. Even though the authors conclude that PP is faster in their platform for 1 K × 1 K comparisons, the execution times for PP increase when the number of PEs is increased from 64 to 128, limiting the scalability. This behavior does not happen with AD, showing a better potential for scalability. Based on these results, we opted not to use PP to retrieve SW alignments in our design but we think PP can be explored in future work.

The second approach that retrieves the optimal alignment for long sequences with multiple PEs is SW# ([10]), which implements the MM algorithm (Section 2.2), with some optimizations introduced by CUDAlign 2.1 [11]. This solution executes with up to two GPUs and, for this reason, we think that SW# may present limited scalability for the sequences longer than 200 MBP. Therefore, we opted to extend CUDAlign 3.0, which executed successfully with up to 64 GPUs, to retrieve SW alignments of huge sequences with the wavefront method.

4 CUDAlign Design for Single GPU

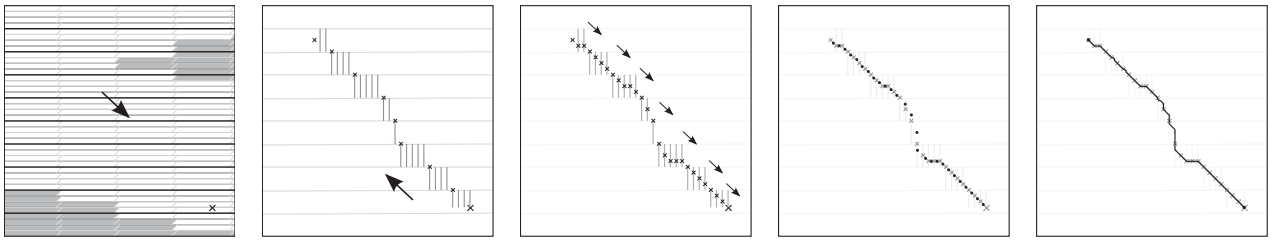
CUDAlign 1.0 [19] was proposed as a linear-space parallel algorithm able to compare Megabase DNA sequences with the affine gap model in a single GPU. Since then, CUDAlign 2.0 [20] and 2.1 [11] were proposed, with several improvements for single GPUs.

CUDAlign aligns two Megabase sequences in six stages (Fig. 5). Stage 1 corresponds to the first phase of the SW algorithm (Section 2.1) and Stages 2 to 5 correspond to the second phase of SW, where the DP matrix is reprocessed to find the points that belong to the optimal alignment [11]. Stage 6 is used to visualize the alignment.

4.1 Obtaining the Optimal Score (Stage 1)

Stage 1 (Fig. 5a) takes as input sequences S_0 and S_1, with sizes m and n respectively. Both sequences are stored in texture memory. 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. A thread is responsible to process α rows, so a block processes αT rows. A diagonal of blocks is called external diagonal and a diagonal of threads, inside a block, is called internal diagonal.

To reduce the execution time of Stage 1, three optimizations were proposed [11], [20]: Cells Delegation, Phase



(a) Stage 1 finds the optimal score and its position. Special rows are stored on disk.
(b) Stage 2 finds crosspoints between optimal alignment and special rows.
(c) Stage 3 finds more crosspoints over special rows stored from previous stages.
(d) Stage 4 executes Myers and Miller's algorithm between successive crosspoints.
(e) Stage 5 obtains the complete alignment between each successive crosspoints.

Fig. 5. General overview of the CUDAlign execution.

Division and Block Pruning. 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 time. To do this, the GPU blocks have a parallelogram shape, instead of the typical rectangular one. The right block processes the pending cells of its left block, with cells delegation.

To avoid concurrency hazards, each external diagonal is processed in two phases. The short phase calculates the first T internal diagonals of the block and the long phase calculates the remaining ones. Each phase corresponds to a kernel, and an optimized kernel is used in the long phase, achieving faster execution time [20].

Block pruning was proposed in CUDAlign 2.1 [11] and its goal is to to eliminate the calculation of blocks of cells that surely do not belong to the optimal alignment. This optimization may reduce the execution time in more than 50 percent if the sequences are very similar.

In order to accelerate the execution of the traceback phase (Section 2.1), CUDAlign stores some rows in a disk space called special rows area (SRA). Fig. 5a shows the outputs of Stage 1. In this figure, special rows are flushed to disk after each group of four blocks is calculated. 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.

4.2 Obtaining the Optimal Alignment (Stages 2–6)

Stage 2 executes in GPU a semi-global alignment in the reverse direction starting from the position where the optimal score was found in Stage 1. The goal of Stage 2 is to find all the crosspoints that belong to the optimal alignment and cross the special rows, including the start point of the alignment. The interceptions are found using the Myers-Miller matching procedure (Section 2.2). During the computation of Stage 2, some special columns are saved to disk. Fig. 5b shows the outputs of Stage 2, including the special columns in vertical lines. 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. At the end of Stage 2, the start and end points of the alignment in each partition are known and this information is used to iteratively compute more points that are part of the alignment in the further stages.

Stage 3 (Fig. 5c) 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. The goal of Stage 3 is then to obtain more crosspoints inside each partition.

Stage 4 executes in CPU a modified Myers and Miller algorithm between each successive pair of crosspoints found on Stage 3, using multiple threads. The goal of Stage 4 is to increase the number of crosspoints until the distance between any successive pair of crosspoints is smaller than a given limit. Fig. 5d shows the additional crosspoints generated 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 Fig. 5e. Stage 6 is an optional stage used only for visualization of the alignment.

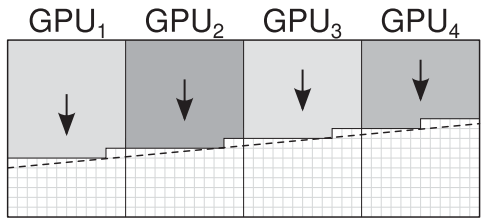


Fig. 6. Columns distributions for four GPUs.

Algorithm 1 presents the chain of inputs and outputs of the CUDAlign stages. The variable $best$ (line 1) contains the optimal score and its position found in Stage 1. Arrays $list_{2-4}$ (lines 2 to 4) hold crosspoints found in Stages 2 to 4. The bin (line 5) and txt (line 6) variables refer to the actual alignment files in binary representation and textual representation, respectively.

Algorithm 1. Overview of CUDAlign (single GPU)

1: $best \leftarrow$ Stage1()
2: $list_2 \leftarrow$ Stage2($best$)
3: $list_3 \leftarrow$ Stage3($list_2$)
4: $list_4 \leftarrow$ Stage4($list_3$)
5: $bin \leftarrow$ Stage5($list_4$)
6: $txt \leftarrow$ Stage6(bin)

5 CUDAlign Design for Multiple GPUs

CUDAlign 3.0 [12] was proposed as a Multi-GPU strategy that executes the first phase of the SW computation (Stage 1) in homogeneous and heterogeneous environments [23]. CUDAlign 4.0, proposed in this paper, extends the ideas of CUDAlign 3.0 in order to use multiple GPUs in both SW phases, obtaining the optimal alignment in a Cluster of GPUs. In this section we will present the design of CUDAlign 4.0. Section 5.1 focuses on the Stage 1 (based on CUDAlign 3.0 with some modifications) and Section 5.2 explains the new parallel traceback strategies proposed for Stages 2-4.

5.1 Multi-GPU Strategy for Stage 1

CUDAlign 3.0 [12] faced the problem of parallelizing Stage 1 using multi-GPUs. The parallelization was done using a multi-GPU wavefront method, where the GPUs are logically arranged in a linear way, i.e., the first GPU is connected to the second, the second to the third and so on (Fig. 6). Each GPU computes a range of columns of the DP matrix and the GPUs transfer the cells of their last column to the next GPU. For environments with homogeneous GPUs, an even distribution of columns is used [12]. For heterogeneous environments, the column distribution works as well using a weighted proportion that considers the computing power of each GPU [23].

