TÉCNICO LISBOA

Accelerating approximate string matching in heterogeneous computing platforms

João Pedro Silva Rodrigues

Dissertação para obtenção do Grau de Mestre em

Engenharia Electrotécnica e de Computadores

Orientadores: Doutor Pedro Filipe Zeferino Tomás
Doutor Nuno Filipe Valentim Roma

Júri

Presidente: Doutor Nuno Cavaco Gomes Horta
Orientador: Doutor Pedro Filipe Zeferino Tomás
Vogal: Doutor Luís Manuel Silveira Russo

Novembro de 2016

Para a Mariana

Acknowledgments

I would like to start by giving my sincerest thank you my supervisors, Dr. Nuno Roma and Dr. Pedro Tomás, for their guidance, insights and the corrections given throughout the development of the work presented in the thesis. I would also like to thank them deeply for the patience they have shown me.

I would also like to extend my appreciation to INESC-ID, and particularly the staff, for providing the space and tools necessary to the development of the experimental work in this thesis.

I must also thank my family for all the support and patience I have received in the course of this work. In particular, to my father, who provided the best conditions for work, and to my mother, who kept pushing me to overcome the difficulties I had encountered. Without their support, this work could not exist.

I would also like to thank my colleagues and friends for all the help and for the discussions that helped me gain a new insight over DNA and bioinformatics.

And lastly, I would like to thank Rita Nogueira for the comprehension and inspiration.

Resumo

A redução dos custos da sequenciação de genomas provocou um crescimento na quantidade de informação genética sequenciada. Para criar um novo genoma a partir destes dados, é preciso alinhar cada leitura a um genoma padrão, permitindo alguns erros entre leitura e genoma. Esta operação de procura inexacta tem custos computacionais muito elevados. Estes custos podem ser reduzidos através de, por um lado, usar procura exacta para efectuar a procura aproximada apenas pequenas partes do genoma-referência (métodos heurísticos), e, por outro, usar plataformas de computação paralelas para realizar a procura exacta e o alinhamento aproximado. O trabalho aqui descrito descreve a concepção e implementação de uma nova ferramenta de procura inexacta para plataformas compostas por múltiplos aceleradores heterogéneos, desenvolvido a partir da ferramenta de procura exacta BowMapCL v1.0. A ferramenta existente foi aumentada através da integração de um mecanismo de filtração, com o objectivo de gerar regiões para a procura aproximada. A procura aproximada, nomeadamente o algoritmo de Smith-Waterman, foi também implementada em GPU. Contrariamente às soluções existentes, a ferramenta aqui descrita (BowMapCL v2.0) foi implementada com recurso ao OpenCL, que permite utilizar diversos aceleradores diferentes, nomeadamente CPUs e GPUs da AMD e NVIDIA. Os resultados indicam que o BowMapCL oferece uma redução do tempo de alinhamento de 3 vezes em relação a ferramentas de alinhamento semelhantes que usam apenas o CPU, e que têm uma performance competitiva em ferramentas baseadas em GPU para sequências inferior a 100 caracteres.

Palavras-chave: Procura aproximada de palavras, Algoritmo de Smith-Waterman, Programação Dinâmica, Computação Paralela, OpenCL, Unidade de Processamento Gráfico

Abstract

The reduction of the genome sequecing costs generated an exponential increase in the quantity of sequenced genetic information. The creation of a new genome from this information involves the aligment of each read to a reference genome, while allowing some mismatches between read and reference. This operation of approximate string match has high computational costs, which can be lowered by creating areas of the reference where a match is more likely through the use of exact search, and by using parallel heterogeneous platform to accelerate the exact search and the approximate matching. The work herein presented describes the conception and implementation of a new tool of approximate string search for heterogeneous platforms, developed from the exact string match tool BowMapCL v1.0. The existing tool was augmented by integrating into it a new filtering procedure to generate results suitable for optimal search. The optimal search algorithm, namely the Smith-Waterman algorithm, was ported to the GPU. Unlike existing solutions, the tool herein described (BowMapCLv 2.0) was implemented through OpenCL, which allows the usage of GPUs from AMD and NVIDIA. The results indicate that BowMapCL offers a reduction of the aligment time of 3 times in relation to similar alignment tools using only CPU, and offers a competitive performance when compared to GPU-based tools for queries with a length inferior to 100.

Keywords: Approximate string search, Smith-Waterman algorithm, Dynamic programming, Parallel computation, OpenCL, Graphics Processing Unit

Contents

List of Tables

List of Figures

Acronyms

BWT Burrows-Wheeler transform.

CPU Central Processing Unit.

DP dynamic programming.

FIFO First In First Out (queue).

FM-index Full-text minute space index.

FPGA Field Programmable Gate Array.

GPGPU General Purpose computation on Graphics Processing Unit.

GPU Graphics Processing Unit.

NGS Next Generation Sequencing.

NW Needleman-Wunsch algorithm.

SW Smith-Waterman algorithm.

1

Introduction

1.1 Definition of the problem

Bioinformatics is an active research area which studies biological data using computer-based methods. An important research area within bioinformatics is the study of genetic information, namely DNA and proteins. As it will been seen in the next chapter, genetic information is encoded through a sequence of characters, similar to regular text. In DNA these characters represent bases, while for proteins they represent aminoacides.

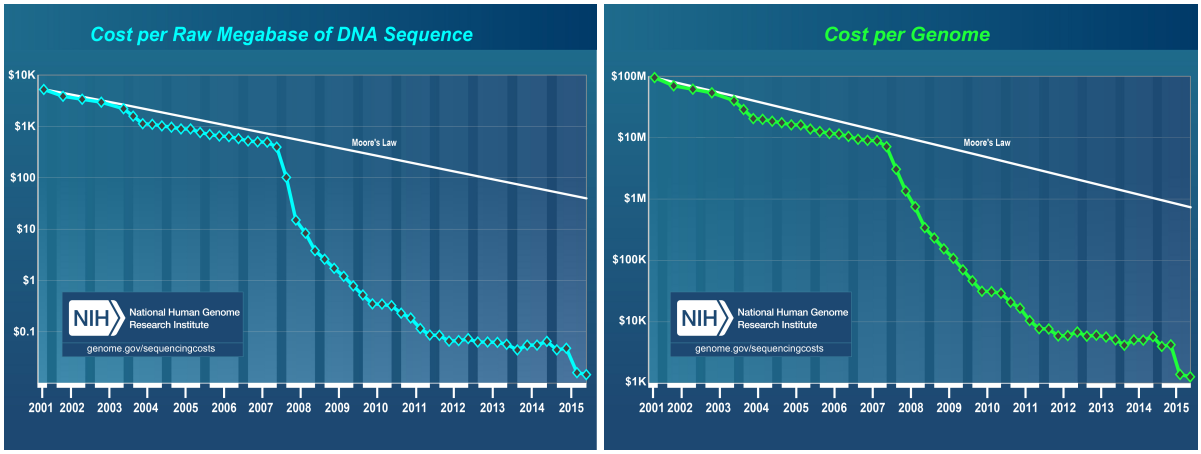
The study of the genome is often performed through the comparative analysis between two (or more) sequences of data, which can be fragments or a complete genome. The discovery of similar regions between the sequences of data is usually performed using sequence alignment tools. The study of the variations of genetic information between the different elements of a population requires the construction of a complete genome for each studied element. As we will see ahead, the most efficient way to reconstruct a genome is with recourse to sequence alignment tools. The complete genomes can then be used to, for instance, study the genes responsible for different diseases [6, 17] or to study how humans spread across the globe [59].

Sequence alignment tools discover the best matching regions, even if they do not match perfectly and contain differences, known as errors, between them. Hence, sequence alignment is an example of the approximate string matching problem.

Approximate string matching performs string matching of a pattern in a text, while allowing errors to occur between the pattern and the text, with the goal of discovering the position of the pattern in the text. Approximate string matching can also discover what are the errors present between the pattern and the text. Usually the size of the text is much greater than the size of the pattern. Approximate string matching is used in several different areas, The areas of application of approximate string matching include signal processing, retrieval of information/text, pattern recognition and data mining [44].

1.2 Motivation

The emergence of Next Generation Sequencing (NGS) has brought a reduction in price per read base, and consequently the reduction of price of sequencing an entire human genome [66], as can be seen in Figure 1.1. NGS, also known as High Throughput Sequencing, breaks the nucleotide chains into shorter fragments than traditional technologies, with lengths ranging from 30 to 600 bases (with emphasis on the lower bound). These short reads are then sequenced in parallel. Considering the short length of fragments, there is a need to increase the coverage depth (the number of reads a single base of the original genome is present) to up to 30 times to ensure the entire reconstitution of the genome. Since the human genome has over 3×10^9 base pairs, hundred of millions of short reads can be generated by a single run of a sequencing machine. The lower cost of sequencing has resulted in an exponential growth in the total amount of raw sequenced DNA, according to Cochrane et al. [11].



(a) Cost per Megabase of DNA sequence (b) Cost per Human genome

Figure 1.1: Decline of sequencing costs of DNA [66]

The size of the human reference genome and the increasing quantities of raw data generated by NGS tools require efficient tools to process them. For instance, in 2012 the Project 1000 Genomes used a compute cluster with 1192 processors to align all the collected data [1].

In order to decrease the execution time of the approximate string matching algorithm it is possible to reduce the approximate string search to specific regions of the original text through the use of an exact string matching procedure [45]. These heuristic approximate string matching algorithms have, however, possibility of missing the optimal positions of an approximate match.

Even with the heuristic techniques, it is very time consuming to align the data (and therefore to reconstruct a genome). To cope with the increasing demands, other computing platforms coupled with CPUs, such as GPGPUs, Field Programmable Gate Arrays (FPGAs) or special-purpose processors have been evaluated [32, 58]. However, implementing an algorithm efficiently in these computing platforms is often more resource consuming due to their specialised nature.

Recently, modern Graphics Processing Unit (GPU) architectures became capable, in addition to their traditional role of graphics processing, of executing general purpose code that would traditionally be executed in a Central Processing Unit (CPU), hence the moniker General Purpose computation on Graphics Processing Unit (GPGPU), and to do so with lower development costs compared to other non conventional computing platforms. Modern GPUs are composed of hundreds data processing lanes, giving them the capability to perform a huge number of computations in parallel. Furthermore, the total bandwidth available to them is also much greater than what is available to modern CPUs. Therefore, GPUs have been used to accelerate many applications requiring more computational power, including approximate string matching. Examples include CUDASW++ [35], CUSHAW2-GPU [37], SOAP3-dp [39] and OCLSW [51], which indeed have lower execution times than competing CPU applications.

The majority of existing approximate string matching applications using GPUs adopt the CUDA programming framework, restricting those applications to run only on Nvidia GPUs. Another widely used heterogeneous programming framework is OpenCL, which can execute in GPUs and other heterogeneous devices, such as FPGAs, accelerators, and even general purpose CPUs. In particular, OpenCL is capable to execute the same code on NVIDIA and AMD GPUs (and CPUs), ensuring a tool that is capable of running every GPU, regardless of the specific machine. Thus, with OpenCL it is possible to build an efficient sequence alignment tool which takes advantage of GPUs without being restricted to a specific GPU maker. In fact, since it is also possible to run OpenCL on the CPU, the execution of the tool is not restricted to the presence of a GPU.

This thesis utilizes as its base the exact search implementation built by Nogueira [47], which is aligned with the objectives of the present work. His tool is a highly parallel implementation of an offline exact string search algorithm targeting GPUGPUs using the OpenCL API. Moreover, special care was taken to ensure that the used host memory and device memory can be adapted, enabling the execution in highly different computing platforms, independently of the size of the input data. It is also scalable in regards with the number of devices and can utilize different heterogeneous devices without loss of efficiency by using a work division scheme.

1.3 Objectives

Using as the starting point the work developed by Nogueira [47], this master thesis aims to build an optimized approximate string matching application. The main objectives for the proposed tool are the following:

- It will target different processing platforms, from mobile GPUs to GPUs from Nvidia and AMD to accelerators such as Intel Xeon Phi and CPUs, through the usage of the OpenCL API, in contrast

to existing solutions that target CUDA-enabled GPUs only. Despite this, the tool will be optimised for GPU.

- Like its predecessor, the tool should also scale the throughput with an increase in the number of computing devices. Existing tools, such as SOAP3-dp or CUSHAW2-GPU, can only be executed on a single GPU. Moreover, the tool should also adopt a dynamic load balancing scheme to ensure that, despite the variation in the processing capability of the accelerators, the execution time is not adversely affected.
- The tool should be agnostic in the data it is capable of processing. Existing tools are built only taking into consideration the area of bioinformatics and are hard-coded to process DNA or protein sequences only, depending on the domain area. This tool should keep the ability of its predecessor to use user-defined data types, including general text, as long as the lexicon can be coded in 8-bit.
- Maintain its predecessor index partitioning schemes to be able to adjust the memory usage to the restrictions of host memory and targeted accelerator memory. Existing state of the art tools have very high requirements in terms of host and GPU memory, with SOAP3-dp, for instance, requiring 16 GB of host memory and 3 GB of GPU memory for alignments against the human genome.

1.4 Main contributions

In the course of this master thesis a new approximate string matching application was developed, focusing on alignment of DNA but capable of operating on different types of data, including text. A new heterogeneous multi-device parallelisation model was devised to allow an efficient execution on platforms composed of multiple CPUs and GPUs. The implementation of this model using the OpenCL framework was accomplished through the implementation of the gaped Smith-Waterman algorithm capable of executing on diverse accelerating devices, such as GPUs from NVIDIA and AMD. The experimental results show that the proposed implementation can offer a speedup of 3 times compared to bowtie2, a CPU-based DNA aligner, and is is faster than SOAP3-dp, a GPU-based DNA aligner, for reads with lengths inferior to 100 bases.

1.5 Thesis outline

This document is divided into 6 chapters, divided in the following manner:

- **Chapter 2 - State of the art of sequence alignment in GPUs**: This chapter present existing approximate string matching algorithms, with an emphasis on parallel implementations of algorithms currently used in bioinformatics. The chapter ends with a summary of the state of the art tools in sequence alignment in parallel architectures.
- **Chapter 3 - Proposed parallel architecture**: In this chapter a description of the envisioned tool is presented, namely, what are the components the tool is composed of, how is parallelism extracted

from the components and how are the multiple devices managed to ensure a well performing scalable tool.

- **Chapter 4 - Implementation details**: This chapter provides an overview of the adopted filtration algorithm. It also details the manner in which the SW algorithm was mapped to the GPUs, and how the memory was arranged to provide an efficient implementation.
- **Chapter 5 - Experimental evaluation**: In this chapter the experimental results from the proposed tool are presented. It includes comparisons against some state of the art tools and the evaluation of certain parameters of the proposed tool.
- **Chapter 6 - Conclusions**: This chapter presents the accomplished results from this work and the possible directions to take in future work.

2

State of the art of Sequence Alignment in GPUs

This chapter presents a small overview of the evolution of sequence alignment algorithms and tools used in the area of bioinformatics, and serves as the algorithmic foundation for the proposed tool.

The chapter starts by introducing the data operated by sequence alignment tools, DNA and proteins. After some common operations using sequence alignment are presented, some classes of optimal sequence alignment algorithms are presented in Section 2.1.

Section 2.2 presents the evolution the two principal types of optimal alignment algorithms, starting from the introduction of dynamic programming until the advent of the gaped Smith-Waterman (SW). In this section it is also presented the evolution of the various techniques utilised to parallelise the SW algorithm, and their respective application in GPUs. Some techniques which lower the memory consumption of SW are also presented.

The section 2.3 has an overview of the non optimal alignment algorithms, which decrease the execution time through heuristic methods. Some state of the art tools, CPU and GPU-based, implementing non optimal alignment algorithms are also presented.

2.1 Introduction to sequence alignment

As stated in section 1.1, sequence alignment is performed on DNA or proteins. DNA, or deoxyribonucleic acid, is a complex organic molecule with two complementary polymers (also known as strands), each composed of a chain of nucleotides. Each of these nucleotides has a nucleobase (or just base), and there are 4 different bases: cytosine, guanine, adenine and thymine, also known as C, G, A and T, respectively. Each of these bases has a complementary base in the other strand, which is, respectively, G, C, T and A. RNA, or ribonucleic acid, shares with DNA the fact that is also a complex organic molecule, but is composed by a single polymer. RNA lacks the base thymine, which is substituted by uracil, or U. DNA and RNA can be translated to aminoacides, to constitute proteins. Each of the approximately 16 existing aminoacides is also represented by a character. Hence, both kinds of genetic information are represented as sequences of characters, which can be viewed as a text, with the information encoded in the sequence in which its bases/aminoacides are arranged.

As previously stated, in order to perform a comparative analysis, it is helpful to perform a sequence alignment to align two or more sequences to find where is their best match. Another common operation in bioinformatics is the creation of a complete genome from a cell. The extraction of the complete genome is performed by DNA sequencing. It works by breaking the long DNA chains into shorter DNA fragments, called reads, that are sequenced through biological methods, such as NGS. These short reads can be arranged to assemble the original sequence. This can be performed by *de novo* assembly, where the reads are assembled into a full genome without any prior knowledge of the genome, by finding overlapping reads. The assembly can also be accomplished, when there is an available reference genome, through a faster method called genome based assembly. In this case, the reads are aligned against the reference genome to find their final positions in the genome [49] using sequence alignment. These positions are then used to assemble the new genome.

Sequence alignment can also be defined as an operation performed between two or more sequences to determine the degree of similarity between the sequences. The best matching regions can also include errors between the pattern and the text. The errors present in sequence alignment include insertion, where a character is present in the pattern but not in the text, see Fig. 2.1a; deletion, if a character present in the text but not in the pattern, see Fig. 2.1b; mutation/substitution of a character between the pattern and the text, see Fig. 2.1c; and, finally, gaps, where several characters are inserted or deleted sequentially between the pattern and the text, shown in Fig. 2.1d. Every match or substitution has an associated substitution score $\delta(x_i, y_j)$, where x_i and y_j are the characters from the pattern and text, respectively, while an individual insertion or deletion has a deletion cost, α. Thus, every alignment has an associated score, and the optimal alignment is the aligment that maximises this score. In the case of nucleotides and aminoacides sequences the sequence alignment score is referred to as sequence homology. Finally, sequence alignment can also be used to extract the aforementioned individual operations needed to change one sequence into the other.

Sequence alignment is a particular case of approximate string matching, where the objective is to find a pattern in a text where one (or both) of them have suffered some degree of corruption, as stated by

```
... C G A C T — A T G ...
    | | | | |   | | |
    C G A C T T A T G
```

(a) Insertion of a base in the pattern

```
... C G A C T T A T G ...
    | |   | | | | | |
    C G — C T T A T G
```

(b) Deletion of a base in the pattern

```
... C G A T T T A T G ...
    | | |   | | | | |
    C G A C T T A T G
```

(c) Substitution of a base between pattern and text

```
... C G — — — T A T G ...
    | |         | | |
    C G A C T T A T G
```

(d) Gap between pattern and text

Figure 2.1: Example of all possible errors that occur in sequence alignment

Navarro [44]. Approximate string matching algorithms can be performed with online algorithms [35, 56] or offline algorithms [23, 42, 64]. Offline algorithms require pre-processing of the text to create data structures which may enable faster execution per processed pattern. The overhead of pre-processing the text means they can only be applied to static text. On the other hand, online algorithms are more suited to changing texts. Furthermore, approximate string matching algorithms, and by extension sequence alignment, are divided into optimal and non-optimal algorithms. Optimal algorithms guarantee that the best matching region(s) are found, unlike non-optimal algorithms. However, non-optimal algorithms are often characterized by smaller execution times.

Bioinformatics applications require the cost $\delta(x_i, y_j)$ to be dependent on the specific characters x_i and y_j, while in regular text the cost can be simply dependent if the characters are different or equal. For the latter case, more efficient approximate string matching algorithms exist. For instance, Ukkonen's cut off heuristic [63] has a time complexity of $\mathcal{O}(kn)$ and a space complexity of $\mathcal{O}(m)$ for k errors, where n and m are the lengths of the pattern and text, respectively. Hence, approximate string matching in bioinformatic tools must be computed using the more flexible dynamic programming (DP) algorithms, with a time complexity of $\mathcal{O}(mn)$. Another difference is that in general texts the score is usually reported in terms of number of edits, and the objective of approximate string matching is to minimize this number, while in biology the score reports the similarity, which must be maximized. These approaches are dual to each other, since reducing the number of edits increases the similarity score, resulting in the same optimal region, but with very different scores. Moreover, in biological sequences it is more important to consider gaps to have a lower cost than the equivalent serial deletion.

If there are only two sequences to be aligned, the alignment is called pairwise alignment, otherwise it is called multi-sequence alignment (MSA), which often utilizes for an initial step pairwise alignment. MSA is often utilised to discover evolutionary relations between different genomes. The present work will focus exclusively in pairwise alignment, since it has a lower time complexity and has more applications than MSA. Pairwise alignment can be classified into global alignment and local alignment. In the next section we will give a closer look to existing algorithms.

2.2 Optimal alignment algorithms

This section will present an overview of the evolution of available optimal string alignment algorithms commonly used in bioinformatics applications, starting with global alignment algorithms and progressing

to local alignment algorithms. This section also introduces the different parallelisation techniques available for an important local alignment algorithm, namely the Smith-Waterman, and that can be used to extract parallelism from the GPUs, as we will see ahead. The section concludes with the algorithms and tools implementing the traceback step in linear memory.

2.2.1 Global alignment

In global alignment the best alignment found must take into account the entire length of both sequences and is therefore more suited for sequences of similar length and with a high degree of homology between themselves. One of the potential uses is tracing the evolution of a specific protein through several species.

2.2.1.A Needleman-Wunsch algorithm

Needleman and Wunsch [46] introduced in 1970 dynamic programming (DP) to determine the optimal global alignment of two sequences. The original Needleman-Wunsch (NW) algorithm, presented in (2.1) only distinguished between matches and mismatches, allowing the creation of gaps without any penalty. The computation of this algorithm has a time complexity of $\mathcal{O}(m^2n + mn^2)$ and a space complexity of $\mathcal{O}(mn)$, where m and n are the lengths of the respective sequences.

The proposed algorithm has two phases: in the first phase a score matrix H is computed whereas in the second phase there is a traceback procedure where the optimal alignment is identified. The score matrix H is composed of $m \times n$ cell values $H_{i,j}$. The traceback starts in the last row or column of the matrix, in the cell with the maximum value, and proceeds to the next highest cell from the preceding row and/or column.

Hence, for two sequences $S_Q = q_1, q_2, \ldots, q_m$ and $S_D = d_1, d_2, \ldots, d_n$ of lengths m and n, respectively, $H_{i,j}$ represents the global alignment score of the prefixes S_Q and S_D ending at q_i and d_j, respectively. The elements of the matrix $H_{i,j}$ are computed, for $1 \leq i \leq m$ and $1 \leq j \leq n$ through the recurrence equation:

$$\begin{aligned} H_{i,j} &= \max \begin{cases} \max\limits_{k \geq 1}\{H_{i-k,j-1}\} + \delta(q_i, d_j), \\ \max\limits_{l \geq 1}\{H_{i-1,j-l}\} + \delta(q_i, d_j) \end{cases} \\ H_{i,0} &= 0 \\ H_{0,j} &= 0 \end{aligned} \tag{2.1}$$

where substitution score $\delta(q_i, d_j)$ is the cost of replacing character q_i with d_j. Matching characters are associated with positive scores, while mismatching characters are associated with negative scores.

Insertion and deletions occur when $\delta(q_i, d_j)$ is a match but the maximum value of $H_{i-k,j-1}$ or $H_{i-1,j-k}$ is not in diagonal $H_{i-1,j-1}$, while matches and substitutions occur when the maximum score is in the diagonal $H_{i-1,j-1}$. In an insertion, a new character is introduced into the reference before a new

match between query and reference, while in a deletion a character from the reference is deleted before a new match. If several insertions or deletions occur together, there is a gap. This algorithm does not penalise insertions, deletions or gaps, which have a cost 0.

Sankoff [56] improved the algorithm of Needleman-Wunsch, by maintaining the use of DP, but by reducing the time complexity of the algorithm to $\mathcal{O}(mn)$ while maintaining the space complexity. A modified version of this version is presented in (2.2). The Needleman-Wunsch algorithm predominantly refers to the version introduced by Sankoff.

Given two sequences $S_Q = q_1, q_2, \ldots, q_m$ and $S_D = d_1, d_2, \ldots, d_n$ of lengths m and n, respectively, $H_{i,j}$ represents the global alignment score of the prefixes S_Q and S_D which end at q_i and d_j, respectively. The elements of the matrix $H_{i,j}$ are computed, for $1 \leq i \leq m$ and $1 \leq j \leq n$, using a modified version of the algorithm adding the cost of a gap $\theta(k)$ of length k where the cost of the gap is linear with the size $k\theta(k) = k\alpha$, through the recurrence equation:

$$
\begin{aligned}
H_{i,j} &= \max \begin{cases} H_{i-1,j-1} + \delta(q_i, d_j), \\ H_{i-1,j} - \theta(k), \\ H_{i,j-1} - \theta(k) \end{cases} \qquad \Rightarrow H_{i,j} = \max \begin{cases} H_{i-1,j-1} + \delta(q_i, d_j), \\ H_{i-1,j} - \alpha, \\ H_{i,j-1} - \alpha \end{cases} \\
H_{i,0} &= -\alpha \times i \\
H_{0,j} &= -\alpha \times j
\end{aligned}
\tag{2.2}
$$

where α is the cost of a gap of size 1, and $\delta(q_i, d_j)$ is the cost of replacing character q_i with d_j. The initial cost $H_{i,0} = -\alpha \times i$ and $H_{0,j} = -\alpha \times j$ can also be change to the $H_{i,0} = 0$ and $H_{0,j} = 0$, as it is presented in the original version in equation 2.1. The former version penalizes alignments that do not start at the beginning of the sequences, while the latter allows the alignment to start where the homology score is maximised. In the latter case, the alignment created is called a semi-global alignment.

After the matrix is built, the score from the cell of the last column and row $H_{m,n}$ indicates the homology score. In alternative, it is also possible to select the homology score as the maximum value from the last column $H_{m,j}$ or the from last row $H_{i,n}$. This last option is also a part of the semi-global alignment.

Furthermore, it is possible to traceback through the matrix, starting from cell which indicates the homology score. For the example of a scoring matrix H in Table 2.1, the traceback starts in the cell $(10, 8)$, colored in gray in the table. Thence, we advance to the cell from the preceding column ($H_{i-1,j}$), row ($H_{i,j-1}$) or diagonal ($H_{i-1,j-1}$) which originated the value the current cell. In the example, the cell $(9, 8)$ gives the current cell a score of $9 - 3 = 6$, the cell $(9, 7)$ gives the current cell a score of $12 + (-2) = 10$ and the cell $(10, 7)$ generates a score of $16 - 3 = 13$, respectively. Since the value of the current cell is 13, the preceding cell is $(10, 7)$, also colored gray. Moreover, since the preceding cell is in the same column, the last "movement" is an insertion, as shown in Figure 2.2. Likewise, the 3 possible cells to precede the now-current cell are $(9, 7)$, $(9, 6)$ and $(10, 6)$, which generate scores of respectively, $12 - 3 = 9$, $11 + 5 = 16$ and $8 - 3 = 5$. Therefore, the preceding cell is the one from the preceding diagonal, as marked in Table 2.1, and corresponds to a match, signalized by a vertical bar in Fig. 2.2.

The procedure is iteratively repeated until the cell $H_{0,0}$ is reached, or, in case of a semi-global alignment, until a cell from the first row or column is reached.

$$\begin{array}{lcccccccccccc} S_Q: & \mathrm{T} & \mathrm{G} & \mathrm{G} & \mathrm{A} & \mathrm{C} & \mathrm{T} & — & \mathrm{A} & \mathrm{T} & \mathrm{G} & \mathrm{A} & — \\ & & | & & & | & | & & | & | & & | & \\ S_D: & \mathrm{C} & \mathrm{G} & — & — & \mathrm{C} & \mathrm{T} & \mathrm{T} & \mathrm{A} & \mathrm{T} & — & \mathrm{A} & \mathrm{T} \end{array}$$

Figure 2.2: Example of optimal global alignment, with gap cost $\alpha = 3$ and $\delta(q_i, d_j) = 5$ if $q_i = d_j$ and $\delta(q_i, d_j) = -2$ if $q_i \neq d_j$.

		0	1	2	3	4	5	6	7	8	9	10
			T	G	G	A	C	T	A	T	G	A
0		0	-3	-6	-9	-12	-15	-18	-21	-24	-27	-30
1	C	-3	-2	-5	-8	-11	-7	-10	-13	-16	-19	-22
2	G	-6	-5	3	0	-3	-6	-9	-12	-15	-11	-14
2	C	-9	-8	0	1	-2	2	-1	-4	-7	-10	-13
3	T	-12	-4	-3	-2	-1	-1	7	4	1	-4	-7
4	T	-15	-7	-6	-5	-4	-3	4	5	9	6	3
5	A	-18	-10	-9	-8	0	-6	1	9	6	7	11
6	T	-21	-13	-12	-11	-3	-2	-1	6	14	11	8
7	A	-24	-16	-15	-14	-6	-5	-4	4	11	12	16
8	T	-27	-19	-18	-17	-9	-8	0	1	9	9	13

Table 2.1: Score matrix of global alignment with linear gap model and traceback colored

2.2.2 Local alignment

Local alignment searches the pair of best matching regions of each sequence and is consequently more suited to searches where the sequences have very dissimilar lengths, where the homology of the sequences is unknown or is potentially low, or if the sequences are similar only in a relatively short region of the entire sequence. The most widely known local alignment algorithm is the Smith-Waterman, which is presented next.

