# Semiglobal Sequence Alignment with Gaps Using GPU

Thomas C. Carroll[1], Jude-Thaddeus Ojiaku[2], and Prudence W.H. Wong[1]

[1]Department of Computer Science, University of Liverpool, Liverpool, UK
{thomas.carroll,pwong}@liverpool.ac.uk
[2]ASML BV, Veldhoven, The Netherlands
jude.ojiaku@asml.com

**Abstract**—In this paper we consider the pair-wise semiglobal 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 is a GPU-based algorithm proposed (Barton et al., 2015). In our work we propose a GPU-based algorithm for the bounded number of gaps case, called GPUGapsMis. We implement the algorithm and compare the performance with the CPU-based algorithm, called CPUGapsMis; The algorithm has two distinct stages: the *alignment phase*, and the *backtrack phase*. We investigate several different approaches, in order to determine the most favourable for this problem, by means of a Hybrid model or a wholly-GPU based model, as well as the alignment of single text sequences or multiple text sequences on the GPU at a time. We show that the alignment phase of the algorithm is a good candidate for parallelisation, with peak speedup of 11 times. We show that although the backtracking phase is sequential, it is more beneficial to perform it the GPU, as opposed to returning to the CPU and performing there. When performing both phases on the GPU, GPUGapsMis achieves a peak speedup of 10.4 times against CPUGapsMis. Our data parallel GPU algorithm achieves results which are an improvement on those of an existing GPU data parallel implementation (Ojiaku, 2014).

**Index Terms**—Graphics processors; Parallel programming; Data communications aspects; Bioinformatics (genome or protein) databases;

✦

## 1 INTRODUCTION

IN THIS paper we consider the pair-wise semiglobal 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 is a task parallel GPU-based algorithm proposed [1], and there is a data parallel GPU-based algorithm [24] for the bounded number of gaps case. In our work we propose a new data parallel GPU-based algorithm for the bounded number of gaps case. We implemented the algorithm and compare the performance with the CPU-based algorithm; The algorithm has two distinct stages: the *alignment phase*, and the *backtracking phase*. We investigate several different approaches, in order to determine the most favourable for this problem, by means of different methods of batching work on the GPU and different methods of performing the backtracking phase of the algorithm. We show that the alignment phase of the algorithm is a good candidate for

parallelisation, with peak speedup of 11 times. We show that despite the backtracking phase being a bad candidate for parallelisation, it is more beneficial to perform it on the GPU, as opposed to performing on the CPU. We achieve a peak speedup of 10.4 times when the backtracking is also performed. Our results are comparable to the task parallel GPU implementation [1] for the single gap case, and show improvement on the data parallel GPU implementation for the multiple gaps case [24].

### 1.1 Sequence Alignment Problem

The problem of finding alignment between two biological sequences has been extensively studied, with the two most famous alignment algorithms being the Smith-Waterman algorithm [30] and the Needleman-Wunsch algorithm [21]. 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 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 [11], BWA [12], GenomeMapper [28], MAQ [13], SOAP2 [14], SHRiMP [27], Stampy [18], REAL [8], 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 such as the GPU. There are many alignment tools available, which use the GPU in order to achieve increase in speed and SOAP3 [16] 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 penalise the occurrence of gap as a whole instead of individual alternations [3]. 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.

Because of the importance of gaps, the alignment problem has been considered in the presence of gaps [3]. 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 [3], a single gap is allowed and the algorithm GapMis is proposed; the case of multiple gaps is also considered 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 [26] and EMBOSS needle [26]. With single gap, a tool called libgapmis using GPU and a *task parallel* approach has been developed in [1] for which an $11\times$ speedup has been reported. With multiple gaps, there is a data parallel algorithm [24] for which a 5 times speedup has been reported, and a data parallel algorithm previously presented by the authors [5], yet this was subsequently found to not compute the optimal solution. Note that GapsMis is not a replacement for Bowtie [11] or BWA [12] but its significance has been established through comparison with EMBOSS water [26] and EMBOSS needle.

## 1.2 The GPU

We briefly discuss the nVidia Compute Unified Device Architecture (CUDA) [23] GPUs, as these are the most popular for scientific computing. The GPU is a massively parallel device, with many low powered processing elements. It is often used as a coprocessor for scientific applications and is connected by the PCIe bus to the CPU. On the GPU, there is the *chip*, and various units of *global memory*. The global memory is often in the order of *gigabytes* in size with access possible both from the CPU and the GPU.

Within the chip, there is a number of *streaming multiprocessors* (SM) (this number depends on the model of GPU). The SM is, from a programming perspective, the main component of the GPU; therefore is quite important to understand. Each SM has lanes of *processing elements* (PE), along with a *shared memory unit*, accessible only to the PEs on that SM. The programmer writes *kernels* (analogous to methods) in the CUDA C programming language, and uses the CUDA API to send both data and the kernel to the GPU. The programmer specifies the launch configuration of the kernel, as a *grid* of *thread blocks* on the GPU. First, the program data and input data is sent to the GPU over the PCIe bus. These transfers are the slowest of the entire program. After these have been transferred, the execution of the kernel will begin.

A *thread block* is a collection of threads which work in cooperation and are run on a single SM, in a single instruction multiple data (SIMD) fashion, with inter-thread communication only possible via shared memory, accessible only to the threads of the thread block. The thread block *conceptually* runs concurrently, yet in reality is divided into *warps*, which are arrays of 32 threads, each run in lock step with one another. The instructions of the kernel for each warp are placed in an instruction queue, and are scheduled for execution on lanes of CUDA cores. Once the instruction has executed, there may be the need to *wait* on a shared memory request or a global memory request. Once the request has been serviced, the next instruction is ready to be scheduled for execution. When a shared memory request is placed by a warp, it is serviced in unit time should each address be within distinct banks. If this is not the case, then a *bank conflict* occurs, and the request is serialised by the hardware into as few non-conflicting requests as possible. When a global memory request is placed by the warp, then it is put into as few memory-block-wide transactions as possible. If all requests by the warp are for addresses within the same memory block, then this is serviced by a single transaction, this is known as memory *coalescing*. Accessing global memory is very expensive, taking up to 800 cycles per block, therefore

it is wise to access global memory with as much coalescing as possible, otherwise the global memory access can throttle a programs performance.

Once an operation has been executed by a warp, the next instruction (possibly for a different warp) in the instruction queue is then scheduled for execution. It is possible to have multiple thread blocks resident on a single SM, provided there are enough shared memory resources for them to execute. CUDA has several generations of architectures, with the devices used for experiments in this paper being from the Kepler architecture [22]. In the Kepler architecture, the maximum amount of thread blocks able to be resident on an SM is 16. If a program is then able to hold 16 blocks on an SMX, it is said to have full *occupancy*. This then means that there are many warps available to execute whist other long-latency operations are being serviced, which will go to hide the latency of these said long latency operations. Due to the way that the instruction queue is populated with ready-state instructions, it is important to ensure that each warp is independent of the rest. It is however possible to synchronise the threads within the thread block, using barrier operations. Likewise, blocks must be independent of one another, though currently, the best way to synchronise blocks with one another is to terminate a kernel and relaunch.

### 1.3 AGPU Model

Most of the existing work on using GPU evaluates these algorithms empirically. Recently, Koike and Sadakane [10] proposed a theoretical model for GPUs called the Abstract GPU model (AGPU). Since known parallel computational models such as the PRAM model are not appropriate for evaluating GPU-based algorithms, it is necessary to have new theoretical model to capture the essence of GPU architectures. Using the AGPU model, it is possible to analyse the asymptotic time complexity of GPU algorithms.

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 blocks accessed 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 block 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 analyse the performance of `GPUGapsMis` based on the AGPU model and present it in Theorem 1.

### 1.4 Biological Problems on GPU

A GPU program must operate in an SIMD fashion, meaning that the same operation is performed in parallel upon different data items. This can lend itself favourably to various bioinformatics tasks, particularly some forms of sequence alignment, where the operations required for each cell follow a strict pattern, and the data dependencies for each cell are in the same relative location to the current cell. It is also important to be able to draw out enough parallelism from the problem, which can be obtained either in a *task parallel* manner, whereby many tasks are parallelised in a thread block, or in a *data parallel* manner, whereby a single tasks is parallelised by a thread block.

Various bioinformatics problems have been tackled using GPU-based algorithms, including BLAST (the Basic Local Alignment Search Tool) [32, 34], the Smith-Waterman global alignment algorithm [15, 17, 19, 31], Needleman-Wunsch local alignment [7, 25] ([29] studies GPU implementation of Smith-Waterman and Needleman-Wunsch with focus towards a hybrid model) and others [4, 9, 33].

### 1.5 Our Contribution

Our contribution is a study of our proposed data-parallel GPU-based algorithm for the pair-wise sequence alignment problem with multiple gaps. The algorithm, which we call `GPUGapsMis`, is based on the GapsMis and GapsPos algorithms in [3], each for the alignment and backtracking functionality, respectively. We give analysis of `GPUGapsMis` on the AGPU model, and give analysis of of observed results with respect to the different approaches.

