BWaveR: Leveraging Succinct Data Structures for DNA Sequence Mapping on FPGA

Guido Walter Di Donato, Alberto Zeni, Lorenzo Di Tucci, Marco D. Santambrogio
Dipartimento di Elettronica, Informatica e Bioingegneria
Politecnico di Milano, Milan, Italy
{guidowalter.didonato, alberto.zeni}@mail.polimi.it
{lorenzo.ditucci, marco.santambrogio}@polimi.it

***Abstract*—The sequencing capability of NGS platforms continues to increase, but mapping short DNA fragments to large genomic sequences still constitutes a challenge in a vast set of bioinformatic applications, including genome resequencing and comparative genome assembly. In this context, succinct data structures are receiving growing interest, allowing to encode large reference sequences in a minimal memory footprint, while still granting fast queries on them. The mapping process requires to access information in such structures through bit-level operations, that can be efficiently performed using custom hardware architectures. In this paper, we present BWaveR, a heterogeneous system for short sequence mapping, based on succinct data structures and leveraging FPGAs to increase the execution efficiency of the sequence matching process. Our design is implemented and evaluated on a single Xilinx Alveo U200, obtaining up to a 70× speedup and a 380× improvement in power efficiency when compared to an optimized pure software implementation. Moreover, it results up to 4.9× faster and 26× more power-efficient than Bowtie2, a state-of-the-art equivalent software run with 16 threads, without any loss in accuracy. The proposed hybrid sequence mapper is made available through an intuitive web application, providing an easily accessible solution for reducing the time and the cost of many bioinformatic applications.**

***Index Terms*—Bioinformatics, Genome Mapping, FM-index, Hardware Acceleration, FPGA, Succinct Data Structure**

I. Introduction

Next-Generation Sequencing (NGS) produced an explosion in the amount of genomic data generated. This rapidly growing availability of sequencing data has been extremely valuable to the development of genomic research and of the emerging field of personalized medicine [1]. However, the computing systems used for subsequent analysis, cannot keep up with the pace of NGS technologies. Consequently, many sequence analysis pipelines still require hours, or even days, to make sense of the raw genomic data and transform them in useful information for diagnosis or research.

In particular, the mapping of a massive amount of short DNA fragments to a large genomic sequence still constitute a challenge in a variety of bioinformatic applications. In genome resequencing, for instance, hundreds of millions of short reads are mapped onto a reference genome where the complete sequence of the concerning species is already known, in order to determine the genetic variations of a sample in relation to the reference. Moreover, the application of sequence mapping is not limited to short reads technology. In fact, most of the existing aligners, for both short and long reads data, rely on a seed-and-extend strategy where the mapping of short DNA fragments is used to determine candidate loci in the genome (seeds) to be extended by the actual alignment algorithm. At the cost of minimal loss in sensitivity, such heuristic allows to drastically reduce the amount of time required by the alignment tools. In certain cases, using such approach can even change the performance bottleneck from the alignment stage to the seed generation stage, as in the work presented in [2].

In the last years, a variety of short sequence mapping tools has been proposed, based on different approaches. Among them, tools based on the Burrows-Wheeler Transform (BWT) [3] of the reference genome, are particularly efficient in both memory footprint and speed. These tools build the FM-index [4] of the BWT sequence, which allows searching for all the occurrences of a pattern of length p in the reference in $O(p)$ time, independently from the reference length. Although the FM-index can make use of several compression techniques, the aforementioned software tools implemented in General Purpose CPUs (GP-CPUs) are unable to take advantage of a compressed index, relying instead on the re-sampling of the index data in order to reduce the space cost.

In this context, succinct data structures [5] can provide a flexible solution, encoding data in a space close to the theoretical lower bound, while still allowing for fast queries on them. However, accessing information in such structures typically requires the use of bit-level operations, which could result in long processing times on common GP-CPUs. Instead, reconfigurable hardware, such as Field-Programmable Gate Array (FPGA), can provide specialized structures that increase the execution efficiency of such operations. Moreover, FPGAs have already proved to be a promising solution for accelerating short sequence mapping, providing substantial application speed-up while allowing reduced energy consumption [6] [7].