5.1.1 Multi-Threaded Design

In CUDAlign 3.0, each GPU is bound to one process and each process has three CPU threads: one manager thread (T_M) that manages GPU computation, and two communication threads (T_C) that handle communication (Fig. 7). Iteratively, the T_M invokes GPU kernels for the current external diagonal and transfers cells of the first/last column between GPU and CPU. The short phase kernel (Section 4) was

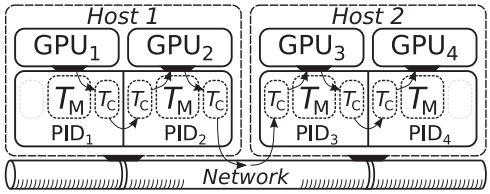


Fig. 7. Multi-GPU threads chaining.

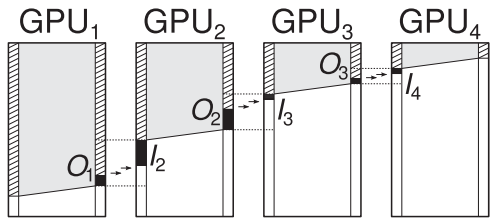


Fig. 8. Multi-GPU buffers in Stage 1.

modified to read/write the first/last columns from global memory, allowing the T_M to transfer cells between GPU/CPU. The T_M interacts with the T_C threads to receive its first column from the previous GPU and to send its last column to the next GPU. The transfers are made over chunks of cells with the same height of the block.

Inter-process communication is performed using sockets. If one host contains more than one GPU, CUDAlign 3.0 forks one process per local GPU. Fig. 7 illustrates the communication of four GPUs, where each host has two GPUs and each GPU has an associated process, with one T_M and two T_C.

5.1.2 Input/Output Buffers

Each T_C is associated to a circular buffer, that can be used as an input buffer (if it receives the column from the previous process) or as an output buffer (if it stores the column to be sent to the next process). If the input buffer is full or the output buffer is empty, its respective T_C blocks. On the other hand, if the input buffer is empty or the output buffer is full, the T_M blocks. It is important that the T_M does not block often, allowing us to overlap communication with computation.

Fig. 8 illustrates the buffers between four GPUs. Buffers I_2, I_3 and I_4 are the input buffers and buffers O_1, O_2 and O_3 are the output buffers. Each output-input pair of buffers $(O_{j-1} \rightarrow I_j)$ from successive GPUs is continually transferring data. These buffers are responsible to hide the interprocess communication in such a way that the overhead and small variations in the network may not be perceptible to the wavefront performance. Also, the amount of data waiting in each buffer may be an indication of the balancing quality. For instance, the pair of buffers $(O_1 \rightarrow I_2)$ in Fig. 8 has more data waiting in the input buffer I_2 than in the output buffer O_1, giving an indication that GPU_1 is producing data faster than GPU_2 can process. On the other hand, buffers $(O_3 \rightarrow I_4)$ are almost empty, indicating that GPU_3 and GPU_4 are processing in almost the same speed.

5.1.3 Modifications in Stage 1 for CUDAlign 4.0

In order to execute the traceback in Multi-GPUs, we introduced two modifications in CUDAlign 3.0. First, the CUDA GPU texture memory has a limit of 2^{27} linear elements [30] and this forbids its usage for sequences larger than 134 MBP. In CUDAlign 3.0, the texture was disabled for such large sequences, slowing down the computation in more than 7 percent [12]. In order to use the texture memory without the 134 MBP size limitation, we implemented in CUDAlign 4.0 a subpartitioning strategy. In this approach, the partitions that do not fit into the texture memory are divided in a grid of subpartitions. The subpartitions are computed one after the other, where the first column/row of one subpartition is obtained from the last column/row of its neighbor subpartitions. The second modification was the writing of special rows in the file system. These special rows are used in the further stages to find points where the optimal alignment crosses them (Section 4.2). The special rows written by one GPU are only read by the same GPU, unless for the border rows, which are shared using the TCP sockets and stored inside a filesystem.

5.2 Multi-GPU Strategy for Traceback

In this section, we detail the multi-GPU IST, which is proposed to obtain the optimal local alignment. IST uses otherwise idle GPU cycles to speculate the locations in which the optimal alignment is likely to be, before the optimal alignment locations are actually known. The IST strategy is built upon a baseline PT strategy (Section 5.2.1). The IST strategy is then presented in Section 5.2.2.

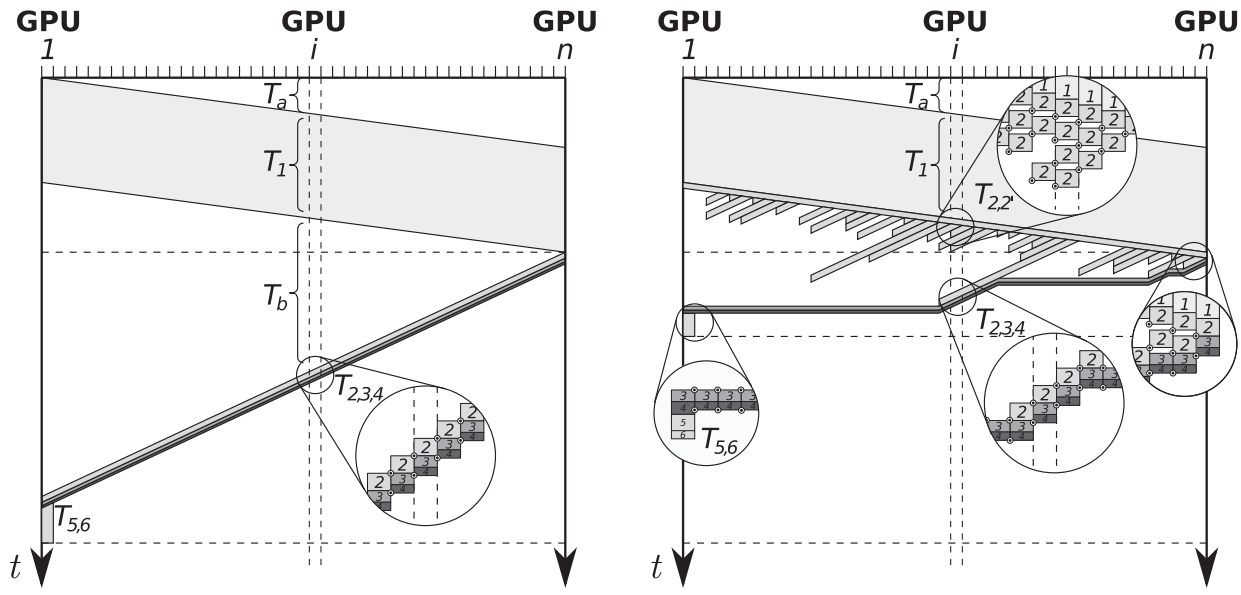
5.2.1 Pipeline Traceback- Baseline Strategy

Our baseline multi-GPU parallelization strategy (PT) employs the multi-GPU structure of CUDAlign 3.0 [12] combined with a pipelined strategy to compute Stages 2 to 4 (Section 4.2) from different partitions in parallel. For the sake of simplicity, whenever we state that GPU_i is processing Stage 4, 5 or 6, we actually mean that the process associated with GPU_i is executing it in CPU.

