UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Departamento de Sistemas Informáticos y Computación

DSIC

Inexact Mapping of Short Biological Sequences in High Performance Computational Environments

Dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the subject of Computer Science

(Programa de doctorado en Informática)

Author: José Salavert Torres

Advisor: Ignacio Blanquer Espert

October 14, 2014

Acknowledgements

First of all, I would like to thank my supervisor Prof. Ignacio Blanquer from *Universitat Politècnica de València* for his guidance during the development of this thesis, and also for the opportunity of working in the field of bioinformatics during these years. Also, special thanks to Dr. Andrés Tomás Domínguez for his introduction to GPU computing and support.

Secondly, I would like to express my appreciation to our collaborators at the *Centro de Investigación Príncipe Felipe* in *Valencia (Spain)*, Dr. Joaquín Tárraga, Dr. Ignacio Medina and Dr. Joaquín Dopazo for their partnership in the combined Burrows-Wheeler Transform seeding and Smith-Watterman alignment tool. I would also like to thank the researchers from *Universitat Jaume I de Castelló*, Hector Martínez, Enrique Quintana, Sergio Barrachina and M^a Isabel Castillo for their efforts in the development of the RNAseq analysis tool and the researchers from *Universitat de València* Pascual Pérez and Vicente Arnau for his work in the methylation analysis. The algorithms implemented in this thesis were necessary during the development phase of these tools.

Thirdly, during the internship period at the *National Institute of Informatics* in *Tokio (Japan)* the assistance received by Prof. Kunihiko Sadakane was greatly appreciated, he provided me very valuable advice on suffix array sorting and compression. The *csalib* library and the algorithm for direct Burrows-Wheeler Transform calculation developed at *National Institute of Informatics* are also employed in this study.

Finally, the authors would like to thank the *Universitat Politècnica de València* in the frame of the grant *High-performance tools for the alignment of genetic sequences using graphic accelerators (GPGPUs) / Herramientas de altas prestaciones para el alineamiento de secuencias genéticas mediante el uso de aceleradores gráficos (GPGPUs)*, research program PAID- 06-11, code 2025.

Además quiero agradecer a mi familia y amigos su apoyo, gracias al cual he podido afrontar el trabajo con energía durante estos años.

Summary

During the past years, DNA sequencers have been constantly improved in performance and operating costs, generating a genomic data deluge. This situation has fostered the general improvement and parallelisation of alignment algorithms, taking profit of different high performance environments.

In bioinformatics, the term *alignment* refers to the comparison of two potentially dissimilar reads of DNA, RNA or proteins. This comparison is made in terms of the relationships between its nucleotides: matches, mismatches, insertions and deletions. When aligning short reads, the more concrete term *sequence mapping* is employed. Several algorithms for the inexact mapping of short biological sequences are presented in this thesis, along with its parallelisation in environments like GPGPU and shared memory.

Currently, inexact mapping methods consist on a combination of seeding techniques followed by local alignment techniques. On the one hand, seeding algorithms are usually based on backward search methods, using the Burrows-Wheeler Transform, the Ferragina and Manzini Index and Suffix Arrays to locate the alignment candidate areas of a read. On the other hand, local alignment algorithms generate matrices of weights using dynamic programming, obtaining the best scoring alignment among the candidate areas.

This thesis focuses in backward search methods. Concretely, we describe the relationships between the Burrows-Wheeler Transform, the Suffix Array and the FM-Index of a reference text.

Two backward search algorithms using the FM-Index have been parallelised using GPGPUs in this thesis. The first one covers exact mapping on GPUs. It can be used to accelerate seeding techniques. The second one is an hybrid CPU-GPU implementation, which performs inexact mapping with one error and returns the pair-ends of a read. Both approaches outperform existing implementations.

Also, an inexact mapping algorithm supporting any number of differences has been implemented. Such algorithm combines backward search with search tree exploration techniques, implementing pruning strategies specifically suited for genomic data. This new approach constitutes the most significant contribution of this thesis, achieving higher sensitivity and a 7x speed-up over similar algorithms. This speed-up has been achieved without employing parallelism techniques.

Finally, during the internship in Japan the algorithm has been modified to support an out-of-core index. This index allows to use the inexact mapping algorithm with large genomes on systems without expensive primary memory configurations.

Resumen

Durante los últimos años, los secuenciadores de ADN han sido mejorados en velocidad y costes de funcionamiento, generando una avalancha de datos genómicos. Esto ha fomentado la mejora y paralelización de los algoritmos de alineamiento, buscando aprovechar los distintos entornos de computación de alto rendimiento.

En bioinformática, el término *alineamiento* se define como la comparación de dos lecturas de ADN, ARN o proteínas potencialmente diferentes. Esta comparación se hace en base a las relaciones entre sus nucleótidos: aciertos, fallos, inserciones y borrados. Más específicamente, cuando se comparan secuencias cortas se emplea el término *mapeo de secuencia*. En esta tesis se describen varios algoritmos para el mapeo inexacto de secuencias biológicas cortas, junto con su paralelización en entornos como GPGPU o memoria compartida.

Actualmente, los métodos de mapeo inexacto consisten en una combinación de técnicas de semilleo seguidas de técnicas de alineamiento local. Por un lado, los algoritmos de semilleo suelen basarse en técnicas de búsqueda hacia atrás, utilizando la transformada de Burrows-Wheeler, el índice de Ferragina y Manzini y matrices de sufijos para localizar las áreas donde podría alinearse una lectura. Por otro lado, los algoritmos de alineamiento local generan matrices de pesos usando programación dinámica, obteniendo así el alineamiento mejor puntuado de entre todas las áreas destacadas.

La tesis se enfoca en el estudio de los métodos de búsqueda hacia atrás. Concretamente, describimos las relaciones entre la transformada de Burrows-Wheeler, las matrices de sufijos y el FM-Index de un texto de referencia.

Dos algoritmos de búsqueda hacia atrás que usan el FM-Index se han paralelizado en GPGPUs. El primero permite mapeo exacto en GPUs y puede usarse para acelerar las técnicas de semilleo. El segundo es una implementación CPU-GPU híbrida, la cual permite mapeo inexacto con un error y devuelve los pares finales de una lectura. Los dos superan a las implementaciones existentes.

Además, se ha implementado un algoritmo de mapeo inexacto que permite cualquier número de diferencias. Dicho algoritmo combina búsqueda hacia atrás con técnicas de exploración de árboles de búsqueda, implementando estrategias de poda específicas para datos genómicos. Este nuevo método constituye la contribución más significativa de la tesis, alcanzando mayor sensibilidad y un speed-up de 7x respecto a algoritmos similares.

Finalmente, durante la estancia en Japón el algoritmo ha sido modificado para trabajar con un índice out-of-core. Dicho índice permite usar el algoritmo de mapeo inexacto con genomas grandes en sistemas con configuraciones de memoria primaria limitadas.

Resum

Durant els últims anys, els seqüenciadors d'ADN han estat millorats en velocitat i costos de funcionament, generant una allau de dades genòmiques. Això ha fomentat la millora i paral·lelització dels algorismes d'alineament, buscant aprofitar els diferents entorns de computació d'alt rendiment.

En bioinformàtica, el terme *alineament* es defineix com la comparació de dues lectures d'ADN, ARN o proteïnes potencialment diferents. Aquesta comparació es fa d'acord amb les relacions entre els seus nucleòtids: encerts, errors, insercions i esborrats. Més específicament, quan es comparen seqüències curtes s'empra el terme *mapatge de seqüència*.

En aquesta tesi es descriuen diversos algorismes per al mapatge inexacte de seqüències biològiques curtes, amb la seua paral·lelització en entorns com GPGPU o memòria compartida.

Actualment, els mètodes de mapatge inexacte consisteixen en una combinació de tècniques de cerca de llavors seguides de tècniques d'alineament local. D'una banda, els algorismes de cerca de llavors solen basar-se en tècniques de recerca cap enrere, utilitzant la transformada de Burrows-Wheeler, l'índex de Ferragina i Manzini i matrius de sufixos per localitzar les àrees on podria alinear-se una lectura. D'altra banda, els algorismes d'alineament local generen matrius de pesos usant programació dinàmica, obtenint així l'alineament millor puntuat d'entre totes les àrees destacades.

La tesi s'enfoca en l'estudi dels mètodes de recerca cap enrere. Concretament, descrivim la relació entre la transformada de Burrows-Wheeler, les matrius de sufixos i el FM-Index d'un text de referència.

Dos algorismes de recerca cap enrere que usen el FM-Index s'hi han paral·lelitzat en GPGPUs. El primer permet mapatge exacte en GPUs i pot usar-se per accelerar les tècniques de cerca de llavors. El segon és una implementació CPU-GPU híbrida que permet mapeig inexacte amb un error i retorna els parells finals d'una lectura. Els dos superen les implementacions existents.

A més, s'ha implementat un algorisme de mapatge inexacte que permet qualsevol nombre de diferències. L'algorisme combina recerca cap enrere amb tècniques d'exploració d'arbres de cerca, implementant estratègies de poda específiques per a dades genòmiques. Aquest nou mètode és la contribució més significativa de la tesi, aconseguint major sensibilitat i un speed-up de 7x respecte a algorismes similars.

Finalment, durant l'estada al Japó l'algorisme ha estat modificat per treballar amb un índex out-of-core. Aquest índex permet usar l'algorisme de mapeig inexacte amb genomes grans en sistemes amb configuracions de memòria primària limitades.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Nowadays, the cost reduction of new sequencing technologies has fostered the consideration of much more data than before. A single biological experiment launched on a current sequencing machine [Church, 2006][Hall, 2007] can easily produce hundreds of Gigabytes or even Terabytes of data. It is forecasted that this situation will aggravate, as *Next Generation Sequencers* (NGS) are continuously improved. This well-known data deluge poses bioinformaticians against the challenge of processing and analysing an avalanche of biological information [Editorial, 2009]. In this scenario, leveraging support for high performance environments may provide the necessary computational power to undertake this challenge.

1.1 Overview

A topic actively revised to satisfy NGS needs is the alignment of DNA sequences [Li and Homer, 2010]. In the field of bioinformatics, the term *alignment* refers to the representation and comparison of the similarity areas between two or more chains of DNA, RNA or protein primary structures. More concretely, when aligning short sequences the term *sequence mapping* is employed.

In an experiment, there can be as many as one billion of reads with an average length of 100 nucleotides. We need an effective solution for mapping billions of short reads.

Moreover, during the sequence mapping process differences may appear, mainly due to the natural genetic variability but sometimes originated by failures in the sequence digitalisation phase. Sequence mapping is the first step in most studies of functional or evolutionary relationships between genes or proteins. Furthermore, the study of the genetic similarities between a patient and an individual with a detected disease may be effectively used in diagnostic medicine.

For these reasons, a mapping algorithm must allow a certain number of variations, guaranteeing that reads slightly different to the reference will be found. Based on the nucleotides in the compared sequences, four different kinds of relationships can be established: matches, mismatches (a difference between a read

nucleotide and a reference nucleotide), insertions (an additional nucleotide in the read) and deletions (a missing nucleotide in the read). We refer to these algorithms as inexact sequence mapping techniques.

Current inexact mapping tools consist on a combination of seeding techniques followed by local alignment techniques. Seeding algorithms usually divide the read into small pieces or *seeds*. Then, these seeds are searched using backward search methods. The locations of the reference where many seeds appear are considered alignment candidate areas. Local alignment algorithms generate matrices of weights using dynamic programming. For each candidate area a matrix of weights is generated, obtaining the best scoring alignment of each area. At the end, the best alignment among all candidate areas is selected. Local alignment algorithms also return the list of differences between the read and the reference string.

1.2 Motivation

The alignment of huge NGS data is an intensive task that requires high-capacity and high-capability computing resources. Among the different available choices that provide faster computation models, *General Purpose Graphic Processing Units* (GPGPUs) based on CUDA [NVIDIA, 2014] or OpenCL [Munshi et al., 2011] are a very cost-effective option. Thanks to these frameworks the GPGPU architecture can be exploited efficiently in general purpose problems, taking into account its micro-grain parallelism following the SIMD model and its complex memory hierarchy.

Studying how to improve backward search methods by enabling GPGPU parallelisation has a great interest. A parallelised backward search algorithm would result in an speed-up of the seeding phase of sequence mapping tools.

The methodology to support inexact mapping using backward search methods is based on the exact mapping procedure, combining a backward search algorithm with a search tree exploration routine that checks all the possible solutions within the number of errors allowed.

Although the search tree exploration routine does not seem to fit the SIMD parallelism model of the GPU, it would also be of great interest to develop and study a hybrid CPU-GPU computation model capable of accelerating the exact mapping procedure on the GPU while using the CPU to check for dissimilarities.

Nevertheless, inexact backward search methods performance exponentially decreases with the number of errors allowed during the alignment. In order to prevent the excessive growth of the search tree, existing solutions limit the number of errors, the type of errors (i.e. not allowing insertions and deletions) or does not allow errors in the first positions of the read.

Developing novel bounding techniques to reduce the search tree growth would increase the number of errors allowed during inexact backward search alignment. Also, it would be of great interest to study bounding techniques that do not limit the kind and position of the errors. Such techniques would guarantee 100% sensitivity during backward search analysis.

Although the combination of seeding techniques with local alignment techniques is faster than performing a local alignment against the full reference and its sensitivity is undisputed when dealing with long reads and big gaps, it is still desirable to further improve mapping algorithms based only in backward search methods. In fact, if more errors are supported by these methodologies, then more occurrences would be mapped faster without the need of the combined approach. Moreover, an increased error granularity in either the seeds or the mapping would result in less sequences lost.

1.3 Contents of the PhD thesis

The contents of this document are organised as follows. Chapter 2 overviews previous research on the main topics discussed in this thesis, along with a review of currently available sequence mapping tools. Chapter 3 summarizes the objectives of this PhD thesis. Chapter 4 explains the parallelisation of two FM-Index based sequence mapping algorithms on GPU. Chapter 5 depicts a backward search algorithm for inexact mapping supporting any number of errors. Chapter 6 contains the contributions of this work. Finally, chapter 7 provides a list with the publications related with this thesis and links to the source code of all the original programs.

Chapter 2

State of the Art

Here, we revise previous research on the main topics of this thesis. Regarding our mapping algorithms, we explain the process to build the *Ferragina and Manzini index* (FM-Index) and how to perform backward search using it. We outline the relationships between the FM-Index, the *Suffix Array* (SA) and the *Burrows-Wheeler transform* (BWT). Also, we describe the GPGPU architecture, the organisation of its SIMD processors and its memory hierarchy. We mention the sensitivity, a statistical measure used in the experiments. Finally, we overview existing sequence mapping techniques and tools.

2.1 Backward search with the FM-Index

When not using an external library, the backward search method employed by our mapping algorithms is based on our own implementation of the FM-Index data structures. This section briefly introduces the reader to the FM-Index search theory.

The FM-Index data structures are obtained from the BWT of a reference string. The BWT [Burrows and Wheeler, 1994] is a reversible transform. Its output is composed by the same symbols as its input, but in a different order. This ordering allows data to be compressed efficiently using RLE (Run Length Encoding) techniques [Manzini, 1999]. Traditionally, the BWT has been used in many compression algorithms. In this approach, it is used to create the FM-Index and align short genome sequences [Li and Durbin, 2009].

2.1.1 BWT calculation

First of all, we need to generate the BWT of a reference genome X. Let $\Sigma = \{A, C, G, T\}$ be an alphabet, and \$ a symbol not included in Σ with less lexicographic value than all the symbols in Σ. Let X be a reference string of length $n + 1$, where n is the length of the full reference genome. Let $X[i]$ be the i-th symbol of string X. X is composed by symbols of Σ and terminated with the \$ symbol.

We construct a matrix M with all the possible suffixes of the reference string X by rotating its values. For example, given the reference string $X =$ "AGGAGC\$" the suffix matrix is as follows:

$$M = \begin{pmatrix} A & G & G & A & G & C & \$ \\ G & G & A & G & C & \$ & A \\ G & A & G & C & \$ & A & G \\ A & G & C & \$ & A & G & G \\ G & C & \$ & A & G & G & A \\ C & \$ & A & G & G & A & G \\ \$ & A & G & G & A & G & C \end{pmatrix} \begin{matrix} 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{matrix}$$

We sort alphabetically the rows of M. The sorted matrix is commonly known as the SA of reference string X:

$$SA = \begin{pmatrix} \$ & A & G & G & A & G & C \\ A & G & C & \$ & A & G & G \\ A & G & G & A & G & C & \$ \\ C & \$ & A & G & G & A & G \\ G & A & G & C & \$ & A & G \\ G & C & \$ & A & G & G & A \\ G & G & A & G & C & \$ & A \end{pmatrix} \begin{matrix} 6 \\ 3 \\ 0 \\ 5 \\ 2 \\ 4 \\ 1 \end{matrix} \tag{2.1}$$

Let S be an array composed by a permutation of the integers from 0 to n. S contains the original positions of each suffix in the reference, being a representation of the SA:

$$S = \begin{pmatrix} 6 & 3 & 0 & 5 & 2 & 4 & 1 \end{pmatrix}$$

Let R be an array composed by a permutation of the integers from 0 to n, which satisfies that $R[S[i]] = i$. Vector R is commonly used as a representation of the *Inverse Suffix Array* (ISA). Given a position in the reference, we can use R to obtain the position of a short read in the SA (the inverse of the operation allowed by S vector). This vector is used by our inexact mapping approach:

$$R = \begin{pmatrix} 2 & 6 & 4 & 1 & 5 & 3 & 0 \end{pmatrix}$$

Let B be an array of symbols of the alphabet Σ where the BWT will be stored, we define it as follows:

$$B[i] = \begin{cases} \$ & \text{if } S(i) = 0 \\ X[S(i) - 1] & \text{if } S(i) \neq 0 \end{cases}$$

Vector B contains the BWT. It always corresponds with the elements of the last column of the SA:

$$B = \begin{pmatrix} C & G & \$ & G & G & A & A \end{pmatrix}$$

2.1.2 Suffix Array sorting algorithms

As it is obtained only once for each reference genome, the calculation of the BWT is not a key processing step when performing sequence mapping. Nevertheless, the memory and time requirements of the BWT generation process can be reduced by using better SA sorting algorithms.

If we sort the suffixes as if they were non-related strings, we will end up with a worst-case asymptotic cost of $\mathcal{O}(n^2 * \log n)$. This worst-case always appears when dealing with genomic data, because there are many regions in which nucleotide chains are arbitrarily repeated. Due to this, typical string sorting algorithms can not be used to obtain the ordered SA.

Fortunately, sorting suffix arrays has a much lower computational cost than sorting non-related strings. A taxonomy of suffix array sorting methods is available in [Puglisi et al., 2007]. As it can be seen in the taxonomy, there are many algorithms with an asymptotic cost of $\mathcal{O}(n * (\log n)^2)$, like the Larsson approach [Larsson and Sadakane, 2007], based on a *ternary quicksort* [Bentley and Sedgewick, 1997]. In this approach, a partial ordering considering only the first k symbols of each suffix contains enough information to allow a subsequent partial ordering with the first $2k$ symbols. This is because we are ordering suffixes of the same reference string.

Recent solutions to the problem include the DC3 algorithm [Kärkkäinen et al., 2006], with a parallel implementation [Kulla and Sanders, 2006]. The fastest approach to the problem is the SA-IS algorithm [Nong et al., 2009], with a memory efficient version that computes the BWT directly [Okanohara and Sadakane, 2009], obtaining the compressed SA positions later [Grossi and Vitter, 2005]. We selected this approach to compute the BWT because it does not need to store the full S vector into memory.

When performing short read alignments, the calculation of the BWT can be accelerated considerably by reducing the complexity of the sorting process. Concretely, we do not need to fully sort the suffixes of the reference string. We could only sort the first k characters of each suffix, where k must be greater than the length of the longest read we are going to map.

For example, in order to map reads with a maximum length of two elements ($k = 2$) in the reference string "AGGAGC$", we will only apply the sorting algo-

rithm to the two first characters of each suffix matrix row:

$$SA' = \begin{pmatrix} \$ & A & G & G & A & G & C \\ \mathbf{A} & \mathbf{G} & \mathbf{G} & \mathbf{A} & \mathbf{G} & \mathbf{C} & \mathbf{\$} \\ \mathbf{A} & \mathbf{G} & \mathbf{C} & \mathbf{\$} & \mathbf{A} & \mathbf{G} & \mathbf{G} \\ C & \$ & A & G & G & A & G \\ G & A & G & C & \$ & A & G \\ G & C & \$ & A & G & G & A \\ G & G & A & G & C & \$ & A \end{pmatrix}$$

In this case:

$$B' = \begin{pmatrix} C & \$ & G & G & G & A & A \end{pmatrix}$$

which is a different value with respect to the full ordering. Rows 2 and 3 are not alphabetically ordered but continue providing an interval with all the occurrences of string "AG". The values of S and R also change:

$$S' = \begin{pmatrix} 6 & 0 & 3 & 5 & 2 & 4 & 1 \end{pmatrix}$$

$$R' = \begin{pmatrix} 1 & 6 & 4 & 2 & 5 & 3 & 0 \end{pmatrix}$$

Given that we know the maximum length of the reads to align (k), SA sorting algorithms can be greatly simplified using this consideration (section 2.1.2). However, many compression techniques will not work properly without a full ordering of the SA, so we do not use this optimisation (this is explained in section 4.1).

2.1.3 FM-Index data structures

The FM-Index data structures [Ferragina and Manzini, 2000] needed by the search algorithm are obtained from the BWT stored in vector B. These data structures are vector C and matrix O.

Let $C(a)$ be the number of symbols in B (excluding \$) lexicographically smaller than $a \in \Sigma$. For example, contents of vector C for B = "CG\$GGAA" are the following:

$$C = \begin{pmatrix} 0 & 2 & 3 & 6 \end{pmatrix}$$

The positions in vector 'C' correspond to 'A', 'C', 'G' and 'T'. For example, the value of the first element of C means that there are no characters lexicographically smaller than 'A'. Accordingly, the rest of the elements denote that there are two characters smaller than 'C' (two 'A'), three ones smaller than 'G' (two 'A' plus one 'C') and six characters smaller than 'T' (three additional 'G').