2.2.2.A Smith-Waterman algorithm

The Needleman-Wunsch-Sankoff algorithm was modified by Smith and Waterman [60] to perform the optimal local alignment of two sequences by clamping negative scores of the alignment matrix to 0.

Hence, for two sequences $S_Q = q_1, q_2, \ldots, q_m$ and $S_D = d_1, d_2, \ldots, d_n$ of lengths m and n, respectively, $H_{i,j}$ represents the local alignment score of the prefixes S_Q and S_D which end at q_i and d_j, respectively. The elements of the matrix $H_{i,j}$ are computed, for $1 \leq i \leq m$ and $1 \leq j \leq n$ through the recurrence equation:

$$
\begin{aligned}
H_{i,j} &= \max \begin{cases} H_{i-1,j-1} + \delta(q_i, d_j), \\ H_{i-1,j}, \\ H_{i,j-1}, \\ 0 \end{cases} \\
H_{i,0} &= 0 \\
H_{0,j} &= 0
\end{aligned}
\tag{2.3}
$$

After the matrix H is completely filled, as in the other algorithms, there is a traceback procedure to determine the optimal alignment. The traceback begins at the cell with the highest score, and proceeds to the cell it originated from, recursively, until the start of one of the sequences or a cell with a value of 0 is reached. This new algorithm maintained a time complexity $\mathcal{O}(mn)$.

Also proposed in Smith and Waterman [60] is a linear gap model, where the cost of the gap is defined as $\theta(k) = k\alpha$, where k is the length of the gap and α is the cost of increasing the gap by one base. When the linear gap model is applied to (2.3), it results in the following recurrence equation:

$$
\begin{aligned}
H_{i,j} &= \max \begin{cases} H_{i-1,j-1} + \delta(q_i, d_j), \\ H_{i-1,j} - \theta(k), \\ H_{i,j-1} - \theta(k), \\ 0 \end{cases} \quad \Rightarrow H_{i,j} = \max \begin{cases} H_{i-1,j-1} + \delta(q_i, d_j), \\ H_{i-1,j} - \alpha, \\ H_{i,j-1} - \alpha, \\ 0 \end{cases} \\
H_{i,0} &= 0 \\
H_{0,j} &= 0
\end{aligned}
\tag{2.4}
$$

The proposed linear model maintains the computational complexity of the algorithm and improves the realism of the homology score, since the existence of gaps in the alignment can now be penalised. Moreover, since the gap penalty can be set lower than the equivalent serial deletion of bases, it is possible to model the existence of gaps independently from the deletion of bases.

2.2.2.B Gaped Smith-Waterman algorithm

The application of the sequence alignment to the alignment of biological sequences required a modeling of the gaps in the algorithm without incurring in heavy penalties in runtime, since in biology the changes between sequences occur more frequently in blocks of added bases, rather than smaller and more frequent changes spread throughout the sequences. Although there had been several proposals to solve the problem [56, 60], they all suffered from an increase in computational time in comparison to the original SW algorithm. In [15] Gotoh introduced a new affine gap model where the gap cost is defined as $\theta(k) = \alpha + (k-1)\beta$, where α is the initial cost of the gap, β is cost to extend the gap and k

is the length of the gap. The advantage of this new model is that it manages to maintain both the space complexity and computational complexity of the original algorithm.

Hence, under the newly introduced gap model, the elements of the matrix $H_{i,j}$ are computed, for $1 \leq i \leq m$ and $1 \leq j \leq n$, using the recurrence equation:

$$
\begin{aligned}
H_{i,j} &= \max \begin{cases} H_{i-1,j-1} + \delta(q_i, d_j), \\ E_{i,j}, \\ F_{i,j}, \\ 0 \end{cases} \\
E_{i,j} &= \max \begin{cases} H_{i,j-1} - \alpha, \\ E_{i,j-1} - \beta \end{cases} \\
F_{i,j} &= \max \begin{cases} H_{i-1,j} - \alpha, \\ F_{i-1,j} - \beta \end{cases} \\
H_{i,0} &= E_{i,0} = F_{i,0} = 0 \\
H_{0,j} &= E_{0,j} = E_{0,j} = 0
\end{aligned}
\tag{2.5}
$$

The values of the cells matrices $E_{i,j}$ and $F_{i,j}$ store the current value of a gap of insertions and deletions, respectively, that can either start or continue at the position (i, j). After the matrix is built, the highest value of the matrix, $H_{i,j}$ is selected, giving the homology score. Furthermore, it is also possible to travel back through the matrix, from $H_{i,j}$ to $H_{i-1,j}$, $H_{i,j-1}$ or $H_{i-1,j-1}$ to check if the there was an insertion, a deletion or a substitution/match from S_D to S_Q, respectively.

An example of a scoring matrix with the cells participating in the traceback marked is presented in Table 2.2, where $\alpha = 5$ and $\beta = 2$, the substitution cost $\delta(q_i, d_j) = 5$ for $q_i = d_j$ and $\delta(q_i, d_j) = -4$ for $q_i \neq d_j$. An example of the auxiliar table, F, is presented in Table 2.3.

		0	1	2	3	4	5	6	7	8	9	10	11
			T	G	C	G	T	A	C	T	G	T	G
0		0	0	0	0	0	0	0	0	0	0	0	0
1	C	0	0	0	5	0	0	0	5	0	0	0	0
2	G	0	0	5	0	10	5	3	0	1	5	0	5
2	C	0	0	0	10	5	6	1	8	0	0	1	0
3	T	0	5	0	5	6	10	5	3	13	8	6	4
5	A	0	0	1	3	1	5	15	10	8	6	4	2
6	T	0	5	0	1	0	6	10	11	15	10	11	6
7	A	0	0	1	0	0	1	11	6	10	11	6	4
8	T	0	5	0	0	0	5	6	7	11	6	16	11

Table 2.2: Score matrix H of local alignment with affine gap model and traceback

The respective local alignment of the sequence pairs is shown in the Figure 2.3:

		0	1	2	3	4	5	6	7	8	9	10	11
			T	G	C	G	T	A	C	T	G	T	G
0		0	0	0	0	0	0	0	0	0	0	0	0
1	C	0	0	0	0	0	0	0	0	0	0	0	0
2	G	0	0	0	0	0	0	0	0	0	0	0	0
2	C	0	0	0	0	5	0	0	0	0	0	0	0
3	T	0	0	0	5	3	1	0	3	0	0	0	0
5	A	0	0	0	3	1	5	0	1	8	3	1	0
6	T	0	0	0	1	0	3	10	5	6	1	0	0
7	A	0	0	0	0	0	1	8	6	10	4	6	1
8	T	0	0	0	0	0	0	6	4	8	6	4	0

Table 2.3: Score matrix F of local alignment with affine gap model and traceback

$$
\begin{array}{lcccccccccc}
S_Q: & \ldots & \mathrm{G} & \mathrm{C} & \mathrm{G} & \mathrm{T} & \mathrm{A} & \mathrm{C} & \mathrm{T} & \mathrm{G} & \mathrm{T} \\
 & & | & | & & | & | & & | & \ddagger & | \\
S_D: & \ldots & \mathrm{G} & \mathrm{C} & — & \mathrm{T} & \mathrm{A} & — & \mathrm{T} & \mathrm{A} & \mathrm{T}
\end{array}
$$

Figure 2.3: Example of optimal local alignment

2.2.3 Parallelism of SW

The search for homology of a given sequence in a reference database is performed by computing the alignment score between a given sequence and each of the sequences present in a reference database (see section 2.1). The growth in genetic information resulted in a necessity of greater velocity in homology search, achieved with the emergence of computers with parallel architectures (where the same operation is applied to different data), such as SIMD vectors or GPUs, leading to the parallelisation of the SW algorithm.

Since one given sequence typically has the alignment score computed against several reference sequences simultaneously, or several sequences are aligned against a single reference, there are two major approaches to extract parallelism: compute the alignment between one query and several references in parallel, known as intertask parallelism, or extract parallelism within the alignment between one query and one reference, referred to as intratask parallelism. Intertask parallelism is an embarassingly parallel problem, since there are no dependencies between the different alignments.

Intratask parallelism is hampered by the dependencies, since each cell $H_{i,j}$ depends simultaneously on the cells $H_{i-1,j}$, $H_{i,j-1}$ and $H_{i-1,j-1}$. For the gaped SW algorithm, henceforth known simply as SW, the cell $H_{i,j}$ depends simultaneously on the cells $H_{i-1,j-1}$, $E_{i,j}$ and $F_{i,j}$. In turn $E_{i,j}$ depends on $H_{i,j-1}$, $E_{i,j-1}$ and $F_{i,j}$ depends on $H_{i-1,j}$, $F_{i-1,j}$, as shown in Fig. 2.4.

However, since each cell depends solely on cells from the two preceding anti-diagonals, the cells in each anti-diagonal can be computed in parallel, leading to a wavefront processing approach, shown in Figure 2.6a. Wozniak [67] combines this approach with shifting each successive row one element to the right more than the preceding row to make memory accesses more regular. Figure 2.5 shows the execution of a wavefront of size 4, with the dependencies to $H_{i-1,j-1}$ colored in black, while dependencies to the values $H_{i,j-1}$, $E_{i,j-1}$ from previous column are colored in blue and the dependencies to the values of the previous row $H_{i-1,j}$, $F_{i-1,j}$ are colored in red. The values $H_{i-1,j}$ and $F_{i-1,j}$ from the first row of the wavefront must be fetched from the memory. After the each wavefront is calculated, $H_{i-1,j}$

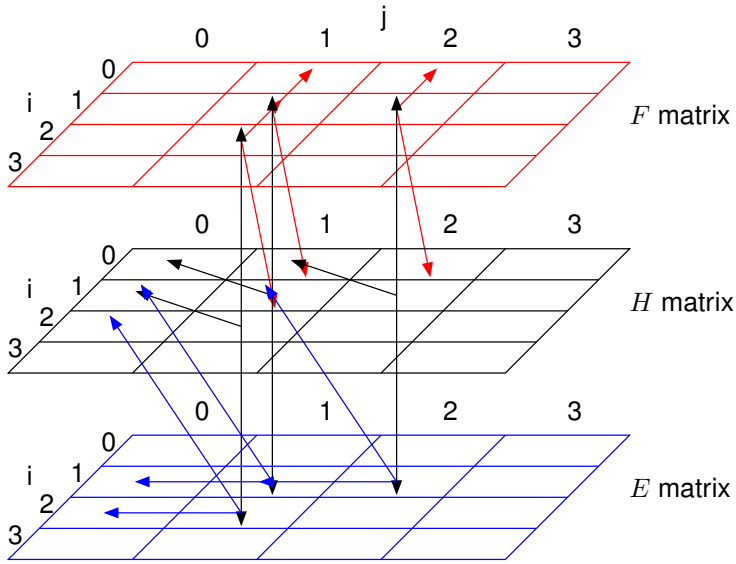


Figure 2.4: Dependencies of the score matrices

and $F_{i-1,j}$ are stored to the memory.

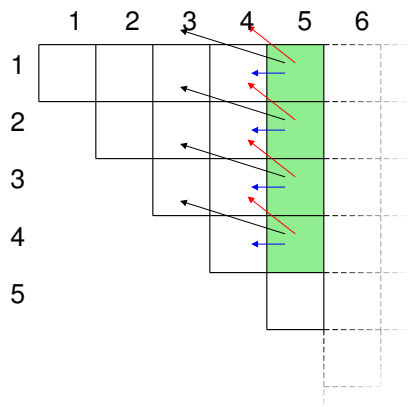


Figure 2.5: Shifted wavefront approach

After the values from the current wavefront are calculated, the algorithm calculates wavefront to the right of the current one, until reaching the end of the query. Afterwards, following 4 rows are calculated in a similar fashion. The reference is padded, guaranteeing the number of rows is multiple of 4.

His implementation (available in algorithm 2.1), achieved a speedup of 2.1 times over scalar code, making use of a 4-wide vector. Moreover, since this implementation only calculates the maximum homology score and is unable to do the traceback step, it is not necessary to store the entire matrix. Hence, the memory requirements are proportional to the size of the smallest reference.

Rognes and Seeberg [52] takes advantage of the fact that most cells from E and F have a negative value and consequently do not contribute to the respective values of H. In fact, cells from E and F are only positive and can contribute to H if the value $H_{i,j}$ is greater than the gap open penalty α, otherwise the cells from the current row of E or current column of F continue negative and do not contribute to values of H and can ignored, simplifying the computation. If, however, this assumption is not true, then it is necessary to re-compute the values of H, E and F.

This new algorithm, presented in algorithm 2.2, operates on several cells from H simultaneously,

Algorithm 2.1 Wozniak parallelisation of SW algorithm

```
 1: procedure SW_WOZNIAK
Require: MemE[i], MemH[i] are initialised to 0
 2:     vScore := 0
 3:     for i := 0, 4, . . . , length(reference)+2 do
 4:         vF := 0
 5:         vE := (MemE[2], . . . , MemE[0], 0)
 6:         vH := (MemH[2], . . . , MemH[0], 0)
 7:         vLast := 0
 8:         vPrev := 0
 9:         for j := 0, 1, . . . , length(query)+2 do
10:             vE := (MemE[j+3], vE(2), vE(1), vE(0))
11:             vH := (MemH[j+3], vH(2), vH(1), vH(0))
12:             vF := MAX(vLast - vGapOpen, vF - vGapExtend)
13:             vE := MAX(vH - vGapOpen, vE - vGapExtend)
14:             vDelta := (δ(q_{j+3}, r_i), δ(q_{j+2}, r_{i+1}), δ(q_{j+1}, r_{i+2}), δ(q_j, r_{i+3}))
15:             vLast := MAX(vPrev + vDelta, 0)
16:             vLast := MAX(vLast, vF)
17:             vLast := MAX(vLast, vE)
18:             vPrev := vH
19:             vH := vLast
20:             memE[j] := vE(3)
21:             memH[j] := vH(3)
22:             vScore := MAX(vScore, vLast)
23:         end for
24:     end for
25: end procedure
```

organized in vectors parallel to the query sequence, as shown in Figure 2.6b. The algorithm iterates over the columns. For each column, which corresponds to a single character of the reference, the vectors E and H of the previous column are loaded, and the values of E and H of the current vector are calculated. If any of the values of the vector H are greater than α, the vector F contains values greater than 0 and that can influence the value of H. Hence, it is necessary to calculate the values of the vector F, while also updating the values of the vectors H and E, until all the values of the vector F are smaller than 0 and can no longer contribute to H. After the correct value of H is reached, the vectors H and E are stored, and the vector below the current vector is calculated. When all vectors from the current column are calculated, the program advances to the following column.

Since a query sequence is compared with multiple database sequences, a new structure which replaces the substitution matrix called query profile was introduced. The query profile indexes the scores by database symbol and query symbol position and allows the whole vector of score to be loaded with one memory access, unlike the load of scores in Wozniak's implementation in the line 14. Rognes and Seeberg [52] implementation calculates the alignment score up 13 times faster than a scalar implementation, using 8-wide vectors; and achieves a speedup of 6 compared to a scalar tool that also does not compute the values of F and E if they have no impact in the correct value of H.

A new striped scheme was presented in 2007 by Farrar [13], with the vectors shape shown in Figure 2.6c. Similarly to Rognes and Seeberg, the matrix is also calculated by iterating through the columns, but the correction to the values occurs after a whole column is calculated. This correction is known as

Algorithm 2.2 Rognes parallelisation of SW algorithm

```
 1: procedure SW_ROGNES
Require: vMemE[i], vMemH[i] are initialised to 0
 2:     vScore := 0
 3:     for i := 0, ..., length(reference) do
 4:         vF := 0
 5:         vX := 0
 6:         c := r_i
 7:         for j := 0, ..., ⌈length(query)/8⌉ do
 8:             vE := vMemE[j]
 9:             vH := vMemH[j]
10:             vT1 := vH ≫ 7
11:             vH := vH ≪ 1 OR X
12:             vX := vT1
13:             vH := vH + vProfile[c][i]
14:             vH := MAX(H, E)
15:             vF := (vH ≪ 1) OR (vF ≫ 7)
16:             vF := vF - vGapOpen
17:             if ANYELEMENT(vF > 0) then
18:                 vT2 := vF
19:                 while ANYELEMENT(vT2 > 0) do
20:                     vT2 := vT2 ≪ 1 - vGapExtend
21:                     vF := MAX(vF, vT2)
22:                 end while
23:                 vH := MAX(vH, vF)
24:                 vF := MAX(vH, vF - vGapOpen)
25:             else
26:                 vF := vH
27:             end if
28:             vMemH[j] = vH
29:             vMemE[j] = MAX(vE - vGapExtend, vH - vGapOpen)
30:             vScore = MAX(vScore, vH)
31:         end for
32:     end for
33: end procedure
```

the lazy F loop, since the values of F are only calculated if they make a difference to the final result. This new approach can be up to 3 times faster than Rognes and Seeberg [52] approach. Due to the lazy loop, the performance of this approach is also more dependent on the input data and the cost function, with higher gap costs improving the execution time, since F will, on average, contribute less to the values of the matrix H.

A graphical visualisation of the differences between the three approaches is presented in Figure 2.6, where each color represents one vector.

2.2.4 State of art of sequence alignment in GPGPUs

In 2006 Liu et al. [31] developed the first tool to take advantage of GPU to calculate the Smith-Waterman algorithm, using OpenGL. This implementation is limited to protein sequences, and only the score result is calculated in the GPU. The extraction of parallelism was done with recourse to the anti-diagonals, with the program building the entire H matrix, which is posteriorly transferred to the main memory, where the traceback occurs. On a contemporary desktop platform, with Nvidia 7800 GTX,

Algorithm 2.3 Farrar's parallelisation of SW algorithm

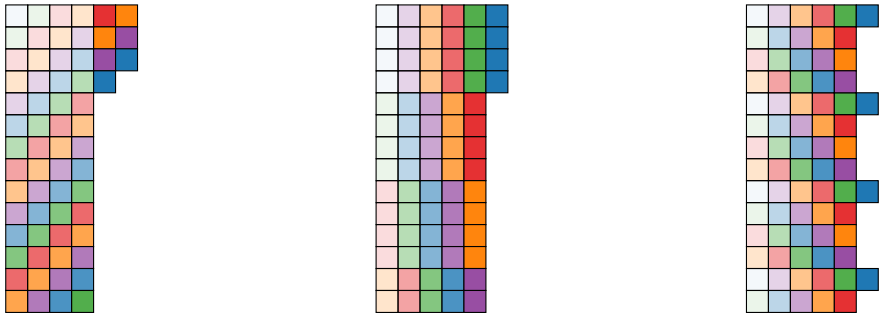
```
 1: procedure SW_FARRAR
Require: vHLoad[i], vHStore[i], vE[i] are initialised to 0
 2:     vScore := 0
 3:     segLen := ⌈length(query)/16⌉
 4:     for i := 0, . . . , length(reference) do
 5:         vF := 0
 6:         vH := vHStore[segLen-1] ≪ 1 SWAP(vHLoad, vHStore)
 7:         for j := 0, . . . , segLen do
 8:             vH := vH + vProfile[i][j]
 9:             vScore = MAX(vScore, vH)
10:             vH := MAX(vH, vE[j])
11:             vH := MAX(vH, vF)
12:             vHStore[j] := vH
13:             vH := vH - vGapOpen
14:             vE[j] := MAX(vE[j] - vGapExtend, vH)
15:             vF := MAX(vH, vF - vGapExtend)
16:             vH := vHLoad[j]
17:         end for                                          ▷ Start of lazy F loop
18:         vF := vF ≪ 1
19:         j := 0
20:         while ANYELEMENT(vF > vHStore[j] - vGapOpen) do
21:             vHStore[j] := MAX(vHStore[j], vF)
22:             vF := vF - vGapExtend
23:             if ++j >= segLen then
24:                 vF := vF ≪ 1
25:                 j := 0
26:             end if
27:         end while
28:     end for
29: end procedure
```

achieved a speedup of 9.4 times over SW running on a single thread in the CPU.

A new tool, SW-CUDA, presented by Manavski and Valle [40] parallelizes the problem by running several alignments at the same time, in an example of intertask parallelisation. In order to augment the locality of accesses to memory, it uses the query profile introduced by Rognes and Seeberg [52]. These improvements result in SW-CUDA executing in a contemporary GTX 8800 to be 1.6 times faster than Farrar's implementation. SW-CUDA only calculates the score of protein sequences.

Liu et al. [35] introduced a new tool, CUDASW++ 1.0, which uses intertask and intratask parallelism to compute the SW score of proteins sequences. In intertask computation, calculating the matrix row by row, or column by column, incurs in l load accesses to the global memory and l store accesses per l cells, as can be seen in Figure 2.7a. To lower these costs it was proposed a scheme where a block of $k \times k$ cells is created. Within the block, the values of the cells are stored in registers or local memory, lowering the number of global memory accesses to 1 per column. With the straightforward scheme, the computation of a cell block requires k^2 global memory loads, but with cells grouped in block requires only k global memory loads.

For the intratask computation it uses anti-diagonals with a changed memory layout to coalesce the memory accesses. The authors found intertask parallelism offers better performance, but at the cost of an increase in memory usage, so it is used for queries of length inferior to 3072. CUDASW++, running

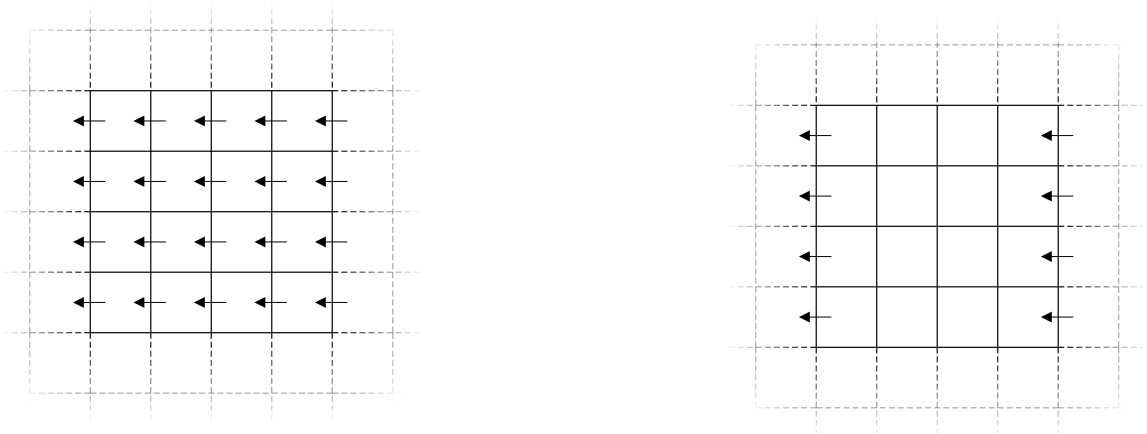


(a) Wavefront approach, introduced by Wozniak[67]

(b) Parallel to the query approach, introduced by Rognes[52]

(c) Striped approach, introduced by Farrar[13]

Figure 2.6: Comparison between parallelism approaches



(a) Memory accesses of straightforward implementation

(b) Memory accesses of block implementation

Figure 2.7: Comparison between memory accesses

in intertask mode, outperforms SW-CUDA, running on the same GPU, by up to 2 times, but the authors found it is only faster than a multi-threaded Farrar's approach for short queries with a length inferior to 850 amino acides.

CUDASW++ 2.0 by Liu et al. [36] maintains the usage of intratask and intertask parallelism of CUDASW++ 1.0, but drops the wavefront parallelisation approach in favor to Farrar's methodology, presented in page 17. Moreover, it introduces the usage of the query profile, which is then packed, to further decrease computation time, as it reduces the number of accesses to the memory and coalesces them, for the two implemented approaches.

Razmyslovich et al. [51] introduced a new tool, using OpenCL, that could simultaneously calculate the score and do the traceback of DNA sequences using non-gaped SW. The calculation of the matrix is done with recourse to diagonal blocks, with the calculation of the optimal alignment offloaded to the CPU by transfering the whole diagonal block to the main memory.

Liu et al. [38] developed CUDASW++ 3.0 in order to exploit simultaneously the CPU and the GPU, to compute the homology score of protein sequences. Moreover, it also uses of true SIMD capabilities inside each GPU thread to calculate 4 alignments per GPU thread.

GSWABE [34] can compute the score and do the traceback of DNA sequences for local (SW), global

(NW) and semi-global (NW with the initial column and row initialized to 0) alignments. It uses a tiling approach, with several different tiles computed simultaneously. For each cell in a tile, the score is calculated, which corresponds to a certain movement, match/mismatch (move from diagonal), insertion (move from upper cell) and deletion (move from left cell). In the case of local alignment, there is another option of movement, a stop. Each tile has a dimension of 4×4, corresponding to 16 cells. Since there 4 possible movements of each cell, the movement correspodent to a single cell can be represented with 2 bits, and movements of a complete tile can be stored inside a single 32 bit value. The traceback is performed in the GPU by transversing the movement values previously computed.

2.2.5 Traceback in linear memory complexity

While it is possible to calculate the homology score of two sequences in linear space $\mathcal{O}(\min(m, n))$, the calculation of the traceback for either NW or SW requires the whole H matrix, which requires $\mathcal{O}(mn)$ space. In 1975 Hirschberg [16] introduced a new approach capable of calculating the Longest Common Subsequence (LCS) in linear space $\mathcal{O}(m + n)$ maintaing the quadratic time $\mathcal{O}(mn)$. The calculation of LCS is performed using DP and share some similarities to SW and NW. Myers and Miller [43] adapted the algorithm to the calculation of NW with the gap model proposed in Gotoh [15]. The matrix is divided into two halves horizontally, in row $i* = \lfloor m/2 \rfloor$. In the top half the matrix is computed using a score only approach, and the cell values from last row ($H_{i*,j}$ and $E_{i*,j}$) are saved. In the bottom half, the score is calculated backwards, from the row m to the row $i*$, and the reversed row is stored ($H^r_{i*,j}$ and $E^r_{i*,j}$). For a fixed column j, the overall score is the score of the forward matrix ending at j plus the reverse matrix starting at j. The column $j*$ that belongs to the best score is found at the column j that maximizes the overall score $\max(H_{i*,j} + H^r_{i*,j}, E_{i*,j} + E^r_{i*,j}) - \beta$. The optimal backtrack is equal to the concatenation of the optimal backtrack from the first cell until the optimal midpoint $i*, j*$, and from the optimal midpoint until the last cell. To find the optimal backtrack of the two submatrices, the previous procedure is repeated recursively, for progressively smaller submatrices, as shown in Figure 2.8. The recursive procedure ends when the backtrack of a submatrix is trivial.

Even though this algorithm maintains the same time complexity as the original NW, it has an execution time 2 times higher than the traditional NW, since effectively the scores of the matrix H are computed twice. The storage costs are $\mathcal{O}(2n)$ for the intermediate rows, since there are two rows, from the forward and reverse matrix, stored in the first level of iteration, with a width of n. The rows are reused in the lower level of iteration, since they are no longer needed in the upper level, and the combined size of the lower levels is equal to the size of the upper level. For the computation of score-only at each phase, it is needed $\mathcal{O}(\min(m, n))$, making the overall space complexity $\mathcal{O}(n + m)$.

For SW it is possible to bound the size of the initial optimal alignment matrix by saving, in the initial score-only alignment, the start and end positions of the best alignment, and then procede using the Hirschberg's approach [9].

Sandes and Melo [54] proposed an implementation of SW algorithm, CUDAlign, capable of performing the alignment and traceback for two large sequences in CUDA enabled GPUs with a linear space

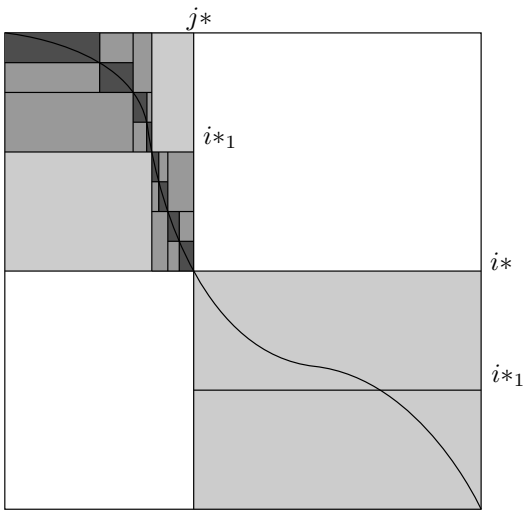


Figure 2.8: Recursive splitting Hirschberg procedure

complexity. For this the authors use a five step approach where, in the initial step, they compute the entire matrix for a pair of sequences calculating only the score, with the position of the highest score saved. The extraction of parallelism in the initial step is done by dividing the matrix into diagonal blocks that can be computed in parallel, as shown in Figure 2.9. Instead of saving only the midpoint position, like Myers, the rows at the end of some diagonal blocks, called special rows, are saved to the disk. These special rows are colored in black in Figure 2.9.