After Stage 1 is completed, the last GPU starts the execution of Stage 2 for the last partition. Whenever the crosspoint from the border column is calculated, its coordinates are sent to the previous GPU, which is responsible for the computation of the neighboring partition. The previous GPU may then start Stage 2. A dependency chain does exist among multiple GPUs. This dependency creates a critical execution path that results in idle time between the end of Stage 1 and the beginning of Stage 2. However, after the dependencies for computation of State 2 have been resolved, the execution of Stages 2, 3, and 4 may be computed in pipeline. This strategy is used for all GPUs. The Stages 5 and 6 are less compute intensive and, as a consequence, they are executed by the first GPU.

Fig. 9a illustrates the Pipelined Traceback execution timeline. The y-axis refers to time and x-axis represents each of the GPUs used, whereas each point shows the stage that a GPU is computing. The white area on the top of the plot (T_a) illustrates the wavefront startup time. The gray areas represent the computational time used for Stages 1 to 6 ($T_{1,2,3,4,5,6}$). The white area T_b between Stages 1 and 2 is the idle time after a GPU is done with Stage 1 for its partition, waiting to start the computation of Stage 2.

Complexity analysis. In this section, we assume that the manager threads do not block. Considering that p GPUs are used in Stage 2 and that there are more GPUs than special



(a) Pipelined Traceback (PT)

(b) Incremental Speculative Traceback (IST)

Fig. 9. Traceback timelines.

rows, we can roughly estimate that each GPU will compute an area of $\frac{m}{p} \times \frac{n}{p}$, resulting in a time complexity of $O(\frac{mn}{p^2})$ per GPU. Because the execution of the p GPUs has a dependency chain, their execution is serialized and the overall execution time is $O(\frac{mn}{p})$—the same execution time complexity of Stage 1. Although the pipelined traceback improves performance by executing Stages 2, 3, and 4 in parallel to the computation of other partitions, still there is a critical execution path resulting from dependencies in Stage 2 (Fig. 9a).

Pseudocode. The pseudocode for the Pipelined Traceback strategy is presented in Algorithm 2, which changes the original CUDAlign 2.1 traceback (Algorithm 1) to execute in multi-GPU environments. Variable n represents the nth GPU and $Recv_i$ and $Send_j$ are the message calls used to receive data from the ith GPU and send data to the jth GPU. To simplify the pseudocode, we ignored some border cases (first and last GPU), where the $Recv_i$ and $Send_j$ calls are not executed. Using the column distribution discussed in Section 5.1, Stage 1 (line 1) is executed for the columns assigned to the nth GPU and the variable $best'_n$ receives the local maximum score and its position in this range of columns. Then, the cumulative best score $best_n$ from GPUs 1 to n is passed through the GPUs (lines 2 to 4). In the last GPU, the variable $crosspoint$ will contain the overall best score and its position (line 6), and Stage 2 will start the execution (line 10) from the position with best score (if it resides in the last GPU column range). Further, the top-left crosspoint $list_2[0]$ (line 11) found in the border column by Stage 2 is sent to the previous GPU (line 12), which receives it in the $crosspoint$ variable (line 8). While the GPU that just received the crosspoint computes Stage 2, the previous GPU pipelines the execution of Stages 3 and 4 (lines 13 and 14). After each GPU is done with Stage 4, it will receive the crosspoints computed by the right-neighbor GPUs (line 15), concatenate them with its own crosspoints, and send them to the left-neighbor GPU (line 17). This process continues up to the first GPU (or the one containing the beginning of the alignment). This GPU computes the final alignment by executing Stages 5 and 6 (lines 19 and 20).

Algorithm 2. Pipelined Traceback (nth GPU)

1: $best'_n \leftarrow$ STAGE1()
2: RECV$_{n-1}$($best_{n-1}$)
3: $best_n \leftarrow max(best_{n-1}, best'_n)$
4: SEND$_{n+1}$($best_n$)
5: **if** n is last GPU **then**
6: $\quad crosspoint \leftarrow best_n$
7: **else**
8: $\quad$ RECV$_{n+1}$($crosspoint$)
9: **end if**
10: $list_2 \leftarrow$ STAGE2($crosspoint$)
11: $left\_crosspoint \leftarrow list_2[0]$
12: SEND$_{n-1}$($left\_crosspoint$)
13: $list_3 \leftarrow$ STAGE3($list_2$)
14: $list_4 \leftarrow$ STAGE4($list_3$)
15: RECV$_{n+1}$($list'_4$)
16: **if** $crosspoint.score \neq 0$ **then**
17: $\quad$ SEND$_{n-1}$($list_4 + list'_4$)
18: **else**
19: $\quad bin \leftarrow$ STAGE5($list_4 + list'_4$)
20: $\quad txt \leftarrow$ STAGE6(bin)
21: **end if**

5.2.2 Incremental Speculative Traceback

We have noticed in analyses of several tracebacks that the crosspoints found in border columns tend to coincide with the best score in these columns. The IST strategy uses this insight to estimate where the optimal alignment will cross intermediate border columns, allowing the execution of Stage 2 for multiple partitions in parallel even before the optimal alignment crosspoints are known. In parallel to this speculative computation, Stage 2 also starts its execution in the last GPU from the coordinate with the *actual optimal score* as described in Section 5.2.1. With IST, however, the GPU that receives the *actual crosspoint* (the point that

belongs the optimal alignment) will first check if that point was guessed correctly and has already been computed during the speculation phase. If this is the case, the GPU will not recompute its partition and will only forward the previously computed border crosspoint to the next GPU. Otherwise, the partition is recalculated.

We have also observed that, in regions with many gaps and mismatches, the actual crosspoint is usually more difficult to be predicted. Therefore, instead of only looking at local results to find the point for speculation, we have modified our solution for using partial results speculated by neighbor GPUs. With this approach, each GPU will forward the crosspoint found during its speculation to the neighbor GPU, which uses it as a new guess. This strategy performs an incremental speculation with increasing probability of success. It is important to highlight that the results of each speculative computation, e.g., special rows, must be saved. As such, we have limited the number of speculation attempts (max_spec_tries) in order to limit memory utilization.

Fig. 9b illustrates the Incremental Speculative Traceback. Stage 1 execution is the same as in the Pipelined Traceback (Section 5.2.1), but speculation in Stage 2 is carried out in previously idle times between the Stages 1 and 2. The incremental speculation is illustrated as many *recomputation lanes* going diagonally down-left. The recomputation lane starting from the last GPU refers to the critical execution path in Stage 2, which goes down in cases of recalculations, due to wrongly speculated crosspoints. Note that Stages 3 to 4 follow the execution of Stage 2 for the actual crosspoints in the optimal alignment.

Complexity Analysis. In this section, we also assume that the manager threads do not block. The time complexity of Stage 2 with IST using p GPUs may be formalized based on the time spent in the recomputation lanes. First we must note that Stage 2 speculations start as soon as Stage 1 finishes in each GPU, and this happens in different times because of the wavefront propagation. Let T_1^i be the Stage 1 execution time in GPU_i, defined by the equation $T_1^i = T_1^1 + (i-1) \cdot t_0$ where t_0 is the time shift between two successive GPUs caused by the wavefront initialization. Let l_i be the length of the recomputation lane that starts at GPU_i and ends at GPU_{i-l_i}. Each Stage 2 recalculation spends t_2 seconds. So, the recomputation lane that begins in GPU_i executes in the time range starting at T_1^i and ending at $T_1^i + l_i \cdot t_2$.