In this paper we present how we leveraged a combination of succinct data structures, namely *wavelet tree* and *RRR sequence*, for accelerating DNA sequence mapping on FPGA. The major contributions of this work include:

- An implementation of a parametrizable succinct data structure for encoding the BWT of large texts in a minimal memory footprint, while still allowing to answer fast rank queries on it. This enables to efficiently store genomic sequences as long as human chromosomes in the limited on-chip memory of the FPGA.

- An application of the proposed data structure to develop a DNA sequence mapping design able to find all occurrences of short DNA fragments, and their reverse complements, in a given reference genome.
- An implementation of our mapping design on a Xilinx Alveo U200 accelerator card, and its evaluation in relation to the optimized pure software implementation and to the state-of-art equivalent software application.

II. Related Work

In the last years, many BWT-based short read mapping software tools have been proposed, such as BWA [8] and Bowtie2 [9]. These tools make use of algorithms based on the FM-index of the BWT sequence to map short DNA fragments onto the reference genome. Their main advantage over the competitor tools, based on hash tables, is that their memory usage is independent of the number of fragments to be aligned, instead of growing linearly with it. Moreover, BWT-based mappers can locate a pattern P in a reference R via a backward search procedure, in a time that is linear in the length of P instead of the length of R. Modified versions of the backward search algorithm can also support substitutions, insertions, and deletions. However the time complexity of these strategies grows exponentially with number of mismatches, thus tools implementing them only allow few differences between substrings of the reference genome and the query sequences.

To the best of our knowledge, only two works in the literature leveraged wavelet trees for the Burrows-Wheeler mapping of short DNA fragments. Kumar et al. [10] 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 Structure Library (SDSL), a powerful and flexible C++ library implementing wavelet trees by 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 Section III-B. 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.

Waidyasooriya et al. [11] proposed an FPGA-based hardware architecture for text searching that leverages a wavelet tree based succinct data structures. The main contribution of their work is the data structure they propose, which considers the hardware specification. It consists of codewords made of a header, containing the partial rank of the corresponding bit vector block, and a body whose size depends on the memory model and on the number of symbols in the text. Thus, rank queries on the text are made through fast binary rank queries on the bit vectors, one for each level of the wavelet tree. Their design consists of two DDR3 RAM memories, where the rank data of the text is stored, and an array of Processing Elements (PEs), which process in parallel the search queries, sent in batches to the FPGA. The proposed design 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.

Many other works have been published that use FPGAs for addressing the problem of mapping short DNA fragments to a reference genome. The claimed speedups of the FPGAs designs proposed in the literature, compared to their equivalent software solutions, has a wide range. Moreover, a comparison between them is not easy because they are tested on different data sets and architectures. Here we summarise some notable efforts strictly related to this work. A more comprehensive survey on reconfigurable acceleration of genetic sequence alignment can be found in [12].

Fernandez et al. [13] developed the first implementation of FM-index on FPGA, and extended their work in [6] to support up to 2 mismatches and 512 concurrently executing threads on a single FPGA of the Convey HC-2ex. Using four Virtex-6 LX760 FPGAs and 96GB of shared memory, their design can provide a speedup up to $12\times$ compared to 8-thread Bowtie. Arram et al. [14], 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. 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. However, their performance results are only an estimate and cannot be measured from a full implementation. In [7], the same authors presented another runtime reconfigurable architecture for FPGAs, entirely based on FM-index. 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 16-thread Bowtie2 aligner, at the cost of a 7% reduction in accuracy. It is important to note that, although these works achieves significant results in the target application, they all leverage a system with multiple FPGAs and high quantity of RAM. Our architecture instead proposes a significant speed-up leveraging a single FPGA and using a limited quantity of RAM, therefore it can be easily replicated to obtain even better performances.

III. Methodology and Implementation

In this section, we present the Burrows-Wheeler mapping problem and our implementation of a succinct data structure for representing efficiently long BWT sequences. We also propose an FPGA design leveraging such data structure, together with its usage in BWaveR, a hybrid DNA sequence mapper.

