Compressed Sparse FM-Index: Fast Sequence Alignment Using Large K-Steps

Rubén Langarita, Adrià Armejach, Javier Setoain,
Pablo Ibáñez-Marín, Jesús Alastruey-Benedé, Miquel Moretó

Abstract—The FM-index is a data structure used in genomics for exact search of input sequences over large reference genomes. Algorithms based on the FM-index show an irregular memory access pattern, resulting in a memory bound problem. We analyze a recent implementation of the FM-index and highlight existing throughput-memory trade-offs, showing that memory requirements limit implementation of large k-steps. We propose COFI, a **CO**mpressed **F**M-**I**ndex for large K-steps. COFI enables a 15-step FM-index using less than 16 GB for a human genome reference of 3 giga base pairs. An algorithm based on this new layout is evaluated on both a Knights Landing (KNL) and an Skylake-based system (SKX). We achieve average speed-ups of 1.46× and 1.39×, respectively, with respect to an state-of-the-art FM-index implementation that is already well optimized.

Index Terms—Compressed FM-index, sequence alignment, Knights-Landing, Skylake.

1 Introduction

Precision medicine holds promise for improving healthcare by leveraging genomic information. Due to the steep decrease in genome sequencing costs in recent years, the amount of data to be processed is increasing dramatically, leading to a significant computation and storage challenge. High-throughput sequencing systems produce a large amount of reads that need to be post-processed. These reads will typically be used for several genomic studies based on sequence alignment pipelines. Most of these software packages require several CPU hours to perform each of these studies [1].

The objective of sequence alignment is to find for each read the best matching locations when compared to a reference genome. In order to reduce the computation and memory requirements, exact matching is performed to restrict the search space. Recent work shows that popular software packages are able to align up to 80% of the reads without errors (exact matches) [2]. Reads that cannot be aligned with exact matching need to go through additional steps. A common approach is to use a seed-and-extend algorithm where reads are partitioned into small chunks (seeds). These chunks are again searched using exact matching in order to find seeds in the reference genome. Candidates are then assigned an alignment score in the extend phase, typically using a dynamic programming scheme based on the Smith-Waterman local alignment algorithm [3].

- *R. Langarita is with the Barcelona Supercomputing Center, Barcelona, Spain. E-mail: {ruben.langarita}@bsc.es*
- *A. Armejach and M. Moretó are with the Barcelona Supercomputing Center, Barcelona, Spain and Universitat Politècnica de Catalunya, Barcelona, Spain. E-mail: {adria.armejach, miquel.moreto}@bsc.es*
- *J. Setoain is with Arm, Cambridge, United Kingdom. E-mail: {Javier.Setoain}@arm.com*
- *P. Ibañez-Marín and J. Alastruey-Benedé are with Departamento de Informática e Ingeniería de Sistemas, Universidad de Zaragoza, María de Luna 1, 50018 Zaragoza, Spain. E-mail: {imarin,jalastru}@unizar.es*

Manuscript received xxx xx, xxxx; revised xxx xx, xxxx.

Exact matching is therefore a key component in sequence alignment pipelines. Since typically used reference genomes are in the range of giga base-pairs (Gbp), significant efforts are needed to reduce memory size requirements. For this reason, many popular sequence aligners are based on the FM-index structure [4, 5, 6, 7, 8], which is well suited for fast exact matches of short reads to a large reference genome. A recent study that characterizes one of these popular tools (BWA-MEM2), shows that time spent using the FM-Index is significant. It consumes between 20% and 45% of the execution time [7].

In this paper we analyze throughput and memory trade-offs in state-of-the-art solutions based on FM-index structures. We show that while a linear improvement in computational throughput can be achieved by increasing the number of bases searched per step (k), the memory requirements quickly become prohibitive as they increase exponentially. To overcome this limitation we propose COFI, a **CO**mpressed **F**M-**I**ndex for large K-steps. We make the following contributions:

