Parallel Exact Spliced Alignment on GPU

Anisio Nolasco Nahri Moreano
School of Computing
Federal University of Mato Grosso do Sul, Brazil
Email: anisio.nolasco@gmail.com, nahri@facom.ufms.br

***Abstract*—Due to the exponential growth of biological DNA sequence databases, some parallel gene prediction solutions on different high performance platforms have been proposed. Nevertheless, few exact parallel solutions to the spliced alignment problem to gene prediction in eukaryotic organisms have been proposed and none of these solutions use GPUs as the target platform. In this paper, we present the development of two GPU accelerators for an exact solution to the spliced alignment problem applied to gene prediction. Our main contributions are: (a) the identification of two forms to exploit parallelism in the spliced alignment algorithm; (b) two GPU accelerators that achieve speedups up to 52.62 and 90.86, respectively, when compared to a sequential implementation. The accelerators performance scales with input data size, outperforming related work results; (c) a particular organization for the data structures of the accelerators in order to optimize their efficiency; (d) a potential parallelism analysis of the biological data set with the goal of measuring the amount of parallelism that would in fact be available to be exploited by a parallel implementation; and (e) an accurate performance estimation model that enabled estimating the accelerators performance, before implementing them.**

***Keywords*—Spliced alignment, Gelfand algorithm, parallelism, accelerator, GPU.**

I. Introduction

In the last years, new DNA sequencing technologies have been causing genomic databases to experience a exponential grow rate in their sizes. As a consequence, a huge amount of new genomic data needs to be analyzed, in order to determine their functional content. Gene prediction is one of the most important steps in the process of understanding the genome of an organism. It refers to identifying biologically functional stretches of sequences (genes) in a DNA sequence.

The spliced alignment algorithm to gene prediction in eukaryotic organisms uses a similarity-based approach, where a related protein sequence from an annotated genome is used to reconstruct the exon structure of the genes in the investigated sequence. The high time complexity of the spliced alignment algorithm, which may lead to high execution times, especially in case of huge genomes, motivates the development of high performance solutions to this gene prediction method.

This paper presents two GPU accelerators for an exact solution to the spliced alignment problem applied to gene prediction. We identify two forms of exploiting parallelism in the spliced alignment problem, intra- and inter-exon parallelism. Our first GPU accelerator exploits only the first form, while both forms are exploited by the second one. We propose an organization for the data structures of the accelerators in order to optimize their efficiency. Besides, our accelerators are able to handle input data sets with long sequences. A comprehensive performance evaluation is performed using a large and representative biological data set, in order to assess the accelerators performance gain.

Before implementing the accelerators, we perform a potential parallelism analysis of our biological data set with the goal of measuring the amount of parallelism that would in fact be available to be exploited by a parallel implementation. We also propose a performance estimation model that enables estimating the accelerators performance and evaluating if there will be performance gains, compared to a sequential solution.

To the best of our knowledge, these are the first GPU-based systems proposed for the acceleration of the spliced alignment algorithm. The Gelfand algorithm is difficult to parallelize and there are very few works about parallel solutions to it in the literature.

This paper is organized as follows. Section II introduces the spliced alignment problem and the Gelfand algorithm. In Section III we describe related works in parallel gene prediction and parallel solutions to the spliced alignment problem. In Section IV we analyze the data dependences in the spliced alignment algorithm and identify forms to exploit parallelism in the algorithm execution. Section V presents a potential parallelism analysis of a representative biological data set and propose a performance estimation model for the parallel solutions to the spliced alignment problem. In Section VI we describe our GPU accelerators and their data structure organization. Section VII presents the accelerators performance evaluation and compares them to other solutions. The accuracy of the performance estimation model is also analyzed. Finally, in Section VIII we summarize the results and suggest future works.

II. Gene Prediction and Spliced Alignment

The problem of gene prediction in eukaryotic DNA sequences consists in finding the initial and final positions of the genes of the sequence, and for each gene, to find the initial and final positions of the exons that constitute the gene [1]. Figure 1 shows an example of a DNA sequence, its genes and exons.

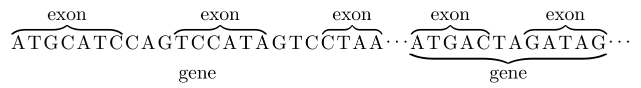

Fig. 1. DNA sequence with genes and exons

Gelfand et al. [2] proposed the spliced alignment algorithm to gene prediction in eukaryotic sequences, using a similarity-based approach: a related protein sequence from an annotated genome is used to reconstruct the exon structure of the genes in the investigated sequence. The method starts with a set of candidate exons of the investigated sequence, which may contain many false exons, but certainly contains all the true ones. Then, using a target DNA sequence (derived from the related protein sequence), it finds the subset of the candidate exons whose concatenation best aligns to the target.

Given a base sequence $s = s_1...s_m$ to be investigated, with a set β of candidate exons, and a target sequence $t = t_1...t_n$, the spliced alignment problem consists in finding the subset of β with the highest similarity score to t. Figure 2 illustrates a spliced alignment, where $\beta = \{b_1, b_2, b_3, b_4, b_5\}$ is the set of candidate exons of s and the subset of β with highest similarity to t is $\{b_1, b_4, b_5\}$.

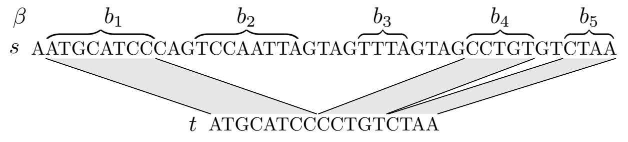

Fig. 2. Spliced alignment between base sequence s, with the set β of candidate exons, and target sequence t

The number of different subsets of β may be huge, but the spliced alignment algorithm is able to find the best alignment among all of them in polynomial time, using a dynamic programming strategy. The similarity scores are kept in a three-dimensional structure S, where $S(i, j, k)$ is the similarity score of the optimal spliced alignment between the i-prefix of s and the j-prefix of t until exon b_k.

For simplicity, a linear gap penalty $-\epsilon$ is used for insertion or deletion. A scoring matrix $\delta(u, v)$ is used for matches and mismatches of symbols. Let $E(k)$ be the index of the last symbol of exon b_k in s.

The recurrence in Equations 1 and 2 compute $S(i, j, k)$. If s_i is not the first symbol of exon b_k we have:

$$S(i,j,k) = \max\begin{cases} S(i-1, j-1, k) + \delta(s_i, t_j) \\ S(i-1, j, k) - \epsilon \\ S(i, j-1, k) - \epsilon \end{cases} \tag{1}$$

On the other hand, if s_i is the first symbol of exon b_k, then:

$$S(i,j,k) = \max\begin{cases} \max_{\forall b_l \text{ preceding } b_k} S(E(l), j-1, l) + \delta(s_i, t_j) \\ \max_{\forall b_l \text{ preceding } b_k} S(E(l), j, l) - \epsilon \\ S(i, j-1, k) - \epsilon \end{cases} \tag{2}$$

After computing S, the similarity score of the optimal spliced alignment is $\max_{1 \leq k \leq |\beta|} S(E(k), n, k)$. The time complexity of the splice alignment algorithm is $O(n \times (\sum_{k=1}^{|\beta|} |b_k|) + n \times |\beta|^2)$.