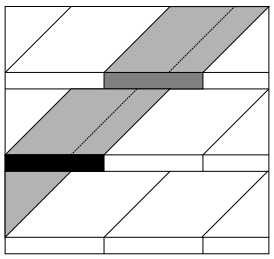


Figure 2.9: Diagonal blocks and special rows of CUDAlign

In the second step, the diagonal blocks are searched in reverse order, from the previously stored highest position until the initial position, by searching the column inside the previous special row where the optimal alignment originate. The generated row and column informations are equivalent to the Myers' optimal midpoint. A third step is conducted, similar to the second, to find more endpoints. In this step, there are two opposing effects in regards to parallelism. On one hand, the search inside all diagonal blocks can be conducted in parallel since the start and end positions are known. On the other hand, there is less parallelism to be extracted from each block since they are smaller. The fourth and fifth steps are conducted in the CPU, and consist in dividing even more the optimal alignment partitions until the optimal alignment can be calculated trivially. In the fourth step, the division of the partitions is done along the bigger dimension to ensure more balanced results. In the fifth and final step the complete optimal alignment is assembled from the previously generated fragments.

SW# is an implementation of the Myers' algorithm [43] using CUDA [18] to compute the optimal alignment of DNA and proteins in linear space. The extraction of parallelism was achieved using anti-

diagonals. The optimal alignment is calculated recursively, until the dimensions of the area to be aligned are small and the CPU takes over. This implementation is slower than that of Sandes and Melo [54] for only the biggest alignments, in the order of 33 Mbase.

2.3 Non-optimal alignment algorithms

Despite the increase of performance of optimal alignment tools, both algorithmic and by taking advantage of GPUs, existent optimal alignment tools are not suited to the alignment of DNA generated by NGS due to the high computational cost required by optimal approximate string matching.

The development of heuristic algorithms lowers the computational resources required for the alignment of DNA. The basis of heuristic approximate string matching algorithms is twofold: on one hand, relies that in every approximate occurrence of a pattern in a text there are some substrings of the pattern that match the text without any errors; on the other hand, many exact string matching approaches have been developed along the years that are characterized by a lower complexity and by lower execution time. Hence, the locations given by the exact search of these strings indicate areas of text where the best possible approximate string matches can occur. To find the approximate string match, the area around each location can then be expanded by matching additional characters from the pattern and text as long as the pattern and text do not diverge significantly. It is also possible to use the areas around the locations of seeds as targets of optimal approximate string matching. The usage of seeds to select areas suitable for a more careful search is called filtration [45].

In the example of Figure 2.10, the substrings of length 2 that start in positions 0, 1, 4, 5 and 9 would be found at least once in the text, substrings of length 3 starting in 0 and 4 would also be found in the text. However, if the substrings of the pattern have length 4 this optimal alignment would not be found in the text.

		0	1	2	3	4	5	6	7	8	9	10	
S_{text}:	...	G	C	A	G	T	G	A	—	—	T	A	...
		|	|	|		|	|	|			|	|	
$S_{pattern}$:		G	C	A	—	T	G	A	C	C	T	A	

Figure 2.10: Example of optimal alignment

Another possible technique is to break the pattern into substrings, but instead performing an exact search of the substrings in the text, the substrings are searched allowing a finite but smaller number of errors than the allowed in original approximate string matching. The substrings with errors are known as neighbours (of the original substring). The approximate search of each substring can also be viewed as an exact search on the neighbours of the said substring. This technique allows bigger substrings to be extracted from the pattern, or to be more sensitive to errors at the cost a slight increase in computation. This technique is called intermediate partioning [45]. In the example of Figure 2.10 and allowing a single mismatch, substrings of the pattern of length 3 that start in the positions 0 to 5 and 8 have an approximate match with text, while substrings of length 4 are found in positions 0 to 4 have an approximate match with the text.

The choice of the filtration parameters, namely the length of the substrings, their position, and in the case of the intermediate partitioning, the number of errors allowed in each substring become instrumental to alignment tools, impacting significantly the execution time and the quality of the optimal approximate match.

The exact search procedure can be performed faster by using offline search algorithms, which preprocess the reference text before the actual search. Several different data structures have been used in bioinformatic applications, such as hash tables, suffix trees and suffix arrays. Some of these data structures allow exact search with a time complexity of $\mathcal{O}(n)$, where n is the size of the pattern. A recent advance in exact search is the usage of the Burrows-Wheeler transform (BWT) and Full-text minute space index (FM-index) data structures on sequence alignment programs. In the following subsection will present some these exact search procedures.

2.3.1 Exact search algorithms

One of the most straightforward data structures used to accelerate exact search are hash-tables. Hash-tables are data structures that map keys to objects or values. Given a key, a hash function generates an index value to an array of buckets, selecting a corresponding bucket where the desired objects are stored. In the general case, it is not possible to assure that all keys have an unique index value, since the number of keys may exceed the number of buckets. Hence, when two keys share an index value, a collision is said to occur. However, if the number of keys is known, finite and reasonable (in other words, it is possible to create a bucket for each key), as it happens in many bioinformatic applications, it is possible to devise a perfect hash function, where collisions are impossible [5]. In exact search, the keys are usually fixed length substrings, also known as k-mers, where k is the length of the substrings, and the objects represent the start positions of the respective substrings in the text.

Figure 2.11 presents an example of a hash-table for overlapping 3-mers of the DNA sequence GCAGTGATAGCATGACCTAG. Since the number of keys (3-mers) is finite and reasonable (there are $4^3 = 64$ possible buckets, which are easily stored into memory), it is possible to construct a perfect hash function. One possible hash function would be to map A,C,G and T to 0, 1, 2 and 3, respectively, and calculate the value of the resulting base-4 integer to select the corresponding bucket.

To search a certain k-mer in a text, the index value of the k-mer is calculated through the same hash function used to generate the hash-table the text. The corresponding bucket contains the positions of the k-mer in the text. If the hash table is well designed, possessing a perfect hash function, the lookup of the correct bucket is performed in constant time, independently of the number of objects stored in the hash table. Next, it is necessary to transverse all the positions in the bucket. On average, each bucket has m/l positions, where m is the size of text and l is the number of buckets, resulting in a total search time of $\mathcal{O}(m/l)$.

The search of patterns bigger than the k-mers in the hash-table is performed by dividing the pattern into non-overlapping k-mers, which are then searched individually. The resulting positions are combined, resulting in a overall position for the pattern. Hence, this search is executed in $\mathcal{O}(n \times m/l)$ time, where

3-mers	Index value	Text position
GCA	$2 \times 4^2 + 1 \times 4^1 + 0 \times 4^0 = 36$	0, 9
CAG	$1 \times 4^2 + 0 \times 4^1 + 2 \times 4^0 = 18$	1
AGT	$0 \times 4^2 + 2 \times 4^1 + 3 \times 4^0 = 11$	2
GTG	$2 \times 4^2 + 3 \times 4^1 + 2 \times 4^0 = 46$	3
TGA	$3 \times 4^2 + 2 \times 4^1 + 0 \times 4^0 = 56$	4, 12
GAT	$2 \times 4^2 + 0 \times 4^1 + 3 \times 4^0 = 35$	5
ATA	$0 \times 4^2 + 3 \times 4^1 + 0 \times 4^0 = 12$	6
TAG	$3 \times 4^2 + 0 \times 4^1 + 2 \times 4^0 = 50$	7, 17
AGC	$0 \times 4^2 + 2 \times 4^1 + 2 \times 4^0 = 10$	8
CAT	$1 \times 4^2 + 0 \times 4^1 + 3 \times 4^0 = 19$	10
ATG	$0 \times 4^2 + 3 \times 4^1 + 2 \times 4^0 = 14$	11
GAC	$2 \times 4^2 + 0 \times 4^1 + 1 \times 4^0 = 33$	13
ACC	$0 \times 4^2 + 1 \times 4^1 + 1 \times 4^0 = 5$	14
CCT	$1 \times 4^2 + 1 \times 4^1 + 3 \times 4^0 = 23$	15
CTA	$1 \times 4^2 + 3 \times 4^1 + 0 \times 4^0 = 28$	16

Figure 2.11: Example of hash-table for the 3-mers of the DNA sequence GCAGTGATAGCATGACCTAG

n is the size off the pattern. Existing tools using hash-tables include FASTA [50], MAQ [26] and SOAP [28].

Despite their widespread use, hash-tables do not have an optimal time complexity. The suffix tree [62, 65], on the other hand, is a data structure which possesses an optimal time complexity, being capable of performing an exact search in linear time in respect to the size of pattern. However, the memory required for the storage of the suffix trees is much higher than the size of the reference, requiring up to 16 bytes to represent a single base pair from a DNA sequence [19]. Despite the high memory consumption of the suffix trees, they are used by MUMmer [12, 19] to perform the alignment of two DNA sequences.

To create a suffix tree, it is necessary to append to the end of the text T a stop character (represented by $). This stop character is lexicographically smaller than all characters present in the text. The suffix tree stores all suffixes of the text T, that is to say, stores all substrings of T starting in the i-th character and that end at the stop character. This data structure can be built in linear space and time in relation to the number of characters of the text [62]. As it can be seen in Figure 2.12, each leaf of the suffix tree stores the index of the position where the suffix begins. Moreover, for each suffix, there is only one path from the root node until the respective leaf. To search a pattern in the text T, one starts in the root node. Then, the child node matching the first characters of the pattern is selected. The next child node is selected by matching the subsequent characters from the pattern, continuing until either a complete match occurs, or if at some point no match is found.

For instance, for the pattern *ssi*, the search would start at the root node. Since the edge *s* matches the first character of the pattern, that correspondent node is the first to be visited. The next characters from the pattern, *si*, match exactly with the edge *si* and the search is stopped at the correspondent node, with the query found in the text. By transversing the leaf nodes of the node where the search stopped, the positions of the text where the pattern is present can be found. The example pattern is found at positions 4 and 7 of the text.

To reduce the memory consumption problems associated with suffix trees, a related structure called

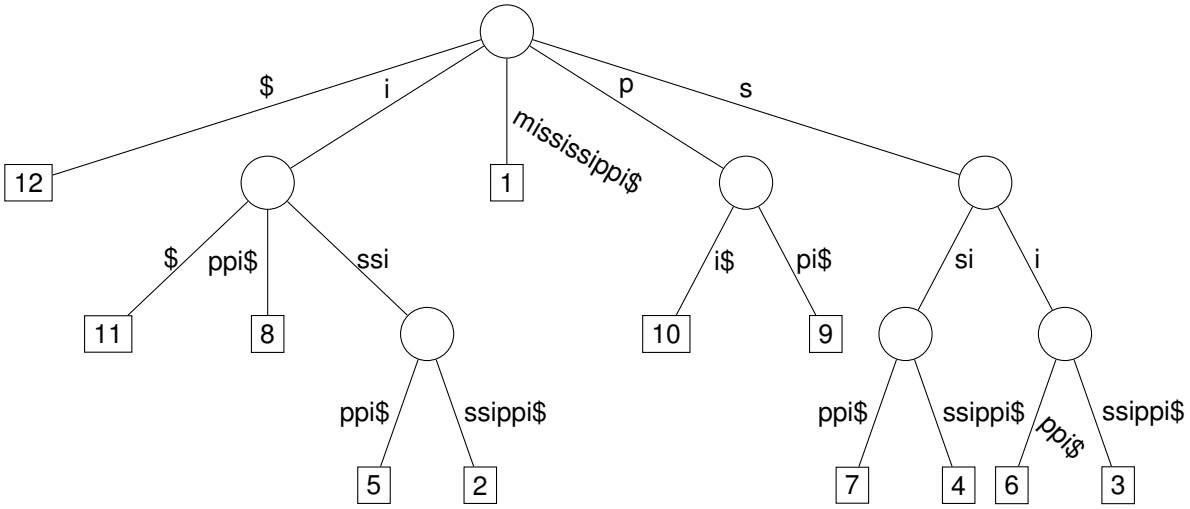


Figure 2.12: Example of suffix tree of *mississipi*

suffix array was developed [41]. Suffix array is a data structure that stores in an array all suffixes of a text T sorted in a lexicographical order. Similarly to the suffix trees, the generation of the suffix array requires appending a lexicographically smaller stop character (e.g. $) to the text. After the generation of all the suffixes of the text, they are sorted in a lexicographically order. Consequently, the i-th suffix will be in the k-th position of the suffix array if it is the k-th lexicographically smaller suffix. The resulting order of the suffixes is the suffix array. The straightforward procedure to generate the suffix array involves sorting the suffixes, an operation which has in the best case a complexity of $\mathcal{O}(m^2 \log(m))$, Manber and Myers [41] also proposed a technique to create the suffix array in $\mathcal{O}(m \log(m))$ time.

An example of the creation of the suffix array for the same text as for the suffix tree is available in Figure 2.13. By comparing the suffix array with the suffix tree, available in Figure 2.12, it is possible to observe that the suffix array can be obtained from the suffix tree, by transversing the leafs of suffix tree using a lexicographical order.

1	`mississippi$`
2	`ississippi$`
3	`ssissippi$`
4	`sissippi$`
5	`issippi$`
6	`ssippi$`
7	`sippi$`
8	`ippi$`
9	`ppi$`
10	`pi$`
11	`i$`
12	`$`

(a) Suffixes of text

12	`$`
11	`i$`
8	`ippi$`
5	`issippi$`
2	`ississippi$`
1	`mississippi$`
10	`pi$`
9	`ppi$`
7	`sippi$`
4	`sissippi$`
6	`ssippi$`
3	`ssissippi$`

(b) Sorted suffixes

Figure 2.13: Example of suffix array of *mississipi*

A pattern can be searched using suffix arrays in $\mathcal{O}(n \log(m))$ time using binary search. The time complexity can be improved by using an auxiliar data structure, called longest common prefix, to $\mathcal{O}(n + \log(m))$. This data structure also allows the construction of the suffix array in linear time $\mathcal{O}(m)$.

Recent sequence alignment tools have introduced the usage of the Burrows-Wheeler transform (BWT), combining an optimal time complexity of $\mathcal{O}(n)$ with a reasonable memory consumption. The BWT is a block-sorting lossless data compression algorithm. The algorithm applies a reversible transformation to a text, reordering it in such a way that other compression methods can more easily compress the text.

Similarly to suffix trees and suffix arrays, the first step of the algorithm is the introduction of a lexicographically smaller stop character, for instance $, to the end of the text, creating a changed text T'. The text T' is then progressively rotated to create a matrix with all possible rotations of the text. This matrix is then sorted lexicographically, and the last column of the matrix is extracted. This column is the BWT of the changed text T', or $BWT(T')$. An important aspect to note is that after the matrix is sorted lexicographically, the order of the rotations is equal to the suffix array indexes. An example of the process of creation of the BWT using the same text as for suffix arrays is presented in Figure 2.14, where the BWT is marked in subfigure 2.14b. It can be seen that there is a great deal of similarity between the creation of the suffix array and the creation of the BWT. In fact, it possible to create the BWT from the suffix array through the equation 2.6, enabling the creation of the BWT in linear time by constructing the suffix array in linear time.

$$BWT(i) = \max \begin{cases} T(SA(i)-1) & \text{if} SA(i) \neq 0 \\ \$ & \text{if} SA(i) = 0 \end{cases} \tag{2.6}$$

The BWT matrix (see example in 2.14b) has a property called last-to-first column mapping, or just last-first (LF) mapping, which states that the i-th occurrence of character c in the last column (F) corresponds to the same text character as the i-th occurrence of c in the first column (L).

1	mississippi$
2	ississippi$m
3	ssissippi$mi
4	sissippi$mis
5	issippi$miss
6	ssippi$missi
7	sippi$missis
8	ippi$mississ
9	ppi$mississi
10	pi$mississip
11	i$mississipp
12	$mississippi

(a) Rotation of the text

12	$mississipp	i
11	i$mississip	p
8	ippi$missis	s
5	issippi$mis	s
2	ississippi$	m
1	mississippi	$
10	pi$mississi	p
9	ppi$mississ	i
7	sippi$missi	s
4	sissippi$mi	s
6	ssippi$miss	i
3	ssissippi$m	i

(b) Sorted cyclic suffixes

Figure 2.14: Example of BWT of *mississipi*

It is possible to generate the first column (F) of the rotation matrix from the $BWT(T')$, or, in other words, from the last column (L) of the rotation matrix, by sorting it lexicographically, since the first column is obtained by sorting lexicographically T', which by definition has the same characters as the $BWT(T')$. After the first column is created, it is possible to recreate the original text by transversing the BWT by using the LF mapping. The first step is to select the first character from F. This character is the

last character from the original text. Using the LF mapping, we proceed to the first occurrence of the selected character in the first column. The character in the last column precedes the already selected character. The algorithm then iterates in the same manner, until the character \$ is found in the last column, signaling the end of the string. An example of this procedure, which is the inverse of the BWT, is presented in Figure 2.15, where the red arrows represent the LF mapping. By following the algorithm, represented by arrows in the example, we can see that the string stored in the BWT ”ipssm\$pissi” is the word ”mississippi\$”.

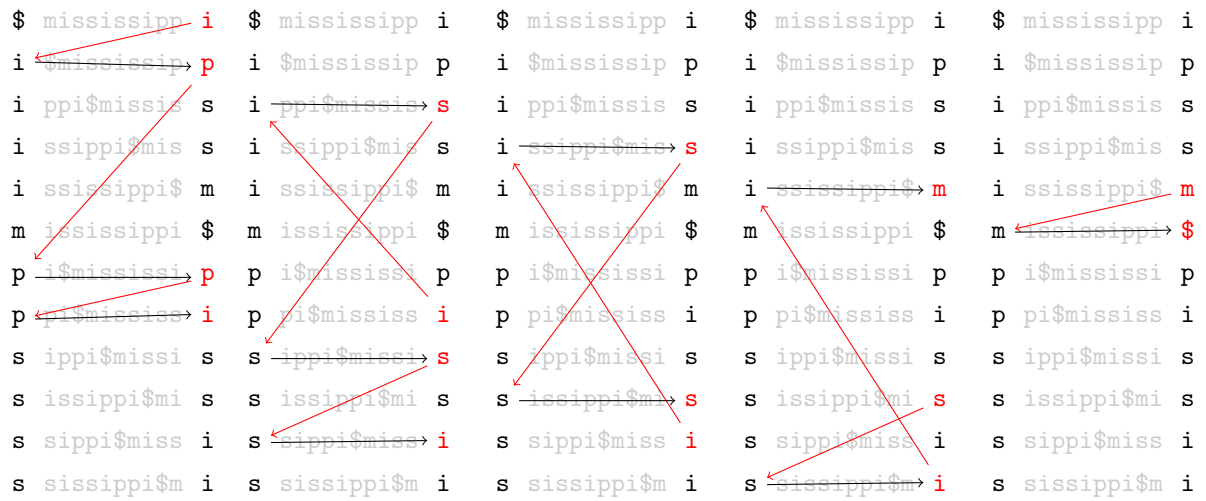


Figure 2.15: Inverse BWT of ipssm\$pissi

Exact search using the BWT is performed by searching the pattern backwards. First, the last character of the pattern is searched in the F column to find a range of positions where the pattern can match. Then, the corresponding L rows are searched for the previous character of the pattern, resulting in a range of L rows. Using LF mapping, this range is converted to a range of F rows. At each new character, the size of the range is either maintained or shrinks. The procedure then repeats until the whole pattern has been matched, or the range becomes 0, in which case the pattern is not present in the text.

An example of the exact search of ”ssi” in the text ”mississippi” is available in Figure 2.16. The first character to be matched is the last character of the pattern. The character ”i” is found 4 times in the text. In the range of ”i”, the next character, ”s”, has a range of 2. Next, the range of ”s” is mapped to the first column using LF mapping, marked in red in the figure. The final character, ”s”, keeps the same range. Since the final range has a size of two, the pattern is found twice in the text.

Ferragina and Manzini [14] proposed a compressed full-text index, by taking advantage of the similarities between the BWT and the suffix array. This data structure is known as Full-text minute space index (FM-index), or Ferragina-Manzini index. Two structures are introduced by this algorithm, the C vector and the OCC matrix. The C vector stores the number of distinct characters present in the text T'. For each character of the text, the number of characters lexicographically smaller than that character is stored in the corresponding array position. The $OCC(c, i)$ matrix stores the number of times a character c is present in the i-th prefixes of the $BWT(T')$, where the i-th preffix is the substring $BWT(T')[1 \ldots i]$.

Figure 2.16: Exact search of ssi using the BWT of mississippi

Using these data structures, the LF mapping can be expressed through following equation:

$$LF(i) = C(L(i)) + OCC(L(i), i) \tag{2.7}$$

Thus, given a character stored in the i-th position of the last column of the BWT, $L(i)$, we can calculate the corresponding position in the first column, $LF(i)$. This enables searching for an exact string match in the same manner as the aforementioned exact search using BWT, by iterating in a backwards manner over the pattern, finding the correspondent range of the suffix using LF mapping equation 2.7.

Even though the OCC matrix occupies $\mathcal{O}(m)$ space, it is possible to sample a few rows, and from those rows recreate all the needed values from the matrix. This technique lowers the memory required by the matrix at the cost of increasing the computational cost. Nevertheless, this characteristic of BWT and FM-index allows the implementation on devices with limited memory, such as CPUs and specially GPUs. Therefore, these data structures were chosen for the implementation of exact search on BowMapCL v1.0 [47], the tool the present work is based on.

2.3.2 Approximate search tools

FASTA [50] achieves faster execution times through the restriction of optimal alignment to the area most likely to contain the maximum local alignment through heuristic methods. It operates in 4 steps: the program searches for subsequences of length *k* (henceforth known as *k-mers*) from the query present in the reference using a lookup table. If several *k-mers* are close to each other (in a diagonal shape) they constitute a region. The 10 best regions are selected according to a score calculated from the number of *k-mers* and the length of the region. In the second step those regions are re-scored taking into account the scoring matrix and insertion/deletion costs. These regions are then tentatively joined to create possible macro-regions. The greatest scoring region will then be searched through a modified Smith-Waterman algorithm in a band centered in the diagonal formed by the macro-region.

BLAST [2] uses differing approaches to DNA and proteins. For proteins a list of *k-mers* from the query is created, where all *k-mers* must score a minimum score against the reference. For DNA all contiguous *k-mers* are extracted. The reference is scanned for *k-mers* hits through the use of a finite state machine. If one is found, the hit is extended in one direction to find a locally maximal segment, ignoring possible gaps.

The emergence of NGS led to appearance of tools using non-optimal approximate matching specially designed to align short reads. MAQ [26] and SOAP [28] used hash-tables as its exact search algorithm, allowing errors in the seeds.

2.3.3 Approximate search using BWT FM-index

Existing offline exact search algorithms, either came with heavy requirements in terms of memory but had good execution times, (e.g., when using suffix arrays), or had higher execution times, but the memory requirements were acceptable (e.g. by using hash tables). The introduction of Burrows-Wheeler transform (BWT) with FM-index led to exact search implementations combining efficient search and reasonable memory requirements.

Bowtie [23] applies this novel data structure to the alignment of DNA reads to perform the exact search. Since reads can have mismatches, bowtie implements a greedy backtracking search, where a small number of bases may be changed (mutated), if the resulting match is longer.

SOAP2 [29] also uses BWT and FM, by breaking the DNA reads into seeds in order to allow mismatches. If only one mismatch is allowed, for instance, the query is divided into two seeds, guaranteeing that if there is one mismatch it will only prevent one seed from being found in the reference.

Bowtie2 [22], like its predecessor, uses BWT with a different approach: overlapping seeds are extracted from the queries and are searched through BWT search, while still allowing a reduced number of mismatches (combining filtration and intermediate partitioning), although the default mode is to search the seeds without any mismatches. The positions of the seeds found are used to start an optimal search using dynamic programming algorithm similar to SW, enabling a better gap model than the existing approach.

2.3.4 BWT FM-index using GPGPUs

Taking advantage of the lower memory requirements of BWT with FM-index, Liu et al. [37] adapted the algorithm to execute efficiently on GPUs. The search of the seeds is done using intermediate partitioning, by allowing each seed to map the text using only substitutions.

Liu et al. [30] converted SOAP2 to be efficiently executed in GPUs, creating SOAP3, achieving a speedup of up to 10 times compared to the CPU version.

SOAP3-dp, presented by Luo et al. [39], is a DNA alignment tool using both BWT and dynamic programming. It works by trying to align each query to the reference using simultaneously the BWT of the reference and the backwards reference, a technique they have designated 2way-BWT, allowing mutations when performing the BWT. If the initial phase fails, the query is divided into seeds, which

are then mapped to the reference to discover the areas where modified SW will take place. Due to the addition of the DP, SOAP3-dp can have better with reads with gaps.

CUSHAW2-GPU is a program developed by Liu and Schmidt [32] to align short reads of DNA. Each read is broken into seeds to be searched in the reference, indicating the possible mapping regions. The mapping regions are searched using score-only SW algorithm. The best scoring region of each query goes through another round of SW, to perform the backtracking.

2.3.4.A Presentation of BowMapCL v1.0

As stated previously, the present work is based upon the tool BowMapCL v1.0, proposed by Nogueira [47]. BowMapCL is a exact search tool targeting highly heterogeneous platforms, using BWT and FM-index to perform the alignment. This tool is capable of performing the exact search in DNA, Proteins or Text.

In contrast to existing alignment tools, the proposed tool uses the OpenCL API to be able to execute in different accelerators from different vendors. Since the accelerator (and host) have varying quantities of memory, the tool is also capable of adjusting several parameters, such as the row sampling, to limit the memory consumption. Moreover, it can also split the reference sequence into multiple blocks and computing the BWT for each individual block, allowing the processing of any reference sequence, regardless of the size of the input data. Since the BWT, as it was seen previously seen, is a structure mostly used for offline exact string search, BowMapCL v1.0 has two operations modes. In the first, the index files for the search are created, taking into account the size of host and accelerating devices memory to create data structures (index files) which fit into the available memory. Furthermore, the reference text can also be broken into blocks to further decrease the memory required. The second mode is the principal operation mode. This mode reads the previously created index files and the file containing the reads/queries, which are then searched exactly in the reference text. The result is the absolute position(s) of the queries in the text.

In order to hide the communication costs between CPU and GPU, Nogueira [47] devised an architecture, see Figure 2.17, where multiple threads, each with its own buffer, enqueue data and kernel executions to the kernel. Another thread, known as consumer thread, generates the queries which will be searched by reading the input file. This architecture will be further explored in the following chapter.

The communication between producer and the consumers is performed through a circular queue, as seen in Figure 2.18. This allows for a scalable architecture, capable of taking advantage of multiple accelerating devices.

2.4 Summary

The discovery of sequence alignment is performed optimally with Needleman-Wunsch algorithm for global discovery, and with Smith-Waterman algorithm for local discovery. Even though SW is ammenable to intertask and intratask parallelisation, and has been sucessfully ported to GPUs, high computational costs prevent its use for alignment of millions of short DNA sequences. The backtrack of the alignment

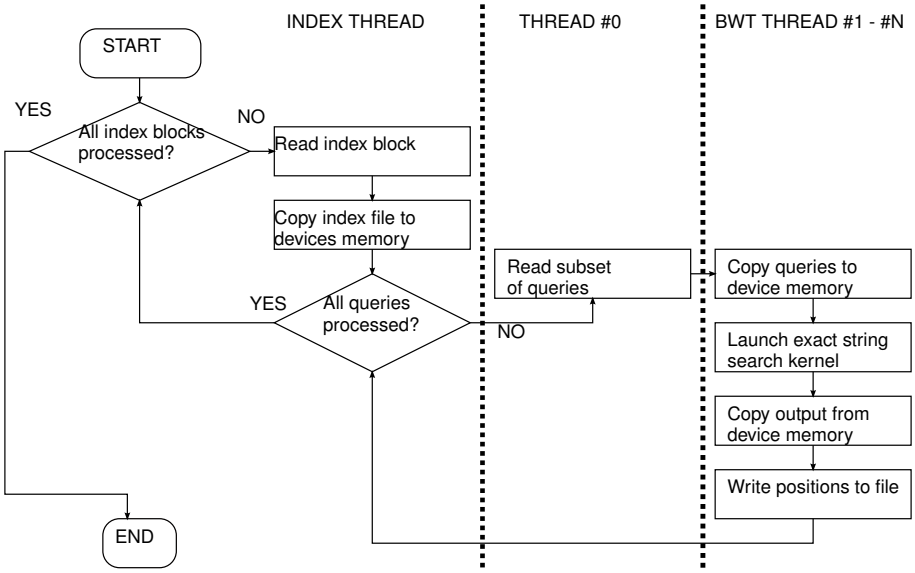


Figure 2.17: Flowchart of the parallel solution BowMapCL v1.0 for exact string matching

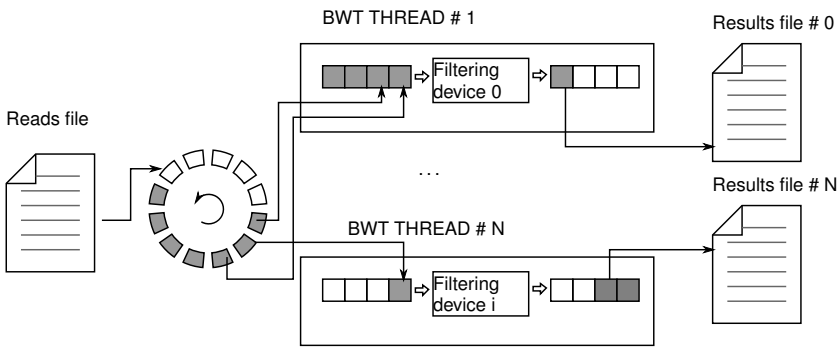


Figure 2.18: Architecture of BowMapCL for exact string matching

can be performed with quadratic memory consumption, or linearly, at the cost of an increase in running time.