Let T_2 be the additional time executed by Stage 2 in all p GPUs after the last GPU finished executing Stage 1. T_2 depends on both the length l_i and the start position of each recomputation lane, considering that a long lane that starts early may end before a short lane that starts late. Thus, T_2 is defined by Equation (3).

$$\begin{aligned} T_2 &= \max_{1 \le i \le p}(T_1^i + l_i \cdot t_2) - T_1^p \\ &= \max_{1 \le i \le p}((i-p) \cdot t_0 + l_i \cdot t_2). \end{aligned} \tag{3}$$

If t_2 is $O(\frac{mn}{p^2})$, the time complexity of Stage 2 is $O(l\,\frac{mn}{p^2})$ where $l\,(1 \le l \le p)$ depends on the length and position of the recomputation lanes. In the best case, all the speculations are correct. So $l = 1$ and the time complexity is $O(\frac{mn}{p^2})$. In the worst case, all speculations are wrong. Thus $l = p$ and the time complexity is $O(\frac{mn}{p})$, which is the time complexity of PT. In order to estimate l, we observed that the recomputation lanes are usually small, but they can be large in two circumstances. The first case happens in regions with high occurrences of mismatches and gaps. Small variations in the alignment of this region may lead to higher number of matches and greedily produce locally better scores. These variations will force the alignment to follow a further path that will introduce much more penalties, leading to a lower score in the far end of the alignment. The second case occurs when there is a repetition of short DNA patterns, either caused by tandem-repeats or repetitions of N's characters due to unsequenced regions in the genome assembly (which tends to be reduced in each assembly release). The repeated pattern may increase the score in many different positions, reducing probability of hit.

Pseudocode. The pseudocode for the IST is presented in Algorithm 3. Without loss of generality, we assume that the optimal alignment position found by Stage 1 is in the last GPU. For the full IST algorithm, lines 5 to 25 of Algorithm 3 replace lines 5 to 12 of Algorithm 2. In Algorithm 3, the best score of the last column calculated by each GPU, with the exception of the last GPU, is used as the first speculated crosspoint (line 9). Further, the speculation loop (lines 12 to 25) computes Stage 2 with different crosspoint coordinates until a not speculated crosspoint is received: meaning that the $actual\ crosspoint$ from the optimal alignment has already been found and it is being used.

During the speculation phase, if the speculated crosspoint was not previously computed, Stage 2 is executed (line 3) and results are cached (line 15), otherwise results are obtained from the cache (line 17). The top-left crosspoint $list_2[0]$ (line 19) computed in Stage 2 is sent to the left-neighbor GPU (line 21) and a new speculated crosspoint is received from the GPU to the right (line 2).

The optimal alignment starts in the last GPU, which uses the optimal score as the first crosspoint (line 6) and defines it as a not speculated crosspoint (line 7). Whenever a not speculated crosspoint is found, the next top-left crosspoint is marked with the same tag (line 20), which is cascaded through all the GPUs and finalizes the execution loop (line 25). The loop ends for each GPU when Stage 2 has been computed for the optimal alignment, and Stages 3 and 4 are pipelined for computation after Stage 2 as discussed in the PT algorithm.

Algorithm 3 was simplified for presentation purpose. In the actual implementation, there is a limitation in the number of times (max_spec_tries) that Stage 2 is executed speculatively. Therefore, when the $Cache$ size reaches max_spec_tries, IST will wait for the actual crosspoint in order to either retrieve the results from cache or re-execute Stage 2 in that GPU.

6 Experimental Results

The experiments were carried out using the XSEDE Keeneland Full Scale (KFS) system, which is a cluster with 264 HP SL250G8 compute nodes. Each node is equipped with two eight-core Intel Sandy Bridge processors, three NVIDIA M2090 GPUs, and 32 GB of RAM. The nodes communicate through a Mellanox FDR InfiniBand interconnect and are attached to a Lustre distributed filesystem. The CUDAlign

TABLE 2
Real Sequences Used in the Tests

Chr.	Human (GRCh37)		Chimp. (panTro4)		Peta Cells	Score	Length	Coverage	Matches	Mismt.	Gaps
	Accession	Size	Accession	Size							
chr01	NC_000001.10	249 M	NC_006468.3	228 M	56.91	84,608,525	255,117,470	99,1%	80,1%	5,3%	14,6%
chr02-A	NC_000002.11	243 M	NC_006469.3	114 M	27.63	74,861,783	118,554,635	73,4%	89,0%	3,1%	7,9%
chr02-B	NC_000002.11	243 M	NC_006470.3	248 M	60.20	93,139,254	135,655,977	53,2%	90,5%	2,1%	7,4%
chr03	NC_000003.11	198 M	NC_006490.3	202 M	40.07	152,598,201	205,950,608	99,9%	92,2%	2,0%	5,8%
chr04	NC_000004.11	191 M	NC_006471.3	193 M	36.99	81,532,618	107,996,353	54,6%	92,7%	1,9%	5,4%
chr05	NC_000005.9	180 M	NC_006472.3	183 M	33.04	63,924,833	87,400,843	46,9%	92,2%	2,7%	5,1%
chr06	NC_000006.11	171 M	NC_006473.3	173 M	29.54	120,465,367	176,613,042	99,4%	90,6%	2,9%	6,5%
chr07	NC_000007.13	159 M	NC_006474.3	162 M	25.75	93,144,399	164,099,089	97,8%	87,8%	3,4%	8,8%
chr08	NC_000008.10	146 M	NC_006475.3	144 M	21.07	101,825,467	138,467,654	92,7%	92,2%	2,2%	5,6%
chr09	NC_000009.11	141 M	NC_006476.3	138 M	19.46	76,043,976	145,206,895	99,8%	86,3%	5,6%	8,1%
chr10	NC_000010.10	136 M	NC_006477.3	134 M	18.10	80,128,465	141,139,131	99,9%	87,1%	3,4%	9,6%
chr11	NC_000011.9	135 M	NC_006478.3	133 M	17.97	80,183,754	138,964,666	99,7%	87,7%	4,7%	7,6%
chr12	NC_000012.11	134 M	NC_006479.3	134 M	17.97	49,076,981	67,897,406	49,2%	91,8%	2,5%	5,7%
chr13	NC_000013.10	115 M	NC_006480.3	115 M	13.26	64,071,638	116,494,539	99,0%	87,8%	7,9%	4,3%
chr14	NC_000014.8	107 M	NC_006481.3	107 M	11.44	82,247,932	108,107,078	98,4%	92,9%	1,8%	5,3%
chr15	NC_000015.9	103 M	NC_006482.3	100 M	10.21	64,957,513	103,625,609	99,1%	89,3%	3,9%	6,8%
chr16	NC_000016.9	90 M	NC_006483.3	90 M	8.13	45,421,118	95,068,007	99,8%	84,5%	4,8%	10,7%
chr17	NC_000017.10	81 M	NC_006484.3	83 M	6.71	22,218,058	35,392,160	41,5%	88,9%	3,1%	8,0%
chr18	NC_000018.9	78 M	NC_006485.3	77 M	5.98	46,959,759	61,253,576	77,3%	93,1%	2,0%	4,9%
chr19	NC_000019.9	59 M	NC_006486.3	64 M	3.76	17,297,608	36,386,342	56,2%	84,8%	4,6%	10,6%
chr20	NC_000020.10	63 M	NC_006487.3	62 M	3.89	40,050,427	65,286,930	99,9%	88,2%	2,6%	9,2%
chr21	NC_000021.8	48 M	NC_006488.2	46 M	2.24	36,006,054	48,579,349	99,0%	91,9%	1,1%	7,1%
chr22	NC_000022.10	51 M	NC_006489.3	50 M	2.55	31,510,791	51,929,087	98,9%	88,5%	3,8%	7,7%
chrX	NC_000023.10	155 M	NC_006491.3	157 M	24.35	59,862,909	157,365,502	95,4%	82,1%	7,1%	10,8%
chrY	NC_000024.9	59 M	NC_006492.3	26 M	1.56	1,394,673	2,283,191	6,0%	88,1%	2,0%	10,0%

implementation used standard C++, CUDA 5.5, Pthreads, and Sockets. CUDAlign 4.0 *kernels* were launched using $B = 64$ CUDA blocks, $T = 128$ threads in each block, and each thread processes $\alpha = 4$ rows of the matrix. We used 8 MB local communication buffers, which is sufficient to hold up to 1 million cells. The SW score parameters used were: match: $+1$; mismatch -3; first gap: -5; extension gap: -2.