III. Related Work

Some works in parallel gene prediction have been presented in the literature. However, most of them are based on statistical approaches to gene prediction and/or applied to prokaryotic organisms. Gene prediction methods for eukaryotic organisms tend to be more complex than those for prokaryotes, because the intron-exon structure of a gene prevails in eukaryotes [3].

Differently from similarity-based approaches, which uses a related protein sequence from a previously annotated DNA sequence as a template for the recognition of unknown genes in the investigated DNA sequence, statistical methods look for features that appear frequently in genes and infrequently elsewhere, and rely on detecting statistical variations between coding and non-coding regions of the investigated sequence [1].

The works in [5]–[8] describe parallel solutions to gene prediction using statistical methods. The parallel gene prediction system described in [6] runs on a cluster and combines statistical and homology-based approaches for prokaryotes. A hardware system for gene prediction is proposed in [7], based on a statistical approach for prokaryotes and targeted to a FPGA. A parallel algorithm to gene prediction is presented in [5], using a statistical approach for eukaryotes and running on a cluster. Two accelerators for gene prediction, running on FPGA and GPU, respectively, are presented in [8] and based on a statistical approach for eukaryotes.

Sarje and Aluru [9] present parallel solutions to the global, local, syntenic, and spliced alignment problems, using the Cell Broadband Engine. Hirschberg [10]'s linear space method is used to compute the optimal score and alignment in parallel. With respect to the spliced alignment and using 16 SPEs (Synergistic Processing Elements), they achieve speedups up to 6, 7, or 10, depending on the sequential solution used for comparison. Nevertheless, in order to solve the spliced alignment problem, the authors model it as a special case of the syntenic alignment, deriving a different and simpler problem than that formulated by Gelfand. One main difference is that in [9], there is no set of candidate exons from the investigated sequence.

Liu and Schmidt [11] develop a framework to implement parallel solutions to Bioinformatics problems based on the dynamic programming technique. According to the authors, their parallel pattern-based system uses generic programming techniques that allow the representation of the algorithms based on the initial values of the dynamic programming matrices, the dependency relationship of the cells of the matrices, and an order to compute the matrices. Details about the spliced alignment solution adopted are not provided. Using a cluster with 32 processors, a speedup of 36 is achieved for the spliced alignment, when compared to the execution on only one processor. The super-linear speedup results are reported to be due to caching effects.

In [4] the authors present a survey of spliced alignment tools, which apply heuristic solutions based on several approaches. Some of these tools can be accelerated using a multi-core processor.

IV. Data Dependences and Parallelism

Despite its polynomial execution time, due to the exponential growth of genomic databases, the spliced alignment algorithm can be very computationally demanding, in both time and memory space. An idea to reduce the algorithm execution time is to calculate cells of the S structure in parallel. In order to investigate ways to exploit parallelism in the algorithm execution, we analyzed the data dependences for computing these cells.

Figures 3 and 4 show (with arrows) the data dependences for computing cell $S(i, j, k)$ in the spliced alignment algorithm. Figure 3 does not show the entire three-dimensional S structure, but only the matrix S_k corresponding to exon b_k. The shaded area represents the cells related to b_k that will be computed by the algorithm. Dependences in the figure are based on Equation 1, for computing $S(i, j, k)$ when i is not the first symbol of b_k. In this case, to calculate $S(i, j, k)$, cells $S(i-1, j-1, k)$, $S(i-1, j, k)$, and $S(i, j-1, k)$ are used.

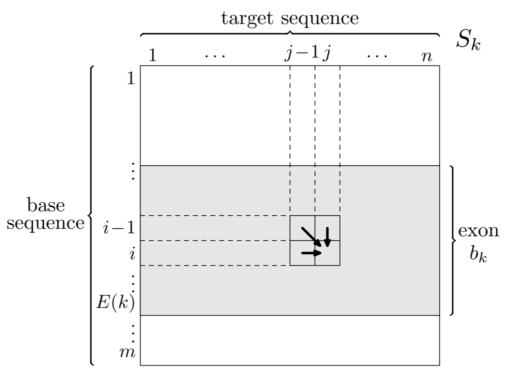

Fig. 3. Data dependences for computing $S(i, j, k)$, when i is not the first symbol of exon b_k

Figure 4 shows matrix S_k corresponding to exon b_k, and another matrix S_l corresponding to exon b_l, which in the figure represents all exons that precede b_k in base sequence s. The dependences in the figure are based on Equation 2, for computing $S(i, j, k)$ when i is the first symbol of b_k. In this case, to calculate $S(i, j, k)$ we use cell $S(i, j-1, k)$ from b_k and cells $S(E(l), j-1, l)$ and $S(E(l), j, l)$ from all exons b_l preceding b_k. The preceding exons are used because we want to find the subset of candidate exons with the best alignment to the target sequence.

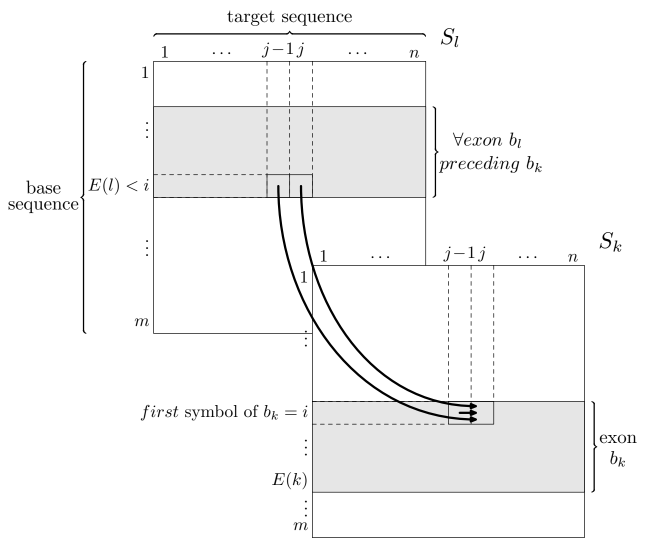

Fig. 4. Data dependences for computing $S(i, j, k)$, when i is the first symbol of exon b_k

A. Intra- and Inter-Exon Parallelism

Figure 5 shows the part of matrix S_k corresponding to exon b_k. The shaded area represents cells that have already been computed and the dashed line indicate the cells in a anti-diagonal. Given the data dependences for computing $S(i, j, k)$ when i is not the first symbol of b_k (Figure 3), we obtain the dependences to compute the cells in the anti-diagonal as indicated by the arrows in Figure 5.