Let $O(a, i)$ be the number of occurrences of symbol $a \in \Sigma$ in $B[0 : i - 1]$. The size of O is $(|B| + 1) \times |\Sigma|$, because the first column represents an empty interval in B and its values are always 0. For example, content of matrix O for B = "CG\$GGAA" is as follows:

$$O = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 1 & 2 \\ 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 2 & 3 & 3 & 3 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \begin{matrix} A \\ C \\ G \\ T \end{matrix}$$

For example, the value at position $O(G, 4)$ is 2 ('G' occupies the third row), this means that 'G' appears twice in $B[0 \dots 3]$.

We also define B_r, O_r, S_r and R_r as the data structures of the reversed reference text X_r. This reverse index allows to change the direction of the analysis during the search, but increases the memory requirements. Bidirectional methods [Lam et al., 2009] solve this issue but may not be efficient in all cases, see section 5.3 for a more detailed discussion.

We may want to calculate B without directly storing vector S, drastically reducing the memory requirements of the BWT generation process [Okanohara and Sadakane, 2009]. In such case a version of S and R compressed with a k ratio can be obtained from the FM-Index data structures. We use the decompression algorithms described in sections 5.1.1 and 5.1.2, starting at the position of the \$ symbol in B and storing only the values of S and R every k positions.

2.1.4 FM-Index of the complementary strand

Sequenced DNA has two different possible strand orientations, which are called the forward and complementary strand respectively. The segments in the complementary strand orientation appear in the opposite direction and the bases are swapped with its complementary ones ($A \leftrightarrow T$, $C \leftrightarrow G$). In this case the DNA segments have been inserted into the double helix in the opposite direction. In order to search for mappings in segments oriented in the complementary strand direction, we can apply two minor modifications to the FM-Index search algorithm.

First, the search direction is inverted, as we need to search for the reverse string. Second, we search for the complementary bases ($A \leftrightarrow T$, $C \leftrightarrow G$) so it is necessary to swap elements of vector C and rows of matrix O, obtaining C_c and O_c as follows:

$$C_c = \begin{pmatrix} 6 & 3 & 2 & 0 \end{pmatrix}$$

$$O_c = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 2 & 3 & 3 & 3 \\ 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 2 \end{pmatrix} \begin{matrix} T \\ G \\ C \\ A \end{matrix}$$

2.1.5 Recursive Backward Search algorithm

Once the FM-Index is calculated, we can use it to perform backward search. Now, we explain the basic methodology of backward search algorithms, after that we describe the specific formulas for searching with the FM-Index. Finally, we clarify these mechanics with an example.

Let W be a substring of X. The SA is alphabetically sorted (equation 2.1), so all suffixes that contain W as prefix appear in a continuous interval $IN = [k, l]$ of SA rows. We define the upper and lower values of this interval as follows:

$$\lfloor IN(W) \rfloor = \min\ [k \mid W \text{ is a prefix of } SA(k, \dots)]$$

$$\lceil IN(W) \rceil = \max\ [l \mid W \text{ is a prefix of } SA(l, \dots)]$$

The starting locations in X of each SA row are stored in vector S. Therefore, given a string W, its appearances in X can be obtained from the defined interval $IN = [k, l]$, being $S[k]$ the first element of S that points to a location of W in X and $S[l]$ the last element of S that points to a location of W in X. If W is an empty string then $k = \lfloor IN([\,]) \rfloor = 0$ and $l = \lceil IN([\,]) \rceil = |S| - 1$, denoting that all the suffixes in SA accept an empty string as prefix.

For example, searching W = "AG" in the reference X = "AGGAGC$", we obtain the interval $[k, l] \leftarrow [1, 2]$, with $S[1] = 3$ and $S[2] = 0$:

$$\begin{array}{rccccccc} & \$ & A & G & G & A & G & C & 6 \\ \lfloor IN(W) \rfloor \rightarrow & \mathbf{A} & \mathbf{G} & C & \$ & A & G & G & 3 \\ \lceil IN(W) \rceil \rightarrow & \mathbf{A} & \mathbf{G} & G & A & G & C & \$ & 0 \\ & C & \$ & A & G & G & A & G & 5 \\ & G & A & G & C & \$ & A & G & 2 \\ & G & C & \$ & A & G & G & A & 4 \\ & G & G & A & G & C & \$ & A & 1 \end{array}$$

Backward search methods allow to recursively obtain the $IN = [k, l]$ interval of a read W, starting from its last symbol up to the first one. In the following recursive equation, BSK and BSL are respectively generic functions that return the lower and upper values of the next interval $IN(aW')$, given the previous interval $IN(W')$ and the preceding symbol a:

$$\lfloor IN(a : W') \rfloor = BSK(a, \lfloor IN(W') \rfloor)$$

$$\lceil IN(a : W') \rceil = BSL(a, \lceil IN(W') \rceil)$$

In this function, operator : extracts the head symbol from a string until reaching the base case of the interval, $IN([\,])$. Then the recursive calls are solved performing the search in backward direction. If W is a substring of X, then the interval values must always satisfy $\lfloor IN(W) \rfloor \leq \lceil IN(W) \rceil$.

Backward search methods determine if W is a substring of X in $\mathcal{O}(|W|)$ time. On each step, a symbol of W is analysed obtaining new values of $\lfloor IN(W') \rfloor$ and $\lceil IN(W') \rceil$ for the current substring W'. As more steps are performed the interval $[\lfloor IN(W') \rfloor, \lceil IN(W') \rceil]$ becomes thinner, until the first symbol.

We can use the FM-Index to perform backward search and obtain $\lfloor IN(W) \rfloor$ and $\lceil IN(W) \rceil$. The recursive function uses the auxiliary vectors C and O. In [Ferragina and Manzini, 2000] the following relation is demonstrated:

$$\lfloor IN(a : W') \rfloor = C(a) + O(a, \lfloor IN(W') \rfloor - 1) + 1$$

$$\lceil IN(a : W') \rceil = C(a) + O(a, \lceil IN(W') \rceil)$$

Following the example in section 2.1.1, we search for string $W =$ "AGG" on the reference $X =$ "AGGAGC\$", the iterations of the FM-Index function for backward search are:

1. Initialise.

$$\begin{array}{rcccccc|c} \lfloor IN([\,])\rfloor \rightarrow & \$ & A & G & G & A & G & C \\ & A & G & C & \$ & A & G & G \\ & A & G & G & A & G & C & \$ \\ & C & \$ & A & G & G & A & G \\ & G & A & G & C & \$ & A & G \\ & G & C & \$ & A & G & G & A \\ \lceil IN([\,])\rceil \rightarrow & G & G & A & G & C & \$ & A \\ \hline & & & & & & & B \end{array}$$

$$\lfloor IN([\,])\rfloor \leftarrow 0$$

$$\lceil IN([\,])\rceil \leftarrow 6 \leftarrow |S| - 1$$

At the beginning no symbols of W have been analysed yet, so we initialise the interval values to the first and last row values of SA. This means that the empty string is a prefix of all the suffixes of X. As stated before, the last column of the SA is vector B.

2. Substring "$\underline{\text{G}}$". Previous interval $[0, 6]$.

$$\begin{array}{rcccccc|c} & \$ & A & G & G & A & G & C \\ & A & G & C & \$ & A & G & \mathbf{G} \\ & A & G & G & A & G & C & \$ \\ & C & \$ & A & G & G & A & \mathbf{G} \\ \lfloor IN(G)\rfloor \rightarrow & \mathbf{G} & A & G & C & \$ & A & \mathbf{G} \\ & \mathbf{G} & C & \$ & A & G & G & A \\ \lceil IN(G)\rceil \rightarrow & \mathbf{G} & G & A & G & C & \$ & A \\ \hline & & & & & & & B \end{array}$$

$$\lfloor IN(G : [])\rfloor \leftarrow C(G) + O(G, 0) + 1 = (3) + (\underline{0}) + 1 = 4$$

$$\lceil IN(G : [])\rceil \leftarrow C(G) + O(G, 6 + 1) = (3) + (\underline{0} + \mathbf{3}) = 6$$

The first symbol to analyse is the last symbol of W. We apply the FM-Index equation to update the interval values. Vector C indicates that there are 3 symbols with a smaller lexicographic value than 'G' in B. For the lower value of the interval, matrix O gives the number of appearances of symbol 'G' in $B[\varnothing]$, which is always 0 because column 0 of O represents an empty interval of B. For the upper value of the interval, matrix O returns the number of appearances of 'G' in $B[0 : 6]$, wich is 3.

The symbols of the first column of SA are sorted alphabetically, so we can intuitively see that in the lower value of the interval the $C(G)$ value points

to the last symbol smaller than 'G' in the first column of SA. In the upper value of the interval we add to the value of $C(G)$ the value of $O(G, 6+1)$, which is a count of all the appearances of 'G' in B.

3. Substring "$\underline{\text{G}}$G". Previous interval $[4, 6]$.

$$
\begin{array}{rcccccccc|c|c}
 & \$ & A & G & G & A & G & C & & \$ \\
 & A & G & C & \$ & A & G & \underline{G} & \rightarrow & \underline{A} \\
 & A & G & G & A & G & C & \$ & & A \\
 & C & \$ & A & G & G & A & \underline{G} & \rightarrow & \underline{C} \\
\hline
 & \underline{G} & \underline{A} & G & C & \$ & A & \mathbf{G} & \rightarrow & \mathbf{G} \\
 & \underline{G} & \underline{C} & \$ & A & G & G & A & & G \\
\lfloor IN(GG)\rfloor, \lceil IN(GG)\rceil \rightarrow & \mathbf{G} & \mathbf{G} & A & G & C & \$ & A & & G \\
\hline
 & & & & & & & B & &
\end{array}
$$

$$\lfloor IN(G : [G])\rfloor \leftarrow C(G) + O(G, 4) + 1 = (3) + (\underline{2}) + 1 = 6$$

$$\lceil IN(G : [G])\rceil \leftarrow C(G) + O(G, 6+1) = (3) + (\underline{2} + \mathbf{1}) = 6$$

As in the previous step, the value of $C(G)$ points to the last symbol smaller than 'G' in the first column of SA, but in this case we search for appearances of substring "GG".

The SA is constructed by shifting the values of X in rows and sorting them (section 2.1.1). The symbols in the last column of SA (vector B) have the property of preceding the symbols in the first column of SA, which is alphabetically sorted. So, the previous interval values $[4, 6]$ indicate that the symbols in $B[0:3]$ precede suffixes lexicographically smaller than the previously analysed symbols (in this case the single symbol 'G'), while the symbols in $B[0:6]$ precede suffixes lexicographically smaller or equal to the previously analysed symbols.

For the lower value of the interval, we obtain from $O(G, 4) = (\underline{2})$ the number of 'G' symbols in the first column that precede suffixes lexicographically smaller than the previously analysed symbols. For the upper value of the interval, we obtain from $O(G, 6+1) = 3 = (\underline{2}+\mathbf{1})$ the number of 'G' symbols in the first column that precede suffixes lexicographically smaller or equal to the previously analysed symbols.

We split the value of $O(G, 6+1)$ into two values, indicating with an underline the number of 'G' preceding suffixes lexicographically smaller than "G" ($\underline{2}$), and with a bold font the number of 'G' preceding suffixes lexicographically equal to "G" ($\mathbf{1}$). Such suffixes have been highlighted in the example matrix, both in the first and last rows, using the same convention.

4. Substring "$\underline{\text{A}}$GG". Previous interval $[6, 6]$.

$$\lfloor IN(AGG)\rfloor, \lceil IN(AGG)\rceil \rightarrow \begin{array}{cccccc|c|c|cc} \$ & A & G & G & A & G & C & & \$ & A \\ \underline{A} & \underline{G} & \underline{C} & \$ & A & G & G & & A & G \\ \mathbf{A} & \mathbf{G} & \mathbf{G} & A & G & C & \$ & & A & G \\ C & \$ & A & G & G & A & G & & C & \$ \\ G & A & G & C & \$ & A & G & & G & A \\ G & C & \$ & A & G & G & \underline{A} & \rightarrow & \underline{G} & \underline{C} \\ \hline G & G & A & G & C & \$ & \mathbf{A} & \rightarrow & \mathbf{G} & \mathbf{G} \\ & & & & & & \mathbf{B} & & & \end{array}$$

$$\lfloor IN(A : [GG])\rfloor \leftarrow C(A) + O(A, 6) + 1 = 0 + (\underline{1}) + 1 = 2$$

$$\lceil IN(A : [GG])\rceil \leftarrow C(A) + O(A, 6 + 1) = 0 + (\underline{1} + \mathbf{1}) = 2$$

This step is similar to the previous one, but now the previously analysed symbols are suffix "GG". For the lower value of the interval, we obtain from $O(A, 6) = (\underline{1})$ the number of 'A' symbols in the first column that precede suffixes lexicographically smaller than "GG". For the upper value of the interval, we obtain from $O(A, 6 + 1) = 2 = (\underline{1} + \mathbf{1})$ the number of 'A' symbols in the first column that precede suffixes lexicographically smaller or equal to "GG". In this case there are no symbols lexicographically smaller than 'A', so $C(A)$ value is 0.

Notice that the interval lower and upper positions change according to the last analysed symbol, in this case 'A'. This means that although on each iteration the interval size becomes equal or narrower, the new interval may not be contained within the range of the previous interval.

5. Result. The final interval IN is $[k, l] = [2, 2]$. $S = (6\ 3\ 0\ 5\ 2\ 4\ 1)$.

   There is only a single position in X: $S[2] \rightarrow X[0] \rightarrow$ "$\underline{\text{AGG}}$AGC$".

2.1.6 Iterative Backward Search algorithm

From here, we refer to the $\lfloor IN\rfloor$ and $\lceil IN\rceil$ interval range as $[k, l]$. The positions of all the occurrences of a string W in X is the contiguous interval $S[k \dots l]$, containing all the suffixes of X that start with W. To obtain $[k, l]$ of a string W we can use algorithm 2.1, which is the iterative version of the recursive function explained before. The initial values are $k = 0$ and $l = |S| - 1$. The variable *index* is a generalisation of the index used for the backward search.

If we perform backward search using the FM-Index, then the *index* variable will contain the C and O data structures and the function **search_iteration** of algorithm 2.1 will be defined as in algorithm 2.2.

On each **search_iteration** a symbol of W is analysed obtaining an equal or narrower $[k, l]$ interval for the larger substring. At the end, if $k \leq l$ string W belongs to X.

Algorithm 2.1: Exact Backward Search

```
1: exact_backward(IN: W, index. OUT: r.)
2:    [k, l] ← [0, size(index) − 1]
3:    for i ← |W| − 1 . . . 0
4:       [k, l] ← search_iteration([k, l], W[i], index)
5:       if k > l break
6:    end for
7:    r ← [k, l] at i with [ ]
8: end function
```

We return the result in variable r using a special notation ($[k, l]$ at i with $[\,]$), this means that we return interval $[k, l]$, pointing to the symbol of W at position i (where the search stopped) and setting and empty error list (this is exact search).

Algorithm 2.2: Search iteration with the FM-Index

```
1: search_iteration(IN: [k, l], b, index. OUT: [k′, l′].)
2:    k′ ← index.C[b] + index.O[b][k] + 1
3:    l′ ← index.C[b] + index.O[b][l + 1]
4: end function
```

Any backward search method providing an implementation of the function **search_iteration** and wrappers to access the SA (vector S) and ISA (vector R) will be compatible with the algorithms described in this thesis. We employ different indexes on each implementation, including CPU and GPU implementations of the FM-Index and the *csalib* [Sadakane, 2010] out-of-core implementation.

2.2 General Purpose GPU

The Moore's law states that the number of transitors on a chip doubles every two years [Moore, 1998][Moore, 1975]. This statement was consistent with what was observed since 1965. The continuous improvement of CPU computer power has always been driven by the Moore's law, enabling more complex operators and better architectures in a reduced space (instruction pipelines, super-scalar architectures, out-of-order executions, vectorial instructions).

Since the beginning of 2000s, the frequency of CPUs has ceased to raise. Heat dissipation is a problem that prevents increasing the clock frequency of electronic circuits and, as a consequence, the Moore's law is no longer valid. Currently, this problem is addressed by designing parallel architectures with low energy consumption. In this type of architectures, machines have many processing cores that are optimized to work efficiently together.

The evolution of mainstream CPU is currently driven by the multiplication of the number of computational cores, rather than dedicating efforts to improve

single-core architectures. However, *multi-core* processors produced by both Intel and AMD use too much energy per instruction, which impedes connecting a large number of them in parallel.

More recently, a new parallel architecture with low energy consumption has appeared. Thanks to the development effort done by the electronic entertainment industry, video cards have greatly increased their computing capacity with a high integration level of concurrent execution cores. *General Purpose Graphical Processing Units* (GPGPU) include the necessary logic to execute any kind of program, becoming powerful general purpose CPU featuring an unbeatable degree of parallelism for specific tasks. These modern graphic cards rely on *many-core* processors, whose energy per instruction is reduced by sharing the same hardware logic among several simultaneous instructions.

These solutions are not so different today, multi-core CPU have increased the number of cores, while many-core GPUs have more complex hardware design. Applications for these architectures follow the parallel programming model, instead of the previously existing serial programming model. When following the parallel programming model, there are several degrees of parallelism depending on the underlying architecture, namely *inter-core* parallelism and *intra-core* parallelism.

Inter-core parallelism is achieved when several cores execute different code concurrently, synchronising their execution to access the same memory space. This degree of parallelism is also called *coarse-grain* parallelism. This is the maximum degree of parallelism that can be achieved between independent cores. Inter-core parallelism can be defined in the programs by using explicit multi-thread programming or with the help of high level frameworks such as OpenMP [Board, 2013]. This parallelism naturally applies to both multi-core CPU and many-core GPGPU processors.

Intra-core parallelism involves parallelism inside a computational core, for example through the *Single Instruction Multiple Data* (SIMD) model. This degree of parallelism is also called *fine-grain* parallelism. The intra-core parallelism can be defined in the programs by using specific calls to vector instructions or can be automatically inferred by an optimising compiler. Again, this parallelism applies to both multi-core CPU and many-core processors. However, the SIMD model is fundamental for the performance of GPGPU, because the SIMD instructions involve much more simultaneous data elements than a mainstream CPU. Another example of intra-core parallelism is out-of-order execution, where independent instructions can be executed simultaneously, regardless of their execution order in the program.

The very good ratio of performance versus cost of GPGPU has fostered bioinformatics researchers to consider such platforms for data analysis [Varré et al., 2011]. Previous scientific research was based on custom designed hardware and parallel supercomputing technologies, whose cost is not within the reach of every research group. High-end GPGPUs can deliver computational capabilities greater than those provided by CPUs. These capabilities became accessible thanks to the CUDA programming environment [NVIDIA, 2014] by Nvidia and more re-

cently with the OpenCL open standard [Khronos, 2011][Munshi et al., 2011] by the Khronos Group.

In some cases, when the memory and computational requirements of an algorithm fit the GPGPU architecture constraints, graphic cards provide a great computational power at an affordable price. However, and due to the programming complexity that GPGPUs imply, the application of these computational resources has been limited to few approaches.

A Nvidia GPU is composed of several multithreaded multiprocessors (SM). An SM is composed of an unique instruction decoder and several thread processors (SP). This configuration allows thread parallelism inside the SM only if each SP executes the same instruction on different data. For this reason the parallelism model of the GPU is often referred as *Single Instruction Multiple Thread* (SIMT), instead of SIMD.

Hence, conditional constructs must be avoided in order to take full profit of the parallelism at SP level. As it can be seen in figure 2.2 several MP can be included in a single *Texture Processor Cluster* (TPC), sharing the same access channel to the global memory. Due to this, memory accesses must be minimised to obtain a good speed-up using the GPU.

Programs that are launched on the GPU are divided into threads of execution. These threads are the operations in which the main program is parallelised. The main program is coded as what is called a kernel. CUDA execution threads are organised in blocks and blocks are organised into a execution grid. Each CUDA kernel parallel execution is structured following its corresponding grid hierarchy (figure 2.3). The details about the development and execution of CUDA kernels can be found at chapter 2 of [NVIDIA, 2014].

At execution time, threads are grouped by warps inside each SM and scheduled to be executed in the SP automatically, this part of the execution process is transparent to the user and can not be controlled. The user only controls which threads are in the same block and can share information using the shared memory of a SM.

A CUDA GPU has different memory types (figure 2.2):

- The *main memory* is the global memory, which allows to store gigabytes of information. All the SM are able to access it. Main memory has a great performance penalty because the information resides outside all the SM chips.
- On the other side, the *shared memory* is the memory of each SM. All the threads within the same block are able to access this memory. This memory is really fast, but of small size and a thread on an SM cannot access the shared memory of another SM.
- *Registers* and *local memory* reside also in the SM hierarchy. However, they contain elements which are private of each thread.
- *Texture cache* and *constant cache* are memory caches that the programmer can use to accelerate the access to the global memory. These caches can

be accessed by all the SM. The *texture cache* provides 2D locality. The *constant cache* provides 64K of space to accelerate repeated access to the same constant values.

All the threads of a block are executed in the same SM, accessing the same shared memory and executing concurrently (figure 2.2). When a thread reads information from the GPGPU global memory the execution speed is reduced. In order to avoid this bottleneck programs must take profit from the shared memory and the registers as much as possible.

General Purpose Graphic Processing Units (GPGPUs) constitute an inexpensive resource for computing-intensive applications able to exploit its memory hierarchy and intrinsic fine-grain parallelism. The industrial effort made to convert GPU into GPGPU has led to an improved architecture, increasing the transfer rate of the bus that connects the GPU to the CPU memory, optimizing certain machine instructions that in principle are not necessary to accelerate graphics and even including in the latest models a double precision floating point unit.

These new functionalities have also led to the emergence of two GPU development frameworks: CUDA [NVIDIA, 2014] and OpenCL [Khronos, 2011][Munshi et al., 2011]. CUDA is a mature solution, but only works on Nvidia GPUs. OpenCL has been designed as a standard GPGPU development framework, being the programming environment of several brands of GPGPUs. OpenCL allows code generation at runtime without the aid of external tools.

The portability of the OpenCL code does not imply that it will work optimally on all GPGPU models without an extra effort from the programmer. Furthermore, CUDA allows C++ language constructs, while the OpenCL standard only allows C language constructs. The CUDA framework also includes several debugging utilities which are not present in OpenCL.

