Pairwise Sequence Alignment with Gaps with GPU

Thomas C. Carroll
Department of Computer Science
University of Liverpool, UK
Email: thomas.carroll@liverpool.ac.uk

Jude-Thaddeus Ojiaku
Department of Computer Science
University of Liverpool, UK
Email: J.Ojiaku@liverpool.ac.uk
ASML Netherlands B.V.
Veldhoven, The Netherlands

Prudence W.H. Wong
Department of Computer Science
University of Liverpool, UK
Email: pwong@liverpool.ac.uk

***Abstract*—In this paper we consider the pair-wise sequence alignment problem with gaps, which is motivated by the *re-sequencing* problem that requires to assemble short reads sequences into a genome sequence by referring to a reference sequence. The problem has been studied before for single gap and bounded number of gaps. For single gap, there was a GPU-based algorithm proposed. In our work we propose a GPU-based algorithm for the bounded number of gaps case. We implemented the algorithm and compare the performance with the CPU-based algorithm in a multithreadded environment; the results are promising with the GPU version achieving a speedup of 30 times.**

I. Introduction

A. Alignment problem

The problem of finding alignment between two biological sequences has been extensively studied. An alignment allows highlight of common areas between sequences, on the premise that homology between two sequences can show some sort of connection, or in the case of an unknown gene sequence, can indicate what gene the sequence is most related to. Roughly speaking, aligning a short pattern sequence to a longer text sequence is to determine whether the pattern exists in the text and if so the positions where it occurs.

With the advances in sequencing technologies, the amount of data that requires alignment has increased drastically. For example, the Illumina HiSeqX Ten sequencer[1] can produce three billion reads (sequences) of length 250 bp (base pairs) in less than three days. The *re-sequencing* problem is to assemble short reads produced by the sequencer (an equipment that takes a physical biological sample and outputs the sequence of nucleobases as a character string) into a genome sequence by referring to a reference genome, requiring "mapping" or "aligning" short reads back to reference sequences. The task is challenging due to the vast amount of data and the large genome sizes.

There is a wide range of short-read alignment tools available, e.g., Bowtie [2], BWA [3], GenomeMapper [4], MAQ [5], SOAP2 [6], SHRiMP [7], Stampy [8], REAL [9], addressing different aspects of the problem. Due to the data size, faster tools are needed. This asserts not just speed requirement on the processors but also leads to high power/energy requirements; furthermore, this potentially causes too high temperature that may damage the processors. To solve this problem, it is nowadays common to exploit multi-processors in particular using graphic processing units (GPU) to achieve drastic increase in speed. SOAP3 [10] is developed using this idea and is currently among the best short-read alignment tools available.

Because of mutations and other biological mechanisms, it is common that sequences in comparison may not be exact match but may have some mismatches. It is important to take into account mismatches otherwise some vital information may be missing. However, allowing mismatches greatly increases the complexity of the problem and algorithms detecting mismatches are significantly slower than their counterparts that detect exact matches. Existing short-read alignment tools including those mentioned above usually only allow a small number of mismatches or do not allow any mismatches because of this.

Differences may appear in the form of a *gap*, which is a consecutive region that appears in the text but not in the pattern or vice versa (i.e., a consecutive sequence of insertions or deletions of letters in the text or the pattern). It has been claimed that it can be desirable to penalize the occurrence of gap as a whole instead of individual alternations [11]. Gaps may occur because of mutation event that a segment of DNA sequence is copied or inserted, replication process that a segment is missing, or genetic transposition that a segment changes position on chromosomes.

For example, suppose we have two sequences `TCGTTA` and `TCTA`. If we do not allow gap, we can align `TCGT` with `TCTA` with two matches. If we allow a gap of any length, we can align `TCGTTA` with `TC**TA` with four matches, where $\star$ represents a gap character. If we allow two gaps, we can align `TCGTTA` with `TC*T*A`, also with four matches.

B. Related work

Because of the importance of gaps, the alignment problem has been considered in the presence of gaps [11], [12], [13]. In addition to allowing mismatches in the form of edit distance or score, the problem also allows for a bounded number of gaps (of any length). In [11], [12], a single gap is allowed and the algorithm GapMis is proposed; while in [13], multiple gaps are allowed and the algorithm GapsMis is proposed. Usually the number of gaps allowed is a small constant independent of the length of the text or pattern. Dynamic programming algorithms have been proposed to find the alignment with the best alignment "score" with a bounded number of gaps.

The algorithms GapMis and GapsMis have been implemented and are shown to perform well against other approaches like EMBOSS water and EMBOSS needle [14]. With single gap, a tool called libgapmis using GPU has been developed in [12] for which a $11\times$ speedup has been reported. With multiple gaps, there is a lack of GPU-based algorithms.

C. *Our contribution*

Our main contribution is to propose a GPU-based algorithm for the pair-wise sequence alignment problem with multiple gaps, which is independent of whether we are using CUDA or OpenCL. The algorithm, which we call GPUGapsMis, is based on the GapsMis algorithm in [13]. To achieve a better result, we try to maximize the amount of parallelism by using appropriate data structures to store the data and hence decrease the I/O to shared and global memory, which could be a bottleneck in speeding up. We also extend the algorithm to allow the use of scoring matrix in addition to the Hamming distance that is considered in GapsMis. We have implemented our algorithm and also a modified version of the sequential algorithm GapsMis with the scoring matrix; we call the extended algorithm CPUGapsMis. We compare the performance of GPUGapsMis and CPUGapsMis and the speedup is about 30 times in computing the alignment score matrix. We also give theoretical analysis of the algorithm GPUGapsMis based on the AGPU model in Section I-D.