- We provide a detailed memory footprint analysis for a state-of-the-art FM-index implementation. We make the key observation that the amount of useful data stored within the FM-index structures remains constant as k increases.
- Based on this observation we propose COFI, a compression scheme for the FM-index data structures and an accompanying algorithm to navigate them. Unlike in prior approaches, COFI's main data structure has a constant size with respect to k. This enables search steps to be performed over a large number of bases at a time. In particular, COFI can perform 15 *k-steps* with a manageable memory footprint of 16 GB.
- We evaluate COFI on two different modern hardware platforms: an Intel Xeon Phi 7230 (KNL) and an Intel Xeon Platinum 8160 Skylake-based (SKX). We employ two reference genomes and a representative set of eleven

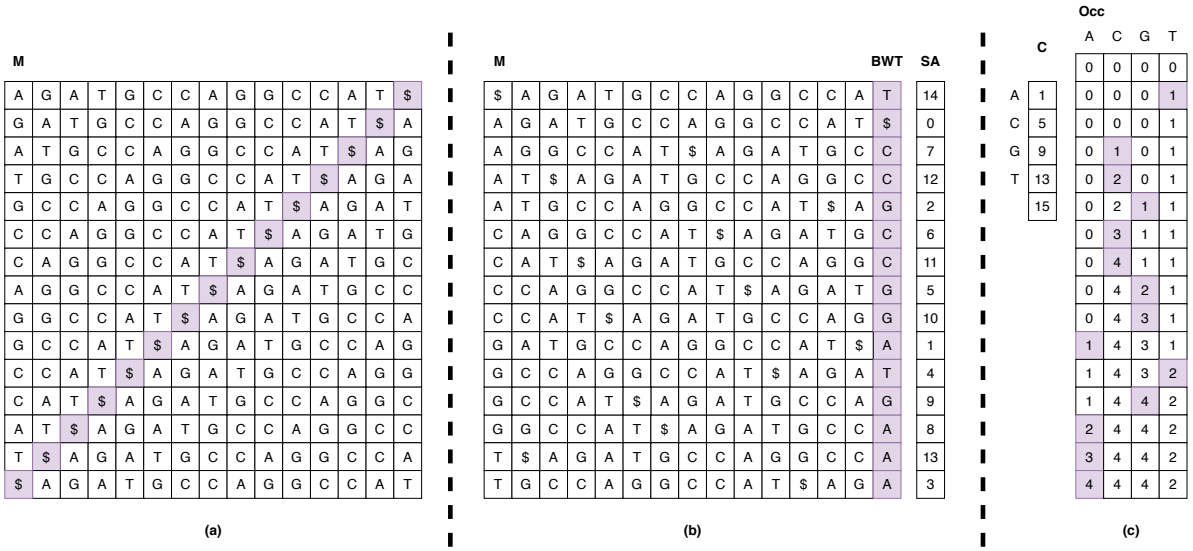


Fig. 1: Procedure to build the FM-index. (a) Step 1: cyclic rotations to generate matrix M. (b) Steps 2 and 3: sort M alphabetically and extract the last column (BWT) and the suffix array (SA). (c) Step 4: build C and Occ structures.