A. Burrows-Wheeler Mapping

The Burrows-Wheeler Mapping process consists of locating a pattern of length p in a long reference sequence, in $O(p)$

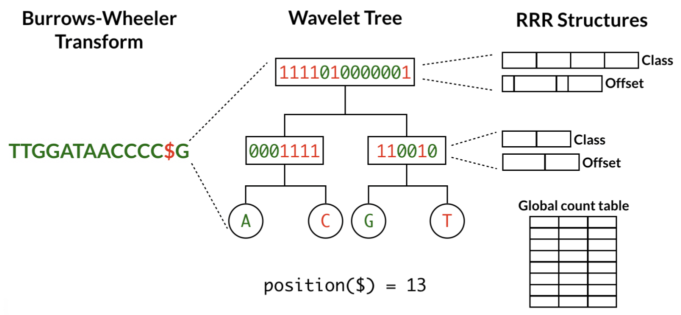

Fig. 1: Overview of the BWaveR data structure: the BWT sequence is represented as a wavelet tree whose nodes are encoded as RRR structures.

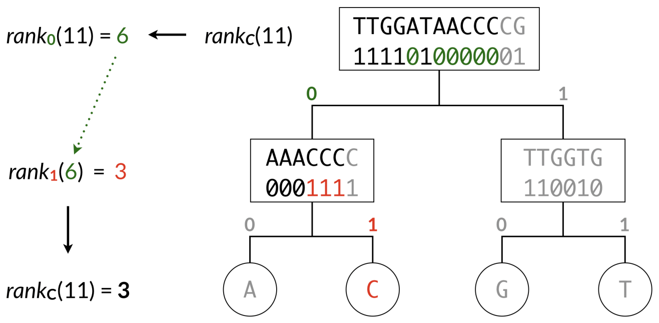

Fig. 2: Example of a rank query on a wavelet tree. To calculate $rank_C(11)$, we compute $rank_0(11) = X$ in the first level (where C is encoded as 0), then we compute $rank_1(X)$ in the second level (where C is encoded as 1).

time. It involves a backward search process supported by the FM-index [4], a full-text compressed index built upon the BWT of the reference sequence and its Suffix Array. 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

$$\forall\ 1 < i \leq N\ :\ \mathcal{T}[\mathcal{SA}[i-1], N]\ <\ \mathcal{T}[\mathcal{SA}[i], N]\ . \quad (1)$$

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 be in the set $\{\mathcal{SA}[i] : start(\mathcal{X}) \leq i \leq end(\mathcal{X})\}$, with

$$start(\mathcal{X}) = \min\{i : \mathcal{X} \text{ is the prefix of } \mathcal{T}[\mathcal{SA}[i], N]\} \quad (2)$$

$$end(\mathcal{X}) = \max\{i : \mathcal{X} \text{ is the prefix of } \mathcal{T}[\mathcal{SA}[i], N]\}. \quad (3)$$

Thus, searching for a sequence $\mathcal{X}$ in $\mathcal{T}$ corresponds to searching for the $\mathcal{SA}$ interval of suffixes of $\mathcal{T}$ starting with $\mathcal{X}$.

The $\mathcal{SA}$ of a string $\mathcal{T}$ is closely related to its Burrows-Wheeler Transform [3], $BWT(\mathcal{T})$, a reversible permutation of the string's characters which 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.

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, like the backward pattern matching algorithm employed in this work. Since the Burrows-Wheeler matrix is sorted in lexicographical order, rows beginning with a given sequence appear consecutively. The backward search procedure leverages this property to iteratively calculate the range of matrix rows beginning with longer suffixes of the query. In fact, Ferragina and Manzini [4] proved that if $\mathcal{X}$ is a substring of $\mathcal{T}$:

$$start(a\mathcal{X}) = C(a) + Occ(a, start(\mathcal{X}) - 1) + 1 \quad (4)$$

$$end(a\mathcal{X}) = C(a) + Occ(a, end(\mathcal{X})), \quad (5)$$

where $C(a)$ returns the number of symbols in $BWT(\mathcal{T})$ that are lexicographically smaller than a, $\forall a \in \sum$, while $Occ(a, i)$ returns the number of occurrences of x in $BWT(\mathcal{T})[1, i]$. They also proved 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 $\mathcal{T}$ by iteratively calculating $start$ and end from the end of $\mathcal{X}$. At each step, the size of the interval $[start,\ end]$ either shrinks or remains the same. When the execution stops, the rows included in such interval correspond to the occurrences of the query in the text, whose positions can be found in the correspondent range of the suffix array.