To achieve greater improvement over the CPU, we try to maximise 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 cause a bottleneck in performance. To allow flexibility of dealing with real data, we also extend the algorithm to allow the use of scoring matrix in addition to the Hamming distance that is considered in GapsMis [3]. We implement our algorithm and a modified version of the sequential algorithm GapsMis with the scoring matrix; we call the extended algorithm `CPUGapsMis`. We also enable the functionality to compute the optimal alignment, as in GapsPos [3], and investigate using a Hybrid backtracking method and a GPU backtracking method. Further to this, we investigate allowing a single text and multiple text sequences to be aligned on the device at one time, with different batching methods.

We compare the performance of `GPUGapsMis` and `CPUGapsMis` and the speed up is 11 times in computing the alignment score matrix, and 10.4 times when the backtracking is also computed. We show that by lowering the amount of communication and data transfer between the GPU and CPU, we are able to yield the most improvement. We also show that despite the backtracking being sequential and inefficient on the GPU (when compared to performing the backtracking on the CPU), it is more beneficial to perform this on the GPU, rather than returning to the CPU for performing the backtracking.

**Organisation of Paper.** The remainder of this paper is organised as follows: Section 2 gives notations required and the problem definition; Section 3 details our proposed solution; Sections 4 and 5 detail experimental evaluation and discuss the results obtained; Finally, Section 6 concludes the paper.

## 2 PROBLEM DEFINITION AND PRELIMINARIES

### 2.1 Notations

We introduce some notations required for the definition of the problem. Consider an alphabet $\Sigma$. 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 $*$ represent the gap character and $\Sigma' = \Sigma \cup \{*\}$. *An aligned pair* is a pair of letters $(x, y)$ such that $(x, y) \in \Sigma' \times \Sigma' \setminus \{*, *\}$. In other words, an aligned pair may involve at most one gap character. An alignment of two strings $X$ and $Y$ is a string of aligned pairs $(x_1, y_1), (x_2, y_2), \cdots, (x_\ell, y_\ell)$ such that removing all the gap characters $*$ from $x_1 x_2 \cdots x_\ell$ gives $X$ (similarly for $Y$). Note that there are $\ell - |X|$ gap characters in the alignment. 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 $\ell$ 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 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. In this case, the alignment has $\alpha$ gaps.

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* which measures 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 $i$. For example 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 by introducing a gap opening penalty and a gap extension penalty, where the gap opening penalty is applied for the first gap character to be inserted in a gap sequence, and the gap extension penalty is applied for each subsequent gap character inserted to the gap sequence. We assert that the gap opening penalty $\delta_P < 0$ is less than the gap extension penalty, $\delta_E < 0$, and that: $\forall \sigma \in \Sigma : \delta(\sigma, *) = \delta(*, \sigma) = \delta_E$. For a gap of length $l$, the gap penalty is calculated as $\delta_P + \delta_E(l - 1)$ The score of an alignment is calculating by adding the scores of all gap sequences and gap-free sequences in the alignment.

### 2.2 Problem Definition

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 $X$ 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.*

Figure 1 shows example alignments. We are required to find the prefixes of text $T$ which satisfy the properties described, because we use the *seed and extend* strategy [2] for alignment, whereby a high quality alignment seed (at the start of the sequences) is matched, and the alignment is then extended. This involves aligning prefixes of the text $T$ with the entirety of the pattern $X$, known as a *semi-global* alignment. This is as opposed to a *global* alignment, which aligns the entirety of $T$ and $P$, and opposed to a *local* alignment, which aligns substrings of both $T$ and $P$.

### 2.3 Dynamic Programming Algorithm

Adapting the dynamic programming algorithm in [3] to allow general score function, our algorithm is based on the following dynamic programming framework. We keep a matrix $G_q[i, j]$, which stores the maximum alignment score between the prefixes $t_1 t_2 \cdots t_i$ of the text $T$ and $x_1 x_2 \cdots x_j$ of the pattern $X$, allowing up to $q$ gaps, where $0 \leq q \leq k$. 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 $\delta_E$ such that $\delta(x, *) = \delta(*, x) = \delta_E$ for all $x \in \Sigma$.

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

$$G_0[i, j] = \begin{cases} 0 & \text{if } i = j = 0 \\ G_0[i-1, j-1] + \delta(t_i, x_j) & \text{if } 1 \leq i = j \leq m \\ -\infty & \text{if } i \neq j \text{ and } 0 \leq i \leq n \text{ and } 0 \leq j \leq m \end{cases}$$

$$G_q[i, j] = \max \begin{cases} 0 & \text{if } i = j = 0 \\ \delta_P + \sum_{l=0}^{j-2} \delta(*, x_l) & \text{if } i = 0 \text{ and } 1 \leq j \leq m \\ \delta_P + \sum_{l=0}^{i-2} \delta(t_l, *) & \text{if } j = 0 \text{ and } 1 \leq i \leq n \\ \max_{r=1}^{j-1} (G_{q-1}[i, j-r] + \delta_P + \sum_{l=j-r+2}^{j} \delta(*, x_l)) \\ \qquad \text{if } 1 \leq i \leq n \text{ and } 1 \leq j \leq m \\ \max_{r=1}^{i-1} (G_{q-1}[i-r, j] + \delta_P + \sum_{l=i-r+2}^{i} \delta(t_l, *)) \\ \qquad \text{if } 1 \leq i \leq n \text{ and } 1 \leq j \leq m \\ G_q[i-1, j-1] + \delta(t_i, x_j) \\ \qquad \text{if } 1 \leq i \leq n \text{ and } 1 \leq j \leq m \end{cases}$$

A naïve implementation of the dynamic programming recurrences would result in an algorithm of $O(knm(n + m))$ time, yet it was demonstrated in [3] that storing the information of the gap insertion points (the value of $r$ which maximises the scores on lines 3 and 4 of the recurrence) would make the look-up possible in $O(1)$ time, giving an improved time complexity of $O(knm)$.

We keep a matrix $H_q$ which stores information on gap length and placement (at which position and in which sequence does the gap occur), for the alignment up to and including the pair $(t_i, x_j)$ which includes at most $q$ gap sequences, for $0 \leq q \leq k$. The cells are populated as shown in the recurrence, with $H_q[i, j]$ being populated after $G_q[i, j]$ has been calculated.

$$H_q[i, j] = \begin{cases} 0 & (t_i, x_j) \text{ in alignment} \\ r > 0 & (t_i, *) \text{ in alignment, gap of } r \\ r < 0 & (*, x_j) \text{ in alignment, gap of } r \end{cases}$$

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

(a) 0-gap alignment, score 10 │ (b) 1-gap alignment, score 16 │ (c) 2-gap alignment, score 14

Fig. 1: Valid alignments for text $TCGTTA$ and pattern $TCTA$, where $\delta_P = -3$, $\delta_E = -1$, $\delta(i,i) = 5$, $\delta(i,j) = 0$, where $i \neq j$.
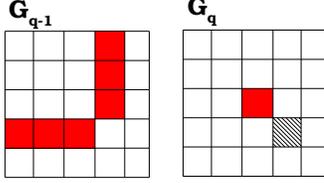


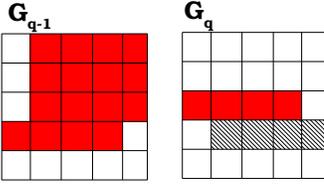Fig. 2: The dependencies whilst calculating cell $G_q[i,j]$ (hatched) are shown in solid filled cells.



Fig. 3: The dependencies whilst calculating the row $G_q[i,*]$ (hatched) are shown in solid filled cells.

The alignment is retrieved using the linear time algorithm *GapsPos* [3]. Starting from the position of the alignment score reported by GapsMis, the alignment is built backwards, moving towards the start of the sequences. The value within each cell of $H_q$ dictates how the row and column indices are adjusted; either both are decremented by one in the case of no gap, or the column index (row index) is decreased by the absolute value of the cell to give a gap in the pattern (text).

## 3 OUR SOLUTION

In the following section we describe `GPUGapsMis`, our solution to the semi-global sequence alignment with bounded gaps problem. We also give theoretical analysis of the proposed solution on the AGPU model.

### 3.1 Idea of Parallelisation

As the recurrence in Section 2.3 shows, the dependencies for the cell $G_q[i,j]$ lie within the cell $G_q[i-1-,j-1]$ and the range of cells $G_{q-1}[0...i, 0...j]$, therefore as shown in Figures 2 and 3, we are able to express parallelism along each row of the dynamic programming matrix in order to create a data-parallel solution. As the dependencies required for calculating cells within $G_q$ all lie either in $G_q$ or $G_{q-1}$, we only require the current and previous one $G$ matrix for computation to be stored.

We keep the following data in the global memory: text sequence data, pattern sequence data, score data and matrices $G_q, G_{q-1}, H$ data for each sequence pair being aligned. Pointers kept in private memory, which point to $G_q$ and

$G_{q-1}$ in global memory, are updated at each iteration of the number of gaps calculated, and the $H$ matrix is only used on the final iteration, as for $q$ gaps, only the data in $H_q$ is required when computing the optimal alignment.