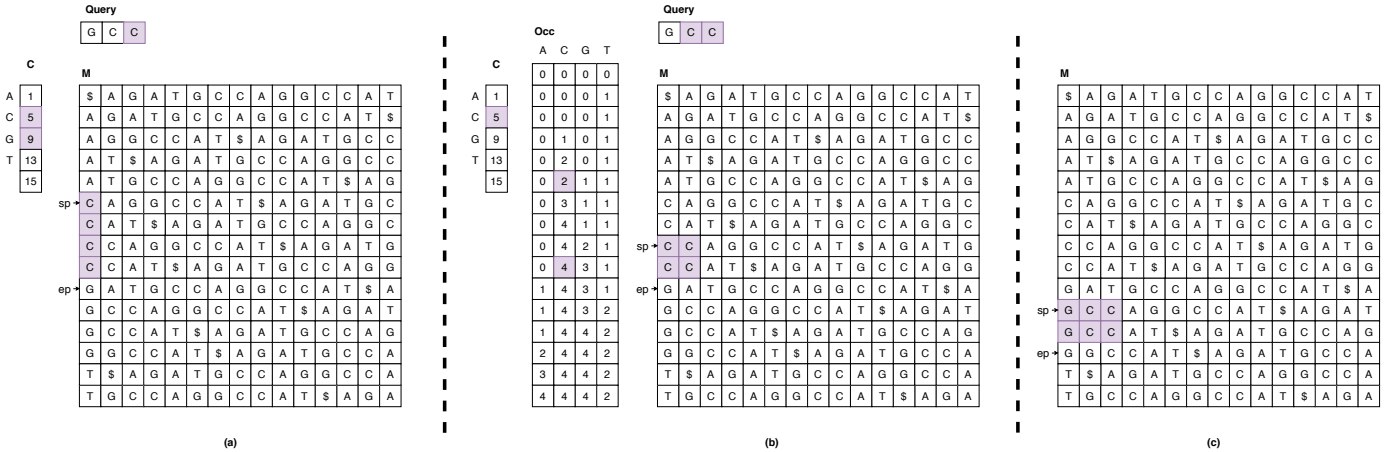


Fig. 2: Backward search example. (a) sp and ep initialization. (b) LF operation for sp and ep. (c) Final state of the query.

different input sequences. We show that COFI consistently outperforms a state-of-the-art FM-index implementation for all inputs, with average speed-ups of 1.46× and 1.39× in KNL and SKX, respectively.

2 Background

The FM-index is a data structure that allows fast string searches over large texts [9]. It is widely used in multiple genomics pipelines to find exact matches of DNA sequences on a reference genome [10, 11]. In particular, in genome sequence alignment where query sequences typically have a few hundred bases and references can range from a subset of chromosomes to entire genomes. As an indication, the human genome is around 3 giga base pairs (Gbp). Given a fixed alphabet, the complexity of a sequential search is $O(n)$, where n is the length of the reference, while a query based on the FM-index has complexity $O(m)$, where m is the length of the sequence to be searched.

In this section, we first describe how to build the FM-index and the search algorithm based on this data structure. Then, we explain several optimizations proposed by prior work that aim to reduce the memory footprint and to speed up the computation.

2.1 FM-index: Construction and Usage

To construct the FM-index of a reference string, its Burrows-Wheeler Transform (BWT) has to be computed [12]. This is achieved by appending a special symbol \$, which is lexicographically smaller than all other symbols, and by performing all the circular shifts as shown in Figure 1a for the string "AGATGCCAGGCCAT". Then, all the rows of the resulting matrix M are sorted lexicographically, as shown in Figure 1b. The last column of M is the BWT of the reference string. For each row of M, its starting position in the original reference is stored in an structure called suffix array (SA) [13], see Figure 1b. When a query finishes, the suffix array is looked up to locate the matching positions in the original reference.

The two structures that constitute the FM-index, *C* and *Occ*, can be generated from the BWT (see Figure 1c). *C* is an array of size $\sigma + 1$, where σ is the number of symbols in the

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCBB.2020.3000253, IEEE/ACM Transactions on Computational Biology and Bioinformatics

