POLITECNICO DI MILANO
Master of Science in Biomedical Engineering
Department of Electronics, Informatics and Bioengineering

POLITECNICO
MILANO 1863

BWaveR: an FPGA-accelerated Genomic Sequence Mapper Leveraging Succinct Data Structures

NECSTLab
Novel, Emerging Computing System Technologies Laboratory
Politecnico di Milano

Advisor: **Prof. Marco D. Santambrogio**
Politecnico di Milano

Co-advisor: **Dott. Ing. Lorenzo Di Tucci**
Politecnico di Milano

Master Thesis of:
Guido Walter Di Donato - 897749

Academic Year 2018-2019

Do I contradict myself ?

Very well, then, I contradict myself ;

I am large - I contain multitudes .

Walt Whitman

Acknowledgments

First, I would like to say thanks to my advisor, Marco Domenico Santambrogio, for being a role model and a friend, and for giving me the opportunity to work on this amazing project. Without his guidance, support, and critical thinking towards my work, this thesis would never have been possible.

Also, I would like to thank all the people working in the NECSTLab for creating a friendly, stimulating, and challenging environment sharing expertise and suggestions. I am very thankful to Lorenzo, who introduced me to the world of genomics and bioinformatics, and followed with interest the development of this work. Huge thanks also to Alberto, not only for giving me lots of useful insights about FPGAs and the tools for programming them, but also, and foremost, for being a great friend.

I am grateful to all the friends I met in these years, who made my journey at Politecnico a wonderful experience. Special thanks to Simone, Alessia, Marco and Valentina: I will remember forever the happy time we spent together.

I also want to say thanks to my beautiful and extraordinary family. Despite being far away, all of you always found a way to be by my side and to make me feel at home, wherever I was.

Last but foremost, my biggest *thank you* goes to my lovely parents, Salvatore and Giovanna, and to my "little" brother, Andrea: you are my inspiration and my motivation. Without the trust, the support, and the encouragement you have always given to me, I might not be the person I am today. I could not have achieved any of this without you.

GWDD

Abstract

The advent of Next Generation Sequencing (NGS) produced an explosion in the amount of genomic data generated, which resulted in the birth and early development of personalized medicine. However, the tools currently employed for the analysis of these data still require too much time and power. Thus, to boost the research in this field, new bioinformatic tools are needed, which can efficiently handle the vast amount of genomic data, in order to keep up with the pace of NGS technologies.

In this scenario, the aim of this thesis is the design and the implementation of a memory-efficient, easy-to-use short sequence mapper, to be employed in various bioinformatic applications. At the core of the proposed tool there is an efficient implementation of a succinct data structure, allowing to compress the genomic data while still providing efficient queries on them. A comprehensive description of the data encoding scheme is presented in this work, together with the characterization of the proposed data structure in terms of memory utilization and execution time.

To improve the performances and the energy efficiency of the sequence mapping process, this thesis also proposes a custom hardware design, which leverages the compression capability of the proposed data structure to fully exploit the

highly parallel architecture of Field Programmable Gate Arrays (FPGAs). We employed such custom hardware architecture to develop BWaveR, a fast and power-efficient hybrid sequence mapper, which is made available through an intuitive web application that guarantees high usability and provides great user experience.

Finally, this work provides a validation of the developed tool, in order to prove the correctness and reliability of the results it produces. Moreover, it presents an extensive evaluation of the performances of the proposed hybrid system, through a comparison with state-of-the-art equivalent software tools. The experimental results show that the proposed hardware architecture is able to provide application speed-up while significantly reducing the energy consumption. Thus, BWaveR constitues a valid solution for accelerating bioinformatic applications involving genomic sequence mapping, allowing users to benefit from hardware acceleration without any development effort or any knowledge of the underlying hardware architecture.

Ampio Estratto

Lo sviluppo di tecnologie di sequenziamento sempre più efficienti, che prendono il nome di Next Generation Sequencing (NGS), ha portato ad un rapidissimo aumento della quantità di dati genomici disponibili, ponendo le basi per la nascita della medicina personalizzata. Purtroppo però, l'analisi di una tale quantità di dati richiede, ad oggi, ancora troppo tempo e troppa energia. Nuovi strumenti computazionali sono quindi necessari per accelerare la ricerca in questo campo, e per garantire lo sviluppo e la democratizzazione della medicina personalizzata.

In tale contesto, lo scopo di questa tesi è la progettazione e la realizzazione di un tool per l'allineamento di sequenze genomiche, che risulti efficiente e facile da utilizzare in diverse applicazioni bioinformatiche. Il funzionamento di tale tool è basato su una struttura dati succinta, che permette di comprimere efficacemente i dati genomici, riducendo l'utilizzo di memoria e permettendo, allo stesso tempo, un rapido accesso a tali dati. In questo elaborato verrà fornita un'ampia descrizione della struttura dati proposta, del suo utilizzo, e delle sue caratteristiche in termini di utilizzo di memoria e tempi d'esecuzione.

Questa tesi presenta anche la realizzazione di BWaveR, un sistema eterogeneo per l'allineamento di sequenze, che si avvale delle ridotte dimensioni della strut-

tura dati proposta per sfruttare al meglio l'architettura parallela dei Field Programmable Gate Arrays (FPGAs). Tali dispositivi offrono vantaggi significativi dal punto di vista dell'efficienza energetica, ma risultano particolarmente difficili da programmare. BWaveR, invece, consente di sfruttare i vantaggi dell'accelerazione hardware attraverso una semplice ed intuitiva applicazione web, che garantisce un facile utilizzo ed una buona user experience.

Il tool sviluppato è stato validato e valutato attreverso il confronto con applicativi software equivalenti. I test di validazione dimostrano l'affidabilità di BWaveR e provano la consistenza dei risultati prodotti. Inoltre, i risultati sperimentali mostrano che l'architettura hardware proposta è in grado di ridurre il tempo d'esecuzione ed il consumo energetico del processo di allineamento di sequenze genomiche. Pertanto, BWaveR si propone come una valida soluzione per accelerare una vasta gamma di applicazioni bioinformatiche, permettendo anche ad utenti senza competenze di programmazione di beneficiare dei vantaggi di un'architettura hardware specializzata.

Contents

List of Figures

List of Tables

List of Algorithms

List of Abbreviations

A Adenine. 7, 30, 64

API Application Programming Interface. 71, 91

ASIC Application Specific Integrated Circuit. 32, 33

BAM Binary Alignment Map. 57, 58

BLAST Basic Local Alignment Search Tool. 4, 50

bp base pairs. 10, 11, 27, 40, 44, 57, 73, 74, 84, 87, 89, 92, 96

BRAM Block Random Access Memory. 44, 73, 75, 92

BWA Burrows-Wheeler Aligner. 38–42, 45

BWT Burrows-Wheeler Transform. 3–5, 21, 24, 38, 40–43, 51–55, 58–64, 67, 68, 72, 73, 75, 79, 87, 91, 92

C Cytosine. 7, 64

CPU Central Processing Unit. 4, 32, 42–45, 54, 55, 71, 72, 91

DAWG Directed Acyclic Word Graph. 39

ddNs dideoxynucleotides. 7

DNA DeoxyriboNucleic Acid. 1, 2, 6–10, 12, 17, 19, 21, 22, 96

DP Dynamic Programming. 2, 37–40, 51, 96

DRAM Dynamic Random Access Memory. 73

DSP Digital Signal Processor. 35

FPGA Field Programmable Gate Array. 4, 5, 20, 21, 33, 35, 40, 41, 43–46, 51–55, 63, 70–74, 80, 82, 90–93, 95, 96

G Guanine. 7, 64

GPU Graphical Processing Unit. 33, 41–43, 45

GUI Graphic User Interface. 76

HGP Human Genome Project. 7

HLS High-Level Synthesis. 35, 71, 75

HTS High Throughput Sequencing. 8

IGSR International Genome Sample Resource. 84

LUT Look-Up Table. 33

MEM maximal exact matching. 38, 40

MUX multiplexer. 33

NCBI National Centre for Biotechnology Information. 22

NGS Next Generation Sequencing. 1, 2, 8, 9, 13, 14, 16, 41, 94

OLC Overlap-Layout-Consensus. 13–15

PC Personal Computer. 54

PE Processing Element. 46

RNA RiboNucleic Acid. 22, 64

RRR Raman, Raman and Rao. 4, 21, 29–31, 41, 51, 54, 63, 64, 66–68, 71, 72, 74, 91, 95

SA Suffix Array. 54, 58–61, 68, 79

SAM Sequence Alignment Map. 57, 58

SDSL Succinct Data Structures Library. 40, 63

SIMD Single Instruction Multiple Data. 39

SIMT Single Instruction Multiple Thread. 42

SLD System-Level Design. 35

SMEMs supermaximal exact matches. 40

SNP Single Nucleotide Polymorphism. 19

SRAM Static Random Access Memory. 33

T Thymine. 7, 64

U Uracile. 64

WGS Whole Genome Sequencing. 7, 84

CHAPTER 1

Introduction and motivation

Precision medicine holds promise for improving many aspects of health and healthcare. This emerging approach for disease treatment and prevention takes into account individual variability in genes, allowing doctors to predict, among different possible treatments, which is best suited to each individual or group of people. It is in contrast to the common *one-size-fits-all* approach, in which disease treatment and prevention strategies are developed for the average person, with less consideration for the differences between individuals [1].

One of the most promising fields of application of precision medicine is oncology. Clinicians and scientists are using Next Generation Sequencing (NGS) tools to gain insight into each patient's disease, and treating cancer at the genomic level is having a profound impact on cancer care [2]. By examining the DeoxyriboNucleic Acid (DNA) of a patient's tumor, it is possible to identify the cancer-causing genes that make it grow. This information allows researchers to find a drug that targets that gene, in order to deactivate it and attack the cancer cells [3].

Twenty years ago, gene sequencing cost millions and took months to receive results. With current technology instead, it is possible to do the sequencing and its interpretation at a reasonable cost, getting this information back to the patient and the patient's doctor within a few weeks [3]. Thanks to the development of NGS tools, the actual sequencing of the DNA now takes only minutes and costs less than 1000$ per whole human genome. The complexity comes from "reading" the produced data in order to identify the abnormal genes patterns, which requires two main steps: genome assembly and genomic analysis.

Genome assembly (or sequence assembly) refers to merging sequenced DNA fragments, called "reads", in order to reconstruct the original longer sequence they generated from [4]. Genomic analysis instead is the identification, measurement, and comparison of genomic features, followed by their annotation, that describes the function of the product of the identified genes [5]. Both these steps are computationally intensive and they share the same bottleneck: sequence alignment, defined as the arrangement of two or more nucleotide (or amino acid) sequences so that their most similar elements are juxtaposed [6]. Various approaches can be found in the literature to address different formulations of the sequence alignment problem, that can be a *global alignment* [7], a *local alignment* [8] or a hybrid version of the problem, often referred as *semi-global* or *glocal alignment* [9].

Many of the proposed approaches rely on Dynamic Programming (DP), an optimization method over plain recursion, which involves storing the results of sub-problems so that it is not necessary to re-compute them when needed later. Although this method allows reducing time complexities from exponential to polynomial, the amount of data to be processed is so vast that other optimizations

are needed to further reduce the execution time. For this reason, many state-of-the-art sequence aligners rely on the *seed and extend* paradigm, consisting in finding exact matches between seeds (very short genomic subsequences), then extending the alignment in the selected regions through a dynamic programming approach [10]. Burrows-Wheeler Mapping is one of the most employed techniques for the exact matching of sequences, thus is often employed for the seeding step [11, 12]. It relies on the Burrows-Wheeler Transform (BWT) of the genomic sequence and its FM-index, that allow to efficiently carry out a backward search of a pattern within a large sequence.

In the last years, hardware accelerators proved to be effective in reducing the computation time of sequence alignment algorithms [13], and in optimizing the performance over power consumption ratio [14]. This is fundamental for decreasing the cost of genome assembly and genomic analysis, and it is crucial for promoting the development and the diffusion of precision medicine. However, hardware accelerators are usually expensive and difficult to program. For this reason, a recent trend in bioinformatics is using hardware accelerators *as a service*, through cloud-based platforms which allow users to easily exploit the computational power of the accelerators, without the need of buying and programming them [15].

Moreover, because of the vast amount of data to be processed, a noticeable effort has been put by researchers in representing the data in the most compact way. Here is where Succinct Data Structures come into play: these structures provide the possibility to encode genomic data in a space close to theoretical lower bound, while still allowing for fast queries on the data [16].

Finally, it is worth to note that many bioinformatic tools are only accessible

through command line interfaces, which limit their usability. A user-friendly easy-to-use interface is a key factor in determining the potential diffusion of a tool and its acceptance level in the bioinformatic community [17]. A famous example is the Basic Local Alignment Search Tool (BLAST), that thanks to its intuitive web-based graphical user interface is the most employed bioinformatic tool ever, with more than 75000 citations in published research papers [18].

1.1 Thesis goal

Considering all the aforementioned reasons, the aim of this work is the implementation of a memory-efficient and easily accessible Burrows-Wheeler mapper, leveraging Field Programmable Gate Arrays (FPGAs) for improving the performance over power ratio of the mapping process. The proposed heterogeneous system exploits the general purpose Central Processing Unit (CPU) for the computation of the Burrows-Wheeler Transform of the reference genome, while the pattern matching (i.e. read mapping) is executed on FPGA. The CPU is also employed for encoding the BWT in a succinct data structure, and for the memory management. Query sequences and their reverse complement are efficiently mapped to the reference genome, leveraging the high parallelism and the efficient bit-level operations of the FPGA architecture, to reduce the computation time and the energy consumption.

The reference genomic sequence, of length N, is encoded in a succinct data structure based on Wavelet Trees and Raman, Raman and Rao (RRR) structures, allowing fast rank and select queries on the BWT. Such succinct representation allows to efficiently perform pattern matching on the reference sequence, without

the need of storing any additional data. Actually, the implemented data structure is even able to compress the size of the data w.r.t. the original representation of the sequence, reducing the required space to $\sim \frac{N}{4}$ bytes, while still permitting to search a pattern of length p in O(p) time, regardless of the reference's size.

The user-friendly web-based interface allows a user to upload the reference genome and the set of reads to be aligned, either in compressed or uncompressed FASTA/FASTQ format. A simple click triggers the computation of the BWT, its encoding in the succinct data structure, and finally the read alignment on FPGA. Thus, the user will get back his/her results without any effort, and without any knowledge of the underlying hardware architecture.

1.2 Outline

The remainder of the document is organized as follow. Chapter 2 will give some insights about genome sequencing, genome assembly, genomic analysis, and the importance of read alignment for the development of precision medicine. Chapter 3 will present the Burrows-Wheeler mapping problem, and the materials and methods employed in the development of the proposed tool. Continuing, Chapter 4 will discuss the state-of-the-art Burrows-Wheeler aligners, and it will present related works on hardware accelerators. Chapter 5 will give a detailed description of the proposed hybrid aligner and of its implementation, while Chapter 6 will present the experimental setup and the obtained results. Finally, Chapter 7 will draw the conclusions and possible future development of the proposed tool.

CHAPTER 2

Background

This Chapter introduces the reader to the genome assembly process and the genomic analysis, which are at the core of personalized medicine. Section 2.1 describes the techniques employed for digitalizing the information contained in DNA, and the fragmented data they produce. Then, Section 2.2 provides a description of the assembly approaches for creating a representation of the original chromosomes from which the DNA originated. Finally, Section 2.3 presents the applications of sequence alignment for genome analysis and their central role in the development of precision medicine.

2.1 Genome sequencing

DNA sequencing is the process of determining the sequence of the four chemical building blocks, called nucleotide bases, that make up a molecule of DNA. The double helix structure, characteristic of the DNA molecule, is due to the fact

that the four chemical bases bind in accordance with what is called "scheme of complementarity of the bases": Adenine (A) always binds to Thymine (T), while Cytosine (C) always binds to Guanine (G). The scheme of complementarity is the basis for the mechanism by which DNA molecules are copied during meiosis, and the pairing also underlies the methods by which most DNA sequencing experiments are done [19]. Today, with the right equipment and materials, sequencing a short piece of DNA is relatively straightforward. However, Whole Genome Sequencing (WGS), that is the process of digitalizing the complete DNA sequence of an organism's genome at a single time, still remains a complex task. It requires breaking the DNA of the organism into many smaller pieces, sequencing them all, and assembling the resulting short sequences into a single long "consensus sequence".

The Human Genome Project (HGP), an international collaborative research program, led to the completion of the first human genome in 2003, after about 15 years of effort [20]. In this project, DNA regions up to about 900 base pairs in length were sequenced using a method called *Sanger sequencing* or *chain termination method*, developed by the British biochemist Frederick Sanger in 1977 [21]. Sanger sequencing is based on the iterative selective incorporation of chain-terminating dideoxynucleotides (ddNs) by DNA polymerase during in vitro DNA replication; ddNs may be radioactively or fluorescently labeled for detection in automated sequencing machines. An overview of the chain termination method is given in Figure 2.1; a comprehensive explanation of the method can be found in [21].

Sanger chain termination method gives high-quality sequences and it is still widely used for the sequencing of individual pieces of DNA. Anyway, whole genomes

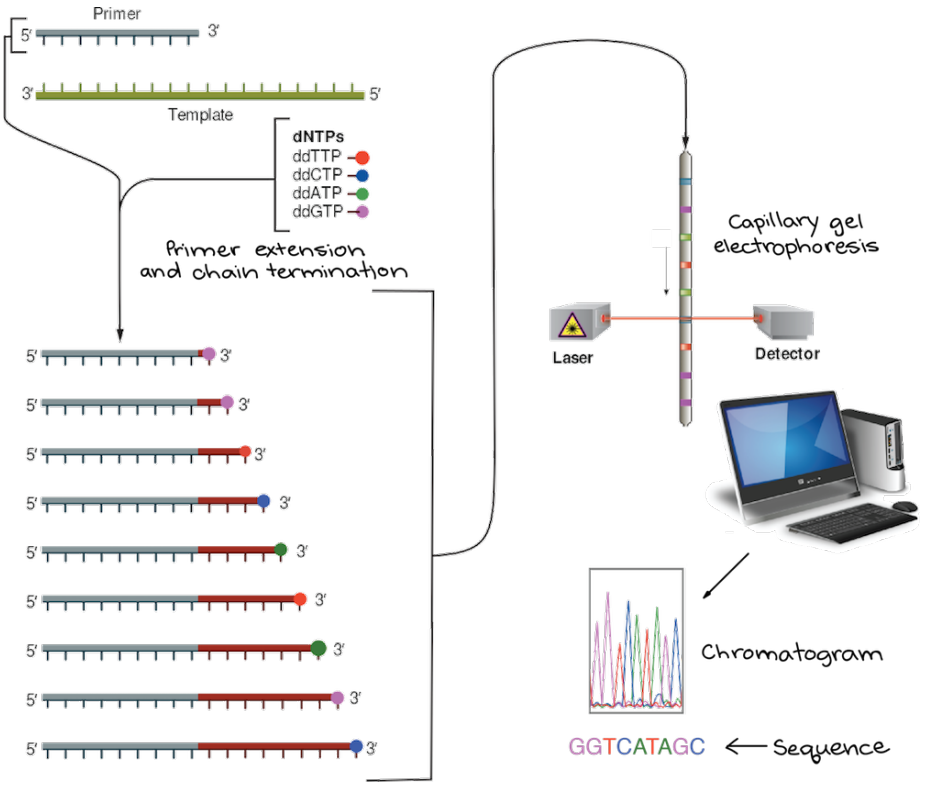

Figure 2.1: The Sanger chain termination method for DNA sequencing

are now typically sequenced using new methods that have been developed over the past two decades. These methods, called *Next Generation Sequencing (NGS)* or *High Throughput Sequencing (HTS)*, have significantly reduced the cost and accelerated the speed of large-scale sequencing. Figure 2.2 shows the Carlson curve, the biotechnological equivalent of Moore's law, describing the rate of DNA sequencing or cost per sequenced base as a function of time. Carlson predicted that the doubling time of DNA sequencing technologies (measured by cost and performance) would be at least as fast as Moore's law. Actually, Moore's Law started being profoundly out-paced in 2008, when sequencing centers transitioned to HTS technologies. This allowed researchers to achieve the "1000$ genome" goal

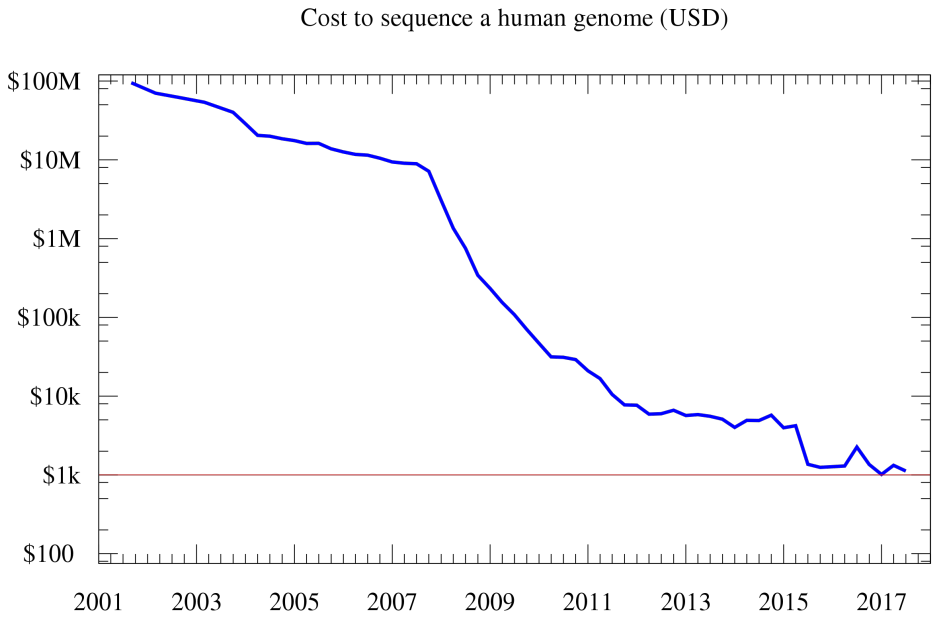

Figure 2.2: Carlson curve: total cost of sequencing a human genome over time

in 2017, and to set more ambitious objectives for the next years, like the "100$ genome" goal set by Illumina CEO [22].

NGS technologies are typically characterized by being highly scalable, allowing the entire genome to be sequenced at once. Usually, this is accomplished by a process called *shotgun sequencing*, which consists of fragmenting the genome into small pieces, randomly sampling for a fragment, and sequencing it using one of a variety of available technologies. Multiple fragments are sequenced at once in an automated process, concurrently producing thousands or millions of text sequences, called *reads*. Figure 2.3 gives an overview of the shotgun sequencing process.

As stated before, different new sequencing technologies have been developed in the last years, intended to lower the cost and the time of DNA sequencing [23].