Non-optimal alignment methods can be performed faster by breaking the reads into seeds, which are matched using exact search or allowing some errors in the exact search. Burrows-Wheeler transform is a data structure which enables fast exact search with reasonable memory usage. Moreover, this data structure has been ported to GPU architectures efficiently. The state of the art alignment tools, shown in table 2.4, combine BWT and optimal search to achieve the required execution times and adequate homology scores, even if there are gaps in the match. Nevertheless, they have some restrictions, some of which are presented in table 2.4. One of differences of the proposed tool, as stated in the objectives, is the capability of a cross-vendor DNA alignment tool, which no other available tool, to the author's knowledgle, is capable of. Moreover, the proposed tool should also be scalable with respect with the number of GPGPUs, unlike all the tools presented in table 2.4. Existing alignment tools are also restricted to the data they operate on. The proposed tool, like its predecessor, should be agnostic in terms of the type of data processed, being capable of support any alphabet A that can be coded in 8-bit chars.

Tool	Device	Multi-device utilization	Data type agnosticism	Unlimited index size	System requirements
Bowtie2	CPU	-	-	-	-
SOAP3-dp	GPU	-	No	-	Many [1]
CUSHAW2-GPU	GPU	-	No	-	Many [2]
BowMapCL	GPU	Multiple GPUs	Yes	Yes	Few

Table 2.4: Comparison of state-of-art alignment tools

[1] 16 GB of main memory, CUDA-enabled GPU with compute capability 2.0 and at least 3 GB of graphics RAM
[2] 6 GB of main memory, CUDA-enabled GPU with compute capability 2.0 and at least 4 GB of graphics RAM

3

Proposed Parallel Architecture

The huge amounts of short reads generated by sequencing machines pose a challenge to DNA alignment tools. The optimal approximate matching algorithms which are best suited to the alignment of biological data are based on dynamic programming, such as the Smith-Waterman algorithm. However, these algorithms have time complexity of $\mathcal{O}(mn)$, where m and n are the lengths of the reference and query, posessing a prohibitive computational cost.

Non-optimal approximate string matching algorithms decrease the execution time of the alignment at the cost of potentially missing some optimal results. This is achieved by extracting seeds from the pattern to reduce the area searched by optimal algorithms. The two available techniques for non-optimal approximate string matching are intermediate partitioning and filtration. In the former, the seeds are matched against the text with a reduced number of errors. When a seed is found, it is extended until the complete pattern is matched. In filtration, however, the seeds are matched exactly against the text, generating areas which will subsequently be searched with an optimal algorithm. It is also possible to combine the two techniques, by trying to match the seeds approximately against the text and searching the resulting positions with an optimal algorithm. Considering the fact that the primary use case of this tool is biological sequences, it is natural to implement filtration and use SW as the optimal algorithm. Hence, as explained in section 2.3, the seeds can therefore match exactly against the text (pure filtration), or match with errors (combined approach).

Considering that the reference sequences are immutable, and can be pre-processed, the exact search is amenable to offline exact search algorithms, capable of achieving $\mathcal{O}(n)$ per query, where n

is the length of the query. The BowMapCL v1.0 exact string matching tool, developed by Nogueira [47] uses BWT and FM-index data structures to create a solution targeting heterogeneous platforms, namely GPGPUs. Since BowMapCL v1.0 only implements exact string matching, the natural approach to attain non-optimal approximate string matching is to rely on pure filtration.

Hence, the main objective of this dissertation is to propose a tool that implements an efficient sequence alignment tool, combining the existing (BowMapCL v1.0) exact string matching with filtration and an optimal search algorithm implementation. This tool, henceforth designated as BowMapCL (v2.0), will be designed to take advantage of the parallelism offered by General Purpose Graphics Processing Units (GPGPUs).

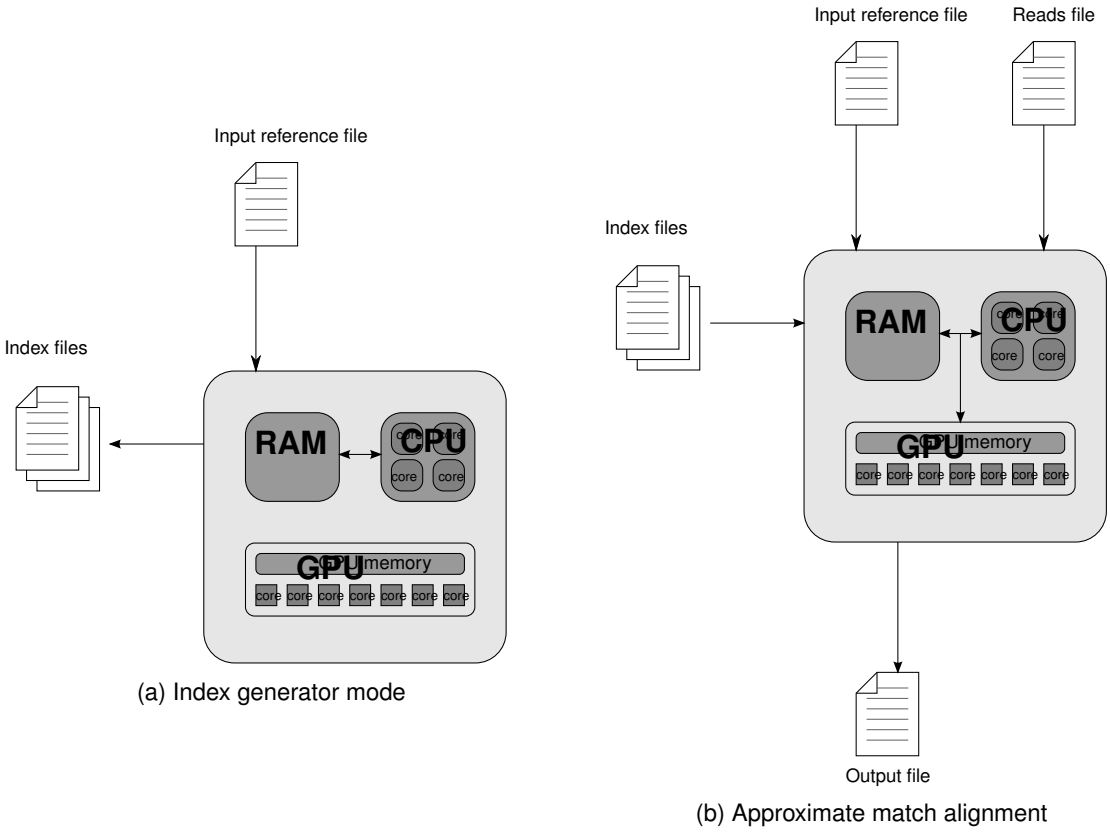


Figure 3.1: Input/output of BowMapCL in each operation mode

BowMapCL has two operation modes: the index generator and the approximate match alignment, the focus of this work. The index generator receives an input reference file in FASTA format [50] and creates all the data structures necessary to perform the exact string matching procedure, namely the BWT, the OCC matrix and the C vector, as well as the suffix array, and stores them in the index files (see Figure 3.1). It is only necessary to execute the index generator if the data structures for the reference file have not been created or to change the size constraints of the data structures necessary for the exact search. In the following sections we will discuss how the approximate string matching mode is implemented.

3.1 Proposed approximate string matching algorithm

In approximate string matching mode, also known as sequence alignment mode (see Figure 3.2) the objective is to find the position corresponding to the best match (or set of matches) between the reference and the query sequence, while allowing errors. To accomplish such a procedure, a reference file is accompanied with the corresponding data structures and a input file of queries (see Figure 3.1). The search is then performed by the following 3 steps. In the first step a batch of queries is read from the input file. This batch is then passed onto the second step, where the potential areas from each query are found through filtration, by breaking each query into overlapping seeds and performing an exact search on each generated seed. The effective positions of the seeds in the text are retrieved using the suffix array, indicating the potential positions of the query. In the third step, an optimal approximate search is performed between each query and the correspondent potential reference areas in order to find the best best matching position, which are then returned.

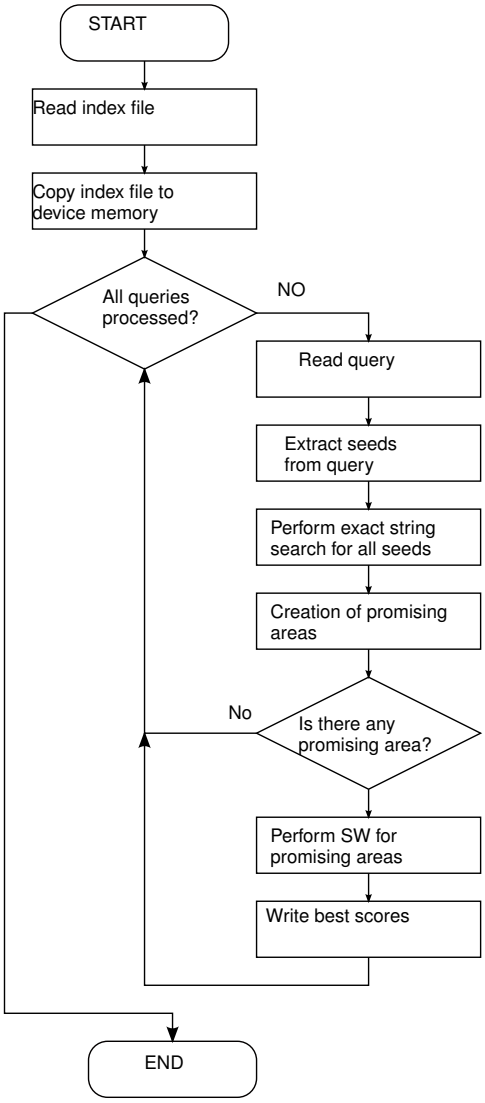


Figure 3.2: Flowchart of single-threaded CPU implementation

3.2 Proposed parallelisation approach overview

In order to create an efficient approximate string alignment tool that efficiently exploits heterogeneous computing devices, the architecture of the program must take into account several different factors, such as: the differing architectures of GPGPUs and CPUs, which are suited to different computational tasks, the memory size restrictions which are present in the computing devices, specially in accelerator devices such as GPUs, the organization of the memory in GPU devices favoring memory accesses in a coalesced manner, the overhead of the input and output from operations to the disk, the reduction of the overhead of data transfers from and to the GPU, and the throughput scalability of the program with regards to the number of accelerating devices, bearing in mind the possible performance asymmetries between devices.

Considering the fact that the size of the memory of GPUs is usually smaller than the host memory and that data in the GPU must be explicitly transferred to and from the host memory, it is necessary for the program to iterate over subsets of queries. For each subset of queries, it is necessary to: (i) read the queries from the input file, (ii) break the queries into seeds, (iii) enqueue the transfer of the seeds over to the GPU, (iv) enqueue and wait for the execution of the exact search kernel, and, finally, (v) enqueue the transfer of the results back to the host memory. The results are then filtered in the CPU to select the most promising areas to be searched by the SW algorithm. For each group of selected queries, it is then necessary to: (i) enqueue the transfer of the queries and reference sections to the GPU, (ii) enqueue and wait for the execution of the optimal search kernel, (iii) enqueue the transfer of the scores back to the host memory, and, (iv) the results are returned. The complete flowchart is as depicted in figure 3.3. The CPU will manage the GPUs, enqueue the data transfers and kernel executions, take care of the input and output of the program and perform the conversion of the SA ranges. The conversion of the SA ranges created by the exact search kernel is performed on CPU since the required suffix array is too large to be stored in the device memory. The accelerator devices, namely the GPU, on the other hand, will perform the most compute-intensive and parallelisable tasks of exact search and optimal alignment.

With regards to the parallelisation of the program across several devices and the overlap of computation and memory transfers in the device, OpenCL starting from version 1.1 has two alternatives (in OpenCL 1.0 only the first alternative was sanctioned):

1. perform the management using exclusively OpenCL functions, by relying on non-blocking calls to submit commands to multiple devices. Moreover, to overlap the computation and communication in a single device and maximise device usage, it is also possible to set the command queue to out-of-order execution, enabling the OpenCL driver to execute non-dependent (from different iterations) commands concurrently.

2. perform the management manually, using several threads concurrently, with each one managing a single command queue.

In the previous work of Nogueira [47], these two options are considered. In the first option, and considering an in-order execution with blocking operations, each thread would not have major operations besides the creation of the seeds, consequently being kept idle most of the time waiting for the

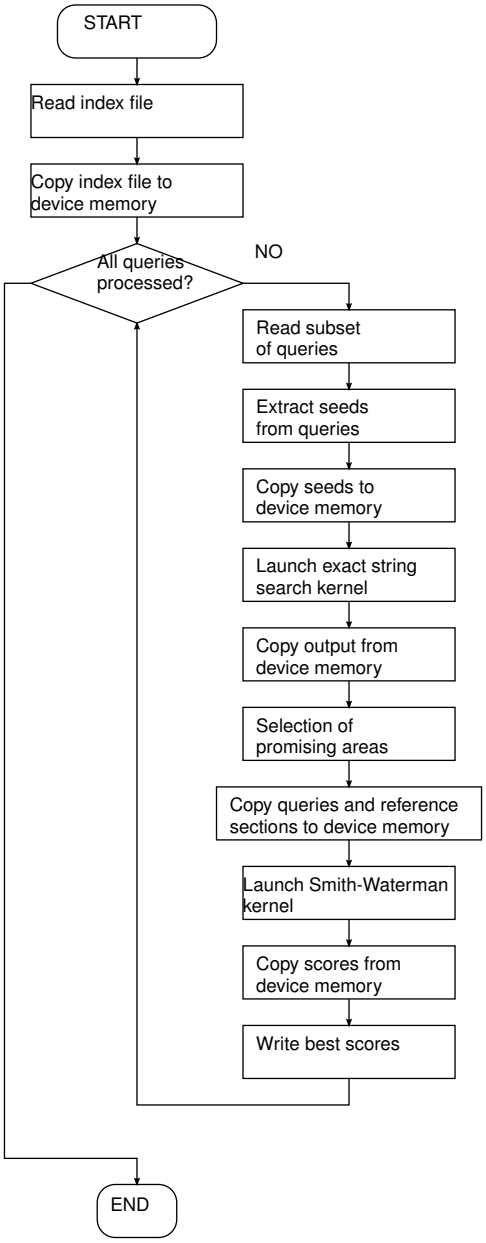


Figure 3.3: Flowchart of GPU based implementation

commands to finish, since it would need to wait for the results of the previous operation before executing the following operation. On the other hand, if the operations are non-blocking, the thread could spend the time preparing the data for the next iteration of the process and/or to prepare the data for another device.

However, if the targeted accelerating device does not support OpenCL non-blocking operations, the thread would be blocked waiting for the completion of each individual command. This precludes the usage of several devices efficiently, since only one command (and therefore one device) could be in execution, per thread. This leaves the program vulnerable to the individual capability of the devices, which may not support non-blocking operations, therefore impacting the efficiency of the program significantly.

Out-of-order command queues allow the thread to enqueue several commands, offloading the tracking of dependencies between the commands to the driver of the device. This enables the creation of a

list of pending tasks to be executed in the device ensuring that the device has a continuous flow of work available, and which is independent of the order the commands were issued. To ensure correct results each command receives a list of events depending on the completion of other commands and that must finish before the present command can start and in turn returns an event that future commands can depend on.

On the other hand, if a command queue is in-order, the commands must be executed by the order they are received. As such, if there is the possibility of running a data transfer for the next iteration while the current iteration has a kernel executing on the device, this possibility must be explicitly managed by the host.

The advantage of using of out-of-order queues is, however, negated by the fact that some OpenCL devices do not support them, executing the commands in the same manner as in-order queues. A common alternative to overlap computation and data transfers without depending on the capabilities of specific devices is to use several in-order queues per device and submit different commands that can run in parallel in different command queues.

By taking such remarks in consideration, the three steps of the approximate string matching form a loosely coupled pipeline, where each step places the generated data into a buffer, from where it is consumed by the following step, thus allowing a more fine-tuned control of the quantities of data processed by each phase. Moreover, this approaches enables the extraction of task level parallelism by taking advantage of the spatial parallelism of GPGPUs.

3.2.1 Single OpenCL device management

To manage the steps of filtering and optimal search, BowMapCL extends the architecture of its predecessor, shown in section 2.3.4.A. This was performed by creating a new set of threads, from thread #SW_1 until #SW_M, which perform the optimal alignment using the GPU. As can be seen in the flowchart presented in Figure 3.4, the SW threads receives the promising regions created by the filtering stage and produce the best scores for each query.

To generate promising regions it was also necessary to extend the exact search. Hence, the thread #0, in charge of reading the files, was also extended to perform, optionally, the reverse-complement of the queries. The filtration step is performed by extending the existing BWT threads. Before the exact search, the seeds are extracted from the queries in the CPU, and after the exact is performed, the most promising regions from each query are selected.

This architecture circumvents the problems described previously and ensures that the device is constantly processing data, even though the individual threads may be stalled waiting for data transfers or waiting for the conclusion of the kernel, since it is possible to have several command queues for a single device, while simultaneously allowing scability in regards to the number of devices.

An important step to feed the threads is the communication between the threads. Since a new stage has been added, it was necessary to add another buffer, between the BWT (filtering) threads and SW (optimal alignment) threads, as shown in Fig. 3.5. Moreover, as it will be seen in the section 4.3, the

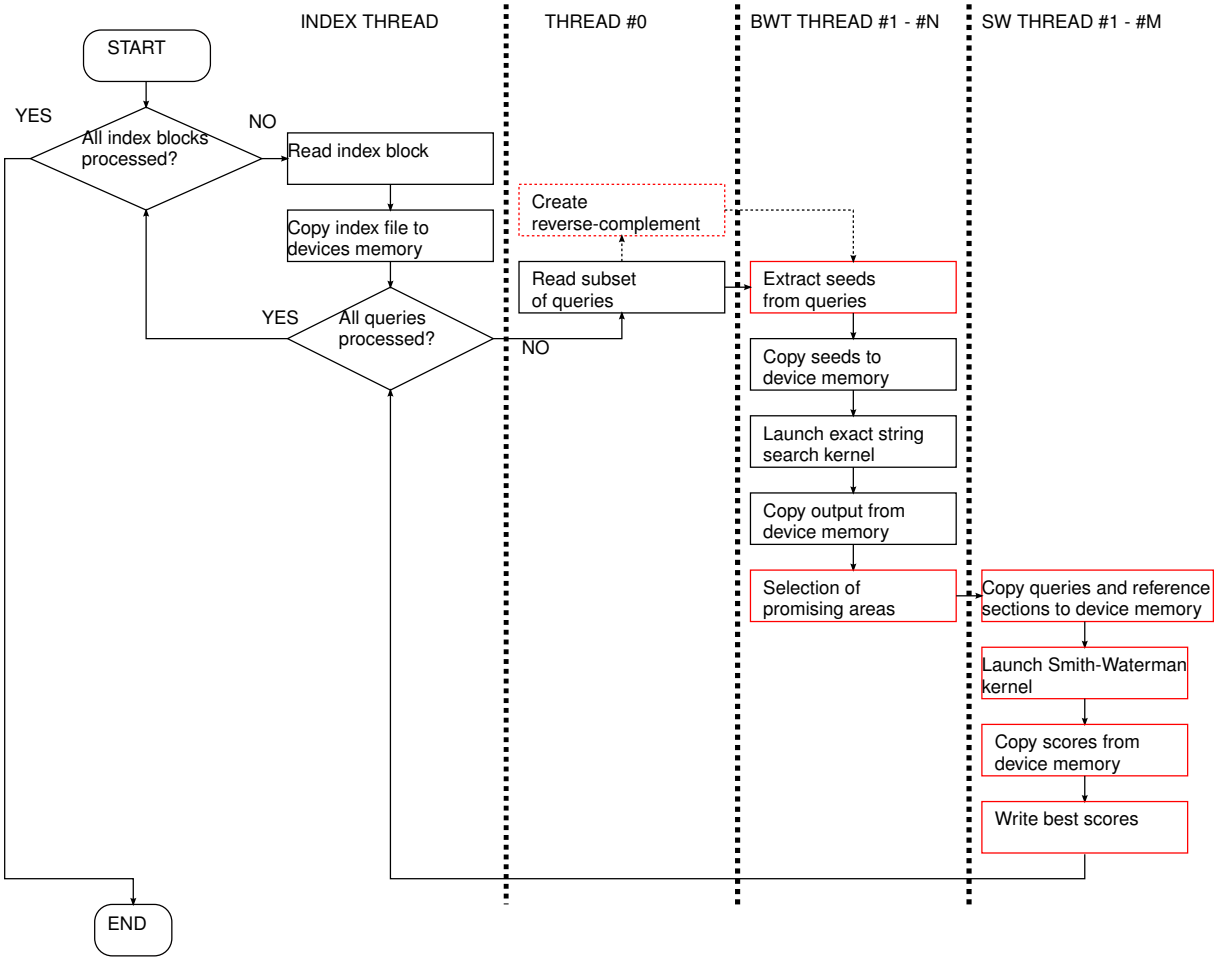


Figure 3.4: Flowchart of parallel solution of non-optimal approximate string matching, with the steps introduced in the present work in red

existing communication buffer was also changed to improve the throughput of the tool.

3.2.2 Filtration algorithm

As previously stated, the BWT this work is based on only performs exact search and does not search for mismatches of any kind. The only existent tool sharing this restriction is bowtie2 [22]. Thus, BowMapCL uses an algorithm inspired by bowtie2 since the choice of the algorithm heavily impacts the quality of the results as well as the execution time. In particular, different read files have different lengths of the reads. Moreover, some reads have fewer mismatches between themselves and DNA, for an hypothetical optimal match. Therefore, an important parameter in filtering is the length of the seeds. An increase in the length, for a constant number of seeds per query, increases the quality of each region found, at the expense of increasing the computational cost of the exact search. Moreover, due to the increase selectivity of the seeds, the number of potential matches also drops, depending on the relative quality of the queries. Another parameter is the distance between seeds, which is inversely proportional to the number of seeds per query. An increase in number of seeds per query increases the probability of finding a match in the reference, while also increasing number of exact searches to be performed.

Hence, it is necessary to minimise the cost of the exact search, while at the same time maximise the

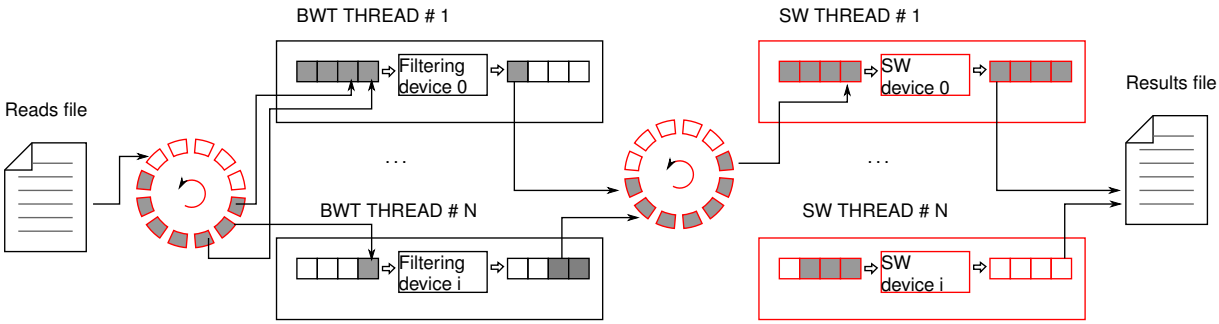


Figure 3.5: Flowchart with buffers between the different stages, with the data flows introduced in the present work in red

number of queries with promising regions and the quality of the found regions, while taking into account the variability of the data. Due to the complexity of the tradeoffs involved, its analysis is outside of the scope of the present work.

The filtration happens in two steps, the creation of the seeds for the exact search, and the selection of the possible optimal matching regions. Both steps are performed on the CPU since they do not map efficiently to GPUs, the latter due to the pattern of memory accesses, and the former because of the conversion of BWT positions to text positions requiring a suffix array, which has a large memory footprint.

As a matter of fact, due to the limited memory of heterogeneous devices, it is necessary to carefully manage the available memory, as we will seen in the next section.

3.3 Memory management

The size of the genomes involved present a challenge to the parallelisation on GPUs. BowMapCL v1.0 adopts efficient data structures in order to minimize the memory occupied by the BWT. Nevertheless, for large genomes, the full index may not fit entirely into the memory of the majority of accelerating devices. To solve this problem and make the tool capable of handling texts of any size, Nogueira [47] implemented a technique where the text is partitioned into blocks that fit into the available memory. This partition is made when the index is generated and the block sizes can either be computed automatically by the index generator or defined by the user.

The search through the blocks is conducted by processing sequentially the index blocks and matching all the queries against each block, since this approach reduces the communication overhead for the data structures. To be able to report only the best result of the optimal alignment, it is necessary to have all the optimal results for all the index blocks simultaneously. This could be achieved by saving all promising areas and conducting the optimal search only after the filtration of all index blocks is finished, or by saving all the optimal scores from the optimal alignment and conducting a post-processing step selecting the best scores. Both methods impose an heavy penalty in the host memory required to store the intermediate results. Consequently, it was chosen to report the best alignment inside each index block and storing the results directly to the output file, possibly resulting in a duplication of results for the best location.

In regards to Smith-Waterman algorithm, the major memory requirement is the storage of the reference block pertaining the current index block in the host side to be able to fetch the reference sections.

3.4 Management of multiple devices

The proposed architecture was designed to be scalable with respect with the number of the devices. As it was previously stated, the OpenCL framework allows the usage of several different classes of devices, including CPUs and GPUs. Nevertheless, in the proposed tool, the usage of CPUs as accelerating devices alongside with GPUs would be detrimental to performance since it would contend with the usage of CPU by the threads controlling the memory transfers to and from the devices and the execution of the kernels.

The proposed tool has two tasks, the exact search kernel and the optimal alignment, which can be executed in parallel. Hence, in the presence of multiple devices, there are two approaches that can be used: differentiate the devices and execute only one of the tasks in each device, or execute both tasks simultaneously in each device, similar to the single device management. The first approach reduces possible contention between tasks and reduces the memory usage in each device. Since the characteristics of the workload are unknown before hand, and can even vary in the course of the execution, it is impossible to distribute the tasks between the devices in such a way as to achieve perfect load balacing, preventing the full utilisation of the device(s). Consequently, it was chosen to execute both tasks simultaneously in all devices.

3.4.1 Load balancing

In the proposed application, after all queries are processed against a given index block, there is a synchronisation point across all devices. If the queries to be searched are divided equally by all devices, and considering the execution time is determined by the time needed to get the results from the last query searched, the execution time will be determined by the device with the lowest available capability, leaving other devices waiting, as seen in Figure 3.6. Moreover, it is not possible to equally distribute the exact search step since each subset of queries can create different numbers of seeds, and the number of potential regions per query can also vary from in each subset of queries. Therefore, this tool is not suited for static load balacing.

In dynamic load balacing, the tasks are assigned at runtime and can therefore be adapted to the variations of the data. Dynamic load balacing can be divided by centralized management mechanism, where a thread, the master, distributes the work load through the workers, or by a decentralized management mechanism, where all workers cooperate amongst themselves to distribute the work load [8]. The latter is suited for tasks with a dynamic number of tasks, unknown *a priori*, or with fine-grained tasks and a huge amount of slaves, white the former is suited for the remainder of ocasions where dynamic programming would be suited, which is the case of the present tool [10].

Taken the above referred load balacing strategies under consideration, the adopted model has two

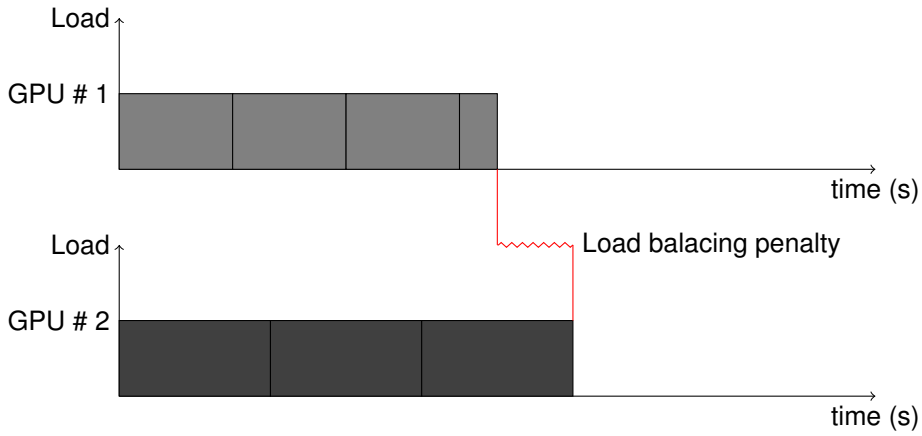


Figure 3.6: Timeline of execution with multiple differing devices

stages, both using a producer-consumer scheme, described in Nogueira [47]. In the first stage, a single producer fetches a constant number of queries from the input file, and stores them in a circular queue, where they are fetched by several consumers, performing the exact search step. After each consumer completes its chunk, it fetches a new chunk of queries. If one device is faster than the others, it will finish the chunks faster and will consequently fetch more chunks.