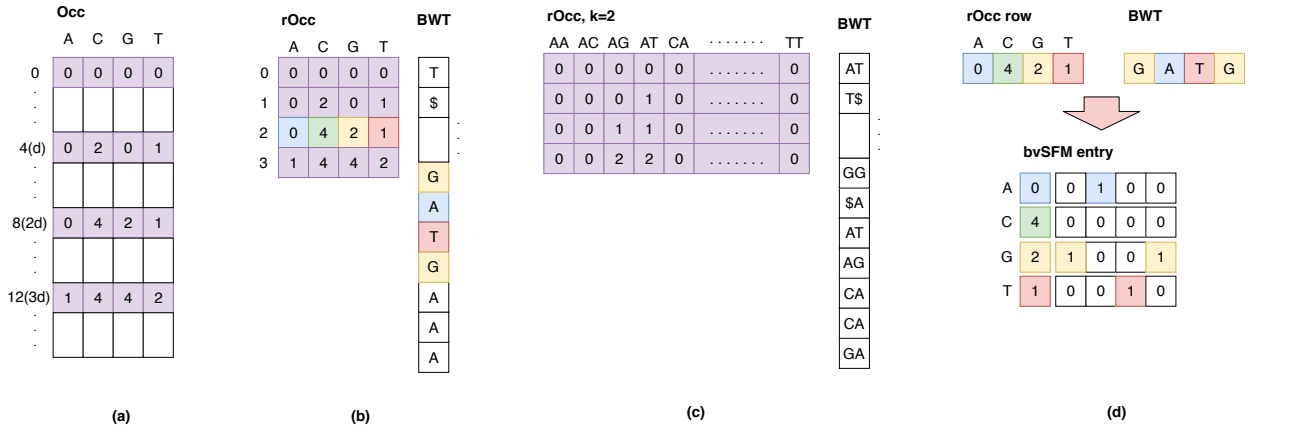


Fig. 3: Several FM-index implementations. (a) Full FM-index with reference length n. (b) Sampled FM-index, one out of d counters is stored. (c) 2-step sampled FM-index, uses an alphabet of 16 symbols. (d) Split bit-vector FM-index, each bit indicates whether a base is in a given BWT position or not.

alphabet. Since DNA has 4 symbols (bases) $\{A, C, G, T\}$, the size of C is 5. $C[s]$ contains the number of bases alphabetically smaller than s in BWT. Occ has a column for each base, and a row for each BWT symbol. Each row corresponds to a BWT position and contains the number of bases of each type in the BWT up to (but not including) that position. Hence, $Occ[p, s]$ contains the number of occurrences of the base s in the BWT range $[0, p)$. For example, in Figure 1c, the $Occ[6, 3]$ cell indicates the number of Ts until the 6$^{\text{th}}$ position of the BWT, that is, in the range [0-5).

To perform exact matching of a query $Q[1..n]$, the *backward search* (BS) algorithm based on the FM-index can be employed [9]. BS defines two pointers (start and end, *sp* and *ep*) that indicate the range where are all the possible candidates of the current search appear in the matrix M. Note that M is not a necessary data structure, we show its instantiation for clarity. BS starts processing the last base in Q, i.e., $s = Q[n-1]$), by initializing the sp and ep pointers with $C[s]$ and $C[s+1]$, respectively (see Equations 1 and 2). Figure 2a shows an example of this stage for the sequence GCC. For each of the remaining bases in the query string, two Last-to-First Mapping operations (LF) are performed, one for each pointer (sp and ep). LF is defined in Equation 3 for a pointer p and a base s [9].

$$sp = C[s] \tag{1}$$

$$ep = C[s+1] \tag{2}$$

$$p = LF(p, s) = C[s] + Occ[p, s] \tag{3}$$

Following the example, Figure 2b shows the effect of the LFs corresponding to the second base $Q[n-2] = C$. The updated pointers indicate the range of rows in M that would contain the partial query we have searched so far. Finally, Figure 2c shows the final position of the pointers after performing all the LF operations. The values of sp and ep indicate the first and last rows of M prefixed by the query $Q[1..n]$.

Each row of M corresponds to one rotation of the original reference. With the SA, we can obtain the starting positions in the original reference of the matches found between sp and ep.

2.2 Sampled FM-index

The Occ structure requires as many rows as the length of the reference (Figure 3a), leading to a large memory footprint. As an example, for the human genome, the Occ structure would occupy around 48 GB. To reduce this footprint, a sampling technique that stores one row out of every d rows of Occ can be applied [14]. The reduced structure, called $rOcc$ (see Figure 3b), can be defined by the following expression: $rOcc[p, s] = Occ[p \times d, s]$. This reduction in memory footprint comes at a computational cost. In order to reconstruct the discarded entries, both the reduced $rOcc$ and the original BWT are needed. The additional computation uses the BWT to count the bases from the last sampled counter in $rOcc$ to the desired position, as in Equation 4.

$$Occ[p, s] = rOcc[p/d, s] + occur(s, BWT[(p - p \bmod d)..p]) \tag{4}$$

For instance, to compute the number of 'A' symbols up to the tenth row of Occ, we add the counter corresponding to the 'A' base found in the second row of the $rOcc$, i.e., 0 (see 'A' counter in blue in Figure 3b), to the number of occurrences of that base in the first two positions of its corresponding BWT block, i.e., 1 (see coloured BWT-block in Figure 3b).

2.3 K-step Sampled FM-index

The *k-step sampled FM-index* (see Figure 3c) searches k bases in a single step [15]. k bases per iteration are processed instead of one, and the rest of the algorithm remains the same.