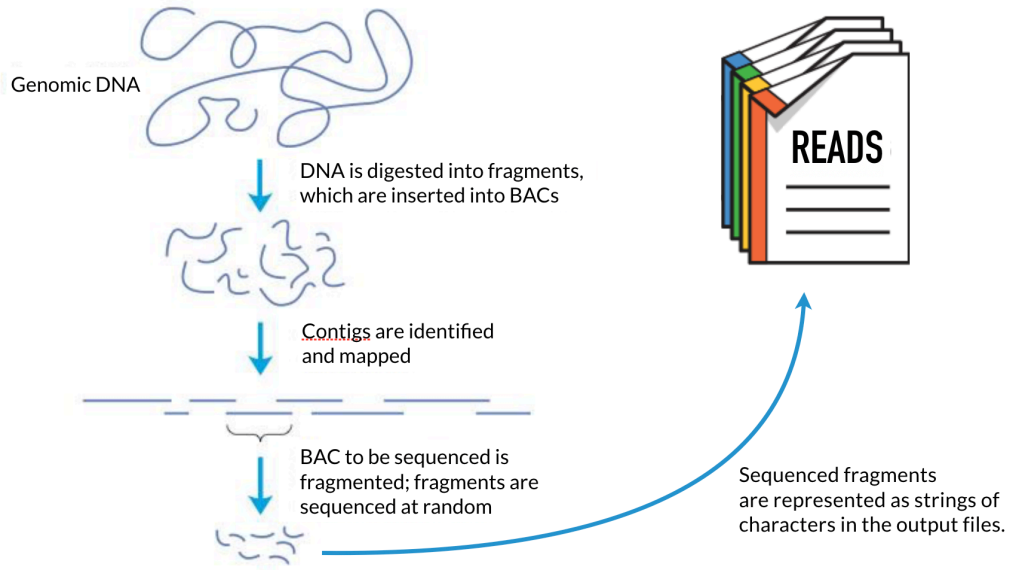

Figure 2.3: Shotgun sequencing overview

These methods can be divided into three main categories, based on the length of the reads they produce, expressed in base pairs (bp):

- *short read*: ranging from 35 to 700 bp;
- *long read*: up to 80.000, with an average of 30.000 bp;
- *ultra-long read*: up to 1.000.000 bp, average depending on the experiment.

Short read technologies, led by Illumina platforms [24], are usually characterized by the highest throughput and scalability, and the lowest error rate (0,1%) and per sample cost [25]. Moreover, in addition to "classical" single-end reads, obtained by reading the DNA fragment only from one of its ends, many of these technologies can also produce *paired-end* and *mate pair* reads. In particular, in paired-end reading when the sequencer finishes one direction at the specified read length, it starts another round from the opposite end of the fragment.

Knowing the distance between each paired read helps to improve their specificity and effectively resolve structural rearrangements, as insertions and deletions. Although mate pair sequencing involves a much more complex process, the result is similar: a pair of reads separated by a known distance. The difference between the two variants is the length of the insert: 200-800 bp for paired-end, 2-5 kbp for mate pair reads.

Long read technologies, in particular Pacific BioSciences platforms [26], are able to produce reads at the highest speed, thus providing a good throughput. Long reads are useful to resolve structural rearrangements and repetitive regions, but their error rate on a single pass is still high (13%). The accuracy can be improved, reducing the error rate to 0,001%, with about 20 passes, but this approach significantly enlarges the cost and the time required for the experiments [25].

Finally, Oxford Nanopore Technologies platforms [27] can sequence ultra-long reads with, in principle, no upper limit to read length. They are characterized by good scalability, from extremely small and portable to extremely powerful and high throughput. However, their error rate is the highest (15%) and they suffer from systematic errors with homopolymers. Improved accuracy can be achieved (3% error rate) by reducing the throughput, at the cost of lower efficiency [25].

2.2 Genome assembly

The simplicity of the shotgun sequencing process captured the interest of mathematicians and computer scientists and led to the rapid development of the theoretical foundations for genome sequence assembly. Because of the randomness of the fragmentation process, the individual fragments could be expected to over-

lap one another, and these overlaps could be recognized by comparing the DNA sequences of the corresponding fragments. Scientists initially studied the computational complexity of the assembly problem formalized as an instance of the shortest common superstring problem, the problem of finding the shortest string that encompasses all the reads as substrings [28]. However, the shortest common superstring problem ignores an important feature of complex genomes: repeats. Most genomes contain DNA segments that are repeated in nearly identical form, thus the correct reconstruction of a genome sequence may not be the shortest. The need to explicitly account for repeats has led to new models of sequence assembly, but with such formulations, genome assembly can be shown to be computationally intractable [29]. Anyway, these negative theoretical results seem to be disconnected to the obvious practical success in sequence assembly witnessed over the past few decades. An explanation for this gap between theory and practice is that theoretical intractability results are based on worst-case scenarios, which rarely occur in practice [30]. For this reason, in the last years practical assemblers were able to produce approximate, still very accurate, genome assemblies, using different approaches that fall into several broad paradigms, outlined below.

2.2.1 De novo genome assembly

De novo genome assembly is very similar to solving a jigsaw puzzle, but without knowing the complete picture we are attempting to reconstruct. Rather all we have are the individual pieces (i.e. sequence reads) and the only way to build the final product is by understanding how each piece connects to another. Thus, de

novo assembly constitutes a very complicated problem and, over the last decades, many approaches have been developed to resolve it.

The *greedy approach* works by making the locally optimal choice at each stage, hoping to find the global optimum. It starts by taking an unassembled read and then extends it by using the current read's best overlapping read on its 3' end. If the contiguous sequence, called *contig*, cannot be extended further, the process is repeated at the 5' end of the reverse complement of the contig. Despite its simplicity, this approach and its variants provide a good approximation for the optimal assembly of small genomes, and many early genome assemblers relied on such a greedy strategy [31]. Anyway, greedy assemblers tend to misassemble big genomes by collapsing repeats and, as the size and complexity of the genomes being sequenced increased, they were replaced by more complex algorithms [32].

The *Overlap-Layout-Consensus (OLC)* approach, as the name suggests, consists of three steps. In the first one, an overlap graph is created by joining each read to its respective best overlapping one, similarly to the greedy approach. The layout stage, ideally, is responsible for finding one single path from the start of the genome traversing through all the reads exactly once and reaching the end of the sequenced genome. Anyway, this ideal scenario is usually not achievable and is the reason why we have so many algorithms trying to find the solution using the same scheme, but different heuristics. Finally, in the consensus stage, groups of reads that overlapped are evaluated in order to identify, by majority voting, which base should be present at each location of the reconstructed genome [29]. Despite this approach to assembly has been very successful, it struggles when presented with the vast amounts of short-read data generated by NGS instruments. In practice, the overlap computation step is both a time and a memory bottle-

neck, as the memory requirements of these OLC assemblers grow quadratically with the depth of coverage and repeat copy number [30].

Graph-based sequence assembly models were developed to efficiently manage a vast amount of data, and they are the state of the art for the de novo genome assembly with high-throughput short-read sequence data. One of the most widespread methods is based on the *de Bruijn graph*, which has its roots in theoretical work from the late 1980s [33]. This approach consists in breaking each read into a sequence of overlapping *k-mers*, very short sequences of length k. The distinct k-mers are then used as vertices in the graph assembly, and k-mers that overlap by $k-1$ positions (e.g. originated from adjacent positions in a read) are linked by an edge. The assembly problem can then be formulated as finding a walk through the graph that visits each edge in the graph once: an Eulerian path problem [34]. In practice, sequencing errors and lack of coverage make it impossible to find a complete Eulerian tour through the entire graph, thus the assemblers attempt to construct contigs consisting of unambiguous, unbranching regions of the graph.

The diffusion of de Bruijn graph approach is partially due to its significant computational advantage with respect to overlap-based assembly strategies: it does not require expensive procedures to identify overlaps between pairs of reads. Instead, the overlap between reads is implicit in the structure of the graph [30]. Moreover, a distinct advantage of de Bruijn graphs over traditional OLC methods lies in the way the redundancy of high coverage NGS data is handled and exploited. As each node must be unique, multiple overlaps between different reads just increase the count (frequency) of each k-mer contained in these overlaps, but not the number of nodes in the graph. As a result, de Bruijn graphs scale with the size and complexity of the target genome rather than the amount and length of the

input reads. The main problem of this approach is that sequencing errors drastically increase the complexity of de Bruijn graphs by introducing additional "fake" k-mers and therefore spurious branches [35]. Moreover, valuable context information stored within the reads is lost for assembly, because the k-mers have to be shorter than the actual read length. As a result, theoretically possible connections between different k-mers, which are not found in the original reads, artificially increase the complexity of the graph [32].

For the above-mentioned reasons, overlap based assembly methods have been revised, by integrating key elements of the de Bruijn graph, resulting in the *string graph approach* [36]. In particular, the de Bruijn graph has the elegant property that repeats get collapsed: all copies of a repeat are represented as a single segment in the graph with multiple entry and exit points, providing a concise representation of the structure of the genome. A similar property can be obtained for overlap-based assembly methods by performing three operations. First, reads that are substrings of some other read are removed. Second, a graph is built where each sequence read is a vertex in the graph, and a pair of vertices are linked with an edge if they overlap. Third, transitive edges are removed from the graph. The result is a drastically simplified and directed overlap graph, called string graph, with basically the same properties and advantages as a de Bruijn graph, but more sequence information represented by every node and edge. However, a major disadvantage of overlap-based approaches remains, since the search for overlaps still requires alignments between every possible read combination. This makes all overlap based approaches, string graph as well as OLC, extremely time-consuming for large sequencing datasets. Anyway, assemblers utilizing the string graph method are definitively not out of the picture and they are even experienc-

ing a renaissance with the increase in read lengths of NGS technologies [37].

2.2.2 Comparative genome assembly

With the rapid growth in the number of sequenced genomes has come an increase in the number of organisms for which two or more closely related species have been sequenced. This led to the possibility of creating *comparative genome assembly* algorithms, which can assemble a newly sequenced genome by mapping it onto a similar reference genome [38].

Looking back at the analogy with the jigsaw puzzle, in comparative assembly we have access to the complete or partial picture of the final product, but some of our puzzle pieces are distorted or torn apart. Even though there is not a complete correspondence between the newly sequenced genome and the reference one, having a "backbone" sequence significantly easies the task of putting the pieces together in the right place. In fact, in order to properly order the massive number of reads sequenced by NGS instruments, we don't need anymore to compute the expensive pairwise alignments of all the reads against each other, as in overlap-based de novo aligners.

Instead, comparative assemblers usually rely on the *alignment-layout-consensus* paradigm [32]. In the alignment stage, reads are mapped to the reference genome, admitting multiple alignments of the same read to different genome's location, and resolving some of these ambiguities by using paired-end or mate pair reads. The alignment of all the reads produce a layout but, because we are aligning reads derived from an unknown *target* genome to a related reference genome that is slightly different, many reads might not align perfectly due to some mismatches,

insertions, deletions or structural variations. These issues are faced by iteratively refining the layout, resolving the divergent areas by generating a consensus sequence and then using that as a template for the alignment. This step is repeated until the issues identified above are not faced anymore.

Even if knowing the reference sequence remarkably reduce the complexity of the genome assembly problem, the alignment of a massive number of reads against a long reference sequence still constitutes a time bottleneck in the assembly process. For this reason, many aligners developed in the last decades rely on the *seed and extend* paradigm [39], consisting of two steps. First, short sequences called *seeds* are extracted from the reads set and are mapped onto the reference genome by looking for exact matches between the seeds and the reference sequence. Then, regions of the reference containing a high number of seeds from the same read are selected as candidates for the extension phase, that consists of aligning the reads to their candidate regions through a dynamic programming algorithm. This approach minimizes the search space, avoiding the need for computing several expensive alignments against the long backbone sequence, thus appreciably lowering the complexity of the assembly process.

2.3 Genomic analysis

The DNA sequence assembly alone is not very significant per se, as it requires additional analysis. Once a genome is sequenced and assembled, it needs to be annotated to make sense of it. *Genome annotation* is defined as the process of identifying the locations of genes and all of the coding regions in a genome and determining what those genes do [40]. The biological information is then at-

tached to the respective sequence element, in the form of an explanatory or commentary note. Thus, identified genes are referred to as *annotated*.

The genome annotation process consists of three steps:

1. identifying non-coding regions of the genome;
2. identifying coding elements on the genome, through a process called *gene prediction*;
3. attaching biological information to the identified elements;

Usually, automatic computer analysis and manual annotation involving human expertise co-exist and complement each other in the same *annotation pipeline*.

The first two steps of the annotation process are generally referred to as *structural annotation*. This part of the genomic analysis consists of identifying all and only the genomic elements of interest, like genes, regulatory motifs, and open reading frames. Information about their localization and structure are added to the assembled genome during this phase. The last step instead, namely *functional annotation*, consists of attaching to the identified elements information about their biochemical and biological function, expression, and involved regulation and interactions [41]. The most employed approach for gene annotation relies on homology-based search tools to look for homologous genes in specific databases; the resulting information is then used to annotate genes and genome. In practice, this means that a variety of known sequences are mapped onto the assembled genome, in order to identify some of its coding regions and understand their function.

Another important piece of the genomic analysis is *variant calling*, which is the identification and analysis of variants associated with a specific trait or population [42]. Genetic variations are the differences in DNA sequences between individuals within a population, and they can be caused by *mutations*, that are permanent alterations to the sequence due to errors during DNA replication, or by *recombination*, that is the mixing of parental genetic material which occurs when homologous DNA strands align and cross over. There are three main types of genetic variation:

- *Single Nucleotide Polymorphism (SNP):* substitution of a single base-pair;
- *Indels:* insertion or deletion of a single stretch of DNA sequence ranging from one to hundreds of base-pairs in length;
- *Structural Variation:* genetic variations occurring over a larger DNA sequence, as copy number variation or chromosomal rearrangement events.

These genetic differences between individuals are crucial not only because they lead to differences in an individual's phenotype, but also because they are related to the risk of developing a particular disease, due to their effects on protein structures [43]. Again, the variant calling process involves the expensive alignment of sequencing data to a reference genome, in order to identify where the aligned reads differ from it and classify these differences.

As introduced in Chapter 1, genome sequencing, assembly and analysis are the core of precision medicine, which takes into account individual variability in genes in order to find the best treatment for a particular disease, in a certain group of individuals. This approach has the potential for improving many aspects

of health and healthcare, and can provide a better understanding of the underlying mechanisms by which various diseases occur. For example, identifying the cancer-causing genes allows researchers to study drugs that target those genes, in order to deactivate them and attack the cancer cells. It is clear that, in an era where genome sequencing is becoming faster and cheaper, the main threat to the development and spreading of precision medicine is given by the long time and the high cost required by genome assembly and genomic analysis. In particular, sequence alignment constitutes the space and time bottleneck for most of the existing approaches to both these problems, and researchers are constantly looking for solutions that can lower the complexity of this task.

Because of the aforementioned reason, this thesis will present a space-efficient short read mapper, that can find application in de novo and comparative genome assembly, as well as in genome annotation or variant calling pipelines. Its user-friendly web-based interface and the support for standard file formats respectively guarantees high usability and easy integration in most of the existing bioinformatic pipelines. Moreover, the proposed tool leverages the FPGA architecture in order to obtain the best performance over power ratio, trying to match the needs of pharmaceutical industries and those of the researchers' community, in the aim of contributing to the spreading and to the democratization of precision medicine.

CHAPTER 3

Problem Description

To allow the reader a better understanding of the works analyzed in the literature, within this Chapter the Burrows-Wheeler Mapping problem and the methods employed in this work are presented. Section 3.1 gives a description of the standard file formats used to store genomic sequences, while Section 3.2 describes the Burrows-Wheeler Transform, the FM-Index, and the backward search algorithm. Then, Section 3.3 introduces the concept of Succinct Data Structure, focusing on the two structures employed in this implementation: RRR and wavelet trees. Finally, Section 3.4 gives an overview of the available hardware accelerators and a description of the FPGA architecture.

3.1 Genomic sequences: FASTA and FASTQ file format

As stated in Section 2.1, sequencing technologies digitalize the information contained in DNA fragments, encoding it in text sequences called reads. In the field

of bioinformatics, the universal standard is FASTA format, a text-based format for representing either nucleotide sequences (DNA or RNA) or amino acid sequences (protein) [44]. The format allows for sequence names and comments to precede the sequences, where nucleotides and amino acids are represented using single-letter codes. The header/identifier line, which begins with a '>', gives a name and/or a unique identifier for the sequence, and may also contain additional information, like National Centre for Biotechnology Information (NCBI) identifiers.

FASTQ format is an extension of FASTA format, for storing a nucleotide sequence and its corresponding quality scores, where both the sequence letter and quality score are encoded with a single ASCII character [45]. Originally developed at the Wellcome Trust Sanger Institute to bundle a FASTA formatted sequence and its quality data, it has recently become the de facto standard for storing the output of high-throughput sequencing instruments. A FASTQ file uses four lines per sequence:

1. *header line:* begins with a '@' character and is the equivalent of FASTA identifier line

2. *sequence line:* contains the raw sequence letters

3. *separation line:* begins with a '+' character and is optionally followed by the same information contained in the header line

4. *quality line:* encodes the quality values for the sequence letters in line 2, and must contain the same number of symbols as letters in the sequence

Quality values Q_i are integer mappings of p_i, that are the probabilities that the corresponding base calls are incorrect. The equation for calculating Q_i is the standard Sanger equation to assess the reliability of a base call, otherwise known as *Phred* quality score:

$$Q_i = -10 \log_{10}(p_i) \,. \tag{3.1}$$

3.2 Burrows-Wheeler Mapping

Finding a match between a query sequence and a long reference with a naive approach is a computationally intensive task, requiring to scan the whole reference sequence while looking for a certain pattern. For this reason, many approaches to resolve this task are based on indexing strategies, that allow to reduce the search space and speed up the matching process. Among these approaches, Burrows-Wheeler Mapping is well known because of its trivial time complexity. In practice, it consists of performing a backward pattern matching against a reference in a time that only depends on the pattern length, but not on the reference length. The backward pattern matching algorithm is presented in the remainder of this Section, together with the three concepts it is based on: suffix arrays, Burrows-Wheeler Transform, and FM-index.

Let $\mathcal{T}[1, N]$ be a sequence of length N drawn from a constant-size alphabet $\sum$, and let $\mathcal{T}[i, j]$ denote the substring of $\mathcal{T}$ ranging from i to j. The *Suffix Array* $\mathcal{SA}$ built on $\mathcal{T}$ is an array containing the lexicographically ordered sequence of the suffixes of $\mathcal{T}$, represented as integers providing their starting positions in the sequence. This means that an entry $\mathcal{SA}[i]$ contains the starting position of the

i-th smallest suffix in $\mathcal{T}$, and thus

$$\forall\, 1 < i \leq N \;:\; \mathcal{T}[\mathcal{SA}[i-1], N] \;<\; \mathcal{T}[\mathcal{SA}[i], N]\,. \tag{3.2}$$

It follows that if a string $\mathcal{X}$ is a substring of $\mathcal{T}$, the position of each occurrence of $\mathcal{X}$ in $\mathcal{T}$ will occur in an interval in the suffix array. This is because all the suffixes that have $\mathcal{X}$ as prefix are sorted together. Based on this observation, we define:

$$start(\mathcal{X}) = \min\{i : \mathcal{X} \text{ is the prefix of } \mathcal{T}[\mathcal{SA}[i], N]\} \tag{3.3}$$

$$end(\mathcal{X}) = \max\{i : \mathcal{X} \text{ is the prefix of } \mathcal{T}[\mathcal{SA}[i], N]\}\,. \tag{3.4}$$

The interval $[start(\mathcal{X}),\ end(\mathcal{X})]$ is called the *suffix array interval* of $\mathcal{X}$, and the positions of all the occurrences of $\mathcal{X}$ in $\mathcal{T}$ are contained in the set

$$\{\mathcal{SA}[i] : start(\mathcal{X}) \leq i \leq end(\mathcal{X})\}.$$

Thus, pattern matching is equivalent to searching for the $\mathcal{SA}$ interval of substrings of $\mathcal{T}$ that match the query.

The suffix array of a string is closely related to its *Burrows-Wheeler Transform (BWT)*, which is a reversible permutation of the characters in the string [46]. This transformation was originally developed within the context of data compression because it rearranges a text into runs of similar characters, making it easily compressible. However, since BWT-based indexing allows large text to be searched efficiently, it has been successfully applied to bioinformatic applications, such as oligomer counting [47] and sequence alignment [11, 48, 49]. The

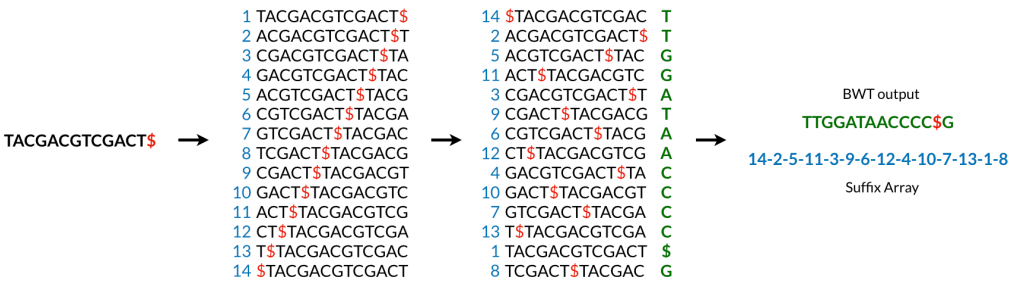

Figure 3.1: Burrows-Wheeler Transform and Suffix Array construction

Burrows-Wheeler Transform of a sequence $\mathcal{T}$, $BWT(\mathcal{T})$, can be constructed as follows:

1. the character \$ is appended to $\mathcal{T}$, where \$ is not in the alphabet $\sum$ and it is considered lexicographically smaller than all symbols in $\sum$;
2. the Burrows-Wheeler matrix of $\mathcal{T}$ is constructed as the $[(N+1)*(N+1)]$ matrix whose rows contain all cyclic rotations of $\mathcal{T}\$$;
3. the rows are sorted into lexicographical order;
4. $BWT(\mathcal{T})$ is the sequence of characters in the rightmost column of the Burrows-Wheeler matrix.

Figure 3.1 gives a graphical representation of the $BWT(\mathcal{T})$ construction, and it shows how it is possible to compute the suffix array of $\mathcal{T}$ at the same time.

The Burrows-Wheeler matrix has a very important property called *last-first mapping*: the i-th occurrence of character c in the last column corresponds to the same text character of the i-th occurrence of c in the first column. This feature underlies algorithms that use BWT-based indexes to navigate or search the text,

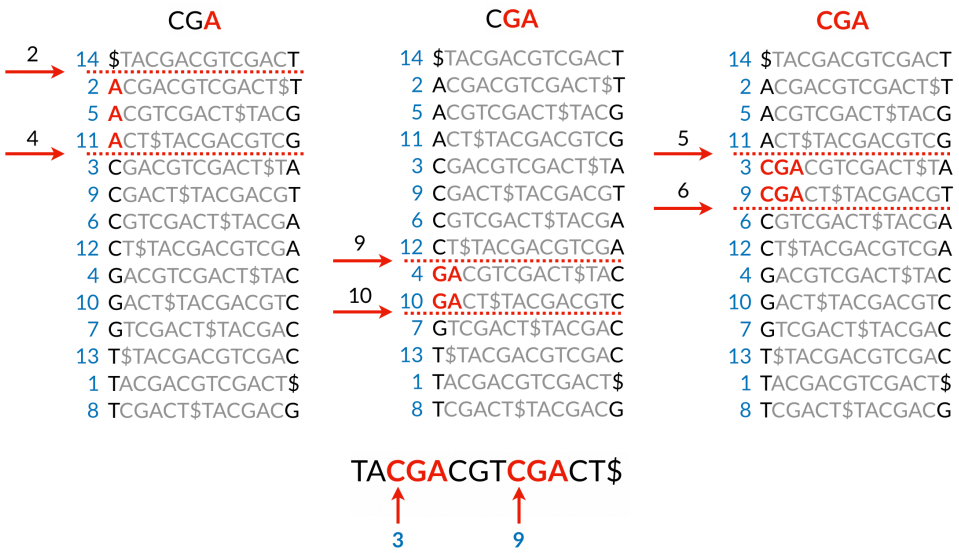

Figure 3.2: The backward pattern matching procedure

like the backward pattern matching algorithm employed in this work. Since the matrix is sorted in lexicographical order, rows beginning with a given sequence appear consecutively. The backward search procedure, showed in Figure 3.2, iteratively calculates the range of matrix rows beginning with longer suffixes of the query. At each step, the size of the interval either shrinks or remains the same. When the execution stops, rows beginning with the entire query sequence correspond to occurrences of the query in the text, and their positions can be found in the correspondent range of the suffix array.