The shared memory space contains the pattern data, the text character for current matrix row $i$, and the buffers required for our aggressive double-buffer technique. This double buffer technique is laid out as follows: $currRow, prevRow$ hold rows $i, i-1$ of $G_q$, $prevGprevRow, prevGcurrRow$ hold rows $i, i-1$ of $G_{q-1}$, along with $maxIVal, maxILoc, maxJVal, maxJLoc$ hold the information relating to optimal gap insertion points from $G_{q-1}$. As with the global memory pointers, $currRow, prevRow, prevGprevRow, prevGcurrRow$ are updated at each row iteration, and filled with any required data. In order to maximise use of global memory access bandwidth, we need to use vectorised memory access operations. In order for vectorised memory accesses to be made possible, we pad with dummy data the shared memory row caches, the patterns, and the matrix rows.

We now explain the intuition behind the parallelisation for a single sequence pair, executed by a single thread block on the GPU. This is repeated for additional sequence pairs in a separate thread block per sequence pair. Initially, the *pattern sequence* is fetched from *global memory* into the shared memory. We calculate matrix $G_0$ followed by $G_1, G_2, ..., G_k, H_k$, for up to $k$ gaps. Each matrix is calculated in a row-wise, data parallel fashion, with parallelism being expressed along each row. As each matrix is being calculated, the row number is iterated, and the number of gaps is iterated.

To calculate a row of $G_q$, we fetch the text character from the global memory, and the relevant gap insertion data relating to $G_{q-1}$. We then initialise the first cell of the row, and proceed to iterate across the row for all threads in a tiling fashion. The data required for the calculation is held in shared memory. At the end of row calculation, we copy the values to global memory and retain in shared memory for the next row, discarding the previous row. At the end of a matrix calculation, the pointers to the current $G$ matrix and previous $G$ matrix are updated, so we using a double buffer approach on several levels.

For a number of gaps $0 < q \leq k$, we calculate the matrices $G_q$ ($H_q$) in the following way, which is explained visually in Figure 4:

- Initialise the first row ($G_q[0, *]$) by storing the values into shared memory $previousRow, hRow$, with each warp of the block taking a tile.
- Store data of $previousRow, hRow$ in global memory.
- Fetch data of $G_{q-1}[0, *]$ from global memory into shared memory $prevGprevRow$, in preparation for calculating the subsequent rows of $G_q$
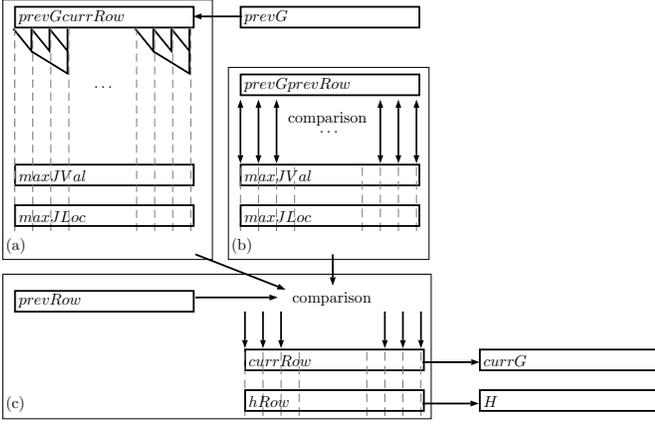- Loop for each row $1 \leq i \leq n$

Fig. 4: Idea of parallelisation for GapsMis. (a) Best gap insertion points in pattern are found. (b) Best gap insertion points in text are updated, if needed (c) Best score is calculated, and placed into global memory.

a) Fetch $G_{q-1}[i, *]$ into shared memory $prevGcurrRow$.

b) Calculate the best gap insertion point into the pattern, for each position $0 \leq j \leq m$, in $O(\log m)$ time. We use a tree-based method for finding the maximal gap insertion point from prevGprevRow. The maximal gap insertion point for $G_q[i, j]$ exists in the range $G_{q-1}[i, 0, ..., j-1]$. We are able to calculate the maximal insertion points for an entire row in the same routine. We calculate, for each position $0 < j < m$ the alignment score and location of the best point, upto but not including $j$ itself. We modify a parallel prefix scan algorithm to use the max operator as opposed to the summation operator to calculate this.

c) Update the gap insertion points into the text, if this is required, by comparing $maxIVal$, $maxILoc$, prevGprevRow.

d) Compare values in shared memory, for the three options of alignment: continue the current alignment ($prevRow$), insert gap in text ($maxIVal$, $maxILoc$), or insert gap in pattern ($maxJVal$, $maxJLoc$). Place optimal value into $currRow$ and relevant gap value into $hRow$. Now place $currRow$, $hRow$ into Global Memory.

e) Update the pointers of ($prevGcurrRow$, $prevGprevRow$) ($prevRow$, $currRow$) in preparation for calculating row $i + 1$

The algorithm $GapsPos$ calculates the optimal alignment path for the two sequences, which we refer to as *backtracking*. GapsPos is performed sequentially using a single thread.

**Difference from existing data-parallel implementation.** Ojiaku [24] proposed a data-parallel solution to this problem, reporting a 5 times speedup against a single thread of the CPU. We evaluate GPUGapsMis using a similar environment as that used in [24]. Our solution differs in that we reduce the amount of host device communication by running for all $k$ gaps in a single kernel run, therefore not requiring any global synchronisation or data transfer between subsequent gap numbers. We also use a parallel tree-based method for finding the optimal gap insertion point, where as [24] uses a sequential method. Further to this, we investigate several approaches to calculating the backtracking, by performing this on the GPU. This is opposed to calculating the backtracking on the CPU only, as in [24].

## 3.2 AGPU Analysis

We now give analysis of GPUGapsMis using the AGPU model [10] which has been discussed in Section 1.3. We present AGPU Pseudocode in Algorithm 1 for GPUGapsMis aligning one sequence pair on a single multiprocessor. This is replicated for all sequence pairs in the input set, with Algorithm 1 corresponding to code run by a single CUDA thread block. Theoretical results are presented in Theorem 1.

The AGPU captures 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

Global memory is accessed with the $\Leftarrow$ operator and shared memory is accessed with the $\leftarrow$ operator. Let $CORE[1, ..., b]$ 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 $m$, where $n \geq m$, $k > 0$ be max number of gaps, $OPEN$ be the gap opening penalty, and $EXT$ be the gap extension penalty.

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

(i) The time complexity is $O(kn\frac{m}{b})$.
(ii) The I/O complexity is $O(qrkn\frac{m}{b})$.
(iii) The global memory usage is $O(hnm)$.
(iv) The shared memory usage is $O(m)$.

*Proof.* We now give a proof of the claims in Theorem 1, with line references to Algorithm 1.

(i) We see that the "Gaps" loop (lines 5-42) iterates $k$ times in total with an additional procedure for initialising $G_0$. We see that the "row" loop (lines 15-41) is iterated $n$ times in total, for all matrices $G_0 \to G_k$. When we examine the contents of the "row" loop, we see that there are several smaller loops each with $O(\frac{m}{b})$ iterations, and the procedure of finding the best gap insertion point takes time $O(\log m)$. The variable $b$ corresponds to the number of cores present in the ATGPU multiprocessor, is dictated by the architecture in use, and is typically much smaller than $m$. Therefore $O(\frac{m}{b}) \geq O(\log m)$, meaning the "row" loop interior is $O(\frac{m}{b})$.

Thus, a single multiprocessor executes in $O(kn\frac{m}{b})$ time.

(ii) We see that a multiprocessor accesses the entire pattern, meaning $\frac{m}{b}$ blocks are accessed. Further, for each