B. *Succinct Data Structure*

The $BWT(\mathcal{T})$ alone can be used to compute $C(a)$ and $Occ(a, i)$ at each step of the backward search procedure, but such an approach would result in long processing times. For this reason, many data structures have been proposed in the literature to efficiently answer the $C(a)$ and $Occ(a, i)$ queries, aiming at achieving the best trade-off possible between memory usage and execution time. In this work, we employed a combination of succinct data structures to encode the BWT of the reference sequence in a minute space, while still allowing for fast queries on it.

Succinct data structures [15] are defined as data structures which use an amount of space close to the information-theoretic lower bound, but still allows for efficient query operations. Unlike general lossless compressed representations, succinct data structures have the ability to use data in-place, without decompressing them first. Here, we combine two different data structures to efficiently encode the BWT sequence.

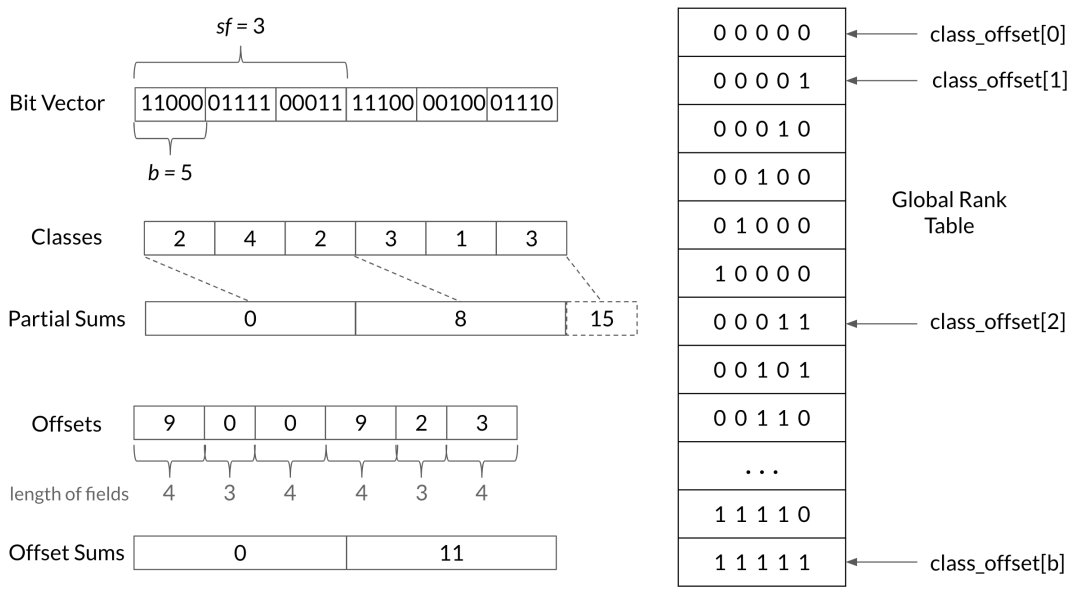

Fig. 3: Overview of our data structure implementing the RRR sequence of a bitvector. The bitvector is not actually stored in the data structure; it is shown in figure only for the seek of clarity.

First, a balanced wavelet tree is employed to represent the sequence as a series of bit-vectors; then, each bit-vector is encoded as an RRR structure, as showed in Figure 1.

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, where $|\sum|$ is the size of $\sum$. 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. After the tree is constructed, a rank query can be done with $\log_2(|\sum|)$ binary rank queries on the bit-vectors, as showed in Figure 2. In this work, we implemented the wavelet tree 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.

We optimized the data structure for sequences from alphabets of 2^N elements, with $N \geq 2$, as in the case of genomic sequences from the alphabet $\{A, C, G, T \parallel U\}$. In this case, instead of storing the special character "$" in the wavelet tree, we store its BWT position in a separate variable, which is checked in the backward search function to adjust the rank queries used for calculating the $start$ and end indices. This avoids to enlarge the alphabet size and to introduce a third deeper level in the tree, thus reducing the data structure's size and the time required by queries on it.

Although the wavelet tree is already a powerful structure, the core of our implementation is the representation of wavelet nodes' bit-vectors, based on the theoretical proposal in [16] by Raman, Raman, and Rao (*RRR*). Their data structure allows 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 $NH_0(\mathcal{B}) + o(N)$ space, where $H_0(\mathcal{B})$ is the *zero-order empirical entropy* of $\mathcal{B}$. The RRR data structure relies on the division of the bit sequence $\mathcal{B}$ into blocks of b bits each, then grouped in superblocks of $sf \times b$ bits, where sf is called *superblock factor*.