Ferragina and Manzini, to whom the backward pattern matching algorithm can be attributed, presented two data structures that allow to rapidly execute the search:

- *the array C* stores, for each symbol in $\sum$, the number of characters with a lower lexicographical value; thus, $C(x)$ is the number of symbols in $BWT(\mathcal{T})$

that are lexicographically smaller than x, $\forall x \in \sum$;

- *the matrix Occ* stores the number of occurrences of each symbol up to a certain position in $BWT(\mathcal{T})$; thus $Occ(x, i)$ is the number of occurrences of x in $BWT(\mathcal{T})[1, i]$.

The authors proved that if $\mathcal{X}$ is a substring of $\mathcal{T}$:

$$start(a\mathcal{X}) = C(a) + Occ(a, start(\mathcal{X}) - 1) + 1 \tag{3.5}$$

$$end(a\mathcal{X}) = C(a) + Occ(a, end(\mathcal{X})) \tag{3.6}$$

and that $start(a\mathcal{X}) \leq end(a\mathcal{X})$ if and only if $a\mathcal{X}$ is a substring of $\mathcal{T}$. This result makes it possible to count the occurrences of $\mathcal{X}$ in $\mathcal{T}$ in $O(|\mathcal{X}|)$ time, by iteratively calculating $start$ and end from the end of $\mathcal{X}$, as showed in Algorithm 1.

3.3 Succinct Data Structures

Every living thing on Earth, from humans to bacteria, has a genome. The size of a genome varies a lot from species to species and it seems to be only slightly related to the complexity of the organism it belongs. For instance, the genome of Viroids (smallest known pathogens) is only ~ 300 nucleotides long, while the human genome has a length of more than 3.2 billions bp. Anyway, our genome is not the biggest one; actually, it looks quite small if compared to that of Amoeba Dubia, a very tiny creature, whose genome is ~ 670 billions bp long, making it the biggest known genome. This means that, assuming to encode each nucleotide in a byte, we would need 670 GB of memory just to store the sequence. Given this

Input: The query pattern $P[1, p]$
Output: Number of occurrences of $P[1, p]$
begin
  $i = p$;
  $c = P[p]$;
  $start = C[c] + 1$;
  $end = C[c + 1]$;
end
while $((start \leq end)$ ***and*** $(i \geq 2))$ **do**
  $c = P[i - 1]$;
  $start = C[c] + Occ[c][start - 1] + 1$;
  $end = C[c] + Occ[c][end]$;
  $i = i - 1$;
end
if $start \leq end$ **then**
  return *"found* $(end - start + 1)$ *occurrences"*
end
else
  return *"pattern not found"*
end

Algorithm 1: Backward search algorithm

knowledge, it turns out that storing this enormous amount of data in a compact space, and making it easily accessible, is a challenging but crucial requirement for enabling some bioinformatic applications. Here is where succinct data structures come into play.

Succinct data structures are defined as data structures which use an amount of space close to the information-theoretic lower bound, but still allow for efficient query operations. Unlike general lossless compressed representations, succinct data structures have the unique ability to use data in-place, without decompressing them first. A succinct data structure, encoding a sequence $\mathcal{T}[1, N]$, has to allow three operations on the data:

1. $access(\mathcal{T}, i)$: returns the i-th element of $\mathcal{T}$, $\mathcal{T}[i]$;

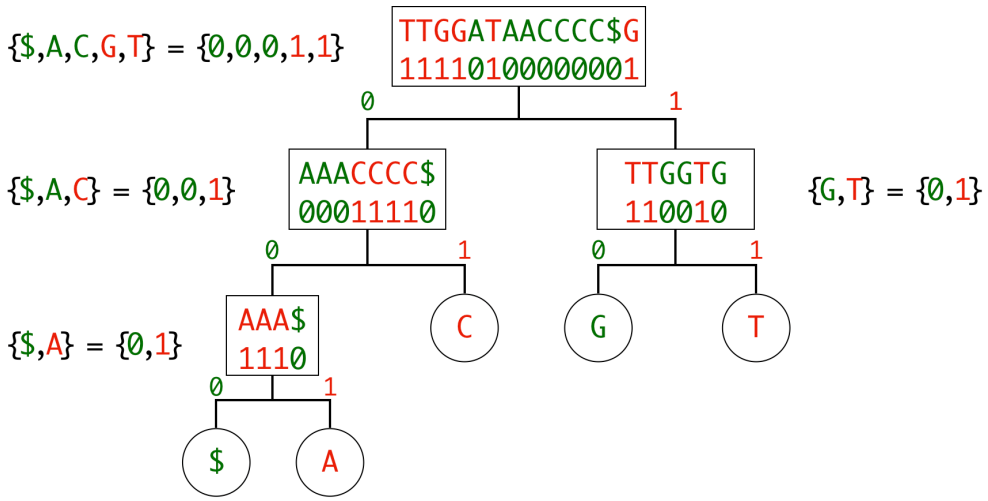

Figure 3.3: Wavelet tree of the sequence TTGGATAACCCC$G

2. $rank_C(\mathcal{T}, i)$: returns the number of occurrences of character C in $\mathcal{T}[1, i]$;

3. $select_C(\mathcal{T}, i)$: returns the position of the i-th character C in $\mathcal{T}$, with $1 \leq i \leq rank_C(\mathcal{T}, N)$.

In this work, a combination of two different data structures is employed to encode the reference sequence in an efficient way: wavelet trees and RRR sequences.

3.3.1 Wavelet Tree

A *wavelet tree* is a succinct data structure to store a string $\mathcal{T}[1, N]$, from an alphabet $\sum$, in a compressed space of $N \log_2(|\sum|)$ bits. Wavelet trees convert a string into a balanced binary tree of bit-vectors, where a 0 replaces half the symbols of the alphabet, and a 1 replaces the other half. At each level, the alphabet is filtered and re-encoded, in order to reduce the ambiguity, until there is no ambiguity at all. The tree is defined recursively as follow:

1. encode the first half of the alphabet as 0, the second half as 1 (e.g. {A,C,G,T} would become {0,0,1,1});
2. group each 0-encoded symbol (e.g. {A,C}) as a sub-tree;
3. group each 1-encoded symbol (e.g. {G,T}) as a sub-tree;
4. recursively apply this steps to each sub-tree until there are only 1 or 2 symbols left (no more ambiguity, a 0 or a 1 can only encode one symbol).

Figure 3.3 shows the wavelet tree of the sequence $\mathcal{T}$ = TTGGATAACCCC$G. Please note that the sequences are only showed for the seek of clarity, while in the actual implementation only bit-vectors are stored in the nodes of the wavelet tree. After the tree is constructed, a rank query can be done with $\log_2(|\sum|)$ binary rank queries on the bit-vectors. For example, if we want to know $rank_A(\mathcal{T}, 11)$, we use the following procedure. Knowing that A is encoded as 0 at the first level, we calculate the binary rank query of 0 at position 11, which is 6. Then we use this value to indicate where to calculate the binary rank of the 0-child. This procedure is applied recursively until a leaf node is reached, as illustrated in Figure 3.4.

3.3.2 RRR Data Structure

The *RRR* theoretical data structure is named after its creators: Raman, Raman, and Rao [50]. The purpose of RRR is to encode a bit sequence $\mathcal{B}[1, N]$ in such a way that supports $O(1)$ time binary rank queries. Moreover, it provides implicit compression, requiring $NH_0(\mathcal{B})+o(N)$, where $H_0(\mathcal{B})$ is the *zero-order empirical entropy* of $\mathcal{B}$. For a sequence $\mathcal{X}$ from an alphabet $\sum$ of size $|\sum| = \sigma$, it and can

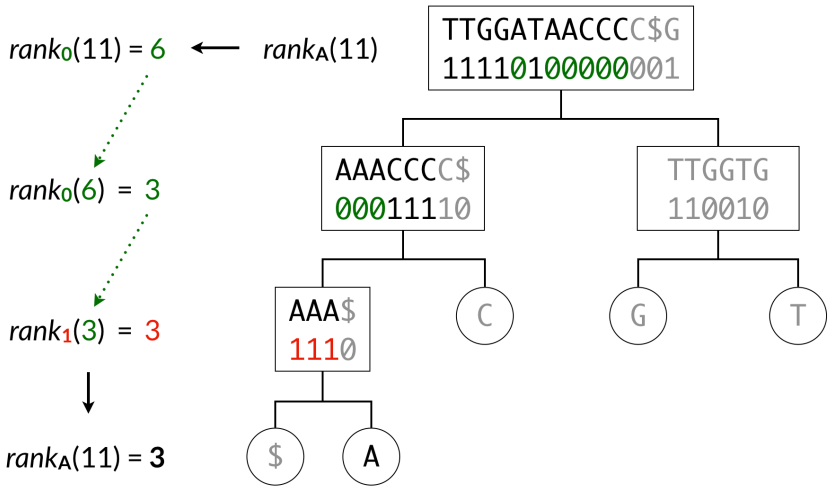

Figure 3.4: A rank query on the wavelet tree

be calculated as

$$H_0(\mathcal{X}) = \sum_{i=1}^{\sigma} \frac{N_i}{N} \log(\frac{N}{N_i}) \tag{3.7}$$

where N_i is the number of occurrences of the i-th element of $\sum$ in the sequence $\mathcal{X}$. The RRR structure can be constructed as follow:

1. divide the bit-vector $\mathcal{B}$ into blocks of b bits each;
2. group these blocks in superblocks of $sf \times b$ bits and, for each superblock, store the sum of previous $ranks_1(\mathcal{B})$ at the superblock boundary;
3. for each block, store a *class* number c, that is the number of 1s in the block;
4. build a table P, which has subtables $P[c]$ for each class c. For each permutation c_i of c 1s over a b-bits block, $P[c_i]$ contains an array of cumulative sums for each position in the block;

5. for each block, store an offset *off* which is an index into the table P pointing to the permutation corresponding to the considered block.

The resulting data structure can answer $rank_1(\mathcal{B}, i)$ query in constant time, according to the following procedure:

1. calculate which block the index is in as $i_b = \frac{i}{b}$;
2. calculate which superblock the index resides in as $i_s = \frac{i_b}{sf}$;
3. set the temporary result to the sum of previous ranks at i_s boundary;
4. add to the result the rank of entire blocks (their classes) after i_s, until i_b is reached;
5. finally, add $P[\mathit{off}(i_b)][j]$, where $j = i \bmod b$.

The final result is the answer to the initial $rank_1(\mathcal{B}, i)$ query.

3.4 Hardware Accelerators

Bioinformatic applications usually require to execute computationally intensive algorithms on a vast amount of data. Indeed, a software implementation on a general purpose Central Processing Unit (CPU) is the fastest to realize, because this platform offers good flexibility and an easy design process. Anyway, such a choice could be inefficient from an energetic and execution time perspective. Following the needs of pharmaceutical industries and those of the researchers' community, moving the implementation to hardware is the best way to optimize bioinformatic algorithms. In this context, there are different platforms upon which the user can choose to implement the algorithm, such as Application Specific Integrated

Circuits (ASICs), Graphical Processing Units (GPUs) and Field Programmable Gate Arrays (FPGAs). ASICs are the best solution both from a performance and power consumption point of view. However, the design and development cost of an ASIC is very high and only justified in case of a large number of deployed systems. In addition, these systems are not reconfigurable, so not flexible. It follows that, once the chosen algorithm is implemented, no changes can be made to the design. Reconfigurability is a feature of both GPUs and FPGAs, so both of them are pretty flexible solutions. GPUs, thanks to their architecture, can considerably accelerate certain types of algorithms, although their power consumption is substantially high. FPGAs, instead, offer a good trade-off between performance and power consumption. Thanks to their flexibility and lack of a fixed hardware architecture, FPGAs can be configured and reconfigured by the user to implement different functions. A hardware implementation on FPGA would give the benefit of a speedup over a pure software implementation while reducing power consumption. Moreover, the development cost of an application to run on an FPGA is remarkably lower compared to the ASIC one.

At the highest level, FPGAs are reprogrammable silicon chips. An FPGA is an array of logic blocks (cells) placed in an infrastructure of interconnections, which can be programmed at three distinct levels: the function of the logic cells, the interconnections between cells, and the inputs and outputs (Figure 3.5). All three levels are configured via a string of bits that is loaded from an external source. Any logic block (Figure 3.6) contains a Look-Up Table (LUT), a register and a multiplexer (MUX), and it can perform different functions, depending on how the Static Random Access Memory (SRAM) is programmed. The content of the SRAMs cells determines the content of the LUTs and, subsequently, the logic

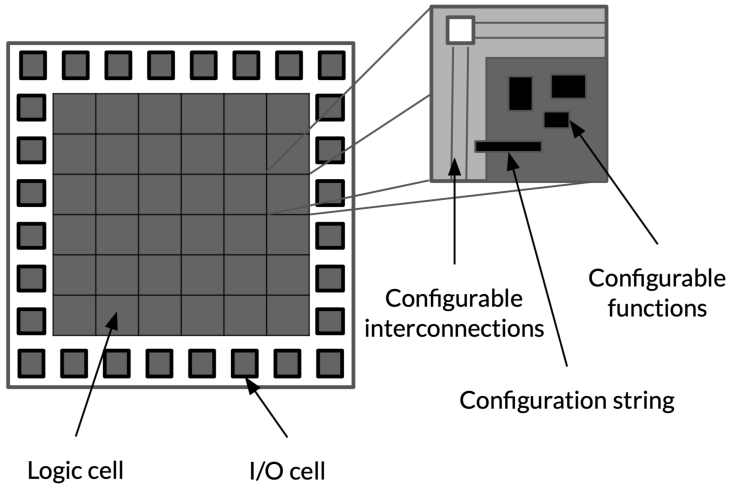

Figure 3.5: FPGA overview

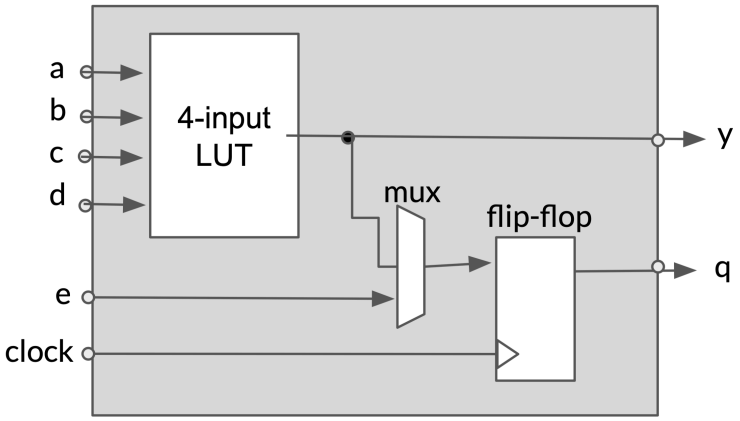

Figure 3.6: A logic block

function realized. The registers act as flip-flops in sampling, and the inputs they receive are selected by the multiplexers.

Modern FPGA families expand upon the above capabilities to include higher level functionality fixed into the silicon. Having these common functions embedded into the silicon reduces the area required and gives those functions increased speed compared to building them from primitives. Examples of these include multipliers, generic Digital Signal Processor (DSP) blocks, embedded processors, high speed I/O logic and embedded memories. These components integrated into the FPGAs aren't reprogrammable (except the connections to which they are connected) and perform specific functions, allowing to lighten the weight of the logic functions that the reprogrammable logic blocks have to implement.

Although the possible advantages of using such an architecture are clear, designing on an FPGA is a complicated procedure, as the design process is different from regular software development. Fortunately, in the last years, many toolchains have been created, aiming to facilitate this procedure. Among these, the *SDAccel* toolchain by Xilinx [51] is a complete design environment, providing raised abstraction for accelerators, simplified host integration and automated infrastructure creation. Users are supported during the whole hardware design flow: *High-Level Synthesis (HLS)* eases the creation and optimization of the hardware cores, and a guided *System-Level Design (SLD)* allows their implementation on the targeted board.

For the aforementioned reasons, the FPGA seems to be the best target for the proposed application. In fact, by leveraging its reconfigurability capabilities is possible to create a hardware implementation aiming at power over performance ratio optimization, in accordance with the needs of the bioinformatic community.

CHAPTER 4

Related work

This Chapter presents a review of the State of the Art for the employed methodologies. Section 4.1 explains the different approaches found in literature for the read alignment against large references, with particular emphasis on the Burrows-Wheeler Mapping. Section 4.2 will describe the details of the best performing hardware implementations of the Burrows-Wheeler based alignment, while Section 4.3 will present the difficulties bioinformaticians have to face because of the low usability of open source tools.

4.1 Read alignment and Burrows-Wheeler mapping

As introduced in Chapter 2, read alignment is the computational bottleneck of several bioinformatic applications, such as de novo and comparative genome assembly, variant calling and genome annotation. For this reason, in the last years, researchers put a big effort into the development of different approaches and al-

gorithms that could lower the complexity of this task. The vast majority of sequence aligners relies on *Dynamic Programming (DP)*, an optimization method consisting in storing the results of sub-problems so that it is not necessary to recompute them when needed later. Dynamic Programming can be used to address the sequence alignment problem in all its formulations (i.e. global, local, glocal). *Global alignment* algorithms find the optimal end-to-end alignment of two sequences, thus they are particularly useful when comparing two similar sequences that have almost the same length. Needleman-Wunsch algorithm [7] was one of the first applications of dynamic programming to compare biological sequences, and after almost 50 years since its publication, it is still the core of many global alignment tools used today [52]. Moreover, little modifications to Needleman-Wunsch algorithm make it suitable for finding the *glocal alignment* between two sequences, that is the best possible partial alignment where gaps at the beginning and/or at the end of one sequence (or even both) are ignored. This is particularly useful when the final part of one sequence overlaps with the beginning part of the other one, or when one sequence is much shorter than the other one [9]. Finally, *local alignment* algorithms search for optimal alignment between local portions of the given sequences, thus they are useful for dissimilar sequences that are suspected to contain a region of similarity within their larger sequence context. Smith-Waterman algorithm [8], which is also based on dynamic programming, is the most employed local alignment algorithm in the literature [52, 53].

Regardless of the chosen sequence alignment formulation, DP-based approaches search for the optimal alignment and have quadratic time complexity w.r.t the length of the aligned sequences [54]. Thus, employing these techniques alone often leads to prohibitive execution times, especially when the number of sequences

to be aligned is very high (e.g. overlap-based de novo genome assembly), or when sequences are very long (e.g. comparative genome assembly). For this reason, many aligners developed in the last years have coupled DP-based alignment algorithms with heuristics which allow to reduce the search space and speed up the alignment process. The most employed heuristic approach is called *seed and extend*, and it proved to achieve an excellent tradeoff between speed and accuracy [39]. This technique consists of two steps:

1. *seeding* is finding exact matches of parts of the query sequence with parts of the reference;

2. *extending* is executing a DP-based alignment algorithm in a local neighborhood of the regions identified in the seeding process.

Modern aligners use two kind of seeds: *fixed-length seeds* [12, 18, 55], and *maximal exact matching (MEM)* seeds [11, 56]. Fixed-length seeds, as the name suggests, are simply overlapping or non-overlapping substrings of the reads, all having the same length. MEM seeds, instead, are the longest matches which cannot be further enlarged in either direction [10].

In order to make seed and extend approach effective, the seeding step needs to be very fast, thus it usually requires a pre-built index of the reference genome. The majority of contemporary sequence aligners compute seeds by either using hash table indices [18, 55] or using BWT-based indices, such as the FM-index [11, 12, 57]. The latter offer an important advantage over the former: their memory usage is independent of the number of reads to be aligned, instead of growing linearly with it. The most relevant and widespread BWT-based aligners are the

Burrows-Wheeler Aligners (BWA [48], BWA-SW [58] and BWA-MEM [11]) and the two versions of Bowtie [49, 12].

BWA [48] and Bowtie [49] use modified versions of the backward search algorithm presented in Section 3.2 to calculate also inexact matches of the query with the reference, and even gapped alignments in the case of BWA. However, the time complexity of these strategies grows exponentially with the number of mismatches, thus these tools allow only few differences between sampled short substrings of the reference and entire query sequences [39]. In order to improve speed, fraction of reads aligned, and quality of the alignments, next versions of the aforementioned tools adopted a seed and extend strategy, coupling the FM-index with DP-based alignment algorithms. In particular, Bowtie2 [12] extracts fixed-length seeds and maps them onto the reference sequence through the same backward search process used in Bowtie, allowing up to a mismatch. Then it applies a seed alignment prioritization, associating to each seed hit (i.e. a locus of the reference sequence that matches the seed) a score which is inversely proportional to the number of loci matched by that seed. Finally, each reference locus is passed, according to its priority, to a SIMD-accelerated DP-based alignment algorithm (global or local), along with information about which seed string gave rise to the hit. BWA-SW [58], instead, builds FM-indices for both the reference and query sequence, representing the reference sequence in a prefix trie and the query sequence in a prefix Directed Acyclic Word Graph (DAWG). The two data structures are then aligned through dynamic programming, employing a couple of heuristics to speed up the process. Although this strategy reduces the number of computational expensive alignments to be computed, it leads to larger memory usage. Moreover, BWA-SW generally shows longer execution time w.r.t. Bowtie2

for reads shorter than 600 bp [12]. For this reason, in BWA-MEM [11] authors adopt a more classical formulation of the seed and extend paradigm. It initially uses the reference's FM-index and the backward search procedure to find, at each query position, supermaximal exact matches (SMEMs), which are the longest exact matches not contained in any other MEM of the query. To reduce mismappings caused by missing seeds, BWA-MEM introduces re-seeding for SMEMs that are longer than a given threshold. Then, it greedily chains colinear seeds that are close to each other, and filters out short chains that are largely contained in long chains in order to reduce unsuccessful seed extensions. Seeds are ranked by the length of the chain it belongs to and then by the seed length. Finally, they are extended with a banded DP algorithm, employing a heuristic that avoids extension through a poorly aligned region with good flanking alignment. This approach makes Burrows-Wheeler Aligner (BWA)-MEM one of the fastest read aligners openly available, without sacrificing its accuracy.

To the best of our knowledge, only two works in the literature [59, 60] leveraged wavelet trees for the Burrows-Wheeler mapping of short reads. In particular, Waidyasooriya et al. proposed an FPGA-based hardware architecture for text search that uses a wavelet tree based succinct data structures, and it will be further discussed in Section 4.2. Kumar et al. [59] proposed BWT-WT, an efficient read aligner able to find exact matches between very short queries and a reference, encoding the reference's BWT as a wavelet tree. Their tool relies on the Succinct Data Structures Library (SDSL) [61], a powerful C++ library implementing succinct data structures. SDSL implements wavelet trees encoding each node in an RRR data structure, in order to compress their size and answer rank/select queries in short time. The same approach is employed in this work and will be discussed

in detail in Chapter 5. In [59], the authors claim that their aligner achieves better compression and higher searching speed in comparison to other BWT-based tools as BWA and Bowtie. However, neither BWT-WT's source code nor its compiled binaries are publicly available, thus their results are not reproducible and a direct comparison is impossible. This makes the tool proposed in this thesis the only freely available, and open source, BWT-based read mapper which leverages wavelet trees and RRRs for achieving high search speed and low memory usage.