Clearly, all cells in this anti-diagonal can be computed in parallel, since there are no dependencies between them. We denominate *intra-exon parallelism* the parallel computation of cells in an anti-diagonal of matrix S_k corresponding to an exon b_k. This form of parallelism is found in some other alignment problems in Bioinformatics. However, the difficulty is to exploit parallelism beyond the exon boundaries, that is, to overcome the data dependences shown in Figure 4.

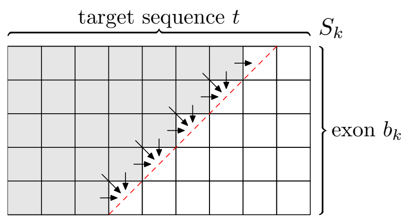

Fig. 5. Intra-exon parallelism: cells in an anti-diagonal of S_k can be computed in parallel

The data dependences in Figure 4, for computing $S(i, j, k)$ when i is the first symbol of b_k, prevent matrices S_l and S_k to be computed in parallel, for all exons b_l preceding exon b_k, that is, when b_l and b_k do not have any stretch of base sequence s in common. Nevertheless, if b_l and b_k do have some stretch of s in common, that is, $E(l) \geq$ first symbol of b_k, the dependences of Figure 4 do not apply.

Figure 6 shows part of matrices S_l and S_k corresponding to exons b_l and b_k, respectively. If b_l and b_k have some stretch of base sequence s in common, S_l and S_k can be computed in parallel. Then an anti-diagonal of S_l can be computed at the same time as an anti-diagonal of S_k, as illustrated by the dashed lines in the figure.

We denominate *inter-exon parallelism* the parallel computation of cells of matrices S_l and S_k from different exons b_l and b_k that have some stretch of s in common. Both forms of parallelism, intra- and inter-exon parallelism, can be combined and used in conjunction, as illustrated in Figure 6.

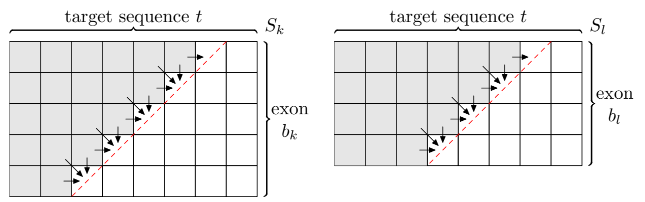

Fig. 6. Inter-exon parallelism: anti-diagonals of S_k and S_l can be computed in parallel

V. Potential Parallelism Analysis and Performance Estimation

Even though in theory intra- and inter-exon parallelism can be exploited in the spliced alignment algorithm, we need to verify if real biological data exhibit potential parallelism, i.e., if we can find opportunities to exploit these parallelism forms in real data.

A. Biological Data

Our biological data set consists of 240 base sequences, obtained from the ENCODE project [12]. Each sequence corresponds to a gene, along with its 1.000 preceding and succeeding bases, from *Homo sapiens*. For each base sequence, up to five target sequences are used. These sequences are cDNA sequences of genes homologous to the base sequence gene, obtained from the HomoloGene database, available at NCBI (National Center for Biotechnology Information) [13]. The GENSCAN [14] gene identification tool was used to construct the set of candidate exons of the base sequences, with true and false exons. More details about the data set construction can be found in [15].

Table I shows some information about the base sequences used and its exons, while Table II shows information about the target sequences. Each execution of the spliced alignment algorithm uses as input a base sequence, its set of candidate exons, and a target sequence. The biological data set created generates a total of 1010 different test cases and provides us solid ground for performing a comprehensive experimental evaluation.

TABLE I. BASE SEQUENCES AND EXONS

Information	Min	Avg	Max
Base sequence length	2,848	42,181.10	575,746
# of exons of a base sequence	2	55.08	622
Exon length	1	149.35	4,778
Σ exon length of a base sequence	442	7,711.73	71,873
# of base sequences	240		

TABLE II. TARGET SEQUENCES

Information	Min	Avg	Max
# of target sequences per base sequence	1	4.20	5
Target sequence length	189	1,460.48	8,652

B. Potential Parallelism Analysis

We want to measure the amount of parallelism that would in fact be available to be exploited by a parallel implementation, using our representative biological data set.

The amount of intra-exon parallelism available to be exploited in a test case execution depends on the size of the anti-diagonals (the number of cells that can be computed in parallel) of matrix S_k, for each exon b_k of the base sequence. The size of the anti-diagonals of S_k is defined by the smallest of its dimensions, which are the exon length and the target sequence length, as shown in Figure 5. We can assume, without loss of generality, that the anti-diagonal size is determined by the exon length. Therefore, the amount of intra-exon parallelism available to be exploited depends on the length of the exons of the base sequence.

For each test case of our data set, we computed the % of exons with a certain length with respect to the total number of exons, for each exon length. We summarized the results from all the test cases, computing an average and producing what we denominated *average test case* result. Figure 7 shows these intra-exon potential parallelism results for the average test case. This histogram represents the % of exons with respect to the total number of exons, for each exon length, in the average test case. The exon length corresponds to the number of cells that can be computed in parallel, represented in the graph in ranges of 32 cells. For instance, the first bar in the histogram indicates that, on the average test case, there are 16.55% of exons with only 32 cells to be computed in parallel. For clarity, bars with % of exons inferior to 0.1% (which correspond to exon lengths longer than 1024) are omitted and sum up 1.14% of the total number of exons.

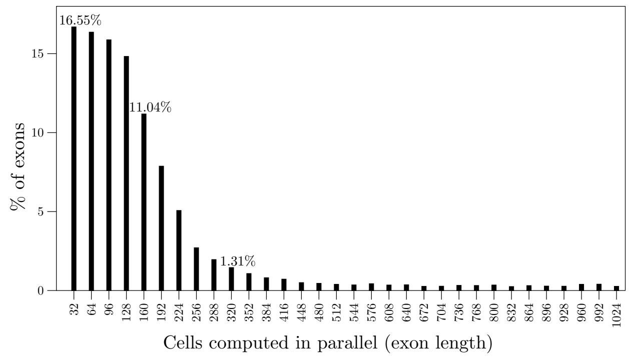

Fig. 7. Potential of intra-exon parallelism in average test case

The amount of inter-exon parallelism available to be exploited in a test case execution depends on the number of different exons b_k that have some stretch of s in common, because the matrices S_k corresponding to these exons can be computed in parallel. In a test case we can have several groups of exons whose matrices can be computed in parallel to each other, but sequentially with respect to the other groups.