2.3 Sensitivity (TPR)

The *sensitivity* is a statistical measure of the performance of a binary classification test, also known in statistics as classification function. Sensitivity (also called the *True Positive Rate* (TPR), or the *recall rate* in some fields) measures the proportion of actual positives which are correctly identified as such and is complementary to the false negative rate.

The equation to obtain the *sensitivity* in an experiment is the following:

$$TPR = \frac{TP}{TP + FN}$$

where TP are the true positives and FN are the false negatives.

We use this parameter to compare our sequence mapping algorithm with similar approaches. Our algorithms focus on strategies to obtain full sensitivity during the search ($TPR = 1$). This means that there are no reads not being found in the reference given the amount of errors allowed (there are no false negatives). The

sensitivity value shows how close to our approach are the results of other backward search mapping algorithms.

2.4 Sequence alignment tools

Currently, several algorithms address the sequence alignment problem. As we have explained in the preceding state-of-the-art sections, there are different design approaches, each one suitable for distinct experimental scenarios [Li and Homer, 2010]. Different bioinformatics applications employ these algorithms. In this section we overview the existing sequence alignment tools, with a special focus on sequence mapping tools [Hatem et al., 2013].

Several alignment solutions available in the literature focus on dynamic programming approaches, like the *Needleman-Wunsch algorithm* [Needleman and Wunsch, 1970] for global alignment or the *Smith-Waterman* [Smith and Waterman, 1981] and *Gotoh* [Gotoh, 1982] algorithms for local alignment. In such approaches an implicit matrix of weights or likelihoods is generated and the maximum scoring path for that matrix is obtained, allowing to select the most biologically probable results. Among the different scoring schemes available, those based on the *Hidden Markov Models* [Durbin et al., 1998] (HMM) are the most complete ones. Working with such mathematical models provides good error sensitivity when searching for homologies, so they are typically used to align dissimilar or gapped sequences of average length; however, their computational cost depends on the length of the read multiplied by the length of the reference genome.

Several tools follow this approach, including FASTA [Lipman and Pearson, 1985][Pearson and Lipman, 1988] and BLAST [Altschul et al., 1990][Altschul et al., 1997]. Concretely, FASTA includes SSEARCH [Pearson, 1991] implementation of the SW algorithm. BLAST adds the concept of *high score pairs* to speed up the process. Both tools feature different alignment kernels with code optimisations for CPU vector instruction sets (Altivec, SSE) and support for parallel computing on specific hardware, among other features.

Statistical models have also been employed to align biological sequences [Durbin et al., 1998]. Concretely, HMMER [Eddy, 1998], SAM [Hughey and Krogh, 1995], Clustal Omega [Sievers et al., 2011] and T-Coffee [Notredame et al., 2000] Rhmm package are based on *Hidden Markov models* [Durbin et al., 1998] and feature computational requirements similar to SW solutions. These tools focus on improving result scoring accuracy by using more precise mathematical models.

Therefore, these options are not efficient when performing the alignment of short reads. State-of-the-art short read mapping techniques are based in backward search methods over *Suffix arrays* [Manber and Myers, 1990] (SA). Some of these methods require the generation of an index with the *Burrows Wheeler Transform* (BWT). One of these indexes is the *Ferragina and Manzini Index* [Ferragina and Manzini, 2005] (FM-Index). The BWT has been originally used in data compression techniques [Burrows and Wheeler, 1994][Manzini, 1999], but the FM-Index [Ferragina and Manzini, 2000] allowed the design of recursive backwards

searching algorithms for inexact mapping [Li and Durbin, 2009]. Backward search techniques using the FM-Index reduce the computational complexity of the exact mapping process to the order of the length of the read. It is nevertheless true that this comes at the expense of greatly reducing the error sensitivity and, consequently, the length of the sequences mapped.

Early short read aligners are based on FM-Index [Ferragina and Manzini, 2005] backward search methods. This is the case of early versions of BWA [Li and Durbin, 2009], SOAP2 [Li et al., 2009], and the currently maintained Bowtie 1 [Langmead et al., 2009]. However, these tools allow a limited number of alignment errors, due to restrictions in memory and performance. For this reason, backward search techniques are employed to locate small segments of the reads (seeds) in the genome, revealing alignment candidate areas that are analysed using dynamic programmming approaches. BWA [Li and Durbin, 2010], Bowtie 2 [Langmead and Salzberg, 2012] and SeqAlto [Mu et al., 2012] combine FM-Index multi-seed preprocessing with dynamic programming methods.

Other approaches create a hash table to store the positions in the reference of all the possible substrings of the read with a given length. This hash table provides all the starting search points of any input sequence, namely *k-mers*. This technique has been very successful for the alignment of short reads. For larger sequences, increasing the size of the segments in the hash table could reduce the execution time, but exponentially increases the size of the hash table. Tools like SSAHA [Ning et al., 2001a], SOAP or BLAT [Kent, 2002] implement this approach. More modern tools like SSAHA2 [Ning et al., 2001b] and GEM [Marco-Sola et al., 2012] locate k-mers to be used as prefixes that are also explored using dynamic programming approaches.

Additionally, there are prefix search techniques based on *Suffix array* (SA) [Manber and Myers, 1990] and enhanced SA [Abouelhoda et al., 2002] theory with applications to bioinformatics. Concretely, essaMEM [Vyverman et al., 2013] and PsiRa [Oguzhan Kulekci et al., 2011] tools are based on sparse SA. Backward search methods can also be applied to SA in a similar fashion as the FM-Index [Sadakane, 2003]. The computational cost of prefix search methods depends on the length of the prefix and a scalar value that can be improved depending on the data structures employed.

One of the first steps in the adaptation of alignment tools to GPGPU architectures is an implementation on OpenGL (an environment totally oriented to graphics and very restricted for general programming) of the Smith-Waterman algorithm [Liu et al., 2006], developed in 2006. Nowadays, with the aid of specialised toolkits like CUDA [NVIDIA, 2014] and OpenCL [Khronos, 2011][Munshi et al., 2011] the complexity of this task has been reduced considerably.

There are several implementations on GPUs of Smith-Waterman [Manavski and Valle, 2008][Ligowski and Rudnicki, 2009][Striemer and Akoglu, 2009] and BLAST [Vouzis and Sahinidis, 2011][Ling and Benkrid, 2010] algorithms. In [Manavski and Valle, 2008] an implementation of the Smith-Waterman algorithm using CUDA is presented, achieving a speed-up between 2 and 30 with respect to the

execution of the same algorithm on conventional CPU architectures. Improving access to data, which is the main bottleneck in CUDA, and using several processing boards concurrently [Ligowski and Rudnicki, 2009] and [Striemer and Akoglu, 2009] achieve improvements five times larger.

Regarding sequence mapping tools based on the FM-Index, CUSHAW3 [Liu et al., 2014] and Barracuda [Klus et al., 2012] support GPU computing. SOAP3 [Liu et al., 2011] allows short-read mappings in the GPU using the BWT transform, and supports inexact searches up to 4 errors without indels. SOAP3-dp [Luo et al., 2013] includes the functionality of SOAP3 an also includes a new sequence mapping pipeline that combines pair-end searching and dynamic programming both on the GPU, this provides a functionality similar to Bowtie 2 and BWA. Also, there are FPGA implementations [Xin et al., 2013].

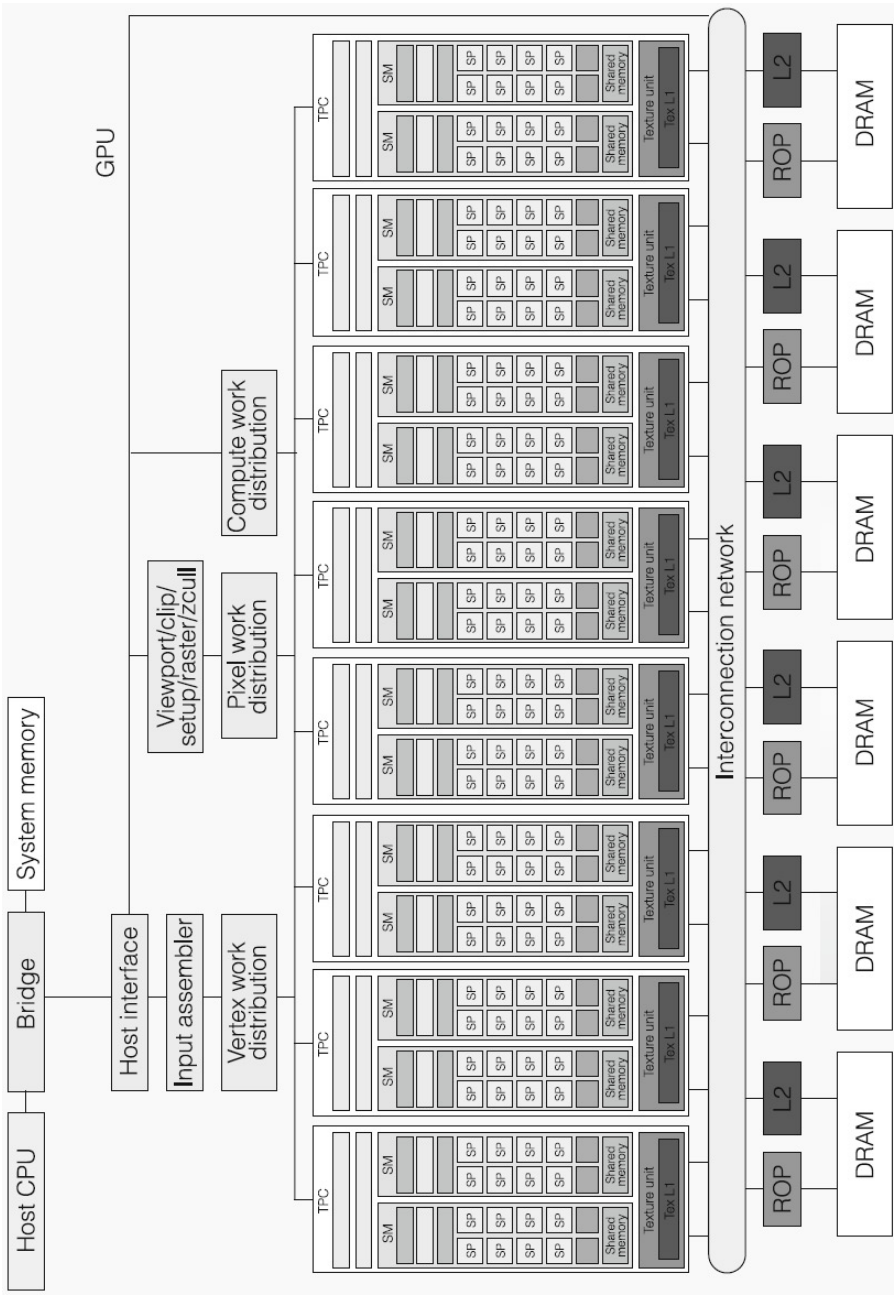


Figure 2.1: Nvidia Tesla GPU architecture

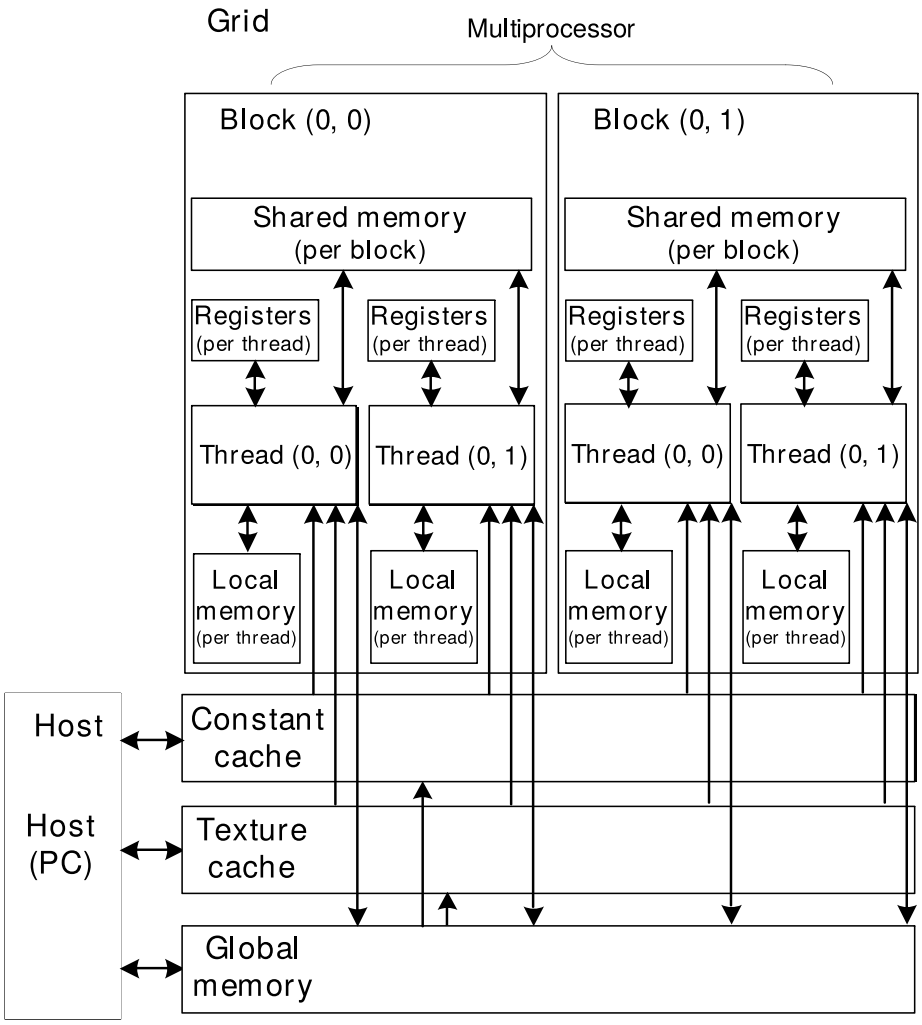


Figure 2.2: Nvidia GPU memory hierarchy

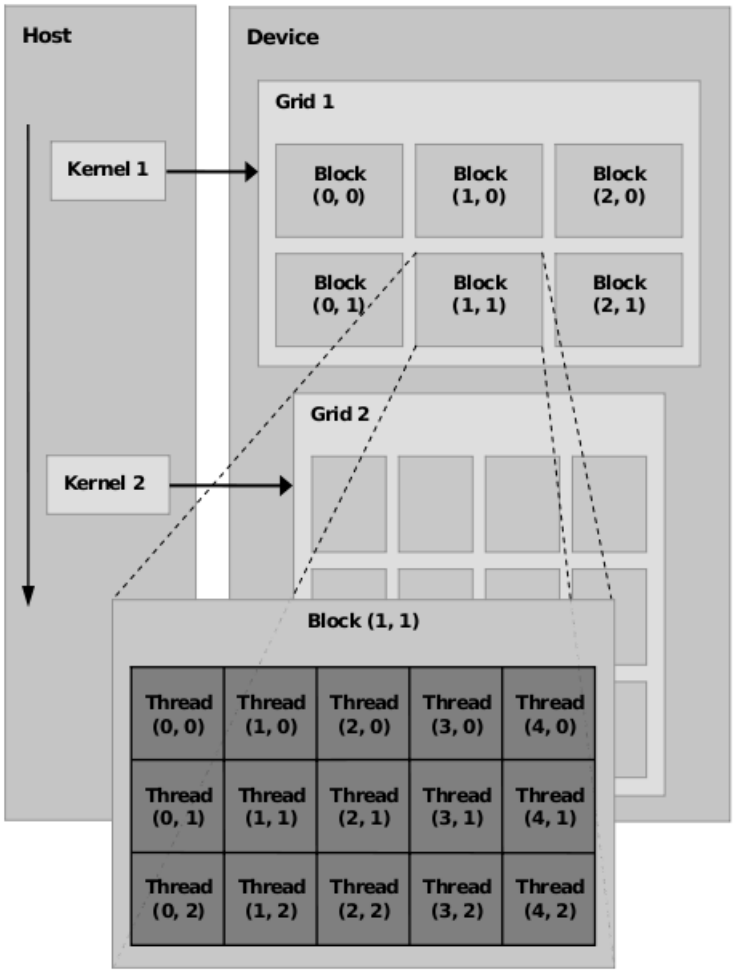


Figure 2.3: Nvidia kernel

Chapter 3

Objectives

The main objective of this thesis is to increase the sensitivity and performance of mapping algorithms based on backward search methods. To achieve this objective, we developed efficient algorithms for exact and inexact alignment capable of taking advantage of different parallel infrastructures, including GPGPU and multi-core processors. Each development in this thesis has a main objective, which implies analysing several other sub-objectives that have guided our work.

The first development focuses on using GPGPUs for solving the exact alignment of short reads with respect to a reference indexed using the FM-Index. This algorithm will be used to find the location of read seeds. This main objective implies:

- To implement a version of the algorithm based on CPUs, including support for inexact searches. This sequential code will be the basis for the GPGPU enabled version of the algorithm.
- To study the data dependencies and parallel symmetry of the BWT searching methods. It is very important to analyse and identify the different parallel flows in the algorithm.
- To define the parallelisation approach of the algorithm considering the restrictions of GPGPUs. According to the study developed in the previous sub-objective, the most effective parallelisation approach could be defined.
- To implement a base version of the algorithm adapted to the restrictions of GPGPUs. Although GPGPU algorithms could be recursive, recursion is only supported on recent cards and iterative versions are more efficient. Moreover, memory size constraints at the different memory hierarchical levels (registry, textures, shared, general, etc.) impose conditions to recursive algorithms that do not fit the present case, since different threads could be at different levels of the recursion, being inefficient.

- To evaluate the different alternatives to reduce the overall response time. There are many features that could be fine-tuned, such as the size of CUDA blocks, the size of the read blocks from the input sequences file or the overlapping between transferring and processing. These features can be evaluated after the algorithm is ported to GPGPU.

- To measure and compare the performance gain with respect to the fastest algorithms in the state of the art and considering the same working conditions. Comparison with respect to representative algorithms using the same approach should be performed for a reasonable large case and considering different working conditions. All the executions should be performed considering the same arguments, ensuring that the output is comparable in all cases.

The second development focuses on extending the exact search parallelisation on GPU to create an hybrid CPU-GPU algorithm. This algorithm allows one error mappings and returns the pair-ends of each read. The sub-objectives for this development further extend the objectives of the previous development:

- To develop various compression and decompression algorithms in order to store and retrieve auxiliary data structures efficiently. The FM-Index generated by the BWT requires tens of gigabytes of main memory during the execution of the search routine. The compression is needed to load the index in the memory of current GPU cards.

- To detect which parts of the computation will execute better on the GPGPU architecture and which ones will execute better on the CPU, separating them into two subroutines.

- To implement a version of the algorithm in which the two previous subroutines are executed on the CPU. This version is compared with the implementation using a single subroutine in order to check the overhead of separating the logic of the program.

- To implement the GPU subroutine using CUDA. We must again fine-tune the CUDA execution variables for the new subroutine.

- To measure and compare the performance and sensitivity with respect to existing algorithms performing a similar task.

The third development consists on an inexact mapping algorithm compatible with different backward search methods and index implementations. We studied the genomic variability, presenting several pruning techniques that speed-up the mapping process. This main objective implies also:

- To develop compression and decompression algorithms for the suffix array vectors, whose size is too big for current desktop workstations.

- An inexact mapping algorithm based on backward search is presented. The algorithm seeks all possible combinations of errors, including insertions, deletions and mismatches.
- To describe the proposed pruning techniques and how they contribute to accelerate the search tree exploration during the mapping process.
- To evaluate the application under different conditions. There are many experimental variables that would affect the performance of the new algorithm, including the size of the reads, the number of errors allowed, the size of the reference genome studied or the size of the spanning tree.
- To compare the alignment sensitivity and execution times of our application with those of other representative tools using the same approach. This comparison should be performed for a reasonable large case, considering the same input arguments and execution environment.
- To use the new algorithm as a preprocessing step for modern sequence mappers. Experiments must be conducted in order to measure the performance gain of modern sequence mappers when using the new algorithm.
- To measure the execution time of the algorithm using an out-of-core implementation of the FM-Index, in order to study its viability on systems with low main memory configurations.

Chapter 4

FM-Index search on hybrid CPU-GPU environments

In this chapter we describe the parallelisation of two sequence mapping algorithms on GPU. The first one covers exact mapping on GPUs [Salavert et al., 2012]. The second one is an hybrid CPU-GPU implementation, which performs inexact mapping with one error and returns the pair-ends of a read [Salavert et al., 2014].

4.1 FM-Index compression

In order to store the FM-Index into GPU we need to compress its data structures, concretely matrix O. The compression technique employed in our experiments is discussed in this section.

Matrix O has a size that depends on the length of the genome times the size of the alphabet. Each element in O is a long integer, so therefore, O is a huge matrix to keep in memory.

In order to compress this matrix, [Li and Durbin, 2009] proposes partially storing vector O in memory, computing the missing values on the fly from B. This technique greatly augments the cost per iteration of the search algorithm. As the main memory of current desktop computers is continuously increasing, we employ an approach with a lower compression ratio than other existing techniques, but also with a lower computational cost increase.

The information of matrix O is split into two matrices O_{count} and O_{disp}. Each row of these matrices stores the information of a different nucleotide, like matrix O does. In our notation matrix O has n columns.

Let O_{count} be a matrix whose elements are bit vectors of size r, such vectors can be stored as integer values. The size of each row is n/r elements of r bits. If the i-th bit of a row is set to 1 this indicates that in the i-th position of B a nucleotide corresponding to the current row symbol appears.

Let O_{disp} be a matrix of integers with size n/r, where each element corresponds to a bit vector in O_{count}. $O_{disp}[a,k]$ contains the number of nucleotides of type $a \in [A,C,G,T]$ before the first bit of $O_{count}[a,k]$.