4.2 Hardware implementations

BWT-based aligners have been widely adopted by the bioinformatic community because of their accuracy and speed. However, even these fast tools showed long execution times when presented with the increasing amount of short-read data generated by NGS instruments. For this reason, in the last years, many researchers focused on hardware-accelerated implementations of these algorithms, experiencing a great performance improvement from their parallelization.

In the context of Burrows-Wheeler Mapping, GPUs and FPGAs have been used for speeding up both the computation of the BWT of the reference sequence and the backward mapping procedure. However, the BWT and the FM-index data structures are computed just once at the beginning of the alignment process, thus their creation only accounts for a minor part of the total execution time. Moreover, the BWT of a certain genome, and its FM-index, can be stored on disk after their computation, so that they can be easily accessed for further analysis without the need of calculating them again. For this reason, in this work, particular attention is paid to the strategies that allow efficient hardware imple-

mentations of the backward search algorithm, which is the real bottleneck of the Burrows-Wheeler mapping process.

4.2.1 GPU-based implementations

Graphical Processing Units (GPUs), thanks to their architecture, are ideal for accelerating highly parallelizable compute-intensive algorithms, so they became very attractive for the bioinformatic community. Liu at al. created SOAP3 [62], the first short read alignment tool that leverages the multi-processors in a GPU to achieve an improvement in speed. The main contribution of this work can be resumed in two points. First, they redesigned the FM-index data structure to reduce memory accesses as much as possible, while retaining the efficiency of the index. In fact, pattern searching on BWT requires many random memory accesses and a direct implementation of the CPU version on GPU would induce memory contention among different threads, degrading the overall performance drastically. The other difficulty that Liu et al. had to face is that GPU works in a Single Instruction Multiple Thread (SIMT) mode, where processors in the same unit must execute the same instruction. Too many diverging branches in the execution path would lead some of the processors to idle. Thus, they derived a useful parameter to determine in runtime whether a pattern would introduce too many branches, in order to stop the execution of these patterns, group them and re-execute the alignment in another round to reduce the idle time of processors. These strategies allowed to achieve speed-ups of up to $7.5\times$ over BWA and Bowtie, while improving the fraction of reads aligned, permitting up to 4 mismatches.

In order to enable multiple mismatches and gaps, which is well suited for real data alignments, the second version of SOAP3, called SOAP3-dp [63], adopts a seed and extend approach. The methods employed in SOAP3 are used for the seeding step, but without allowing mismatches between the query and the reference. For what concern the seed extension, the authors exploited compressed indexing and memory-optimizing dynamic programming in order to tackle a large number of candidate regions in parallel on GPU. This allowed them to examine gapped alignments extensively and achieve a good improvement in both speed (up to $2.63\times$) and sensitivity (up to 28%) over the previous version of the tool, proving the effectiveness of the seed and extend approach in conjunction with hardware accelerators.

Despite these positive results, it should be noticed that when GPUs are used, the speed-up in performance is paid in terms of power consumption. In fact, as already said earlier, GPUs are very power consuming compared to other hardware accelerators and even to general purpose CPUs. Field Programmable Gate Arrays (FPGAs), instead, thanks to their reconfigurability and a reduced clock frequency, proved to be very power efficient. Furthermore, since the algorithms and data structures employed in the BWT-based alignment do not require floating point operations, FPGAs' highly parallel architecture has been vastly investigated in literature for speeding up this process.

4.2.2 FPGA-based implementations

Several efforts have been made to accelerate sequence alignment using FPGAs, exploiting their high parallelism. Among them, FPGA-based acceleration of the

Smith-Waterman local alignment algorithm is the most common approach [14]. Olson et al. [64] implemented a seed and extend alignment algorithm, using a hash table for seeding and Smith-Waterman algorithm for the extension step. Both the seed location and score table computation are performed in hardware, while the backtracking (i.e. the reconstruction of the actual alignment) is executed on CPU. Their design is partitioned into 8 boards, each comprising a single Xilinx Virtex-6 FPGA, and can align 50 million reads in less than a minute, but achieving only a low fraction of mapped reads.

Fernandez et al. [65] presented the first FPGA-based hardware implementation of the FM-index. In their work, the index data structures required for the backward search (C and Occ) are stored in on-chip Block Random Access Memory (BRAM). The design is implemented on a single Xilinx Virtex-6 FPGA and can exactly align 1000 reads in 60.2 μs. However, storing the whole Occ matrix led to an early saturation of the FPGA's BRAM, thus their implementation only supports reference sequences long up to 460000 bp, which is often insufficient even for small bacterial genomes. In their second work, the authors proposed FHAST [66], an extension of their previous design which also allows for approximate alignment (up to 2 mismatches). This is done through multiple exact matching modules aligning the permuted reads. In this implementation, the FM-index data structures are not stored in the BRAM anymore, thus it is suitable also for longer references. Even if this choice impacts negatively the performance of the proposed design, executing FHAST on the Convey HC-1 platform with four Xilinx Virtex-5 FPGAs still led to a speed-up of $2.43\times$ compared to Bowtie.

In [67], Arram et al. presented a runtime reconfigurable architecture based on FM-index for accelerating short read alignment using FPGAs. Their archi-

tecture exploits the reconfigurability of FPGAs to develop an alignment design which supports exact and approximate alignment with up to 2 mismatches. The proposed alignment design comprises 3 stages: exact alignment, one mismatch alignment, and two mismatches alignment. The reads are first processed by the exact alignment module. Then, the FPGA fabric is reconfigured and any unaligned read is processed by the slower one- and two-mismatches alignment modules. This approach, implemented on 8 Altera Stratix V FPGAs, each connected to 48GB of DRAM, led to a $28\times$ speed-up and a $29\times$ energy reduction compared to the CPU-based Bowtie2 aligner, with a 7% reduction in accuracy. Moreover, their design achieved a $9\times$ speed-up and a $25\times$ energy reduction compared to the GPU-based Soap3, without accuracy loss.

In [68] instead, the same authors presented an FPGA implementation of a seed and extend algorithm, based on FM-index for the seeding step and Smith-Waterman algorithm for the extension one. Even this implementation shows a reconfigurable design, where only exact matching units are implemented at the beginning. When enough reads which cannot be exactly mapped onto the reference are stored, the design reconfigures the FPGA to allow inexact matching units to start processing those reads. The authors claim that their design can achieve a performance of 129 millions *bases per second* on a single Xilinx Virtex-6 FPGA, which is $78\times$ faster than BWA on CPU, and $35\times$ faster than SOAP3 on GPU. However, their performance results are only an estimate and cannot be measured from a full implementation.

As previously introduced, in parallel to the work described in this thesis, Waidyasooriya et al. [60] proposed an FPGA-based hardware architecture for text searching that leverages a wavelet tree based succinct data structures. Their

design consists of two DDR3 RAM memories, where the rank data of the text are stored, and an array of *Processing Elements (PEs)*, which process in parallel the search queries, sent in batches to the FPGA. The main contribution of their work is the data structure they propose, which considers the hardware specification. It consists of code words, made of a header and a body whose sizes depend on the memory model (i.e. how many bits are accessed from the memory by each read) and on the number of symbols in the text. The bit-vectors representing the wavelet tree's nodes are divided into blocks which are stored in the bodies of the words. For each word, the header contains the partial rank of the corresponding bitvector, up to the block encoded in the previous word. Thus, rank queries on the text are made through fast binary rank queries on the bitvectors, one for each level of the wavelet tree. The proposed designed is able to store the encoded data in a space that is just 5.5% larger than the original data size. However, the authors did not report the performance of their architecture in terms of execution time.

It is important to note that, among the presented works, those which achieve the most significant results in the target application all leverage a system with multiple FPGAs and high quantity of RAM. Our architecture instead provides a significant speed-up over state-of-the-art equivalent software applications, leveraging a single FPGA and using a limited quantity of RAM, therefore it can be easily replicated to obtain even better performances.

4.3 Usability of the alignment tools

Although memory usage and execution time are key aspects one should take into account when choosing an aligner for a bioinformatic project, they are not the

only ones. In fact, an optimal tool in terms of speed and time performance is completely useless if it is employed by the bioinformatic community. It turns out that a fundamental features to consider is the tool's *usability* or *ease of use*. Usability is defined as the degree to which a tool can be used by end users to achieve quantified objectives with effectiveness, efficiency, and satisfaction. Even though researchers have put a big effort into developing new alignment algorithms and improving their performances, low usability is still a critical aspect of many available tools.

The vast majority of alignment tools, even most modern ones, is based on command-line user interfaces, that make them only accessible through the terminal. This leads to a series of difficulties that bioinformaticians have to face when using these programs. Actually, troubles can begin even before the usage of a program, during its installation. In fact, open source aligners usually require downloading the source code, compiling it, and finally installing it. Although accurate instructions are generally provided by the creators of the tools, many problems can arise because of missing dependencies, user permissions, and system specifications. This is particularly true when installing a tool on a server, rather than on a personal computer.

Fortunately, in the last years, package management systems were created, aiming to help users to find and install open source tools. A popular example is Conda [69], a package manager able to quickly install, run, and update packages and their dependencies. In particular, Bioconda [70] is a channel for the Conda package manager specialized in bioinformatic software. However, many tools are not accessible yet through the Bioconda channel and, as previously said, the installation is not the only problem to face when dealing with available open source applica-

```
Bowtie 2 version 2.3.5 by Ben Langmead (langmea@cs.jhu.edu, www.cs.jhu.edu/~langmea)
Usage:
  bowtie2 [options]* -x <bt2-idx> {-1 <m1> -2 <m2> | -U <r> | --interleaved <i>} [-S <sam>]

  <bt2-idx>  Index filename prefix (minus trailing .X.bt2).
             NOTE: Bowtie 1 and Bowtie 2 indexes are not compatible.
  <m1>       Files with #1 mates, paired with files in <m2>.
             Could be gzip'ed (extension: .gz) or bzip2'ed (extension: .bz2).
  <m2>       Files with #2 mates, paired with files in <m1>.
             Could be gzip'ed (extension: .gz) or bzip2'ed (extension: .bz2).
  <r>        Files with unpaired reads.
             Could be gzip'ed (extension: .gz) or bzip2'ed (extension: .bz2).
  <i>        Files with interleaved paired-end FASTQ/FASTA reads
             Could be gzip'ed (extension: .gz) or bzip2'ed (extension: .bz2).
  <sam>      File for SAM output (default: stdout)

  <m1>, <m2>, <r> can be comma-separated lists (no whitespace) and can be
  specified many times.  E.g. '-U file1.fq,file2.fq -U file3.fq'.

Options (defaults in parentheses):

 Input:
  -q                 query input files are FASTQ .fq/.fastq (default)
  --tab5             query input files are TAB5 .tab5
  --tab6             query input files are TAB6 .tab6
  --qseq             query input files are in Illumina's qseq format
  -f                 query input files are (multi-)FASTA .fa/.mfa
  -r                 query input files are raw one-sequence-per-line
  -F k:<int>,i:<int> query input files are continuous FASTA where reads
                     are substrings (k-mers) extracted from a FASTA file <s>
                     and aligned at offsets 1, 1+i, 1+2i ... end of reference
  -c                 <m1>, <m2>, <r> are sequences themselves, not files
  -s/--skip <int>    skip the first <int> reads/pairs in the input (none)
  -u/--upto <int>    stop after first <int> reads/pairs (no limit)
  -5/--trim5 <int>   trim <int> bases from 5'/left end of reads (0)
  -3/--trim3 <int>   trim <int> bases from 3'/right end of reads (0)
  --trim-to [3:|5:]<int> trim reads exceeding <int> bases from either 3' or 5' end
                     If the read end is not specified then it defaults to 3 (0)
  --phred33          qualities are Phred+33 (default)
  --phred64          qualities are Phred+64
  --int-quals        qualities encoded as space-delimited integers

 Presets:                 Same as:
  For --end-to-end:
   --very-fast            -D 5 -R 1 -N 0 -L 22 -i S,0,2.50
   --fast                 -D 10 -R 2 -N 0 -L 22 -i S,0,2.50
   --sensitive            -D 15 -R 2 -N 0 -L 22 -i S,1,1.15 (default)
   --very-sensitive       -D 20 -R 3 -N 0 -L 20 -i S,1,0.50

  For --local:
   --very-fast-local      -D 5 -R 1 -N 0 -L 25 -i S,1,2.00
   --fast-local           -D 10 -R 2 -N 0 -L 22 -i S,1,1.75
   --sensitive-local      -D 15 -R 2 -N 0 -L 20 -i S,1,0.75 (default)
   --very-sensitive-local -D 20 -R 3 -N 0 -L 20 -i S,1,0.50

 Alignment:
  -N <int>           max # mismatches in seed alignment; can be 0 or 1 (0)
  -L <int>           length of seed substrings; must be >3, <32 (22)
  -i <func>          interval between seed substrings w/r/t read len (S,1,1.15)
  --n-ceil <func>    func for max # non-A/C/G/Ts permitted in aln (L,0,0.15)
  --dpad <int>       include <int> extra ref chars on sides of DP table (15)
  --gbar <int>       disallow gaps within <int> nucs of read extremes (4)
  --ignore-quals     treat all quality values as 30 on Phred scale (off)
  --nofw             do not align forward (original) version of read (off)
  --norc             do not align reverse-complement version of read (off)
  --no-1mm-upfront   do not allow 1 mismatch alignments before attempting to
                     scan for the optimal seeded alignments
  --end-to-end       entire read must align; no clipping (on)
   OR
  --local            local alignment; ends might be soft clipped (off)

 Scoring:
  --ma <int>         match bonus (0 for --end-to-end, 2 for --local)
  --mp <int>         max penalty for mismatch; lower qual = lower penalty (6)
  --np <int>         penalty for non-A/C/G/Ts in read/ref (1)
  --rdg <int>,<int>  read gap open, extend penalties (5,3)
  --rfg <int>,<int>  reference gap open, extend penalties (5,3)
  --score-min <func> min acceptable alignment score w/r/t read length
                     (G,20,8 for local, L,-0.6,-0.6 for end-to-end)

 Reporting:
  (default)          look for multiple alignments, report best, with MAPQ
   OR
  -k <int>           report up to <int> alns per read; MAPQ not meaningful
   OR
  -a/--all           report all alignments; very slow, MAPQ not meaningful

 Effort:
  -D <int>           give up extending after <int> failed extends in a row (15)
  -R <int>           for reads w/ repetitive seeds, try <int> sets of seeds (2)

 Paired-end:
  -I/--minins <int>  minimum fragment length (0)
  -X/--maxins <int>  maximum fragment length (500)
  --fr/--rf/--ff     -1, -2 mates align fw/rev, rev/fw, fw/fw (--fr)
  --no-mixed         suppress unpaired alignments for paired reads
  --no-discordant    suppress discordant alignments for paired reads
  --dovetail         concordant when mates extend past each other
  --no-contain       not concordant when one mate alignment contains other
  --no-overlap       not concordant when mates overlap at all

 Output:
  -t/--time          print wall-clock time taken by search phases
  --un <path>        write unpaired reads that didn't align to <path>
  --al <path>        write unpaired reads that aligned at least once to <path>
  --un-conc <path>   write pairs that didn't align concordantly to <path>
  --al-conc <path>   write pairs that aligned concordantly at least once to <path>
    (Note: for --un, --al, --un-conc, or --al-conc, add '-gz' to the option name, e.g.
    --un-gz <path>, to gzip compress output, or add '-bz2' to bzip2 compress output.)
  --quiet            print nothing to stderr except serious errors
  --met-file <path>  send metrics to file at <path> (off)
  --met-stderr       send metrics to stderr (off)
  --met <int>        report internal counters & metrics every <int> secs (1)
  --no-unal          suppress SAM records for unaligned reads
  --no-head          suppress header lines, i.e. lines starting with @
  --no-sq            suppress @SQ header lines
  --rg-id <text>     set read group id, reflected in @RG line and RG:Z: opt field
  --rg <text>        add <text> ("lab:value") to @RG line of SAM header.
                     Note: @RG line only printed when --rg-id is set.
  --omit-sec-seq     put '*' in SEQ and QUAL fields for secondary alignments.
  --sam-no-qname-trunc Suppress standard behavior of truncating readname at first whitespace
                       at the expense of generating non-standard SAM.
  --xeq              Use '='/'X', instead of 'M,' to specify matches/mismatches in SAM record.
  --soft-clipped-unmapped-tlen Exclude soft-clipped bases when reporting TLEN

 Performance:
  -p/--threads <int> number of alignment threads to launch (1)
  --reorder          force SAM output order to match order of input reads
  --mm               use memory-mapped I/O for index; many 'bowtie's can share

 Other:
  --qc-filter        filter out reads that are bad according to QSEQ filter
  --seed <int>       seed for random number generator (0)
  --non-deterministic seed rand. gen. arbitrarily instead of using read attributes
  --version          print version information and quit
  -h/--help          print this usage message
```

Figure 4.1: Bowtie2 command-line user interface

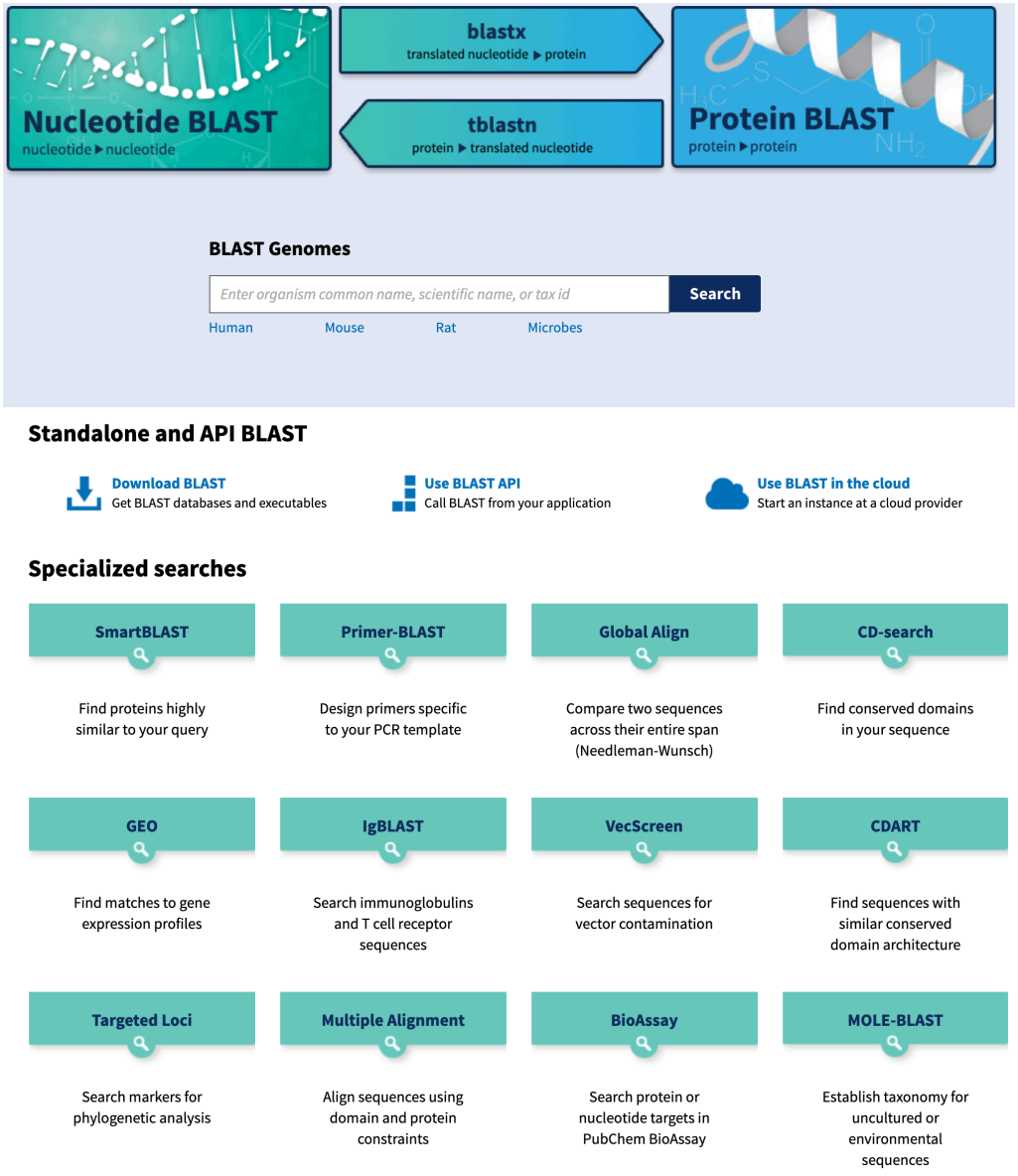

Figure 4.2: BLAST web-based user interface

tions.

Command-line user interfaces are often difficult to use, requiring users to write long complicated commands, thus providing a bad user experience. This is particularly true when the employed tool allows users to choose among a vast pool of options. A noticeable example is Bowtie2, whose extensive list of available commands and options is shown in Figure 4.1. Although the availability of many different commands and parameters guarantees high flexibility to a tool, it often leads to very long learning times and low overall usability.

For this reason, in order to improve the user experience, some tools rely on intuitive graphic user interfaces. The most renowned example is the Basic Local Alignment Search Tool (BLAST), which is the most employed bioinformatic tool ever, with more than 75000 citations in published research papers [18]. The diffusion of this tool is related to different factors, among which the choice of various specific applications, the availability of many sequencing databases, and, last but not least, a user-friendly web-based interface [71] (Figure 4.2). Another important example is represented by Bowtie2 itself, whose first web-based frontend has been released in march 2019 [17], in its beta version. Bowtie2 UI is the first attempt at a web interface for the Bowtie family of aligners, aiming at improving its usability. Given this knowledge, it looks clear the importance of an intuitive graphic interface for boosting the diffusion of a tool among the bioinformatic community.

4.4 Summary

This chapter provided a description of the principal approaches employed in the literature for the sequence alignment, with particular regard to the Burrows-Wheeler mapping of short reads.

The overview of the current state of related works shows that the seed and extend paradigm can lead to a great tradeoff between the speed and the accuracy of the alignment. For this reason, in the proposed work only the exact matching problem is tackled, aiming to the implementation of a BWT-based exact mapper to be used both as a standalone exact aligner, or in conjunction with DP-based alignment algorithms in a seed and extend gapped aligner.

Also, wavelet trees and RRR data structures proved to be effective in reducing the memory footprint of the mapping process, thus they are further investigated in this work. The encoding scheme proposed here, given by the combination of such data structures, thanks to its small memory footprint, is suitable for a hardware implementation on FPGA, which already showed their great potential in speeding up the matching process and reducing its power consumption. Moreover, FPGAs can provide specialized structures that have the potential to imrove the execution efficiency of the bit-level operations required for accessing information in the proposed succinct data structure.

Finally, it has been demonstrated how the usability of open source alignment tools is critical for their adoption by the bioinformatic community, while command-line user interfaces usually provide a bad user experience. Thus, the proposed tool is specifically thought to be accessible through an intuitive user-friendly interface, in order to guarantee high ease of usage and short learning

time. This allows users to benefit of acceleration on FPGAs, with little to no knowledge of the underlying hardware architecture.