---

**Algorithm 1** AGPU Pseudocode for `GPUGapsMis`

---

1: **for all** $MP_\rho \in MP[0, ..., h-1]$ **do in parallel**
2:    **for all** $core_\epsilon \in CORE[0, ..., b-1]$ **do in parallel**
3:       Point $prevG$ to $G_{q-1}$ and $currG$ to $G_q$
4:       Initialise $G_0$ into $prevG$
5:       // Calculate $G_q$ for $q = 1 \rightarrow k$ gaps
6:       **for** $q = 1 \rightarrow k$ **do**
7:          // Initialise $G_q[0, *]$
8:          **if** $\epsilon == 0$ **then**
9:             $prevRow[0] \leftarrow 0$
10:          **for** $(j = \epsilon + 1; j \leq m; j += b)$ **do**
11:             $prevRow[j] \leftarrow OPEN + (j-1)EXT$
12:             // Place $prevRow$ into $currG$
13:             $currG[0, j] \Leftarrow prevRow[j]$
14:             // Initialise $maxILoc$ and $maxIVal$
15:             $maxIVal[j] \leftarrow prevRow[j]$
16:             $maxILoc[j] \leftarrow 0$
17:          // Calculate $G_q[i, *]$
18:          **for** $(i = 1; i \leq n + 1; i++)$ **do**
19:             $t \Leftarrow t[i]$ //Get Text Char
20:             **for** $(j = \epsilon + 1; j \leq m; j += b)$ **do**
21:                //Fetch $G_{q-1}[i, *]$
22:                $prevGCurrRow[j] \Leftarrow prevG[i, j]$
23:                //Update $maxILoc$ and $maxIVal$ if needed
24:                $p[\epsilon] \leftarrow (i - maxILoc[j] - 1) * EXT$
25:                **if** $maxIVal[j] + p[\epsilon] < prevGPrevRow[j]$ **then**
26:                   $maxIVal[j] \leftarrow prevGPrevRow[j]$
27:                   $maxILoc[j] \leftarrow i - 1$
28:                //Initialise $maxJLoc$ and $maxJVal$
29:                $maxJLoc[j] \leftarrow j$
30:                $maxJVal[j] \leftarrow prevGCurrRow[j]$
31:              Use Tree based method to calculate the Max values
32:              //Calculate the values to place into the cells
33:              Initialise cell $G_q[i, 0]$
34:              **if** $\epsilon == 0$ **then**
35:                 $currRow[0] \leftarrow ((i - 1) * EXT) + OPEN$
36:              **for** $(j = \epsilon + 1; j \leq m; j += b)$ **do**
37:                 Look in $prevRow[j - 1]$ to continue alignment
38:                 Look in $maxJVal$ for gap in Pattern, applying penalty
39:                 Look in $maxIVal$ gap in Text, applying penalty
40:                 Place max in $currRow[j]$
41:                 Calculate $hRow[j]$
42:              Copy $currRow$ to $currG[i, *]$, $hRow$ to $H[i, *]$
43:              Update $currRow$ and $prevRow$ pointers
44:              Update $prevGPrevRow$ and $prevGCurrRow$ pointers
45:          Update $currG$ and $prevG$ pointers
46:    **end parallel for**
47:  **end parallel for**
48: Report alignment score: $\max_{0 \leq \gamma \leq n} G_k[\gamma, m]$

---

individual row, we see that there are $4\frac{m}{b} + 1$ blocks of global memory accessed (for the text character, for fetching $prevGCurrRow$, for storing $currRow$ and for storing $hRow$). Therefore, we see that each multiprocessor accesses $kn4\frac{m}{b} + kn$ blocks of global memory. Across the entire algorithm aligning $qr$ sequence pairs, $qrkn4\frac{m}{b} + qrkn = O(qrkn\frac{m}{b})$ global memory blocks are accessed.

(iii) We see that for a multiprocessor aligning a sequence pair, the amount of global memory used is $2(n+1)(m+1)$ for the two $G$ matrices, plus $n$ ints for the text and $m$ ints for the pattern, therefore for $h$ multiprocessors aligning $h$ sequence pairs, the amount of global memory used is $O(hnm)$.

(iv) We see that for the shared memory data structures, no index over the value of $m$ is ever read or written in any multiprocessor, this makes the complexity of shared memory used to be $O(m)$. □

## 4 EXPERIMENTAL SETTING

Sequence alignment tools are typically used to search databases of known sequences, in order to find the best match for a query sequence, or set of query sequences.

**Multiple Pairwise Sequence Alignment.** In order to simulate a database search for the most optimal alignment for a set of query sequences, we align a set of query (pattern) sequences with a set of target (text) sequences.

Let $\mathcal{T} = t_1, t_2, ..., t_q$ be the set of text sequences, and $\mathcal{P} = p_1, p_2, ..., p_r$ be the set of pattern sequences. We want to simulate searching in a database for the text sequence which gives the best alignment score for each individual pattern sequence. Let $\mathcal{S} = s_1, s_2, ..., s_{qr}$ be the set of sequence pairs, that is $\mathcal{S} = \mathcal{T} \times \mathcal{P}$. For each $s_i \in \mathcal{S}$, we solve the Semiglobal Sequence alignment with a bounded number of gaps problem, with either `GPUGapsMis` or `CPUGapsMis` - a sequential implementation of GapsMis on a single CPU thread.

**Input Data.** The sequence data used is taken from the NCBI DNA sequence database *GenBank* [20]. From the database, we choose from a selection of genomic data, namely *e.coli* and *Ralstonia solancearum*. 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 performance of `GPUGapsMis` with real sequence data, as all data is treated identically by the algorithm.

For our 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.

The sequences are stored in text files containing one sequence per line. There are eight input files for text sequences; each file contains $16, 32, 64, ..., 2048$ sequences, and each text sequence is 250bp in length. There are four input files for pattern sequences; the length of pattern sequences in each file is $50, 100, 150, 200$ bp, and each pattern file contains $100$ pattern sequences. Each input set is formed by taking one text sequence file and one pattern sequence file.

**Approaches.** For evaluating the most effective way to use the GPU device as a co-processor for `GPUGapsMis`, we use several approaches detailed below, summarised in Table 1. We run control experiments with two versions of `CPUGapsMis`; `CPU-A` computes the alignment scores only, and `CPU-B` computes the alignment with backtracking.

There are in total six distinct approaches used in experiments with `GPUGapsMis`. The approaches for `GPUGapsMis` consist of a *batching method* and, where appropriate, a *backtracking method*. `GPU-A` computes the alignment scores only. Two approaches are considered for the batching method used when computing the alignment; *single text batching method* denoted by `-S`, and *multiple text batching method* denoted by `-M`. There are two approaches considered when we compute backtracking: `GPU-B` computes alignment with backtracking entirely on the GPU (we refer to this as the *GPU backtracking method*), and `GPU-H` computes the alignment scores on the GPU and computes backtracking on the CPU

|         | Batching      | Backtracking |
|---------|---------------|--------------|
| CPU-A   | -             | -            |
| CPU-B   | -             | On CPU       |
| GPU-S-A | Single Text   | -            |
| GPU-M-A | Multiple Text | -            |
| GPU-S-H | Single Text   | Hybrid       |
| GPU-M-H | Multiple Text | Hybrid       |
| GPU-S-B | Single Text   | GPU          |
| GPU-M-B | Multiple Text | GPU          |

TABLE 1: Summary of approaches.

(we refer to this as the *Hybrid backtracking method*).

**Single Text Batching Method.** In the single text batching method, single text sequence is sent to the GPU, along with all pattern sequences. It is then aligned with all pattern sequences, before the next text is sent to the GPU for alignment with all pattern sequences. More precisely, the text data for $t_i \in \mathcal{T}$ is sent to the GPU, along with all pattern data. The kernel is run, and any output data is returned to the host. This is repeated for subsequent text sequences, meaning sequence data requires $O(qrm)$ words transferred to the GPU, and $O(rm)$ space allocated on the GPU. Single text batching method is denoted by (s) against the algorithm name.

**Multiple Text Batching Method.** In the multiple text batching method, we send multiple text sequences, along with all pattern sequences to the GPU, then allocate space in the GPU memory for $\ell$ sequence pairs to be aligned. The sequence data requires $O(qn + rm)$ words transferring to the GPU and $O(qn + rm)$ space allocated on the GPU. The $qr$ alignment tasks required for aligning all sequence pairs in $\mathcal{S}$ are executed in $\lceil \frac{qr}{\ell} \rceil$ batches to ensure enough global memory is available to store the required matrices. The kernel is run for each batch, returning any output data to the host.

**GPU Backtracking Method.** In the GPU backtracking method, the backtracking algorithm GapsPos is performed on the GPU inside the same Kernel as the alignment scores calculation, by a single thread. The calculated data of size $O(qrg)$ is then returned to the host.

**Hybrid Backtracking Method.** In the hybrid backtracking method, the alignment score calculation is performed on the GPU. The backtracking $H$ matrices of size $O(qrnm)$ are returned to the host asynchronously at the end of the kernel execution for each thread block, and GapsPos is performed on the CPU.

**Verification of Correctness.** Testing was carried out, whereby output matrices were compared between the CPU and GPU in order to verify the correctness of the calculations. This verification was done using 16 text sequences of length 250bp and 100 pattern sequences of each available length.

**Performance Measurement.** To evaluate the performance, we compare three measurements. *Latency* is measured as the total time taken. *Throughput* is a measure of how fast the data matrices are filled and is measured in Mega Cell Updates per Second (MCUPS). Precisely throughput is calculated by dividing the total number of cells of $G$ and $H$ matrices to be updated in the entire execution, by the time taken to compute them. *Improvement ratio* is calculated as $\frac{CPU\,Latency}{GPU\,Latency}$, yet as this compares the performance of CPUGapsMis and GPUGapsMis, it could be calculated using throughput to obtain identical values. If this improvement ratio value is greater than 1, then GPUGapsMis has yielded an improvement against CPUGapsMis.