Algorithm 3. Speculative Traceback (nth GPU)

```
 4: ...
 5: if n is last GPU then
 6:    crosspoint ← best_n
 7:    crosspoint.speculated ← false
 8: else
 9:    crosspoint ← best score in last column
10:    crosspoint.speculated ← true
11: end if
12: repeat
13:    if Cache[crosspoint] is empty then
14:       list_2 ← Stage2(crosspoint)
15:       Cache[crosspoint] ← list_2
16:    else
17:       list_2 ← Cache[crosspoint]
18:    end if
19:    left_crosspoint ← list_2[0]
20:    left_crosspoint.speculated ← crosspoint.speculated
21:    Send_{n-1}(left_crosspoint)
22:    if crosspoint.speculated then
23:       Recv_{n+1}(crosspoint)
24:    end if
25: until crosspoint.speculated is false
26: ...
```

We intended to compare the results of CUDAlign 4.0 with a state-of-the-art tool. In order to do that, we downloaded SW# [10], which compares long sequences in GPUs (Section 3) and is freely available. In our tests, we confirmed that SW# executes nucleotide comparisons with up to two GPUs, which would lead to excessively long execution times for the long chromosome comparisons.

6.1 Sequences Used in the Tests

Our experiments used real DNA sequences retrieved from the National Center for Biotechnology Information (NCBI), available at *www.ncbi.nlm.nih.gov*. We compared all human (GRCh37) and chimpanzee (panTro4) homologous chromosomes, which produces a test set with 25 pairs of sequences.

The accession numbers and sizes of the chromosomes are presented in Table 2. They vary from 26 MBP to 249 MBP, and produce DP matrices from 1.56 to 56.91 Peta Cells. For the sake of validation, optimal local scores, alignment length, coverage, percentage of matches and mismatches, and gaps obtained are also presented in the same table. In this first moment, we compared the chromosomes with the purpose of obtaining the optimal alignments, using a variable number of GPUs. The optimal local alignments are presented in Fig. 10, where we can see that many comparisons produced huge local alignments.

6.2 Performance of Pipelined Traceback

The performance and scalability of the baseline PT (Section 5.2.1) was evaluated using sequences chr22 and chr16. The number of nodes was varied from 1 to 128 (3 to 384 GPUs). Table 3 presents the experimental results for the entire execution (Total), which are also broken into the time

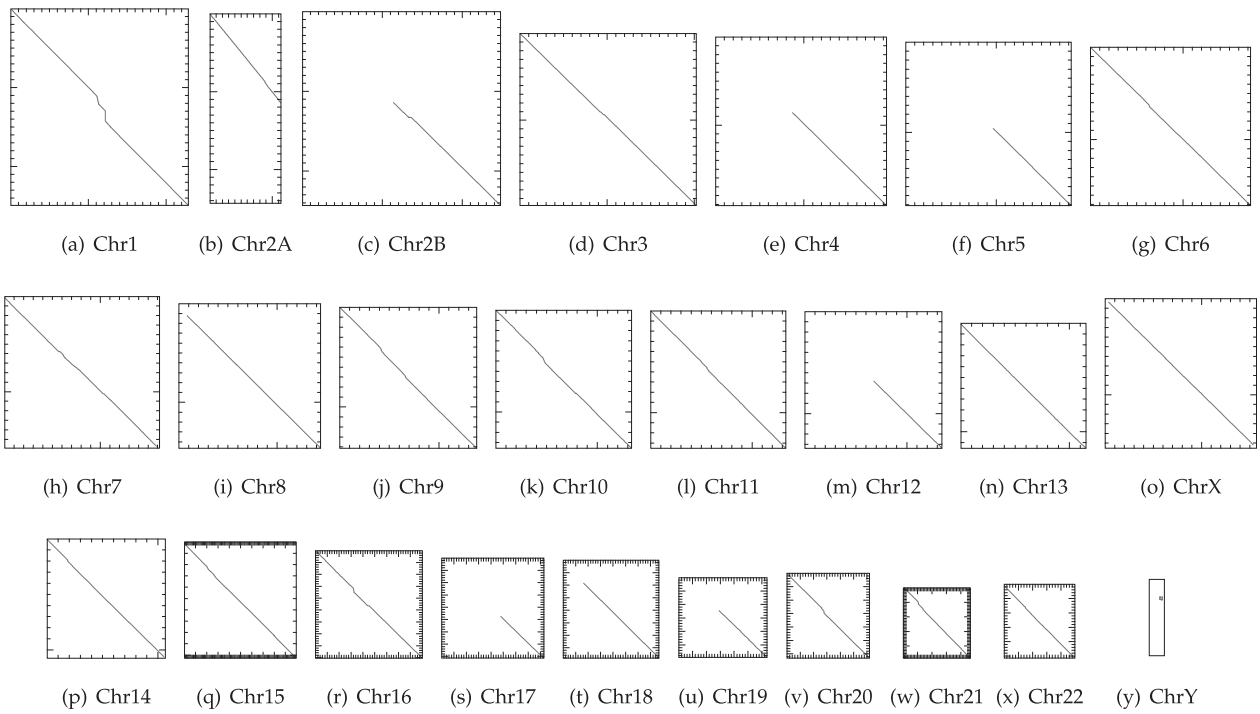


Fig. 10. Alignment plots between human and chimpanzee homologous chromosomes.

spent in Stage 1 and the remaining stages (Traceback). As shown, the speedups attained with 128 nodes for chr22 and chr16 were, respectively, 26.9× and 29.7× (21.0 and 23.2 percent of parallel efficiency).

The breakdown of the total execution shows that the Stage 1 of CUDAlign has a much better scalability. Stage 1 attained speedups of 84.0× and 97.3× with 128 nodes (65.6 and 76.0 percent of parallel efficiency), resulting in a peak performance of 8.3 and 9.7 TCUPS for chr22 and chr16, respectively. Stage 1 results of chr22 and chr16 are consistent with the ones obtained in CUDAlign 3.0 [12]. The PT traceback phase, on the other hand, was not able to efficiently use the parallel environment and, as a consequence, limited the scalability of the whole application. For instance, the percentage of the execution time spent in the traceback phase increased from about 4 to 71 percent as the number of nodes used was scaled from 1 to 128. This negative impact of the traceback to the whole application performance is highly reduced when IST is used, as shown in Section 6.3.

TABLE 3
Chr22 and Chr16 Execution Times (PT only)

	Nodes /GPUs	Total Time	Total Spd.	Stage 1 Time	Stage 1 Spd.	Traceback Time
chr22	1/3	26,660 s	1.0x	25,809 s	1.0x	851 s
	2/6	13,986 s	1.9x	13,220 s	2.0x	766 s
	4/12	7,342 s	3.6x	6,617 s	3.9x	725 s
	8/24	4,002 s	6.7x	3,348 s	7.7x	654 s
	16/48	2,362 s	11.3x	1,716 s	15.0x	647 s
	32/96	1,534 s	17.4x	925 s	27.9x	610 s
	64/192	1,092 s	24.4x	507 s	50.9x	585 s
	128/384	993 s	**26.9x**	307 s	**84.0x**	686 s
chr16	1/3	85,173 s	1.0x	81,543 s	1.0x	3,629 s
	2/6	44,403 s	1.9x	41,139 s	2.0x	3,265 s
	4/12	24,634 s	3.5x	21,397 s	3.8x	3,237 s
	8/24	13,767 s	6.2x	10,688 s	7.6x	3,079 s
	16/48	8,192 s	10.4x	5,404 s	15.1x	2,788 s
	32/96	5,377 s	15.8x	2,780 s	29.3x	2,597 s
	64/192	3,693 s	23.1x	1,477 s	55.2x	2,216 s
	128/384	2,870 s	**29.7x**	838 s	**97.3x**	2,033 s

6.3 Impact of Incremental Speculative Traceback

The experimental evaluation of the impact of IST to the performance was carried out using five pairs of homologous chromosomes: chr22, chr16, chr13, chr8, and chr5. These sequences were selected intending to provide a wide range of variation in the DP matrix size calculated (2.55, 8.13, 13.26, 21.07, 33.04 Peta cells, respectively).