The results from the exact search (consumers from the previous phase) are placed into another circular queue, from where the consumers, performing the optimal alignment, fetch chunks. Likewise, if a secondary consumer is faster, it will fetch more chunks from the circular queue that the slower consumer, balancing the load between consumers.

3.5 Summary

This chapter describes the proposed tool, starting by exposing what is the approach taken to perform the non-optimal approximate alignment. The following section describes how is parallelism extracted from the algorithm, and discusses various approaches to the management of the devices. The filtration algorithm is also presented. Finally, it is presented the approach taken to parallelise the tool across several heterogeneous devices.

4

Implementation details

Following a high level architecture of the proposed tool, the present section describes the mapping of the architecture to the targeted GPUs. It starts by presenting the data types available for the alignment and by detailing the steps necessary to find all the solutions for DNA. Aditionally, it also describes the implementation of the filtration algorithm is detailed. A new communication mechanism is also proposed to lower the communication costs. Finally, the implementation of the optimal alignment is discussed, by presenting how the intratask and intertask schemes were mapped to the GPU.

4.1 Data representation

As stated in the goals for this tool, the reference text should not have any restrictions in terms of its alphabet. Consequently, the tool should be capable of performing optimal string matching over DNA sequences, amino acids sequences (i.e. Proteins) and generic text. Nevertheless, BowMapCL v1.0 and the proposed tool have a capability to easily augment the types of data it can operate on. This is achieved by creating a new data type by adding it to an *enum*. It is also necessary to add a new map and inverse map, required for the BWT, to the new data type, by changing the appropriate function. Finally, the number of possible characters of the new data type must be added into two functions. This capability was used to introduce a new type of data, extended DNA (DNA_EXT), comprising the complete nucleic acid notation. The default data types of the proposed tool and the input data type range, which

is equivalent to the cardinality of the alphabet A, including the necessary stop character, are presented in table 4.1.

	DNA	DNA_EXT	Proteins	Text
Number of possible characters	5	16	25	128

Table 4.1: Default input data type ranges

The cardinality of the alphabet is important since the size of the data structures used in the exact search and in optimal alignment are proportional the cardinality of the alphabet. In particular, in the data structures pertaining to exact search, the size of the OCC matrix is defined as $cardinality \times textlength$, while the C vector has $cardinality + 1$ elements. In the data structures pertaining to the optimal alignment, the substitution matrix $\delta(q_i, d_j)$ has a size of $cardinality \times cardinality$.

4.1.1 Approximate DNA matching

As stated in section 2.1, DNA is composed of 2 complementary strands, with opposite directions, usually called + (plus strand) and - (minus strand) to distiguish them. DNA sequencers operate on both strands simultaneously, since they are indistinguishable. Without loss of generality, assuming the reference genome is the + strand, only the reads sequenced from the + strand will be able to match the reference directly since only they match the direction of the reference. To match the reads from the - strand, it is necessary recreate the + strand. This is performed by reversing the direction of the read and replacing the bases with their complement, a operation known as reverse complement. The reverse-complemented reads can then be matched to the reference + strand. To convert the base to its complement efficiently, a table composed of the complementary base was created. This table is indexed by the base, enabling the complement to be found in a single memory access. The possible conversions are available in table 4.2. With this design, it was found that the reverse complement can be performed quickly.

In DNA there are two possible sets of bases. The basic set is the set containing A, C, G and T (or U), which are the effective bases. The other set is an extended set, which contains the uncertain bases. For instance, if a base can be either A or G, then it can be coded as base R. Both sets can appear as pattern or as text. If the genome and all the searches can be encoded in the basic set, then the user should select the basic set. If, on the other hand, the queries or the genome require the extended set, then it is necessary to select the extended set when creating the index files.

4.2 Filtration

As previously stated, the filtration algorithm used by BowMapCL is inspired by bowtie2 and shares the same default parameters. In the first step of the filtration algorithm, the seeds are extracted from the query at regular intervals, with overlaps between themselves. Langmead and Salzberg [22] found that for current technologies, a seed length between 20 and 25 performed well, and chose a default of seed length of 22, a choice the filtration algorithm of the proposed tool mimicks. Since the queries can vary

Base	Complementary base
A	T
T	A
G	C
C	G
U	A
Y	R
R	Y
S	S
W	W
K	M
B	V
D	H
H	D
V	B
N	N

Table 4.2: Complementary DNA bases

significantly in length, the authors of bowtie2 report that it is advantageous to set the interval length to a sublinear function of the read length. The default function used in bowtie2, also used in the current tool, is to set the interval length between consecutive seeds $I(x)$ to $I(x) = \max(1, \lfloor 1 + 1.15 \times \sqrt{x} \rfloor)$, where x is the length of the query.

The result from the exact search from each seed is a range of transformed positions, each corresponding to a position in the original text. The number of results have a great variation, with some seeds not present in the text, some are present only a few times, and some seeds generate ranges with thousands of results. These latter seeds are not very selective, thus are not very interesting to analyse. Moreover, the conversion procedure from transformed positions to text positions is time consuming.

Consequently, in the second step of the filtration, for every read bowtie2 selects, randomly and up to a configurable quantity, positions from the set of all positions from every seed from a given read, which are then converted indicating a section of reference to search. This selection is biased towards seeds with smaller ranges, since they are more selective.

The proposed tool follows an analogue procedure. For every read, BowMapCL orders the seeds by increasing range, then selects the positions from the seed with the smallest range, until a configurable quantity of search regions is reached. If this quantity is not reached, then the positions from the next seed are selected, until the quantity is reached or until the program runs out of seeds for the current query. The selected positions are converted to effective text positions, which in turn generate search regions around these positions. If two search regions from a read overlap, they are joined and more positions are converted to search regions.

4.3 Inter-thread communication

As described in the architecture, the communication between the different phases is accomplished through circular buffers, to create a First In First Out (FIFO) mechanism. The circular buffer is implemented by creating a queue, shared by the producers and consumers. This queue is divided into several

blocks of equal size. Each of the producers and consumers have a single block, with the same size as the blocks from the shared buffer. When a producer fills their block, the data from the block is copied to a free block in the circular buffer. If a free block is not available, the producer waits until one is available. After the block has been copied, it is signalled through a semaphore that a filled block is available, and the producer can continue operating, filling its private block. Likewise, when a consumer needs more data, it waits until a filled block is available. Then, it copies the data from the block into its private block and signals that a new block is empty, through another semaphore. The consumer can then start to consume the data from its private block. In order to avoid data races between the different threads, a locking scheme was implemented by Nogueira [47]. Hence, the copy operation is locked through a mutex, and only one thread, either producer or consumer, can access the circular buffer. As can be seen in the scheme of Fig. 4.1, only one thread, in this case the producer n, can access the circular buffer. When a producer has finished the copy operation, the control of the buffer can be either be given to another producer or to one of the consumers. The selected consumer will then copy the oldest block available, and so on, until all blocks have been consumed.

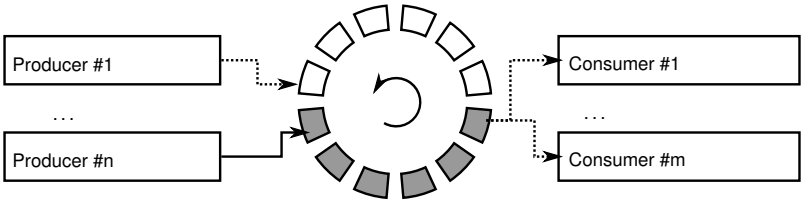


Figure 4.1: BowMapCL v1.0 buffering scheme, using a circular buffer

In the course of the work, it was found that communication between the thread #0, responsible for the reading of the input file, and the filtering threads was taking a significant amount of time due to the duplicated copy of the data, once from the thread #0 to the queue and another from the queue to the filtering thread. To reduce the communication costs between the input thread and the filtering threads, a new communication scheme was devised and implemented.

In this new scheme, shown in Fig. 4.2, the circular buffer, the producers and the consumers store references to the blocks, instead of the blocks. Hence, instead of copying the data to and from the circular buffer, it is only necessary to send the reference of a filled block, already containing all of the data.

When the producer needs a new block, it waits until an empty buffer is available. The empty buffers are stored in a circular buffer, known as the return buffer, shown at the top in the fig. 4.2. After the producer has fetched a reference to an empty buffer, the data is stored into the block, until the block is full or there is no more data. The reference to the block is then sent into the forward circular buffer, shown at the bottom of fig. 4.2, and the producer can fetch another empty block. If a consumer thread can process more data, it fetches the location of a previously filled buffer from the forward circular buffer. The data is then consumed directly from that buffer, obliviating the need for copies. After the consumer thread has consumed all of the data of the block, i.e., the block is empty, its reference is placed onto the return circular queue, returning the empty buffers back to the producers. This scheme allows the proposed communication scheme to operate allocate all the needed memory before hand.

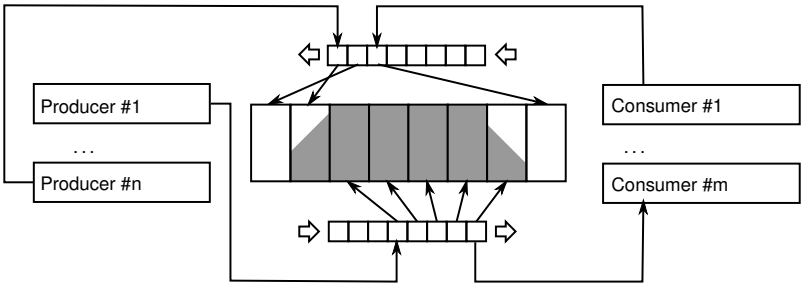


Figure 4.2: Proposed buffering scheme, without requiring copies

As expected, the non-copy scheme managed to reduce the time spent waiting for new data in the producers. However, since it is slightly more complex, it has been used between the thread #0 and the filtering threads, while the old scheme continues to be used between the filtering threads and the exact search threads.

The following section will present in more detail how the exact search consumes its data efficiently.

4.4 Optimal search: Smith-Waterman kernel

The exact search threads, as we can recall from Fig. 3.3, receive data from the filtering threads and send it to the GPU. At the core of the exact search sits the Smith-Waterman kernel, executing in the GPU. After the scores are computed, they are transfered to the host memory, where the best are chosen and returned to the user.

As previously stated, the most computationally expensive procedure of the exact search is the execution of the Smith-Waterman alignment for each area found by filtering. However, it is possible to extract parallelism from the execution of several alignments simultaneously, or within a alignment, as seen in subsection 2.2.3. In order to select the best approach, the two approaches were evaluated. In the next subsection the two approaches will be examined more closely.

4.4.1 Intratask parallelism

Intratask parallelism involves extracting parallelism from the computation of a single alignment. Computation in OpenCL occurs in two levels of granularity. On a lower level, all work items execute the same operation, but on different data, extracting data level parallelism. However, work items are also grouped into work groups. Inside each work group, work items can be related between each other, sharing data and flow control, since they are executed in a single compute unit. However, different work groups do not share information.

Since intratask parallelism requires communication between the vector elements/work items, it can only be applied at the work group level. Consequently, the intratask parallelism mode combines intratask and intertask parallelism by distributing a single aligment to a work group, but performing several different alignments across different work groups.

Inside each work group, the approach taken to extract parallelism from a single alignment is the

striped layout pioneered by Farrar [13], since, as we saw in subsection 2.2.3, this approach offers the best performance. Moreover, it has already been successfully ported to GPU [36].

The computation of Smith-Waterman, using Farrar's algorithm, see algorithm 2.3 on page 19, computes the matrix column by column. Inside each column, the computation occurs in two major steps. The first step computes the intermediate alignment scores, $H_{i,j}$, without taking into account the intra-column dependencies, i.e., the values from F. The values F only contribute to the alignment scores when there is a gap in the reference. The second step, known as the lazy F loop, calculates the values F and corrects the intermediate alignment scores if any of the $H_{i,j}$ values calculated in the first step are not correct. Since Farrar [13] noted that this seldom occurs, this algorithm is faster than other intra-task algorithms.

The original lazy F loop from algorithm 2.3 was replaced by a version proposed by Szalkowski et al. [61]. Algorithm 4.1 has the pseudo-code of the reworked lazy F loop. Unlike the original, the new version has two nested loops with a defined count, giving to opportunity for simplification of the flow control on the GPU.

Algorithm 4.1 Reworked lazy F loop

```
 1: procedure LAZY_LOOP
 2:     for j := 0, 1, . . . , local_size do
 3:         vF := vF ≪ 1
 4:         for k := 0, 1, . . . , segLen do
 5:             vHStore[j] := MAX(vHStore[j], vF)
 6:             if ANYELEMENT(vF > vHStore[j] - vGapOpen) == False then
 7:                 Break lazy loop
 8:             end if
 9:             vF := vF - vGapExtend
10:         end for
11:     end for
12: end procedure
```

In the OpenCL computation model, a work group can be viewed as a virtualized SIMD vector with the number of elements equal to the local work size. There are, however, certain restrictions to this model. In particular, OpenCL 1.2 does not provide any language constructs to easily transport values from one work item to another. It also does not provide mechanisms to make the execution flow dependent on the values from all of the work items. Hence, while the adaptation of the striped approach is fairly direct, since the vector operations are largely independent, the AnyElement function or the vector left shift require the usage of local memory, which is accessible to all work items inside a work group. As we can see, these functions are particularly important to the lazy F loop, since the AnyElement provides an early exit out of the lazy loop, and the vector left shift is needed to correct the values.

The AnyElement function returns true, for all work items, if a given condition is true for any work item. The implementation of the AnyElement used in the present work is shown in listing 4.1. The function has, as its input values, the condition "cond", a local memory address where the values will be stored, the number of work items and, finally, the number of the work item that called the function. When this function is called, every work item inside of the work group enter it, each with its own conditional and work item number (private state), but with a shared work group size work and the same local memory

address. Since it is necessary for all work items to see the same shared state, the condition is stored into the local memory array ”cmp”. A barrier is used to ensure that all work items see the memory in the same state. Each thread then scans the whole array of conditions to find if the condition is true. If any of the values is true, then the result is true. Otherwise, then the result is false. Since it is necessary that all threads exit this function simultaneously, a barrier is used to synchronize all work items.

Listing 4.1: Implementation of AnyElement in OpenCL

```
bool AnyElement(bool cond, local bool * cmp, size_t local_size, size_t tid){
        cmp[tid] = cond;
        barrier(CLK_LOCAL_MEM_FENCE);
        bool decision = false;
        for(size_t k = 0; k < local_size; ++k){
                if(cmp[k] == true){
                        decision = true;
                }
        }
        barrier(CLK_LOCAL_MEM_FENCE);
        return decision;
}
```

The vector shift left is a procedure that moves a given value from the current work item to the work item to its left. For example, if a value is stored in the work item #5, it is moved to the work item #4, whilst the value from the work item #6 is moved to the work item #5. The implementation of the vector left shift is shown in listing 4.2. To move the data between work items, every work item stores the respective value to be shifted in an local memory array. To ensure every work items see the correct value, a barrier is called. Each work item then accesses the value stored by the work item to its right, with the exception of the leftmost work item, which uses 0.

Listing 4.2: Implementation of intra vector left shift in OpenCL

```
shift_aux[tid] = regF;
regF = 0;
mem_fence(CLK_LOCAL_MEM_FENCE);
if(tid > 0){
        regF = shift_aux[tid - 1];
}
mem_fence(CLK_GLOBAL_MEM_FENCE);
```

To compute a new column in intratask parallelism, it is necessary to fetch, for each row, the respective values of the matrices H and E. In order to reduce the number of global memory accesses, it is attractive that the two values are stored together.

The optimal memory size transfer per work item for the global memory for GPUs, as stated in [48, p. 28] and [3, p. 6-35], is 32 bit. Hence, by using only 16 bit for each of the values H and E, both values can be stored together using a single global write. Using 16 bit restricts the maximum alignment score to 65535. Since this sufficient to store the most common alignments of short reads, H and E are stored in a *ushort2* vector data type.

To maximize the performance of the memory system, it is also necessary to coalesce the memory accesses, specially to the combined H and E cells. The figure 4.2 presents a snapshot of the compution of a single column of a matrix with 17 rows using a vector of width 4. There are two obvious options to

store the elements of the column. It is possible to store the rows sequentially, resulting in the memory layout of Fig. 4.3a, where the first row is stored in the first memory position, the second row is stored in the second memory position, and so on. The benefit of this scheme is the simplicity of the memory indexing, since the work item working in the n_{th} row needs to access the n_{th} memory position of the row buffer. However, as we can see in fig. 4.3b, the memory accesses, for instance, in green, are not contiguous, therefore the memory accesses cannot be coalesced.

It is therefore necessary to have rows spaced $segment\_length$ adjacent in memory to coalesce the memory accesses, where $segment\_length = \lceil m/\text{workgroup\_size} \rceil$. Hence, all rows from the first iteration of vector are stored side-by-side, then the second iteration of vector is stored, and so on. Figure 4.3c represents an example of an implementation with the aforementioned memory layout. As a downside, this layout requires a small increase in the memory required to store the buffer.

Row	Memory location
0	0
1	1
2	2
3	3
4	4
5	5
6	6
...	...
12	12
13	13
14	14
15	15
16	16

(a) Linear memory layout

(b) Striped approach

Row	Memory location
0	0
1	4
2	8
3	12
4	16
5	1
6	5
...	...
12	10
13	14
14	18
15	3
16	7

(c) Striped memory layout

Figure 4.3: Example of memory layouts for a matrix with 17 rows and a vector size of 4

4.4.2 Intertask parallelism

On the other hand, in intertask parallelisation each work item performs a complete alignment between a read and reference section. This arrangement, unlike intratask parallelisation, does not have dependencies between each work item, even inside the same work group. However, since there are more matching procedures occurring simultaneously, the memory requirements are higher.

In intertask parallelisation, the matrix can be built column by column (or equivalently, row by row) or anti-diagonal by anti-diagonal, shown in Fig. 4.4. The advantages of the former include the regularity of the memory accesses pattern, since every column has the same amount of rows, unlike the anti-diagonal, since every anti-diagonal can have a different amount of rows. The choice of building row by row or column by column is important for the memory usage when the sizes of the read (n) and the reference (section) (m) are very dissimilar, since the memory usage can be proportional to either n or m, which is not the case here, since we are only interested in a reference section enveloping the read, in which case m and n are of similar size.

Since the substitution matrix has frequent memory accesses to different memory locations, which

(a) Column by column approach (b) Anti-diagonal approach

Figure 4.4: Comparison of intertask approaches

can not be coalesced, and it has a relatively small size, it is an ideal candidate to be loaded onto the local memory. Another possible way to increase performance is the usage of a query profile, introduced in section 2.2.4. The query profile would be indexed to a single query. However, since each query is aligned against a small number of regions, the cost of the creation of the query profile would not be recovered.

In order to reduce memory accesses to the global memory, the kernel implements a tiling approach similiar to CUDASW++ v2.0. The matrix is computed in stripes the same length as the reference section (see Figure 4.5), and with a width adjustable at compile time in the kernel, which means it can be adjustable at run time due to the architecture of OpenCL, where the kernels are compiled at runtime. Inside this stripe, the matrix is filled row-wise until the complete stripe is computed. After a stripe is complete, the next stripe is computed, start at the first row. The intermediate values of H, F and E between the rows are stored in registers. At the start of each row, the value of H and E from the previoues stripe is fetched from the global memory; at the end of the row, the current values of H and E are stored in the global memory.

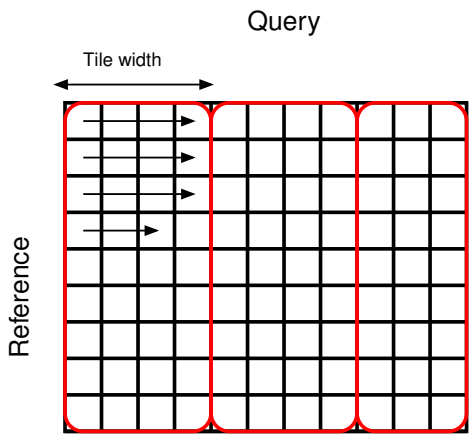


Figure 4.5: Tiling approach

The characters of the reference and of the query are stored in the *char* data type. To reduce the number of accesses to the global memory, the characters from the query are packed into groups of 4, in a *char4* vector data type, in the CPU. This allows fetching 4 characters from the global memory at once, which not only reduces the number of memory accesses as it also increases the size of the memory transaction to 32 bits per work item. A similar packing scheme was tried from the characters for the

reference sections but it was found that performance was not improved.

The reference sections are arrange in such a way that the values from a same row and adjacent threads are in adjacent memory positions. Similarly, the packed queries from a same column and adjacent threads are in adjacent memory positions. This allows memory accesses from reference sections and queries to be coalesced, further increasing the memory throughput.

Similarly to intratask parallelism, the computation of a new row requires fetching the respective values of the matrices H and E. In order to reduce the number of global memory accesses, the two values are stored together, into a *ushort2* vector data type, each with a bit width of 16, with a maximum alignment score of 65535.

4.5 Summary

This chapter describes the most significant implementation details of the proposed tool in order to perform the optimal alignment in heterogeneous computing platforms. In particular, it provides an overview of the techniques used to ensure that the communication costs in the host side are minored. It also provides an overview of the filtration algorithm. Finally, the enhacements to the data structures used by the optimal kernel are presented.

5

Experimental results

5.1 Testing framework

In order to assess the performance of the proposed tool, a series of experiments were devised. The present chapter is divided into two main sections: the evaluation of the optimal alignment step, and the evaluation of the complete non-optimal alignment tool, composed of exact search, filtration and optimal alignment.

Since the proposed tool operates on several different types of data, the evaluation of the tool requires several different datasets. For DNA testing, we chose as a representative reference dataset the human genome GRCh37.75 [21], with an approximate size of 3 GB, since it represents a widely used reference against which reads are aligned [1]. Furthermore, its size represents a challenge to sequence alignment. Several different sequence read files were selected, varying in the average length of the reads and in the number of reads (which are also known as spots). These files are summarised in table 5.1 and can be accessed through the NCBI Short Read Archive [24].

Accession reference	Length of reads	Number of reads
SRR001115	47	10M
SRR3317506	51	26M
SRR211279.1	100	25M
ERR1344794	302	35M

Table 5.1: Considered sequenced reads

The evaluation of the optimal alignment step was conducted using protein data, since the existing implementations to calculate the score using a gaped SW model can only perform optimal alignment of proteins.

The tests were conducted in a platform with quad core Intel Core i7-4770K operating at 3.5 GHz with 32 GB of RAM, and 2 GPUs, namely: an Nvidia GeForce GTX 780 Ti GPU with 3 GB of graphics RAM and a GeForce GTX 660 Ti GPU with 2 GB of graphics RAM. The tests were conducted using only Nvidia GeForce GTX 780 Ti, unless noted otherwise.

The quantitative evaluation of the proposed tool was performed against several state of the art tools. The optimal alignment step was compared against CUDASW++ 2.0 [36] since this tool is the fastest tool performing score-only gaped SW using solely GPUs, but it restricted to performing alignment in proteins. For the evaluation of the complete proposed tool, the state of the art tools chosen for the evaluation are the CPU-based bowtie2 and the GPU-based SOAP3-dp. Such as in the herein proposed tool, these programs use filtration to generate results which are subsequently processed through optimal alignment. Moreover, both tools perform the exact search inside filtration using the BWT and FM-index. These two tools are also among the fastest alignment tools, for CPU and GPU, respectively.

5.2 Optimal alignment step

In this section the performance of the optimal alignment step will be evaluated. Accordingly, the section will start by analysing the proposed intratask parallelism, then the proposed intertask parallelism. Finally, the two modes are compared against the state of the art optimal alignment tools. As previously stated, since CUDASW++ 2.0 [36] can only perform optimal alignment for proteins, the tests in the following section were conducted by aligning only proteins. The database of proteins used is the simulated simdb, with 585 MB, available from Liu et al. [36]. This database represents an optimal case for intertask parallelism, since it is composed 200 000 sequences, each with a length of 3000 amino acides, giving an indication of the maximum performance attainable by optimal alignment tools. A set of proteins of various sizes, shown in table 5.2, were aligned against this database. The set of proteins is comprised of proteins with a length varying from 144 amino acides until 5478 amino acides and is also available from Liu et al. [36].

5.2.1 Intratask parallelism

The steps for the optimal alignment algorithm were profiled individually by restricting the execution to one CPU thread assisting the GPU. With this restriction, there is no overlap between computation in the GPU and memory transfers to and from the device, enabling the duration of each step to be easily quantified. In this test the protein P04775 was aligned against the aforementioned simdb. Considering the test was conducted in a NVIDIA GPU, the size of the workgroup was set at the warp size, i.e. 32, to use all streaming processors within a core.

The producer thread took 2.59 s to read all queries from the database file. The optimal alignment

Accession reference	Length of protein
P02232	144
P05013	189
P14942	222
P07327	375
P01008	464
P03435	567
P42357	657
P21177	729
Q38941	850
P27895	1000
P07756	1500
P04775	2005
P19096	2504
P28167	3005
P0C6B8	3564
P20930	4061
P08519	4548
Q7TMA5	4743
P33450	5147
Q9UUKN1	5478

Table 5.2: Considered sequenced reads

kernel represented the most time consuming operation, see Figure 5.1, occupying 229.58 s of the total consumer time, 293.44 s. Since the time for the producer, in charge of input operations, is smaller than the total exact search time, we can see that the overlap allowed by the producer-consumer scheme mitigates the I/O costs.

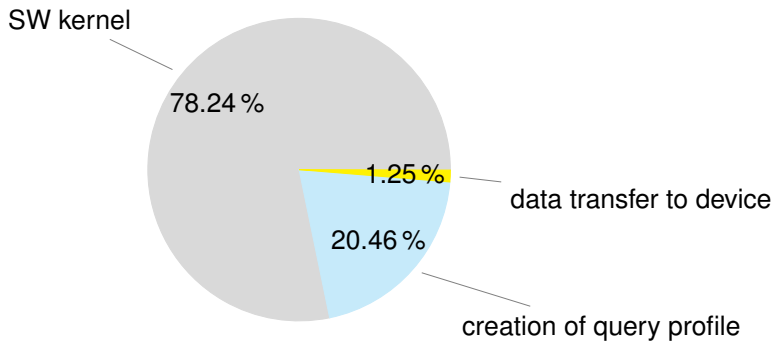


Figure 5.1: Intratask optimal alignment execution time profile of using P04775 against database simdb, scored with BLOSUM62 substitution matrix

5.2.1.A Optimal number of intratask consumer threads

The overlap of the transfer costs with the kernel execution is achieved by creating multiple consumer threads, each with its own set of buffers. By varying the number of consumer threads, one can see how many threads are required to maximize the performance of the GPU, by partially overlapping kernel computation with data transfers and CPU-side computation. As it can be seen in Figure 5.2, two consumer threads can overlap most computation and data transfers, lowering the total execution time. A third consumer thread has a reduced impact of 2.9 % in the execution time, and additional consumer

threads do not have an impact in execution time since there is no more parallelism to be extracted from the GPU.

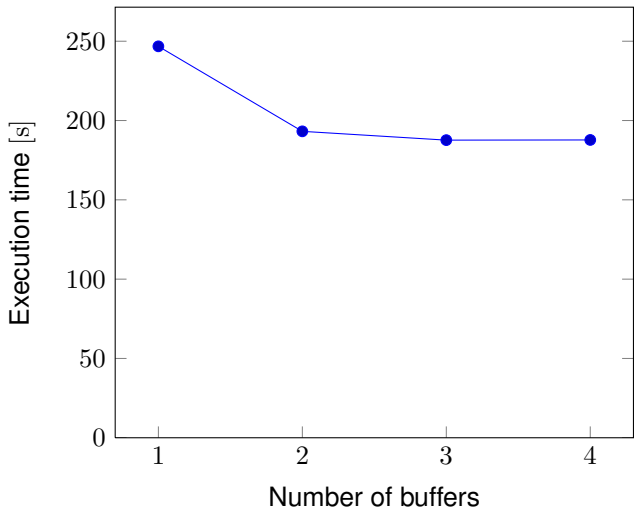


Figure 5.2: Impact of number of buffers in total execution time of intratask using P04775 (2005 aminoacides) against database simdb, scored with BLOSUM62 substitution matrix

The following tests studying the performance of intratask parallelism were conducted using 3 sets of buffers in order to extract the maximum amount of computation from the GPU.