In our implementation of the RRR sequence, shown in Figure 3, we combine the theoretical proposal in [16] with commonsense decision. To encode a bit-vector $\mathcal{B}[1, N]$, with a block size b and a superblock factor sf, our data structure contains seven fields:

- an array of $\frac{N}{b}$ 4-bits fields 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;
- a pointer to a shared array of *permutations* P (called Global Rank Table in Figure 3), which contains 2^b short unsigned integers (16 bits) storing all the possible blocks of b bits, sorted per class and then in ascending order;
- a pointer to a shared array of $b + 1$ unsigned integers storing the *class offset* of each class c, that is 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;

- an array of three unsigned integers storing, respectively, the length of the bit-vector $\mathcal{B}$ (N), the block size (b), and the superblock factor (sf).

This implementation allows to answer binary rank queries on a bit-vector $\mathcal{B}[1, N]$ in $O(sf)$ time, through the procedure in Algorithm 1, while occupying only

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

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. Moreover, when encoding BWT sequences from any alphabet $\sum$ of size $|\sum| \geq 3$, 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 the RRRs encoding all the wavelet nodes.

In summation, our data structure enables answering rank operations in $O((\log_2 |\sum|)sf)$ time for any arbitrary sequence from an alphabet $\sum$. Moreover, since the size of the *offset* field of each RRR sequence depends only on the zero-order empirical entropy of the bit sequence, our data structure can efficiently represent the BWT of long sequences. This is due to the fact that the transformation rearranges a sequence into runs of similar symbols, significantly reducing its entropy. Therefore, the nodes of the wavelet tree, which represents the BWT sequence, will be low-entropy bit-vectors, which can be efficiently encoded as RRR sequences.

```
Input: $RRR$ struct of the bitvector $\mathcal{B}$, 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 $PartialSum[p/(sf \times b)]$
else if $(p \ mod \ b) == 0$ then
    for $i = sf \times (p/(sf \times b))$ to $p/b$ do
        $count = count + class[i]$;
    return $PartialSum[p/(sf \times b)] + count$
else
    $OffsetPos = OffsetSum[p/(sf \times b)]$;
    for $i = sf \times (p/(sf \times b))$ to $p/b$ do
        $count = count + class[i]$;
        $OffsetPos = OffsetPos + \lceil \log_2 \binom{b}{class[i]} \rceil$;
    $off \leftarrow \lceil (\log_2 \binom{b}{class[p/b]}) \rceil$bits from offset bit-vector;
    $cloff = ClassOffset[class[p/b]]$;
    $count += rank_1(Permutation[cloff + off], p \ mod \ b)$;
    return $PartialSum[p/(sf \times b)] + count$
```

Algorithm 1: Binary rank function on RRR sequences

C. Architecture Implementation

After the BWT structure is computed, it can be employed for mapping reads onto the reference sequence in an efficient way. The mapping process 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. Xilinx SDAccel has been used to perform the bitstream generation step and for the runtime management of the application. The host code of the application has been written using OpenCL, and the communication between host and FPGA device is performed using the OpenCL APIs.

The execution flow starts by transferring the BWT data structure on the FPGA. The data are then stored on the on-chip Block RAM (BRAM) of the device, in order to increase data locality and avoid the long latency caused by accessing memory cells of an external DRAM. For our hardware implementation, we fixed the block size $b = 15$ for each RRR sequence, while we decided to allow for any superblock factor $sf >= 50$. The low memory footprint of the data structure allows us to load on a single FPGA genomic sequences as long as human chromosomes, containing up to $\sim$100 millions bases. 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. 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 some additional information. We modeled the query size to exploit the memory burst and, to store sequences long up to 176 bases, enough for the targeted application. When a query sequence $\mathcal{X}$ is loaded, its reverse complement $\overline{\mathcal{X}}$ is computed. Then, both $\mathcal{X}$ and $\overline{\mathcal{X}}$ are mapped onto the reference. The backward search for $\mathcal{X}$ and $\overline{\mathcal{X}}$ is executed in parallel, leveraging fast bit-level rank operations on the BWT structure. 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 of the suffix array, $\mathcal{SA}[start(\mathcal{X}), end(\mathcal{X})]$, and $\mathcal{SA}[start(\overline{\mathcal{X}}), end(\overline{\mathcal{X}})]$.