Following the example of section 2.1.3, the values of O_{count} and O_{disp} are:

$$O_{count} = \begin{pmatrix} 0000 & 0011 \\ 0100 & 0000 \\ 0010 & 1100 \\ 0000 & 0000 \end{pmatrix} = \begin{pmatrix} 0 & 3 \\ 4 & 0 \\ 2 & 12 \\ 0 & 0 \end{pmatrix} \begin{matrix} A \\ C \\ G \\ T \end{matrix}$$

$$O_{disp} = \begin{pmatrix} 0 & 0 \\ 0 & 1 \\ 0 & 1 \\ 0 & 0 \end{pmatrix} \begin{matrix} A \\ C \\ G \\ T \end{matrix}$$

The function in algorithm 4.1 obtains the value of $O[a,i]$ from O_{count} and O_{disp}. Notice that a bit-count operation is employed after a bit shift to extract the number of 1 before position $i \bmod r$ of $O_{count}[a, i/r]$ (the last bit corresponding to $B[0..i]$), then we add this value to the total count of 1 in $O_{disp}[a, i/r]$.

Algorithm 4.1: Matrix O decompression

getOcompValue(a, i, O, r)
{
$pos \leftarrow i/r$
$disp \leftarrow i \bmod r$
$bits \leftarrow bitcount(O_{count}(a, pos) \texttt{ >> } (r - (disp + 1)))$
return $O_{disp}(a, pos) + bits$
}

The *bit-count* operation is implemented at hardware level in many CPU and GPU, being a fast implementation. Matrix O reaches 48 Gigabytes when indexing the *Homo sapiens* genome, requiring only 2 Gigabytes with $r = 64$. An acceptable size in current workstations.

4.2 GPU exact search algorithm

In order to parallelise exact mapping on GPU several reads are analysed at the same time using the multi-threaded architecture of the GPU. Recursive programming is not an efficient approach for GPU parallelisation, so we decided to employ an iterative algorithm in order to improve the speed-up. The algorithm presented in section 2.1.6 establishes the basis for the parallel implementation on GPU.

This section presents a simplification of the real code of the CUDA kernel, outlining the differences and optimisations needed to take full advantage of the Nvidia architecture. The complementary strand variations of the FM-Index described in section 2.1.4 are implemented in GPU following the same principles.

4.2.1 Implementation details

The function in algorithm 4.2 receives the following parameters: W is a vector with the reads to be matched by the kernel in four bases per byte format, nW is a vector with the length of each formatted read and nWe is a vector with the length of the real unformatted read, C and O are the FM-Index of the reference genome and k and l are vectors whose components are initialised to the start values of $\lfloor R \rfloor = 0$ and $\lceil R \rceil = |B| - 1$. *MAXLINECOMP* is the maximum length of an encoded sequence.

Algorithm 4.2: GPU exact search algorithm

exact_iterative_search_gpu(W, nW, nWe, C, O, k, l)
{
$offset \leftarrow blockIdx.x * blockDim.x + threadIdx.x$
if $threadIdx.x < 4$ **then**
  $Cshared[threadIdx.x] \leftarrow C[threadIdx.x]$
end if
_syncthreads()
$k2 \leftarrow k[offset]$
$l2 \leftarrow l[offset]$
// Modulo operator can be expensive
$shift \leftarrow index[nWe[offset] \mod 4]$
_syncthreads()
$b = W[offset * MAXLINECOMP + nW[offset] - 1]$
for $i \leftarrow nW[offset] - 1 \ldots 0$ **do**
  // This loop (j) must be unrolled
  for $j \leftarrow shift \ldots 0$ **do**
    $val \leftarrow (b \gg (j * 2))\&3$ // Unencoded $W[i]$
    $k2 \leftarrow Cshared[val] + O[val, k2] + 1$
    $l2 \leftarrow Cshared[val] + O[val, l2 + 1]$
  end for
  $shift \leftarrow 3$
  if $k2 > l2$ **then break**
end for
_syncthreads()
$k[offset] = k2$
$l[offset] = l2$
}

The algorithm returns two vectors k and l, with the final values of $\lfloor R(W) \rfloor$ and $\lceil R(W) \rceil$ of all the search strings analysed in parallel.

Each thread of the exact search algorithm is responsible of mapping a different read, which is selected by calculating the *offset* of the thread at the beginning of the algorithm.

The decision on which component should be stored in which memory is important for performance. The C vector, which is composed of only 4 elements, is stored in shared memory in order to increase the speed of the algorithm. Vector O must be fetched by the SM from global memory due to its size, and this constitutes the most expensive operation in the kernel execution.

As each element of the C vector is copied into shared memory by a different thread, the *_syncthreads()* function is called to avoid accesses to uninitialised shared memory.

The CUDA kernel is invoked several times. Before each invocation, new reads must be transferred from system main memory to the GPU global memory (vector O is transferred to GPU only once at the beginning of execution). We propose to encode the reads into a four bases per byte format in order to accelerate memory transfers from CPU to GPU. Depending on the CUDA hardware version and the configuration (plain desktop vs high-end workstation) this approach can provide faster results. We noticed that in newer CUDA hardware these memory transfers have been significantly improved, mitigating this issue. Moreover, recent CUDA versions include the *cudaMallocHost* function, which enables faster memory access by improving memory addressing performance. This means that currently we can encode the reads using one byte per base, allowing the treatment of special symbols like N (unknown base). Another consideration is that in our implementation the inner loop is unrolled, processing each byte element in separate code.

Also, we need to know how many bases the last element of the formatted read has. Although the pseudo-code includes a modulo operation, we should avoid it on the implementation since modulo operations can be slow on GPU. However in our case it does not affect the performance of the CUDA kernel implementation.

The $k2 > l2$ condition will end the execution earlier if the substring is not found in the reference genome. When programming in the GPU we normally avoid conditional clauses, but in this case it just ends the analysis of a read, stopping further computation and improving the speed-up. We did not expected a performance increase when adding the $k2 > l2$ condition. Nevertheless, stopping a thread will not stop the rest of threads executed in the same context. This is reflected in section 4.2.3, where the optimum number of threads per block appears to be 32, the lowest value possible. Avoiding this scenario is difficult as we are not able to know the iteration in which the $k2 > l2$ condition will become true, but this optimisation reduces the problem.

4.2.2 Pthread multi GPU version

Many Nvidia CUDA professional solutions come with multiple graphic cards (commonly two). In this section we describe how to enhance the algorithm to take advantage of multi-card systems. In addition, input and output operations can be serialised with the GPU computation.

As we want to take profit of the parallelism that segmentation allows, the whole mapping process has been segmented into several steps (read, load to GPU, match sequences, copy results from GPU, write results...). *CUDA threads* are suited for the management of multiple GPUs at the same time, but lack many features needed for concurrent execution and process synchronisation, this is why we have implemented our solution using *pthreads*. Figure 4.1 illustrates the behaviour of the application, we have split the process into four concurrent CPU threads.

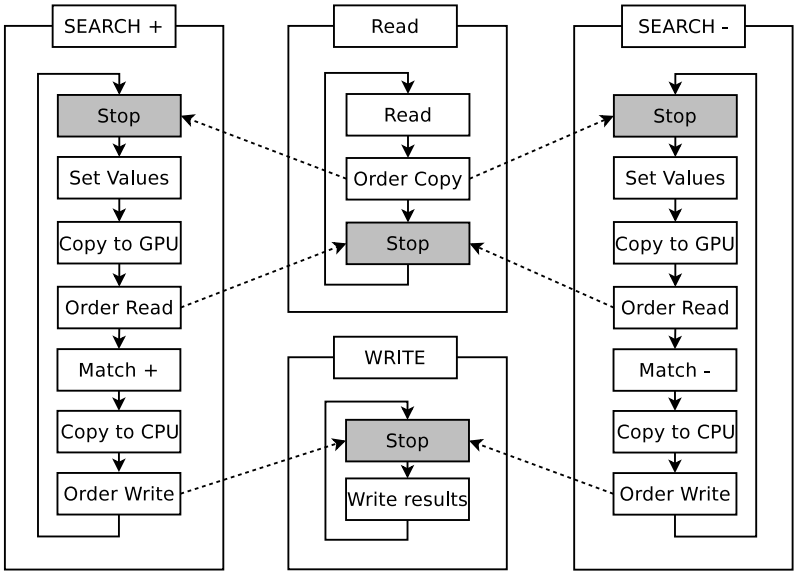


Figure 4.1: Multiple threads in CPU

Concretely, the *read* and *write* threads are responsible of disk input/output operations, loading blocks of reads and saving the results of the mappings respectively. The threads *search +* and *search -* set up some variables, load to GPU the reads provided by the *read* thread, launch forward (+) and reverse (−) strand versions of the mapping kernel on each GPU and return results to main memory. This set up allows to load new reads from disk while simultaneously mapping in multiple cards the sequences read previously. This approach is really effective and we recommend it as an intelligent an effective way of taking full advantage of hybrid CPU-GPU computational power.

4.2.3 Experimental results

All the tests in this section where executed in an Nvidia Tesla S1070 solution, with two Intel® Xeon® E5520 CPU running at 2.27GHz (8 threads), 24GB of RAM and 2 PCI Express interfaces attached to 2 Bull R422E2 nodes. Each of the Bull nodes features 2 Tesla T10 processor GPUs with 30 multiprocessors running at 1.30 GHz (which provide 240 simultaneous CUDA threads). In the experiments we only use one of the Bull nodes with an Intel CPU.

The disk drive is a 2TB Seagate ST3250310NS, with 7200 RPM spin speed, SATA 3Gb/s serial connection, 64MB cache and a sustained data transfer rate of 140 MB/s.

The dataset is composed by reads extracted from the *Drosophila melanogaster* genome with an average length of 100 nucleotides (ranging from 30 to 200 nucleotides). This is the worst case scenario possible because all the nucleotides of each read will be analysed by the backward search implementations.

Profiling of the GPU algorithm

In this section we realise a profiling of the algorithm, measuring separately the different steps of the mapping process. Such steps are the following: read vector O from disk, copy vectors C and O to GPU memory, read the reads from disk, copy the reads to GPU, execute the GPU mapping kernel, copy the results back to CPU main memory and copy the results to disk.

By studying the time that takes to complete each task, we can detect the bottlenecks easily. Also, this study allows us to separate the tasks that rely on the input/output capabilities of the system and are common to all the compared sequencers from those whose performance depends only on the implementation and optimisation of the algorithm.

We executed a single threaded GPU version of the algorithm and aligned 2 million reads with a length ranging from 30 to 200 nucleotides against the *Drosophila melanogaster* genome. The results depicted in figure 4.2 show that loading from disk the vector O consumes about 60% of the execution time. Size of vector O is constant, so we need to accelerate the load or align more sequences in order to take profit of the algorithm.

In any case, if the operative system has enough main memory, the vector O file will remain cached and the following executions will not suffer this initialisation penalty, as it can be seen in figure 4.3. Another solution would be to implement a search service that loads at start-up the vector O and waits for user calls requiring the mapping of reads.

Figure 4.3 also shows that reading and copying the results to disk are the most time consuming parts. This is partially solved in the *pthread* multi GPU version, in which the tasks of reading the reads from disk and executing the GPU mapping kernel are overlapped. Overlapped tasks should consume approximately the same amount of time.

Tesla S1070 has a very good memory transfer compared to other solutions, so the copy of the vector O to GPU is performed quite efficiently. Also, we observed

that the time of the *copy the reads to GPU* and *copy the results back to CPU* steps is negligible. On lower-end GPUs this behaviour is not so optimal, forcing us to encode the reads in a four bases per byte format. This measure improves the performance of the *copy the reads to GPU* step by 3x on such hardware.

The chart reveals that a monetary investment in fast hard drives should be considered when performing mapping. Moreover, using one hard disk to load reads and a second disk drive to write results simultaneously should increase the performance of the application on systems with limited input/output cache. This might be applicable to all the short-read alignment tools studied in this article.

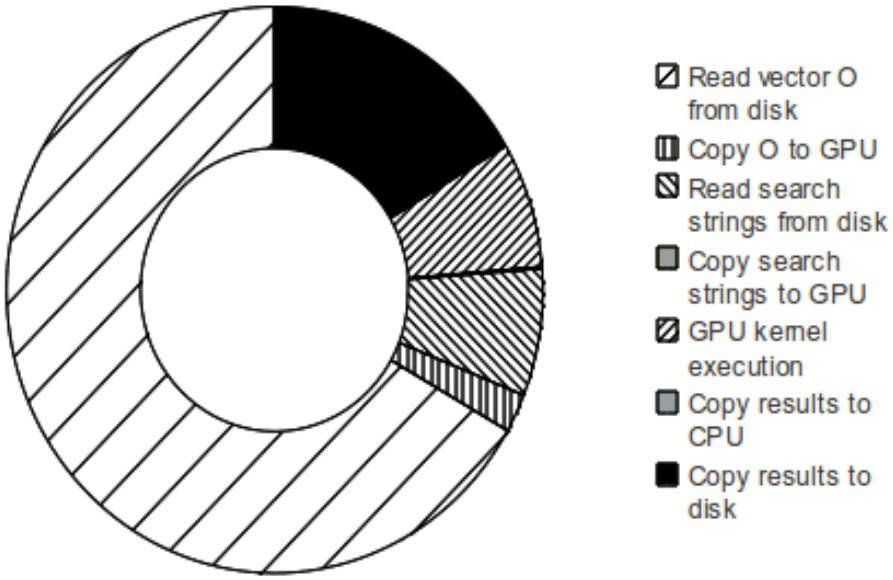


Figure 4.2: Profiling of the algorithm without operative system disk cache

Parameter tweak

After profiling the algorithm we decided to optimise various execution parameters. This task should be performed for every system and experiment configuration, leaving the application ready for a production environment.

The first one is the optimum CUDA block size. CUDA blocks group many GPU threads that are executed in the same SM, sharing its internal memory resources.

In order to obtain the optimum block size for our GPGPU configuration we launched different experiments mapping 2 million of sequences and measuring only the GPU kernel execution time of a single graphic card.

Results in figure 4.4 demonstrate that 32 is the optimum CUDA block size, while bigger blocks slow down the algorithm significantly and smaller sizes are not allowed on our Tesla T10 GPU.

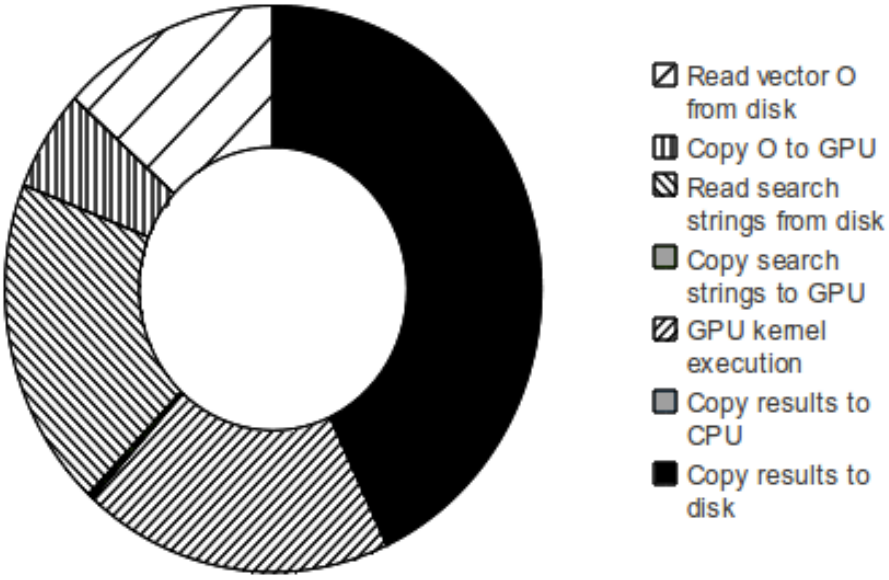


Figure 4.3: Profiling of the algorithm with operative system disk cache

In general, GPGPU memory transfers have a great initial latency. This means that if we apply an algorithm in scenarios that require small data quantities to be transfered to GPU and little amount of GPU processing cycles, then we will suffer a great execution penalty.

Consequently, we present here an study of the algorithm performance against the number of reads to be aligned. These strings are copied into GPU memory before each kernel execution. In this experiment, we measured the overall time of the following three steps of the GPU algorithm: copy the reads to GPU, execute the GPU mapping kernel and copy the results back to CPU main memory. Also, we executed the CPU version of the algorithm on one of the CPU cores and obtained the alignment time for each amount of reads.

Figure 4.5 displays the speed-up of the GPU algorithm in one Tesla T10 when compared to the execution time in one core of our Intel® Xeon® E5520 CPU. We noticed that unless we exceed a data threshold of 4096 reads, we will not take full advantage of the GPU computational power. The maximum number of reads depends on the free space left in the device after copying vector O. The chart reveals that copying more than 256000 reads results in a very low speed-up increase.

Each read has a maximum of 200 elements, encoded in 4 bases per byte format, so each read has a size of 50 bytes. With 4096 reads there is a peak in the speed-up, which means that our graphic card is able to store in cache about 200 MBytes of data. A read block of 4096 reads is a good configuration, but if we have to

process an enormous amount of strings, we can see on the chart that as the read block size is increased so does the speed-up, until reaching a limit.

Also, this study reveals that the GPU algorithm has a convenient 12x speed-up over CPU. For the tools analysed, the maximum speed-up in a current CPU launching 8 concurrent threads has been 6x-7x. This tendency is expected to be maintained as GPGPUs are constantly becoming faster. Moreover, our multi-threaded version of the aligner supports multiple cards, being able to employ one graphic card for the forward strand and other for the reverse strand, reading the search sequences only once.

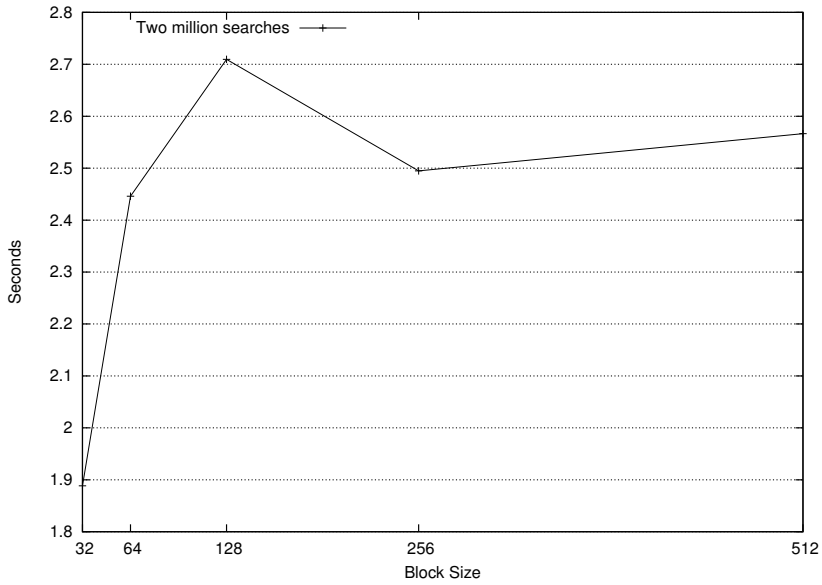


Figure 4.4: Obtaining the optimum GPU block size

Comparison against similar applications

In this last study we compare the performance of similar tools based on the Burrows-Wheeler transform when performing an alignment against the *Drosophila melanogaster* dataset. This section describes the results published in [Salavert et al., 2012]. A more modern comparison against the exact mapping algorithm in SOAP3-dp in modern SSD disks is presented in section 4.3.3.

In this last study we employ 2 GPUs, one for the reverse and another for the strand mapping, against a CPU with 8 concurrent execution threads. We executed all the CPU mappers with 8 threads, allowing parallelism on the CPU. Moreover, all the comparisons were performed activating the 'exact search' map options of every program.

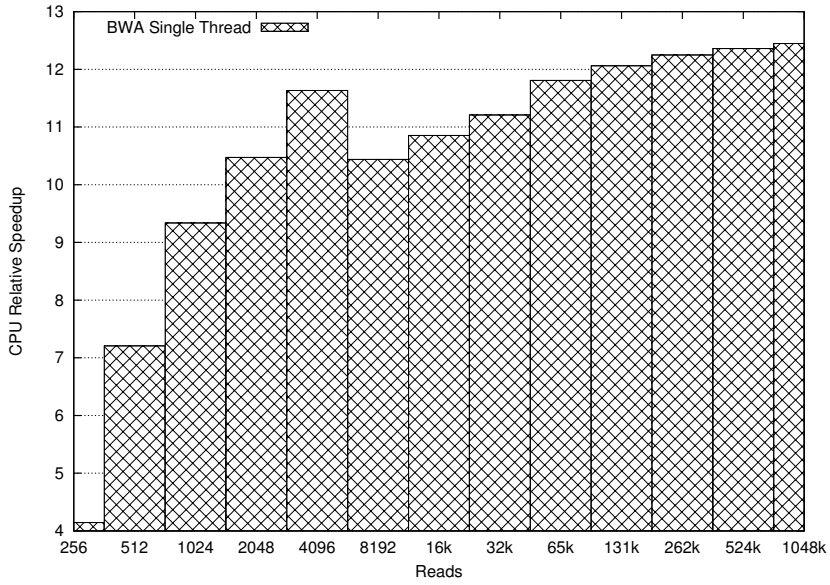


Figure 4.5: Speedup impact of the number of reads per GPU kernel execution

SOAP3 is compared under the same conditions. The execution time of SOAP3 is obtained by adding the execution time of the alignment binary to the view script execution, which converts the binary alignment output into a SAM file. We add these values because all the tools in the comparison write its results directly in a human readable format of bigger size.

Notice that as the internals of the mappers are different, it is difficult to perform a totally equal comparison. We want to show with these tests the effectiveness of the optimisation for exact searches of our GPU implementation. Also, we want to demonstrate that a hybrid CPU/GPU thread model with concurrent CPU threads dedicated to input/output tasks improves disk throughput significantly. It is out of the scope of this article to perform a full feature comparison.

Figure 4.6 shows the resulting times when looking only for the first appearance of each sequence in the reference. This is a bad scenario for an algorithm that returns the interval of all the occurrences in BWT of the correct mappings, but the results are less conditioned by the input/output load.

It can be seen that our GPU implementation is 3x faster compared to Bowtie implementation and 4x faster than SOAP2. Also, the graph shows that SOAP2 loses against Bowtie in this particular configuration of the experiment.

SOAP3 times reflect that our approach is 3x faster due to a better handle of disk operations and the intermediate step needed by SOAP3 to convert the results from binary to SAM format and be comparable to the rest of tools in the study. We discuss this aspect in the comments to the second graph.