As mentioned in related work (Section I-B), the performance of GapMis and GapsMis is good relative to other similar alignment tools in terms of accuracy. Since our algorithm is based on the CPU algorithm GapsMis, the alignment results computed will be the same and so in terms of accuracy it will be good relative to other alignment tools. Our major contribution is to speed up the computation of the solution.

D. *AGPU model*

Most of the existing work on using GPU evaluates these algorithms empirically. Recently, Koike and Sadakane [15] has proposed a new theoretical model for GPUs called the Abstract GPU (AGPU) model. Since known parallel computational models such as the Parallel Random-Access Machine (PRAM) models [16], [17], [18] are not appropriate for evaluating GPU-based algorithms, it is necessary to have new theoretical model to capture the essence of GPU architecture. (Another algorithmic model for GPUs has been developed by Sitchinava and Weichert [19].) PRAM models consist of multiple cores and a single shared memory unit and are standard computational models for parallel algorithms. However, as claimed in [15], algorithms developed on the PRAM models do not always show good performance on GPUs due to the substantial differences from the actual GPU architectures. The AGPU model is designed to analyze asymptotic computational complexities of GPU-based algorithms and to pinpoint the bottleneck for the performance. The AGPU model captures factors affecting the performance such as coalescing, bank conflict and multithreading. Similar to usual asymptotic analysis of algorithms, the AGPU model provides a mechanism to analyze the performance of GPU algorithms before implementation and hence save effort from implementing algorithms that are analyzed to run badly asymptotically.

In the AGPU model, GPU algorithms are measured by time complexity, I/O complexity, the amount of global memory used, and the amount of shared memory used.

The *time complexity* measures the number of instructions each multiprocessor executes. Should there be thread divergence within a multiprocessor, all paths are counted for the time complexity. Where the time complexity of multiple multiprocessors vary, the largest complexity is used.

The *I/O complexity* measures the total number of global memory access operations performed by all multiprocessors. Because the amount of parallelism for memory requests to be fulfilled is dependent on the bandwidth of the architecture, the I/O Complexity is defined as the summation of all global memory requests from all multiprocessors.

The *amount of global and shared memory used* measures the memory usage of the algorithm. If the amount of shared memory used varies amongst the multiprocessors, the largest value is taken.

We analyze the performance of GPUGapsMis based on the AGPU model and present it in Theorem 1.

E. *Other biological problems with GPU*

GPU is made up of many small processing elements and is able to exhibit a massive scale of parallelism in its execution. GPUs were initially developed for graphical applications and have since evolved into a general purpose computing device. As the financial hardware cost continues to fall, they are becoming a more feasible solution to implementing large scale parallelism in programs. Thanks to frameworks such as OpenCL and CUDA, the capabilities of the hardware are becoming more and more accessible to developers, yet the skill set required for programming on the GPU is still requiring a large learning curve, and the time and labor costs associated with developing algorithms for the GPU is still comparatively high, when compared to a standard sequential algorithm. Despite these issues, various bioinformatics problems have been tackled using GPU, including Smith-Waterman global alignment algorithm [20], [21], [22], [23], BLAST (Basic Local Alignment Search Tool) [24], [25], and others [26], [27], [28].

II. Preliminaries

A. *Problem Definition*

We first introduce some notations required for the definition of the problem. Consider an alphabet Σ. A string a is a *substring* of string b if there exist two (possibly empty) strings s_1 and s_2 such that $s_1 a s_2 = b$. Furthermore, a is a *prefix* (*suffix* resp.) of b if s_1 (s_2 resp.) is an empty string.

Let $*$ be the gap character and $\Sigma' = \Sigma \cup \{*\}$. *An aligned pair* is a pair of letter (x, y) such that $(x, y) \in \Sigma' \times \Sigma' \setminus \{*, *\}$, i.e., an aligned pair may involve at most one gap character. An alignment of two strings X and Y is a string of aligned pairs (x_1, y_1), (x_2, y_2), $\cdots$, (x_ℓ, y_ℓ) such that removing all the gap characters $*$ from $X' = x_1 x_2 \cdots x_\ell$ gives X (similarly for Y). Note that there are $\ell - |X|$ gap characters in X'. In the alignment of X and Y, we say that x_i *matches* y_i if $x_i = y_i$; x_i is *substituted* by y_i if $x_i \neq y_i$ and both are not $*$; y_i is *inserted* if $x_i = *$; x_i is *deleted* if $y_i = *$.

A sequence of ℓ aligned pairs $(x_1, y_1), (x_2, y_2), \cdots, (x_\ell, y_\ell)$ is called a *gap sequence* if either all x_i equal $*$ or all y_i equal $*$. The sequence is called a *gap-free sequence* if none of the x_i nor y_i equals to $*$. In other words, an alignment with α gaps can be viewed as $z_0 g_0 z_1 g_1 ... z_{\alpha-1} g_{\alpha-1} z_\alpha$ where z_0 is a possibly empty gap-free sequence, $z_1 ... z_\alpha$ are non-empty gap-free sequences, and $g_0 ... g_{\alpha-1}$ are gap sequences.