In conclusion, the tool proposed in this thesis is the only easily accessible, freely available, BWT-based read mapper which leverages succinct data structures and FPGAs, for achieving high search speed, small memory footprint, and low power consumption.

CHAPTER 5

Proposed solution: Design & Implementation

This Chapter provides a detailed description of the proposed solution and of its implementation. Section 5.1 presents an outline of the proposed sequence mapper, explaining its components and their interactions. Section 5.2 will describe the software implementation of the Burrows-Wheeler Transform computation and of the backward search algorithm, starting from the naive solutions and explaining all the optimizations. Section 5.3 will present the hardware implementation of the backward search algorithm, focusing on the optimizations made to leverage FPGAs' parallelism and bit-level operations. Finally Section 5.4 will describe the web-based user interface, and how to use the proposed system.

5.1 Outline of the proposed sequence mapper

In order to explain the single processing steps implemented for efficiently mapping sequence reads onto a large reference, it is useful to start from a global description of the proposed system, allowing the reader to have a comprehensive idea. The proposed sequence mapper, called BWaveR, is a web-based heterogeneous system leveraging FPGAs for improving performances and reducing the energy consumption. Figure 5.1 shows an outline of the proposed hybrid system. The platform is accessed through an HTML-based web interface, which allows users to upload the reference sequence and the sequencing data to be mapped onto it, start the mapping process, and finally download an output file containing the results. The Python web-server, built with Flask [72], is in charge of the management of the whole process, transferring files from and to the user PC, and assigning tasks to the host CPU and to the FPGA co-processor(s).

The overall work-flow is presented in Figure 5.2, which highlights how it can be divided into three main steps:

1. *BWT and SA computation*: the reference sequence stored in the uploaded FASTA file is processed in order to compute its Burrows-Wheeler Transform and its suffix array;

2. *BWT encoding*: the BWT of the reference sequence is used to build the proposed succinct data structure, a balanced wavelet tree where each node is encoded as an RRR sequence;

3. *Reads mapping*: the sequence reads contained in the submitted FASTQ file are mapped onto the reference sequence through the backward search pro-

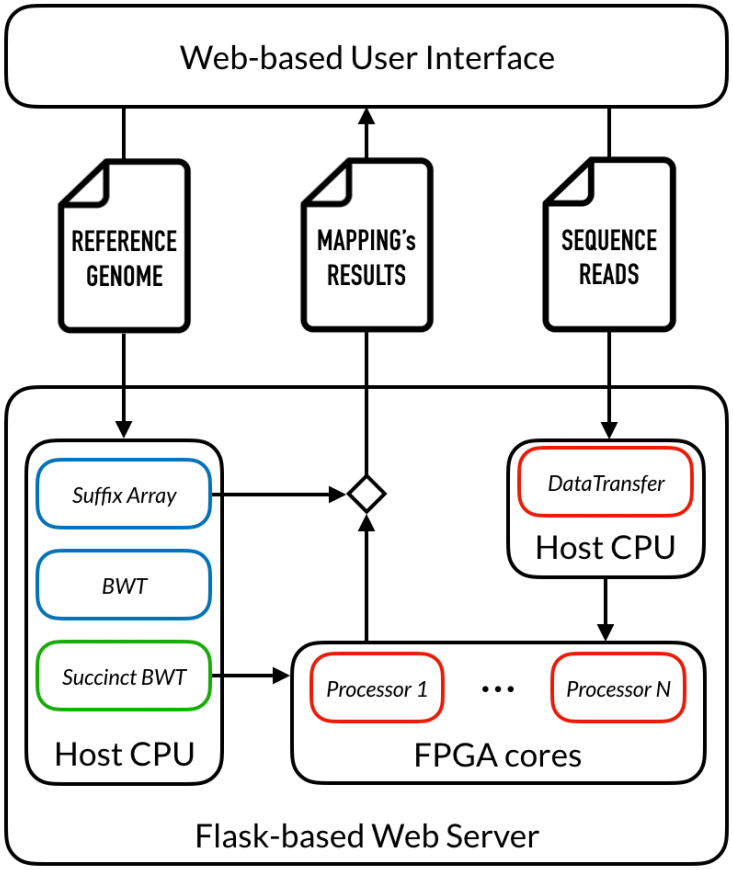

Figure 5.1: Outline of the proposed sequence mapper

cedure, employing the previously created data structure for fast queries on the BWT.

The host CPU is in charge of the first two processing steps and of all the memory management, while the reads mapping task, which is the time bottleneck of the application, is assigned to the FPGA co-processor(s).

The input reference and query sequences can be provided as FASTA and FASTQ files respectively. The system is able to handle both the uncompressed (.fasta and .fastq) and the compressed (.fa.gz and .fq.gz) formats, without performance

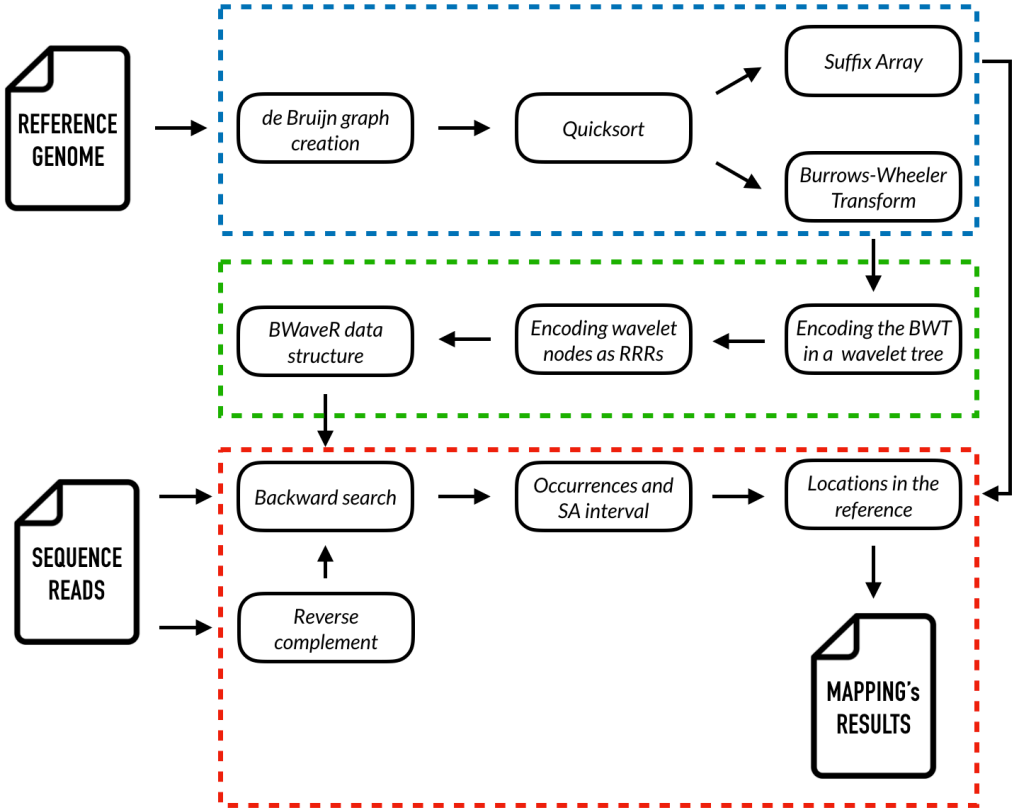

Figure 5.2: BWaveR sequence mapper: overall work-flow

degradation. The results of the mapping are encoded in the BWaveR output format and returned to the user as a compressed .txt.gz file, in order to reduce its download time. The proposed output format is a text-based human-readable format aiming to minimize the size of the resulting file while retaining all the useful information. At the beginning of the file there is a header with two fields: the first begins with the "@" symbol, followed by the name of the reference file, while the second begins with the "$" symbol, followed by the name of the sequencing data (i.e. reads) file. For each mapped read, the file contains an header line, starting with the ">" symbol, and an occurrences line. The header line provides:

```
@U00096.2.fas
$ecoli1000.fq

> r373/1 | [35 bp] | + 1:
    39353

> r376/1 | [35 bp] | + 2:
    2727106 3424705
> r376/1 | [35 bp] | - 5:
    697882  472945  431543  604067  4413850

> r378/1 | [35 bp] | + 1:
    2120873
```

Figure 5.3: BWaveR output file format: an example

- the name of the read in the original FASTQ file;
- the length of the read in bp;
- the orientation of the reference where occurrences of the read are found ("+" for the original strand represented in the FASTA file, "−" for its complementary strand);
- the number of occurrences.

Finally, the occurrences line provides the positions of all occurrences of the read in the reference sequence (or in its reverse complement), separated by a space. Figure 5.3 shows some lines of an example output file produced by the BWaveR sequence mapper.

The choice of using a custom output format, instead of standard sequence alignment formats as .SAM and .BAM files, is driven by the aim of minimizing the size of the resulting file. The text-based Sequence Alignment Map (SAM) format and its binary equivalent, Binary Alignment Map (BAM) format, have been created for storing biological sequences aligned to a reference sequence [73].

In order to handle all the possible gapped alignments between two sequences, SAM and BAM files store much more information than what is needed in the case of exact matches. Thus, the size of the output file can be significantly reduced by keeping only the useful information, as is done in the BWaveR output format.

5.2 Software implementation

As already introduced, the work-flow of the proposed sequence mapper is made of three main steps, namely *BWT and SA computation*, *BWT encoding*, and *reads mapping*. All these steps have been implemented in a software written in the C++ programming language [74]. The backward search is the time and space bottleneck of the whole application, requiring to process a vast amount of sequencing data, thus the main focus of the proposed work was the optimization of this processing step and of the data structures employed in it. On the other hand, the *BWT* and its suffix array are computed only at the beginning of the whole mapping process for a given genome, and they can even be reused if we want to align a different set of reads to the same reference. However, this first processing step is both computationally expensive and memory intensive, thus it is worth to optimize it as well, in order to reduce its time and space complexity.

The definitions of Burrows-Wheeler Transform and of Suffix Array of a sequence were already given in Section 3.2. As previously said, computing the *BWT* of a string $\mathcal{T}$, of length N, involves two main steps: building the $[(N+1)*(N+1)]$ Burrows-Wheeler Matrix containing the cyclic rotations of $\mathcal{T}$, and sorting its rows in lexicographical order. Subsequently, this procedure has a quadratic complexity $O(N^2)$ both in time and space; given the massive size of genomic refer-

ence sequences, this can lead to impracticable memory usage and execution time. For the aforementioned reasons, two optimizations were applied in this work to reduce the execution time and the memory footprint of the *BWT* and *SA* computation.

The first one is the choice and the implementation of one of the most efficient sorting algorithm: *quicksort*, which allows to reduce the time complexity of the sorting procedure to $O(N \log N)$. The quicksort algorithm employs a divide and conquer strategy for efficiently sorting an array: it picks an element from the array, called pivot, then it partitions (i.e. reorder) the array so that smaller elements come before the pivot and greater elements come after it (after partitioning, the pivot is in its final position). This steps are recursively applied to the sub-arrays of elements with smaller and greater values, separately, until the original array is completely sorted [75].

The second optimization aims to reduce the memory usage of this first processing step exploiting de Bruijn Graphs, which were already introduced in Section 2.2.1. In order to construct the *BWT* of the reference sequence, and its suffix array, a customized de Bruijn graph of the reference sequence was built, according to the following procedure. First, the reference sequence is broken into a sequence of overlapping short sequences of length k, called k-mers. Then, k-mers are used as keys for indexing a hash table, whose elements store the positions of the occurrences of the correspondent k-mers in the reference sequence, together with the symbols that precede and succeed the k-mers in those positions. Thus, each entry of the hash table associates a k-mer to one or more *tuples*, which are

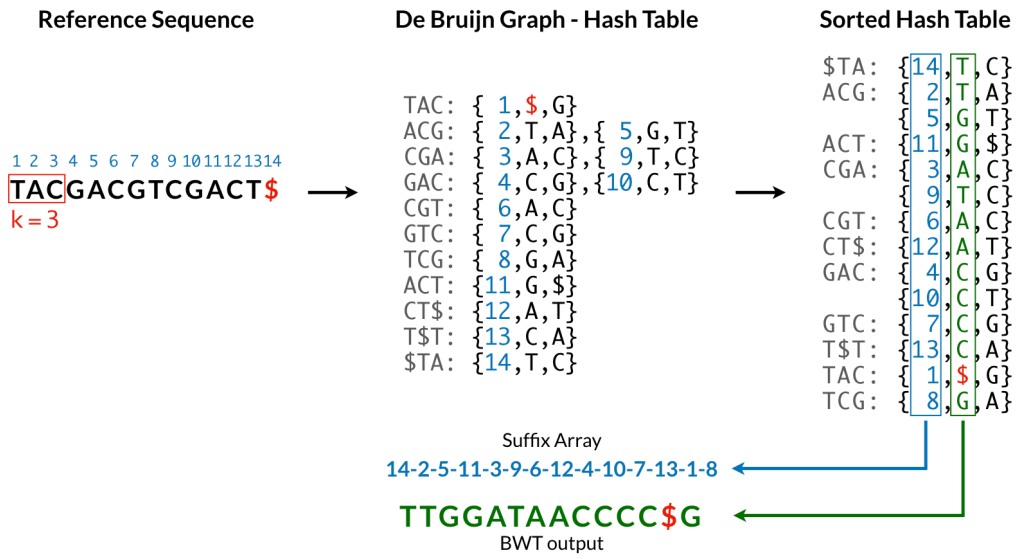

Figure 5.4: Optimized BWT and suffix array computation via de Bruijn graph

ordered sets defined as:

$$\{position\ in\ the\ reference,\ preceding\ symbol,\ following\ symbol\}.$$

Such a data structure allows to compute the *BWT* and the *SA* of the reference sequence in $O(N)$ space, significantly reducing the memory footprint of the proposed sequence mapper. In order to achieve this, first we have to resolve the ambiguity due to k-mers which occur more than once in the reference sequence. To do this, the tuples in each element of the hash table (i.e. corresponding to the same k-mer) are sorted according to the lexicographical order of the reference's substrings beginning at the positions contained in the tuples. This is done by navigating the de Bruijn graph, starting from those positions, until different "following symbols" are encountered. More in detail, to reach the following node of the graph, the new key (i.e. the new k-mer) is calculated by appending the "fol-

lowing symbol" to the current key, then the precise occurrence of the k-mer is selected, according to its "position in the reference". Finally, after the ambiguity is resolved, the *BWT* and the *SA* can be computed by sorting the elements of the hash table according to the lexicographical order of the corresponding k-mers. In fact, the sorted sequence of "preceding symbols" corresponds to the *BWT* of the reference sequence, while the sorted sequence of "positions in the reference" corresponds to its suffix array. The overall procedure is shown in Figure 5.4.

As already explained in Section 3.2, the *BWT* and the suffix array of the reference sequence $\mathcal{T}$ can be employed to perform an efficient backward search on the reference, able to find all the occurrences of a pattern of length p in $O(p)$ time. The backward search algorithm, presented in Algorithm 1, relies on the FM-index of the reference sequence, which can be built starting from its *BWT*.

The naive implementation of the FM-index is based on two data structure: the array C and the matrix Occ. Despite its simplicity, this approach provides the best performance in terms of execution time of the backward search. In fact, at each step of the pattern matching algorithm, the computation of the $start$ and end indices only requires a couple of memory accesses to each data structure, and two additions. Thus, it can be performed in trivial time. On the other hand, the matrix Occ requires $20 \times N$ bytes of memory to store the number of occurrences of each symbol up to each position in $BWT(\mathcal{T})$; this makes the naive implementation impracticable when dealing with large genomes, which can contain billions of bases.

For this reason, many different approaches were proposed in the literature to reduce the size of the FM-index, aiming at achieving the best tradeoff possible between memory usage and execution time. In this work, a combination of dif-

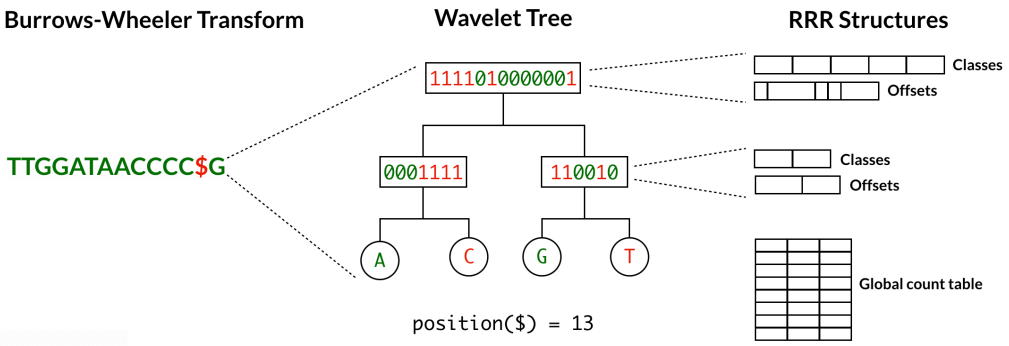

Figure 5.5: Overview of the BWaveR data structure

ferent succinct data structures is employed to encode the BWT of the reference sequence $\mathcal{T}$ in a minute space, while still allowing for fast queries on it. First, a balanced wavelet tree is employed to encode the transformed sequence as a series of bit-vectors. Then, each bitvector is encoded as an RRR structure, as showed in Figure 5.5.

The rationale behind the choice of this particular encoding is driven by the following consideration. Each memory access to the Occ matrix, $Occ(x, i)$, provides the number of occurrences of x in $BWT(\mathcal{T})[1, i]$, which is exactly the definition of a $rank$ query. Thus:

$$Occ(x, i) \equiv rank_x(\mathcal{T}, i). \tag{5.1}$$

Subsequently, at each iteration of the backward search algorithm, the $start$ and end indices can be calculated as:

$$start(a\mathcal{X}) = C(a) + rank_x(\mathcal{T}, start(\mathcal{X}) - 1) + 1 \tag{5.2}$$

$$end(a\mathcal{X}) = C(a) + rank_x(\mathcal{T}, end(\mathcal{X})). \tag{5.3}$$

Given this knowledge, a balanced wavelet tree is used for encoding the *BWT*, since it provides a fast rank query on the transformed sequence through $\log_2(|\sum|)$ binary rank queries on the bit-vectors, where $|\sum|$ is the number of different symbols in $BWT(\mathcal{T})$. Then, the bit-vectors constituting the nodes of the wavelet tree are encoded as RRR sequences, which compress the data while allowing fast binary rank queries.

Despite the availability of open source libraries implementing succinct data structures, such as the Succinct Data Structures Library (SDSL) [61], a custom C++ library has been designed from scratch for the BWaveR sequence mapper, in order to guarantee total control over the implemented data structures and facilitate the hardware implementation on FPGA.

The wavelet tree has been implemented as a hierarchy of wavelet nodes, each declared as a *struct* containing five fields:

- a *bit-vector*, implemented as an RRR sequence;
- two pointers to wavelet nodes structs, pointing, respectively, to the *child-zero* node and the *child-one* node of the current wavelet node;
- two arrays of characters containing, respectively, the *alphabets of the child nodes* of the current wavelet node.

Such a struct allows to recursively construct the balanced wavelet tree of any arbitrary sequence from any arbitrary alphabet, and answering rank queries on it, following the procedures illustrated in Section 3.3. This guarantees high flexibility of usage, making the proposed wavelet tree implementation suitable for any application involving rank queries on large texts.

Nevertheless, since this C++ library was written for being used in the proposed sequence mapper, the data structure has been further optimized for representing genomic sequences. In fact, such sequences have an alphabet of only four elements: A, C, G, and T or Uracile (U), in case of RNA reads. Thus they could be represented in a wavelet tree with just two levels. However, the Burrows-Wheeler transformation introduces the special character "$" in the sequence, enlarging the alphabet size, and subsequently requiring a third deeper level in the tree. This would negatively affect both the size of the data structure and the time required to perform a rank query on it. To avoid such a drawback, the proposed implementation does not store the special character "$" in the wavelet tree. Instead, its position in the *BWT* is stored in a separate variable (Figure 5.5), which is checked in the backward search function to adjust the rank queries used for calculating the $start$ and end indices.

Although the wavelet tree is already a powerful data structure, the core of the proposed encoding of the *BWT* sequence is the implementation of bit-vectors, based on the theoretical proposal in [50] by Raman, Raman and Rao (RRR). As previously explained, their theoretical data structure allows to encode a bit sequence in such a way that supports $O(1)$ time binary rank queries. This data structure relies on the division of the bit sequence $\mathcal{B}$ into *blocks* of b bits each, then grouped in *superblocks* of $sf * b$ bits, where sf is called *superblock factor*. In this work, we combine faithful implementation of the theory with some practical decisions, aiming to obtain the minimal memory footprint for our representation. In particular, each RRR sequence encoding a bit-vector $\mathcal{B}[1, N]$ has been implemented as a struct containing nine fields:

- an array of $\frac{N}{b}$ 4-bits elements, storing the *class* number c of each block (i.e. numbers of 1s in the block);
- an array of $\frac{N}{sf*b}$ unsigned integers (32 bits) storing the *partial sum* at each superblock left boundary (i.e. $rank_1(\mathcal{B}, e)$, where e is the position in $\mathcal{B}$ of the last element included in the previous superblock);
- a pointer to an array of *permutations* P, shared among all the RRRs, which contains 2^b short unsigned integers (16 bits) storing all the possible blocks of b bits, sorted per class;
- a pointer to a shared array of *class offsets* (unsigned integers) storing, for each class c, the offset to the first element in P belonging to that class, $P[c]$;
- an *offset* bit-vector containing $\frac{N}{b}$ fields of different lengths ($\lceil \log_2 \binom{b}{c} \rceil$ bits), storing, for each block, the offset between the class offset $P[c]$ and the index of the permutation in P corresponding to the considered block;
- an array, *offset sum*, of $\frac{N}{sf*b}$ unsigned integers storing, for each superblock, the starting position in the *offset* bit-vector of the field associated to the first block included in the considered superblock;
- three variables storing, respectively, the length of the bit-vector $\mathcal{B}$ (N), the size of the blocks (b), and the superblock factor (sf).