Figure 4.7 studies the execution of the algorithm when looking for all the occurrences of each sequence in the reference (-a option). There are short reads in the input files that will produce a considerable amount of mappings in this test. Our tool focuses on obtaining 100% accuracy and its output will contain all these matches, showing the results of all the reverse and strand alignments.

Before discussing the times, we comment the size of the output files of each tool for 40 million of reads, in order to obtain a global view and present an equal comparison.

Our tool writes a file with 190 million alignments, SOAP2 and Bowtie provide around 150 million of results (not the same number) and SOAP3 provides only 80 million of alignments. The output of our program has been modified to be similar to the output of the CPU solutions.

Bowtie and SOAP2 reflect similar times in the chart (being SOAP2 a little faster), as they are based in the same algorithm and write to disk the same amount of data. Our solution achieves a speed-up greater than 2.5 against CPU implementations, while showing more alignments in its results.

SOAP3 is only showing 80 millions of alignments in its results, due to this its execution time can not be compared equally with the rest of the tools without studying the input/output load. Benchmark results for 40 million of reads show that our tool takes less time to write a 27GB file with 190 million of results than SOAP3 writing two files of 9GB with 80 million of results (one for the binary output and another for the file converted into SAM format).

While our tool takes 317 seconds to perform all the alignments, SOAP3 finishes its tasks in 475 seconds. Of these 475 seconds, 190 seconds are spent in the alignment and writing the binary intermediate file and 285 seconds are employed in converting the binary file into SAM format.

As stated in section 4.2.3, although the algorithm shows a 12x speed-up the disk operations consume the major part of the execution time. This is the reason why the execution times between the GPU and the CPU approaches do not differ so much.

We also observed that SOAP3 reduces significantly the time consumed when performing output operations by writing the results to disk using its own binary format. This accelerates SOAP3 when working with the rest of the SOAP pipeline. We consider this a good approach, but needs an extra step to convert the output to SAM format and attain compatibility.

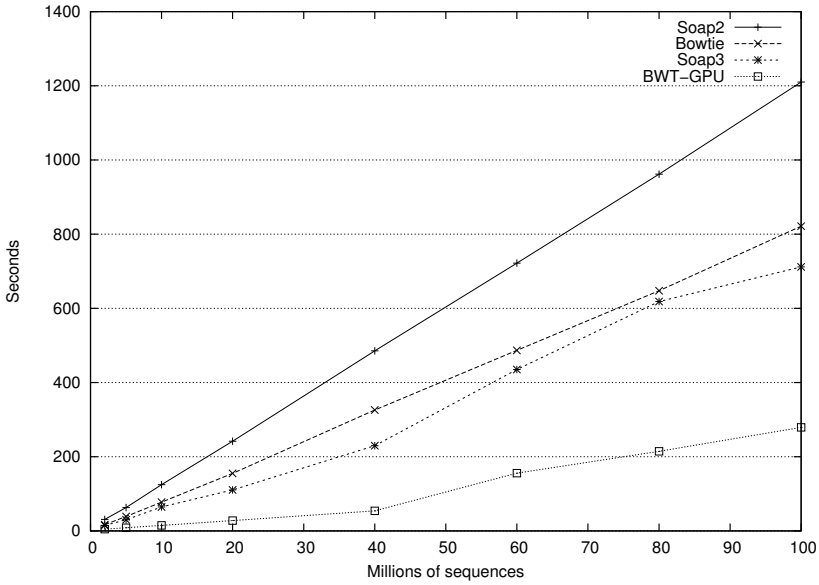


Figure 4.6: Short sequence mapping tools comparison finding only the first occurrence

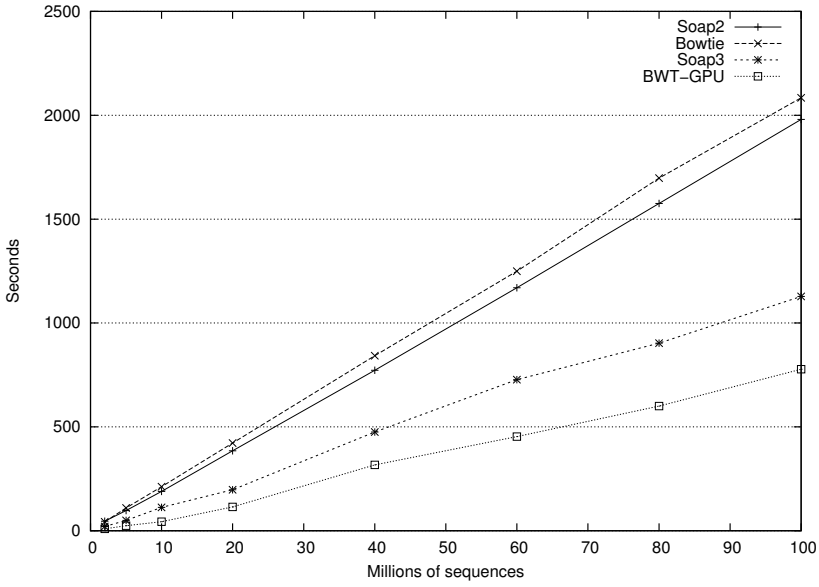


Figure 4.7: Short sequence mapping tools comparison finding all occurrences

4.3 Hybrid CPU-GPU inexact search algorithm

Some bioinformatics applications, like RNAseq analysis [Martínez et al., 2013b] [Martínez et al., 2013a], take advantage of backward search methods to find the pair-ends of a sequence, whose contents are then analysed with a local alignment algorithm.

In the first step of gene expression a particular segment of DNA is copied into RNA by the enzyme RNA polymerase, this is called a transcript. An exon is any nucleotide sequence encoded by a gene that remains present within the final mature RNA product of that gene, after introns have been removed by RNA splicing.

Mapping reads in the context of transcripts is a problem of much higher complexity than simply mapping reads onto the genomic sequence. Eukaryotic transcriptomes are complex, with an average of more than nine transcripts per gene 11 with exons of a 250bp average length that span over several hundreds of kilobases, and more new transcript isoforms are continuously being discovered.

Particularly, splices near the ends of reads can be especially difficult to align, given that a minimum amount of sequence is needed to confidently identify exon boundaries. In addition, new strategies to speed-up runtimes are needed.

The computation of the pair-ends can be effectively done in a GPGPU and then combined in an hybrid pipeline with a lightweight CPU function allowing one error sequence mapping. This special case is an improvement over non-hybrid exact mapping on GPU. Its main advantage is that it allows to find the pair-ends plus one error mappings with very little overhead. Moreover, it constitutes a real hybrid computation approach: some steps are executed on the GPU and the rest on the CPU (algorithm 4.3).

4.3.1 Implementation details

The hybrid inexact mapping method is based on the pseudo-code in algorithms 4.4 and 4.5, which describe the backward direction routines. In these algorithms, functions **exact_backward** and **search_iteration** are defined in algorithms 2.1 and 2.2.

Algorithm 4.3: Sequential execution

```
1: main()
2:    vk, vl, pair ← backward_vector_gpu(W, index)
3:    vk_i, vl_i, pair_i ← forward_vector_gpu(W, index)
4:    results += backward_helper_cpu(W, vk, vl, pair_i, index)
5:    results += forward_helper_cpu(W, vk_i, vl_i, pair, index)
6: end program
```

This design separates the code that will execute better in the *Single Instruction Multiple Thread* (SIMT) architecture of the GPU from the code that will be faster on the CPU.

The **backward_vector_gpu** function (algorithm 4.4) returns the values of all the subsequent $[k, l]$ intervals calculated during an exact search of a read W. These values are stored in vectors vk and vl. Also, it returns the last position with an interval satisfying $k \leq l$ in variable $pair$; this is used to obtain the pair-ends after the execution of the algorithm.

When an interval does not satisfy $k \leq l$, all the remaining elements of the vector are filled with the last non-satisfying values of $[k, l]$. The time consumed by the filling loop in the GPU is insignificant and it is needed to stop the exploration in the **backward_helper_cpu** algorithm when reaching the position where a substring is not present.

Algorithm 4.4: Backward Vector GPU

```
1:  backward_vector_gpu(IN: W, index. OUT: vk, vl, pair.)
2:    [k, l] ← [0, size(index) − 1]
3:    pair ← 0
4:    for i ← |W| − 1 ... 0
5:      [k, l] ← search_iteration([k, l], W[i], index)
6:      if k > l then
7:        pair ← i + 1
8:        break
9:      end if
10:     [vk(i), vl(i)] ← [k, l]
11:   end for
12:   for i ← pair − 1 ... 0
13:     [vk(i), vl(i)] ← [k, l]
14:   end for
15: end function
```

The **backward_helper_cpu** function (algorithm 4.5) performs the inexact search. It receives as parameters the value $pair_i$, which is calculated by the **forward_vector_gpu** function, and the interval vectors vk and vl, which are calculated by the **backward_vector_gpu** function. Before starting the analysis with one error, we check the values of vk and vl at the starting position to include the exact matching case in the *results*. In each iteration the helper function reads the values of the vectors, instead of spending time calculating them.

The analysis starts in the position of the last valid interval of the opposite direction ($pair_i$), but only if it is smaller than the middle position of the read. The $pair_i$ value indicates the longest valid substring of the read starting from the beginning. As we are searching allowing just one error and we know that at $pair_i$ position we must allow a dissimilarity, we will only find mappings if the substring between $pair_i$ and the end of the read is present in the reference. This is similar to the strategy presented in [Li and Durbin, 2009], but in this case we do not need to implicitly calculate a bounding vector.

Algorithm 4.5: Backward Helper CPU

```
1:  backward_helper_cpu(IN: W, vk, vl, pair_i, index. OUT: results
2:    if vk(0) ≤ vl(0) then
3:      results += [vk(0), vl(0)] with [ ]
4:    end if
5:    pos ← min(|W|/2, pair_i) + 1
6:    range ← vl(pos + 1) − vk(pos + 1)
7:    for i ← pos . . . 0
8:      range_p ← range
9:      range ← vl(pos) − vk(pos)
10:     if range_p = range continue
11:     results += exact_backward(W, [vk(i + 1), vl(i + 1)], i − 1, D(i), index)
12:     for b ∈ {A, C, G, T}
13:       [k, l] ← search_iteration([vk(i), vl(i)], b, index)
14:       if k ≤ l then
15:         if b ≠ W[pos] then
16:           results += exact_backward(W, [k, l], i, I(i, b), index)
17:           results += exact_backward(W, [k, l], i − 1, M(i, b), index)
18:         end if
19:       end if
20:     end for
21:   end for
22: end function
```

During the mapping with one error, the algorithm only explores the possible deletions, insertions and mismatches (D,I,M) if the number of suffixes in the current $[vl(pos), vk(pos)]$ interval is different to the values of the last interval ($range_p = range$). When the $range$ value becomes smaller after analysing a symbol, it indicates that we have lost some strings that could be mapped allowing errors in that position. Notice that after a position with an invalid $[k, l]$ interval $range_p = range$ will always be true, as we filled the rest of the vector with the same value in the **vector_gpu** function.

4.3.2 Bounding techniques explanation

Given X = "AGGAGC$" and W = "AGGATC", the forward and backward **vector_gpu** subroutines calculate all the intermediate $[k, l]$ values. If we subtract $l-k$ for all vector positions we obtain the number of appearances of each substring minus one.

Backward results on each position ($\leftarrow$):

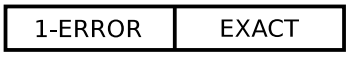
0	1	2	3	**4**	5
A	G	G	A	T	C
-1	-1	-1	-1	-1	0

Forward results on each position ($\rightarrow$):

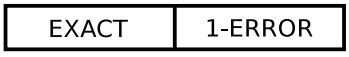
0	1	**2**	3	**4**	5
A	G	G	A	T	C
1	1	0	0	-1	-1

The one error analysis starts from the middle of the read, taking the interval values from the previously calculated vectors. This reduces the variability when looking for possible errors, as strings already have some length when allowing insertions deletions and mismatches [Lam, Li, Tam, Wong, Wu, and Yiu, 2009].

Backward search ($\leftarrow$):



Forward search ($\rightarrow$):



The initial exact search may not reach the middle position with a valid substring. In such cases the analysis in the opposite direction must reach the last correct position before allowing dissimilarities. In the example, during the exact search in backward direction we detect an error at position 4 ($\leftarrow$). In this case we must search without errors in the forward direction until that position, allowing errors only at positions 4 and 5 ($\rightarrow$).

Moreover, only the read positions where the number of results change are studied for possible errors. Looking at the forward results vector these positions are 2 and 4 ($\rightarrow$), needing to search for errors only at position 4 when using all the pruning strategies.

4.3.3 Experimental results

All the executions have been performed in a PC with an Intel(R) Core(TM) i7-3930K CPU running at 3.20GHz speed, 64GB of DDR3 1066 MHz RAM and a Raid 0 of two OCZ-VERTEX4 SSD drives.

The machine has two Nvidia GeForce GTX 680 GPGPUs with 4GB of RAM. In the hybrid CPU-GPU tests the reads are mapped against the Drosophila melanogaster genome. Two million exact matches with lengths 50-200 bps have been extracted from this genome. As all the reads mapped are present in the reference this constitutes a worst-case scenario in which the bounding techniques lose effectiveness.

Profiling of the hybrid algorithm

The test in table 4.1 demonstrates the effectiveness of the hybrid parallelisation model. We mapped a small set of 4000 reads against the *Drosophila melanogaster* genome, extracted from the two million dataset. First, we measure the execution time of the original algorithm on one CPU core, this algorithm only allows one error. Second, we divide the logic of the original algorithm in the two subroutines described (**vector** and **helper**). We observe only a 7% overhead when separating the logic. Also, we see that the **vector** function performs the 92% of the computation. Finally we execute the **vector** subroutine on the GPU, obtaining a 10.5 speed-up (including memory transfers).

As we did not introduced the code of the **helper** function on the GPU, we can still parallelise it on the CPU. This parallelisation will be more effective, since the helper function contains all the conditional execution code which is not suited for the GPU SIMT model. Moreover, the CPU workload will be reduced in normal use as this is a worst-case scenario.

Function call	Microseconds
Original CPU	61868
Vector CPU + Helper CPU	60450 + 5289
Vector GPU + Helper CPU	5786 + 5289

Table 4.1: Effectiveness of the hybrid model

The test in table 4.2 consisted in executing all the function calls of the hybrid algorithm sequentially. We also employed a small set of 4000 reads, in this case to measure the impact of each step in the total execution time. Notice that writing

the results to disk takes almost half of the time. For this reason, while the GPU is working we concurrently write to disk.

Function call	Microseconds	Percentage
disk_read	1045	4.5%
cpu_to_gpu	417	1.8%
vector_gpu	3392	14.76%
gpu_to_cpu	1977	8.6%
helper_cpu	5289	23%
disk_write	10849	47%
TOTAL	22969	100%

Table 4.2: Sequential execution of the hybrid CPU-GPU algorithm

Comparison against similar algorithms

In figure 4.8, we compare the execution times of our GPU implementations for exact and one error mapping against our CPU implementation and SOAP3-dp with the dynamic programming functionality disabled. SOAP3-dp provides a full solution combining backward search and dynamic programming, but we want to validate our backward search implementation against a similar approach.

We employed the 2 million dataset and measured the tools under the most similar conditions possible. As we already demonstrated in [Salavert et al., 2012], we outperform SOAP3-dp when performing exact mapping on GPU.

The hybrid CPU-GPU approach is slightly slower than SOAP3-dp when allowing 1 error, but as it can be seen in figure 4.9 and table 4.3 we are finding 63% more mapping locations due to the support for insertions and deletions with one error (SOAP3-dp is not using dynamic programming). Also, notice that in these tests most of the time is spent in disk writes (almost 50%).

We obtained similar results following different approaches. So, we can conclude that the hybrid model presented in this paper is a valid and different approach for backward search inexact mapping using the GPU.

The main advantage of this algorithm is that it allows to increase the sensitivity of backward search mapping with one error on GPU without decreasing the speed-up provided by the architecture. In addition, when a read is not found the algorithm returns its pair-ends. The pair-ends can be used as seeds for a secondary local alignment algorithm (like Smith-Waterman), which may not be executed on the GPU.

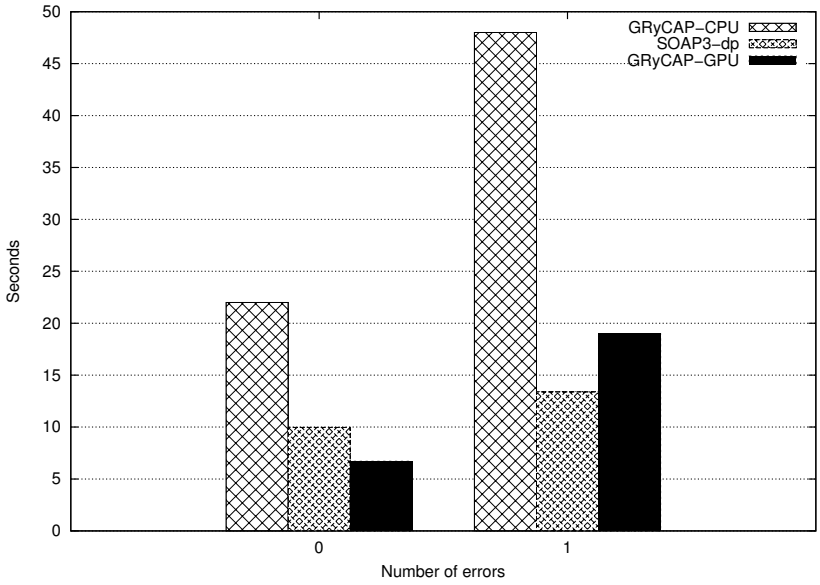


Figure 4.8: Hybrid CPU-GPU, execution times

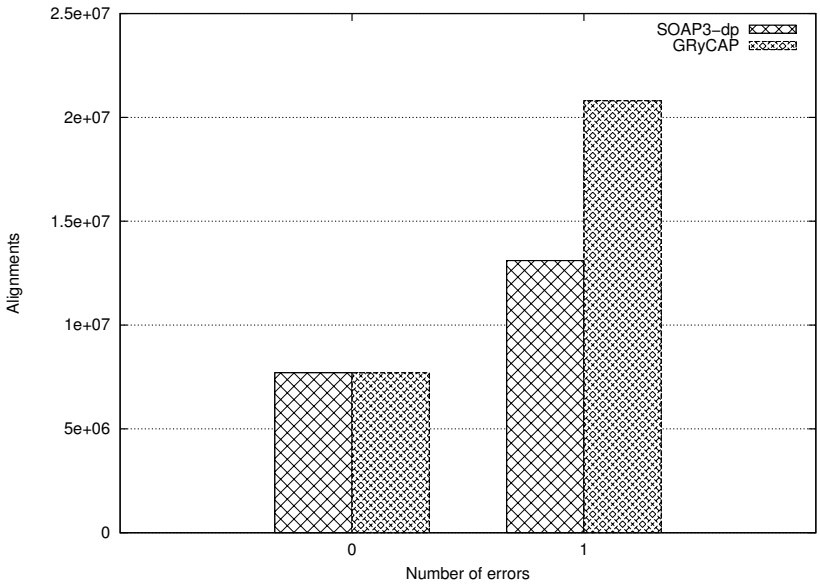


Figure 4.9: Hybrid CPU-GPU, mapping locations found

Table 4.3: Comparison with SOAP3-dp.

	Time	Locations	Sensitivity
GRyCAP-CPU			
0 errors	22s	7698222	1
1 errors	48s	20797694	1
SOAP3			
0 errors	9.97s	7698222	1
1 errors	13.38s	13101388	0.6299
GRyCAP-GPU			
0 errors	6.7s	7698222	1
1 errors	18.98s	20797694	1

4.3.4 Conclusions

GPGUs are clearly a good alternative to provide the computing power required by modern genomic challenges. The capabilities of having close-to-data computing resources is critical for problems dealing with terabytes and, in a short time, petabytes of data. We described the design and implementation of two alignment algorithms based on the FM-Index on GPUs. The algorithms consider only error-free and one error alignments. This may not be adequate for long sequence mapping or in the advent of single nucleotide polymorphisms, mutations or reading errors. However, the algorithms are suited for finding seeds and pair-ends of reads before a more in-depth analysis using dynamic programming algorithms. It is important to remark that these algorithms alone do not constitute a mapping solution, they have to be combined with other dynamic programming algorithms for sequence mapping.

The exact algorithm achieves a good speed-up factor, between 2.5 and 4 with respect to state-of-the-art alignment methods using the FM-Index (figures 4.6 and 4.7), and also considering the same execution conditions (no reading errors, same I/O load and same number of matches requested). Excluding the I/O and using only one graphic card (measuring only strand matching), the speed-up factor of the algorithm is 12x in GPU when compared to CPU (figure 4.5).

The one error hybrid implementation has been tested against SOAP3-dp in a modern SSD disk drive, showing a similar execution time, but finding 50% more

mapping locations (figure 4.9). Excluding the I/O and using only one graphic card (measuring only strand matching), the speed-up factor of the GPU subroutine is 10x when compared to CPU. The speed-up achieved is greater than the GPU parallelisation of the complete one error algorithm.

The performance of these implementations is bounded by the disk performance. We performed tests in HDD and SSD disk drives. For the exact mapping case, I/O operations take two thirds of the total processing time (using a HDD). For the hybrid one error algorithm, I/O operations take half of the total processing time (using a SSD). However, tendency on I/O drive optimisation has a positive effect that increases the algorithm execution speed.

The sensitivity of the algorithms has been studied and results are even better. Since the algorithms presented in this thesis do not take any heuristic assumption that could prevent sequences from being found in the reference, we find more mapping locations.

Notice that the use case proposed (the exact alignment of short reads with respect to known reference genomes), could reasonably be solved in a large extent by exact searching, due to the short frequency of errors in short reads and the comparison against known references. The GPU algorithms can be used in a first step to quickly remove the matchings without errors, lightening the workload of a secondary inexact mapper based on dynamic programming.