D. BWaveR: a hybrid DNA sequence mapper

The described pattern matching design is integrated in BWaveR, a heterogeneous DNA sequence mapper, leveraging FPGAs for improving the performance over power ratio. Figure 4 shows an outline of the proposed system. The system is accessed through an intuitive web interface, which allows users to upload the reference and query sequences as FASTA and FASTQ files respectively, both in uncompressed or gzipped

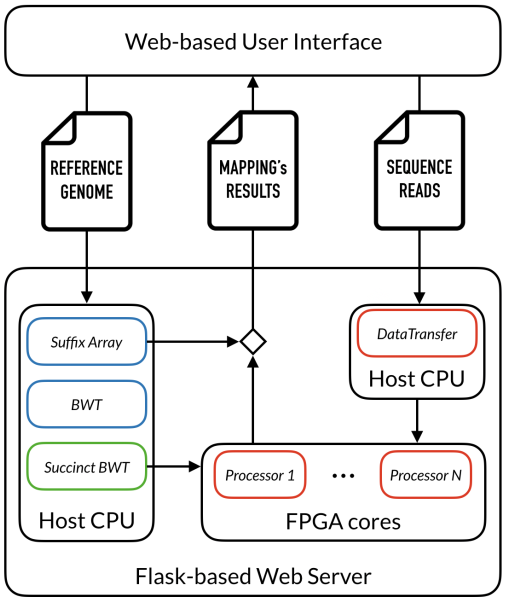

Fig. 4: Outline of BWaveR hybrid sequence mapper. The colors of the blocks are based on the 3 steps of the workflow: blue for *BWT and SA computation*, green for *BWT encoding*, red for *Sequence mapping*.

formats. A Python web-server, built with Flask, manages the user interface, the data transfer from and to the user PC, and the execution of the pipeline. The overall work-flow can be divided into three main steps:

1) *BWT and SA computation:* the FASTA (gzipped or uncompressed) file containing the reference sequence is processed to compute its BWT and its suffix array, which are then stored in a file;
2) *BWT encoding:* the BWT sequence is used to build the succinct data structure described in Section III-B;
3) *Sequence mapping:* the sequences contained in the FASTQ (gzipped or uncompressed) file are mapped onto the reference sequence through the backward search on the previously created data structure.

The host CPU is in charge of the first two processing steps and of all the memory management. Instead, the reads mapping task, is assigned to the FPGA co-processor to speed-up the process and improve the performance over power ratio of the system. After all the reads have been processed, the results are made available and can be downloaded by the user. Thus, our system allows users to leverage hardware acceleration to perform an efficient genomic analysis, without any effort nor any knowledge of the underlying hardware architecture.

IV. Experimental Results

The 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. The final design has been benchmarked on a Xilinx Alveo U200 powered by an UltraScale+ XCU200 FPGA on the Nimbix Cloud computing platform. We

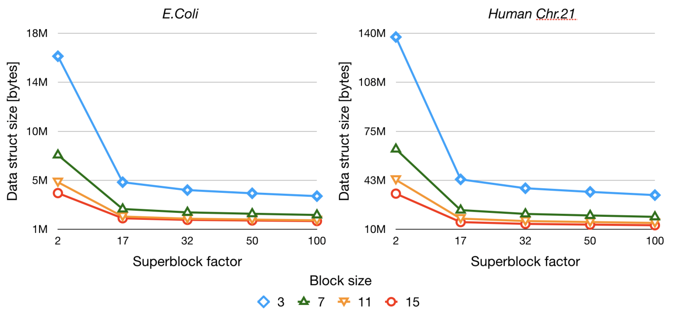

Fig. 5: Data structure's size for E.Coli (left) and Human Chr.21 (right), with different combination of b and sf parameters