A schema of the described data structure is reported in Figure 5.6. Such an implementation guarantees high flexibility of usage, by allowing any sf value, and block sizes $b \leq 15$ which, according to theory[50], allow to optimally encode bit-vectors long up to $10^{2b} = 10^{30}$ elements. The main difference between

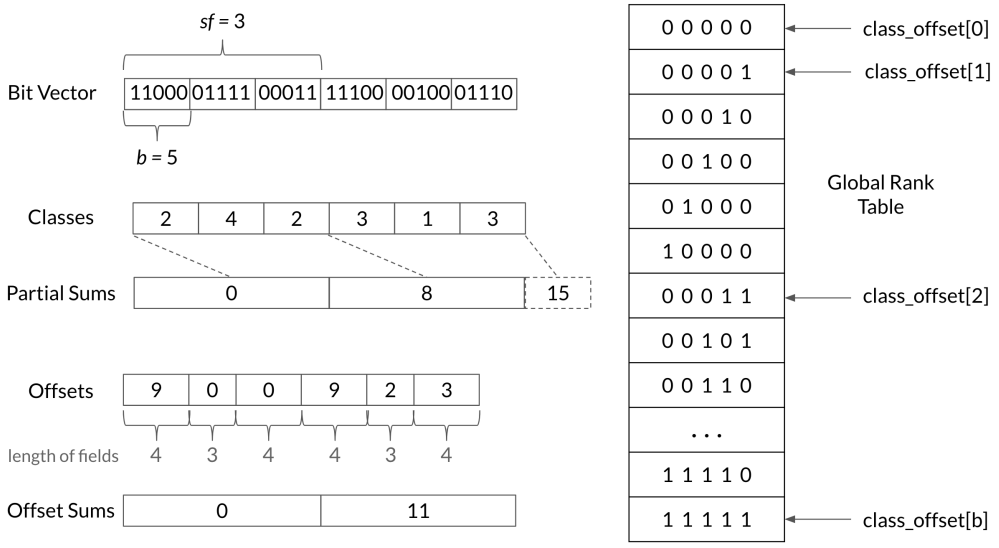

Figure 5.6: Overview of the implemented RRR sequence

the presented implementation and the theoretical proposal, presented in Section 3.3.2, is that we don't store the cumulative sums for each position of all possible b-bits blocks. Instead, we only store the permutations corresponding to the possible blocks, and we calculate the partial sum when needed. Although this choice introduces a little computational overhead, binary rank queries can be efficiently performed on the implemented structure, following the procedure showed in Algorithm 2. Moreover, our implementation allows to store the RRR sequence of a bit-vector $\mathcal{B}[1, N]$ in

$$\frac{N}{2b} + 2 \times \left(4\frac{N}{sf \times b}\right) + 2 \times 2^b + 4 \times (b+1) + 3 + \frac{\lambda}{8} = \\ = \frac{(sf+16)N}{2 \times sf \times b} + 2^{b+1} + 4b + 7 + \frac{\lambda}{8} \text{ bytes,}$$

where $\lambda = \sum_{i=0}^{N/b} \lceil \log_2 \binom{b}{c_i} \rceil$ is the total length (in bits) of the *offset* bit-vector.

It is important to notice that when multiple RRR sequences are employed,

Input: The RRR sequence of the bitvector $\mathcal{B}$ and the position p
Output: $rank_1(\mathcal{B}, p)$: number of 1s in $\mathcal{B}[1, p]$
$count = 0$;
if $(p \ mod \ (sf \times b)) == 0$ **then**
  return $PartialSum[p/(sf \times b)]$
end
else if $(p \ mod \ b) == 0$ **then**
  for $i = sf \times (p/(sf \times b))$ **to** p/b **do**
    $count = count + class[i]$;
  end
  return $PartialSum[p/(sf \times b)] + count$
end
else
  $OffsetPos = OffsetSum[p/(sf \times b)]$;
  for $i = sf \times (p/(sf \times b))$ **to** p/b **do**
    $count = count + class[i]$;
    $OffsetPos = OffsetPos + ceil(\log_2 \binom{b}{class[i]})$;
  end
  $off \leftarrow$ read $ceil(\log_2 \binom{b}{class[p/b]})$bits from the offset bit-vector;
  $cloff = ClassOffset[class[p/b]]$;
  $count = count + rank_1(Permutation[cloff + off], p \ mod \ b)$;
  return $PartialSum[p/(sf \times b)] + count$
end

Algorithm 2: Binary rank function on RRR sequences

as in the case of this work, the amount of space required for each structure is even lower, because the *permutations* array and *class offsets* array are stored only once, and shared among all the RRRs. Moreover, while all the other terms can be tuned by the choice of the b and sf parameters, the amount of memory required to store the *offset* array is mainly dependent on the zero-order empirical entropy of the bit sequence $\mathcal{B}$. This makes the proposed succinct data structure very effective at representing the *BWT* of long sequences. In fact, as previously said, the Burrows-Wheeler transformation rearranges a text into runs of similar characters, significantly reducing its entropy. Therefore, encoding the *BWT* in

a wavelet tree generates low-entropy bit-vectors, showing long runs of 1s or 0s, which can be efficiently exploited by the RRR data structure.

Finally, after the computation of the *BWT* and its compressed encoding, the resulting data structure can be employed for efficiently mapping reads onto the reference sequence. In order to find also the occurrences of the reads in the complementary strand of the reference sequence, the reverse complement of each read is computed and mapped onto the reference too. At the cost of a little computational overhead, this choice allows to avoid storing the reverse complement of the reference, which would almost double the memory usage of proposed sequence mapper. Both the reads and their complementary sequences are mapped onto the reference through a modified version of the backward search algorithm, showed in Algorithm 3.

The positions of the occurrences of a read $\mathcal{X}$ in the reference sequence are contained in the set

$$\{\mathcal{SA}[i] : start(\mathcal{X}) \leq i \leq end(\mathcal{X})\},$$

where $[start(\mathcal{X}), end(\mathcal{X})]$ is the suffix array interval of $\mathcal{X}$. Instead, the set of positions of the occurrences of $\mathcal{X}$ in the complementary strand of the reference sequence is calculated as

$$\{((N+1) - \mathcal{SA}[i] - n),\ \forall i : start(\overline{\mathcal{X}}) \leq i \leq end(\overline{\mathcal{X}})\},$$

where N and n are, respectively, the lengths of the reference and of the query sequence, and $[start(\overline{\mathcal{X}}), end(\overline{\mathcal{X}})]$ is the SA interval of the reverse complement

of $\mathcal{X}$.

Input: The query pattern $P[1, p]$ and the position of \$ in the BWT
Output: Number of occurrences and suffix array interval of $P[1, p]$
begin
  $i = p$;
  $c = P[p]$;
  $start = C[c] + 1$;
  $end = C[c + 1]$;
end
while $((start \leq end)$ ***and*** $(i \geq 2))$ **do**
  $c = P[i - 1]$;
  if $end < pos(\$)$ **then**
    $start = C[c] + rank_{WT}(c, start - 1) + 1$;
    $end = C[c] + rank_{WT}(c, end)$;
    $i = i - 1$;
  end
  else
    $end = C[c] + rank_{WT}(c, end)$;
    if $start < pos(\$)$ **then**
      $start = C[c] + rank_{WT}(c, start - 1) + 1$;
    end
    else
      $start = C[c] + rank_{WT}(c, start - 2) + 1$;
    end
  end
end
if $start \leq end$ **then**
  return *"(*end*-*$start$*+1) occurrences, SA interval = [*$start, end$*]"*
end
else
  return *"pattern not found"*
end

Algorithm 3: Backward search algorithm with succinct data structures

5.3 Hardware implementation

The first step required when approaching hardware accelerators is the study of the employed algorithms, in order to identify the most compute intensive functions. In this project, the profiling tool *Callgrind* and the visualization tool *KCachegrind* [76] were used to profile a naive implementation of the Burrows-Wheeler mapping, based on the Occ and c data structures presented in Section 3.2.

The analysis showed that the backward search algorithm is the computational bottleneck, occupying more than 90% of the execution time when a large number of reads were processed, as in real cases of application. However, implementing a naive version of the FM-index on FPGA would not have been a smart choice. In fact, in the naive implementation, the largest part of the time is employed for reading values from the Occ data structure, in order to recursively compute the $start$ and end indices. Moreover, a further analysis of backward search highlighted the random pattern of the memory accesses to the Occ matrix, which denies the possibility to pack the data for exploiting data locality. This, together with the low on-chip memory resources of FPGAs, makes the naive algorithm not suitable for a hardware acceleration.

Given this knowledge, the main focus of this project has been the design and implementation of a succinct data structure, which can be stored in a small memory space while still enabling the efficient backward pattern matching. This is achieved by the proposed encoding scheme, which introduces a computational overhead, leaving space for many hardware optimizations. In particular, FPGAs can provide specialized structures that increase the execution efficiency of bit-level operations required for accessing information in the proposed succinct data

structure.

As already said, our solution is a hardware-software codesign, where all the data structures used in the mapping are created by the host CPU. The mapping process instead has been implemented on FPGA, in order to improve the efficiency of the bit-level operations involved in the backward search. The design of the architecture has been described using C++ and Xilinx Vivado HLS[51]. Xilinx SDAccel[51] has been used to perform the next steps, from the synthesis of the custom core, to the generation of the configuration bitstream. SDAccel also takes care of the runtime management of the application. The host code of the application has been written using OpenCL[77], and the communication between the host and the FPGA device is performed using the OpenCL APIs.

Although the SDAccel Toolchain eases the design process on FPGA, various challenges raised when creating a custom hardware architecture leveraging the combination of a wavelet tree and RRR sequences to perform an efficent backward search. First, the software implementation presented in Section 5.2 makes extensive usage of dynamic memory allocation in order to exploit the compression ability of the encoding scheme. Clearly, dynamic memory allocation is not supported in High-Level Synthesis (HLS) tools, which prevents the creation of the data structure directly on the FPGA. Moreover, Vivado HLS does not allow the usage of nested data structures. Therefore, the implementation of the wavelet tree was modified by fixing its depth. In particular, instead of the hierarchical encoding, based on many structs containing a single wavelet node and pointers to its children nodes, the wavelet tree has been entirely stored in a single struct.

The lack of dynamic memory allocation on the FPGA leads to two possible approaches for leveraging the proposed data structure. The first one, consists of

fixing the dimensions of all the fields in the struct. Thus, one should know their sizes at design time in order to fully exploit the compression capability of the RRR sequences, resulting in an implementation suitable only for a specific reference sequence. The second approach, is storing the struct in an external memory, and trasferring on the board only the data required for computing the indices at a given iteration of the search algorithm. This strategy allows for high flexibility, but the latency introduced by the external memory accesses significantly degrades the performances of the system.

In this work, we aimed at finding the best tradeoff between the flexibility of the design and its performance. To this extent, we decided to overestimate the memory requirements at design time, in order to achieve a certain flexibility, at the cost of a reduced efficiency in resource utilization. However, to limit such drowback, we generated multiple cores with increasing dimensions of the data structure, up to the saturation of the FPGA's on-chip memory. Thus, after the BWT is encoded by the host CPU, we can select the design with the minimum memory usage that is still able to store the whole BWT structure. This guarantees high flexibility, allowing to efficiently use BWaveR with reference sequences of various length, while reducing the waste of on-chip resources.

A simple schematic of the developed design is presented in Figure 5.7, which shows all the communication channels employed to transfer data to and from the FPGA core. More in detail, the first channel (i.e. *r_h* in Figure 5.7) is employed to transfer query sequences to the custom kernel, and to get mapping results back to the host device. Instead, all the other channels are used to transfer, in parallel, the different fields of the reference structure to the backward search core.

Figure 5.7 also displays the FPGA resources used by the developed architec-

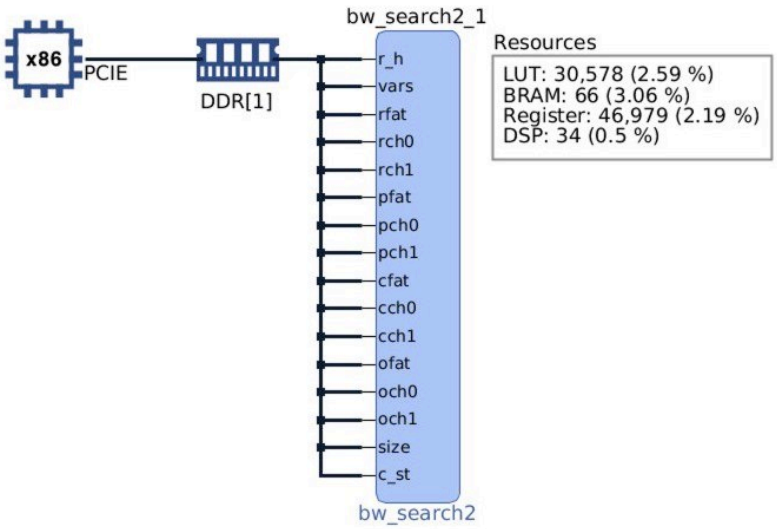

Figure 5.7: Schematic of the hardware design

ture. In particular, the presented resource utilization refers to a core handling reference sequences long up to 7.5 millions bp, that we employed in our experiments on bacterial genomes, discussed in Chapter 6. We can see that, as expected, the BRAM is the most used resource. However, the utilization is only $\sim$ 3%, which allows to replicate the kernel in a multi-core design to further boost the performances of the proposed system.

The execution flow of the mapping process starts by transferring the BWT data structure on the FPGA device. The data are then stored on the on-chip BRAM of the device, in order to increase data locality and avoid the long latency caused by accessing memory cells of an external DRAM. Since the BWT structure remains the same during the kernel execution, this data is loaded only once, at the beginning of the execution. Having this data always available on the device permitted us to simplify the design of the architecture and to improve the performances of the kernel.

For our hardware implementation, we fixed the block size $b = 15$ for each RRR sequence, while we decided to allow for any superblock factor $sf >= 50$, according to the user needs. This enabled us to pre-compute the static global count table and the class offsets of the RRR sequences, which can be efficiently accessed through lookup tables. Moreover, the low memory footprint of the resulting data structure allows us to load on a single FPGA genomic sequences as long as human chromosomes, containing up to $\sim$200 millions bp.

In order to leverage the full bandwidth on the FPGA board, each port of the FPGA is set to load 512 bits blocks to exploit memory burst. This optimization not only allows us to read more data per clock cycle, but also reduces the complexity of the routing and the logic circuits on the FPGA, as we would need to allocate smaller buffers.

After allocating the reference's data, the architecture iteratively fetches query sequences from the host's memory, map them and their reverse complement onto the reference, and send back the results to the host machine. We implemented the query as a 512-bit data structure, which stores the sequence to be searched and the $start$ and end indices for both the original sequence and its reverse complement. We modeled the query size to exploit the memory burst and, to store sequences long up to 176 bp, where each base is encoded as an arbitrary precusion integer of 2 bits. Moreover, although the maximum allowed length is enough for the target application, the query structure's size can be easily set equal to every multiple of 512-bit, according to the user needs, without performance degradation.

When a query sequence $\mathcal{X}$ is loaded on the FPGA, its reverse complement $\overline{\mathcal{X}}$ is computed by the kernel. Then, both $\mathcal{X}$ and $\overline{\mathcal{X}}$ are mapped onto the reference structure. The backward search for $\mathcal{X}$ and $\overline{\mathcal{X}}$ is executed in parallel and

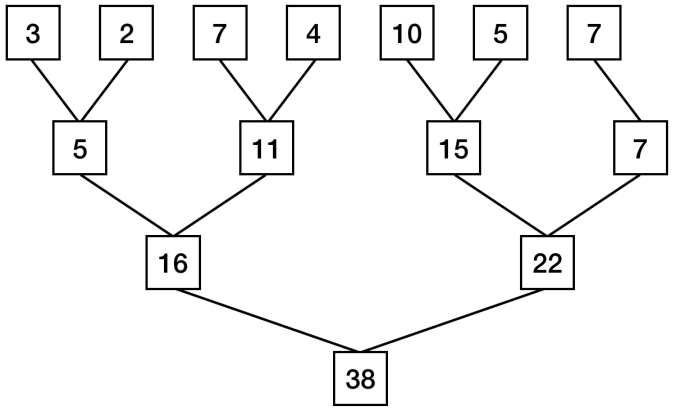

Figure 5.8: Example of reduction: sum of array elements

optimized, through loop unrolling and pipelining pragmas. To allow multiple concurrent memory accesses, the various fields of the structure have been partitioned and stored in BRAM cells with 2 read/write ports. Fast rank operations on the BWT structure are executed through optimized range-selection hardware operations on arbitrary precision precision data types. Moreover, to improve the efficiency of the loop pipelining pragma, we employed a *reduction* strategy to update the *count* value (see Algorithm 2) by efficiently summing the *class* values of consecutive blocks. This technique (an example is showed in Figure 5.8) removes the dependency of accumulating various results over the same variable, allowing the HLS tool to efficiently pipeline the loop.

When both the alignments are complete, the results are sent back to the host machine and a new query is loaded. The core continues loading and aligning sequences until there is no more data to map onto the reference genome. After all the $start$ and end indices have been computed, the positions of the occurrences of each sequence $\mathcal{X}$ and $\overline{\mathcal{X}}$ in the reference are retrieved by the host CPU, in the corresponding sets: $\mathcal{SA}[start(\mathcal{X}), end(\mathcal{X})]$, and $\mathcal{SA}[start(\overline{\mathcal{X}}), end(\overline{\mathcal{X}})]$.

5.4 BWaveR web application

The usability of an open source tool is a critical aspect in determining how much the tool is employed by researchers in their projects. Section 4.3 presented the difficulties that bioinformaticians have to face while installing and using sequence aligners based on command-line user interfaces, which often lead to bad user experience. A review of the state-of-the-art solutions showed that the availability of an intuitive *Graphic User Interface (GUI)* can provide a significant boost to the diffusion of an sequence mapping tool among the bioinformatic community. For this reason, an user-friendly web interface was developed for the proposed sequence mapper, in order to make it easily accessible even to people which are not used to command-line tools.

The HTML-based web interface allows user to interact in an intuitive way with a Python web server, which manages the whole sequence mapping process, from the upload of FASTA and FASTQ files, to the download of the produced results. The web server, which acts as a wrapper of the heterogeneous sequence mapper, has been implemented with Flask [72], a lightweight web application framework with the ability to scale up to complex applications.

An outline of the proposed web application is presented in Figure 5.9. Each block of the flow diagram corresponds to a different page, characterized by its own functions. In particular, Flask's *route()* decorators have been used to bind the different functions implemented in the server to different URLs.

The *Home Page*, showed in Figure 5.10, has a very simple design, with the BWaveR logo in the upper center of the page, and only two buttons underneath: *Start* and *Info*. Clicking the *Info* button, the user is redirected to a page containing

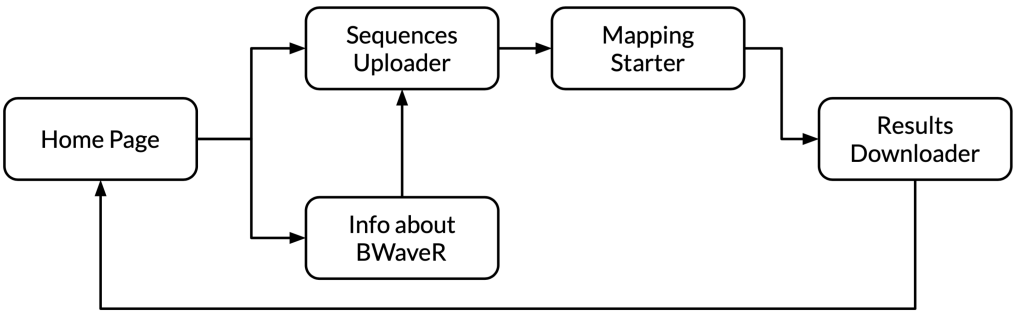

Figure 5.9: Overview of the BWaveR web application

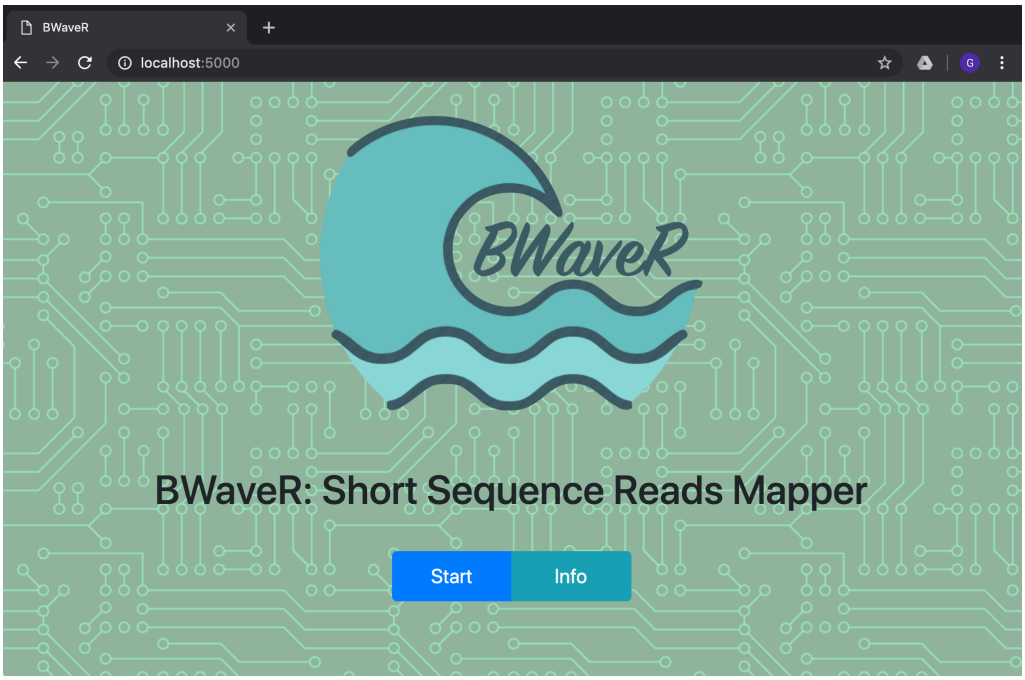

Figure 5.10: BWaveR web app: Home Page

a brief description of the web app, and of the data structures employed in the BWaveR mapper. Since the *Information* page is static, the only function associated to it is the rendering of the page itself, with descriptive text and images. At the bottom of the page, a *Start* button redirects the user to *Sequences Uploader* page.

As the name suggests, this page allows users to upload one reference sequence

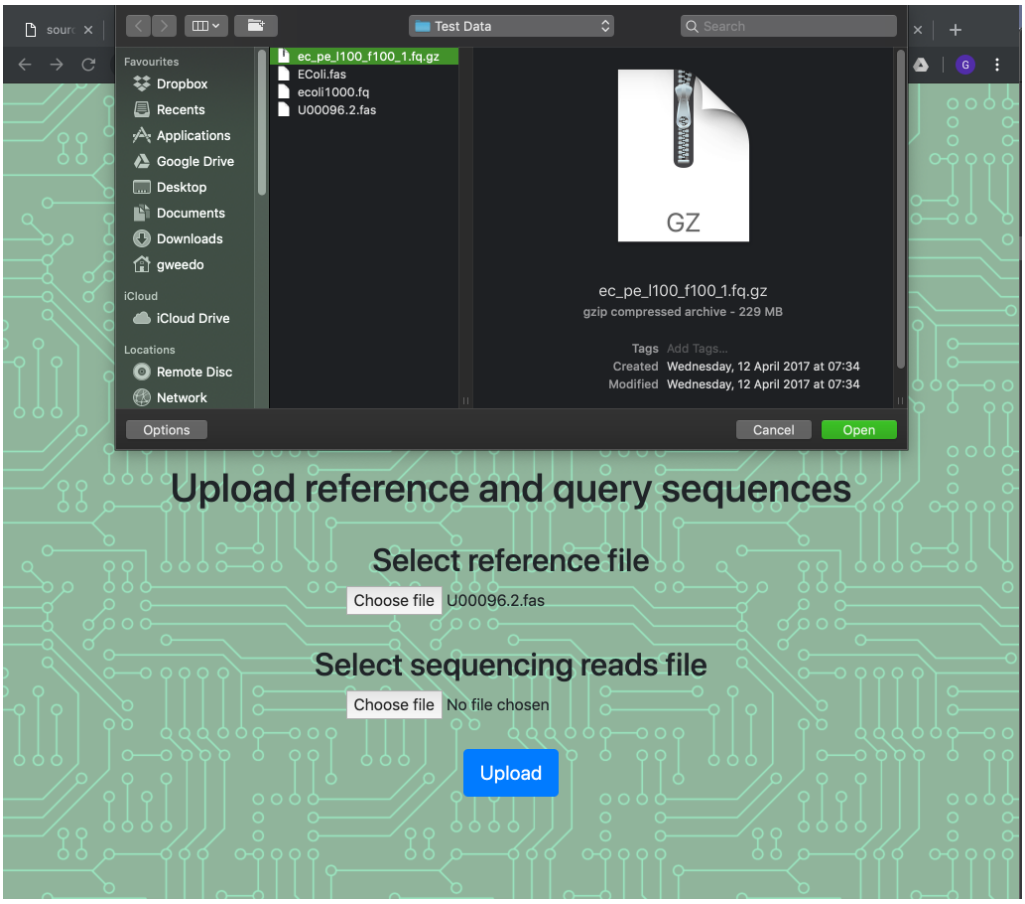

Figure 5.11: BWaveR web app: Sequences Uploader page

and one or more reads files. The sequences can be uploaded in the form of FASTA and FASTQ files, but also in the compressed *gzipped* formats, whose reduced size can significantly decrease the data transfer time. The upload page contains two *Choose file* buttons, which allows users to browse their local harddrive and select the files they want to upload, as showed in Figure 5.11. After choosing the files, the data transfer can be started by clicking on the *Upload* button.