For example, for $k = 2$ the alphabet changes from $\{A, C, G, T\}$ to $\{AA, AC, AG, AT, CA, ..., TT\}$. The size of C and $rOcc$ structures increases by a factor of 4 (the size of the original alphabet), every time k is incremented by one. In this case, the trade off is an exponential memory footprint increase for a linear increase in computation throughput as k increases.

2.4 bvSFM: Improving Data Locality

The *split bit-vector sampled FM-index* (*bvSFM*) speeds up computation by improving data locality [16]. The sampled

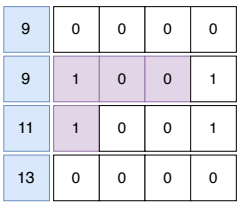


Fig. 4: $bvSFM$ column for the symbol G. Each $bvSFM$ entry has a counter (blue boxes, left) and a bit vector (right, 4 bits). $C['G'] = 9$ is already added to the counters, so there is no need to add it during the LF operation as in Equation 3.

FM-index layout is changed in order to store all the data needed to compute an LF in a single cache block. The counter for a base in a sampled $rOcc$ row is placed close to a bit-vector that encodes the occurrences of that base in the corresponding BWT block. The length of each bitmap in a $rOcc$ row is equal to the sampling rate. A bit set to 1 indicates that the BWT contains the symbol at that position, as shown in Figure 3d. Instead of iterating the BWT to count the occurrences of a base, we just need to count the number of bits set in the bit vector. This operation can be efficiently implemented by the *popcount* instruction. We term $bvSFMentry$ the set of sampled counters and their corresponding bitmaps in a $rOcc$ row (see Figure 3d). As proposed in Chacón et al. [15], a preprocessing step adds the C values to the $rOcc$ counters in order to avoid a C read and an addition operation for each LF.

2.5 bvSFM example

To illustrate how this approach works, lets consider the FM-index shown in Figure 3b. We take as example the last LF operation of the example shown in Figure 2b. The current values of the pointers are $sp = 7$ and $ep = 9$, and the next symbol to be processed by the backward search algorithm is $Q[i] = G$. Figure 4 shows the $bvSFM$ column corresponding to the symbol G.

The two LF operations, one for sp and one for ep, required for each symbol s in the query are performed as follows:

- Calculate the $bvSFM$ entry indexes: p/d ($7/4 = 1$ for sp, and $9/4 = 2$ for ep).
- Load the $bvSFM$ entries, which contain the sampled counters $rOcc[p/d, s]$ ($rOcc[1, G] = 9$ for sp and $rOcc[2, G] = 11$ for ep), and its corresponding bit-vectors in contiguous memory locations (1001 for both sp and ep).
- Calculate the bit-vector indexes: $p \bmod d$ ($7 \bmod 4 = 3$ for sp and $9 \bmod 4 = 1$ for ep).
- Select the bits to perform the $popcount$. These bits, shaded in Figure 4, are obtained by performing a bit-wise and operation between the bit vector and a mask of "$p \bmod d$" 1s (1110 and 1000 masks for sp and ep, respectively).
- Perform a $popcount$ over the selected bits ($popcount(100_2) = 1$ for sp and $popcount(1_2) = 1$ for ep).
- The number of occurrences of the symbol G up to each pointer position is computed by adding the sampled counter to the result of the $popcount$ operations: $rOcc[p/d, s] + popcount(masked\_bit\_vector)$ ($9 + popcount(100_2) = 9 + 1 = 10$ for sp, and $11 + popcount(1_2) = 11 + 1 = 12$ for ep).

The values of sp and ep will be 10 and 12, respectively.

3 Trade-offs in FM-index Optimizations

3.1 Summary

The FM-index algorithm is widely used in genomics toolflows as it enables linear-cost exact matching over large references with manageable memory footprints. The most efficient way to increase computational throughput is by increasing the number of symbols searched per step (k). However, it quickly imposes a prohibitive memory consumption. With $k = 2$, the full FM-index is almost 200 GB for the human genome. Therefore, sampling techniques must be used to lower the memory footprint, trading-off some of the performance gains, as additional computation is necessary. Finally, *bvSFM*, a state-of-the-art technique, proposes an FM-index layout that uses bitmaps to speed-up this additional computation while improving memory locality.