Finally, the inexact alignment problem is generally tackled in CPU by combining the exact alignment algorithm with a depth first tree exploration strategy [Li and Durbin, 2009]. Other solutions try to support inexact searches on GPU parallelising these methods, but as the search tree grows exponentially with the number of errors supported (breaking GPU parallelism) there is an error limit conditioned by the hardware. Finding alternative strategies to enable fast inexact searching on CPU with the aid of GPU is the key for further research achievements. The one error mapping algorithm presented in this thesis is an original research effort in this direction.

Chapter 5

Faster and more accurate inexact mapping using advanced tree exploration on backward search methods

In this chapter we describe a backward search algorithm for inexact mapping supporting any number of errors. We support all type of errors (insertions, deletions and mismatches) in all read positions. This algorithm is intended as an extra preprocessing step before the seed location phase of current sequence mapping tools.

The algorithm relies in several techniques, which together deal with the variability of the genomic data. None of these strategies decrease the sensitivity of the search. Moreover, many of these optimisations are standalone concepts, this means that other algorithms could benefit from the optimisations compatible with their approach.

In this study, we used both our own implementation of the FM-Index and *csalib* [Sadakane, 2010] implementation. In *csalib*, the data structures are not loaded into main memory [Mäkinen et al., 2004], but accessed from disk by demand using *mmap*. Such properties may be useful in memory demanding tasks, like mapping against big genomes.

5.1 Suffix Array compression

In order to deal with large genomes, like the human genome, we need to compress the BWT data structures, concretely vectors S and R. In this section we describe how to compress and expand the suffix array and the inverse suffix array vectors.

5.1.1 Vector S compression

Vector S allows to obtain the original positions in the reference of the matchings and is often referred as the SA.

Let $Scomp$ be an integer vector (32 bit are enough to index all the positions of the human genome). Let n be the size of vector S. $Scomp$ size is n/r where r is the compression ratio. Each element of $Scomp$ satisfies that $Scomp[k] = S[k * r]$.

It is possible to reconstruct S from $Scomp$ following the principles in [Grossi and Vitter, 2005], as stated in [Li and Durbin, 2009]:

$$S[k] = S[(\Psi^{-1})^{(j)}(k)] + j$$

Where Ψ^{-1} is the inverse compressed suffix array and $(\Psi^{-1})^{(j)}$ denotes applying it for j times:

$$\Psi^{-1}(i) = C(B[i]) + O(B[i], i + 1)$$

As it can be seen in algorithm 5.1, in order to obtain $S[k]$ we repeatedly apply Ψ^{-1} until we reach some j value which satisfies that the position $S[(\Psi^{-1})^{(j)}(k)]$ is a multiple of r whose value is stored in $Scomp$.

Algorithm 5.1: Vector S decompression

```
getScompValueB(k, Scomp, C, O, n, r)
{

i ← k
j ← 0

while i mod r ≠ 0 do
  if B[i] = −1 then
    i ← 0
  else
    i ← C[B[i]] + O[B[i]][i + 1] {Col. 0 is -1 in O}
  end if
  j ← j + 1
end while

return (Scomp[i/r] + j) mod (n − 1)

}
```

Vector S size reaches 12 Gigabytes when indexing the human genome, this size is divided by r using this compression routine. However, the number of iterations required to compute each position is not fixed, and varies between 1 and r.

For example, given the suffix array of the reference X = "AGGAGC$", we want to find the value of $S[5]$, but we only know the values of $S[0] = 6$ and $S[6] = 1$ because $r = 6$. We can employ algorithm 5.1 to obtain the value of $S[5]$ using the Ψ^{-1} function:

$$
\begin{array}{rcccccccc|l}
 & & \$ & A & G & G & A & G & C & \\
\Psi^{-1}(5) = 1 & \rightarrow & A & \underline{G} & \underline{C} & \underline{\$} & A & G & G & j = 1 \\
 & & A & G & G & A & G & C & \$ & \\
 & & C & \$ & A & G & G & A & G & \\
\Psi^{-1}(\Psi^{-1}(5)) = 4 & \rightarrow & G & A & \underline{G} & \underline{C} & \underline{\$} & A & G & j = 2 \\
k = 5 & \rightarrow & \underline{G} & \underline{C} & \underline{\$} & A & G & G & A & j = 0 \\
\Psi^{-1}(\Psi^{-1}(\Psi^{-1}(5))) = 6 & \rightarrow & G & G & A & \underline{G} & \underline{C} & \underline{\$} & A & j = 3 \\
 & & & & & & & & \mathbf{B} &
\end{array}
$$

Each time that the function Ψ^{-1} is recursively applied we obtain the position in the SA of the suffix that is shifted one symbol to the right in relation with the current one. We know that the position of this shifted suffix in the reference is the position of the previous suffix minus one, this is why we keep the number of recursive calls in variable j and add it at the end. In this case:

$$S[5] = S[\Psi^{-1}(\Psi^{-1}(\Psi^{-1}(5)))] + 3 = S[6] + 3 = 1 + 3 = 4$$

which is the position in X of substring "GC$", the suffix in $S[5]$.

As stated in section 2.1.3, given vector B and the position of the suffix ending with '$' (the position of the '$' symbol in B) we can obtain all the values of vector S. This also applies for vector R decompression, discussed in next section.

5.1.2 Vector R compression

Vector R size also reaches positions in the SA of the reference substring and is often referred as the ISA. The formulas used to compress the inverse suffix array rely on the same Ψ^{-1} function used in suffix array compression.

Let *Rcomp* be an integer vector (32 bits are enough to index all the positions of the human genome). Let n be the size of vector R. *Rcomp* size is n/r where r is the compression ratio. Each element of *Rcomp* satisfies that $Rcomp[k] = R[k * r]$.

Vector R is useful to change the direction of the search without employing bidirectional methods [Lam et al., 2009], but duplicates the memory needs of the algorithm. We discuss how to change the direction of the search in section 5.3.

It is possible to reconstruct R from *Rcomp* by designing an algorithm that applies the compression routines found in [Sadakane, 2003] and [Li and Durbin, 2009]:

$$R[k] = (\Psi^{-1})^{(j)}(R[k + j])$$

Where Ψ^{-1} is the inverse compressed suffix array and $(\Psi^{-1})^{(j)}$ denotes applying it for j times:

$$\Psi^{-1}(i) = C(B[i]) + O(B[i], i+1)$$

We calculate j as follows:

$$j \leftarrow (r - (k \bmod r)) \bmod r$$

As it can be seen in algorithm 5.2, with the j value we are able to obtain the value of the first $R[k+j]$ position, as it is a multiple of r stored in *Rcomp*. After that, we apply Ψ^{-1} for j repeated times to obtain $R[k]$.

Algorithm 5.2: Vector R decompression

```
getRcompValueB(k, Rcomp, C, O, n, r)
{

j ← (r − (k mod r)) mod r
aux ← k + j

if aux < n then
  i ← Rcomp[aux/r]
else
  i ← Rcomp[0] {Special case}
  j ← n − k
end if

while j > 0 do
  if B[i] = −1 then
    i ← 0
  else
    i ← C[B[i]] + O[B[i]][i + 1] {Col. 0 is -1 in O}
  end if
  j ← j − 1
end while

return i

}
```

Vector R reaches 12 Gigabytes when indexing the human genome, this size is divided by r using this compression routine. Also, the number of iterations to recover the positions varies between 1 and r.

For example, given the suffix array of the reference X = "AGGAGC$", we want to find the value of $R[4]$, but we only know the values of $R[0] = 2$ and

$R[6] = 0$ because $r = 6$. We can employ algorithm 5.2 to obtain the value of $R[4]$ using the Ψ^{-1} function.

First, we need to obtain the value of j, in this case:

$$j = (r - (k \bmod r)) \bmod r = (6 - (4 \bmod 6)) \bmod 6 = 2$$

The value of j represents the difference between the position in the reference X for which we want to obtain the SA position and the next reference position with its SA position stored in R. In this case we look for the SA position of the substring starting at $X[4]$ "C$", as we do not have the value of $R[4]$ stored in memory we apply $(\Psi^{-1})^{(j)} = (\Psi^{-1})^{(2)}$ function to the value at position $R[4 + j] = R[6]$.

$$\begin{array}{rcccccc|c}
R[k] = R[6] = 0 & \rightarrow & \underline{\$} & A & G & G & A & \underline{G} & \underline{C} \\
 & & A & G & C & \$ & A & G & G \\
 & & A & G & G & A & G & C & \$ \\
\Psi^{-1}(0) = 3 & \rightarrow & \underline{C} & \underline{\$} & A & G & G & A & \underline{G} \\
 & & G & A & G & C & \$ & A & G \\
\Psi^{-1}(\Psi^{-1}(0)) = 5 & \rightarrow & \underline{G} & \underline{C} & \underline{\$} & A & G & G & A \\
 & & G & G & A & G & C & \$ & A \\
\hline
 & & & & & & & & \mathbf{B}
\end{array}$$

Each time that the function Ψ^{-1} is recursively applied we obtain the position in the SA of the suffix that is shifted one symbol to the right in relation with the current one. As we obtained the position in the SA of $X[4 + j] = X[6]$ with $R[6]$, we have to apply Ψ^{-1} two times to obtain the position in the SA of $X[4]$. In this case:

$$R[4] = \Psi^{-1}(\Psi^{-1}(R[4 + 2]) = \Psi^{-1}(\Psi^{-1}(0) = 5$$

which is the position in the SA of substring "GC$" the substring starting at position $X[4]$.

As stated in section 2.1.3, given vector B and the position of the suffix ending with '$' (the position of the '$' symbol in B) we can obtain also the values of vector R. As both vectors S and R can be obtained by iteratively applying the Ψ^{-1} function we can calculate both at the same time, using a single call to function Ψ^{-1}. However, the most time-consuming part of this procedure is writing the values of these vectors to main memory in a non-sequential order.

5.2 Search tree exploration prototype

When performing inexact mapping a recursive approach over a search tree can be employed [Li and Durbin, 2009]. This analysis depends on three factors. The first one is the current state of the backward search, each path from the root to any node of the tree represents a sequence of symbols that has lead to a different $[k, l]$ interval. The second one is the specific variability of the reference genome studied, on the initial tree levels only few symbols have been processed, so branches for all possible errors widely satisfy $k \leq l$ and grow uncontrollably. The third one is the

uniqueness of each read, which can be initially studied to determine the minimum number of errors needed to map it.

We developed a search tree exploration algorithm that greatly reduces the tree growth during inexact search. It employs a faster iterative approach, using several lists to store partial results. These lists store previous results, next results to explore and final results. We named these lists rl_p, rl_n and rl_f in the pseudocode.

Algorithm 5.3 is a simplified prototype of the final approach. It lacks the bounding techniques described in next section, so its execution is not as efficient. We use it to explain the behaviour of the complete algorithm as it is based on the same subroutines: a selective **exact** search procedure that detects and annotates the positions where it is worth to study sequence errors and a conservative **branch** procedure with specific rules for genomic data.

Algorithm 5.3: Search prototype

```
1: prototype (IN: W, index, errors. OUT: rl_f.)
2:   rl_p.add([0, size(index) − 1] at |W| − 1 with [ ])
3:   for errors . . . 1
4:     rl_n, rl_f ← exact(W, True, 0, rl_p, index)
5:     rl_p ← branch(W, rl_n, index)
6:   end for
7:   rl_f ← exact(W, False, 0, rl_p, index)
8: end function
```

The execution starts by adding a single partial result to the previous results list. This first single result contains the initial interval, no symbols of W analysed and an empty error list. After that, the **exact** and **branch** subroutines are executed alternatively, increasing the partial results stored in the previous and next lists. At the end, the last **exact** call returns the final results.

The **exact** subroutine (algorithm 5.4) input variables are *inexact*, which indicates if it must perform an exact search or allow errors, and *last*, which indicates the last symbol to analyse (in this case the full string). It takes partial results from rl_p and analyses them, inserting in rl_n new partial results for each position requiring branches. These partial results denote other possible mappings in the reference that differ from the current read at the detected positions. In order to detect these positions we demonstrate the following condition.

Let $[k', l'] \leftarrow$ **search_iteration**$([k, l], a)$ be two subsequent SA intervals in a forward search, where $a = W[i]$. We define $res = l - k + 1$ and $res' = l' - k' + 1$, these values indicate respectively the number of appearances of substrings $V = W[0 : i - 1]$ and $V' = W[0 : i]$ in the reference X. We demonstrate that for any read W and any BWT index $res' \leq res$ is always true. If V appears res times in X, then $res' \in [0, res]$, because V is the prefix of V' ($V' = Va$).

Algorithm 5.4: Exact Subroutine

```
1: exact(IN: W, inexact, last, rl_p, index. OUT: rl_n, rl_f)
2:   for r ← rl_p[1 ... rl_p.size]
3:     [k', l'] ← r.[k, l]
4:     res' ← l' − k'
5:     for i ← r.position ... last
6:       [k, l] ← [k', l']
7:       if k > l break
8:       [k', l'] ← search_iteration([k, l], W[i], index)
9:       res ← res'
10:      res' ← l' − k'
11:      if res' < res and inexact = True then
12:        rl_n.add([k, l] at i with r.er)
13:      end if
14:    end for
15:    if k' ≤ l' then
16:      rl_f.add([k', l'] at last − 1 with r.er)
17:    end if
18:  end for
19:  emptyStack(rl_p)
20: end function
```

We observed that the number of potential results remains stable ($res = res'$) and near its final value after several **search_iteration** (15 in *Drosophila melanogaster* and 31 in *Homo sapiens*). The positions with possible errors are the ones in which $res' < res$, showing that the interval has lost reads that could be mapped allowing errors. This pruning is based on the current state of the search.

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

Figure 5.1: Number of partial solutions. Values of res during a forward search of string "AGGATC" against the reference "AGGAGC$"

Figure 5.1 shows the res values of the substrings of "AGGATC" during a forward search against the reference "AGGAGC$". The possible error branches should only be studied at positions 2 and 4 of the string, the substrings "AG" and "AGGA" where the values of res change. For substring "AG" this means that there is an alternative solution with "AGC" in the reference, instead of "AGG". For substring "AGGA" the alternative solution is "AGGAG" instead of "AGGAT". When $res = -1$ the string does not belong to the reference ($k > l$). When studying

larger reads, the pruning is not effective in the first iterations, but later on the values of res stabilise.

This technique also eliminates redundant results, i.e. when mapping string "TGGGGGA" into reference "...TGGGGA...", we would obtain five different results, one for each possible deletion of any of the 'G' nucleotides. Now, the $res' < res$ condition is only true in the last 'G', obtaining a single deletion as result.

The **branch** subroutine (algorithm 5.5) extracts partial results from rl_n, generates new branches by studying the outcomes of adding different errors at $r.position$ and stores the valid ramifications in rl_p. The notation $p.\{D, I(b), M(b)\} : r.er$ indicates that we add a deletion, insertion or mismatch with symbol b in position p to the list of errors of the current partial result $r.er$. Unlike this approach, algorithms that use backward search only for seeding do not need to obtain alignment information before the dynamic programming alignment phase.

As the branches that do not satisfy $k \leq l$ are eliminated, this pruning depends on the variability of the reference genome.

Algorithm 5.5: Branch Subroutine

```
1:  branch(IN: W, rl_n, index. OUT: rl_p)
2:    for r ← rl_n[1 ... rl_n.size]
3:      p ← r.position
4:      rl_p.add(r.[k, l] in p − 1 with p.D : r.er)
5:      for b ∈ {A, C, G, T}
6:        [k', l'] ← BWiteration(r.[k, l], b, index)
7:        if k' ≤ l' then
8:          if b ≠ W[pos] then
9:            rl_p.add([k', l'] in p with p.I(b) : r.er)
10:           rl_p.add([k', l'] in p − 1 with p.M(b) : r.er)
11:         end if
12:       end if
13:     end for
14:   end for
15:   emptyStack(rl_n)
16: end function
```

Matches have precedence over insertions, so insertions of the currently matching symbol are not studied. Moreover, all the pairs of consecutive errors are analysed in order to further reduce the growth of the search tree, forbidding those equivalent to a single error.

For simplicity, the following restrictions are not described in the pseudo-code of the branch subroutine:

- Pairs of consecutive insertions and deletions (`I-D` or `D-I`) are not allowed. Inserting a nucleotide and immediately removing it has no significance. This

rule avoids *indel* chains like `I-D-I-D-I`. Also, `I-I-I-D-D` chains are avoided, as `I-M-M` chains are equivalent.

- A mismatch after an insertion is not allowed if the original nucleotide in the mismatch position is the same as the nucleotide of the insertion. In such cases `I-M` is equivalent to `I`.
- A mismatch after a deletion is not allowed if the nucleotide of the mismatch is the same as the nucleotide eliminated by the deletion. In such cases `D-M` is equivalent to `D`.
- An insertion after a mismatch is not allowed if the nucleotide of the insertion is the same as the original nucleotide in the mismatch position. In such cases `M-I` is equivalent to `I`.
- A deletion after a mismatch is not allowed if deleted nucleotide is the same as the nucleotide of the mismatch. In such cases `M-D` is equivalent to `D`.

After applying these rules the growth of the spanning tree is halved. In addition, inexact searches with up to 2 errors will not produce repeated results.

5.3 Search tree exploration complete algorithm

The bounding strategies of **branch** and **exact** are based on $k \leq l$ and $res' < res$ conditions, being not effective with few symbols analysed. The final algorithm depicted in figure 5.2 and algorithm 5.6 solves this issue with no penalty in sensitivity.

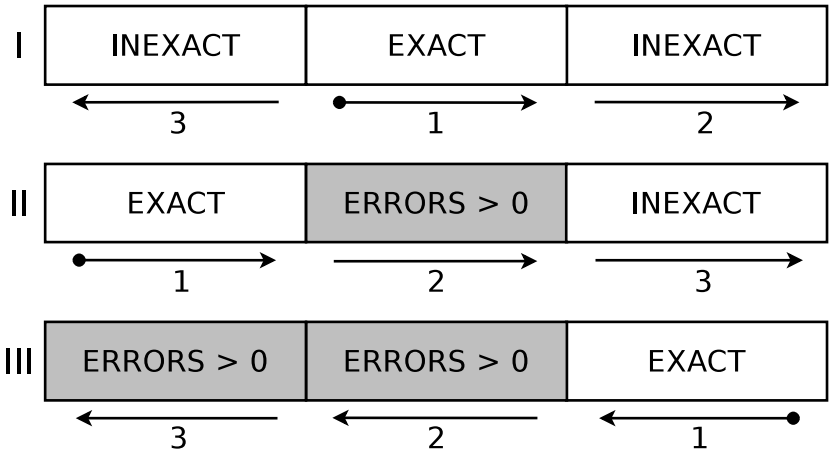


Figure 5.2: Complete inexact search algorithm. Example for 2 errors.

For the complete algorithm to work we need backward and forward versions of the **branch** and **exact** subroutines (**branchB** and **branchF**) and a new function to change the direction of the search in the partial results that reach the end of the read (**change_direction**).

The complete algorithm is based on the work presented in [Lam et al., 2009], with improvements to avoid repeated computations and extended support for more than two errors with insertions, deletions and mismatches. We do not use bidirectional BWT, as it may not be so efficient with backward search methods based on SA that need a binary search in each iteration. Moreover, keeping track of the reverse and strand SA intervals also increases the number of memory writes when managing the partial results. Nevertheless, a bidirectional method would reduce the memory requirements of the algorithm.

In order to allow e errors we conceptually divide the read in $e+1$ segments and perform $e+1$ steps. Figure 5.2 shows an example for two errors ($e = 2$): the steps are in roman numerals, the arrows indicate the direction of the analysis in each segment and the arrow numbers the order in which the segments are analysed.

The first segment of each step is analysed using an exact search, if the segment is not found in the reference the whole step is skipped. After this initial exact analysis the pruning methods are effective and the remaining segments can be analysed with inexact search. Due to this, the number of errors allowed by the algorithm depends on the length of the read and the minimum segment size (31 for human genome and 15 for Drosophila melanogaster, $segsize$ in algorithm 5.6). Also, it is worth to mention that these exact segments could be reused later as seeds to find local alignment regions.

In step I (algorithm 5.6), after analysing block of arrow 2 the direction of the search is changed. As we have the SA and ISA of the reference and its reverse in memory we can use the **change_direction** subroutine to change the direction of the search.

The **change_direction** function takes the $[k, l]$ interval of a partial result and obtains the $[k', l']$ interval in the SA of the reverse reference. For each position $i \in [k, l]$ we obtain the original positions in the reference with the S vector. Then, we correct this position with the error and current position displacement. Finally, the R_i vector containing the positions in the SA of the reverse reference is used to obtain the values in $[k', l']$. These values are not sorted, so we have to search all the values and select the maximum and the minimum.

$$k' = R_i[S.siz - S[k] - (r.end - r.start + 2) - err]$$

In practical use the size of the SA intervals when changing direction is very small (almost always equal to 1), so this computation does not affect the performance. Notice that at each step the starting block and the direction maximises the number of symbols analysed before a direction change.

In step II and III, during the analysis of partial results in the blocks marked with $errors > 0$ the next partial results must contain at least one error within the

Algorithm 5.6: Complete Inexact Search (Step I figure 5.2)

```
1: InexactSearch(W, index, rl_p, rl_n, rl_pi, rl_ni, rl_f, segsize)
2: {
3:   ...
4:
5:   e ← (size(W)/segsize) − 1
6:   seg ←The last position of the central block
7:
8:   r ← [0, size(index) − 1] at |W| − 1 with [ ]
9:   r, rl_f ← exactF(W, False, seg, r, index)
10:
11:  if r.k ≤ r.l then
12:    add_result(r, rl_pi)
13:    while e > 0
14:      seg ← The last position of the current segment
15:      rl_ni, rl_p ← exactF(W, True, seg, rl_pi, index)
16:      change_direction(rl_p, index)
17:      rl_n, rl_f ← exactB(W, True, seg, rl_p, index)
18:      rl_pi ← branchF(W, rl_ni, index)
19:      rl_p ← branchB(W, rl_n, index)
20:      e ← e − 1
21:    end while
22:
23:    rl_ni, rl_p ← exactF(W, False, seg, rl_pi, index)
24:    change_direction(rl_p, index)
25:    rl_n, rl_f ← exactB(W, False, seg, rl_p, index)
26:
27:  end if
28:
29: ...
30: }
```

block, due to this when the analysis reaches the end of the block the partial result with no errors in that segment is not added to the next results list. The logic of this bounding can be added to the **exact** subroutine with an extra condition in line 15 of algorithm 5.4 to discard exact segments. This behaviour avoids repeated computation as the segments with no errors where already checked in the exact blocks of previous steps. Notice that the order of the steps maximises the appearances of $errors > 0$ blocks immediately after the exact block.