Given two strings X and Y, we can measure the quality of an alignment of X and Y by a score function $\delta(\cdot)$. For any letters x and y in $\Sigma \cup \{*\}$, $\delta(x, y)$ gives the *score value* measuring the similarity between them. We assume that $\delta(x, x)$ is higher than $\delta(x, y)$ for $x \neq y$. The score between two strings X and Y, denoted by $\delta(X, Y)$ is defined as the sum of $\delta(x_i, y_i)$ over all i. E.g., setting $\delta(x, x) = 1$ and $\delta(x, y) = 0$ for $x \neq y$ simply counts how many matches we have.

In addition, we distinguish one gap of a certain length and two gaps with the same total length as the one gap by introducing a gap opening penalty. The score of an alignment is defined taking into account the gap opening penalty for each gap in the alignment. Now we are ready to define the pair-wise sequence alignment problem with bounded number of gaps.

Definition 1: Given a text T of length n, a pattern of length $m < n$, and an integer $k > 0$, the problem is to find all prefixes T' of T where the corresponding alignment of T' and X in the form $z_0 g_0 z_1 g_1 ... z_{\alpha-1} g_{\alpha-1} z_\alpha$ satisfies the property that $\alpha \leq k$ and the score is the maximum.

For example, consider the text = TCGTTA and the pattern = TCTA. The following figures show valid 0, 1, and 2 gap alignments for them.

Fig. 1: 0-gap alignment

```
T C G T T A
| | − −
T C T A
```

Fig. 2: 1-gap alignment

```
T C G T T A
| |     | |
T C * * T A
```

Fig. 3: 2-gap alignment

```
T C G T T A
| |   |   |
T C * T * A
```

B. Dynamic Programming Algorithm

Adapting the dynamic programming algorithm in [13] to allow general score function, our CPUGapsMis algorithm is based on the following dynamic programming framework. We keep a matrix $G_g[i, j]$, for $0 \leq g \leq k$, which stores the maximum alignment score between the prefixes $t_1 t_2 \cdots t_i$ of the text T and $p_1 p_2 \cdots p_j$ of the pattern P. Here we denote by $\eta < 0$ the so called *gap opening penalty* which represents the penalty of the existence of a gap. We assume that the *gap extension penalty* is the same regardless of which letter is aligned with the gap character, i.e., there exists a constant γ such that $\delta(x, *) = \delta(*, x) = \gamma$ for all $x \in \Sigma$.

Note that the restriction on the number of gaps can be observed by calculating the matrix up to G_k.

$$G_0[i, j] = \begin{cases} G_0[i-1, j-1] + \delta(t_i, x_j) & \text{if } i = j \\ -\infty & \text{otherwise} \end{cases}$$

$$G_g[i, j] = \max \begin{cases} G_g[i-1, j-1] + \delta(t_i, x_j) \\ \max\limits_{r=1}^{j-i} G_{g-1}[i, j-r] + \eta + \sum\limits_{l=j-r+2}^{j} \delta(*, x_l) \\ \qquad \text{if } i < j \\ \max\limits_{r=1}^{i-j} G_{g-1}[i-r, j] + \eta + \sum\limits_{l=i-r+2}^{i} \delta(t_l, *) \\ \qquad \text{if } i > j \end{cases}$$

III. Our Solution

A. Idea of Parallelization

Our algorithm takes as input a set of q text sequences $\mathcal{T} = T_1, T_2, \cdots, T_q$, a set of r pattern sequences $\mathcal{P} = P_1, P_2, \cdots, P_r$, and a scoring matrix. This means that the whole experiment contains $q \times r$ sequence pairs where a *sequence pair* refers to the alignment of two sequences, $T \in \mathcal{T}$ and $P \in \mathcal{P}$. For simplicity, we discuss the case when the length of text and pattern sequences is n and m, respectively.

Examining the dynamic programming algorithm in details reveals its *non-serial monadic* nature. Therefore, we are able to compute the alignment score matrices $G[]$ in a row-wise fashion, with each cell in a row being independent of all others in that row. This enables us to exploit a good level of parallelism, having all threads in a block calculate for the entirety of a row in one step.

Suppose there are b threads in a block. In the case of $b < m$, the computation of a row is *tiled*, so that the number of steps for a block to calculate a row is $\lceil \frac{m}{b} \rceil$, with any threads not calculating for a cell lying idle. Notice that there are idle threads only in the final tile step for any row.

To align each sequence pair, we initialize the matrices for 0 gaps, and then enter a loop for calculation of 1 up to k gaps. The alignment score is stored in the device global memory, and the backtracking data is sent to host-pinned memory on a *zero copy memory buffer*, which requires no explicit calling and is able to operate asynchronously from the kernel execution, therefore saving execution time and device storage space.

To further save device storage space, we do not keep all the entries of the alignment score matrices. In the shared memory, we store the entries of the current two rows in the $G[\,]$ matrix of a certain gap number (instead of all rows); while in the global memory, we store the values of two $G[\,]$ matrices of the current gap number and the last gap number. Therefore, after completing the calculation of one row, the previous row data can be erased and replaced by the current row (which in the next iteration becomes the previous row). With this rolling method, the device storage usage can be kept to a minimal.

B. AGPU Analysis

The AGPU model [15] consists of a *Host* (CPU) and a *Device* (GPU).

The device consists of:

- p *cores*.
- one *global memory unit*.
- h *multiprocessors*

The h multiprocessors contain the following:

- b cores
- a *shared memory unit* of size M words, divided amongst b memory banks

Let $MP[1,...h]$ be the set of multiprocessors in the AGPU machine, $CORE[1,...,b]$ be the set of cores within each multiprocessor, $\mathcal{T} = T_1, T_2, ..., T_q$ be the set of texts - each of length n, $\mathcal{P} = P_1, P_2, ..., P_r$ be the set of patterns - each of length $m < n$, $k > 0$ be max number of gaps, $numTiles = \lceil \frac{m}{b} \rceil$, $previousG, currentG$ be pointers to GPU Global Memory, $previousRow, currentRow$ be pointers to Shared memory within each multiprocessor, and H be a pointer to host-pinned memory, on a zero-copy buffer.