Once the transfer is completed, the users are redirected to the *Mapping Starter* page, where they are notified about the outcome of the data upload. If data are correctly uploaded, a simple click on the *Start Mapping* button can trigger the

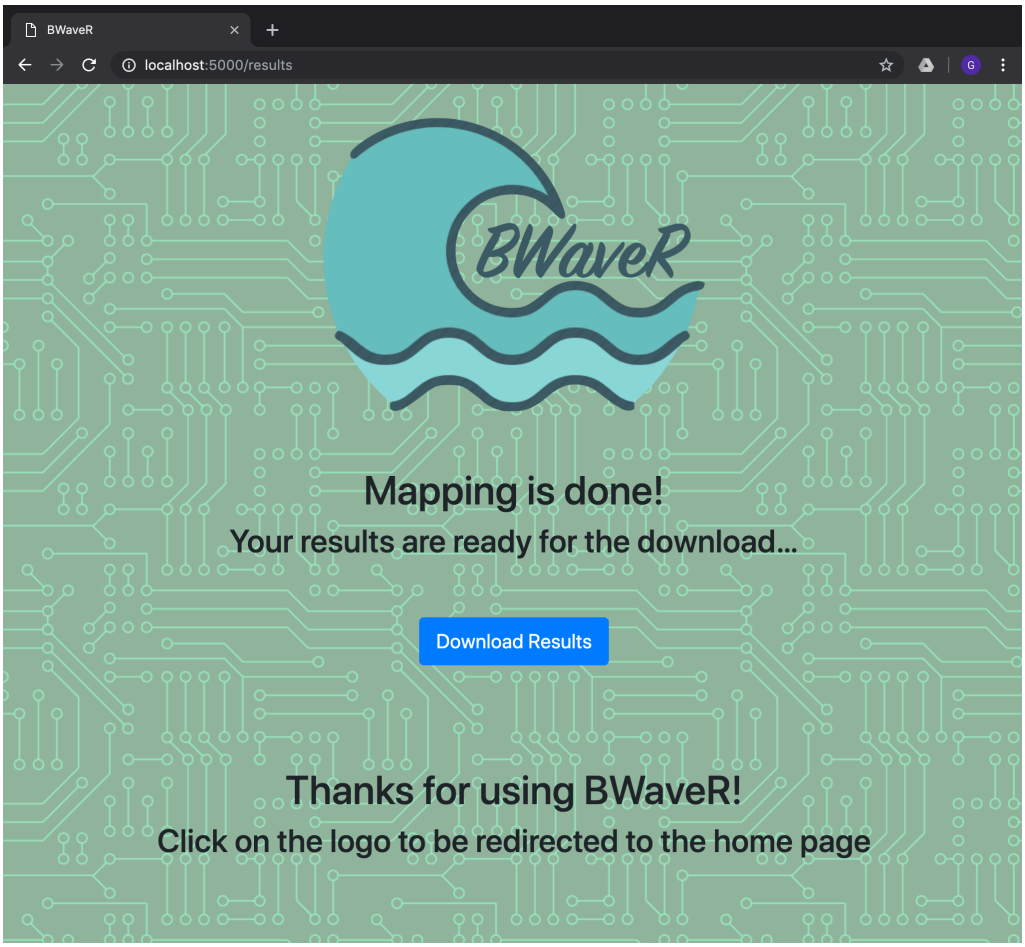

Figure 5.12: BWaveR web app: Results Downloader page

execution of the whole mapping process, from the computation of the BWT and SA of the reference sequence, to the writing of the results file.

Finally, once the output file is ready, the users are redirected to the *Results Downloader* page (Figure 5.12). Here, another simple click, this time on the *Download Results* button, will trigger the download of the gzipped results file. After downloading their results, clicking the BWaveR logo will redirect users to the Home Page, where they can easily start a new sequence mapping experiment. The described application flow allows users to efficiently map genomic sequences

by exploiting the potential of FPGA-acceleration, without any development effort or any knowledge of the underlying hardware architecture.

CHAPTER 6

Experimental Evaluation

This Chapter will present the approches used for the validation and evaluation of the proposed system, together with the obtained results. In detail, Section 6.1 will describe the experimental setup together with the experimental design, while Section 6.2 will provide a description the validation process employed to assess the reliability and accuracy of the results produced by the BWaveR mapper. Sections 6.3 and 6.4 will present a characterization of the implemented succinct data structure, in terms of memory utilization and execution time. Finally, Section 6.5 will describe the performances of the developed custom hardware architecture, in terms of execution time and power consumption.

6.1 Experimental setup and design

The BWaveR web application runs on a server implemented with Flask, written in the Python programming language. As already described in Section 5.4, the Python code acts as a wrapper of the BWaveR binaries leveraging the proposed succinct data structure. The core functions of the sequence mapper, presented in this thesis, have been entirely implemented in C++, a high-level general-purpose computer programming language [74]. This language has been chosen since it provides object-oriented, generic, and functional features in addition to facilities for low-level memory manipulation and efficient mapping to machine instructions, which can be leveraged to achieve better performances compared to higher-level programming languages. Moreover, C++ programs offer high portability, since they can be compiled for different operating system and computer platform with trivial changes to their source code. Last, but not least, the Vivado HLS tool, which has been employed for the design of the custom hardware architecture, supports the C++ language as input, allowing to generate a design for FPGA starting from a C++ source code.

All the validation and evaluation experiments, described in the remainder of this Chapter, have been executed on a remote server equipped with a 2.3GHz 16-core Intel Xeon E5-2698 v3, 128GB of 2133MHz DDR4 RAM, and a Xilinx Alveo U200 (Figure 6.1) powered by an UltraScale+ XCU200 FPGA, chosen as the target for the hardware implementation.

In order to provide a characterization of the proposed succinct data structure, an evaluation framework has been implemented as a Bash Shell Script [78]. This allowed to automatize the execution of many different software tests, with various

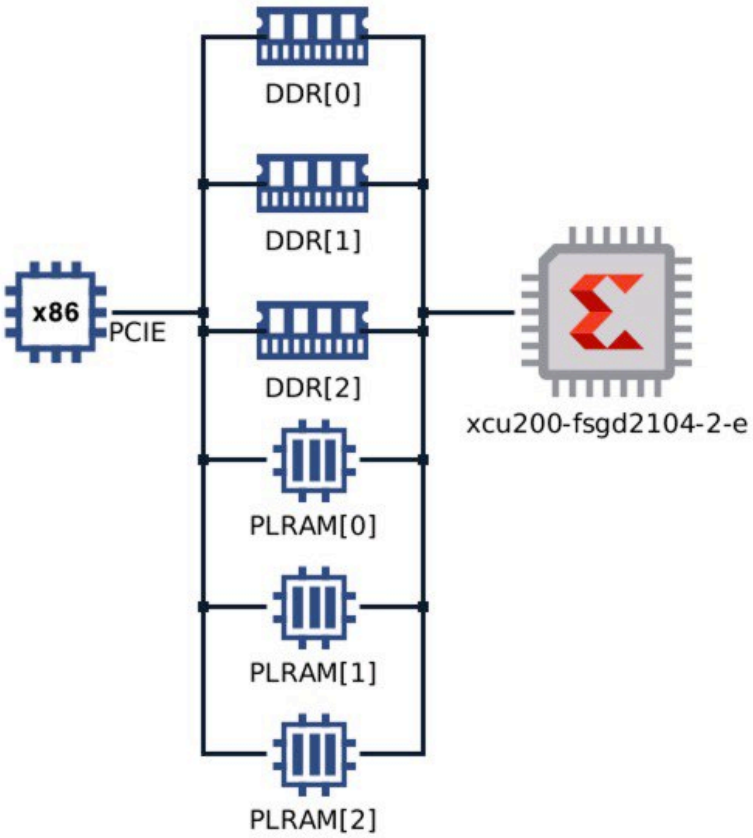

Figure 6.1: Simple schematic of Xilinx Alveo U200 board

combinations of the parameters characterizing the data structure (i.e. *block size* and *superblock factor*). In detail, the experiments were run using all the possible combinations of b and sf, selected among the following sets of values:

$$b = \{3, 7, 11, 15\}$$

$$sf = \{2, 7, 17, 32, 50, 100\}$$

The tests were run both on Escherichia Coli and Homo Sapiens genomic data. In detail, the sequence U00096.3, complete genome of Escherichia Coli (str. K-

12, substr. MG1655), available at [79], was used as a reference in the bacterial tests. For what concern human tests, the Chromosome 21 was extracted from the GRC Human Build 38 patch release 12 (GRCh38.p12), available at [80]. The sequences were pre-processed in order to eliminate gaps, represented as "N" in the FASTA files, and then employed as references in the mapping experiments. The positions and lengths of the gaps, for each reference, were stored in a .csv file which was employed, at the end of the mapping experiments, to efficiently retrieve the positions of the mapped reads on the original gapped reference sequence.

A library of paired-end reads obtained through a Whole Genome Sequencing experiment on Escherichia Coli (ERX008638) has been randomly sampled multiple times with the Seqtk toolkit, available at [81]. This allowed to create a set of five different libraries of about 120000 sequences of length 100 bp. For the human tests, instead, three libraries of single-end reads of 100 bp were downloaded from the International Genome Sample Resource (IGSR) *1000 Genomes phase 3* archive [82], and sampled with Seqtk.

To validate and evaluate the proposed mapper, we compared its outputs and its perfromances to those obtained with Bowtie2 [12], an equivalent state-of-the-art software tool. To provide the fairest comparison, Bowtie2 was run with the options `-a` and `--score-min C,0,-1`, which perform only the BWT-based mapping step to find all and only the exact matches between the query sequences and the reference.

6.2 Validation

For each set of experiments, described in the previous section, we tested the consistency of all the output files produced by mapping reads from the same library to the same reference. In order to do that, we wrote a bash shell script where all the possible pairs of outputs were compared using the *diff* command. The analysis showed that all the files produced by the same combination of inputs (reference and reads) were exactly the same, proving that the accuracy of the mapping process is not affected by the tuning of the b and sf parameters. Moreover, the comparison of the results obtained with the software and hardware implementations showed no differences, thus proving the relyability of our custom hardware architecture.

To assess the correctness of the mapping done by BWaveR, Bowtie2 has been used to map the same read sets without admitting mismatches between the reads and the reference. A Python script has been written to automatically compare the .SAM file produced by Bowtie2 with BWaveR's output. The results of the comparison showed that, at each run, the two tools provide exactly the same mapping. This was expected, since both BWaveR and Bowtie2, when run with the appropriate options, do not make usage of any heuristic to find exact matches between the reads and the reference sequence.

6.3 Memory usage

The first aim of our experiments was to characterize the memory footprint of the proposed data structure, in relation to different values of the b and sf parameters.

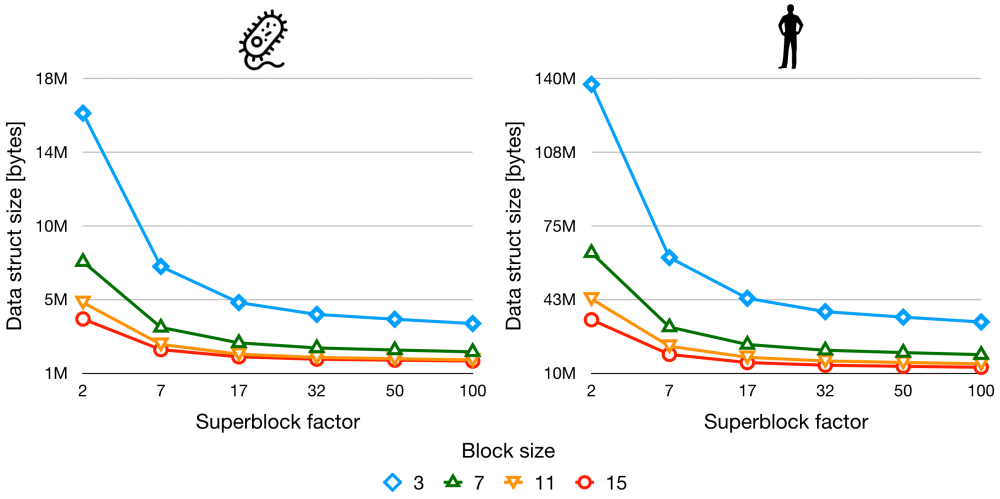

Figure 6.2: Data structure's size for E.Coli (left) and Human Chr.21 (right)

Table 6.1: Data structure's size for E.Coli (Bytes)

$b \setminus sf$	2	7	17	32	50	100
3	16.004.218	7.166.738	5.087.338	4.405.026	4.126.650	3.879.202
7	7.443.460	3.655.972	2.764.796	2.472.380	2.353.076	2.247.028
11	5.089.978	2.679.762	2.112.650	1.926.554	1.850.642	1.783.154
15	4.142.227	2.374.739	1.958.851	1.822.387	1.766.715	1.717.227

Table 6.2: Data structure's size for Human Chr.21 (Bytes)

$b \setminus sf$	2	7	17	32	50	100
3	137.467.426	61.108.146	43.141.258	37.245.866	34.840.554	32.702.498
7	63.193.039	30.467.639	22.767.543	20.240.951	19.210.095	18.293.783
11	42.845.067	22.019.811	17.119.755	15.511.923	14.855.931	14.272.827
15	33.678.497	18.406.641	14.813.265	13.634.177	13.153.121	12.725.513

To this extent, Table 6.1 and Table 6.2 report, respectively, the amount of memory required by the succinct representation of the E.Coli and of the Human Chr.21 genomic sequences. A graphical representation of these results is provided by Figure 6.2, showing that increasing the *block size* and *superblock factor* allows to achieve a better compression. Considering that the E.Coli sequence contains 4641652 bp and the Human Chr.21 (without gaps) contains 40088623 bp, encoding each base as a character (1 byte) would require, respectively, ∼4.64 MB and ∼40.1 MB of memory for storing the *BWT* of the given sequences. The proposed data structure, instead, requires ∼1.71 MB for encoding the E.Coli's *BWT* and ∼12.7 MB for the Human Chr.21 when a $b = 15$ and $sf = 100$, reducing the memory requirements up to 68.3%. Moreover, the *BWT* alone cannot provide the fast rank operations required by the backward search algorithm, instead it is employed to compute the Occ matrix, which can answer fast rank querie. Such a matrix would require ∼92 MB of memory in the case of the E.Coli, and ∼801 MB in rhe case of the Human Chr.21. Thus, the proposed data structure can provide the same information as the Occ matrix, while reducing the memory usage up to 98.4%. Finally, it is worth to notice that even better compression ratio can be obtained for higher values of the b and sf parameters. However, the reduction in the memory footprint would not be significant, thus it would not be worth the computational overhead introduced.

6.4 Execution time

The validation framework described in section 6.2 has also been used to obtain results about the amount of time required for the creation of the BWaveR succinct

Table 6.3: Succinct encoding time for E.Coli (sec)

$b \setminus sf$	2	7	17	32	50	100
3	0.43	0.42	0.42	0.45	0.44	0.43
7	0.76	0.74	0.72	0.73	0.71	0.71
11	0.95	0.95	1.01	0.98	0.96	0.94
15	2.24	2.17	2.12	2.11	2.14	2.13

Table 6.4: Succinct encoding time for Human Chr.21 (sec)

$b \setminus sf$	2	7	17	32	50	100
3	3.44	3.35	3.29	3.38	3.34	3.35
7	4.65	4.59	4.53	4.56	4.57	4.53
11	5.90	5.85	5.87	5.82	5.83	5.81
15	10.88	10.81	10.94	10.72	10.82	10.76

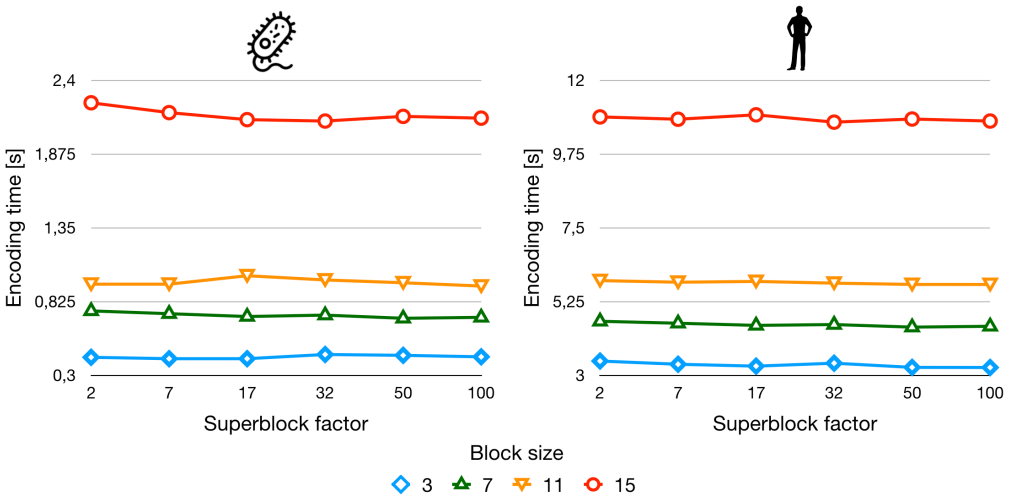

Figure 6.3: Succinct encoding time for E.Coli (left) and Human Chr.21 (right)

data structures, and for the execution of the backward search algorithm.

Table 6.3 and Table 6.4 report the time required for the construction of the succinct data structure, respectively for E.Coli and for the Human Chr.21, with respect to the *block size* and *superblock factor* parameters. A graphical representa-

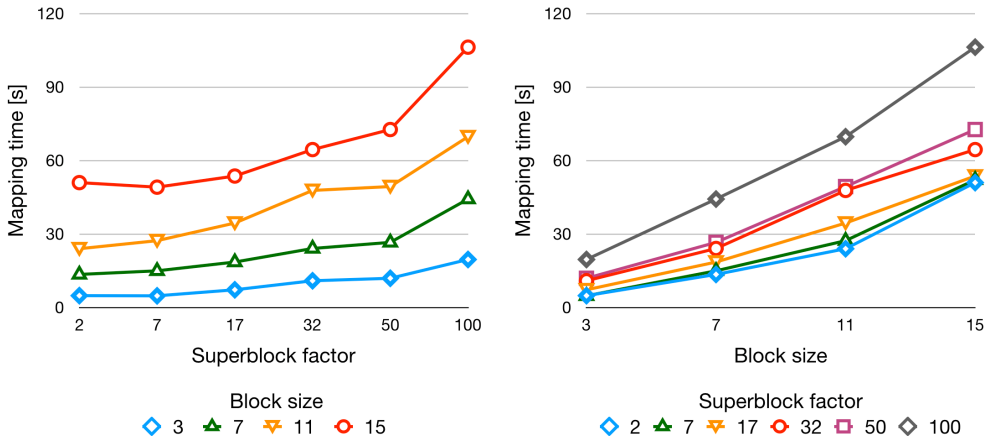

Figure 6.4: Reads mapping time for E.Coli

tion of these results is provided by Figure 6.3, which clearly shows that the encoding time has a strong dependence from the block size, while it is almost constant when the superblock factor is changend.

While memory usage becomes smaller and smaller with the increase of the b and sf parameters, an opposite behavior can be expected by the execution time of the mapping. Figure 6.4 reports the time required for mapping ∼120k 100bp reads, and their reverse complements, to the E.Coli reference genome. Moreover, the presented results confirm the theoretical hypotheses about the direct proportion between the b and sf parameters and the time required for the mapping, showing that the dependence from the *block size* is sligthly stronger than the dependence from the *superblock factor*.

Another interesting result is presented in 6.5, showing that the amount of time required for the mapping is not dependent on the size of the reference genome. Instead, the mapping time depends only on the number of processed reads and on the *mapping ratio*, that is the percentage of reads which find occurrences in

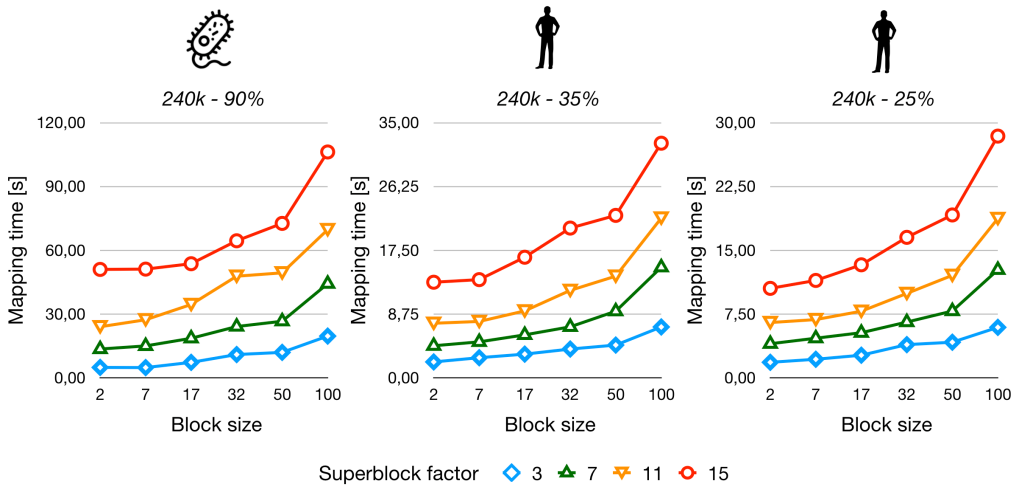

Figure 6.5: Reads mapping time comparison

the reference genome. This is due to the fact that the backward search algorithm process entirely only those reads that actually map on the reference sequence, while it stops earlier if the searched pattern does not appear in the invstigated genome.

Indeed, as already explained, the memory occupation and the execution time are inversely proportional when dealing with the FM-index, and all the existing approach in the literature try to find the best tradeoff between these two aspects. However, it should be noticed that a globally optimal solution to the problem does not exist. For example, a simple naive implementation could be the best option in those cases where a short execution time is needed and the memory resources do not constitute a constraint. On the other hand, approaches involving data compression, despite the computational overhead they introduce, could be well suited for platforms with limited memory resources, such as embedded systems or FPGAs. Given this knowledge, the possibility of controlling the behavior of the implemented data structure, by tuning its parameters, makes this

encoding suitable for a large number of applications. In fact, small values of the parameters allow to obtain a memory intense representation, capable of answering rank queries without much computation. On the other hand, big values of the paramenters enable great compression ratios, resulting in a compute intensive representation requiring many more operations for answering the same rank queries.