5.2.1.B Global and local work group sizes for intratask

The impact of the global and local work sizes in the execution time was studied by varying the sizes independently. The global work size should be large enough to minimize the overhead of the invocation of the kernel and the transfer of results to and from the device. The number of queries aligned simultaneously was varied, with the results presented in Figure 5.3. In intratask, each work group performs a single alignment, unlike intertask, where each work item performs a single alignment. Consequently, fewer queries need to be enqueued to the GPU to extract parallelism when compared to intertask parallelism.

It was found that starting from 200 queries aligned in one kernel execution, the execution time is approaches the optimal level. By increasing even further the number of queries aligned, another effect comes into play and the execution time increases, albeit very slightly. A possible cause for this effect is that for a great numbers of queries, the computation and communication do not fully overlap, increasing the total execution time. Another possible cause is the increased contention in the management of the GPU, namely the memory elements and cache.

The effects of changing the number of work items per work group, also known as local work size, was also studied, with the results presented in Figure 5.4. As stated previously, in intratask parallelism selecting a local work size is equivalent to selecting the width of a virtualised vector performing the optimal alignment.

Local work sizes inferior to the warp size (32), for the evaluated GPU, present increased execution times. Since the instructions are executed with warp granularity, using these small local work sizes result

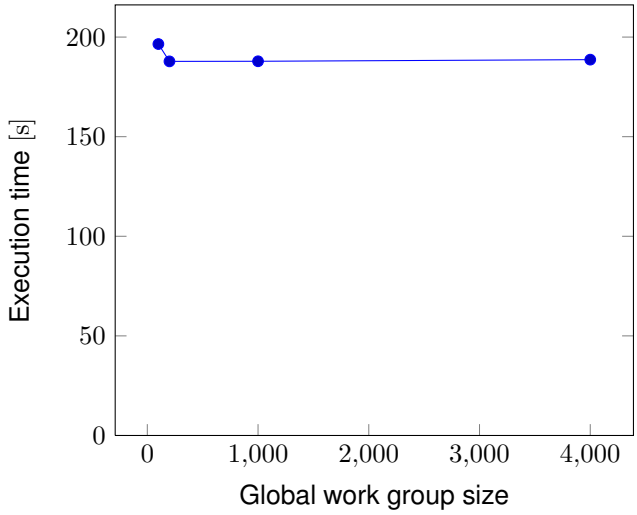


Figure 5.3: Impact of global work size in total execution time of intratask using P04775 (2005 aminoacides) against database simdb, scored with BLOSUM62 substitution matrix

in instructions executed but that do not contribute to result. Hence, as the local work size decreases, the work performed by the GPU per warp stays approximately the same, while the effective work per warp lowers, requiring more warps to be executed, increasing the execution time.

By increasing the local work size even further, in multiples of the warp size, the execution times increases. A possible cause for this effect is the overhead of the intra-vector calculations, namely the AnyElement function (at listing 4.1) and the intra vector left shift (at listing 4.2). Both of these functions are more heavily used in the lazy F loop, which therefore requires the most communication within the work group. Thus, a test was devised, where the lazy F loop is not executed, with the results available in Fig.5.4. As can be seen, with the reduced intra-task communication an increase in local work size leads to a further reduction of the execution times.

One of the causes for the increased cost of intra-task communication as the local work size increases is the AnyElement function, which has a cost that increases at least linearly with the local work size. Moreover, as can be seen from the algorithm, an increase in the size of the virtual vector leads to a reduction of the number of iterations, further increasing the previous effect.

The effects of the query size in the execution time of intratask implementation were also studied. Two sequences of differing size, P04775 and P27895, with a respective length of 2005 and 1000, were aligned against simdb, using a multi-threaded consumer-producer scheme, with 3 threads feeding the GPU. A detailed execution profile was collected, and is available in tables 5.3 and 5.4, respectively. As we can see from the tables, the biggest query is slightly faster when initialisating the device and reading the database into memory. However, this difference is explained by the variability between runs. By comparing biggest overall consumer thread times for each query (giving an approximate total execution time), the bigger sequence requires an execution time 2.000 times greater, but has 2.005 times the length of the smaller sequence. Thus, the total execution time scales linearly with the length of the query.

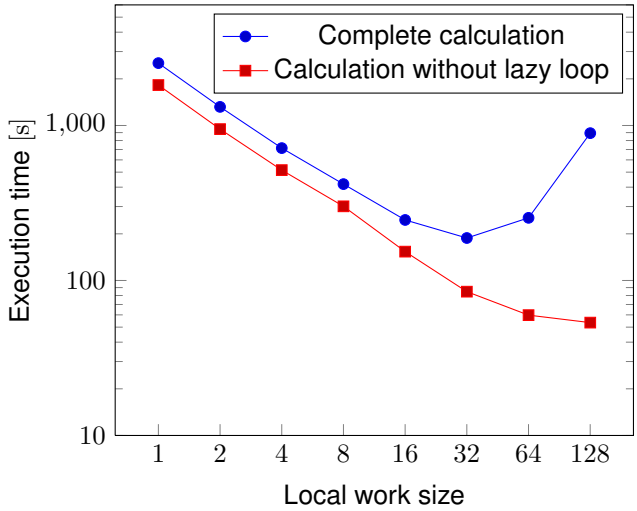


Figure 5.4: Impact of local work size in total execution time of intratask parallelism using P04775 against database simdb, scored with BLOSUM62 substitution matrix

	Time spent per thread (seconds)				
Single thread tasks	Main thread	Producer	Cons. #0	Cons. #1	Cons. #2
Creation of environment	0.8				
Read database from file		4.9			
Semaphore wait for new queries			0.1	0.1	0.1
Arranging the queries			14.9	16.3	16.2
Send data to the device			3.7	3.7	3.7
SW kernel execution			542.8	541.4	541.3
Total	0.8	4.9	561.5	561.6	561.3

Table 5.3: Execution time of each operation in intratask, using P04775 against database simdb, scored with BLOSUM62 substitution matrix

	Time spent per thread (seconds)				
Single thread tasks	Main thread	Producer	Cons. #0	Cons. #1	Cons. #2
Creation of environment	1.0				
Read database from file		5.2			
Semaphore wait for new queries			0.1	0.1	0.1
Arranging the queries			15.3	15.8	14.9
Send data to the device			3.7	3.8	3.8
SW kernel execution			261.5	260.8	261.9
Total	1.0	5.2	280.7	280.6	280.7

Table 5.4: Execution time of each operation in intratask, using P27895 against database simdb, scored with BLOSUM62 substitution matrix

5.2.2 Intertask parallelism

Similarly to the intratask parallelism, the steps for the optimal alignment algorithm were profiled individually by restricting the execution to one CPU thread to assist the GPU. With this restriction enacted, there is no overlap between computation in the GPU and memory transfers to and from the device, enabling the duration of each step to be easily quantified. For this test, the workgroup was set at 128, since, as we will see ahead, it has the lowest execution time. Using the same dataset used in intratask parallelism, the producer thread took 2.40 s to read all queries from the file. The optimal alignment ker-

nel represented the most time consuming operation, occupying 22.59 s of the total optimal alignment thread time, 25.26 s, as can be seen in Figure 5.5. Since the time for the producer, in charge of input operations, is smaller than the total exact search time, we can see that the overlap allowed by the producer-consumer scheme mitigates the I/O costs.

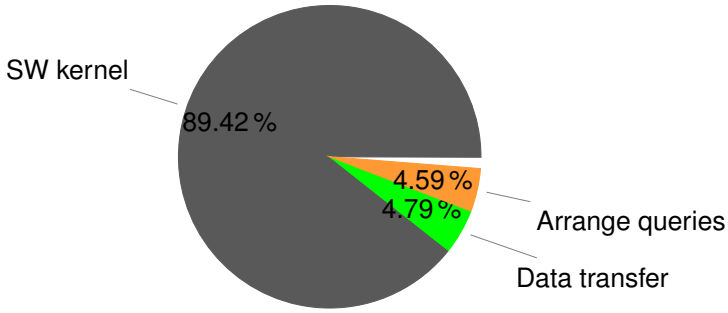


Figure 5.5: Execution time profile of intertask optimal alignment using P04775 against simdb, scored with BLOSUM62 substitution matrix

5.2.2.A Optimal number of intertask consumer threads

The overlap of the transfers to and from the GPU with the kernel execution is achieved by creating multiple consumer threads, each with its own set of buffers. By varying the number of consumer threads, one can see how many threads are required to maximize the performance of the NVIDIA GTX 780 Ti GPU. As it can be seen in Figure 5.6, it is only necessary to execute three consumer threads to minimise the execution time, with additional consumer threads not having an impact in execution time since the GPU is already being fully utilized.

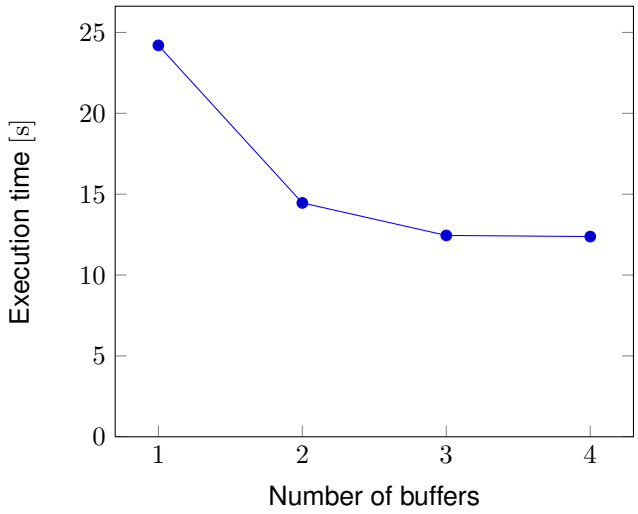


Figure 5.6: Impact of number of buffers in total execution time of intertask using P04775 against database simdb, scored with BLOSUM62 substitution matrix

The following tests in this section were conducted using 3 sets of buffers in order to maximise the overlap between host-side computation and kernel execution, thus minimising execution time.

5.2.2.B Global and local work group sizes for intertask

The impact of the global and local work sizes in the execution time was studied by varying the global and local sizes independently. Global work size should be large enough to minimize the overhead of the invocation of the kernel and the transfer of results to and from the device. By varying the size of the number of queries to be aligned simultaneously, it was found that increasing global work sizes decreases the execution time since the approximately constant overhead can be distributed through more queries, or, alternatively, the complete tool is executed with less kernel executions, resulting in a smaller number of overheads. By analysis of figure 5.7, we can see that for global work sizes over 8192 the changes in execution time are reduced.

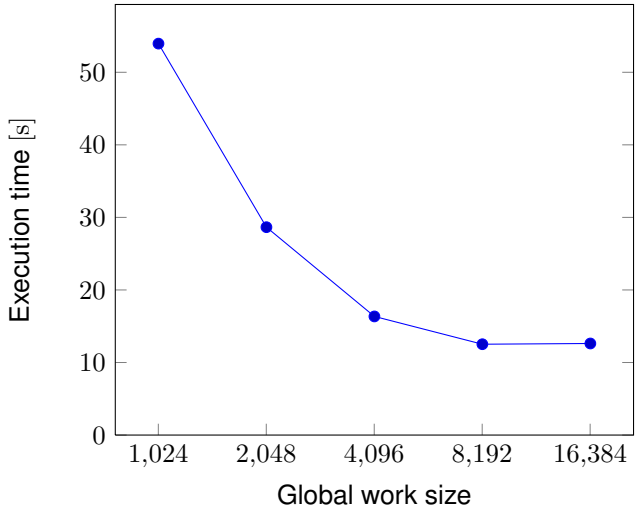


Figure 5.7: Impact of global work size in total execution time of intertask using P04775 against database simdb, scored with BLOSUM62 substitution matrix

In regards to the local work size, using fewer than 32 tasks per work group (the size of a warp) is not recommended, resulting in increased execution time since the streaming processors of the GPU are not fully utilized. The local work size was then increased, always in multiples of the warp size (for NVIDIA GPUs, 32) to maintain the full occupancy of the streaming processors. As we can see in Fig. 5.8, execution time decreases slowly from 32 tasks until 128 tasks, reaching a global minimum, and from that point on the execution time increases slightly.

To study the effects of the query size in the execution time, two sequences of differing size, P27895 and P04775, with a respective length of 1000 and 2005, were aligned against simdb using a multi-threaded consumer-producer scheme, with 3 threads feeding the GPU. A detailed execution profile was collected, and is available in tables 5.5 and 5.6, respectively. It is possible to see that the initialisation of the devices took approximately the same amount of time in both runs, as does sending data for the GPU. However, reading the database was slightly faster for the biggest query. This difference, however, is within the variability between different runs. By a comparison of the biggest of the consumer times (giving a very approximate total execution time), the bigger sequence requires an execution time 1.853 times greater, but has 2.005 times the length of the smaller sequence. Thus, the bigger sequence is aligned 1.07 times faster than was expected from a comparison of the query lengths, despite of the

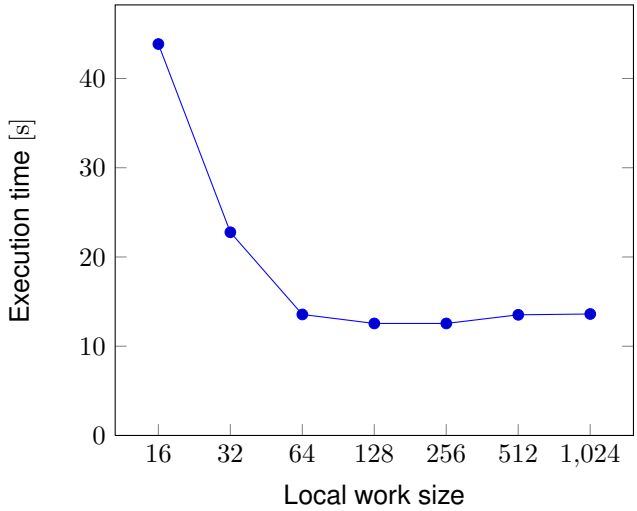


Figure 5.8: Impact of local work size in total execution time of intertask using P04775 against database simdb, scored with BLOSUM62 substitution matrix

worse-than-linear scaling of the time to arrange the queries.

	Time spent per thread (seconds)				
Single thread CPU tasks	Main thread	Producer	Cons. #0	Cons. #1	Cons. #2
Creation of environment	0.1				
Read database from file		6.5			
Semaphore wait for new queries			0.2	0.2	0.2
Arranging the queries			2.3	2.5	2.4
Send data to the device			0.8	0.7	0.7
SW kernel execution			34.1	32.4	33.9
Total	0.1	6.5	37.4	35.8	37.2

Table 5.5: CPU-based execution time of each operation in intertask, using P27895 against database simdb, scored with BLOSUM62 substitution matrix

5.2.3 Performance comparison

The intratask and intertask parallelism schemes were compared, alongside CUDASW++ 2.0, by performing the alignment of the proteins in table 5.2 against the protein database simdb. Since every query and database has a different length, corresponding to different execution times, the results of the different tools were normalized by calculating the number of SW cells updated per second, abbreviated as GCUPS, and calculated as

$$GCUPS = \frac{|Q||D|}{t \times 10^9}$$

where $|Q|$ is the length of the query sequence, $|D|$ is length of the database sequence and t is the total execution time in seconds.

As one can see in Figure 5.9, the performance of the analysed tool and the two parallelisation modes of BowMapCL increases with the size of the query, since bigger queries have more operations to dilute the non-recurring overhead costs such as the initialisation of the devices. Moreover, the data transfer costs are $\mathcal{O}(m+n)$ while the computational cost are $\mathcal{O}(mn)$, leading to a reduction the communication

	Time spent per thread (seconds)				
Single thread CPU tasks	Main thread	Producer	Cons. #0	Cons. #1	Cons. #2
Creation of environment	0.2				
Read database from file		5.1			
Semaphore wait for new queries			0.3	0.3	0.3
Arranging the queries			2.4	2.1	2.5
Send data to the device			0.8	0.8	0.8
SW kernel execution			16.7	16.8	15.9
Total	0.2	5.1	20.2	20.1	19.5

Table 5.6: CPU-based execution time of each operation in intertask, using P04775 against database simdb, scored with BLOSUM62 substitution matrix

overhead per calculated cell with the increase of the size of the queries.

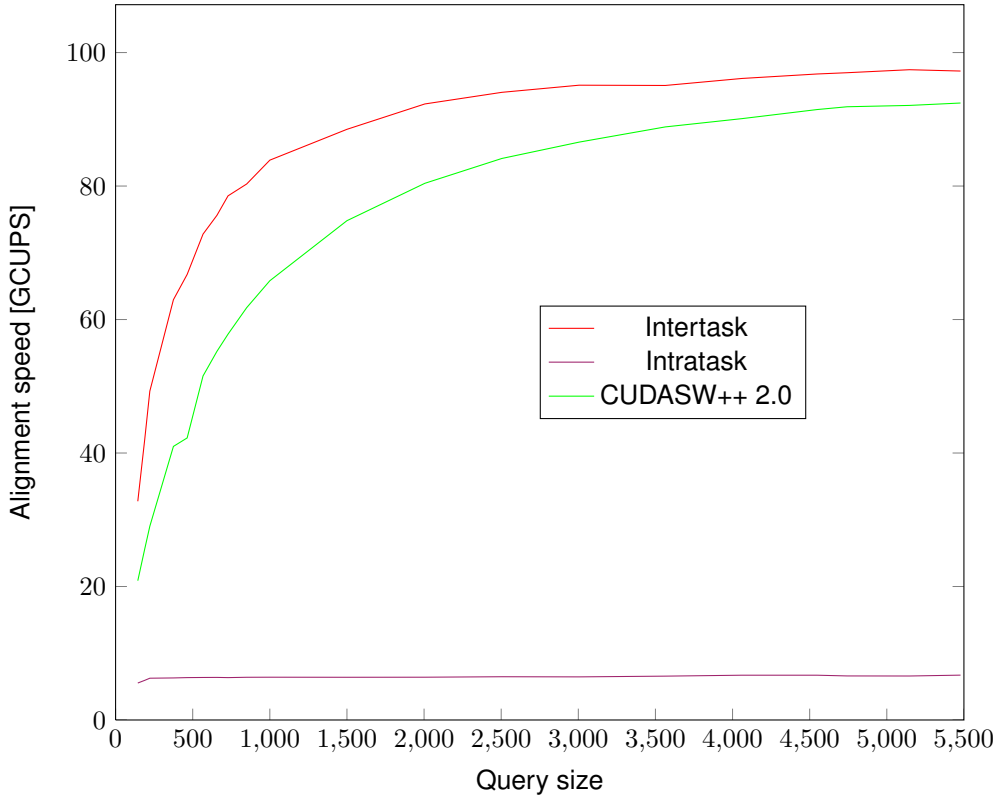


Figure 5.9: Comparison of GCUPS of CUDASW++ 2.0, intratask mode and intertask mode aligning the proteins from table 5.2 against database simdb, scored with BLOSUM62 substitution matrix

Intratask parallelism has the smallest performance of the analysed tools, reaching a maximum of 6.72 GCUPS. Intertask parallelism, in contrast, reaches a maximum of 97.43 GCUPS, and is over 14 times faster than intertask parallelism. For the complete tool, the extraction of parallelism is consequently done with recourse to intertask parallelism since it offers better performance.

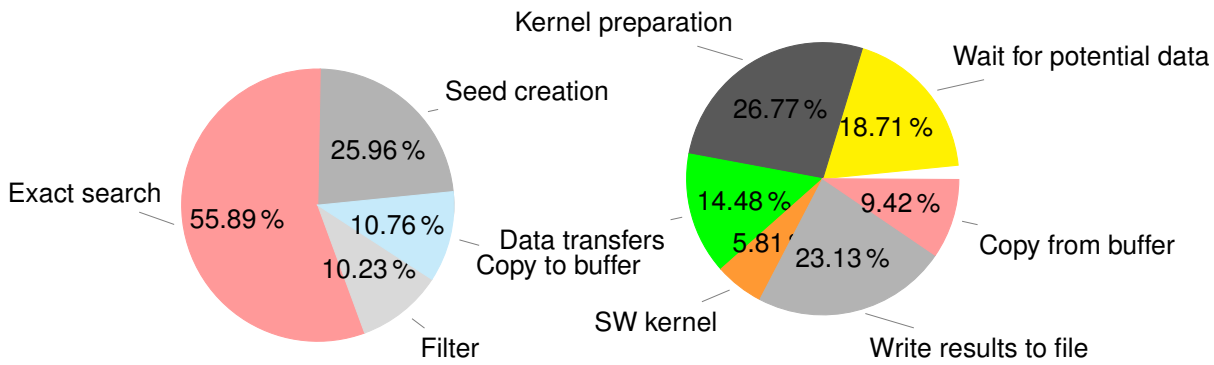
When the intertask parallelism is compared against CUDASW++ 2.0, we find that intertask parallelism can be up to 1.70 times faster, for a query length of 222. The speed advantage is lowered as the size of the queries increases, at which point the intertask is 5 % faster than CUDASW++ 2.0. Coincidentally, it is at this point that CUDASW++ 2.0 is most performant, reaching 92.44 GCUPS.

5.3 Evaluation of the complete tool

5.3.1 Execution profile

The complete tool was profiled by restricting the execution to one CPU thread associated to the GPU performing the filtration and one CPU thread associated to the GPU performing the optimal alignment. With this restriction enacted, there is no overlap between computation in the GPU and memory transfers to and from the device in each phase, enabling the duration of each step of each phase to be easily quantified. The three steps: reading the queries, filtering and optimal alignment, occur simultaneously and take approximately the same time since the downstream phase can only conclude after receiving all the data from the upstream phase. However, the producers, which execute the upstream phase, can not terminate until all the processed data is stored in the circular queues. If the consumers are slower than the producers and the circular queue is already full, the producer must wait for the consumption of data by the consumer, thereby delaying the producers. Thus, the producers will conclude only shortly before the consumers.

The tool was profiled by aligning the DNA file SRR001115 against the the human genome. Due to the size of the available memory, the reference genome was partitioned in 6 blocks. The file SRR001115 is composed of 10 million reads with a length of 47 bases. The producer thread took 112.12 s to read all queries from the file. The filtration phase was concluded in 122.37 s; the distribution of the time the different steps took is shown in Figure 5.10a. The most time consuming step of filtering is the exact search, which is composed of data transfers to the device, execution of the kernel, and retrieval of the result. Regarding the optimal alignment phase, see Figure 5.10b, which took 122.32 s, the operation consuming more time was the preparation of the data to be sent to the device, which includes the selection of the sections from the reference and arranging the data for the GPU buffers, followed by the execution of the kernel itself. It is also possible to see that a significant chunk of time of the optimal phase is spent waiting for the results from the filtering step, indicating that for the current dataset one filtering thread processes data at an inferior rate to one optimal alignment thread.



(a) Filtering execution time profile (b) Optimal alignment execution time profile

Figure 5.10: Execution time profile of the proposed tool

5.3.2 Effect of the number of threads

As we saw in the previous section, the rate at which data is processed by each type of thread differs. Hence, in this section a series of experiments were devised to ascertain the effect of the number of filtering threads and optimal alignment threads in the execution time. One important aspect to note is that the processing rate is dependent on the data and the platform. Nevertheless, this study may present an useful starting point for the adjustment of the runtime parameters.

As previously stated, the adoption of a multiple producer-multiple consumer scheme enables a constant flow of data to be processed by the GPU. However, when using a single CPU thread the time spent processing data in the CPU is superior to the time spent by GPU in the exact search kernel and in the optimal alignment kernel. Therefore, multiple threads are needed to ensure that the GPU is fully exploited.

To discover the impact of the number of filtering threads and optimal alignment threads in the execution, SRR001115 was aligned against the human genome with different combinations of the number of filtering threads and optimal alignment threads.

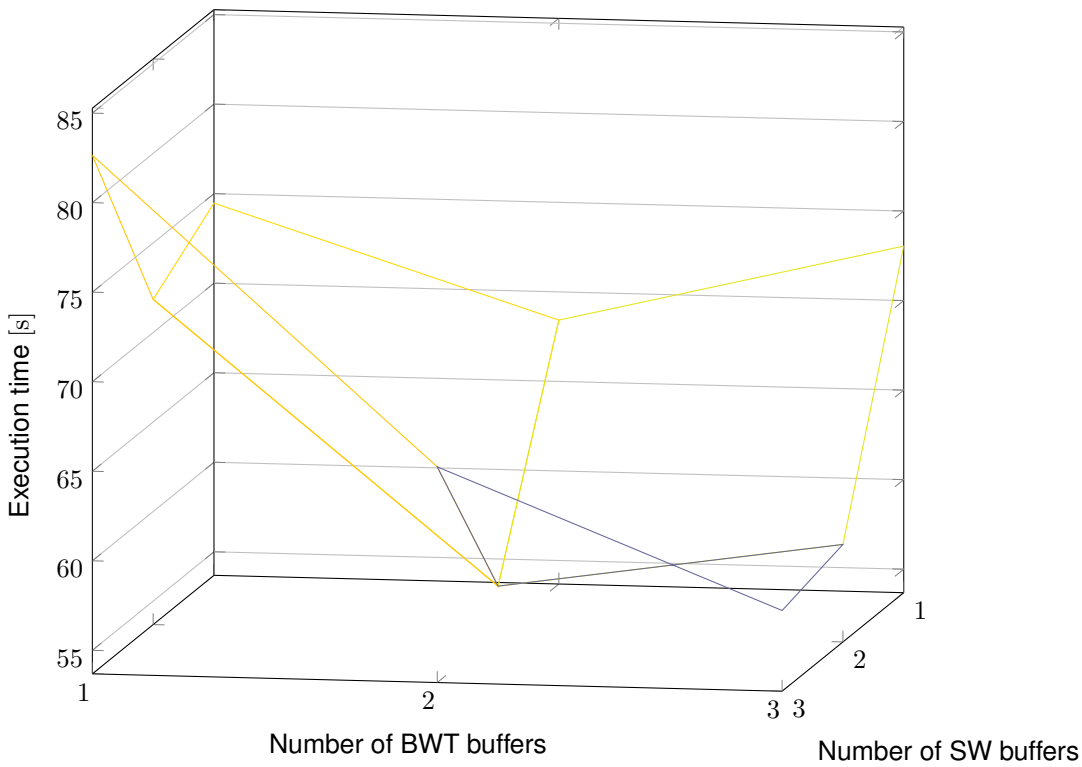


Figure 5.11: Execution time of aligment in relation with the number of buffers, for SRR001115 against the human genome

As can be seen in Figure 5.11, for a certain number of optimal alignment threads, increasing the number of filtering threads increases the performance of the program, until an optimal point. Thus, until this point the execution time is limited by capability of the filtering threads. After this point, adding more filtering threads reduces the performance since the bottleneck becomes the optimal alignment, since the filtering threads cannot produce data at a higher rate higher than the data consumption rate of the optimal alignment threads. Moreover, an increased number of threads leads to increased contention in the

CPU. A similar effect happens if the number of filtering threads is fixed, where an increase in alignment threads increases the performance, until reaching an optimal point where adding more alignment threads reduces performance due to the increased contention.

Another possible effect for the limitation of scaling is the fact that the producer thread cannot generate data at a sufficient rate to maintain 3 filtering threads. To study this effect, the program was profiled again, with 2 filtering threads and 2 optimal alignment threads. As we can see in table 5.7, the filtering threads, downstream from the producer, spent a significant amount of time waiting from data. Thus, further scaling (and a consequent reduction in the execution time) is prevented by the producer thread.

	Time spent per thread (seconds)					
Single thread CPU tasks	Main	Producer	BWT thread #0	BWT #1	SW thread #0	SW #1
Creation of environment	1.5					
Reverse complement		22.3				
Wait for new queries			29.8	29.7		
Seed creation			3.9	3.8		
Exact search			12.7	12.7		
Regions selection			2.3	2.4		
Copy data to buffer			1.5	1.5		
Wait for regions					35.8	35.4
Copy data from buffer					1.4	1.3
Optimal search					10.1	10.4
Writing results					4.2	4.4
Total	1.5	52.4	52.4	52.4	52.4	52.4

Table 5.7: CPU-based execution time of some alignment operations in the first block, alignment of SRR001115 against the human genome

Considering these two types of effects, for the current platform under study the optimal point is 2 filtering threads and 2 optimal alignment threads, which was the number of threads chosen for the following tests.

5.3.3 Global work size optimal values

The number of queries processed per batch of the filtering threads was varied to study the effects on the global execution time, with the results presented in Figure 5.12. The best performance is achieved by setting a number of queries large enough to minimise the effects of the overhead of the data transfer and kernel execution, which can be achieved with 1000 queries. As we can see in the graph, the impact of the number of the queries is small on the execution time as long as its number is not very small (that is, more than 500). This is explained by the fact that the queries are divided into seeds and the effective number of seeds searched in the GPU is larger than the number of queries, since each query generates several seeds.

Similarly, the number of optimal alignments per batch of the optimal thread was also varied, with the results presented in Figure 5.13. To minimise the overhead of the kernel invocation and the data transfers, it is necessary to align a large number of queries. As we can seen in the Figure 5.13, a number of 2000 queries guarantees a smaller execution time.