The $bvSFM$ structure of an FM-index with a sampling rate of $d = 64$ and $k = 2$ steps occupies 12 GB. This was found to be the best configuration as increasing k further led to worse memory management. Moreover, increasing the sampling rate to keep the memory footprint in check requires a chain of dependent *mod* and *popcount* operations that degrades performance.

3.2 Memory Footprint Analysis

The FM-index memory usage grows exponentially with k. However, the difference between two consecutive rows of the Occ structure is just on one of the counters (see Figure 1c). For instance, with $k = 5$, each Occ row has 1024 counters (4^k) but only one changes its value from one row to the next. Thus, the fraction of *useful counters* is $1/4^k$, i.e., one per row. With a sampled FM-index, at most d counter increments occur over consecutive rows of the $rOcc$. Hence, the maximum fraction of *useful counters* is $d/4^k$. A key observation is that the maximum number of *useful counters* in a $rOcc$ row is equal to d, regardless of the value of k. Therefore, as k grows, the fraction of *useful counters* decreases exponentially.

For $bvSFM$, the 4^k bitmaps in a $bvSFMentry$ have a constant number of set bits, equal to the sampling factor d (length of each bitmap). That is, if $BWT[i]$ is equal to the symbol s, the bitmap of the symbol s contains a set bit (1) in the i^{th} position, and the rest of the symbols have an unset bit (0) in the i^{th} position of their bitmaps (see Figure 3d). The fraction of *useful bits* in the bitmaps is $1/4^k$, and the total number remains constant regardless of the value of k. For instance, with $k = 10$ and $d = 64$, each $bvSFM$ entry contains 64 Mb ($64 * 4^{10}$ bits), of which only 64 are set.

Figure 5 shows how the size of a sampled ($d = 64$) $bvSFM$ structure increases with k (black line, right y-axis), quickly becoming intractable. Stacked bars show the percentage of the $bvSFM$ size dedicated to counters (white background) and bitmaps (gray background). Moreover, we also show the percentage of the $bvSFM$ size allocated to

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCBB.2020.3000253, IEEE/ACM Transactions on Computational Biology and Bioinformatics

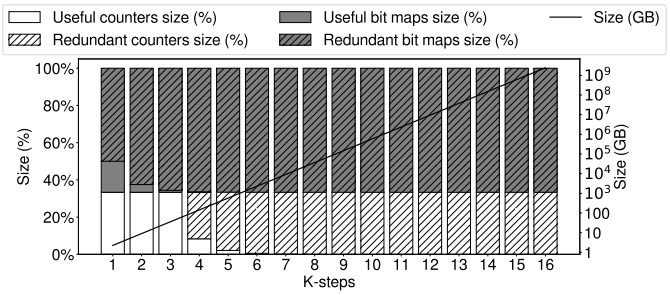


Fig. 5: Size, useful and redundant data of the $bvSFM$ structure for $d = 64$ and different values of k.

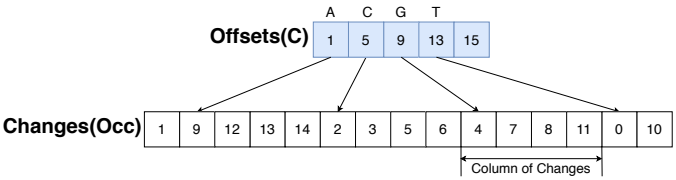


Fig. 6: *COFI* data structures: Offsets and Changes.

useful and redundant counters, and to useful and redundant bits in the bitmaps. We can observe that the amount of redundant data quickly becomes dominant. Almost all the data is redundant for $k \geq 5$, because the amount of useful data stays constant while the size of the data structure increases. Therefore, there is an opportunity to devise a new data structure and an accompanying algorithm that can perform large k-steps if we are able to store only useful data.

4 COFI: Compressed Sparse FM-index

A large k leads to better throughput. However, memory usage grows exponentially with k, quickly becoming unmanageable. By leveraging the observation that the amount of useful information stored in the FM-index remains constant, we propose to use a large *k-step* and compress the sparse FM-index information while keeping a low overhead to reconstruct the index from the compressed data structures. The following subsections introduce *COFI*, a **CO**mpressed **F**M-**I**ndex for large K-steps.