The execution times and TCUPS for all stages, Stage 1, and Stage 2–6 (traceback) are presented in Table 4 for IST and PT. As shown, we have attained up to 10.37 TCUPS for all stages and 11.08 TCUPS if only Stage 1 is considered. Additionally, higher TCPUS are achieved when longer sequences are used. PT and IST execute the same code in Stage 1 and, as such, the execution times are very similar and most of the cases have a difference of up to about 1 percent, except for some cases highlighted in Section 6.5.

The speculation hits of the IST strategy for comparisons chr22, chr16, chr13, chr8 and chr5 (with 128 nodes) were 69.2, 88.3, 79.4, 99.7 and 98.2 percent respectively. Considering the high speculation effectiveness, the results show that IST was able to significantly improve the performance of PT for all sequences and number of nodes used. The performance improvement of IST over PT for the traceback phase is also presented in Table 4. IST improved this phase in 2.15–2.81×, 4.35–5.03×, 3.85–7.66× for comparisons using, respectively, chr22, chr16, and chr13. Additionally, the traceback gains with IST for chr8 and chr5 with 128 nodes were of 18.30× and 21.03×, respectively. Regarding the overall execution time, IST was up to 2.93× faster than PT for chr8 with 128 nodes. It must be noted that the speculations do not introduce noticeable

TABLE 4
PT and IST Execution Times

	Cmp.		Total Time (TCUPS)	Stage 1 Time (TCUPS)	Traceback Time (PT/IST)	
16 nodes (48 GPUs)	chr22	PT	2362s (1.08)	1716s(1.49)	647s	2.15x
		IST	2001s (1.27)	1701s(1.50)	301s	
	chr16	PT	8192s (0.99)	5404s(1.50)	2788s	4.35x
		IST	6063s (1.34)	5422s(1.50)	641s	
	chr13	PT	12825s (1.03)	8921s(1.49)	3904s	7.66x
		IST	9375s (1.41)	8865s(1.50)	510s	
32 nodes (96 GPUs)	chr22	PT	1534s (1.66)	925s(2.76)	610s	2.71x
		IST	1139s (2.24)	914s(2.79)	225s	
	chr16	PT	5377s (1.51)	2780s(2.93)	2597s	4.60x
		IST	3323s (2.45)	2758s(2.95)	565s	
	chr13	PT	8113s (1.63)	4536s(2.92)	3577s	4.90x
		IST	5212s (2.54)	4482s(2.96)	730s	
64 nodes (192 GPUs)	chr22	PT	1092s (2.34)	507s(5.03)	585s	2.81x
		IST	766s (3.33)	558s(4.57)	208s	
	chr16	PT	3693s (2.20)	1477s(5.50)	2216s	5.03x
		IST	1914s (4.25)	1473s(5.52)	441s	
	chr13	PT	5390s (2.46)	2364s(5.61)	3026s	3.86x
		IST	3237s (4.10)	2453s(5.41)	785s	
128 nodes (384 GPUs)	chr22	PT	993s (2.57)	307s(8.31)	686s	2.61x
		IST	569s (4.49)	306s(8.33)	263s	
	chr16	PT	2870s (2.83)	838s(9.70)	2033s	4.94x
		IST	1305s (6.23)	894s(9.10)	412s	
	chr13	PT	4176s (3.18)	1427s(9.29)	2749s	3.85x
		IST	2126s (6.24)	1412s(9.39)	715s	
	chr8	PT	6515s (3.24)	2020s(10.43)	4495s	18.30x
		IST	2219s (9.50)	1973s(10.68)	246s	
	chr5	PT	6490s (5.09)	2982s(**11.08**)	3508s	**21.03x**
		IST	3188s (**10.37**)	3021s(10.94)	167s	

overhead since IST executes during the time the GPU would otherwise be idle.

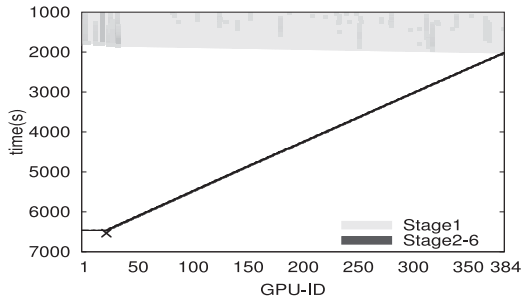
The performance improvement variations are result of the IST ability to correctly speculate, which depends on the characteristics of the alignments. The next section presents a detailed evaluation of the impact of these characteristics to the performance.

6.4 Effect of Alignment Characteristics to IST

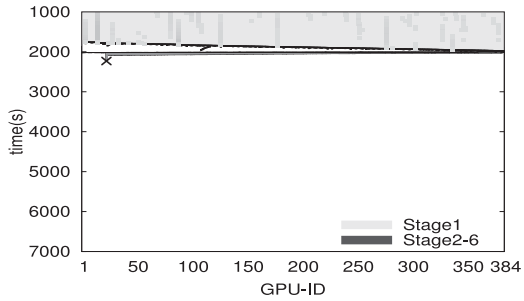
To evaluate the impact of the alignment characteristics to the IST performance we have created timelines of the executions for chr8 and chr16 with 128 nodes (Figs. 11 and 12). The light gray area on the top of the plots represents Stage 1 execution, while diagonal recomputation lanes pointing to the bottom-left corner refer to the traceback phase (Stages 2 to 5). In the PT timelines, the white area between the Stage 1 and the traceback are idle times as a consequence of dependencies in Stage 2. In the IST timelines, the speculated phase of Stage 2 presents black diagonal lines right after Stage 1. In both cases, a cross ($\times$) marks the total execution time.

The tracebacks with PT (Figs. 11a and 12a) are presented as diagonal lines to the bottom-left corner. Here we can clearly notice the large amount of idle time between Stages 1 and 2. Stages 3–6 are pipelined and they are usually executed in a very short time.

In tracebacks with IST (Figs. 11b and 12b) we can notice a different behavior, where the speculated Stage 2 executions are presented as a diagonal line right after the end of Stage 1. From this line, there are other lines that go in the bottom-left direction, representing the recomputation lanes due to IST. Some of these lanes span over many GPUs, as can be seen in the central GPUs of chr16 (Fig. 12b). In chr8, the speculative prediction was very efficient (Fig. 11b), producing a



(a) Chr8 - PT

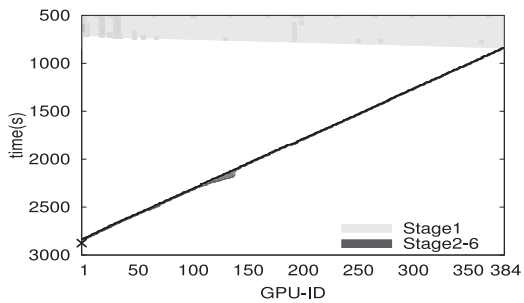


(b) Chr8 - IST

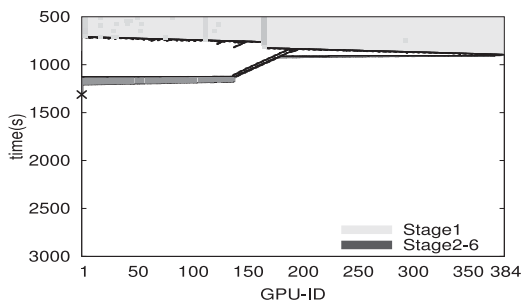
Fig. 11. Stage timelines for PT and IST - Chr8.

traceback path almost straight to the left and reducing the traceback time from 4495 s to 246 s (18.30$\times$).