The pseudo code is given as follows; Theorem 1 gives analysis of GPUGapsMis on the AGPU model.

Theorem 1: The performance of GPUGapsMis on the AGPU model satisfies the following properties.

(i) Time complexity is $O(kn\frac{m}{b})$;
(ii) Shared Memory used is $O(m + b)$;
(iii) Global Memory used is $O(nmh)$;
(iv) I/O complexity is $O(nmhk)$.

Our experiments in Section IV analyze in detail the running time of our algorithm. In particular, Section IV-D discusses that the experimental results match the theoretical analysis in Theorem 1 (i). For the global memory used, as to be shown in the pseudo code, the amount of space required for storing the dynamic programming arrays is proportional to nmh, the text sequence proportional to n, the pattern sequences proportional to hm, and the score matrix is constant. Therefore, the total memory used is $O(nmh)$. On the other hand, shared memory is used during the alignment of one sequence pair and requires size proportional to m as well as some overhead for each core, hence $O(m+b)$. As for the I/O complexity we need to access all data in the global memory for each gap value, and hence with a blow up of k total to $O(nmhk)$.

```
for each $t_a \in \mathcal{T}$ do
    Copy $t_a$ to GPU global memory
    Split $\mathcal{P}$ into $f$ batches $F = F_1, ..., F_f$
    for each $F_c \in F$ do
        Copy $F_c$ to GPU global memory
        //$F_c$ contains $h' \leq h$ patterns from $\mathcal{P}$
        for all $\rho \in MP[1, ..., h']$ do in parallel
            //Multiprocessor $\rho$ will align $F_\rho$ with $t_a$
            for all $\epsilon \in CORE[1, ..., b]$ do in parallel
                //Initialise $G_0$ matrix
                //Initialise $G_0[0,0]$:
                Initialise $G_0[0, \epsilon - 1]$ into
                  $previousRow[\epsilon - 1]$
                //Initialise rest of 0th Row
                for $tile = 0 \to numTiles$ do
                    $j = tile * b + \epsilon$
                    if $j \leq m$ then
                        Initialise $G_0[0, j]$ into
                          $previousRow[j]$
                    end if
                end for
                Copy $previousRow$ to
                  $previousG[0, 0...m]$
                //Initialise remainder rows
                for $i = 1 \to n + 1$ do
                    Initialise $G_0[i, \epsilon - 1]$ into
                        $currentRow[\epsilon - 1]$
                    for $tile = 0 \to numTiles$ do
                        $j = tile * b + \epsilon$
                        if $j \leq m$ then
                            Initialise cell $G_0[i, j]$ into
                                $currentRow[j]$
                            //Value of $G_0[i, j]$ is either $-\infty$
                            or $previousRow[j-1] + score$
                        end if
                    end for
                    Copy $currentRow$ to
                        $previousG[i, 0...m]$
                    Swap $currentRow$ and
                        $previousRow$ pointers
                end for
                Calculate for upto $k$ gaps
                for $g = 1 \to k$ do
                    //Initialise 0th Row
                    //Initialise $G_g[0, 0]$:
                    Initialise $G_g[0, \epsilon - 1]$ into
                        $previousRow[\epsilon - 1]$
                    //Initialise rest of 0th Row
                    for $tile = 0 \to numTiles$ do
                        $j = tile * b + \epsilon$
                        if $j \leq m$ then
                            Initialise cell $G_g[0, j]$ into
                                $previousRow[j]$
                        end if
                    end for
                    Copy $previousRow$ to $currentG[0, 0...m]$
                    for $i = 1 \rightarrow n + 1$ do
                        Initialise cell $G_g[i, \epsilon - 1]$ into $currentRow[\epsilon - 1]$
                        for $tile = 0 \rightarrow numTiles$ do
                            $j = tile * b + \epsilon$
                            if $j \leq m$ then
                                Look in $previousG$ for the optimal gap insertion point
                                Look in $previousRow[j-1]$ for optimal precomputed solution for no gap inserted
                                Calculate optimal score for $G_g[i, j]$
                                Place in $currentRow[j]$
                                Place output data in $H$
                            end if
                        end for
                        Copy $currentRow$ to $currentG[i, 0...m]$
                        Swap $currentRow$ and $previousRow$ pointers
                    end for
                    Swap $currentG$ and $previousG$ pointers
                end for
            end parallel for
        end parallel for
    end for
end for
```

IV. Experimental Results

A. Setting

The machine which was used to conduct the experiments was of the following specification: *AMD A10-5800K APU*, *NVIDIA GTX 650 GPU*, *16 GB memory*, *Ubuntu 14.04.2 LTS x86_64 Operating System*. The AMD A10-5800K APU contains 4 cores at a maximum clock speed of 4.3 GHz . The NVIDIA GTX 650 contains 2 multiprocessors (384 CUDA Cores) with a maximum clock speed of 1.06 GHz and a global memory size of 1024MB.

The experiments conducted, as detailed below, run both alignment algorithms CPUGapsMis and GPUGapsMis, outputting the alignment score data, ready for the backtracking phase using GapsPos [13] – the algorithm presented with GapsMis which calculates the optimal alignment path (see Section V for further discussion about this).