**Hardware.** We run the experiments on a custom built system of the following specification: AMD A10-5800K APU, NVIDIA NVIDIA GTX 680 GPU, 16 GB RAM, Ubuntu 16.04 OS, CUDA 8. The block size for the CUDA experiments was set at $max(32, m - (m \mod 32))$, with a maximum size of 256. This value was found empirically to give the best performance. The A10-5800K APU has 4 cores and a base clock rate of 3.8 GHz. The GTX 680 GPU has 8 Streaming Multiprocessors, 1536 CUDA cores and 2GB device memory, a base clock rate of 1006 MHz, and is of the nVidia Kepler architecture family. This is a similar hardware setting to that used by Ojiaku [24], yet Ojiaku used an Intel i7-3930k CPU with 6 cores and a clock rate of 3.2 GHz. As we only consider single thread execution on the CPU, the different number of cores of the two CPU does not make comparison unfair. The CPU used for our evaluation has a faster clock rate (3.8 GHz) than that used by Ojiaku (3.2 GHz), meaning the improvement ratio reported using the AMD APU may be lower than if the Intel CPU was used due to the CPU reference program having lower latency on the AMD APU. Also of note is that our machine runs Ubuntu 16.04, whereas the machine used by Ojiaku was running Windows 7. This is of importance because the proprietary nVidia drivers used on the Ubuntu system are different from those used in Windows, so performance could be affected. We also use an nVidia GTX650 GPU on the same system, in order to investigate how GPUGapsMis scales on different hardware.

## 5 RESULTS

In this section, we present and discuss results from experiments carried out as described in Section 4. Following from the AGPU analysis in Section 3.2, we expect that the latency of GPUGapsMis is lower than CPUGapsMis, that latency increases linearly as input size increases, and that the improvement ratio of GPUGapsMis against CPUGapsMis decreases as the pattern length increases, because the amount of shared memory used corresponds with the pattern length, thereby affecting the occupancy level on the GPU.

We look to evaluate the performance change of GPUGapsMis as the input size increases, and to validate the AGPU analysis given in Section 3.2. We carry out all experiments described in Section 4, with all results presented in the supplementary material. In order to look closely at the trends, we focus in this section discussion on two settings: (i) increasing number of sequence pairs with pattern length fixed at 200bps; and (ii) increasing pattern length with number of sequence pairs fixed at 204800. Both settings investigate the effect of increasing data size. The results presented here appear in tables (see the supplementary material) as either the final rows, or the bottom-right sub tables. These results are representative of all other experiment results obtained. We also compare the performance of GPUGapsMis against the algorithm presented in [24]

Each figure in this section is made up of three subfigures; (a) latency results; (b) calculated throughput; (c) calculated improvement ratio.

## 5.1 Single Text Batching Method Results

First, we investigate results achieved by `GPUGapsMis` using the single text batching approach. Sections 5.1.1 and 5.1.2 discuss computing alignment scores only, and computing alignment scores with backtracking, respectively. Section 5.1.3 gives a summary.

### 5.1.1 Alignment Scores Only

**Results.** Figures 5 and 6 show that the latency of `CPU-A` (black solid curve) and `GPU-S-A` (red dotted curve) increase linearly with the increase in size of input data. `GPU-S-A` has smaller latency than `CPU-A` in all cases and therefore outperforms `CPU-A` in all cases. The rate of increase in latency is 7.3 higher for `CPU-A` than for `GPU-S-A`.

This agrees with the AGPU analysis given in Section 3.2.

Figure 5 shows that the throughput of `CPU-A` stays constant while the improvement ratio and the throughput of `GPU-S-A` decrease as the pattern length increases. The throughput drops from 86.3 MCUPS at pattern length 50, to 74.5 MCUPS at pattern length 200, with improvement dropping from 8.4 to 7.3 times. Figure 6 shows that for increasing number of sequence pairs, the throughput (around 74 MCUPS) and the improvement ratio (around 7.6) of `GPU-S-A` remain stable.

**Discussion.** We see that the throughput and improvement ratio of `GPU-S-A` relative to `CPU-A` is sensitive to increasing pattern length, yet not sensitive to increasing number of sequence pairs to align. These performance metrics are less stable for increase in pattern length because shared memory use increases with pattern length, lowering the occupancy rate. This means less warps are available for hiding the latency of global memory access operations. In turn, input sets will take longer to process as the number of sequence pair alignment tasks concurrently run on the SM is decreased.

**Comparison against existing work.** The blue dotted curve in Figures 5 and 6 show the performance of the algorithm proposed in [24], `GPU-O`. We see that for some smaller pattern lengths, there is no improvement achieved, however as the pattern length is increased, we see that the performance level of `GPU-O` drops. `GPU-S-A` is less sensitive to increase in pattern length and for pattern lengths 150 or greater, `GPU-S-A` out performs `GPU-O`. Figure 5a shows the trend of `GPU-S-A` latency is the less steep of all. At its peak, `GPU-S-A` achieves throughput 23MCUPS higher than `GPU-O`, and a greater speedup of 7.59 against 5.29 of `GPU-O`.

To further confirm the trend of improvement of `GPU-S-A` against `GPU-O`, we give comparison of the approaches aligning 204800 longer sequence pairs, where the text length is fixed at 500bps, and the pattern length is between 50 and 450. The results in Figure 7 show that the trend of `GPU-S-A` outperforming `GPU-O` for pattern lengths of 150 or greater continues when we align longer sequences.

### 5.1.2 Alignment Scores with Backtracking

**Results.** Figures 8 and 9 show that when backtracking is also calculated, similar trends occur.

When we compare `GPU-B` and `GPU-H`, we see that the GPU backtracking approach (`GPU-B`) always outperforms the hybrid backtracking approach (`GPU-H`). In more details, Figure 8 shows when the pattern length increases, `GPU-H` achieves an improvement ratio of about 3.1 times while `GPU-B` achieves 7.0-7.8 times. With increasing number of sequence pairs, Figure 9 shows the improvement ratios of `GPU-H` and `GPU-B` are 3.1 times and 7.2 times, respectively.

**Discussion.** We note that when backtracking is included, the throughput achieved is higher; see `GPU-S-A` vs `GPU-S-B` in Figures 6b and 9b and `CPU-A` vs `CPU-B` in Figures 5b and 8b. This is because the additional requirement to populate the $H$ matrices require less work per cell than when populating the $G$ matrices. Each row of the $G$ matrices requires $O(\log m)$ computation by the multiprocessor, yet only $O(1)$ additional computation is required to calculate the values for each row of the $H$ matrices.

The improvement ratio achieved by `GPU-S-B` was slightly lower than `GPU-S-A`, as shown in Figure 10. The backtracking algorithm GapsPos is a serial computation which has not been parallelised, and is not efficient on the GPU. Therefore it is faster on the CPU than on the GPU, giving rise to the lower improvement ratio exhibited by `GPU-S-B` compared to `GPU-S-A`.

Figures 8 and 9 show that `GPU-S-H` achieved lower throughput than all other `GPUGapsMis` approaches, and exhibit lower sensitivity to increasing pattern length. The reason for this is the higher amount of data transfer between the CPU and the GPU. The cost associated with data transfer between CPU and GPU is very high, and can create a bottleneck in a GPU program.

### 5.1.3 Summary

In summary, taking into account of all experimental results presented in the supplementary material, `GPU-S-A` is on average 7.7 times faster than `CPU-A`. The peak improvement ratio is 8.4 times, when the pattern length is 50 and number of sequence pairs is 204800. Note that the throughput achieved in this setting is 86.4 MCUPS. On the other hand, when backtracking is considered, the peak throughput is increased to 121 MCUPS, though the improvement ratio is 7.8 times which is lower than the 8.4 times without backtracking. This peak occurs at the same input setting as above. This higher throughput but lower improvement ratio is due to less work required to calculate the additional cells during the backtracking phase, and the sequential backtracking algorithm being inefficient on the GPU.

On average, over all experiment settings we see that the throughput increases by 33.4 MCUPS when backtracking is considered, compared to the alignment scores only counterpart. The improvement ratio of `GPU-S-B` decreases by 0.4 on average, when compared to `GPU-S-A`. The improvement ratio of `GPU-S-H` decreases by 4.3 on average, when compared to `GPU-S-B`.

## 5.2 Multiple Text Batching Results

We now investigate results achieved by `GPUGapsMis` using the multiple text batching approach.

### 5.2.1 Alignment Scores Only

As shown in Figures 11 and 12, there are similar trends in latency, throughput and improvement ratio exhibited
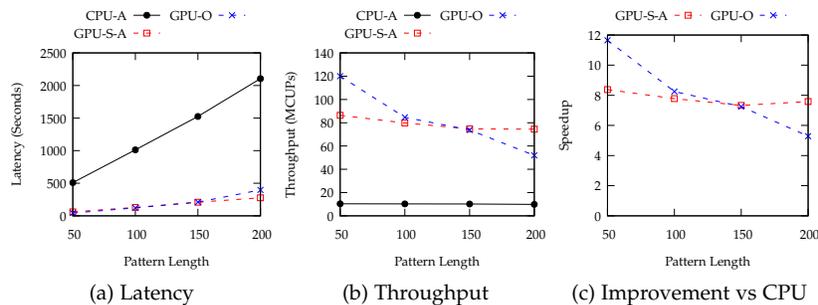
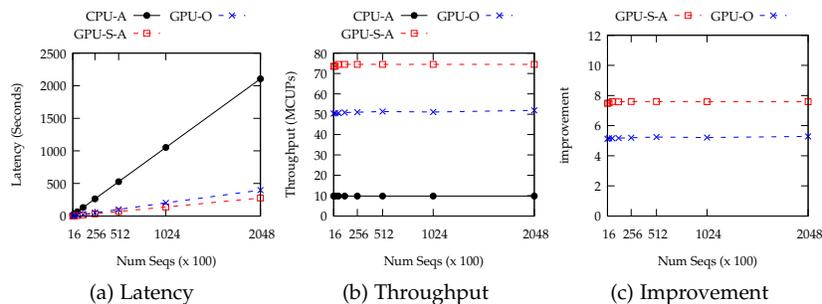Fig. 5: Result for `GPU-S-A` and `GPU-O`, for input sets containing 204800 sequence pairs.