For instance, Figure 8 shows a base sequence s with candidate exons $b_1, ..., b_5$. Exons b_2 and b_3 have symbol T in common, and b_4 and b_5 have symbols AT in common. Therefore, in order to compute the similarity score structure S respecting the data dependences, matrix S_1 corresponding to b_1 must be computed alone (a group of one exon), because all other matrices depend on it. Then S_2 and S_3 can be computed in parallel (a group of two exons), exploiting inter-exon parallelism, however both S_4 and S_5 depend on them and cannot be computed at the same time. Finally, S_4 and S_5 can also be computed in parallel to each other (another group of two exons).

$$s = \overbrace{\mathrm{ATGC}}^{b_1}\mathrm{AG}\underbrace{\mathrm{AG}\rlap{$\overbrace{\mathrm{TGT}}^{b_3}$}\mathrm{T}}_{b_2}\mathrm{GTGTAG}\underbrace{\mathrm{CC}\rlap{$\overbrace{\mathrm{ATGA}}^{b_5}$}\mathrm{AT}}_{b_4}\mathrm{GA}$$

Fig. 8. Base sequence s and candidate exons $b_1, ..., b_5$: matrices S_2 and S_3 can be computed in parallel, as well as S_4 and S_5

For each test case of our data set, we computed the % of groups with a certain number of exons with respect to the total number of groups executed, for each number of exons. Again we summarized the results from all test cases, computing an average and producing the average test case result. Figure 9 shows the inter-exon potential parallelism results for the average test case. This histogram represents the % of groups of exons with respect to the total number of groups executed, for each number of exons in a group, in the average test case. The number of exons in a group corresponds to the number of matrices that can be computed in parallel. For instance, the second bar in the histogram indicates that, on the average case, there are 21.35% of groups with two exons (with two matrices to be computed in parallel).

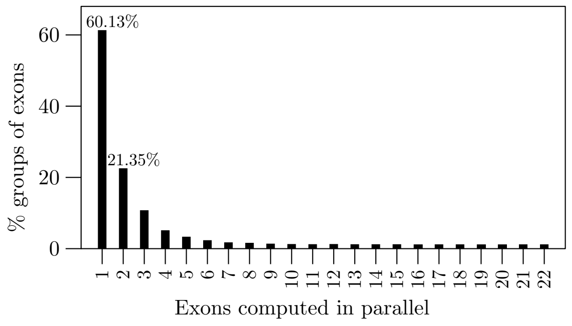

Fig. 9. Potential of inter-exon parallelism in average test case

Analyzing the average test case results, we see that in more than 50% of exons, at least 128 cells can be computed in parallel, what is a good amount of intra-exon parallelism to be exploited. Besides, almost 40% of groups of exons have at least two exons, that is, present inter-exon parallelism to be exploited. Therefore, combining both forms of parallelism can be a interesting approach to achieve a good performance in the spliced alignment problem.

C. Performance Estimation

Before developing and implementing parallel solutions for the spliced alignment algorithm, we want to estimate their execution time. The idea is to have a estimative of the performance gain that the parallel solutions will be able to achieve, with respect to a sequential solution, in order to decide if it is worth to implement them or not.

Using the time to compute a cell of the similarity score structure S as our unit of time, the estimated execution time of a solution is computed considering which cells can be computed in parallel and which are computed sequentially by the solution.

In our performance estimation model, given a base sequence $s = s_1...s_m$ with a set β of candidate exons, and a target sequence $t = t_1...t_n$, the estimated execution time of a sequential solution is simply the total number of cells of S that must be computed, as indicated in Equation 3, because the sequential solution will compute these cells one at a time.

$$\text{estimated time}_{\text{seq}} = n \times \sum_{k=1}^{|\beta|} |b_k| \tag{3}$$

In a parallel solution that exploits only intra-exon parallelism, the S_k matrices corresponding to the exons, are computed sequentially with respect to each other. However, each matrix S_k has the cells of a anti-diagonal computed in parallel, while the successive anti-diagonals are computed one after another. This way, the estimated time to compute S_k is the number of anti-diagonals on it, which is the sum of its dimensions (exon length and the target sequence length), minus one, that is:

$$|b_k| + n - 1$$

Then, the estimated execution time of a parallel solution exploiting intra-exon parallelism is obtained as indicated in Equation 4.

$$\text{estimated time}_{\text{intra-exon par}} = \sum_{k=1}^{|\beta|} (|b_k| + n - 1) \tag{4}$$

In a parallel solution that exploits both intra- and inter-exon parallelism, we have groups of exons whose matrices are computed in parallel to each other, but sequentially with respect to the other groups.

Given a group $\mathcal{G}$ of exons b_k, corresponding to matrices S_k that can be computed in parallel, the anti-diagonals of these different matrices can be computed in parallel, and the estimated time to compute the entire group $\mathcal{G}$ is defined by the matrix which has more anti-diagonals to compute. Therefore, the estimated time to compute the matrices of all exons in $\mathcal{G}$ is:

$$\max_{\forall b_k \in \mathcal{G}} |b_k| + n - 1$$

Finally, the estimated execution time of a parallel solution exploiting intra- and inter-exon parallelism is obtained as indicated in Equation 5.

$$\text{estimated time}^{\text{intra-exon}}_{\text{inter-exon}}\text{par} = \sum_{\forall \mathcal{G}} \max_{\forall b_k \in \mathcal{G}} |b_k| + n - 1 \tag{5}$$

We measured, using Equations 3, 4, and 5, the estimated execution times, for the sequential, intra-exon only, and intra- and inter-exon parallel solutions, for all test cases of our biological data set. Figure 10 compares the results, where the test cases are ordered by the estimated execution time of the sequential solution and all estimated times are shown using logarithmic scale.

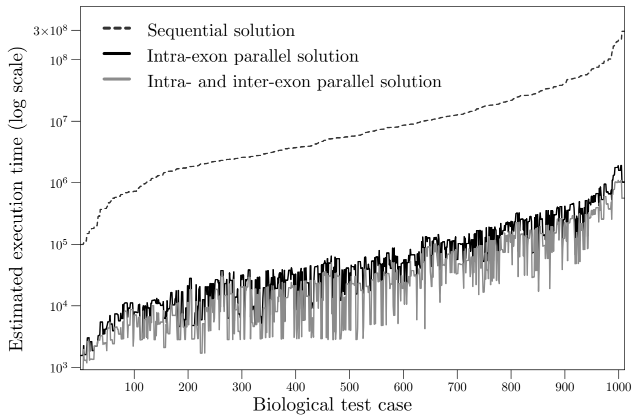

Fig. 10. Estimated execution times: sequential and parallel solutions

Estimated times for the solution exploiting only intra-exon parallelism show approximately a 100-fold improvement with respect to the estimated times for the sequential solution. The estimated times for the solution that exploits both forms of parallelism are always lower than or equal to those obtained for the solution that exploits only intra-exon parallelism, with approximately a 10-fold improvement in many test cases. Therefore, we conclude that it is worth to develop parallel solutions to the spliced alignment algorithm exploiting intra- and inter-exon parallelism. However, when implementing our parallel solutions we cannot expect to achieve performance gains in the same proportion as estimated in this analysis, because our targeted platform may not be able to exploit all parallelism available in the biological data and our performance estimation model does not consider any control, communication or synchronization overhead.