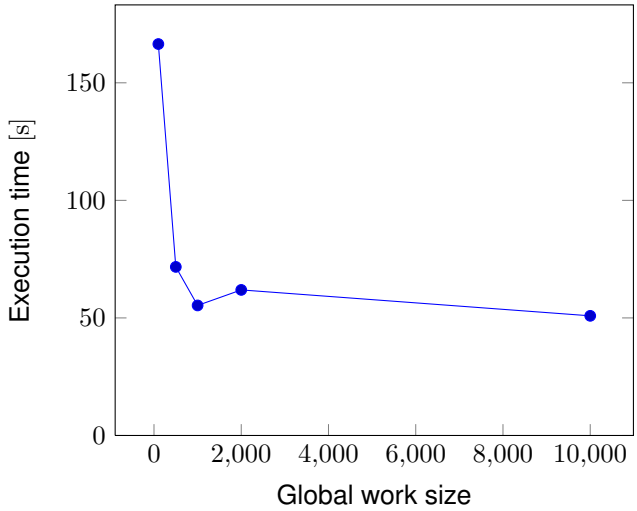


Figure 5.12: Impact of global work size of filtering in the execution time of alignment, alignment of SRR001115 against the human genome

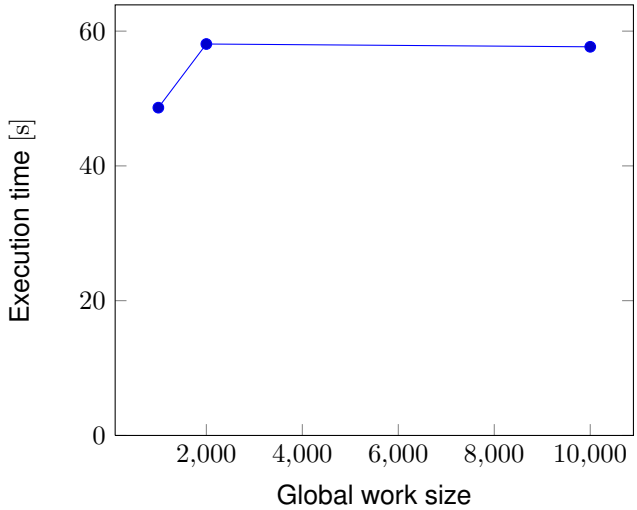


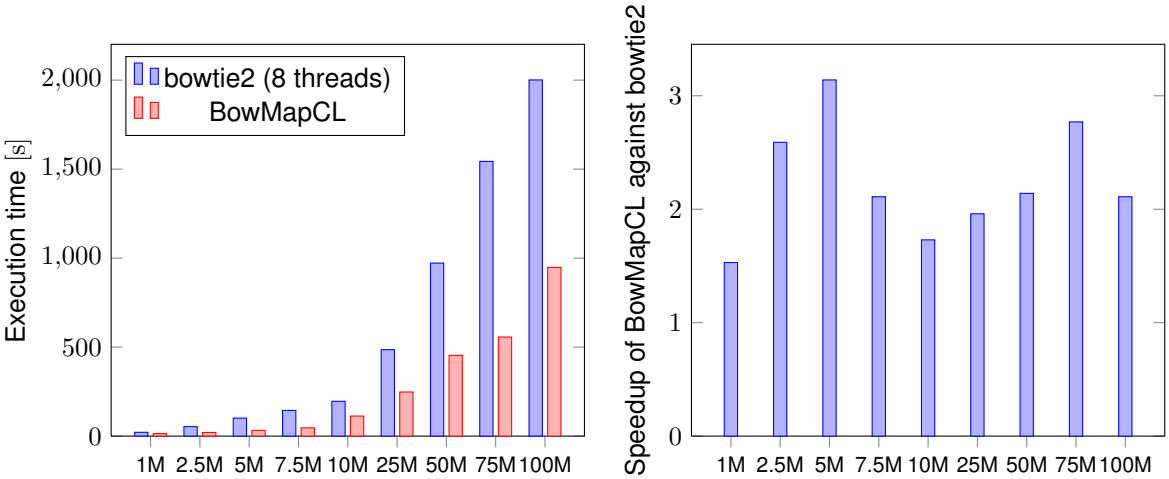
Figure 5.13: Impact of global work size of optimal alignment in the execution time of alignment, alignment of SRR001115 against the human genome

5.3.4 Performance comparison

To quantitatively evaluate the performance of BowMapCL, the tool was compared against several state of the art alignment tools, namely the CPU based bowtie2 and the GPU based SOAP3-dp. The reference genome is the Human genome, and a varying number of queries taken from SRR001115 were aligned against the reference genome. The default parameters of the tools were used, excluding the gap open and gap extend penalty, which was set to -4 and -2, respectively.

As can be observed in Figure 5.14, compared with the CPU-based bowtie2, the proposed tool offers speedups of up to 3 times. For files with a lower number of queries, for instance, 1 million queries, the speed advantage of BowMapCL is lower due to the the initialisation overhead introduced by OpenCL and by the overall GPU computation, such as memory transfers, which have a significant impact on the total execution time.

When compared against SOAP3-dp, a sequence alignment tool which uses GPUs, the proposed tool

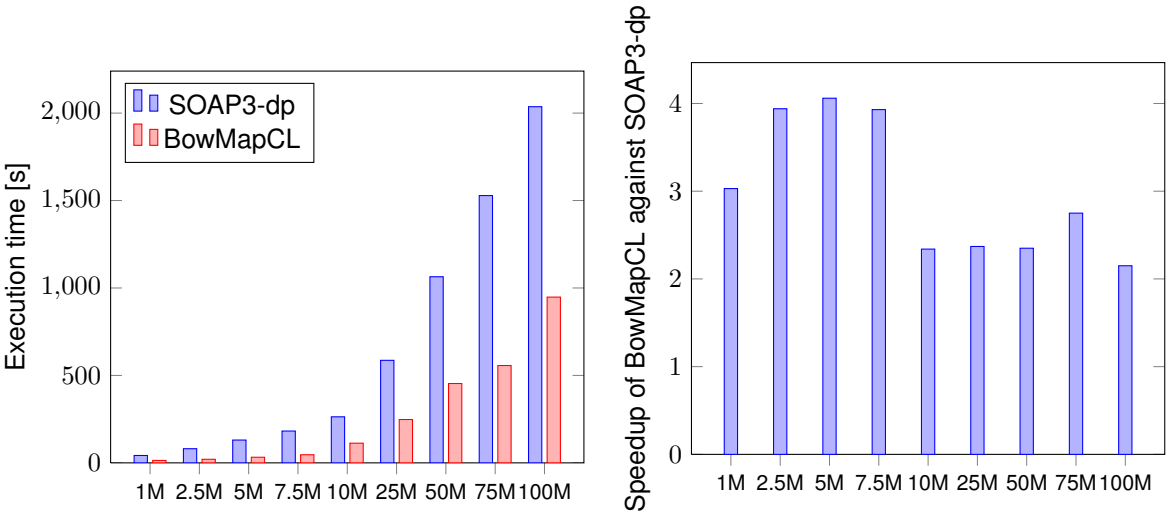


(a) Comparison of execution time between bowtie2 and proposed tool

(b) Speedup of proposed tool in relation to bowtie2

Figure 5.14: Comparison between CPU-based bowtie2 and BowMapCL, aligning a varying the number of reads taken from SRR001115 against the reference genome

is faster is up to 4 times faster for reads files containing fewer than 10 million queries. For files with more than 10 million queries, the proposed tool achieves speedups of 2.



(a) Comparison of execution time between SOAP3-dp and proposed tool

(b) Speedup of proposed tool in relation to SOAP3-dp

Figure 5.15: Comparison between GPU-based SOAP3-dp and BowMapCL, aligning a varying the number of reads taken from SRR001115 against the reference genome

5.3.5 Scalability

It is desirable to have a tool that scales linearly with the number of queries and their length. For a given query length, the filtration step, including the exact search, and the optimal alignment are expected to have a linear time complexity in regard to the number of queries. The exact search kernel execution time only depends on the size of the seeds and their number. For a given query length the size of the

seeds is constant, as is the the number of seeds per read. Hence, the execution time is $\mathcal{O}(k)$, where k is the number of queries. Likewise, if the length of the reads is fixed, the optimal aligment kernel execution time is also directly proportional to the number of queries, since there are $\mathcal{O}(k)$ alignments performed.

In regards to the length of the query, the filtration algorithm uses a sub-linear $\mathcal{O}(\sqrt{n})$ distance between seeds, with respect to the size of the query n. Hence, the number of seeds generated per query in the filtration is also sub-linear, $\mathcal{O}(\sqrt{n})$. Therefore, the number of exact search alignments performed in the filtration step is $\mathcal{O}(k_{BWT}\sqrt{n})$, where k_{BWT} is the number of queries at the exact search, which is equal to the total number of queries. Since the time required to perform the exact search of each seed is linear with size of the seed, which is constant in the proposed filtration algorithm, the total kernel execution time has a time complexity of $\mathcal{O}(k_{BWT}\sqrt{n})$. In regards with the optimal alignment step, the time complexity for each alignment is $\mathcal{O}(mn)$, where m is the size of the reference section and n is the size of a read. However, the number of alignments performed in the optimal search k_{SW} is the number of queries which survived the filtration step, and is data dependent. The size of the refence section m enveloping the read is in general linearly proportional to the size of the read n, albeit larger than the read. Thus, the kernel execution time for optimal aligment is $\mathcal{O}(k_{SW}n^2)$, resulting in an overall complexity of $\mathcal{O}(k_{BWT}\sqrt{n} + k_{SW}n^2)$.

5.3.5.A Number of reads

According to the previous discussion, the execution time is expected to be directly proportional to the number of reads. To evaluate the impact of the number of queries, the file SR001115, with 10M reads, was partitioned to generate files with 1M, 2.5M, 5M and 7.5M of queries. The file was also concatenated against itself to generate files with a 25M, 50M, 75M and 100M reads.

The concatenation was performed so that the number of optimal alignments is also proportional to the number of queries, enabling simultaneous comparison of all stages of the tool. As it can be seen in Figure 5.16, the execution time is approximately linear with the number of queries, as it was expected.

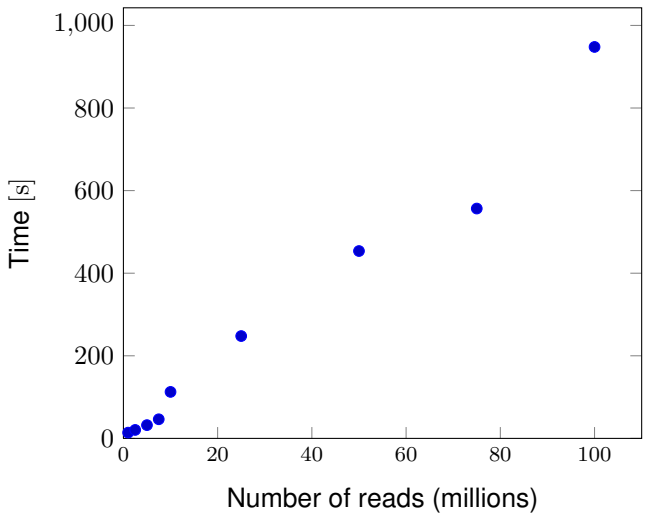


Figure 5.16: Scalability in regards with the number of reads, from 1M to 100M reads of length 47, aligned against the Human Genome

5.3.5.B Length of reads

To evaluate of the scalability of BowMapCL regarding the length of the queries, 3 real short reads file, with a query length of 51, 100 and 302 and containing 25M reads were aligned against the human genome. These files were obtained by trucating the previously mentioned files SRR3317506, SRR211279.1 and ERR1344794, respectively. Since the different files generate a different number of optimal alignments, also represented in Figure 5.17 is the execution time of the exact search and optimal alignment kernels. As we can see, for the query length of 51 the time spent outside the kernels dominates the execution time. In other words, for the studied smallest query length the overhead of initialisation dominates.

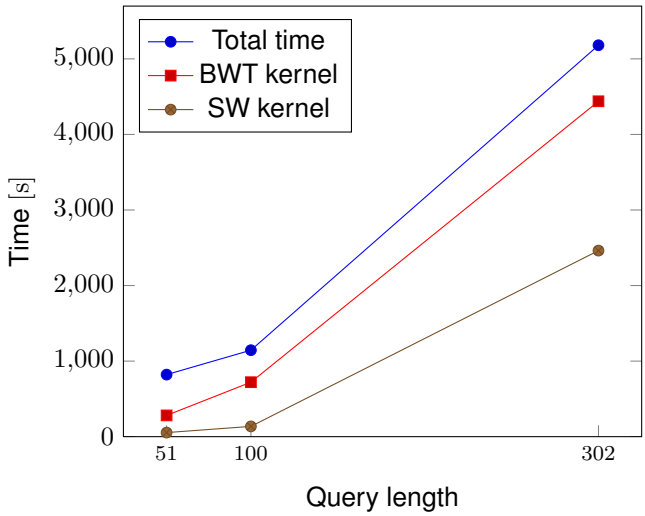


Figure 5.17: Scalability in regards with the length of reads, from a length of 51 to 302, for 25M reads, aligned against the Human Genome

The time spent on the BWT kernel for 100 characters is 2.56 times higher than the time for 51 characters, while the query length is only 1.96 times longer. Since we are expecting an asymptotic complexity of $\mathcal{O}(\sqrt{n})$, we can calculate the constant hidden in the complexity as $2.56/\sqrt{1.96} = 1.83$. Applying this constant to the scalability between 51 and 302 characters, we are expecting that the exact search of the seeds generated by queries with 302 characters to take $1.83 \times \sqrt{302/51} = 4.45$ times longer than the exact search for the case of the queries with 51 characters, while in reality is 15.77 times higher.

This discrepancy is explained by two factors. On one hand, due to nonlinearity of the seeding function, the number of seeds created, hence alignments performed, grows faster than the expected $\mathcal{O}(\sqrt{n})$, as we can see in table 5.8. For 100 characters the number of seeds created is 25 % higher than the expected, while for 302 characters the number of seeds in 53 % higher than the expected. The biggest factor, however, is the kernel itself. As the number of seeds increases, the exact search time per seed increases, in spite of the size of seed being maintained.

In terms of the optimal alignment kernel, the analysis is further complicated by the fact that the optimal aligment is depedent on the size of the regions and the number of regions is dependent on the filtering algorithm. By normalising the time spent in the optimal alignment kernel with the number of

alignments performed, available in table 5.8, we can see that the alignment for reads with a size of 100 and 302 is 1.87 and 23.01 times greater than the baseline (51 bases), respectively. These numbers are better than the expected execution times 3.84 and 35.06 larger than the baseline, respectively. This is explained by the improvement in GCUPS as the query size grows, seen in Figure 5.9 of section 5.2.3, resulting in larger queries being performed faster than expected simply by looking at their sizes.

Size of the reads	Total number of exact searches	Total number of optimal alignments
51	1.200×10^9	251.31×10^6
100	2.099×10^9	337.98×10^6
302	4.481×10^9	498.20×10^6

Table 5.8: Comparison between number of operations performed between read files with 25 million reads, of size 51, 100 and 302

5.3.5.C Number of GPU devices

For the evaluation of the scalability of the proposed tool in regards to the number of GPUs, an different computing platform was used, with a eight core Intel Core i7-5960X at 3 GHz, 32 GB of RAM, two Nvidia GeForce GTX 980 GPU with 4 GB of graphics RAM. Figure 5.18 presents the variation of the performance with the total number of buffers used, using the same number of threads for the filtering and optimal alignment stages. For this analysis, a group of one filtering thread and an optimal alignment thread is called a buffer. By using more than one set of buffers per GPU, it is possible to overlap multiple concurrent operations, such as memory transfers and kernel computation. Moreover, there are more threads available for the CPU-side computations, increasing the parallelism. With 2 sets of buffers, the speedup achieved with one GPU is 1.63 times.

It is also possible to use multiple GPUs to extract higher performance. Using the same amount of buffers, to normalise the host side computation, it is expected that multiple GPUs have a higher performance due to the reduced contention per GPU, since the number of buffers per GPU is reduced.

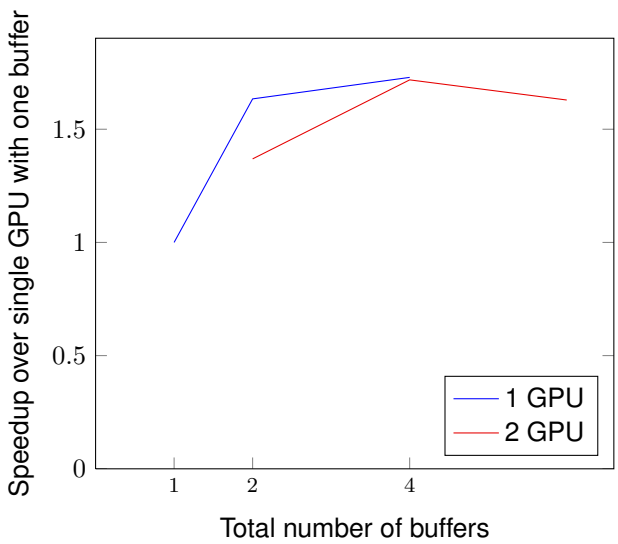
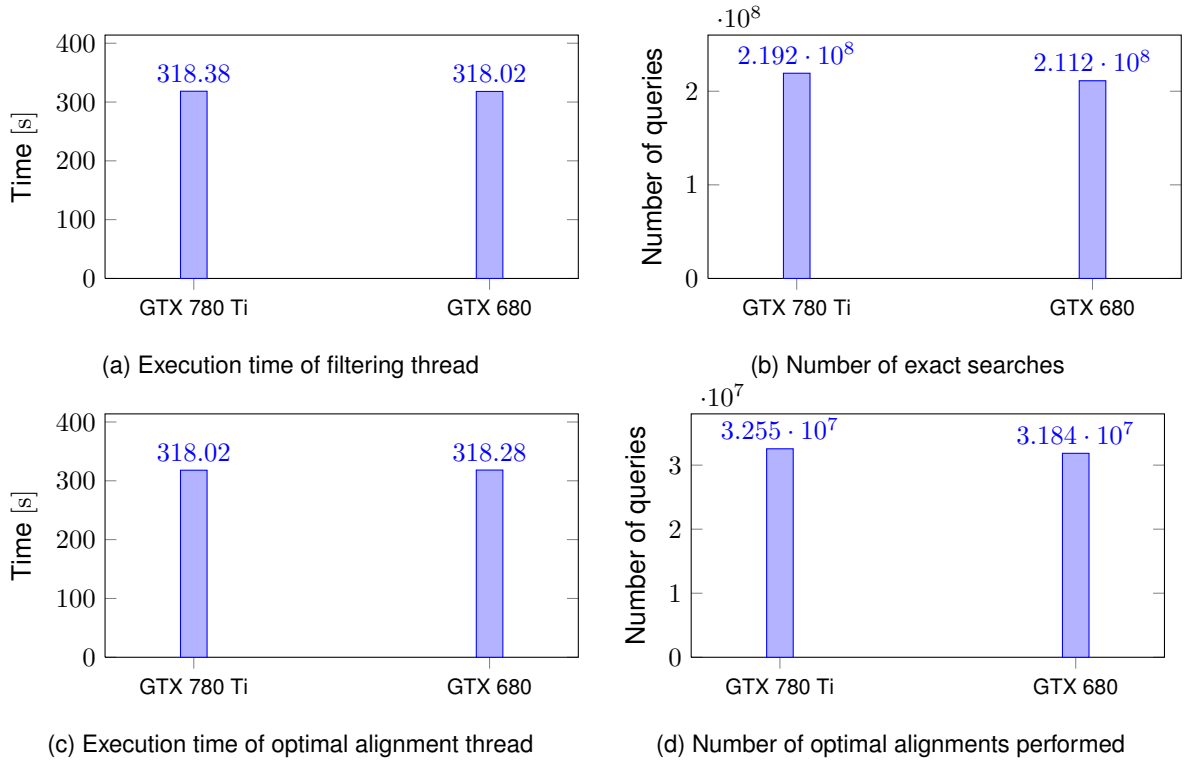


Figure 5.18: Scalability in regards with the number of buffers, in the alignment of SRR3317506

It can be seen that, in both cases, the achieved speedups plateau for a total number of buffer larger than 4. This is because the producer, in charge of reading the query files, can not feed more than 4 sets of buffers. Thus, there is not advantage in running the tool on more than one (fast) GPU, since it is enough to fully saturate the I/O.

5.3.5.D Load balacing

To evaluate the load balacing scheme, the two GPUs in the selected platform, GTX 780 Ti and GT 680, with different processing capabilities, were used to align the SRR001115 file against the human genome, with one thread for optimal alignment and one thread for filtering, per GPU, in a total of 4 threads. As it was expected, due to the different capabilities of the GPUs, they achieve different performance levels in exact search and in optimal alignment, as can be seen in Figure 5.19. To compensate this difference, the tool distributes the workload (chunks of queries) unevenly in order to minimise the overall processing time. Moreover, the final chunks of queries are partitioned into smaller chunks, reducing even further the assimetries between execution times between devices, managing a difference of 220 ms for the filtering threads and 260 ms for the optimal alignment threads.



(a) Execution time of filtering thread

(b) Number of exact searches

(c) Execution time of optimal alignment thread

(d) Number of optimal alignments performed

Figure 5.19: Load balacing evaluation of a platform with two GPUs, matching SRR001115 against the human genome

It is interesting to note that despite the GTX 780 Ti being 1.07 times faster per exact search than the GTX 680, it only performs 1.04 times more exact searches than the weaker GPU (see Figure 5.19b). This happens because the overall filtering time is dominated by the CPU computation, and the GPU execution time has a small impact on the total time of filtering for a chunk of reads.

Similarly, in optimal alignment the GTX 780 Ti is 1.71 faster, but performs only 1.02 times more

alignments, since, once again, the GPU computation is a small part of optimal alignment.

5.3.6 Alignment sensitivity

To quantitatively assess the sensitivity of the alignments of the proposed tool, several real datasets were aligned against the human genome. The Figure 5.20 shows the percentage of reads from each file sucessfully aligned against the genome for all the evaluated tools. The execution times of the tools are presented in table 5.9. The proposed tool has, on average, a sensitivity, i.e., percentage of reads sucessfully aligned with the reference, inferior to the bowtie2 and SOAP3-dp, with an sensitivity of 79.7 %, whereas bowtie2's sensitivity is 85.4 % and SOAP3-dp is 91.2 %. The existent differences are due to the differences between the different filtration algorithms. Despite BowMapCL's inspiration on bowtie2, bowtie2 has another mechanism that improves alignment sensitivity and quality, namely re-seeding. In re-seeding, when the seeds from a query do not generate suitable regions or the regions have low quality, the queries are divided into seeds, but using different starting positions as the original seeding procedure. This way, it is possible to generate new regions (and new aligments) that BowMapCL would miss even if the starting regions on both programs would be the same. The increased difference between BowMapCL and SOAP3-dp indicates a higher difference in the filtering procedure. An important difference, but not the only one, is that SOAP3-dp allows mismatches even in filtering, resulting in an increased number of suitable regions.

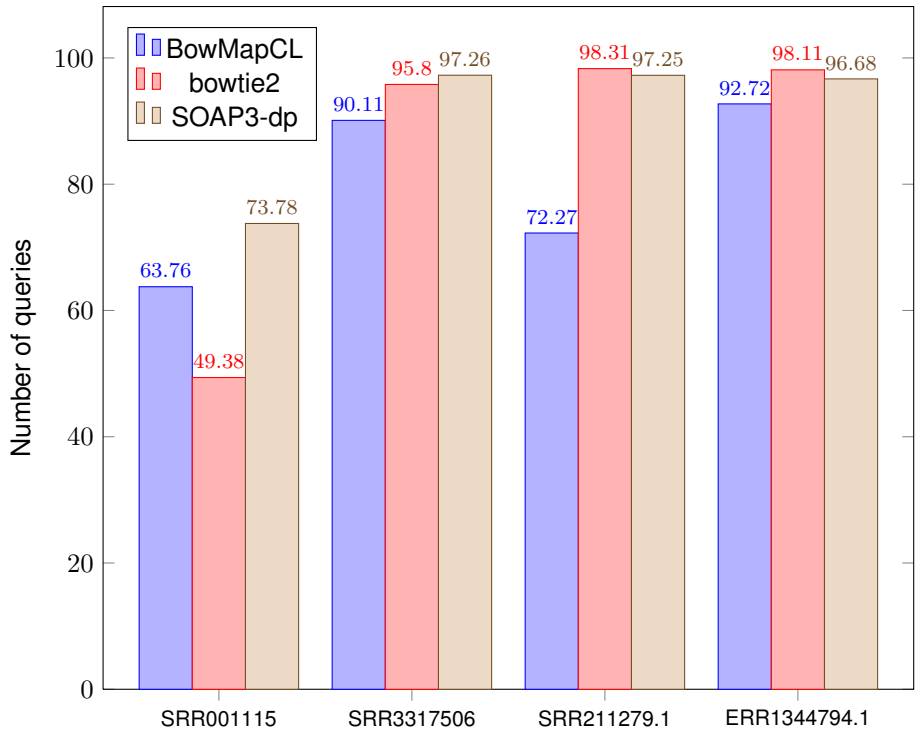


Figure 5.20: Alignment sensitivity comparison

In terms of execution times, BowMapCL is faster than bowtie2 for all the evaluated datasets, with a maximum speedup of 4.00 times, with both tools presenting an marked increase in execution times with the increase of the read length. In contrast, SOAP3-dp has a small variation with the size of the read

length, and BowMapCL is only competitive in read files composed of reads with a length inferior to 100.

Accession reference	BowMapCL	bowtie2	SOAP3-dp
SRR001115	48.79	195.325	264.299
SRR3317506	338.42	482.031	230.182
SRR211279.1	488.01	1414.65	181.353
ERR1344794.1	793.36	2885.727	249.036

Table 5.9: Comparison of execution times for multiple read files, in seconds

5.3.7 Alignment quality

It is also desirable, for an alignment tool, that the generated alignments are in effect correspondent to the best positions of the text. For real DNA data it is not possible to obtain the effective positions of each read in the text. Thus, to quantitively examine the aligment quality of the proposed too, 3 datasets of reads from the human genome were generated with recourse to wgsim [27], with mutation rates of 1 %, 5 % and 10 %. In order to evaluate the impact of indels, a fourth dataset was created, with a mutation rate of 1 %, but with an indel fraction of 30 % and a indel extension probability of 50 %, increased over the default values of 15 % and 30 %, respectively. The datasets are composed of 100 000 reads with 100 bases in length. When compared against state of the art alignment tools, BowMapCL showed comparable sensitive figures for error rates inferior to 10 %, but the quality of the results is inferior to bowtie2 and SOAP3-dp, as can be seen in Figure 5.21.

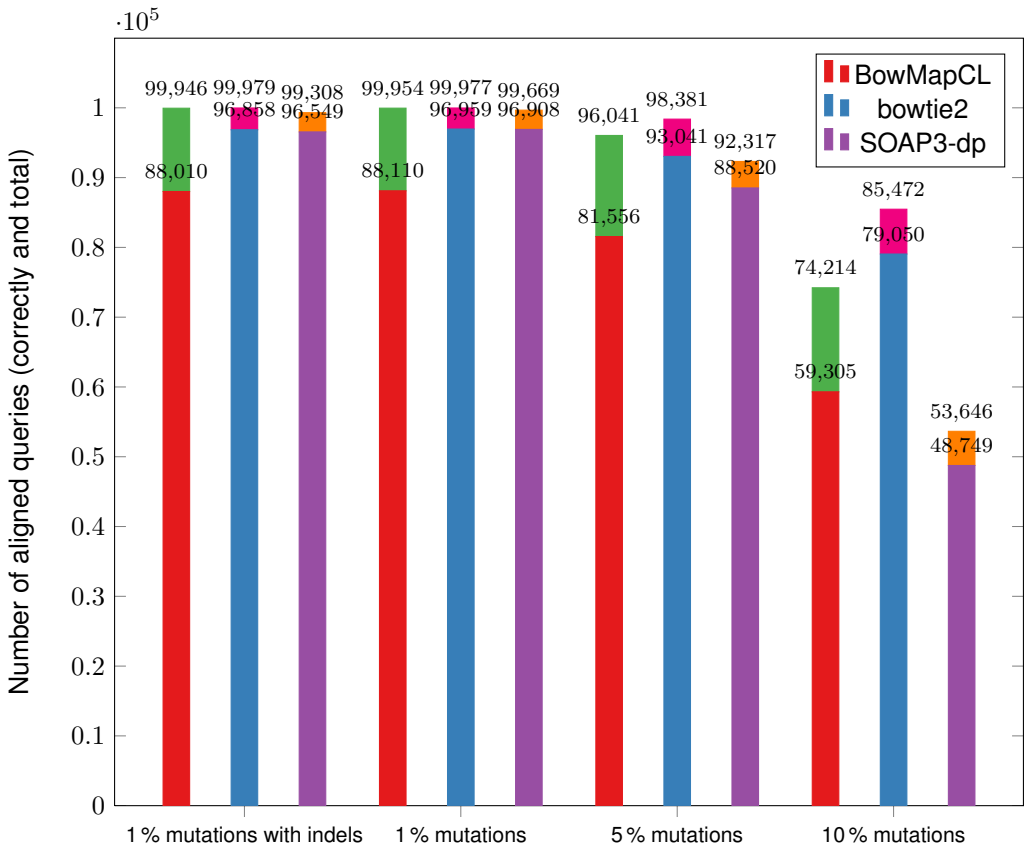


Figure 5.21: Alignment quality comparison for 100 000 simulated reads of length 100

5.3.8 Filtering parameters

As previously stated, the choice of the heuristic parameters in the filtering stage, namely the size of the seeds, the distance between each other and the maximum number of areas searched by the optimal alignment per read, determine the efficiency and efficacy of the DNA alignment tools. In this subsection, the impact of size of the seeds and maximum number of searchable areas will be analysed, using the SRR211279.1 file, composed of 25M reads with a length of 100.