Fig. 6: Result for `GPU-S-A` and `GPU-O`, for input sets containing patterns of length 200.
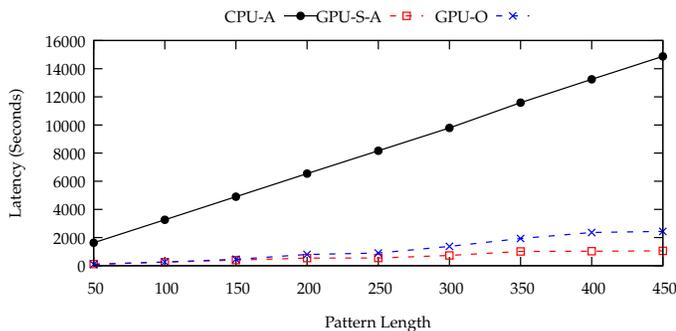


Fig. 7: Latency of `GPU-S-A` and `GPU-O`, for input sets containing 204800 sequence pairs with texts of length 500.

by `GPU-M-A` to those exhibited by `GPU-S-A` discussed in Section 5.1.1.

By examining Figures 11b and 12b closer, we see that `GPU-M-A` achieves greater throughput than `GPU-S-A`. This is because `GPU-M-A` requires less host device communication than `GPU-S-A`. In Section 5.1.3 `GPU-S-H` was negatively affected by increased host device data transfer and therefore exhibited lower sensitivity to increasing pattern length with fixed number of sequence pairs, being shown as a flatter and lower trend in throughput and improvement ratio when compared to `GPU-S-B`. This is a similar scenario, as `GPU-S-A` has a greater host device data transfer requirement than `GPU-M-A`. This is amplified by the lower number of host device synchronisations required by `GPU-M-A` compared to `GPU-S-A`.

### 5.2.2 Alignment Scores with Backtracking

We see in Figures 13 and 14 that `GPU-M-B` and `GPU-M-H` exhibit trends similar to their respective single text batching counterparts, `GPU-S-B` and `GPU-S-H`.

Similar to Section 5.2.1, the multi text batching `GPU-M-B` and `GPU-M-H` perform consistently better than the single text counterpart `GPU-S-B` and `GPU-S-H`, respectively. This is because each of the multi text approaches require less host device communication and data transfer than their single text counterpart. As previously explained, the data transfer between host and device is very expensive and can be detrimental to the performance, therefore reducing the amount of this type of data transfer as much as possible would benefit the improvement ratio against the CPU, as has been demonstrated here.

An interesting result is the throughput and improvement ratio of `GPU-M-H`, which monotonically increases as pattern length is increased, as shown in Figures 13b and 13c. This is the only GPU approach to exhibit such a characteristic. `GPU-M` can schedule at most $qr$ threadblocks on the GPU in a single batch, whereas `GPU-S` is more limited and can only schedule up to $r$ threadblocks in a single batch. Therefore when $H$ matrices are returned asynchronously to the host upon termination of the kernel, there are more threadblocks ready for execution in `GPU-M-H` than `GPU-S-H`, meaning `GPU-S-H` is not able to hide the latency of asynchronous data transfer as effectively as `GPU-M-H`.

### 5.2.3 Summary

In summary, taking into account of all experimental results presented in the supplementary material, we see that the peak performance of `GPU-M-A` and `GPU-M-B` occur in the
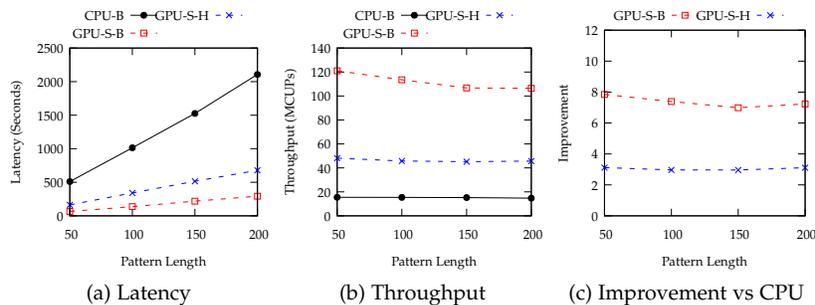
(a) Latency      (b) Throughput      (c) Improvement vs CPU

Fig. 8: Result for `GPU-S-B` and `GPU-S-H`, for input sets containing 204800 sequence pairs.



(a) Latency      (b) Throughput      (c) Improvement vs CPU
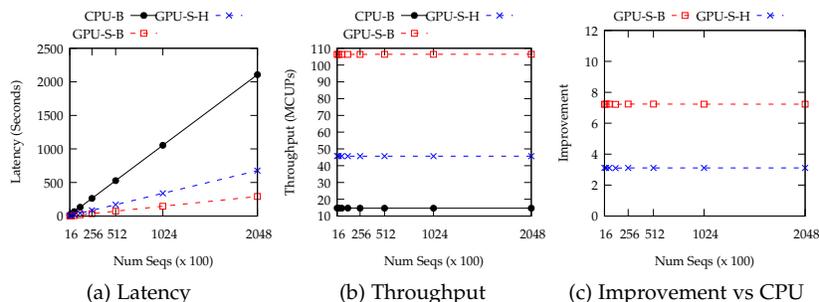
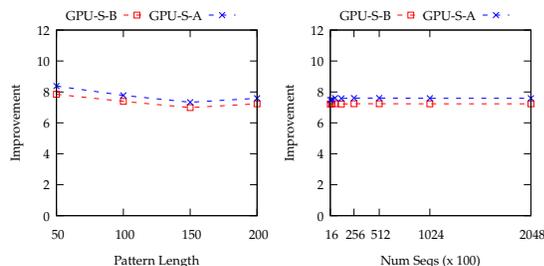Fig. 9: Result for `GPU-S-B` and `GPU-S-H`, for input sets containing patterns of length 200.



Fig. 10: Comparison of improvement ratio between `GPU-S-A` and `GPU-S-B`.

same setting; when pattern length is 50, for 204800 sequence pairs. `GPU-M-A` is on average 10.1 times faster than `CPU-A` and increases the improvement ratio on average by 2.3 compared to `GPU-S-A`. The peak improvement ratio is 11 times, when the pattern length is 50 and number of sequence pairs is 204800. Note that the throughput achieved in this setting is 113.2 MCUPS. On the other hand, when backtracking is computed, the peak throughput is increased to 161 MCUPS, though the improvement ratio is 10.4 times which is lower than the 11 times without backtracking. As with single text batching, this higher throughput but lower improvement ratio is due to less work required to calculate the additional cells for backtracking, and the sequential backtracking algorithm being inefficient on the GPU.

The improvement ratio of `GPU-M-H` decreases by 6.1 on average, when compared to `GPU-M-B`. On average, `GPU-M-H` causes an increase in improvement ratio by 0.6 and an increase in throughput by 9.8 MCUPS when compared to

`GPU-S-H`.

We see that `GPU-M-B` increases throughput yet lowers the improvement ratio achieved, when compared to `GPU-M-A`. Throughput of `GPU-M-B` increases on average by 45.8 MCUPS compared to `GPU-M-A`, and the improvement ratio decreases by 0.3 on average. `GPU-M-H` achieved higher throughput and higher improvement ratio than `GPU-S-H`, yet does not outperform `GPU-B`.

### 5.3 Improvement on Different GPU Devices

By running `GPUGapsMis` on GPUs with more resources, it is expected that a higher level of improvement against `CPUGapsMis` would be achieved, however some parallel algorithms are not able to take advantage of extra resources past a certain point, due to excessive communication overhead. We wish to investigate whether a GPU with more resources is negatively affected in performance gained, when compared to a lower specification GPU, due to finite global memory access bandwidth and costly access latency. The increased number of alignment tasks (threadblocks) running concurrently on the GPU could create a communication bottleneck when serving global memory requests.