VI. GPU Solutions to Spliced Alignment

In order to evaluate separately the impact on performance of the intra- and inter-exon parallelism, we developed two GPU accelerators to the spliced alignment algorithm. The first one exploits only intra-exon parallelism, while the second accelerator exploits both forms of parallelism. Our accelerators receive as input a test case, that is, a base sequence s with a set β of candidate exons, and a target sequence t, and computes the similarity score of the optimal spliced alignment. The accelerator programs were developed using the CUDA programming model [16].

A. First GPU Accelerator

Since the first GPU accelerator exploits only intra-exon parallelism, the S_k matrices corresponding to the exons are computed sequentially with respect to each other. Each matrix S_k has the cells of a anti-diagonal computed in parallel, while the successive anti-diagonals are computed sequentially. This is implemented through a CUDA kernel for computing matrix S_k corresponding to exon b_k. For each exon, a new kernel invocation is made, only after the previous invocation has finished its execution. Each invocation creates a grid with only one block, associated to exon b_k, and this block can have up to 1024 threads, depending on the exon length.

The kernel execution has three phases. In the first phase, we obtain the scores from exons b_l preceding b_k needed to compute S_k, as indicated in the first two lines of Equation 2. Each thread is associated to a symbol of target sequence t. There is a sequential loop for scanning all exons b_l preceding b_k. At each iteration of the loop, one matrix S_l (corresponding to exon b_l) is accessed from GPU global memory, and row $E(l)$ of S_l (last row of b_l) is read. Each thread reads a different element of the row, in a way that all threads access the entire row in parallel, achieving memory coalescency. Each thread compares the value it read with the maximum value accumulated at the previous loop iterations, so that at the end of the loop execution, it obtains the maximum of the values read. These maximum values computed by the threads are stored in a structure allocated in GPU shared memory. Computing the maximum values from exons b_l preceding b_k and storing them in shared memory prevent us from performing the same computation repeatedly, since different cells of S_k have dependences on the same cells of matrices S_l.

In the second phase, we compute matrix S_k. Each thread is associated to a symbol of exon b_k. There is a sequential loop for scanning all anti-diagonals of S_k. At each iteration of the loop, one anti-diagonal is computed, with each thread computing a different cell of the anti-diagonal, in a way that all threads compute the entire anti-diagonal in parallel. In this phase, all scores are accessed in the structure allocated in shared memory and no access to scores in global memory is done.

Finally, in the last phase of the kernel execution, we copy row $E(k)$ of S_k (last row of exon b_k) from the structure in shared memory back to global memory. Again, each thread is associated to a symbol of target sequence t. Each thread writes a different element of the row, in a way that all threads access the entire row in parallel achieving memory coalescency.

Some thread synchronizations are needed at certain points in the kernel. After some initializations and before the first phase, a synchronization is necessary. Between the first and second phases, too. Finally, inside the second phase, at the end of each loop iteration, we synchronize the threads again.

Our GPU accelerator is able to handle input test cases with exon lengths and/or target sequence length longer than the maximum number of threads of the GPU used (1024). In order to do that, we implemented all three phases of the kernel using the tiling technique [17], with each thread being associated to several cells, instead of only one.

B. Second GPU Accelerator

The first accelerator does not use all GPU resources and does not exploits all parallelism available in the application. Therefore a second and improved solution is proposed. The second GPU accelerator exploits both intra- and inter-exon parallelism. Each matrix S_k has the cells of a anti-diagonal computed in parallel. Besides, for the groups of exons whose matrices can be computed in parallel to each other, but sequentially with respect to other groups, we compute the anti-diagonals of these different matrices in parallel.

Again, this is implemented through a CUDA kernel for computing matrix S_k corresponding to exon b_k. For each group $\mathcal{G}$ of exons b_k, corresponding to matrices S_k that can be computed in parallel, a new kernel invocation is made, only after the previous invocation has finished its execution. Each invocation creates a grid with $|\mathcal{G}|$ blocks, each block associated to an exon $b_k \in \mathcal{G}$, and this block can have up to 1024 threads, depending on the exon length. Therefore, this second accelerator uses several SMs (Streaming Multiprocessors) of the GPU and makes a better use of the GPU resources than the first accelerator.

The kernel is similar to the one of the first accelerator, with the same three phases, since its goal is to compute matrix S_k. The main difference is the execution space created on the kernel invocation, that for the second accelerator can have several blocks of threads, instead of only one.

Another important difference between the accelerators is that, in the second one, it is necessary to identify the groups of exons whose matrices can be computed in parallel. This is performed by the host processor in parallel to the GPU. The host makes a non-blocking kernel invocation for a group $\mathcal{G}$ of exons. While the GPU computes the matrices of $\mathcal{G}$, the host identifies the next group $\mathcal{G}'$. Then, when the GPU finishes executing that first kernel invocation, the host is ready to issue the kernel invocation for $\mathcal{G}'$.

The second GPU accelerator also applies the tiling technique to handle input test cases with exon lengths and/or target sequence length longer than the maximum number of threads of the GPU used.

C. Data Structure Organization

Equations 1 and 2 compute a total of $n \times \sum_{k=1}^{|\beta|} |b_k|$ cells of the S similarity score three-dimensional structure. Storing this structure completely in the global memory of the GPU would limit the size of the input test cases that the accelerator would be able to handle. A biological test case with a long target sequence, and/or a base sequence with many exons and/or long exons could produce a S structure exceeding the capacity of the GPU global memory.

In order to overcome this limitation, only the last row (row $E(l)$) of each matrix S_l, corresponding to exon b_l, is stored in GPU global memory. As Figure 4 shows, this is the unique row of S_l that can be needed when computing matrix S_k, of a exon b_k succeeding b_l.

As described previously, a structure is allocated in GPU shared memory for storing matrix S_k, while it is computed in the second phase of the kernel, where the cells in a anti-diagonal are computed in parallel. Again, storing this matrix completely in the shared memory could limit the size of the input test cases of the accelerator. Observing Figure 5 we conclude that, for computing the current anti-diagonal, only two immediately preceding anti-diagonals are used. This way, we allocate space in shared memory for only three anti-diagonals of matrix S_k, and we reuse them for computing subsequent anti-diagonals of the matrix.

Figure 11(a) shows matrix S_k corresponding to exon b_k organized in its conventional form. Cells with the same color are in the same anti-diagonal and are computed in parallel by different threads. Using this conventional organization, the threads perform relatively complex index calculations in order to access the cells of an anti-diagonal. Besides, the approach of allocating only three anti-diagonals and reusing them to compute all anti-diagonals saves memory space, but generates divergence among the threads. In a GPU execution, divergence happens when threads in the same warp take different execution paths, and these paths are executed sequentially until the threads join the same execution path again. Divergences can reduce execution parallelism on GPU and, consequently, degrade performance [17].