B. Data

The sequence data we used is taken from the NCBI DNA sequence database *GenBank* [29]. From the database, we choose from a selection of genomic data including *e.coli*, *Ralstonia solancearum* and others. We randomly select sequences from the database and further process each sequence by randomly removing some bases such that the length of the sequence becomes the length of the specific experiment sequence pair. This process produces synthetic data, yet since it is taken from real data, it is more realistic than that which is randomly generated (it is much more difficult to generate accurate and realistic patterns). The synthetic data used will give a good view of the algorithms performance with real sequence data, as all data is treated identically by the algorithm.

For experiments, we consider different input sets of text sequences and pattern sequences and for each set of sequences, we measure the performance of aligning all the sequence pairs in the set. E.g., for an input set of q text sequences and r pattern sequences, we align all $q \times r$ sequence pairs. We fix the text sequence length to be 250bp for all input sets and vary the pattern sequence length in different input sets with the values $\{50, 100, 150, 200\}$ bp. Each input set is made up of 100 pattern sequences and the certain number of text sequences which varies with $\{16, 32, 64, 100\}$ in different input sets, and hence the number of sequence pairs in each input set takes the value $\{1600, 3200, 6400, 10000\}$, respectively. As we vary both the pattern sequence length and the number of text sequences, there are altogether 16 input sets.

The sequences are stored in files. There are four input files for text sequences, each file contains $16, 32, 64, 100$ sequences, respectively (each text sequence have length 250bp). There are four input files for pattern sequences, the length of pattern sequences in each file is $50, 100, 150, 200$ bp, respectively (each pattern file contains 100 pattern sequences). Each input set is formed by taking one text sequence file and one pattern sequence file. Due to the way we organize the files, there are four input sets with 1600 sequence pairs, four with 3200 pairs, four with 6400 and four with 10000.

C. Conduct of experiments and measurements

The software first reads the input files containing text sequences, pattern sequences and the score matrix. It then processes the data and creates a mapping of the texts and patterns to the alphabet using integer values. Once this processing of the input data is completed, the experiment commences. To evaluate the performance, we compare the "latency" (time taken) as well as the "throughput" (amount of output per unit time) on both the CPU and GPU. *Latency* is measured as the total time taken from the point where the input processing finishes to the point where the alignment for all sequence pairs has completed. The splitting of sequence pairs into any required batches, and the setting up of the relevant environment is included within this measurement. *Throughput* is a measure of how fast the output data matrix is filled and is measured in Giga Cell Updates per Second (GCUPS). Precisely throughput is calculated by dividing the total amount of output data by the time taken to compute them.

To give a fair comparison between the GPU and the CPU, CPUGapsMis uses OpenMP to provide multithreadding on the CPU. To this end, we use two threads in CPUGapsMis, enabling the CPU to concurrently execute two alignment tasks. This matches the capability of the GPU, which executes one alignment task on each of its two multiprocessors. In order to provide wider comparison, we also conduct experiments with CPUGapsMis using one and four threads.

D. Results and Discussion

In this section we discuss the results of the experiments. We verify the correctness of the results via comparison with the output from GapsMis [13]. The experiments reveal that GPUGapsMis outperforms CPUGapsMis by a good factor. The results are shown in Figures 4, 5, 6, 7, and Table I.

We first consider the results for 10,000 sequence pairs (i.e., 100 pattern sequences and 100 text sequences) in Figures 4 and 5. Both figures show how the measurement varies with the increase in pattern length. Figure 4 shows that the latency of GPUGapsMis is consistently smaller than that of CPUGapsMis for all different pattern length even when four threads are used. The figure, as expected, also shows that the latency increases (linearly) with the increase in pattern length. This agrees with the theoretical results in Theorem 1 for GPUGapsMis. We also note that the increase in latency versus pattern length is more gentle for GPUGapsMis and also for CPUGapsMis with more threads. On the other hand, Figure 5 shows that the throughput stays stable as we increase the pattern length.

We then consider the results for a fixed pattern length of 200 and varying the number of sequence pairs in comparison (Figures 6 and 7). We see that similar to varying the pattern length, the latency increases with the number of sequence pairs as more data is to be handled. The throughput stays stable with increasing number of sequence pairs.

Table I shows all the results of our experiments. A closer look will illustrate that the trends mentioned above hold for all the experiments we have done. Furthermore, we have calculated the speedup of GPUGapsMis relative to the CPUGapsMis, i.e., for the third row (labeled "Speedup on GPU") of the table for each of the input sets, we calculate the ratio of the latency of CPUGapsMis to that of GPUGapsMis. For example, for 1600 sequence pairs and pattern length 50, the speedup is 35, which equals to 4960/142. The speedup varies in the range of 13-57 depending on the pattern length and the number of CPU threads. The 2-thread experiment providing fairer comparison gives a speedup factor of 20-34 while comparing with 4-threads, the speedup ranges from 13-21.

GPUGapsMis outperforms CPUGapsMis for all experiments performed and on the latency and throughput metrics measured, giving speedup of 13-57 times. We see that the speedup achieved by using GPUGapsMis sits at a relatively stable level for each value of pattern sequence length, thus suggesting that in the current implementation of GPUGapsMis, the performance improvement achieved is linked to the size of the sequences to align. This is likely due to the need for the GPU to *tile* the calculations for sequences over the length of the CUDA Block Size, which is set at 32, meaning that the longer pattern sequences require more GPU execution time per row, as each thread within the block has more work. Likewise,

Fig. 4: Latency (on log-scale) of CPUGapsMis and GPUGapsMis with text length 250bp and varying pattern lengths, for 10,000 sequence pairs.