We test this by investigating how results of `GPUGapsMis` on GTX680 (already discussed) compare to results on GTX650. GTX650 and GTX680 has 2 and 8 SMs, clock speed of 1.2GHz and 1 GHz, and global memory of 1GB and 2GB, respectively. GTX680 has more Streaming Multiprocessors than GTX650, so it can run more alignment tasks concurrently than GTX650. Therefore we expect GTX680 to outperform GTX650 when running `GPUGapsMis`. Assuming that all data fits on the GPU memory, we must decide how much we expect GTX680 to outperform GTX650. GTX680 has 4 times the resources
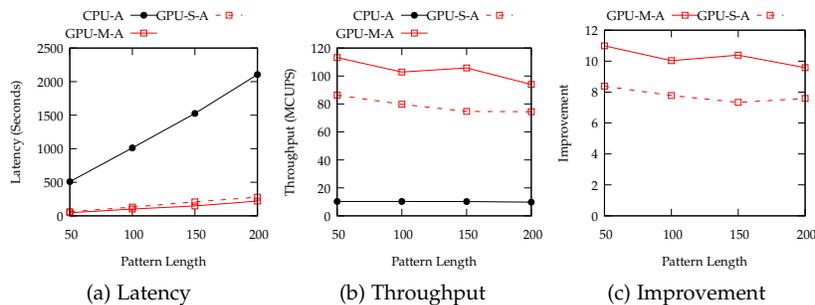
Fig. 11: Result for `GPU-M-A`, performing alignment scores phase only with multiple text batching, for input sets containing 204800 sequence pairs.
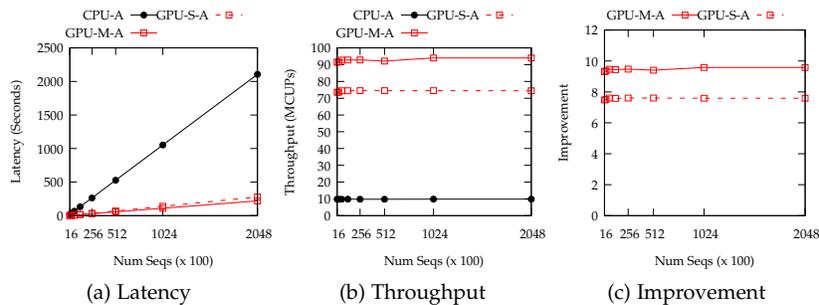


Fig. 12: Result for `GPU-M-A`, performing alignment scores phase only, with multiple text batching, for input sets containing patterns of length 200.
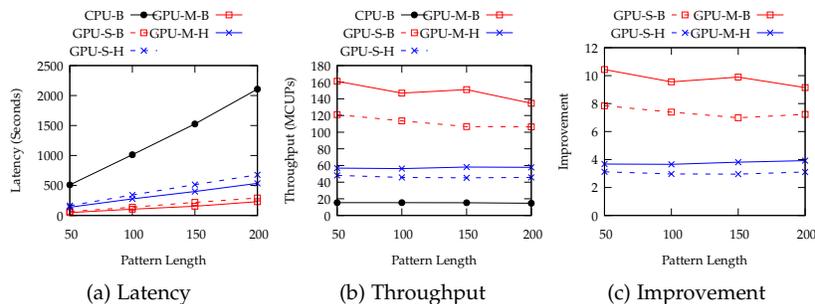


Fig. 13: Result for `GPU-M-B` and `GPU-M-H`, with multiple text batching, for input sets containing 204800 sequence pairs.

of GTX650, but a clock speed that is only 83% of GTX650. Therefore we can estimate that GTX680 will be around 3.3 times faster than GTX650. The global memory bandwidth of GTX680 is only 2.4 times of GTX650, so there is potential for some applications to encounter a bottleneck in global memory access on GTX680, yet not GTX650.

We run the best performing approach of `GPUGapsMis`, `GPU-M-B` on GTX650. If `GPUGapsMis` has 3.3 or greater improvement on GTX680, compared to GTX650, then we should expect that running `GPUGapsMis` on a Kepler GPU with specifications higher than GTX680 would yield greater improvement still. The results obtained achieved are summarised in Table 2.

GTX680 outperforms GTX650 in all cases, by a ratio of 3.5 times. This ratio remains constant throughout increase in pattern length and throughout increase in number of

| GPU | GTX650 | GTX680 |
|---|---|---|
| Num SM | 2 | 8 |
| Clock Speed | 1.2GHz | 1GHz |
| Resource Ratio | 1 | 4 |
| Expected Improvement | 1 | 3.3 |
| Observed Improvement | 1 | 3.5 |

TABLE 2: Summary of GTX650 and GTX680 comparative resources and comparative performance of `GPU-M-B`.

sequence pairs. Figures 15 and 16 demonstrate that the performance of `GPU-M-B` exhibits similar trends on GTX650 as on GTX680, and show the ratio of improvement between the two GPUs unaffected by input data size.

We are able to conclude that `GPUGapsMis` adapts to a GPU of different specification well, and that any communication overhead is not exaggerated by a disproportionate
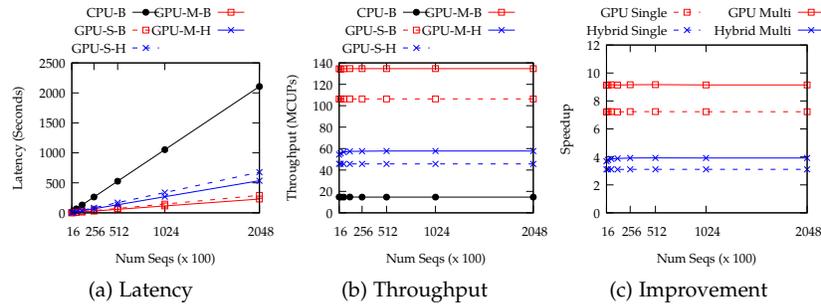
Fig. 14: Result for `GPU-M-B` and `GPU-M-H`, with multiple text batching, for input sets containing patterns of length 200.
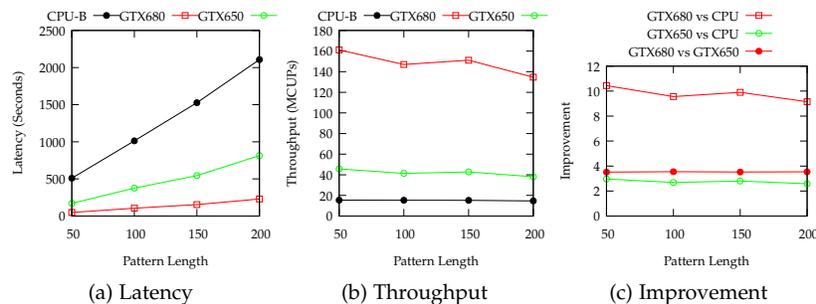


Fig. 15: Result for `GPU-M-B`, with multiple text batching, running on GTX 650 for input sets containing 204800 sequence pairs.
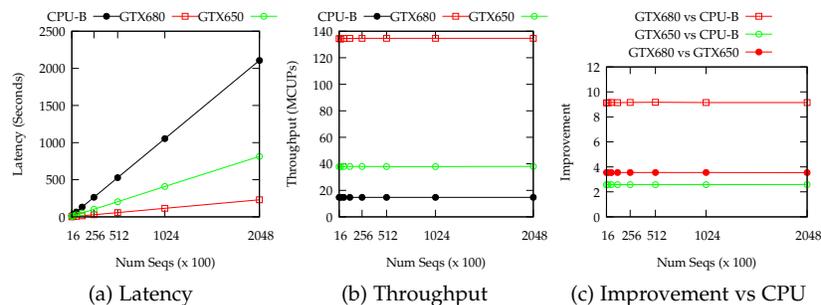


Fig. 16: Result for `GPU-M-B`, with multiple text batching running on GTX 650 for input sets containing patterns of length 200.

amount, as resources available are increased. Therefore, we are able to have confidence that proportionally better speedup would be possible, should higher specification GPUs be used to run `GPUGapsMis`.

## 6 CONCLUSION

We present a study on a GPU-based algorithm to solve the pairwise semi-global sequence alignment with bounded number of gaps problem, using a data-parallel approach. We analyse our algorithm `GPUGapsMis` on the AGPU model, with theoretical analysis confirmed by observed results. We achieve greater speedup compared to a previous data-parallel approach.

- We achieve peak speedup against the CPU of 11 times when only alignment scores are computed, and

10.4 times when backtracking is also computed. We achieve greater speedup compared to a previous data-parallel approach [24].

- We show that the best performance is achieved by `GPU-M-B`, with multi text batching and backtracking computed on the GPU. Of all approaches considered, `GPU-M-B` requires the least host device communication.

- We show that the performance scales well on a GPU of better specification.

In the future, it would be interesting investigate different data-parallel approaches to lower the amount of shared memory required, as well as investigate task parallel methods. In addition to this, it would also be interesting to look at ways to improve the performance of the backtracking phase, possibly by using a task-parallel GPU kernel. We use only a

single GPU device in this paper, so it would be interesting to investigate using multiple GPU devices to test further scalability, as well as to use higher specification GPUs to verify the improved speedup claim. Our results show that the amount of data transfer required can have a tangible effect on the performance of the algorithm, yet this is not captured in the analysis given by the AGPU model. Recently, the authors propose the Abstract Transferring GPU (ATGPU) [6], an improved abstract GPU model including data transfer, so it would be particularly interesting to analyse different approaches of `GPUGapsMis` using the ATGPU. Furthermore, it would be interesting to consider GPU variants for other alignment problems, e.g. those that may replace BWA or Bowtie.