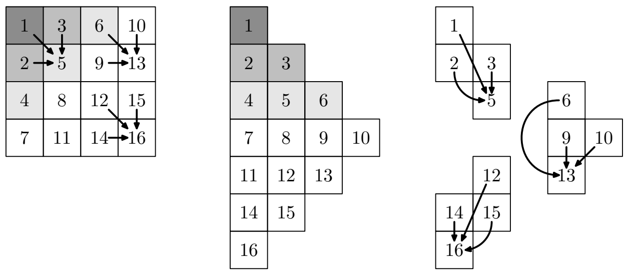

Fig. 11. (a) Conventional organization of S_k and data dependences with uniform access pattern; (b) Skewed organization of S_k; (c) Data dependences with non-uniform access pattern, from skewed organization of S_k

The solution is to organize the S_k matrix cells in a skewed form, as shown in Figure 11(b). This way, an anti-diagonal can be treated as a row, and the cells in the same anti-diagonal can be handled as occupying contiguous positions in the memory. For ease of explanation, matrix cells are numbered sequentially based on their order in the skewed organization of S_k.

Figure 11(a) also shows, with arrows, the data dependences for computing some cells of S_k, based on Equation 1. For instance, for computing cell 5 we use cells 1, 2, and 3. Figure 11(c) highlights how these data dependences appear in the skewed organization of S_k. We observe that, in the skewed organization, the data dependences no longer have an uniform access pattern to the matrix cells. When the threads access the matrix cells in parallel, this non-uniform access pattern produces complex index computations and divergences among threads, which may have a negative impact on performance.

Once more, the solution is to reorganize S_k matrix cells, this time, modifying the skewed organization of S_k by aligning all cells to the right, as shown in Figure 12(a). Figure 12(b) highlights how the data dependences appear in the skewed and right-aligned organization of S_k. An uniform access pattern to the cells is achieved again, eliminating the complex index computations and the thread divergences.

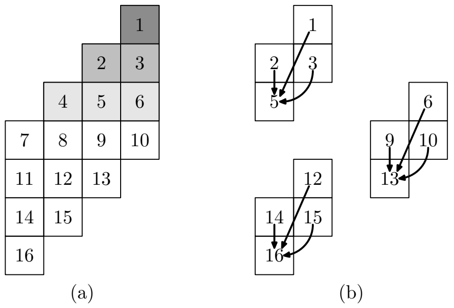

Fig. 12. (a) Skewed and right-aligned organization of S_k; (b) Data dependences with uniform access pattern, from skewed and right-aligned organization of S_k

In the skewed and right-aligned organization of S_k, the threads compute in parallel all cells in a row. Each thread is assigned to a different column of the structure. Only three rows are allocated in GPU shared memory for this structure and used to store the current row and the two previous ones, and them reused for computing all subsequent rows.

We also used pinned memory for optimizing the transfers of input data structures from host memory to GPU global memory over the PCI-Express bus [18].

VII. Results and Discussions

The execution platform used in this work consists of a GPU NVIDIA GeForce GTX 460 with 1GB RAM connected to a host computer (through a PCI-Express interface) with an Intel Core 2 Quad processor and 4GB RAM. The host computer is also used for executing a sequential implementation of the Gelfand spliced alignment algorithm.

A. Accelerators Performance

We evaluated the GPU accelerators developed using the same biological data set described in Section V and used for the potential parallelism analysis and performance estimation. Now, the goal is to measure the actual performance gain achieved with the accelerators.

Figure 13 compares the actual execution times of the sequential implementation of the Gelfand spliced alignment algorithm and the first and second GPU accelerators, for all 1010 test cases of our data set. The test cases are ordered by the execution time of the sequential implementation and execution times are shown using logarithmic scale.

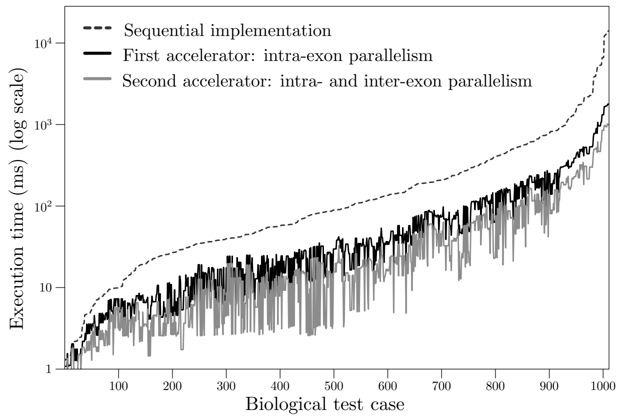

Fig. 13. Actual execution times: sequential implementation, first and second GPU accelerators

The execution time of the sequential implementation depends mainly on the total number of cells to be computed. The first and second GPU accelerators exploit only intra- and both intra- and inter-exon parallelism, respectively. Therefore, their execution times depends on other factors besides the number of cells to be computed, which includes the length of the exons (for both accelerators) and the number and cardinality of groups of matrices S_k that can be computed in parallel (only for the second accelerator). This explains why the execution time curves of the accelerators are ragged, specially the second one, with ups and downs.

The execution times produced by the first accelerator are almost always much lower than the execution times of the sequential implementation, while the second accelerator achieved execution times almost always much lower than the first one.

The first accelerator produced an execution time longer than the sequential solution in only 0.4% of the test cases. Analyzing these test cases, we found out they have minimum average exon length among all test cases and, consequently, little intra-exon parallelism available. The second accelerator was faster than the sequential solution for all test cases. Finally, in only 0.4% of test cases the second accelerator produced an execution time longer than the first one. Analyzing these test cases, we verified they have no inter-exon parallelism available. Therefore, for these test cases both accelerators exploited only intra-exon parallelism, and the second accelerator suffered from a small overhead from looking for inter-exon parallelism opportunities. Nevertheless, in these cases the difference in the execution times of the accelerators is insignificant and on average 0.002ms.

Figure 14 shows the speedup achieved with the first and second GPU accelerators with respect to the sequential implementation of the Gelfand spliced alignment algorithm. The test cases ordering is the same of Figure 13. The first accelerator produced a maximum speedup of 7.89, and 3.57 on average, while the second accelerator achieved a maximum speedup of 31.51, and 7.28 on average.

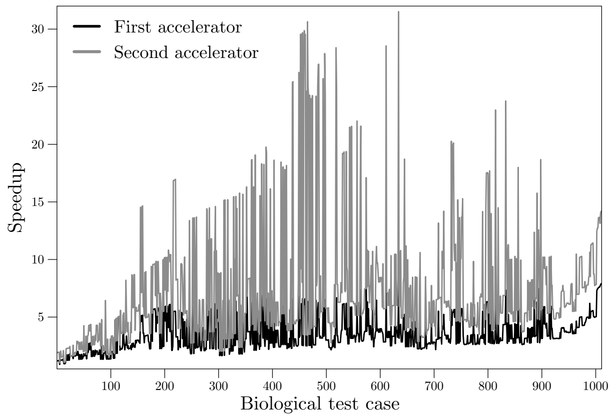

Fig. 14. Speedup of first and second GPU accelerators wrt. sequential solution

The performance gain produced by the first accelerator is limited by the length of the exons, since it exploits only the intra-exon parallelism. The second accelerator combines this form of parallelism with the inter-exon parallelism, exploiting better the available parallelism in the test case. Therefore, the speedup of the second accelerator is always greater than or equal the speedup of the first accelerator, except for test cases with no inter-exon parallelism available.