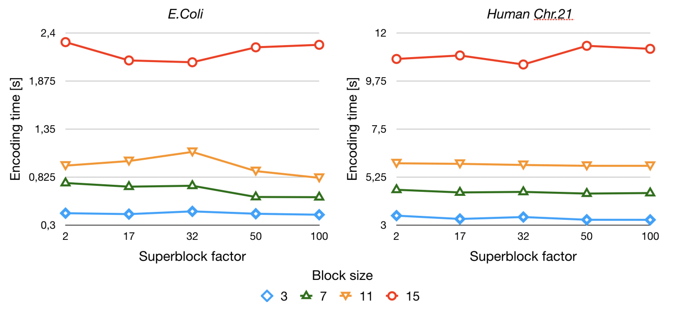

Fig. 6: Data structure's building time for E.Coli (left) and Human Chr.21 (right), with different combination of b and sf parameters

compared the performances of our architecture with a software version of the same tool that aims to minimize the memory footprint. We also compared the performances of our design to those of Bowtie2 [9], that was run with the options `-a` and `--score-min C,0,-1` in order to find all and only the exact matches between the query sequences and the reference. We run all the software tests on server with an Intel Xeon E5-2698 v3 with a peak frequence of 2.3 GHz and 128 GB of RAM. For the power consumption analysis we used the reference values of $135W$ for the Intel Xeon and $25W$ for the Xilinx Alveo U200.

To provide a characterization of the proposed succinct data structure, different tests were run both on the complete

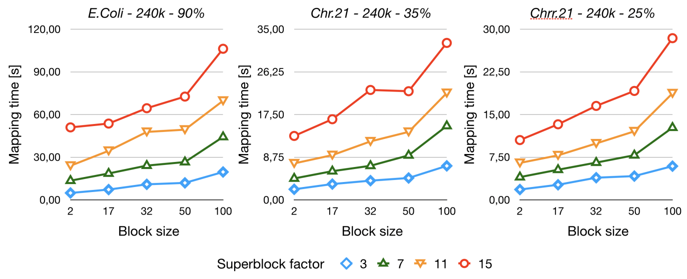

Fig. 7: Comparison of the time required for mapping 240k reads on E.Coli (left) and Human Chr.21 (center and right) references with different percentage of mapped reads.

genome of the Escherichia Coli (sequence U00096.3) on the and Chromosome 21 from the GRC Human Build 38 patch release 12 (GRCh38.p12). Many combinations of the parameters defining the data structure (i.e. *block size* and *superblock factor*) were employed for the tests. Figure 5 reports the memory required by the BWT structure of the E.Coli and of the Human Chr.21, showing that increasing the *block size* and *superblock factor* allows to achieve a better compression. The uncompressed representation (1 byte per character) requires respectively ~4.64 MB and ~40.1 MB for the storing the BWT of the employed references. The proposed data structure, instead, requires ~1.72 MB for encoding the E.Coli's BWT and ~12.73 MB for the Human Chr.21 when $b = 15$ and $sf = 100$, reducing the memory requirements up to 68.3%. In Figure 6 we can see the time required for building the BWT structure for E.Coli and for the Human Chr.21 sequences, with respect to the *block size* and *superblock factor* parameters. Results show that the encoding time has a direct dependence from the block size, while it is almost constant when the superblock factor is changed.

Figure 7 shows the time required for mapping ~240K sequences of 100bp to the E.Coli and Human Chr.21 references, with different mapping ratios. Results show a direct proportion between both the b and sf parameters and the mapping time. Moreover, we can see that the search time is not dependent on the size of the reference genome. Instead, it depends only on the number of processed reads and on percentage of reads which find occurrences in the reference genome (mapping ratio). This is due to the fact that the backward search algorithm processes entirely only those reads that actually map on the reference sequence, while it stops earlier if the searched pattern doesn't occur.

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. For the tests presented in Table I and Table II we used $sf = 50$ for all the executions of BWaveR, both on CPU and FPGA. Table I reports the performances of BWaveR and Bowtie2 when mapping 100 millions 35bp reads, and their reverse complements, on the E.Coli reference genome. It shows that our implementation on a single FPGA core is 68.23× faster than the equivalent software solution, and 17.2× more efficient than Bowtie2 when run on 16 threads.

Table II shows the results of the alignments on a varying number of reads of length 40bp on the Chromosome 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 our execution time, 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 very high number of alignments. When aligning 100M reads we achieve a 4.91× speed-up against Bowtie2 running on 16 threads, and a 70.39× speed-up when compared to our software implementation. The low power consumption of the FPGA also allowed us to be 380.09× more efficient than our software implementation, and up to 274.2× more efficient than Bowtie2 on single thread.