In chr16 with PT (Fig. 12a) we noticed a higher elapsed time in Stages 3–6 between GPUs 110 and 140. This occurred because there is a large gap region around GPU 140, and this produced disproportional partitions that required more time in Stage 4. This additional time was propagated to other GPUs (110-139) due to the stage 4 communication path (line 15 in Algorithm 2). The same behavior happened in chr16 with IST (Fig. 12b), but the communication spanned more GPUs (1–139) because all these GPUs precomputed



(a) Chr16 - PT



(b) Chr16 - IST

Fig. 12. Stage timelines for PT and IST - Chr16.

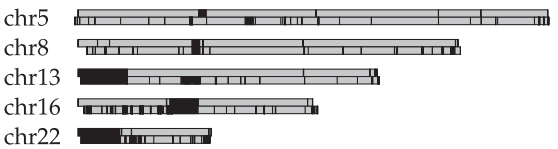


Fig. 13. Unsequenced regions (N-regions) in homologous chromosomes (human above, chimpanzee below).

the partitions successfully during the Stage 2 speculation phase, and, consequently, all these GPUs needed to wait for the Stage 4 results calculated by GPU 140.

As stated in Section 5.2.2, one reason to occur a recomputation lane is the existence of regions with very high incidence of N's, caused by unsequenced regions in the genomic assembly. Fig. 13 illustrates the sizes and positions of the N-regions (in black) in each chromosome. The length and coverage percentage of the largest N-regions in the human chr22, chr16, chr13, chr8 and chr5 are, respectively, 16 MBP (31 percent), 11 MBP (12 percent), 19 MBP (16 percent), 3 MBP (2.1 percent) and 3 MBP (1.7 percent). We observed that the percentages of N-regions and their locations give a very good prediction of the traceback speedup for the IST method (Table 4). For instance, the chr22 alignment has the largest N-region (in percentage) and it presented the smallest traceback speedups (below 2.81×). On the other hand, chr8 and chr5 alignments have the smallest N-regions, presenting higher speedups (18.30 and 21.03×, respectively).

6.5 Buffer Analysis of Stage 1

In Figs. 11 and 12, we have also represented the output buffer usage in Stage 1 (Section 5.1.2) using different levels of dark gray columns, which indicate an unbalanced usage of the buffer. When a GPU is slower than its left-neighbor GPU, its buffer starts to be filled until it eventually becomes full, cascading this slow down to previous GPUs. For instance, in chr16 with IST (Fig. 12b) there are dark gray lines close to the middle GPUs indicating an unbalanced performance in that GPU. This behavior produced a Stage 1 time difference of 6 percent (55 s) when comparing PT and IST executions (Table 4).

IST and PT execute the same code in Stage 1 and these performance differences are not related to the traceback method. This unbalancing may be the result of the following factors: 1) very similar sequence regions, leading to frequent updates in the current best scores, causing a small overhead due to additional instructions executed in the GPU *kernel*; 2) concurrent resource usage with other jobs in the cluster, which may slow down the file system performance or increase the network latency. The first factor is meant to be reproducible between runs with the same sequences, but the second factor is usually irreproducible. Additionally, the second factor may produce long-term impact in the wavefront balancing, considering that job concurrency is systemic. With these considerations and analyzing the timelines, we may infer that most of the unbalancing scenarios were caused by irreproducible factors due to intra-cluster concurrency.

7 Conclusion and Future Works

This paper presented CUDAlign 4.0, a multi-GPU algorithm able to compute the optimal alignment of huge DNA sequences. CUDAlign 4.0 was tested in the KFS system, a cluster with 264 homogeneous hosts with three GPUs each. We were able to pairwise align the human × chimpanzee homologous chromosomes using the SW algorithm. As far as we know, there is no other implementation in the literature that is able to execute such chromosome comparison using an exact algorithm.

The sizes of the sequences used in the tests varied from 26 MBP up to 249 MBP and the tests computed DP matrices from 1.56 up to 60.20 petacells. In KFS, CUDAlign 4.0 presented results up to 10.37 TCUPS when using 384 GPUs. To our knowledge, this is the highest CUPS rate obtained for SW executions. By incrementally speculating the points that belong to the optimal alignment, the proposed IST strategy was able to to improve 2.15× up to 21.03× the traceback execution time, when compared to the baseline strategy (PT). For instance, for the chr5 comparison with 384 GPUs, the traceback time was reduced from 3508 s down to 167 s.

This paper did not intend to give detailed biological analysis of the chromosomes, but it showed that CUDAlign 4.0 is able to produce the optimal alignment even for the largest human chromosomes. With CUDAlign 4.0, the use of exact methods is now feasible for whole chromosome alignment with huge sequences.

As future work, we intend to integrate our Block Pruning method [11] to the Stage 1 of CUDAlign 4.0, which may significantly reduce the execution time in this phase. We also want to produce sub-optimal alignments to increase the alignment coverage of homologous chromosomes, as well as to produce alignments in the reverse complement strands. Additionally, the static column distribution presented in this paper is suited for dedicated environments, but it may be improved with dynamic load balancing [31] for non-dedicated environments.

Acknowledgments

The authors would like to thanks the support of the grant SEV-2015-0493 of Severo Ochoa Program, awarded by the Spanish Government, PHBP14/00081(Programa Hispano-Brasileno) of the Spanish Ministry of Education, Culture and Sport, TIN2015-65316 of the Spanish Ministry of Science and Technology and 2014-SGR1051 of the Generalitat de Catalunya. This work is also partially supported by CNPq/Brazil (grants 242800/2012-2, 313931/2013-5, 211456/2013-6, 305208/2014-4 and 446297/2014-3). The authors would also like to thank the Keeneland team for letting us use their GPU cluster.

References

[1] D. W. Mount, *Bioinformatics: Sequence and Genome Analysis*. Cold Spring Harbor, NY, USA: CSHL Press, 2004.

[2] T. F. Smith and M. S. Waterman, "Identification of common molecular subsequences," *J. Mol. Biol.*, vol. 147, no. 1, pp. 195–197, 1981.

[3] S. Rajko and S. Aluru, "Space and time optimal parallel sequence alignments," *IEEE Trans. Parallel Distrib. Syst.*, vol. 15, no. 12, pp. 1070–1081, Dec. 2004.

[4] K. Hamidouche, et al., "Parallel smith-waterman comparison on multicore and manycore computing platforms with BSP++," *Int. J. Parallel Program.*, vol. 41, no. 1, pp. 111–136, 2013.

[5] A. Sarje and S. Aluru, "Parallel genomic alignments on the cell broadband engine," *IEEE Trans. Parallel Distrib. Syst.*, vol. 20, no. 11, pp. 1600–1610, Nov. 2009.