We also evaluated the throughput measure CUPS (Cell Updates per Second), which indicates how many cells of the dynamic programming matrices are computed in one second. Table III shows the average and maximum MCUPS (10^6 CUPS) achieved by the sequential implementation, first and second accelerators, among all test cases.

TABLE III. AVERAGE AND MAXIMUM MCUPS FOR SEQUENTIAL IMPLEMENTATION, FIRST AND SECOND ACCELERATORS

Solution	Avg MCUPS	Max MCUPS
Sequential implementation	62.01	83.90
First accelerator	219.83	506.71
Second accelerator	452.73	2083.81

B. Accelerators Scalability and Comparison with Related Work

We want to evaluate how well the accelerators scale for larger input data sizes and to compare their performance with other solution in related work. Figure 15 shows the second GPU accelerator execution times, for all 1010 test cases of our data set. However, here the test cases are ordered by the product $m \times n$ of base and target sequences length. Both execution times and products $m \times n$ are shown using logarithmic scale. We observe that the accelerator performance scales linearly with the product of the input sequences length, as we expected.

In Figure 16 we reproduce the results from Sarje and Aluru [9] (described in Section III) for the spliced alignment running in the Cell Broadband Engine with 16 SPEs using a synthetic data set. The figure shows the execution times with respect to the product $m \times n$ of the input sequences

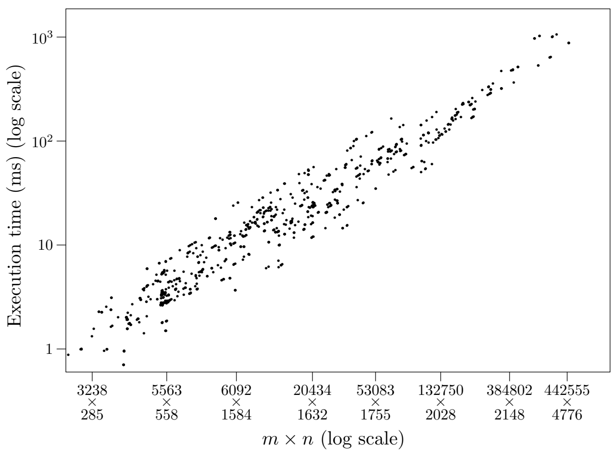

Fig. 15. Second GPU accelerator execution times with biological test cases ordered by $m \times n$: m and n are base and target sequences length, respectively

length. Even though the comparison of these results with those in Figure 15 is not precise given the differences in the experiments, we can see that the second GPU accelerator outperforms the results in Figure 16. For instance, the largest input data size in Figure 16 is 2816×2816, which takes approximately 140ms. For test cases with similar sizes, as 6092×1585, the second accelerator takes much shorter execution times, approximately 10ms. Besides, the second accelerator is able to handle much larger input data sizes. Finally, the authors report a maximum of 755 MCUPS, while our second accelerator achieved a maximum of 2083.81 MCUPS.

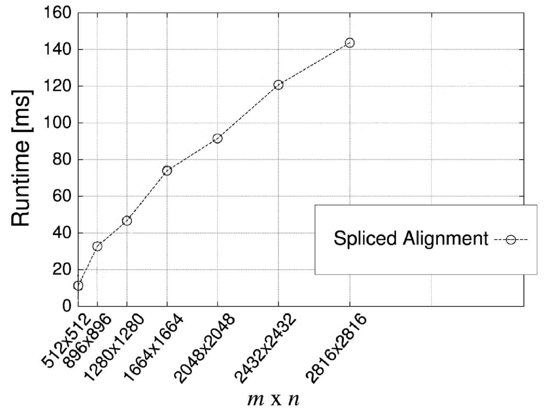

Fig. 16. Results from Sarje and Aluru [9]: execution times in the Cell Broadband Engine with 16 SPEs (m and n are input sequences length)

We also obtained performance results for aligning a sequence containing the IL1RAPL1 gene, along with its 400,000 preceding and succeeding bases, from *Homo sapiens* chromosome X, to the cDNA sequence of a homologous gene from *Canis lupus familiaris*. The base and target sequences have length 2,168,787 and 2,355, respectively. Using the GENSCAN tool, a set of 1,878 candidate exons was generated. These input data constitute a test case much larger than those in our biological data set. The execution times of the sequential implementation, the first and second accelerators were 357,506.67ms, 6,794.17ms and 3,934.57ms, respectively. The first and second accelerators achieved speedups of 52.62 and 90.86, respectively, with respect to the sequential implementation. Both accelerators were able to handle this larger input data and reached much higher speedups than those obtained with the initial data set.

We also tried to run the IL1RAPL1 gene input data using the spaln2 spliced alignment heuristic tool [4], using a multi-core processor, in order to compare the performance obtained to our exact GPU accelerators. However, this tool is not able to handle base sequences with length longer than 100,000.

C. Accuracy of Performance Estimation

In Section V we performed a performance estimation, before implementing the accelerators, with the goal of estimating their execution times. We concluded it was worth to implement them, because the results from the analysis indicated performance gains with solutions exploiting intra- and inter-exon parallelism. Now that we have the actual performance results, we can compare the estimative and actual execution times and evaluate how precise our performance estimation model is.

In Figure 17 we overlap the estimated execution times for the intra- and inter-exon parallel solution, taken from Figure 10, with the actual execution times of the second GPU accelerator, taken from Figure 13, for all test cases. In both curves, the test cases ordering is the same of Figure 10. Both curves are in logarithmic scale, but use different units of measurement for time. In our performance estimation model we use the time to compute a cell of the similarity score structure S as our unit of time, while actual execution time is given in milliseconds. This explains why the curves have different ranges in the y-axis. We overlapped the curves to stress how precise the performance estimation model was. The ragged shape of both curves match, point by point, with ups and downs. We conclude that the performance estimation model predicted accurately the performance of the second accelerator.

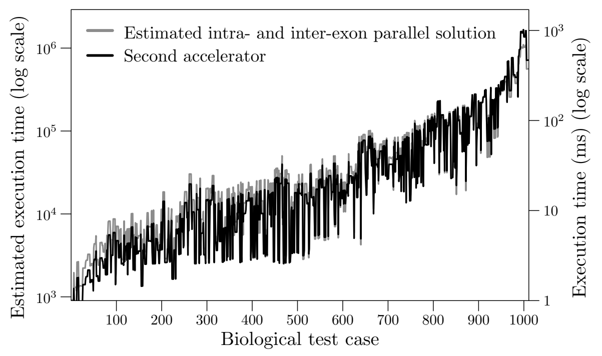

Fig. 17. Estimated and actual execution times overlapped, for the second accelerator

Figures 18 and 19 also compare estimated and actual execution times for the first accelerator and for the sequential implementation of the spliced alignment algorithm, respectively. Similar conclusions about the performance estimation accuracy can be drawn.