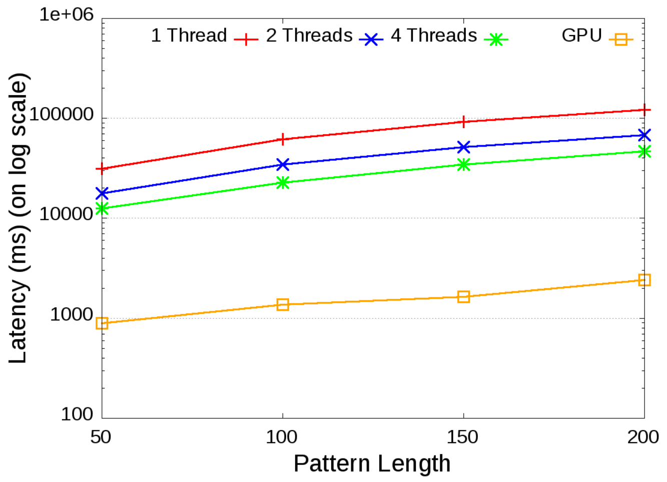
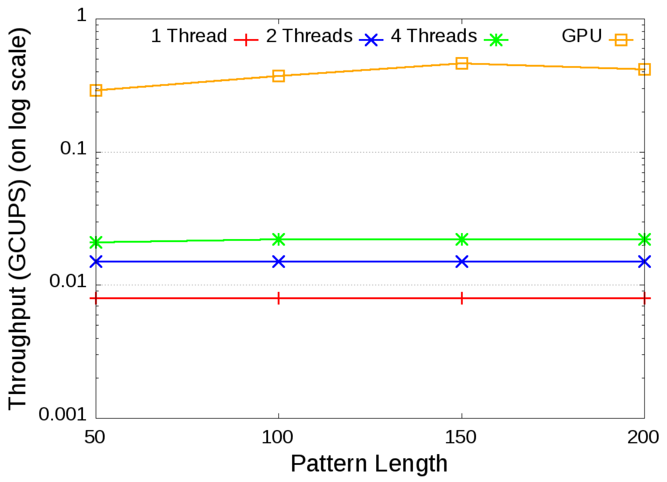


Fig. 5: Throughput (on log-scale) of CPUGapsMis and GPUGapsMis with text length 250bp and varying pattern lengths, for 10,000 sequence pairs.



for CPUGapsMis, as the pattern sequence length is increased, there is naturally more data to process, giving us the expected increase in latency.

One notices that the rate of increase in the latency of GPUGapsMis is of a slower rate than that of CPUGapsMis. This is thanks to the parallelism of GPUGapsMis against the serial execution of CPUGapsMis, with regards to computing for a row. This means that we can expect the performance of GPUGapsMis to scale up very well when faced with larger sets of data, which would otherwise make CPUGapsMis impractically slow. When the sequence length reached 200, we see a decrease in the rate of growth in levels of speedup and increased throughput, which is possibly due to the text-wise execution of GPUGapsMis.

We expect much better results in the future, when

TABLE I: Latency (in milliseconds), Throughput (in GCUPS) for CPUGapsMis and GPUGapsMis, and Speedup achieved by GPUGapsMis over CPUGapsMis)

Pattern Length		50				100			
# Seq pairs	Measurement	# CPU Threads			GPU	# CPU Threads			GPU
		1	2	4		1	2	4	
1600	Latency (ms)	4960	2912	2011	142	9809	5656	3744	217
	Throughput (GCUPS)	0.008	0.014	0.021	0.295	0.008	0.014	0.022	0.374
	Speedup on GPU	35	21	14	1	45	26	17	1
3200	Latency (ms)	9952	5691	3800	285	19667	11103	7689	434
	Throughput (GCUPS)	0.008	0.015	0.022	0.294	0.008	0.015	0.021	0.374
	Speedup on GPU	35	20	13	1	45	26	18	1
6400	Latency (ms)	19815	11323	7691	576	39512	22600	14679	870
	Throughput (GCUPS)	0.008	0.015	0.022	0.291	0.008	0.014	0.022	0.373
	Speedup on GPU	34	20	13	1	45	26	17	1
10000	Latency (ms)	31248	17739	12531	897	61490	34377	22711	1364
	Throughput (GCUPS)	0.008	0.015	0.021	0.292	0.008	0.015	0.022	0.372
	Speedup on GPU	35	20	14	1	45	25	17	1
Pattern Length		150				200			
# Seq pairs	Measurement	# CPU Threads			GPU	# CPU Threads			GPU
		1	2	4		1	2	4	
1600	Latency (ms)	14838	8596	5522	262	19580	11244	7605	412
	Throughput (GCUPS)	0.008	0.014	0.022	0.463	0.008	0.014	0.021	0.392
	Speedup on GPU	57	33	21	1	48	27	18	1
3200	Latency (ms)	29715	17886	10781	520	38941	21949	14445	836
	Throughput (GCUPS)	0.008	0.014	0.022	0.466	0.008	0.015	0.022	0.386
	Speedup on GPU	57	34	21	1	47	26	17	1
6400	Latency (ms)	59059	33097	22154	1044	77844	43533	29311	1654
	Throughput (GCUPS)	0.008	0.015	0.022	0.465	0.008	0.015	0.022	0.390
	Speedup on GPU	57	32	21	1	47	26	18	1
10000	Latency (ms)	91807	51789	34332	1631	121972	68422	46667	2420
	Throughput (GCUPS)	0.008	0.015	0.022	0.465	0.008	0.015	0.022	0.417
	Speedup on GPU	56	32	21	1	50	28	19	1

Fig. 6: Latency (on log-scale) of CPUGapsMis and GPUGapsMis with text length 250bp and pattern length 200bp, for varying number of sequence pairs.