5.3.8.A Length of extracted seeds

The seed length is an important factor in the sensitivity and in execution time of the proposed tool. Smaller seed lengths are expected to have a smaller exact search time in the kernel, since each seed is smaller and the number of seeds extracted from each read is approximately constant. However, smaller seeds have more random occurrences throughout the text, generating more areas where the optimal alignment is performed but may not necessarily generate correct alignments.

Figure 5.22 presents the effects on the sensitivity and in execution times of the variation of seed lengths in the alignment of a file with 25M reads of length 100. For seed lengths inferior to 15, the increase in seed length increases slightly the execution time, due to the increase in the exact search kernel. Since bigger seeds also have a bigger probability to generate a good alignment, the sensitivity is also increased. From a length of 15 until 22, the increase in seed length visibly reduces the execution times since the filtering generates less areas to be searched in optimal alignment, while also reducing the number of optimal alignments. The variation of seed length from 22 to 51 has a much smaller impact on the execution times, which suffer a slight reduction due to the increased selection in filtering, while also generating more alignments. Moreover, the number of generated seeds is also reduced. Starting from length 51, the sensitivity is decreased since the exact search involves searching increasing seeds without any mutation in the reference. Lastly, it is possible to observe that for this file 76.783 % of the reads are found exactly in the text, by setting a seed length equal to the length of the reads.

5.3.8.B Number of areas

As it was referred before, the maximum number of areas selected by the filtering stage per read is also determines the qualitative aspects of the application. An increase in the maximum number of areas increases the sensitivity, since more areas are searched in the optimal alignment stage, at the cost of an increase in computation time.

As can be seen in Figure 5.23, increasing the number of areas increases the sensitivity, but with diminishing returns, since going from 20 areas to 100 areas gives an increase of sensitivity of 0.47 %, whereas the execution time increases almost linearly, increasing 4.25 times.

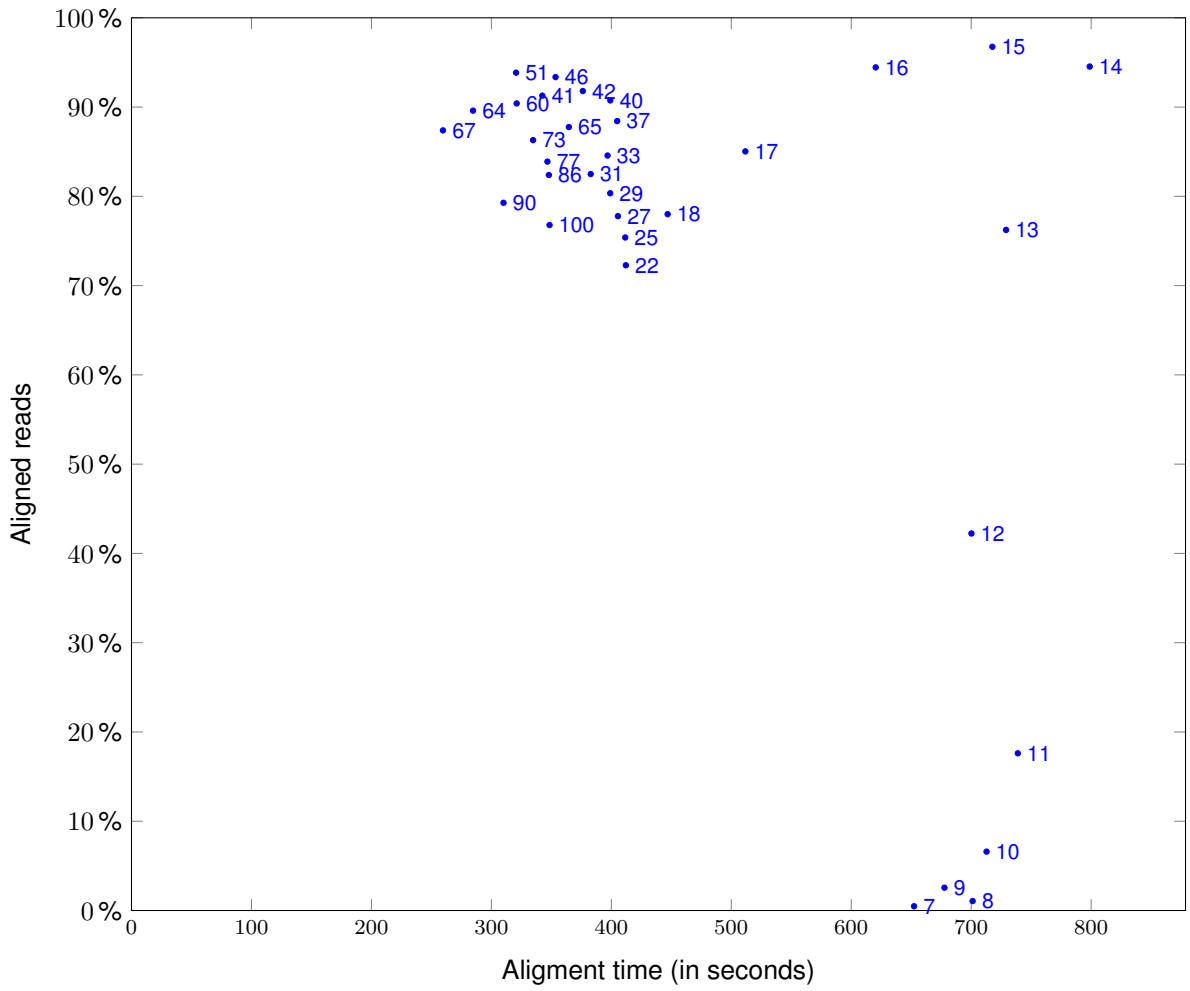


Figure 5.22: Study of sensitivity in relation to the size of seeds of the proposed tool in the alignment of SRR211279.1 (25M reads with length 100) against the Human Genome

5.4 Summary

This chapter presents a comprehensive evaluation of the proposed heterogeneous alignment tool. This evaluation was conducted by evaluating only the optimal alignment portion of the tool and by evaluating the complete tool. In regards to the optimal alignment, it was found that the performance offered by the intertask parallelism is higher than that of the intratask parallelism, reaching speedups of 14 times higher. In comparison to the CUDASW++ 2.0 tool [36], the intertask approach is always faster. The biggest speed advantage occurs for the queries with a smaller size (around 500 characters long), where intertask can be up to 1.7 times faster. For longer queries, the speed advantage of intertask reduces gradually until a 5 % advantage.

In relation to the complete tool, BowMapCL was evaluated in regards to the performance improvements and in terms of the quality of the generated alignments, in relation to existing CPU and GPU-based alignment tools. BowMapCL can reach a speedup of 3× when compared to the CPU-based bowtie2. Coincidentally, the proposed tool can also reach speedups of 3× when compared against the GPU-based SOAP3-dp. However, the proposed tool has a lower sensitivity (number of alignments cre-

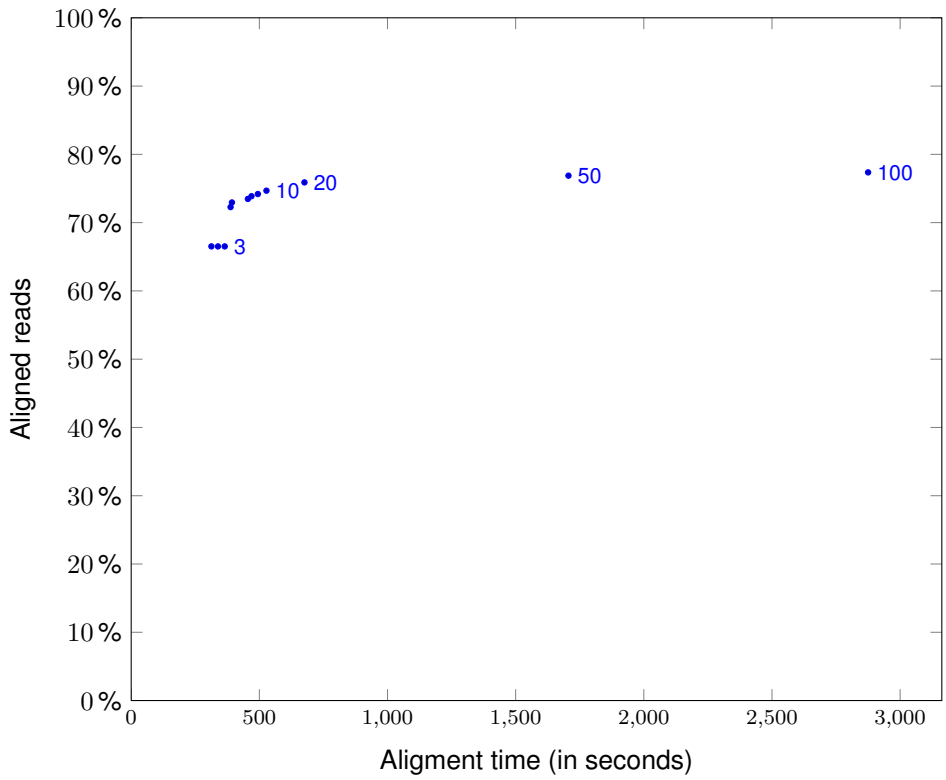


Figure 5.23: Study of sensitivity in relation to the number of optimal alignment areas of the proposed tool in the alignment of SRR211279.1 (25M reads with length 100) against the Human Genome

ated) and a lower quality (number of correct alignments) when compared against either bowtie2 and BowMapCL.

Regarding the usage of multiple GPUs to increase the performance offered, it was found that the performance increase is small, and, therefore, the tool is not scalable.

6

Conclusions

The present work proposes a new approximate string matching tool, capable of using heterogeneous multi-device parallelisation. Approximate string matching tools have several applications, such as retrieval of documents or signal processing. Another important application is in bioinformatics, particularly in the alignment of DNA against a genome. Due to the size of data involved, several applications have been created which use efficient algorithms to align the data in a reasonable amount of time. Current state of the art tools use heuristic techniques, where small segments of the pattern are searched using efficient exact string matching algorithms, namely the BWT. The resulting regions can then be aligned using optimal algorithms, such as the Smith-Waterman algorithm. To decrease even further the time required to align genomic data, several GPU-based tools have appeared [32, 39].

Nogueira [47] proposed a new tool, BowMapCL 1.0, capable of performing exact string using multi-device heterogeneous systems. In addition to be faster than existing CPU-based and GPU-based exact string matching tools, it can also operate on several different types of data, including DNA, proteins and general text. Moreover, BowMapCL 1.0 is also the first GPU-based tool to utilise OpenCL, which is capable of running in NVIDIA and in AMD hardware.

With these advantages in mind, the present dissertation extends BowMapCL 1.0 to create an approximate string matching tool, called BowMapCL 2.0. In this work, a filtration mechanism was created, which creates seeds, searches them in an exact manner using BowMapCL 1.0 exact search and selects the best regions. The best regions are then search using the optimal Smith-Waterman (SW) algorithm, which was ported to GPU using the OpenCL API.

The extraction of parallelism in the SW can be performed by mapping each alignment to a work item, an approach known as intertask parallelism, or by using several work items to perform a single alignment, known as intratask parallelism. The developed intertask parallelism approach was found to be 14 times faster than the developed intratask approach. When comparing the intertask approach with the state of the art optimal alignment tool, CUDASW++ 2.0, the proposed tool can offer speedups of up to 1.7 times. Even in worst scenario for BowMapCL, large queries, it maintains an advantage of 5 %.

A producer-consumer scheme, proposed and implemented by Nogueira [47], with multiple threads dedicated to each GPU device, allows the overlap of data tranfers to the GPU, CPU computation and I/O operations to and from the disk with computation on the GPU, improving the execution times. Furthermore, the division of the algorithm into two stages, filtering an optimal alignment, operating concurrently on the GPU, enables the exploitation of spatial parallelism in the GPU.

When compared to state of the art tools using similar approaches, the proposed tool provides speedups of up to 3 times when compared to a multi-threaded CPU-based tool. However, it is only competitive against the evaluated GPU-based tool for DNA read lengths inferior to 100 bases. For this case, BowMapCL can be up to 4 times faster than the GPU-based SOAP3-dp. The average alignment quality of the proposed tool is 80 %, 11 % lower than bowtie2. Compared to the GPU-based SOAP3-dp, the alignment quality of BowMapCL is only 2 % lower, since it the proposed tool has a 11 % better quality for alignments with a 10 % mutation rate, while having a relative worse quality for lower mutation rates.

6.1 Future Work

There are several envisioned possibilities to improve the proposed tool in terms of performance, scalability, quality of results and usability, which could be the object of further studies. One of the factors more determinant to the performance characteristics of the proposed tool is the heuristic parameters of filtering, which could be changed to enhance the sensitivity and/or execution times of the proposed tool, with emphasis on bigger patterns, where the proposed tool is less competitive.

Another introduction, which would be a major addition, is, in the exact search step, to allow substitutions, insertions and deletions, thereby creating an inexact search step, which could improve sensitivity and performance. Moreover, the adoption of an inexact approximate search would open the possibility to bypass the optimal alignment for some queries, improving performance.

In terms of usability, adding the capability of performing the alignment for read pairs to the proposed tool would improve the types of data the tool is capable of operating upon, since there are several alignment machines that generate paired reads. The restriction of 8 bit characters could also be lifted, thereby allowing the approximate string matching to operate on text from languages which can not be represented in an 8-bit alphabet.

Other possible improvement over the proposed tool is to calculate the optimal alignment in the accelerating device, in addition to the currently calculated homology score.

Bibliography

[1] G. R. Abecasis, D. Altshuler, A. Auton, L. D. Brooks, R. M. Durbin, et al. A map of human genome variation from population-scale sequencing. Nature, 467(7319):1061–1073, Oct 2010. doi: 10.1038/nature09534.

[2] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic local alignment search tool. J. Mol. Biol., 215(3):403–10, Oct 1990. doi: 10.1016/S0022-2836(05)80360-2.

[3] AMD Inc. AMD Accelerated Parallel Processing OpenCL ™Programming Guide. AMD Inc., November 2013.

[4] R. Apweiler, A. Bairoch, C. H. Wu, W. C. Barker, and B. Boeckmann. UniProt: the Universal Protein knowledgebase. Nucleic Acids Res., 32(Database issue):D115–119, Jan 2004. doi: 10.1093/nar/gku989.

[5] Djamal Belazzougui, Paolo Boldi, Rasmus Pagh, and Sebastiano Vigna. Monotone minimal perfect hashing: Searching a sorted table with o(1) accesses. In Proceedings of the Twentieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA '09, pages 785–794, Philadelphia, PA, USA, 2009. Society for Industrial and Applied Mathematics. doi: 10.1137/1.9781611973068.86. URL http://dl.acm.org/citation.cfm?id=1496770.1496856.

[6] L. Bertram and R. E. Tanzi. Genome-wide association studies in Alzheimer's disease. Hum. Mol. Genet., 18(R2):R137–145, Oct 2009.

[7] M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical report, 1994.

[8] Daniel Cederman and Philippas Tsigas. On dynamic load balancing on graphics processors. In Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware, GH '08, pages 57–64, Aire-la-Ville, Switzerland, Switzerland, 2008. Eurographics Association. ISBN 978-3-905674-09-5. URL http://dl.acm.org/citation.cfm?id=1413957.1413967.

[9] K. M. Chao, R. C. Hardison, and W. Miller. Recent developments in linear-space alignment methods: a survey. J. Comput. Biol., 1(4):271–291, 1994.

[10] David Clarke, Alexey Lastovetsky, and Vladimir Rychkov. Dynamic load balancing of parallel computational iterative routines on highly heterogeneous hpc platforms. Parallel Processing Letters,

21(02):195–217, 2011. doi: 10.1142/S0129626411000163. URL http://www.worldscientific.com/doi/abs/10.1142/S0129626411000163.

[11] G. Cochrane, B. Alako, C. Amid, L. Bower, and A. Cerdeno-Tarraga. Facing growth in the European Nucleotide Archive. Nucleic Acids Res., 41(Database issue):D30–35, Jan 2013. doi: 10.1093/nar/gks1175.

[12] A. L. Delcher, S. Kasif, R. D. Fleischmann, J. Peterson, O. White, and S. L. Salzberg. Alignment of whole genomes. Nucleic Acids Res., 27(11):2369–2376, Jun 1999. doi: 10.1093/nar/27.11.2369.

[13] Michael Farrar. Striped Smith-Waterman speeds database searches six times over other SIMD implementations. Bioinformatics, 23(2):156–61, Jan 2007. doi: 10.1093/bioinformatics/btl582.

[14] P. Ferragina and G. Manzini. Opportunistic data structures with applications. In Foundations of Computer Science, 2000. Proceedings. 41st Annual Symposium on, pages 390–398, 2000. doi: 10.1109/SFCS.2000.892127.

[15] O. Gotoh. An improved algorithm for matching biological sequences. J. Mol. Biol., 162(3):705–8, Dec 1982.

[16] D. S. Hirschberg. A Linear Space Algorithm for Computing Maximal Common Subsequences. Commun. ACM, 18(6):341–343, June 1975. ISSN 0001-0782. doi: 10.1145/360825.360861.

[17] A. D. Johnson and C. J. O'Donnell. An open access database of genome-wide association results. BMC Med. Genet., 10:6, 2009.

[18] M. Korpar and M. Šikic. SW#-GPU-enabled exact alignments on genome scale. Bioinformatics, 29(19):2494–2495, Oct 2013. doi: 10.1093/bioinformatics/btt410.

[19] Stefan Kurtz, Adam Phillippy, Arthur L. Delcher, Michael Smoot, and Martin Shumway. Versatile and open software for comparing large genomes. Genome Biology, 5(2):1–9, 2004. ISSN 1465-6906. doi: 10.1186/gb-2004-5-2-r12.

[20] T W Lam, W K Sung, S L Tam, C K Wong, and S M Yiu. Compressed indexing and local alignment of DNA. Bioinformatics (Oxford, England), 24(6):791–7, Mar 2008. doi: 10.1093/bioinformatics/btn032.

[21] E. S. Lander, L. M. Linton, B. Birren, C. Nusbaum, and M. C. Zody. Initial sequencing and analysis of the human genome. Nature, 409(6822):860–921, Feb 2001. doi: 10.1038/35057062.

[22] Ben Langmead and Steven L. Salzberg. Fast gapped-read alignment with Bowtie 2. Nat. Methods, 9(4):357–9, Apr 2012. doi: 10.1038/nmeth.1923.

[23] Ben Langmead, Cole Trapnell, Mihai Pop, and Steven L. Salzberg. Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. Genome Biol., 10(3):R25, 2009. doi: 10.1186/gb-2009-10-3-r25.

[24] R. Leinonen, H. Sugawara, and M. Shumway. The sequence read archive. Nucleic Acids Res., 39 (Database issue):19–21, Jan 2011. doi: 10.1093/nar/gkq1019.

[25] H. Li and R. Durbin. Fast and accurate long-read alignment with Burrows-Wheeler transform. Bioinformatics, 26(5):589–595, Mar 2010. doi: 10.1093/bioinformatics/btp698.

[26] H. Li, J. Ruan, and R. Durbin. Mapping short DNA sequencing reads and calling variants using mapping quality scores. Genome Res., 18(11):1851–1858, Nov 2008. doi: 10.1101/gr.078212.108.

[27] H. Li, B. Handsaker, A. Wysoker, T. Fennell, and J. Ruan. The Sequence Alignment/Map format and SAMtools. Bioinformatics, 25(16):2078–2079, Aug 2009. doi: 10.1093/bioinformatics/btp352.

[28] R. Li, Y. Li, K. Kristiansen, and J. Wang. SOAP: short oligonucleotide alignment program. Bioinformatics, 24(5):713–714, Mar 2008. doi: 10.1093/bioinformatics/btn025.

[29] 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, 25(15):1966–1967, Aug 2009. doi: 10.1093/bioinformatics/btp336.

[30] 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, 28(6): 878–879, Mar 2012. doi: 10.1093/bioinformatics/bts061.

[31] Weiguo Liu, Bertil Schmidt, Gerrit Voss, Andre Schroder, and Wolfgang Muller-Wittig. Biosequence database scanning on a GPU. In Proceedings 20th IEEE International Parallel Distributed Processing Symposium, pages 8 pp.–, April 2006. doi: 10.1109/IPDPS.2006.1639531.

[32] Yongchao Liu and B. Schmidt. CUSHAW2-GPU: Empowering Faster Gapped Short-Read Alignment Using GPU Computing. Design Test, IEEE, 31(1):31–39, Feb 2014. ISSN 2168-2356. doi: 10.1109/MDAT.2013.2284198.

[33] Yongchao Liu and Bertil Schmidt. Long read alignment based on maximal exact match seeds. Bioinformatics, 28(18):i318–i324, Sep 2012. doi: 10.1093/bioinformatics/bts414.

[34] Yongchao Liu and Bertil Schmidt. GSWABE: Faster GPU-accelerated Sequence Alignment with Optimal Alignment Retrieval for Short DNA Sequences. Concurr. Comput. : Pract. Exper., 27(4): 958–972, March 2015. ISSN 1532-0626. doi: 10.1002/cpe.3371.

[35] Yongchao Liu, Douglas L. Maskell, and Bertil Schmidt. CUDASW++: optimizing Smith-Waterman sequence database searches for CUDA-enabled graphics processing units. BMC research notes, 2:73, 2009. doi: 10.1186/1756-0500-2-73.

[36] Yongchao Liu, Bertil Schmidt, and Douglas L Maskell. CUDASW++2.0: enhanced Smith-Waterman protein database search on CUDA-enabled GPUs based on SIMT and virtualized SIMD abstractions. BMC research notes, 3:93, 2010. doi: 10.1186/1756-0500-3-93.

[37] Yongchao Liu, Bertil Schmidt, and Douglas L Maskell. CUSHAW: a CUDA compatible short read aligner to large genomes based on the Burrows-Wheeler transform. Bioinformatics (Oxford, England), 28(14):1830–7, Jul 2012. doi: 10.1093/bioinformatics/bts276.

[38] 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. doi: 10.1186/1471-2105-14-117.

[39] R. Luo, T. Wong, J. Zhu, C. M. Liu, X. Zhu, E. Wu, L. K. Lee, H. Lin, W. Zhu, D. W. Cheung, H. F. Ting, S. M. Yiu, S. Peng, C. Yu, Y. Li, R. Li, and T. W. Lam. SOAP3-dp: fast, accurate and sensitive GPU-based short read aligner. PLoS ONE, 8(5):e65632, 2013. doi: 10.1371/journal.pone.0065632.

[40] 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. doi: 10.1186/1471-2105-9-S2-S10.

[41] Udi Manber and Gene Myers. Suffix arrays: A new method for on-line string searches. In Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms, SODA '90, pages 319–327, Philadelphia, PA, USA, 1990. Society for Industrial and Applied Mathematics. ISBN 0-89871-251-3. doi: 10.1137/0222058.

[42] E. W. Myers. A sublinear algorithm for approximate keyword searching. Algorithmica, 12(4):345–374, 1994. ISSN 1432-0541. doi: 10.1007/BF01185432.

[43] E. W. Myers and W. Miller. Optimal alignments in linear space. Comput. Appl. Biosci., 4(1):11–17, Mar 1988. doi: 10.1093/bioinformatics/4.1.11.

[44] Gonzalo Navarro. A Guided Tour to Approximate String Matching. ACM Comput. Surv., 33(1): 31–88, March 2001. ISSN 0360-0300. doi: 10.1145/375360.375365.

[45] Gonzalo Navarro, Ricardo Baeza-yates, Erkki Sutinen, and Jorma Tarhio. Indexing Methods for Approximate String Matching. IEEE Data Engineering Bulletin, 24(4):19–27, 2001.

[46] S. B. Needleman and C. D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. J. Mol. Biol., 48(3):443–53, Mar 1970.

[47] David Alberto Baião da Constantina Jácome Nogueira. Accelerating a BWT-based exact search on multi-GPU heterogeneous computing platforms. Master's thesis, Instituto Superior Técnico, Lisboa, Portugal, 2014.

[48] NVIDIA Corporation. CUDA C Best practices guide. NVIDIA Corporation, September 2015.

[49] C. B. Olson, M. Kim, C. Clauson, B. Kogon, C. Ebeling, S. Hauck, and W. L. Ruzzo. Hardware acceleration of short read mapping. In Field-Programmable Custom Computing Machines (FCCM), 2012 IEEE 20th Annual International Symposium on, pages 161–168, April 2012. doi: 10.1109/FCCM.2012.36.

[50] W. R. Pearson and D. J. Lipman. Improved tools for biological sequence comparison. Proc. Natl. Acad. Sci. U.S.A., 85(8):2444–8, Apr 1988.

[51] D. Razmyslovich, G. Marcus, M. Gipp, M. Zapatka, and A. Szillus. Implementation of Smith-Waterman Algorithm in OpenCL for GPUs. In Parallel and Distributed Methods in Verification, 2010 Ninth International Workshop on, and High Performance Computational Systems Biology, Second International Workshop on, pages 48–56, Sept 2010. doi: 10.1109/PDMC-HiBi.2010.16.

[52] T. Rognes and E. Seeberg. Six-fold speed-up of Smith-Waterman sequence database searches using parallel processing on common microprocessors. Bioinformatics (Oxford, England), 16(8): 699–706, Aug 2000.

[53] Torbjørn Rognes. Faster Smith-Waterman database searches with inter-sequence SIMD parallelisation. BMC Bioinformatics, 12(1):221, 2011. ISSN 1471-2105. doi: 10.1186/1471-2105-12-221.

[54] E. F. de O. Sandes and A. C. M. A. de Melo. Smith-Waterman Alignment of Huge Sequences with GPU in Linear Space. In Parallel Distributed Processing Symposium (IPDPS), 2011 IEEE International, pages 1199–1211, May 2011. doi: 10.1109/IPDPS.2011.114.

[55] Edans Flavius O. Sandes and Alba Cristina M.A. de Melo. CUDAlign: Using GPU to Accelerate the Comparison of Megabase Genomic Sequences. SIGPLAN Not., 45(5):137–146, January 2010. ISSN 0362-1340. doi: 10.1145/1837853.1693473.

[56] D. Sankoff. Matching sequences under deletion-insertion constraints. Proc. Natl. Acad. Sci. U.S.A., 69(1):4–6, Jan 1972.

[57] I. Savran, Yang Gao, and J.D. Bakos. Large-Scale Pairwise Sequence Alignments on a Large-Scale GPU Cluster. Design Test, IEEE, 31(1):51–61, Feb 2014. ISSN 2168-2356. doi: 10.1109/MDAT.2013.2290116.

[58] H. A. Shah, L. Hasan, and N. Ahmad. An optimized and low-cost FPGA-based DNA sequence alignment–a step towards personal genomics. Conf Proc IEEE Eng Med Biol Soc, 2013:2696–2699, 2013.

[59] A. Sharman. The many uses of a genome sequence. Genome Biol., 2(6):REPORTS4013, 2001.

[60] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. J. Mol. Biol., 147(1):195–7, Mar 1981.

[61] Adam Szalkowski, Christian Ledergerber, Philipp Krähenbühl, and Christophe Dessimoz. SWPS3 - fast multi-threaded vectorized Smith-Waterman for IBM Cell/B.E. and x86/SSE2. BMC Res Notes, 1:107, 2008. doi: 10.1186/1756-0500-1-107.

[62] E. Ukkonen. On-line construction of suffix trees. Algorithmica, 14(3):249–260, 1995. ISSN 1432-0541. doi: 10.1007/BF01206331.

[63] Esko Ukkonen. Finding approximate patterns in strings. J. Algorithms, 6(1):132–137, 1985. doi: 10.1016/0196-6774(85)90023-9.

[64] Esko Ukkonen. Approximate string matching over suffix trees, pages 228–242. Springer Berlin Heidelberg, Berlin, Heidelberg, 1993. ISBN 978-3-540-47732-7. doi: 10.1007/BFb0029808.

[65] Peter Weiner. Linear pattern matching algorithms. In Proceedings of the 14th Annual Symposium on Switching and Automata Theory (Swat 1973), SWAT '73, pages 1–11, Washington, DC, USA, 1973. IEEE Computer Society. doi: 10.1109/SWAT.1973.13.

[66] Kris A. Wetterstrand. DNA Sequencing Costs: Data from the National Human Genome Research Institute (NHGRI) Genome Sequencing Program (GSP). `www.genome.gov/sequencingcosts`, Feb 2016.

[67] A. Wozniak. Using video-oriented instructions to speed up sequence comparison. Comput. Appl. Biosci., 13(2):145–50, Apr 1997.