The tree exploration is a combination of *breadth first search* (BFS) and *depth first search* (DFS), being a bit more complex than the pseudo-code in algorithm 5.6. The initial levels are explored using BFS while the last ramification, where many partial results are discarded and only a few final results are kept, is explored using DFS. This avoids a great amount of memory writes in the partial results lists. Using DFS in the initial levels has little effect in the overall performance.

The last bounding technique depends on the uniqueness of the read and is based on what was presented in [Li and Durbin, 2009], but adapted to the new algorithm and the direction changes. Before each step of the complete algorithm, a vector D with the same size of the read is built. The D vectors contain an approximation of the number of errors needed to map the read at each position, based on the sub-strings of X present in W.

In algorithm 5.7, vector D for step I of the complete algorithm is obtained. It receives as parameters the read W and the *start* and *end* positions of the exact segment of the current step, returning vector D. The values of D for the exact segment are already calculated ($D[start \dots end] = 0$). The direction of the calculation is changed in the second loop. In the last loop the values are adjusted as we calculate vector D in the same direction of the search, which benefits caching. The condition "is not a substring" is implemented using a **search_iteration**.

In order to implement this bounding, an extra condition at line 11 of algorithm 5.4 must check the current number of errors against the value of $D[i]$.

5.4 Compatibility interface

The implementation of the search tree exploration algorithm is based on replaceable components. So we added support for a new backward search runtime using *csalib* interfaces. The *csalib* library provides several backward search implementations which are either based on the FM-Index or SA. The main difference of *csalib* with our current implementation is that the data structures are not loaded into main memory [Mäkinen et al., 2004], but accessed from disk by demand using *mmap*. Such properties may be useful in memory demanding tasks, like mapping against big genomes. We benchmark our inexact mapping algorithm, comparing our in memory implementation of the FM-Index with the *csalib* out-of-core implementation for DNA.

Algorithm 5.7: Calculate D forward (Step I figure 5.2)

```
1:  calculateDF (IN: W, start, end. OUT: D.)
2:    D[...] ← 0
3:    z ← 0
4:    j ← start
5:    for i ← end + 1 ... |W|
6:      if W[j, i] is not a substring of X then
7:        z ← z + 1
8:        j ← i + 1
9:      end if
10:     D[i] ← z
11:   end for
12:
13:   j ← start − 1
14:   for i ← start − 1 ... 0
15:     if W[j, i] is not a substring of X then
16:       z ← z + 1
17:       j ← i + 1
18:     end if
19:     D[i] ← z
20:   end for
21:
22:   last ← D[0]
23:   for i ← 0 ... |W|
24:     D[i] ← last − D[i]
25:   end for
26: end function
```

Our CPU algorithm is compatible with any backward search implementation providing the following interfaces:

$$[k', l'] \leftarrow \textbf{search\_iteration}([k, l], symbol, index) \quad (5.1)$$

$$position \leftarrow \textbf{get\_sa}(suffix, index) \quad (5.2)$$

$$suffix \leftarrow \textbf{get\_isa}(position, index) \quad (5.3)$$

$$size \leftarrow \textbf{size\_sa}(index) \quad (5.4)$$

This simplicity eases portability. Function 5.1 is a single backward search iteration. In a single iteration we have an initial $[k, l]$ interval in the suffix array and after analysing a *symbol* we end with an equal or narrower $[k', l']$ interval. This function must also work in forward direction by only changing the *index*.

Function 5.2 returns the original position in the reference of a given suffix array position, while function 5.3 is its inverse. Finally, function 5.4 returns the size of the suffix array.

5.5 Experimental results

All the executions have been performed in a PC with an Intel(R) Core(TM) i7-3930K CPU running at 3.20GHz speed, 64GB of DDR3 1066 MHz RAM and a Raid 0 of two OCZ-VERTEX4 SSD drives. The operative system is Ubuntu Linux 14.04 64 bit. Compiler is gcc 4.8.2. All the tests in the results section have been launched sequentially, using a single execution thread with no parallelism involved.

The same index has been generated for all tools: Ensembl 68 human genome built upon GRCh37. The program dwgsim 0.1.8 from SAMtools was used to simulate two datasets of 2 million high quality Illumina reads. One dataset contains 250 bps reads while the other contains 400bps reads. The datasets contain reads with a maximum of 2 N's and 0.1% of mutations with 10% indels.

5.5.1 Comparison with other FM-Index only algorithms

As we stated before, our algorithm is not intended as a full sequence mapper, only a preprocessing step for modern sequence mappers. The purpose of this study is to provide a fair comparison against similar algorithms based only on FM-Index backward search, performing the experiments under the same input, execution arguments and system environment.

We only found similar implementations to our algorithm in Bowtie 1, SOAP 2 and BWA-backtrack. Comparing with these tools gives an idea of the impact in the performance of the improvements described in this paper.

The tools have been run with `-a` option in order to find all the possible mappings of each read; except BWA-backtrack, which focuses on finding the best mapping for each read.

Our algorithm has been run with a stack size of 50.000 partial results, big enough to deal with all the partial results without discarding any read locations. We also conducted tests with an stack size of 500 partial results, which increases

performance without significant mapping location loss. The minimum segment size needed to deal with the human genome variability is 31 nucleotides, allowing up to 7 errors with the 250bps dataset.

Results in table 5.1 and figure 5.3 show that our algorithm with a stack size of 50000 achieves a 8x speed-up over Bowtie 1 when aligning with 3 errors and a 7x speed-up over SOAP2 when aligning with 2 errors. Our algorithm can map with 5 errors in less time than Bowtie 1 with 3 errors. Execution times for the exact mapping case with no errors were similar for all the algorithms studied except BWA-backtrack. In general our algorithm is faster than the other approaches. This difference increases with the number of errors.

Table 5.1 also shows the percentage of reads found and the total mapping locations. The percentage represents if a read is found at least once in the reference, while in the mapping locations a read may appear several times. These values demonstrate that our algorithm performs an equivalent computation to Bowtie 1 and Soap 2, finding a similar amount of reads and mapping locations when allowing the same number of errors. Compared with BWA-backtrack we find the same percentage of reads.

Regarding the experiments with an stack size of 500 elements, our algorithm is 13x faster than BWA-backtrack when mapping with 6 errors. Results in table 5.1 and figure 5.4 show that limiting the stack size to 500 elements has little effect in the percentage of reads found (up to 2% less with 6 errors). However, the execution time is greatly decreased as the number of total mapping locations is reduced. As the mapping locations found with a small stack size are the ones with less errors, this approach is very useful for finding the best alignments.

During execution, our implementation has a memory footprint of 7GB, while Bowtie 1, SOAP 2 and BWA-backtrack consumed around 3 GB of RAM. This difference is because our algorithm is using indices for both forward and backward search. Although our algorithm requires more memory, it is still able to run in current desktop computers.

5.5.2 Preprocessing step for modern aligners

The purpose of the experiments in this section is to quantify how well our algorithm would perform as a preprocessing step for modern sequence mappers, concretely Bowtie 2 v2.2.3 and BWA-MEM v0.7.10. Such mappers combine backward search seeding with local alignment algorithms based on dynamic programming.

We compare the execution times of these modern sequence mappers alone against a pipeline that launches our algorithm, annotates the reads found when it has finished and then launches one of the mappers to find the remaining reads. Using this configuration the sensitivity is not modified, finding the same amount of reads.

Bowtie 2 and BWA-MEM have been run with its default execution parameters, finding in most cases the best occurrence of each read. Our algorithm has been run with a similar configuration by limiting the size of the partial result lists to 500 elements.

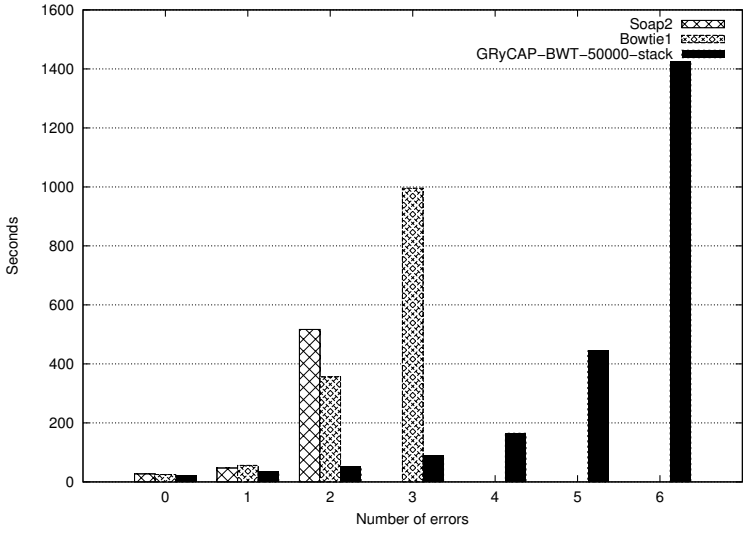


Figure 5.3: Backtracking tools all mapping locations. 2 Million 250bps reads. Execution times from 0 to 6 errors.

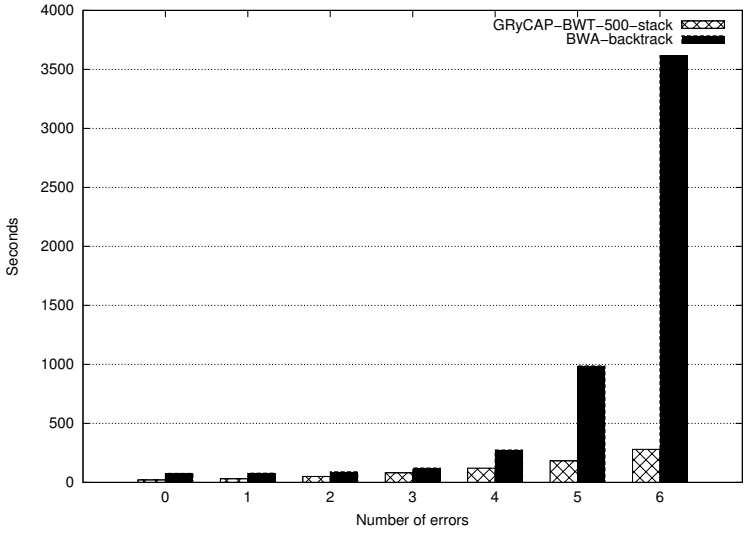


Figure 5.4: Backtracking tools best mapping locations. 2 Million 250bps reads. Execution times from 0 to 6 errors.

Figure 5.5 shows execution times for mapping the whole 250bps dataset. Bowtie 2 execution took 21m 47s and BWA-MEM execution took 19m 52s. For the combined alignment our algorithm was executed allowing up to 5 errors, finding 54.46% of the reads in 3m 2s. The remaining reads where feed to Bowtie 2 and BWA-MEM resulting in a total execution time of 12m 37s and 15m 1s respectively. This combined approach improves total alignment time by 42% for Bowtie 2 and 25% for BWA-MEM. Bowtie 2 found 94.26% of the reads and BWA-MEM found 94.48%, the same amount of reads where found when using our algorithm as a preprocessing step.

Figure 5.6 shows execution times for the 400bps dataset, Bowtie 2 took 40m 35s and BWA-MEM took 33m 09s. Our algorithm was executed allowing up to 9 errors, finding 62.21% of the reads in 9m 24s. The combined approach with Bowtie 2 took 24m 33s (40% faster) and with BWA-MEM took 25m 57s (21% faster). Bowtie 2 found 94.46% of the reads and BWA-MEM found 94.48%, the same amount of reads where found when using our algorithm as a preprocessing step.

Interestingly, these results reveal a greater performance improvement when combining our algorithm with Bowtie 2. In these experiments, BWA-MEM is faster than Bowtie 2 for aligning all reads. However, when using the preprocessing proposed in this paper Bowtie 2 becomes faster.

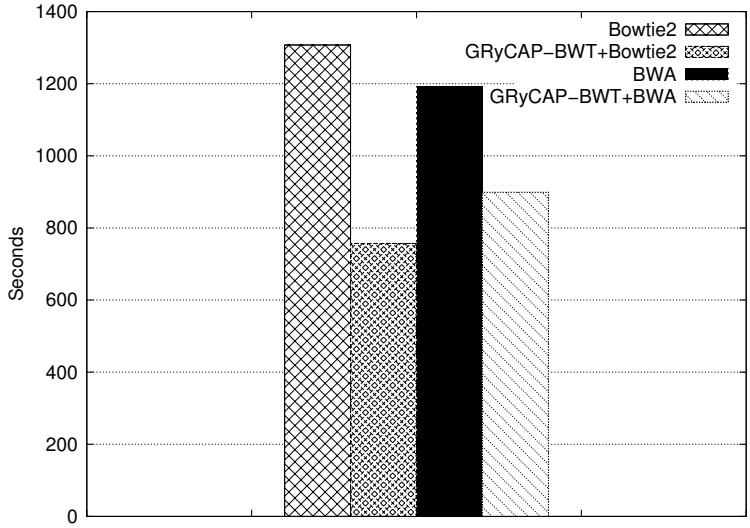


Figure 5.5: BWT and SW tools. 2 Million 250bps reads. Execution times comparing the new algorithm, the modern mappers and the combination of both.

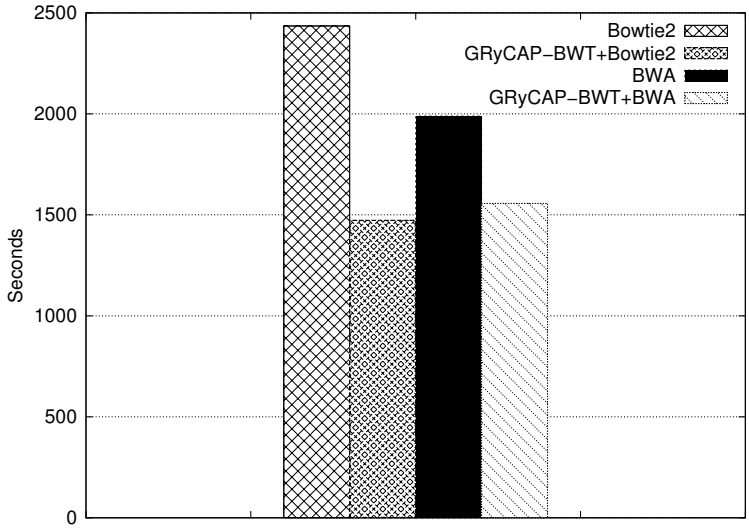


Figure 5.6: BWT and SW tools. 2 Million 400bps reads. Execution times comparing the new algorithm, the modern mappers and the combination of both.

5.5.3 Comparison between BWT and *csalib* runtimes

When dealing with big genomes, the size of the SA may be greater than the memory capacity of the machine. In this section we compare the speed of our algorithm using the BWT and the *csalib* out-of-core runtimes. This library is developed at the National Institute of Informatics in Tokyo (Japan) [Sadakane, 2010].

The BWT runtime uses about 7GB of RAM for the human genome index, while the *csalib* runtime does not load the index into memory. With *csalib* the index is directly read from disk using *mmap*, needing only a few hundreds of Megabytes of RAM to map reads.

Figure 5.7 shows a 100% increase in execution time when using the *csalib* runtime, across all error configurations. When using csalib the asymptotic cost of the algorithm is not modified, demonstrating the viability of this approach with currently affordable SSD disk configurations.

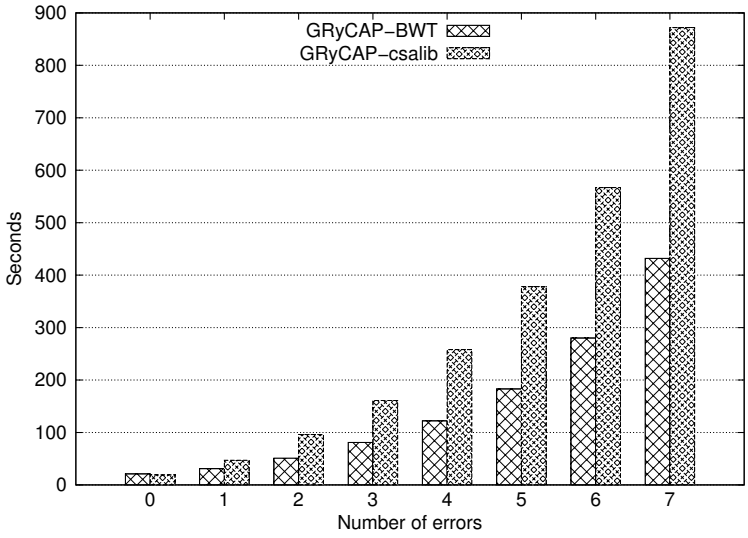


Figure 5.7: BWT and csalib runtimes. 2 Million 250bps reads. Execution times from 0 to 7 errors.

5.5.4 Asymptotic analysis

In this section we analyse the asymptotic cost of the algorithm. We also compare it with the other approaches studied.

The function that defines the growth of a search tree is $\mathcal{O}(k^e)$, where k is the branching factor and e is the depth of the tree. The branching factor of a search tree is obtained by dividing the total number of branches by the number of nodes with descendants.

In a trivial algorithm a full search tree is spawn at every position of the search string. The branching factor of the tree depends on the alternatives available: match, 4 mismatches and 4 insertions (one for each symbol). So, the asymptotic cost for an algorithm without optimisations is $\mathcal{O}(9^e)$, where the depth of the tree e is the number of errors allowed during the search.

Bowtie 1 and SOAP2 do not support indels, reducing the cost to $\mathcal{O}(5^e)$. This cost reduction comes at the expense of exploring less options and finding less mapping locations.

We employ the pruning techniques described previously to reduce the tree growth. The effectiveness of these techniques depends on the read analysed and the variability of the genome. Also, the worst case happens when few symbols of W are analysed, because all the branches exist in the reference. In practice, we perform exact search on the first symbols allowing errors later (figure 5.2). This way the number of branches is reduced.

Due to these variant factors, we studied the average branching factor of the search tree experimentally. We have randomly chosen 1000 reads from the 2 Million 250bps dataset. We aligned these reads with our algorithm allowing different number of errors in order to obtain the average branching factor. This parameter is an estimation of the growth of the tree, obtaining an asymptotic cost of $\mathcal{O}(2.53^e)$.

This is a great improvement compared with an algorithm without optimisations ($\mathcal{O}(9^e)$), while still allowing errors in any position of the read (including indels).

5.6 Conclusions

Improving previous research [Li and Durbin, 2009; Lam et al., 2009], we have developed a fast backward search algorithm for inexact sequence mapping (including mismatches, insertions and deletions). This algorithm is up to 13x faster than similar algorithms implemented in Bowtie, SOAP2 and BWA-backtrack. This impressive speed-up allows to handle more errors than before within a reasonable amount of time.

The proposed algorithm has been validated as a mapping preprocessing step, reducing the number of reads to align by 55%. This improves execution time of Bowtie 2 and BWA-MEM by about 40% and 20% respectively, mapping the same amount of reads in the same positions. The practical limit of errors allowed in this preprocessing appears when the seeding and local alignment becomes faster. We obtain good results with both 250bps and 400bps datasets, allowing up to 5 and 9 errors respectively.

Our implementation is built upon a modular architecture, being compatible with different backward search techniques. We tested an out-of-core implementation of the FM-Index provided by *csalib* library, obtaining reasonable execution times, showing the viability of cost-effective secondary memory configurations.

As future work, the computation of the exact segments done by the algorithm could be reused in the seeding phase, improving even further the speed of the overall process.

Furthermore, the mapping locations obtained by the algorithm must processed taking into account quality scores and gap penalties to match the criteria of the mapping tool on which is integrated. This post-process will not affect the logic nor the performance of the proposed algorithm.

Table 5.1: Results for Soap 2, Bowtie 1 and the new algorithm. The dataset contains 2 million 250bps reads.

	Time	% Found	Locations
Soap 2			
0 errors	25s	0.51%	11025
1 errors	41s	3.22%	71365
2 errors	6m 34s	10.25%	243599
Bowtie 1			
0 errors	24s	0.51%	11025
1 errors	51s	3.22%	71365
2 errors	4m 58s	10.25%	243599
3 errors	12m 13s	22.53%	594626
BWA-backtrack			
0 errors	1m 17s	0.51%	
1 errors	1m 19s	3.22%	
2 errors	1m 30s	10.29%	
3 errors	2m 2s	22.65%	
4 errors	4m 35s	38.73%	
5 errors	16m 28s	55.44%	
6 errors	60m 17s	69.78%	
GRyCAP-BWT	*Stack size 50000*		
0 errors	21s	0.51%	11025
1 errors	35s	3.22%	72546
2 errors	52s	10.29%	253479
3 errors	1m 28s	22.66%	644415
4 errors	2m 45s	38.74%	1188595
5 errors	7m 24s	55.46%	1820725
6 errors	23m 45s	69.78%	2830556
GRyCAP-BWT	*Stack size 500*		
0 errors	21s	0.51%	11025
1 errors	31s	3.22%	72546
2 errors	51s	10.29%	246841
3 errors	1m 21s	22.60%	515399
4 errors	2m 1s	38.44%	881503
5 errors	3m 3s	54.46%	1296682
6 errors	4m 40s	67.46%	1716369
7 errors	7m 12s	76.19%	2135443

Chapter 6

General conclusions

In this section we overview the achievements of the developments presented in this thesis, along with their practical applications in several bioinformatics pipelines. Section 6.1 covers the contributions of the exact mapping algorithm on GPU. Section 6.2 covers the contributions of the hybrid CPU-GPU pair-end algorithm allowing one error. Finally, section 6.3 describes the contributions related to the advanced search tree exploration algorithm for inexact mapping using backward search methods.

6.1 Exact mapping on GPU algorithm

The first contribution of this thesis is the parallelisation on GPU of a backward search algorithm using the FM-Index. We developed an exact GPU sequence mapper that outperforms the state-of-the-art FM-Index GPU implementation of SOAP3 [Salavert et al., 2012].