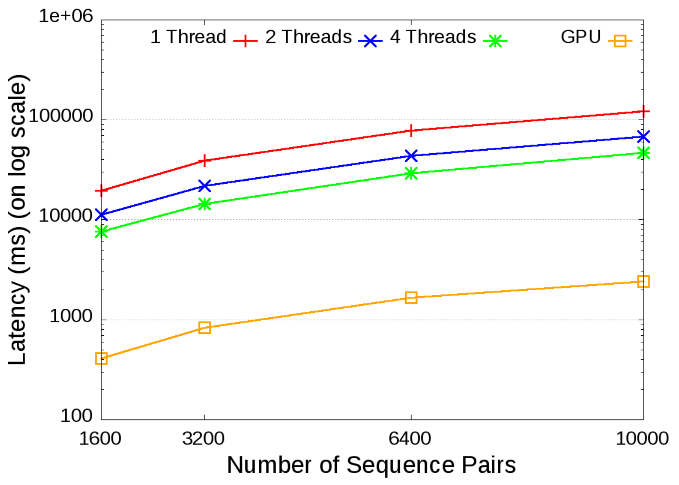
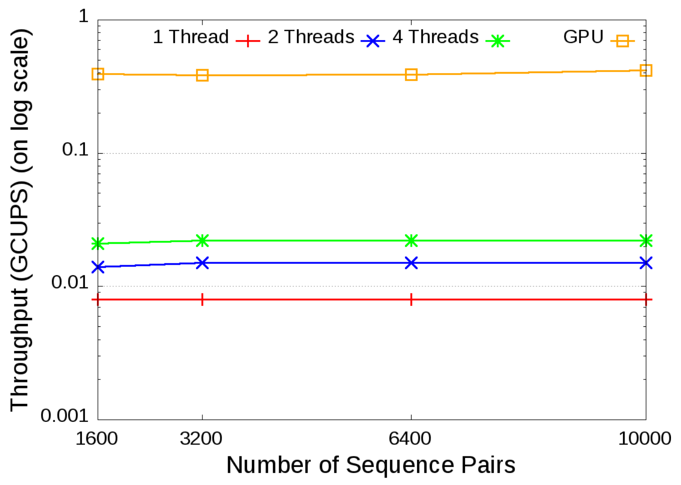


Fig. 7: Throughput (on log-scale) of CPUGapsMis and GPUGapsMis with text length 250bp and pattern length 200bp, for varying number of sequence pairs.



GPUGapsMis is altered to handle sequence pairs of multiple texts on a more flexible basis. These results, though not including the backtracking functionality, clearly demonstrate the suitability of the GPU for the sequence alignment problem with multiple gaps, and give a good indication as to the improvement we may be able to expect once we enable GPUGapsMis to compute the optimal alignment in a backtracking function.

V. Conclusion

In this paper we consider the pair-wise sequence alignment problem with gaps, which is motivated by the resequencing problem. We explain the problem and discuss related work. Single gap and multiple gaps algorithms have been proposed in [11], [12] and [13], respectively. A GPU parallel algorithm has been proposed for the single gap problem [12]. Our contribution is designing a GPU parallel algorithm allowing multiple gap, which is based on the algorithm in [13]. We implement our algorithm and compare with multithreaded CPU algorithm. The results are promising with the GPU version achieving a speedup of 30 times.

This work will first be extended by allowing several batches, with multiple texts, to be concurrently executed on the GPU, to provide not only an increase in throughput, but a decrease in latency under certain conditions, and greater energy efficiency than at present. We will then be providing functionality to perform the computing of the optimal alignment path on the GPU, which we currently perform serially on the CPU. Once we combine GPUGapsMis with the GPU based backtracking algorithm, we expect to see large increases in performance on existing serial solutions to the problem. Further to this, we look to allow alternative gap cost functions, thereby providing a greater amount of biological accuracy to the results as well as retaining the high sensitivity of the pairwise dynamic programming approach. We also would like to investigate the use of multiple GPU devices.

Acknowledgment

The authors thank Solon Pissis for bringing the problem to our attention and discussing ideas on GPU approaches. Thomas C. Carroll is supported by a studentship from the Engineering and Physical Sciences Research Council, UK.

References

[1] "Illumina HiSeqX Ten System," http://www.illumina.com.

[2] B. Langmead, C. Trapnell, M. Pop, S. L. Salzberg *et al.*, "Ultrafast and memory-efficient alignment of short DNA sequences to the human genome," *Genome biol*, vol. 10, no. 3, p. R25, 2009.

[3] H. Li and R. Durbin, "Fast and accurate short read alignment with Burrows–Wheeler transform," *Bioinformatics*, vol. 25, no. 14, pp. 1754–1760, 2009.

[4] K. Schneeberger, J. Hagmann, S. Ossowski, N. Warthmann, S. Gesing, O. Kohlbacher, and D. Weigel, "Simultaneous alignment of short reads against multiple genomes," *Genome Biol*, vol. 10, no. 9, p. R98, 2009.

[5] R. Li, Y. Li, K. Kristiansen, and J. Wang, "SOAP: short oligonucleotide alignment program," *Bioinformatics*, vol. 24, no. 5, pp. 713–714, 2008.