TABLE I: Performances of BWaveR and Bowtie2 when aligning 100 millions 35bp 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×
Power efficiency	1×	368,43×	263,32×	34,3×	17,2×

TABLE II: Performances of BWaveR and Bowtie2 when aligning 1, 10 and 100 millions 40bp reads to Chromosome 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×
	Power 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×
	Power 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×
	Power efficiency	1×	380,09×	274,2×	51,34×	26,52×

V. Conclusions

In this paper we presented BWaveR, a hybrid system to accelerate the mapping of short DNA fragments to a reference genome, based on succinct data structures. We have described the implementation of a parametrizable data structure that provides a flexible solution to the Burrows-Wheeler mapping problem. The possibility of controlling the memory/time behavior of the data structure makes this encoding suitable for various applications, on different platforms. The data structure has been employed to develop a DNA sequence mapping design able to find exact matches of short DNA fragments in a reference genome. We have also proved that FPGAs can be leveraged to improve the execution efficiency of the bit-level operations involved in such design. Our implementation on a single Xilinx Alveo U200 achieves a speedup of up to 70× over its equivalent software implementation. It also results to be 4,9× faster and 26,5× more energy efficient than Bowtie2 run on 16 threads.

Future work involves to extend our mapping design to approximate string matching, and to allow reference sequences longer than 100 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.

References

[1] Blanca de Unamuno Bustos, Rosa Murria Estal, Gema Pérez Simó, Inmaculada de Juan Jimenez, Begoña Escutia Muñoz, Mercedes Rodríguez Serna, Victor Alegre de Miquel, Margarita Llavador Ros,

Rosa Ballester Sánchez, Eduardo Nagore Enguídanos, Sarai Palanca Suela, and Rafael Botella Estrada. Towards personalized medicine in melanoma: Implementation of a clinical next-generation sequencing panel. *Scientific Reports*, 7(1):495, 2017.

[2] Yupeng Chen, Bertil Schmidt, and Douglas L Maskell. A hybrid short read mapping accelerator. *BMC bioinformatics*, 14(1):67, 2013.

[3] Michael Burrows and David J Wheeler. A block-sorting lossless data compression algorithm. 1994.

[4] Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. In *Proceedings 41st Annual Symposium on Foundations of Computer Science*, pages 390–398. IEEE, 2000.

[5] Thomas C Conway and Andrew J Bromage. Succinct data structures for assembling large genomes. *Bioinformatics*, 27(4):479–486, 2011.

[6] E. B. Fernandez, J. Villarreal, S. Lonardi, and W. A. Najjar. Fhast: Fpga-based acceleration of bowtie in hardware. *IEEE/ACM transactions on computational biology and bioinformatics*, 12(5):973–981, 2015.

[7] 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)*, 14(3):668–677, 2017.

[8] Heng Li and Richard Durbin. Fast and accurate short read alignment with burrows–wheeler transform. In *Bioinformatics*, 2009.

[9] Ben Langmead and Steven L Salzberg. Fast gapped-read alignment with bowtie 2. *Nature methods*, 9(4):357, 2012.

[10] 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*, pages 133–138, May 2015.

[11] Hasitha Muthumala Waidyasooriya, Daisuke Ono, Masanori Hariyama, and Michitaka 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)*, page 354, 2015.

[12] Ho-Cheung Ng, Shuanglong Liu, and Wayne Luk. Reconfigurable acceleration of genetic sequence alignment: A survey of two decades of efforts. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–8. IEEE, 2017.

[13] Edward Fernandez, Walid Najjar, and Stefano Lonardi. String matching in hardware using the fm-index. In *2011 IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 218–225. IEEE, 2011.

[14] James Arram, Kuen Hung Tsoi, Wayne Luk, and Peiyong Jiang. Reconfigurable acceleration of short read mapping. In *2013 IEEE 21st Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 210–217. IEEE, 2013.

[15] Guy Joseph Jacobson. Succinct static data structures. 1988.

[16] Rajeev Raman, Venkatesh Raman, and S. Srinivasa 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, pages 233–242, Philadelphia, PA, USA, 2002. Society for Industrial and Applied Mathematics.