6.5 Hardware implementation

The custom hardware architecture has been benchmarked using the OpenCL *events* that provide an easy to use API to profile the code that runs on the FPGA device. We compared the performances of our design, in terms of execution time and energy efficiency, to those of the BWT-based mapping step of Bowtie2. We also compared the performances of our architecture with an optimized software version of the same tool. In our hardware implementation, we fixed the block size $b = 15$ and the superblock factor $sf >= 50$ for each RRR sequence. To obtain the fairest comparison, we made the same modifications to our software implementation, enabling fast access to the static global count table and the class offsets of the RRR sequences via lookup tables. For the tests presented in this Section, we used $sf = 50$ for all the executions of BWaveR, both on CPU and FPGA.

For the power analysis, we estimated the energy efficiency of the tested tools using the *performance over power ratio*, according to the following formula:

$$Energy\ Efficiency\ [\frac{sec^{-1}}{W}]\ = \frac{Execution\ Time^{-1}}{Power\ Consumption}\,. \tag{6.1}$$

In particular, we used the power consumption reference values of $135W$ for the

Table 6.5: Performances of BWaveR and Bowtie2 when mapping 100 millions 100bp reads to E.Coli reference.

	BWaveR		Bowtie2		
	FPGA	CPU	1 thread	8 threads	16 threads
Time [ms]	3623	247214	176683	23016	11542
Speed-up	1×	68,23×	48,76×	6,34×	3,18×
Energy efficiency	1×	368,43×	263,32×	34,3×	17,2×

Intel Xeon and $25W$ for the Xilinx Alveo U200.

Table 6.5 reports the performances of BWaveR and Bowtie2 when mapping 100 millions 100bp reads, and their reverse complements, on the E.Coli reference genome. It shows that our implementation on a single FPGA core is $68.23\times$ faster than the equivalent software solution, and $17.2\times$ more efficient than Bowtie2 when run on 16 threads. As already showed in Section 5.3, and in particular in Figure 5.7, the core employed for the E.Coli tests uses only $\sim 3\%$ of the BRAM available on the XCU200 FPGA, which is the critical resource. Consequently, it is possible to replicate the proposed mapping kernel in a multi-core design to further boost the performances of the hybrid sequence aligner.

Table 6.6 shows the results of the alignments on a varying number of reads of length 100bp on the Human Chr.21 reference. As we can see from the results, our architecture performs better when we increase the number of reads to be aligned. The reason of this behavior is the fact that the load of the BWT structure introduces a fixed overhead in the execution time of the developed kernel, regardless of the number of reads we need to align. When the number of sequences to align increases, the speed-up increases too, making our architecture very efficient for a massive number of alignments, as in the case of real-world

Table 6.6: Performances of BWaveR and Bowtie2 when aligning 1, 10 and 100 millions 100bp reads to Chr.21 reference.

		BWaveR		Bowtie2		
		FPGA	CPU	1 thread	8 threads	16 threads
1 M	Time [ms]	242	3302	1891	344	180
	Speed-up	1×	13,62×	7,78×	1,41×	0,74×
	Energy efficiency	1×	73,54×	42,11×	7,65×	4,01×
10 M	Time [ms]	460	28658	19126	3483	1823
	Speed-up	1×	62,4×	41,63×	7,57×	3,96×
	Energy efficiency	1×	336,94×	224,87×	40,95×	21,43×
100 M	Time [ms]	3783	266253	192075	35969	18575
	Speed-up	1×	70,39×	50,77×	9,51×	4,91×
	Energy efficiency	1×	380,09×	274,2×	51,34×	26,52×

applications. When aligning 100M reads, our single-core architecture achieves a $4.91\times$ speed-up against Bowtie2 running on 16 threads, and a $70.39\times$ speed-up when compared to our optimized software implementation. The low power consumption of the FPGA also allowed us to be $380.09\times$ more efficient than our software implementation, and up to $274.2\times$ more efficient than Bowtie2 on single thread. These results show that the proposed hardware architecture is able to accelerate bioinformatic applications involving genomic sequence mapping, while significantly reducing the energy consumption, according to the needs of pharmaceutical industries and those of the researchers' community.

CHAPTER 7

Conclusions and Future Work

This thesis can be considered as a part of the greater work of the scientific community, aiming to the development and diffusion of the personalized medicine, also known as precision medicine. This approach already proved to be effective in the war to some of the most aggressive deseases, such as melanoma and other kind of cancers, thus it is gaining the attention of researchers from all around the world.

The development of Next Generation Sequencing technologies produced an explosion in the amount of genomic data generated, and the bioinformatic tools available for the processing of these data can not keep up with its pace. In particular, sequence alignment proved to be the computational bottleneck of many bioinformatic applications, leading to protracted times and high costs for the required processes. Moreover, the alignment tools openly available usually have low usability, which is a threat for the advancement of the research in the field.

For these reasons, this thesis presented the design and the implementation of an easy-to-use Burrows-Wheeler sequence mapper, accessible via a web applica-

tion which provides a great user experience. The implemented sequence mapper, called BWaveR, leverages a combination of powerful succinct data structures to achieve high flexibility of usage, especially in those cases where memory resources are a problem. BWaveR can be used for many different bioinformatic applications, both as a standalone tool for the exact matching of biological sequences, or as a part of more complex sequence aligners based on the seed and extend paradigm.

This work also presented the development of a custom hardware architecture for sequence mapping, leveraging the compression capability of the proposed data structure to fully exploit the highly parallel architecture of FPGAs and their low power consumption. The simple hardware design presented in this work has been integrated in BWaveR, resulting in a unique hybrid sequence mapper, which offers to users the benefits of the acceleration on FPGA, without requiring any programming effort or any knowledge of the underlying hardware architecture.

A comprehensive explanation of wavelet trees and RRR sequences can be found in this work, together with the description of the optimizations that led to the practical implementation employed in the BWaveR mapper. A validation of the presented system has been done, which proved the reliability of the results it produces. Moreover, a characterization of the proposed data structure was provided, in terms of memory occupation, time required for its creation, and time required for mapping sequence reads. The experiments revealed a great compression capability, which allowed to retain all the information necessary for answering rank queries in short time, while reducing the memory occupation up to 98.3%. The experimental evaluation also proved that FPGAs can be leveraged to improve the execution efficiency of the bit-level operations required to access

information in the proposed data structure. Our single-core implementation, on a single Xilinx Alveo U200, achieves a speedup of up to 70× over the optimized software implementation. It also results to be 4,9× faster and 26,5× more energy efficient than Bowtie2, an equivalent state-of-the-art software application (run on 16 threads), without any loss in accuracy.

7.1 Future Work

Although some great results have been achieved in this work, the potentiality of the proposed hybrid sequence mapper can be further exploited. For this reason, future work involves to extend our mapping design to approximate string matching, and to allow reference sequences longer than 200 millions bp. We also plan to leverage the FPGA's parallelism to develop a multi-core architecture where multiple DNA fragments are mapped at the same time. Finally, we would like to integrate BWaveR in real sequence analysis pipelines, and automating their implementation from a high-level description. In particular, we plan to couple BWaveR with an accelerated DP-based alignment algorithms, to create a fast and power efficient gapped sequence aligner. Such a tool would provide a great service to the scientific community, enabling users with no knowledge of hardware acceleration to perform a multitude of bioinformatic experiments in a way that is easy, fast and cheap. We firmly believe that this would be a great step towards the development, diffusion and democratization of personalized medicine.

Bibliography

[1] N. R. Council, *Toward Precision Medicine: Building a Knowledge Network for Biomedical Research and a New Taxonomy of Disease*. Washington, DC: The National Academies Press, 2011.

[2] B. de Unamuno Bustos, R. Murria Estal, G. Pérez Simó, I. de Juan Jimenez, B. Escutia Muñoz, M. Rodríguez Serna, V. Alegre de Miquel, M. Llavador Ros, R. Ballester Sánchez, E. Nagore Enguídanos, S. Palanca Suela, and R. Botella Estrada, "Towards personalized medicine in melanoma: Implementation of a clinical next-generation sequencing panel," *Scientific Reports*, vol. 7, no. 1, p. 495, 2017.

[3] HealthMatters, "What is precision medicine?." `https://healthmatters.nyp.org/precision-medicine/`. [Online; accessed 01/04/2020].

[4] A. K. Kesarwani, A. Malhotra, A. Srivastava, G. Ananda, H. Ashoor, P. Kumar, R. K. Kesharwani, V. K. Sarsani, Y. Li, J. George, and R. K. M. Karuturi, "Genome informatics," in *Encyclopedia of Bioinformatics and Computational Biology* (S. Ranganathan, M. Gribskov, K. Nakai, and C. Schönbach, eds.), pp. 178 – 194, Oxford: Academic Press, 2019.

[5] J. F. Abril and S. Castellano, "Genome annotation," in *Encyclopedia of Bioinformatics and Computational Biology* (S. Ranganathan, M. Gribskov, K. Nakai, and C. Schönbach, eds.), pp. 195 – 209, Oxford: Academic Press, 2019.

[6] S. Beretta, "Algorithms for strings and sequences: Pairwise alignment," in *Encyclopedia of Bioinformatics and Computational Biology* (S. Ranganathan, M. Gribskov, K. Nakai, and C. Schönbach, eds.), pp. 22 – 29, Oxford: Academic Press, 2019.

[7] 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*, vol. 48, no. 3, pp. 443 – 453, 1970.

[8] T. F. Smith and M. S. Waterman, "Identification of common molecular subsequences.," *Journal of molecular biology*, vol. 147 1, pp. 195–7, 1981.

[9] A. Poliakov, C. B. Do, I. Dubchak, M. Brudno, O. Couronne, S. Malde, and S. Batzoglou, "Glocal alignment: finding rearrangements during alignment," *Bioinformatics*, vol. 19, pp. 54–62, 07 2003.

[10] N. Ahmed, K. Bertels, and Z. Al-Ars, "A comparison of seed-and-extend techniques in modern dna read alignment algorithms," in *2016 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*, pp. 1421–1428, Dec 2016.

[11] H. Li, "Aligning sequence reads, clone sequences and assembly contigs with bwa-mem," *arXiv preprint arXiv:1303.3997*, 2013.

[12] B. Langmead and S. L. Salzberg, "Fast gapped-read alignment with bowtie 2," *Nature methods*, vol. 9, no. 4, p. 357, 2012.

[13] Y. Liu, A. Wirawan, and B. Schmidt, "Cudasw++ 3.0: accelerating smith-waterman protein database search by coupling cpu and gpu simd instructions," in *BMC Bioinformatics*, 2012.

[14] L. Di Tucci, D. Conficconi, A. Comodi, S. Hofmeyr, D. Donofrio, and M. D. Santambrogio, "A parallel, energy efficient hardware architecture for the meraligner on fpga using chisel hcl," in *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 214–217, May 2018.

[15] Illumina, "Dragen bio-it platform." `https://www.illumina.com/products/by-type/informatics-products/dragen-bio-it-platform.html?scid=2018-269ECL299`. [Online; accessed 01/04/2020].

[16] A. J. Bromage and T. C. Conway, "Succinct data structures for assembling large genomes," *Bioinformatics*, vol. 27, pp. 479–486, 01 2011.

[17] B. Langmead, "Bowtie2 web user interface (beta)." `http://18.218.70.192/beta/bt2-ui/`. [Online; accessed 01/04/2020].

[18] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman, "Basic local alignment search tool," *Journal of Molecular Biology*, vol. 215, no. 3, pp. 403 – 410, 1990.

[19] NIH, "Dna sequencing fact sheet - national human genome research institute." https://www.genome.gov/10001177/dna-sequencing-fact-sheet/. [Online; accessed 01/04/2020].

[20] DOE, "Human genome project information archive." https://web.ornl.gov/sci/techresources/Human_Genome/index.shtml. [Online; accessed 01/04/2020].

[21] F. Sanger, S. Nicklen, and A. R. Coulson, "Dna sequencing with chain-terminating inhibitors," *Proceedings of the National Academy of Sciences*, vol. 74, no. 12, pp. 5463–5467, 1977.

[22] Forbes, "Illumina promises to sequence human genome for $100 - but not quite yet." https://www.forbes.com/sites/matthewherper/2017/01/09/illumina-promises-to-sequence-human-genome-for-100-but-not-quite-yet/#6a843be0386d. [Online; accessed 01/04/2020].

[23] J. Shendure, S. Balasubramanian, G. M. Church, W. Gilbert, J. Rogers, J. A. Schloss, and R. H. Waterston, "Dna sequencing at 40: past, present and future," *Nature*, vol. 550, pp. 345 EP –, Oct 2017. Review Article.

[24] "Illumina." https://www.illumina.com/. [Online; accessed 01/04/2020].

[25] J. Besser, H. A. Carleton, P. Gerner-Smidt, R. L. Lindsey, and E. Trees, "Next-generation sequencing technologies and their application to the

study and control of bacterial infections," *Clinical Microbiology and Infection*, vol. 24, no. 4, pp. 335–341, 2018.

[26] "Pacific biosciences." `https://www.pacb.com/`. [Online; accessed 01/04/2020].

[27] "Oxford nanopore technologies - dna nanopore sequencing." `https://nanoporetech.com/applications/dna-nanopore-sequencing`. [Online; accessed 01/04/2020].

[28] D. Maier, "The complexity of some problems on subsequences and supersequences," *J. ACM*, vol. 25, pp. 322–336, Apr. 1978.

[29] J. D. Kececioglu and E. W. Myers, "Combinatorial algorithms for dna sequence assembly," *Algorithmica*, vol. 13, no. 1-2, p. 7, 1995.

[30] J. T. Simpson and M. Pop, "The theory and practice of genome sequence assembly," *Annual review of genomics and human genetics*, vol. 16, pp. 153–172, 2015.

[31] PHRAP, "Documentation for phrap and crossmatch (version 0.990319)." `http://www.phrap.org/phredphrap/phrap.html`. [Online; accessed 01/04/2020].

[32] B. Wajid and E. Serpedin, "Review of general algorithmic features for genome assemblers for next generation sequencers," *Genomics, proteomics & bioinformatics*, vol. 10, no. 2, pp. 58–73, 2012.

[33] P. A. Pevzner, "1-tuple dna sequencing: computer analysis," *Journal of Biomolecular structure and dynamics*, vol. 7, no. 1, pp. 63–73, 1989.

[34] P. A. Pevzner, H. Tang, and M. S. Waterman, "An eulerian path approach to dna fragment assembly," *Proceedings of the national academy of sciences*, vol. 98, no. 17, pp. 9748–9753, 2001.

[35] D. R. Zerbino and E. Birney, "Velvet: algorithms for de novo short read assembly using de bruijn graphs," *Genome research*, vol. 18, no. 5, pp. 821–829, 2008.

[36] E. W. Myers, "The fragment assembly string graph," *Bioinformatics*, vol. 21, no. suppl_2, pp. ii79–ii85, 2005.

[37] J. T. Simpson and R. Durbin, "Efficient de novo assembly of large genomes using compressed data structures," *Genome research*, vol. 22, no. 3, pp. 549–556, 2012.

[38] M. Pop, A. Phillippy, A. L. Delcher, and S. L. Salzberg, "Comparative genome assembly," *Briefings in bioinformatics*, vol. 5, no. 3, pp. 237–248, 2004.

[39] H. Li and N. Homer, "A survey of sequence alignment algorithms for next-generation sequencing," *Briefings in bioinformatics*, vol. 11, no. 5, pp. 473–483, 2010.

[40] E. V. Koonin and M. Galperin, *Sequence—evolution—function: computational approaches in comparative genomics*. Springer Science & Business Media, 2013.

[41] Wikipedia, "Dna annotation."

https://en.wikipedia.org/wiki/DNA_annotation. [Online; accessed 01/04/2020].

[42] EBI, "What is variant calling?." https://www.ebi.ac.uk/training/online/course/human-genetic-variation-i-introduction/variant-identification-and-analysis/what-variant. [Online; accessed 01/04/2020].

[43] T. Kishikawa, Y. Momozawa, T. Ozeki, T. Mushiroda, H. Inohara, Y. Kamatani, M. Kubo, and Y. Okada, "Empirical evaluation of variant calling accuracy using ultra-deep whole-genome sequencing data," *Scientific reports*, vol. 9, no. 1, p. 1784, 2019.

[44] ZhangLab, "What is fasta format?." https://zhanglab.ccmb.med.umich.edu/FASTA/. [Online; accessed 01/04/2020].

[45] P. J. Cock, C. J. Fields, N. Goto, M. L. Heuer, and P. M. Rice, "The sanger fastq file format for sequences with quality scores, and the solexa/illumina fastq variants," *Nucleic acids research*, vol. 38, no. 6, pp. 1767–1771, 2009.

[46] M. Burrows and D. J. Wheeler, "A block-sorting lossless data compression algorithm," 1994.

[47] J. R. Healy, E. E. Thomas, J. T. Schwartz, and M. Wigler, "Annotating large genomes with exact word matches.," *Genome research*, vol. 13 10, pp. 2306–15, 2003.

[48] H. Li and R. Durbin, "Fast and accurate short read alignment with burrows–wheeler transform," in *Bioinformatics*, 2009.

[49] 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*, vol. 10, p. R25, Mar 2009.

[50] R. Raman, V. Raman, and S. S. Rao, "Succinct indexable dictionaries with applications to encoding k-ary trees and multisets," in *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '02, (Philadelphia, PA, USA), pp. 233–242, Society for Industrial and Applied Mathematics, 2002.

[51] Xilinx, "Sdaccel development environment." `https://www.xilinx.com/products/design-tools/software-zone/sdaccel.html`. [Online; accessed 01/04/2020].

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

[53] D. Lipman and W. Pearson, "Rapid and sensitive protein similarity searches," *Science*, vol. 227, no. 4693, pp. 1435–1441, 1985.

[54] J. Pevsner, *Bioinformatics and Functional Genomics*. Wiley Publishing, 2nd ed., 2009.

[55] Novocraft, "Novoalign reference manual." `http://www.novocraft.com/`

documentation/novoalign-2/novoalign-reference-manual/. [Online; accessed 01/04/2020].

[56] Y. Liu and B. Schmidt, "Long read alignment based on maximal exact match seeds," *Bioinformatics*, vol. 28, pp. i318–i324, 09 2012.

[57] R. Li, C. Yu, Y. Li, T.-W. Lam, S.-M. Yiu, K. Kristiansen, and J. Wang, "Soap2: an improved ultrafast tool for short read alignment," *Bioinformatics*, vol. 25, no. 15, pp. 1966–1967, 2009.

[58] H. Li and R. Durbin, "Fast and accurate long-read alignment with Burrows–Wheeler transform," *Bioinformatics*, vol. 26, pp. 589–595, 01 2010.

[59] S. Kumar, S. Agarwal, and R. Prasad, "Efficient read alignment using burrows wheeler transform and wavelet tree," in *2015 Second International Conference on Advances in Computing and Communication Engineering*, pp. 133–138, May 2015.

[60] H. M. Waidyasooriya, D. Ono, M. Hariyama, and M. Kameyama, "An fpga architecture for text search using a wavelet-tree-based succinct-data-structure," in *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, p. 354, The Steering Committee of The World Congress in Computer Science, Computer …, 2015.

[61] S. Gog, T. Beller, A. Moffat, and M. Petri, "From theory to practice: Plug and play with succinct data structures," in *13th International Symposium on Experimental Algorithms, (SEA 2014)*, pp. 326–337, 2014.

[62] B. Wang, C. Yu, C.-M. Liu, E. Wu, K. Zhao, R. Luo, R. Li, S.-M. Yiu, T.-W. Lam, T. Wong, X. Chu, and Y. Li, "SOAP3: ultra-fast GPU-based parallel alignment tool for short reads," *Bioinformatics*, vol. 28, pp. 878–879, 01 2012.

[63] R. Luo, T. Wong, J. Zhu, C.-M. Liu, X. Zhu, E. Wu, L.-K. Lee, H. Lin, W. Zhu, D. W. Cheung, *et al.*, "Soap3-dp: fast, accurate and sensitive gpu-based short read aligner," *PloS one*, vol. 8, no. 5, p. e65632, 2013.

[64] C. B. Olson, M. Kim, C. Clauson, B. Kogon, C. Ebeling, S. Hauck, and W. L. Ruzzo, "Hardware acceleration of short read mapping," in *2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines*, pp. 161–168, IEEE, 2012.

[65] E. Fernandez, W. Najjar, and S. Lonardi, "String matching in hardware using the fm-index," in *2011 IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines*, pp. 218–225, IEEE, 2011.

[66] E. B. Fernandez, W. A. Najjar, S. Lonardi, and J. Villarreal, "Multithreaded fpga acceleration of dna sequence mapping," in *2012 IEEE Conference on High Performance Extreme Computing*, pp. 1–6, IEEE, 2012.

[67] J. Arram, T. Kaplan, W. Luk, and P. Jiang, "Leveraging fpgas for accelerating short read alignment," *IEEE/ACM Transactions on Computational Biology and Bioinformatics (TCBB)*, vol. 14, no. 3, pp. 668–677, 2017.

[68] J. Arram, K. H. Tsoi, W. Luk, and P. Jiang, "Reconfigurable acceleration of short read mapping," in *2013 IEEE 21st Annual International Symposium*

on Field-Programmable Custom Computing Machines, pp. 210–217, IEEE, 2013.

[69] Anaconda, "Conda - package, dependency and environment management for any language." `https://conda.io/projects/conda/en/latest/index.html`. [Online; accessed 01/04/2020].

[70] R. Dale, B. Grüning, A. Sjödin, J. Rowe, B. A. Chapman, C. H. Tomkins-Tinch, R. Valieris, J. Köster, *et al.*, "Bioconda: a sustainable and comprehensive software distribution for the life sciences," *bioRxiv*, p. 207092, 2017.

[71] NCBI, "Blast - basic local alignment search tool." `https://blast.ncbi.nlm.nih.gov/Blast.cgi`. [Online; accessed 01/04/2020].

[72] A. Ronacher, "Flask: web development, one drop at a time." `http://flask.pocoo.org/`. [Online; accessed 01/04/2020].

[73] Samtools, "Sequence alignment/map format specification." `https://samtools.github.io/hts-specs/SAMv1.pdf`. [Online; accessed 01/04/2020].

[74] B. Stroustrup, *The C++ programming language*. Pearson Education, 2013.

[75] C. A. R. Hoare, "Quicksort," *The Computer Journal*, vol. 5, pp. 10–16, 01 1962.

[76] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," *ACM Sigplan notices*, vol. 42, no. 6, pp. 89–100, 2007.

[77] J. E. Stone, D. Gohara, and G. Shi, "Opencl: A parallel programming standard for heterogeneous computing systems," *Computing in Science Engineering*, vol. 12, no. 3, pp. 66–73, 2010.

[78] Wikibooks, "Bash shell scripting." `https://en.wikibooks.org/wiki/Bash_Shell_Scripting`. [Online; accessed 01/04/2020].

[79] NCBI, "U00096.3 escherichia coli str. k-12 substr. mg1655, complete genome." `https://www.ncbi.nlm.nih.gov/nuccore/U00096.3?report=fasta`. [Online; accessed 01/04/2020].

[80] NCBI, "Human genome resources at ncbi." `ftp://ftp.ncbi.nlm.nih.gov/refseq/H_sapiens/annotation/GRCh38_latest/refseq_identifiers/GRCh38_latest_genomic.fna.gz`. [Online; accessed 01/04/2020].

[81] H. Li, "Seqtk: Toolkit for processing sequences in fasta/q formats." `https://github.com/lh3/seqtk`. [Online; accessed 01/04/2020].

[82] IGSR, "Human genome resources at ncbi." `ftp://ftp.1000genomes.ebi.ac.uk/vol1/ftp/phase3/data`. [Online; accessed 01/04/2020].