[6] L. Wienbrandt, "The FPGA-based high-performance computer RIVYERA for applications in bioinformatics," in *Language, Life, Limits: 10th CiE*, New York, NY, USA: Springer, 2014, pp. 383–392.
[7] S. Sarkar, G. R. Kulkarni, and P. P. Pande, "Network-on-chip hardware accelerators for biological sequence alignment," *IEEE Trans. Comput.*, vol. 59, no. 1, pp. 29–41, Jan. 2010.
[8] Y. Liu, T. Tam, F. Lauenroth, and B. Schmidt, "SWAPHI-LS: Smith-Waterman algorithm on Xeon Phi coprocessors for long DNA sequences," in *Proc. IEEE Int. Conf. Cluster Comput.*, 2014, pp. 257–265.
[9] F. Ino, Y. Munekawa, and K. Hagihara, "Sequence homology search using fine grained cycle sharing of idle GPUs," *IEEE Trans. Parallel Distrib. Syst.*, vol. 23, no. 4, pp. 751–759, Apr. 2012.
[10] M. Korpar and M. Sikic, "Sw#-gpu-enabled exact alignments on genome scale," *Bioinformatics*, vol. 29, no. 19, pp. 2494–2495, 2013.
[11] E. F. O. Sandes and A. C. M. A. Melo, "Retrieving smith-waterman alignments with optimizations for megabase biological sequences using GPU," *IEEE Trans. Par. Dist. Syst.*, vol. 24, no. 5, pp. 1009–1021, May 2013.
[12] E. F. O. Sandes, G. Miranda, A. C. M. A. Melo, X. Martorell, and E. Ayguad, "Cudalign 3.0: Parallel biological sequence comparison in large GPU clusters," in *Proc. 14th IEEE/ACM Int. Symp. Cluster, Cloud Grid Comput.*, 2014, pp. 160–169.
[13] A. Altenhoff, et al., "The OMA orthology database in 2015: Function predictions, better plant support, synteny view and other improvements," *Nucleic Acids Res.*, vol. 43, no. D1, pp. 240–249, 2015.
[14] M. Forster, et al., "Vy-PER: Eliminating false positive detection of virus integration events in next generation sequencing data," *Nature Sci. Rep.*, vol. 5, pp. 1–13, Jul. 2015.
[15] M. Brudno, et al."Lagan and multi-lagan: Efficient tools for large-scale multiple alignment of genomic DNA," *Genome Res.*, vol. 13, pp. 721–731, 2003.
[16] R. Uricaru, C. Michotey, H. Chiapello, and E. Rivals, "Yoc, a new strategy for pairwise alignment of collinear genomes," *BMC Bioinformatics*, vol. 16, pp. 1–16, 2015.
[17] K. Gao and J. Miller, "Human-chimpanzee alignment: Ortholog exponentials and paralog power laws," *Comput. Biol. Chem.*, vol. 53, pp. 59–70, 2014.
[18] B. Kehr, D. Weese, and K. Reinert, "Stellar: Fast and exact local alignments," *BMC Bioinformatics*, vol. 12, pp. 1–12, 2011.
[19] E. F. O. Sandes and A. C. M. A. Melo, "CUDAlign: Using GPU to accelerate the comparison of megabase genomic sequences," in *Proc. 15th ACM SIGPLAN Symp. Principles Practice Parallel Programm.*, 2010, pp. 137–146.
[20] E. F. O. Sandes and A. C. M. A. Melo, "Smith–Waterman alignment of huge sequences with GPU in linear space," in *Proc. IEEE Int. Symp. Parallel Distrib. Process.*, 2011, pp. 1199–1211.
[21] 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 Res. Notes*, vol. 3, no. 1, p. 93, 2010.
[22] Y. Liu, A. Wirawan, and B. Schmidt, "CUDASW++ 3.0: Accelerating smith-waterman protein database search by coupling cpu and GPU simd instructions," *BMC Bioinformat.*, vol. 14, p. 117, 2013.
[23] E. F. Sandes, G. Miranda, A. Melo, X. Martorell, and E. Ayguade, "Fine-grain parallel megabase sequence comparison with multiple heterogeneous GPUs," in *Proc. ACM SIGPLAN Symp. Principles Practice Parallel Programm.*, 2014, pp. 383–384.
[24] E. W. Myers and W. Miller, "Optimal alignments in linear space," *Comput. Appl. Biosci.*, vol. 4, no. 1, pp. 11–17, 1988.
[25] O. Gotoh, "An improved algorithm for matching biological sequences," *J. Mol. Biol.*, vol. 162, no. 3, pp. 705–708, Dec. 1982.
[26] D. S. Hirschberg, "A linear space algorithm for computing maximal common subsequences," *Commun. ACM*, vol. 18, no. 6, pp. 341–343, 1975.
[27] G. F. Pfister, *In Search of Clusters: The Coming Battle in Lowly Parallel Computing*. Englewood Cliffs, NJ, USA: Prentice-Hall, Inc., 1995.
[28] J. Blazewicz, et al., "Protein alignment algorithms with an efficient backtracking routine on multiple GPUs," *BMC Bioinformat.*, vol. 12, no. 1, p. 181, 2011.
[29] C. Dicker, S. Kelly, J. Sibandze, and J. Mache, "Viability of the parallel prefix algorithm for sequence alignment on massively parallel GPUs," in *Proc. Int. Conf. Parallel Distrib. Process. Techn. Appl.*, 2014, pp. 655–658.
[30] NVIDIA. CUDA Programming Guide 6.0, 2014.
[31] E. F. Sandes, C. G. Ralha, and A. C. Melo, "An agent-based solution for dynamic multi-node wavefront balancing in biological sequence comparison," *Exp. Syst. Appl.*, vol. 41, no. 10, pp. 4929–4938, 2014.

Edans Flavius de Oliveira Sandes received the BSc degree in computer science in 2006, the MSc degree in informatics in June 2011, and the PhD degree in informatics in September 2015, all from the University of Brasilia, Brazil. His research interests include computational biology, GPGPUs, and load balancing.

Guillermo Miranda received the MSc degree in computer science from the Universitat Pompeu Fabra in July 2006. He was a Research Support Engineer in the Barcelona Supercomputing Center for four years. His research interests cover task-based parallelization, scheduling, and NUMA architectures.

Xavier Martorell received the M.S. and Ph.D. degrees in computer science from the Technical University of Catalunya (UPC) in 1991 and 1999, respectively. He has been an associate professor in the Computer Architecture Department, UPC, since 2001. His research interests include the areas of operating systems, runtime systems, compilers, and applications for high-performance multiprocessor systems. He is leading the Parallel Programming Models team at the Barcelona Supercomputing Center.

Eduard Ayguade received the engineering degree in telecommunications and the PhD degree in computer science, both from the Universitat Politecnica de Catalunya (UPC), Spain, in 1986 and 1989, respectively. Since 1997, he has been a full professor of the Department of Computer Architecture, UPC. His research interests include processor microarchitecture, multicore architectures, and programming models and their architectural support. He has published more than 100 papers and participated in several research projects in the framework of the European Union and research collaborations with companies. He is the associate director for research on Computer Sciences at the Barcelona Supercomputing Center.

George Teodoro received the MS and PhD degrees in computer science from the Universidade Federal de Minas Gerais (UFMG), Brazil, in 2006 and 2010. He is currently an assistant professor in the Computer Science Department, University of Brasilia (UnB), Brazil. His primary areas of expertise include high-performance runtime systems for efficient execution of biomedical and data-mining applications on distributed heterogeneous environments.

Alba Cristina Magalhaes Melo received the PhD degree in computer science from the Institut National Polytechnique de Grenoble (INPG), France, in 1996. She is currently an associate professor in the Department of Computer Science, University of Brasilia, Brazil. Her research interests include accelerators, cluster computing, cloud computing, and bioinformatics. She is a senior member of the IEEE and IEEE Computer Society.