VIII. Conclusion

This paper presented two GPU accelerators for the spliced alignment problem applied to gene prediction. To the

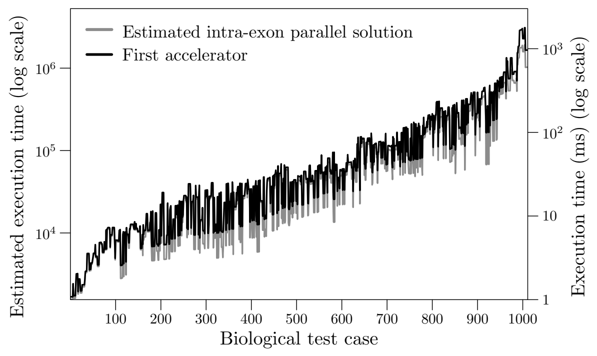

Fig. 18. Estimated and actual execution times overlapped, for the first accelerator

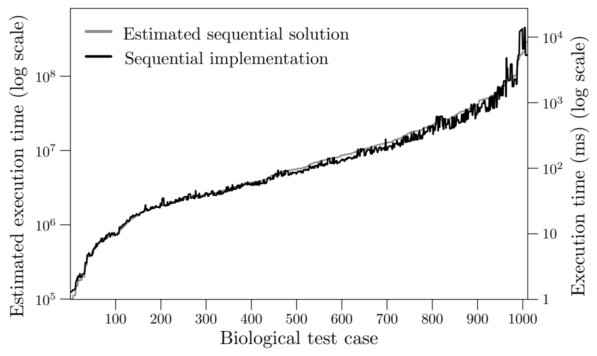

Fig. 19. Estimated and actual execution times overlapped, for the sequential implementation

authors' knowledge, these are the first GPU-based solutions proposed for the spliced alignment problem.

We identified two forms of exploiting parallelism in the spliced alignment algorithm, intra- and inter-exon parallelism. Our first GPU accelerator exploits only the first form, while both forms are exploited by the second one. The results show that we should exploit both forms of parallelism in order to have more significant performance gains.

We performed a comprehensive performance evaluation using a modest GPU and a large and representative biological data set. The accelerators produced excellent performance results. The first and second accelerators achieved maximum speedups of 52.62 and 90.86, respectively, when compared to a sequential implementation, and produced the maximum of 506.71 and 2083.81 MCUPS, respectively. We concluded that the second accelerator performance scaled linearly with the product of the input sequences length, outperforming the results from a related work. Using a more powerful GPU, our accelerators would achieve an even better performance, since less tiling would be necessary.

Since the main GPU optimizations involve the efficient use of the memory hierarchy, we proposed an organization for the data structures of the accelerators in order to maximize coalescency in global memory accesses, maximize the use of shared memory, minimize thread divergences, and simplify index computations. Besides, our accelerators are able to handle input data sets with long sequences and exons, applying the tiling technique.

Before implementing the accelerators, we performed a potential parallelism analysis of our biological data set with the goal of measuring the amount of parallelism that would in fact be available to be exploited by a parallel implementation. We also proposed a performance estimation model that enabled estimating the accelerators performance and evaluating if there would be performance gains, compared to a sequential solution. The experiments showed that this model predicted very accurately the performance of the accelerators.

Despite the excellent results achieved with intra- and inter-exon parallelism, an interesting research subject is to investigate other forms of exploiting parallelism in the spliced alignment algorithm, and how they can be mapped on a GPU or other platform.

References

[1] N. Jones and P. Pevzner, *An Introduction to Bioinformatics Algorithms*. The MIT Press, 2004.

[2] M. Gelfand, A. Mironov, and P. Pevzner, "Spliced Alignment: A New Approach to Gene Recognition," in *Proceedings of the 7th Annual Symposium on Combinatorial Pattern Matching*, 1996, pp. 141–158.

[3] A. Baxevanis and B. Ouellette, *Bioinformatics – A Practical Guide to the Analysis of Genes and Proteins*, 3rd ed. Wiley-Interscience, 2004.

[4] H. Iwata and O. Gotoh, "Benchmarking spliced alignment programs including Spaln2, an extended version of Spaln that incorporates additional species-specific features," *Nucleic Acids Research*, vol. 40, no. 20, pp. 1–9, 2012.

[5] J. Puthukattukaran, S. Chalasani, and P. Senapathy, "Design and Implementation of Parallel Algorithms for Gene-Finding," in *Proceedings of the 3rd IEEE International Symposium on High Performance Distributed Computing*, 1994, pp. 186–193.

[6] J. Chang *et al.*, "Space-Gene: Microbial Gene Prediction System Based on Linux Clustering," *Genome Informatics Series*, vol. 14, pp. 571–572, 2003.

[7] G. Chrysos, E. Sotiriades, I. Papaefstathiou, and A. Dollas, "A FPGA Based Coprocessor for Gene Finding Using Interpolated Markov Model (IMM)," in *Proceedings of the International Conference on Field Programmable Logic and Applications*, 2009, pp. 683–686.

[8] N. Chrysanthou, G. Chrysos, E. Sotiriades, and I. Papaefstathiou, "Parallel Accelerators for GlimmerHMM Bioinformatics Algorithm," in *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition*, 2011, pp. 94–99.

[9] A. Sarje and S. Aluru, "Parallel Genomic Alignments on the Cell Broadband Engine," *IEEE Transactions on Parallel and Distributed Systems*, vol. 20, no. 11, pp. 1600–1610, 2009.

[10] D. Hirschberg, "A Linear Space Algorithm for Computing Maximal Common Subsequences," *Communications of the ACM*, vol. 18, no. 6, pp. 341–343, 1975.

[11] W. Liu and B. Schmidt, "A Generic Parallel Pattern-Based System for Bioinformatics," in *Proceedings of the European Conference on Parallel Processing*, 2004, pp. 989–996.

[12] T. E. P. Consortium, "An integrated encyclopedia of DNA elements in the human genome," *Nature*, no. 489, pp. 57–74, 2012.

[13] NCBI, "National Center for Biotechnology Information," http://www.ncbi.nlm.nih.gov/, accessed June, 2014.

[14] GENSCAN, "The GENSCAN Web Server at MIT," http://genes.mit.edu/GENSCAN.html, accessed June, 2014.

[15] R. Kishi, "Identificação de Genes e o Problema do Alinhamento Spliced Múltiplo," Master's thesis, Faculdade de Computação da Universidade Federal de Mato Grosso do Sul, 2010.

[16] NVIDIA Corporation, "CUDA C Programming Guide," http://docs.nvidia.com/cuda/cuda-c-programming-guide/, accessed June, 2014.

[17] D. Kirk and W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann, 2010.

[18] S. Cook, *CUDA Programming – A Developer's Guide to Parallel Computing with GPUs*. Morgan Kaufmann, 2013.