Our algorithm takes advantage of GPU parallel granularity by performing multiple searches at the same time, one for each GPU execution thread. This maximises memory locality and symmetric access to the reads. All the searches are performed with respect to the same reference search tree.

The thesis analyses the behaviour of the algorithm in GPU, showing a good scalability in the performance, only limited by the size of the GPU shared memory. The FM-index is too large to fit into the GPU SM shared memory, being stored in global memory, which is slower.

Another contribution of this development is the description of useful theory regarding the FM-index compression and how to access the shrank information efficiently during the search process on the GPU (see section 4.1).

Results in our hardware configuration indicate that, excluding disk transfers, the exact search on GPU works 12 times faster than the same algorithm on CPU. Moreover, the GPU computations and the IO operations have been effectively parallelised in a concurrent pipeline.

We measured the performance of our algorithm against existing CPU mapping tools based in similar approaches, namely SOAP2 and Bowtie, obtaining a 3-4x speed-up. When compared with SOAP3, which is also based on GPU, we also obtained a 3x speed-up. The bottleneck of this problem are the input and output operations, which consume 60% of the execution time. The tests where done in a regular hard drive, showing that our algorithm manages better the input and output operations. This is the reason of the better execution times.

This algorithm can be employed to quickly identify matchings without errors and to accelerate the seeding step of combined aligners, highlighting the regions to be studied with a local alignment algorithm.

The algorithm has been used in third party developments. Concretely, it has been included into the *Open source for Computational Biology* (OpenCB) toolkit (`https://github.com/opencb`). OpenCB is a complete set of open source bioinformatics tools available for the research community. In this pipeline, the mapping locations highlighted by the GPU exact search routine are analysed with a parallelised version of the Smith-Waterman algorithm using the Intel AVX instruction set.

Backward search methods may also have applications in the alignment of proteins. However, in this case the alphabet needed to represent all the aminoacids is much bigger and the spatial cost of the FM-Index depends on the alphabet size (matrix O has a row for each alphabet symbol). For this problem a backward search technique relying only on the SA [Manber and Myers, 1990] must be employed.

Additionally, there are prefix search techniques based on *Suffix array* (SA) [Manber and Myers, 1990] and enhanced SA [Abouelhoda et al., 2002] theory with applications to bioinformatics.

6.2 Hybrid CPU-GPU pair-end algorithm

As an extension of the exact mapping on GPU study case, we developed an algorithm to find one error mappings that also returns the pair-ends of a read [Salavert et al., 2014].

This algorithm takes advantage of concurrent CPU and GPU execution, becoming a real hybrid CPU-GPU implementation. Some parts of the algorithm are executed on the GPU, while other parts are executed on the CPU with a very litlle overhead. The computation done in the GPU is used to obtain the pair-ends of a read and, with very little CPU overhead, the mappings with one error.

The logic of this algorithm allows to build a pipeline to execute concurrently both computations. The speed-up of this hybrid CPU-GPU implementation is 10x when compared with the CPU only implementation, including the data transfers between CPU and GPU memory.

In this case we compared our algorithm with the latest version of SOAP3, called SOAP3-dp. Our implementation outperforms SOAP3-dp in terms of paral-

lelism and sensitivity. For these experiments we employed a raid of SSD drives, accelerating the speed of the write operations.

Analysis applications requiring the pair-ends of a read can take advantage of this algorithm. Moreover, alignment tools based on backward search seeding can employ seeds with one error using this development.

The hybrid pair-end algorithm was also included in the OpenCB pipeline, concretely in the RNA-seq analysis tool [Martínez et al., 2013b] [Martínez et al., 2013a]. During RNA-seq mapping the pair-ends highlight the exon-exon junctions of the mRNA.

GPGPU provide computer power at a reasonable price. Currently, a great amount of research effort is focused on finding problems suitable for this relatively new architecture. However, the SIMT parallelism model and the memory architecture impose many restrictions.

Sequence mapping is a concrete problem with a good speed-up when executed on GPU. The main drawback of this problem is the size of the search index, which forces to access global memory instead of using the shared memory.

An hybrid approach allows more complex problems to take advantage of the GPGPU. This is the case of the inexact sequence mapping algorithm allowing one error. This implementation was possible because memory transfers between the system main memory and the GPGPU memory take only a small fraction of the GPU execution time.

When implementing programs on the GPU we must take into account the architecture restrictions. Otherwise we may put our efforts in a program that executes properly on the GPU but does not take enough advantage of the parallelism.

Recently, Nvidia announced a change in the arquitecture of their new GPGPU. The new hardware will have less SM processors, but each one of the SM will have more SP. This may affect algorithms that can not take full advantage of the SP and depend on the amount of SM to achieve an acceptable speed-up.

6.3 Advanced search tree exploration inexact mapping algorithm

Existing sequence mapping approaches based only on backward search techniques combine the exact mapping procedure with a search tree exploration of all the possible solutions. Due to this, the complexity of the search tree grows exponentially when the number of errors allowed is increased. Available backward search algorithms based on this approach only allow a maximum of 4 errors during the alignment and even forbid insertions and deletions. Moreover, in some implementations errors in the first nucleotides of the reads are not considered in order to decrease the complexity of the search tree exploration.

Another significant contribution of this thesis is a backward search algorithm for inexact mapping supporting any number of errors. This algorithm is intended as an extra preprocessing step before the seed location phase of current sequence mapping tools. Our main goal is to reduce the number of read locations to be

analysed by more expensive combined strategies. This goal is achieved with a more efficient algorithm capable of dealing with longer reads and able to reuse part of the backward search preprocessing to accelerate the seeding phase. Similar existing algorithms only support efficiently about 2 errors, our objective is to demonstrate that this support may be increased by improving previous research [Li and Durbin, 2009][Lam et al., 2009].

This new algorithm is based on advanced search tree exploration techniques. We support all type of errors (insertions, deletions and mismatches) in all the positions of the read. We employ bounding strategies that directly deal with the genomic variability and considerably reduce the complexity of the search tree. The bounding strategies focus on full sensitivity, so the algorithm can find all the existing matches within the number of errors allowed. Nevertheless, the user can limit the size of the search tree and perform faster searches that discard mapping locations, obtaining a similar behaviour to the mappers that return only the best mapping of each read.

The proposed algorithm achieves higher sensitivity and a 13x speed-up over similar algorithms, allowing 6 errors. These results have been obtained under the same circumstances and without employing parallelisation techniques. Our algorithm supports insertions, deletions and mismatches in all the positions of the read by using a simple bidirectional BWT implementation.

The tests have been performed with the human genome. Smaller gnomes like *Drosophila melanogaster* have less variability. In such genomes the segment size parameter of the algorithm will be considerable smaller (concretely 35 for the human genome and 15 for the *Drosophila melanogaster* genome). So, given two datasets with reads of the same size the algortihm will allow more errors when studying the *Drosophila melanogaster* genome.

With a dataset simulating high quality Illumina reads and searching the human genome our algorithm can deal with 400bps reads with an alignment distance of up to 10 errors. Longer reads need more errors to be mapped. Increasing the number of errors the amount of reads aligned grows, but the maximum number of errors allowed by the algorithm depends on the length of the read and the variability of the reference genome. Moreover, the execution time grows exponentially with the number of errors, so there is a limit in the length of the reads determined by the maximum number of errors and the resulting execution time. Concretely, we were able to deal with reads of a maximum length of 500 nucleotides.

Modern aligners combine backward search with local alignment algorithms. Although our algorithm is only based on backward search and it is not intended as a complete sequence mapping tool, it could be used as a preprocessing step to modern sequence mappers. In our experiments the algorithm reduces by 55% the number of reads to be aligned, reducing the overall execution time of BWA-MEM and Bowtie2 about a 20–40%. Furthermore, the algorithm has being designed to be easily ported to high performance computing platforms and part of its computation could be reused in the seeding location phase.

This algorithm is based on replaceable components, providing the necessary interfaces to be compatible with any tool using a backward search method. We performed an study using both our own backward search implementation of the FM-Index and the implementation for DNA in *csalib* [Sadakane, 2010], this library implements several backward search methods based on either the FM-Index or SA [Sadakane, 2003].

In *csalib*, the data structures are not loaded into main memory [Mäkinen et al., 2004], but accessed from disk by demand using *mmap*. When using currently affordable SSD disks, these techniques allow to perform inexact mapping against long reference indexes in systems with low memory configurations. The overhead of the out-of-core index only increases the execution time by a 60%.

We described the compatibility interfaces that allow to use our algorithm with different index implementations easily. In the future, the algorithm will be modified to support faster indexes based on SA supporting backward search.

Chapter 7

Relevant publications and source code

Regarding the algorithm for exact search on GPU, we published a paper entitled *"Using GPUS for the exact alignment of short-read genetic sequences by means of the Burrows-Wheeler transform"* in the magazine *"IEEE/ACM Transactions on Computational Biology and Bioinformatics"* [Salavert et al., 2012].

The colaboration with *Centro de Investigación Príncipe Felipe* in the development of the OpenCB pipeline resulted in a paper entitled *"Acceleration of short and long DNA read mapping without loss of accuracy using suffix array"* in the magazine *"Bioinformatics"* [Tárraga et al., 2014].

Regarding the hybrid CPU-GPU algorithm that finds the pair-ends of the reads and allows one error sequence mapping, we published a paper entitled *"Inexact sequence mapping study cases: Hybrid GPU computing and memory demanding indexes"* in the proceedings of the *"International Work-Conference on Bioinformatics and Biomedical Engineering"* [Salavert et al., 2014] this paper was also selected for magazine publication. An extended version is under revision.

Regarding the inexact mapping algorithm a paper entitled *"Faster and more accurate inexact mapping using advanced tree exploration on backward search methods"* has been submitted to the *"BMC Bioinformatics"* magazine and is still under revision.

The colaboration with *Universitat Jaume I de Castelló* has resulted in a paper entitled *"Highly sensitive and ultrafast read mapping for RNA-seq analysis"*, submitted to *"Bioinformatics"* and still under revision.

The developments described in this thesis are part of a collaboration between *Universitat Politècnica de València*, *Centro de Investigación Príncipe Felipe* of València and *Universitat Jaume I de Castelló*.

The source code of the implementations described in this thesis is available at:

`http://josator.github.io/gnu-bwt-aligner/`

The code integration with the OpenCB pipeline is available at:

`https://github.com/josator/bioinfo-libs`

The OpenCB code is available at:

`https://github.com/opencb`

Bibliography

M. I. Abouelhoda, S. Kurtz, and E. Ohlebusch. The enhanced suffix array and its applications to genome analysis. In *In Proc. Workshop on Algorithms in Bioinformatics, in Lecture Notes in Computer Science*, pages 449–463. Springer-Verlag, Heidelberger, Berlin, 2002.

Altschul et al. Basic local alignment search tool. *J. Mol. Biol.*, pages 403–410, 1990.

S. F. Altschul, T. L. Madden, A. A. Schaffer, J. Zhang, Z. Zhang, W. Miller, and D. J. Lipman. Gapped Blast and Psi-Blast: a new generation of protein database search programs. *Nucleic Acids Res.*, 25:3389–3402, 1997.

J. L. Bentley and R. Sedgewick. Fast algorithms for sorting and searching strings. In *Proceedings of the eighth annual ACM-SIAM symposium on Discrete algorithms*, SODA '97, pages 360–369. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1997.

O. A. R. Board. Specification. `http://openmp.org/wp/about-openmp/`, 2013.

M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical Report 124, SRC (Digital, DEC, Palo Alto), 1994.

G. Church. Genomes for all. *Scientific American*, 294:47–54, 2006.

R. Durbin, S. Eddy, A. Krogh, and G. Mitchison. *Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids*. Cambridge University Press, Cambridge, 1998.

S. R. Eddy. Profile hidden markov models. *Bioinformatics*, 14(9):755–763, 1998.

Editorial. Metagenomics versus moore's law. *Nature Methods*, 6(9):623, 2009.

P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *FOCS*, pages 390–398. 2000.

P. Ferragina and G. Manzini. Indexing compressed text. *J. ACM*, 52(4):552–581, 2005.

O. Gotoh. An improved algorithm for matching biological sequences. *J. Mol. Biol.*, 162:705–708, 1982.

Grossi and Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SICOMP: SIAM Journal on Computing*, 35, 2005.

N. Hall. Advanced sequencing technologies and their wider impact in microbiology. *J Exp Biol*, 210(9):1518–1525, 2007.

A. Hatem, D. Bozdag, A. Toland, and U. Catalyurek. Benchmarking short sequence mapping tools. *BMC Bioinformatics*, 14(1):184, 2013.

R. Hughey and A. Krogh. Sam: Sequence alignment and modeling software system. Technical report, University of California at Santa Cruz, Santa Cruz, CA, USA, 1995.

IEEE, editor. *23rd IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2009, Rome, Italy, May 23-29, 2009*. IEEE, 2009.

J. Kärkkäinen, P. Sanders, and S. Burkhardt. Linear work suffix array construction. *J. ACM*, 53(6):918–936, 2006.

W. J. Kent. BLAT - the BLAST-like alignment tool. *Genome Res.*, 12(4):656–664, 2002.

Khronos. The opencl specification. `http://www.khronos.org/registry/cl/specs/opencl-1.0.29.pdf`, 2011. Version 1.1 Revision 44.

P. Klus, S. Lam, D. Lyberg, M. Cheung, G. Pullan, I. McFarlane, G. Yeo, and B. Lam. Barracuda - a fast short read sequence aligner using graphics processing units. *BMC Research Notes*, 5(1):27, 2012.

F. Kulla and P. Sanders. Scalable parallel suffix array construction. In *Proceedings of the 13th European PVM/MPI User's Group Conference on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, EuroPVM/MPI'06, pages 22–29. Springer-Verlag, Berlin, Heidelberg, 2006.

T. Lam, R. Li, A. Tam, S. Wong, E. Wu, and S. Yiu. High throughput short read alignment via bi-directional bwt. In *Bioinformatics and Biomedicine, 2009. BIBM '09. IEEE International Conference on*, pages 31 –36. 2009.

B. Langmead and S. L. Salzberg. Fast gapped-read alignment with Bowtie 2. *Nat Meth*, 9(4):357–359, 2012.

B. Langmead, C. Trapnell, M. Pop, and S. L. Salzberg. Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. *Genome Biology*, 10(R25), 2009.

N. J. Larsson and K. Sadakane. Faster suffix sorting. *Theoretical Computer Science*, 387(3):258 – 272, 2007. The Burrows-Wheeler Transform.

H. Li and R. Durbin. Fast and accurate short read alignment with burrows-wheeler transform. *Bioinformatics*, 25(14):1754–1760, 2009.

H. Li and R. Durbin. Fast and accurate long-read alignment with Burrows–Wheeler transform. *Bioinformatics*, 26(5):589–595, 2010.

H. Li and N. Homer. A survey of sequence alignment algorithms for next-generation sequencing. *Briefings in Bioinformatics*, 11(5):473–483, 2010.

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, 2009.

L. Ligowski and W. Rudnicki. An efficient implementation of smith waterman algorithm on GPU using CUDA, for massively parallel scanning of sequence databases. In [IEEE, 2009], pages 1–8.

C. Ling and K. Benkrid. Design and implementation of a CUDA-compatible GPU-based core for gapped BLAST algorithm. *Procedia CS*, 1(1):495–504, 2010.

D. J. Lipman and W. R. Pearson. Rapid and sensitive protein similarity searches. *Sci.*, 227:1435–1441, 1985.

C. Liu et al. Soap3: Gpu-based compressed indexing and ultra-fast parallel alignment of short reads. In *Third Workshop on Massive Data Algorithmics, Paris, France*. 2011.

W. Liu, B. Schmidt, G. Voss, A. Schroder, and W. Muller-Wittig. Bio-sequence database scanning on a GPU. In *Proceedings of the fifth IEEE International Workshop on High Performance Computational Biology*. Rhodes Island, Greece, 2006.

Y. Liu, B. Popp, and B. Schmidt. Cushaw3: Sensitive and accurate base-space and color-space short-read alignment with hybrid seeding. *PLoS ONE*, 9(1):e86869, 2014.

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.

V. Mäkinen, G. Navarro, and K. Sadakane. Advantages of backward searching; efficient secondary memory and distributed implementation of compressed suffix arrays. In *Proceedings of the 15th International Conference on Algorithms and Computation*, ISAAC'04, pages 681–692. Springer-Verlag, Berlin, Heidelberg, 2004.

S. Manavski and G. Valle. CUDA compatible GPU cards as efficient hardware accelerators for smith-waterman sequence alignment. *BMC Bioinformatics*, 9(S-2), 2008.

U. Manber and G. 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. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1990.

G. Manzini. An analysis of the burrows-wheeler transform. In *Proceedings of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 669–677. ACM-SIAM, N.Y., 1999.

S. Marco-Sola, M. Sammeth, R. Guigo, and P. Ribeca. The GEM mapper: fast, accurate and versatile alignment by filtration. *Nat Meth*, 9(12):1185–1188, 2012.

H. Martínez, J. Tárraga, I. Medina, S. Barrachina, M. Castillo, J. Dopazo, and E. S. Quintana-Ortí. Concurrent and accurate rna sequencing on multicore platforms. *CoRR*, abs/1304.0681, 2013a.

H. Martínez, J. Tárraga, I. Medina, S. Barrachina, M. Castillo, J. Dopazo, and E. S. Quintana-Ortí. A dynamic pipeline for rna sequencing on multicore processors. In *EuroMPI*, pages 235–240. 2013b.

G. Moore. Progress in digital integrated electronics. In *Electron Devices Meeting, 1975 International*, volume 21, pages 11–13. 1975.

G. Moore. Cramming more components onto integrated circuits. *Proceedings of the IEEE*, 86(1):82–85, 1998.

J. C. Mu, H. Jiang, A. Kiani, M. Mohiyuddin, N. B. Asadi, and W. H. Wong. Fast and accurate read alignment for resequencing. *Bioinformatics*, 2012.

A. Munshi, B. Gaster, T. G. Mattson, J. Fung, and D. Ginsburg. *OpenCL Programming Guide*. Addison Wesley, Upper Saddle River, NJ, 2011.

S. B. Needleman and C. D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48(3):443 – 453, 1970.

Z. Ning, A. J. Cox, and J. C. Mullikin. SSAHA: A fast search method for large DNA databases. *Genome Res.*, 11(10):1725–1729, 2001a.

Z. Ning, A. J. Cox, and J. C. Mullikin. Ssaha: A fast search method for large dna databases. *Genome Research*, 11(10):1725–1729, 2001b.

G. Nong, S. Zhang, and W. H. Chan. Linear suffix array construction by almost pure induced-sorting. In *Data Compression Conference, 2009. DCC '09.*, pages 193–202. 2009.

C. Notredame, D. G. Higgins, and J. Heringa. T-Coffee: A novel method for fast and accurate multiple sequence alignment. *Journal of molecular biology*, 302(1):205–217, 2000.

NVIDIA. Nvidia cuda c programming guide. `http://docs.nvidia.com/cuda/cuda-c-programming-guide/`, 2014.

M. Oguzhan Kulekci, W.-K. Hon, R. Shah, J. Scott Vitter, and B. Xu. Psi-ra: a parallel sparse index for genomic read alignment. *BMC Genomics*, 12(Suppl 2):S7, 2011.

D. Okanohara and K. Sadakane. A linear-time burrows-wheeler transform using induced sorting. In J. Karlgren, J. Tarhio, and H. Hyyrö, editors, *String Processing and Information Retrieval*, volume 5721 of *Lecture Notes in Computer Science*, pages 90–101. Springer, Heidelberg, Berlin, 2009.

Pearson and Lipman. Improved tools for biological sequence comparison. In *Proc. Natl. Acad. Sci.*, volume 85, pages 24444–24448. 1988.

W. R. Pearson. Searching protein sequence libraries: Comparison of the sensitivity and selectivity of the smith-waterman and {FASTA} algorithms. *Genomics*, 11(3):635 – 650, 1991.

S. J. Puglisi, W. F. Smyth, and A. H. Turpin. A taxonomy of suffix array construction algorithms. *ACM Comput. Surv.*, 39(2), 2007.

K. Sadakane. New text indexing functionalities of the compressed suffix arrays. *Journal of Algorithms*, 48(2):294 – 313, 2003.

K. Sadakane. A library for compressed full-text indexes. `https://code.google.com/p/csalib/`, 2010.

J. Salavert, I. Blanquer, and Others. Using gpus for the exact alignment of short-read genetic sequences by means of the burrows-wheeler transform. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 9(4):1245–1256, 2012.

J. Salavert, A. Tomas, I. Medina, and I. Blanquer. Inexact sequence mapping study cases: Hybrid gpu computing and memory demanding indexes. In *International Work-Conference on Bioinformatics and Biomedical Engineering*, volume 1, pages 362–373. 2014.

F. Sievers, A. Wilm, D. Dineen, T. J. Gibson, K. Karplus, W. Li, R. Lopez, H. McWilliam, M. Remmert, J. Soding, J. D. Thompson, and D. G. Higgins. Fast, scalable generation of high-quality protein multiple sequence alignments using Clustal Omega. *Molecular Systems Biology*, 7(1), 2011.

T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, Vol. 147:195–197, 1981.

G. M. Striemer and A. Akoglu. Sequence alignment with GPU: Performance and design challenges. In [IEEE, 2009], pages 1–10.

J. Tárraga, V. Arnau, H. Martínez, R. Moreno, D. Cazorla, J. Salavert-Torres, I. Blanquer-Espert, J. Dopazo, and I. Medina. Acceleration of short and long dna read mapping without loss of accuracy using suffix array. *Bioinformatics*, 2014.

J.-S. Varré, B. Schmidt, S. Janot, and M. Giraud. *Manycore high-performance computing in bioinformatics*. World Scientific, 2011.

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

M. Vyverman, B. De Baets, V. Fack, and P. Dawyndt. essamem: finding maximal exact matches using enhanced sparse suffix arrays. *Bioinformatics*, 29(6):802–804, 2013.

Y. Xin, B. Liu, B. Min, W. X. Li, R. C. Cheung, A. S. Fong, and T. F. Chan. Parallel architecture for {DNA} sequence inexact matching with burrows-wheeler transform. *Microelectronics Journal*, 44(8):670 – 682, 2013.