## References

[1] Nikolaos Alachiotis, Simon Berger, Tomáš Flouri, Solon P Pissis, and Alexandros Stamatakis. libgapmis: extending short-read alignments. *BMC Bioinformatics*, 14(Suppl 11):S4, 2013.

[2] Stephen F Altschul, Warren Gish, Webb Miller, Eugene W Myers, and David J Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215(3):403–410, 1990.

[3] Carl Barton, Tomáš Flouri, Costas S Iliopoulos, and Solon P Pissis. Global and local sequence alignment with a bounded number of gaps. *Theoretical Computer Science*, 582:1–16, 2015.

[4] Alhadi Bustamam, Kevin Burrage, and Nicholas 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*, 9(3):679–692, 2012.

[5] Thomas C Carroll, Jude-Thaddeus Ojiaku, and Prudence W H Wong. Pairwise Sequence Alignment with Gaps with GPU. In *Proceedings of the IEEE International Conference on Cluster Computing (Workshop on Heterogeneous and Unconventional Cluster Architectures and Applications)*, 2015.

[6] Thomas C Carroll and Prudence W H Wong. An Improved Abstract GPU Model with Data Transfer. In *Proceedings of the International Conference on Parallel Processing Workshops*, 2017.

[7] Reza Farivar, Harshit Kharbanda, Shivaram Venkataraman, and Roy H. Campbell. An algorithm for fast edit distance computation on GPUs. *2012 Innovative Parallel Computing*, pages 0–8, 2012.

[8] Kimon Frousios, Costas S. Iliopoulos, Laurent Mouchard, Solon P. Pissis, and German Tischler. REAL: an efficient REad ALigner for next generation sequencing reads. *Proceedings of the First ACM International Conference on Bioinformatics and Comptuational Biology*, pages 154–159, 2010.

[9] Kai J Kohlhoff, Marc H Sosnick, William T Hsu, Vijay S Pande, and Russ B Altman. CAMPAIGN: an open-source library of GPU-accelerated data clustering algorithms. *Bioinformatics*, 27(16):2321–2322, 2011.

[10] Atsushi Koike and Kunihiko Sadakane. A Novel Computational Model for GPUs with Application to IO Optimal Sorting Algorithms. In *2014 IEEE International Parallel and Distributed Processing Symposium Workshhops*, pages 614–623, 2014.

[11] Ben Langmead, Cole Trapnell, Mihai Pop, and Steven Salzberg. Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. *Genome biology*, 10(3):R25, 2009.

[12] Heng Li and Richard Durbin. Fast and accurate short read alignment with Burrows-Wheeler transform. *Bioinformatics*, 25(14):1754–1760, 2009.

[13] Ruiqiang Li, Yingrui Li, Karsten Kristiansen, and Jun Wang. SOAP: short oligonucleotide alignment program. *Bioinformatics*, 24(5):713–714, 2008.

[14] Ruiqiang Li, Chang Yu, Yingrui Li, Tak Wah Lam, Siu Ming Yiu, Karsten Kristiansen, and Jun Wang. SOAP2: An improved ultrafast tool for short read alignment. *Bioinformatics*, 25(15):1966–1967, 2009.

[15] Lukasz Ligowski, Witold Rudnicki, Łukasz Ligowski, and Witold Rudnicki. An efficient implementation of Smith Waterman algorithm on GPU using CUDA, for massively parallel scanning of sequence databases. In *IEEE International Symposium on Parallel & Distributed Processing*, pages 1–8. IEEE, 2009.

[16] Chi Man Liu, Thomas Wong, Edward Wu, Ruibang Luo, Siu Ming Yiu, Yingrui Li, Bingqiang Wang, Chang Yu, Xiaowen Chu, Kaiyong Zhao, Ruiqiang Li, and Tak Wah Lam. SOAP3: ultra-fast GPU-based parallel alignment tool for short reads. *Bioinformatics*, 28(6):878–879, 2012.

[17] Yongchao Liu, Adrianto Wirawan, and Bertil Schmidt. CUDASW++3.0 accelerating Smith-Waterman protein database search by coupling CPU and GPU SIMD instructions. *BMC Bioinformatics*, 14(117), 2013.

[18] Gerton Lunter and Martin Goodson. Stampy: A statistical algorithm for sensitive and fast mapping of Illumina sequence reads. *Genome Research*, 21(6):936–939, 2011.

[19] Svetlin A Manavski and Giorgio Valle. CUDA compatible GPU cards as efficient hardware accelerators for Smith-Waterman sequence alignment. *BMC Bioinformatics*, 9(Suppl 2):S10, 2008.

[20] NCBI. NCBI Genbank. https://www.ncbi.nlm.nih.gov/genbank/.

[21] S B Needleman and C D Wunsch. A general method applicabe to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48:443–453, 1970.

[22] NVidia. Kepler Architecture. http://international.download.nvidia.com/pdf/kepler/NVIDIA-Kepler-GK110-GK210-Architecture-Whitepaper.pdf.

[23] nVidia. Programming Guide:: CUDA Toolkit Documentation. http://docs.nvidia.com/cuda/cuda-c-programming-guide/, aug 2017.

[24] Jude-Thaddeus Ojiaku. *A Study of Time and Energy Efficient Algorithms for Parallel and Heterogeneous Computing*. PhD thesis, University of Liverpool, Dec 2014.

[25] Sooraj Puthoor, Ashwin M Aji, Shuai Che, Mayank Daga, Wei Wu, Bradford M Beckmann, and Gregory Rodgers. Implementing directed acyclic graphs with the heterogeneous system architecture. In *Proceedings of the 9th Annual Workshop on General Purpose Processing using Graphics Processing Unit*, pages 53–62. ACM, 2016.

[26] Peter Rice, Ian Longden, and Alan Bleasby. EMBOSS: The European Molecular Biology Open Software Suite. *Trends in Genetics*, 16(1):276–277, 2000.

[27] Stephen M. Rumble, Phil Lacroute, Adrian V. Dalca, Marc Fiume, Arend Sidow, and Michael Brudno. SHRiMP: Accurate mapping of short color-space reads. *PLoS Computational Biology*, 5(5):1–11, 2009.

[28] Korbinian Schneeberger, Jörg Hagmann, Stephan Ossowski, Norman Warthmann, Sandra Gesing, Oliver Kohlbacher, and Detlef Weigel. Simultaneous alignment of short reads against multiple genomes. *Genome Biology*, 10(9):R98, 2009.

[29] Mohammed A Shehab, Abdullah A Ghadawi, Luay Alawneh, Mahmoud Al-Ayyoub, and Yaser Jararweh. A hybrid cpu-gpu implementation to accelerate multiple pairwise protein sequence alignment. In *8th International Conference on Information and Communication Systems*, pages 12–17. IEEE, 2017.

[30] T F Smith and M S Waterman. Identification of Common Molecular Subsequences. *Journal of Molecular Biology*, 147:195–197, 1981.

[31] Gregory M Striemer and Ali Akoglu. Sequence alignment with GPU: Performance and design challenges. In *IEEE International Symposium on Parallel & Distributed Processing*, pages 1–10. IEEE, 2009.

[32] Panagiotis D Vouzis and Nikolaos V Sahinidis. GPU-BLAST: using graphics processors to accelerate protein sequence alignment. *Bioinformatics*, 27(2):182–188, 2011.

[33] Ling Sing Yung, Can Yang, Xiang Wan, and Weichuan Yu. GBOOST: a GPU-based tool for detecting gene–gene interactions in genome–wide case control studies. *Bioinformatics*, 27(9):1309–1310, 2011.

[34] Kaiyong Zhao and Xiaowen Chu. G-BLASTN: accelerating nucleotide alignment by graphics processors. *Bioinformatics*, 30(10 2014):1384–1391, 2014.

**Thomas C. Carroll** received his BSc degree in Computer Science from University of Liverpool, UK in 2014. He is currently studying towards his PhD, also at the University of Liverpool. His research interests are on parallel programming in particular in GPU optimisation, and in abstract modelling of GPU.

**Jude-Thaddeus Ojiaku** is a Design Engineer at ASML, The Netherlands. He received his PhD in Computer Science from University of Liverpool, UK in 2016. His research interests are on parallel programming in particular GPU optimisation. He has been involved in the following projects: Intrafield Corrections for Dedicated Chuck Overlay, Introduction of NXT:1980Di machine, Reticle Shape Correction at Reticle Stage, Cross-Matching Compensation and calibration test software owner for machine focus calibration.

**Prudence W.H. Wong** is a professor in the Department of Computer Science, University of Liverpool, UK. She received her PhD from The University of Hong Kong in 2003. Her research interests are on design and analysis of algorithms, combinatorial optimisation with applications to other discipline including computational biology. She is on the Editorial Board of Information Processing Letters, The Computer Journal, Algorithms.