[6] R. Li, C. Yu, Y. Li, T.-W. Lam, S.-M. Yiu, K. Kristiansen, and J. Wang, "SOAP2: an improved ultrafast tool for short read alignment," *Bioinformatics*, vol. 25, no. 15, pp. 1966–1967, 2009.

[7] S. M. Rumble, P. Lacroute, A. V. Dalca, M. Fiume, A. Sidow, and M. Brudno, "SHRiMP: accurate mapping of short color-space reads," *PLoS Computational Biology*, vol. 5, no. 5, 2009.

[8] G. Lunter and M. Goodson, "Stampy: a statistical algorithm for sensitive and fast mapping of illumina sequence reads," *Genome research*, vol. 21, no. 6, pp. 936–939, 2011.

[9] K. Frousios, C. S. Iliopoulos, L. Mouchard, S. P. Pissis, and G. Tischler, "REAL: an efficient read aligner for next generation sequencing reads," in *Proceedings of the First ACM International Conference on Bioinformatics and Computational Biology*. ACM, 2010, pp. 154–159.

[10] C. M. Liu, T. Wong, E. Wu, R. Luo, S. M. Yiu, Y. Li, B. Wang, C. Yu, X. Chu, K. Zhao, R. Li, and T. W. Lam, "SOAP3: ultra-fast GPU-based parallel alignment tool for short reads," *Bioinformatics*, vol. 28, no. 6, pp. 878–879, 1 2012.

[11] T. Flouri, K. Frousios, C. S Iliopoulos, K. Park, S. P Pissis, and G. Tischler, "GapMis: a tool for pairwise sequence alignment with a single gap," *Recent patents on DNA & gene sequences*, vol. 7, no. 2, pp. 84–95, 2013.

[12] N. Alachiotis, S. Berger, T. Flouri, S. P. Pissis, and A. Stamatakis, "libgapmis: extending short-read alignments," *BMC bioinformatics*, vol. 14, no. Suppl 11, p. S4, 2013.

[13] C. Barton, T. Flouri, C. S. Iliopoulos, and S. P. Pissis, "GapsMis: flexible sequence alignment with a bounded number of gaps," in *Proceedings of the International Conference on Bioinformatics, Computational Biology and Biomedical Informatics*, ser. BCB'13, 2013, pp. 402–411.

[14] P. Rice, I. Longden, A. Bleasby *et al.*, "Emboss: the european molecular biology open software suite," *Trends in genetics*, vol. 16, no. 6, pp. 276–277, 2000.

[15] A. Koike and K. Sadakane, "A novel computational model for gpus with applications to efficient algorithms," *International Journal of Networking and Computing*, vol. 5, no. 1, pp. 26–60, 2015.

[16] S. Fortune and J. Wyllie, "Parallelism in random access machines," in *STOC*. ACM, 1978, pp. 114–118.

[17] R. M. Karp and V. Ramachandran, "Parallel algorithms for shared-memory machines," in *Handbook of Theoretical Computer Science (Vol. A)*, J. van Leeuwen, Ed. MIT Press, 1990, pp. 869–941.

[18] J. Jájá, *An introduction to parallel algorithms*. Addison-Wesley, 1992.

[19] N. Sitchinava and V. Weichert, "Provably efficient gpu algorithms," *arXiv preprint arXiv:1306.5076*, 2013.

[20] S. A. Manavski and G. Valle, "CUDA compatible GPU cards as efficient hardware accelerators for Smith-Waterman sequence alignment," *BMC Bioinformatics*, vol. 9, no. Suppl 2, p. S10, 2008.

[21] L. Ligowski and W. Rudnicki, "An efficient implementation of Smith Waterman algorithm on GPU using CUDA, for massively parallel scanning of sequence databases," in *IPDPS*. IEEE, 2009, pp. 1–8.

[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 Bioinformatics*, vol. 14, no. 117, 2013.

[23] G. M. Striemer and A. Akoglu, "Sequence alignment with GPU: Performance and design challenges," in *IPDPS*. IEEE, 2009, pp. 1–10.

[24] P. D. Vouzis and N. V. Sahinidis, "GPU-BLAST: using graphics processors to accelerate protein sequence alignment," *Bioinformatics*, vol. 27, no. 2, pp. 182–188, 2011.

[25] K. Zhao and X. Chu, "G-BLASTN: accelerating nucleotide alignment by graphics processors," *Bioinformatics*, vol. 30, no. 10 2014, pp. 1384 – 1391, 1 2014.

[26] L. S. Yung, C. Yang, X. Wan, and W. Yu, "GBOOST: a GPU-based tool for detecting gene–gene interactions in genome–wide case control studies," *Bioinformatics*, vol. 27, no. 9, pp. 1309–1310, 2011.

[27] K. J. Kohlhoff, M. H. Sosnick, W. T. Hsu, V. S. Pande, and R. B. Altman, "CAMPAIGN: an open-source library of GPU-accelerated data clustering algorithms," *Bioinformatics*, vol. 27, no. 16, pp. 2321–2322, 2011.

[28] A. Bustamam, K. Burrage, and N. A. Hamilton, "Fast parallel Markov clustering in bioinformatics using massively parallel computing on GPU with CUDA and ELLPACK-R sparse format," *IEEE/ACM Transactions on Computational Biology and Bioinformatics (TCBB)*, vol. 9, no. 3, pp. 679–692, 2012.

[29] "NCBI GenBank," http://www.ncbi.nlm.nih.gov/genbank/.