4.1 COFI Data Structures: Offsets and Changes

If we take as a reference the full FM-index Occ structure (see Figure 1c), we propose to store only the row indexes where a counter changes for each symbol in the alphabet. In other words, a column would now store the indexes where its corresponding symbol appears in the BWT. We call this new structure $Changes$. The size of $Changes$ is constant for any value of k: there are as many elements as in the BWT. The size of the columns is now variable, we store them consecutively as shown in Figure 6. In order to find where each column starts and ends, we can reuse the C structure, no modifications are needed. In *COFI*, we call this structure $Offsets$ despite being the same as the original C. If we count the occurrences of the symbol s, the limits of its column are $Offsets[s]$ and $Offsets[s+1]$. The final layout of *COFI* is shown in Figure 6.

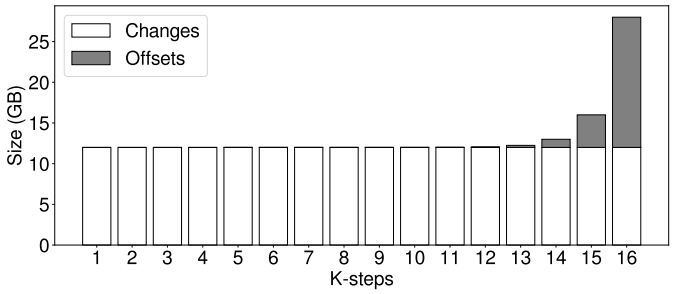


Fig. 7: *COFI* data structures size for various values of k.

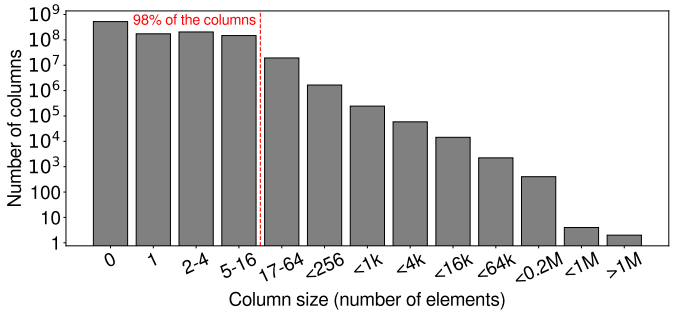


Fig. 8: Histogram of column sizes for the human reference GRCh38. Both axis are in logarithmic scale.

4.2 Memory Footprint Analysis

In previous proposals, the size of the main data structures increases exponentially with k. In *COFI*, the size of $Offsets$ also increases exponentially at the same rate. However, it is a small structure, only 4 entries for $k = 1$, or 16 entries for $k = 2$. On the other hand, $Changes$ is an array with the same number of elements as the BWT, keeping its size manageable for any value of k. Figure 7 shows the size of the proposed data structrures for a full human genome reference (GRCh38, 3 giga base pairs) and different values of k. $Changes$ occupies a constant amount of 12 GB. For $k = 15$, $Offsets$ occupies 4 GB, totaling 16 GB. As a comparison point *bvSFM* requires 12 GB with $k = 2$ and $d = 64$.

An important consideration of COFI is that the columns in the $Changes$ structure have different sizes, leading to non-uniform search times for different symbols. Figure 8 shows the distribution of the column sizes for the human reference GRCh38 with $k = 15$. The X axis represents the column size in bins that follow a logarithmic scale. The Y axis, also in logarithmic scale, indicates the number of columns within each bin. The size of most columns is smaller than that of a 64-byte cache block: 98% of the columns have 0 to 16 elements. Nevertheless, large columns can impose a large overhead and are more likely to appear during the search, as we show in our evaluation. Hence, the proposed search algorithm has to take large columns into account, even if they represent a small fraction of the total number of columns.

4.3 Backward Search in COFI

To perform the two LF operations for symbol s and pointers sp and ep in *COFI*, the following steps need to be taken:

Algorithm 1 Backward Search in COFI

Input: Q query, length = |Q|, Changes, Offsets
Output: sp, ep
1: sp = Offsets[Q[lenght-1]]
2: ep = Offsets[Q[lenght-1]+1]
3: **for** $i = length - 2$ to 0 **do**
4: left = Offsets[Q[i]]
5: right = Offsets[Q[i]+1]
6: sp = find_col_index(Changes, left, right, sp)
7: ep = find_col_index(Changes, left, right, ep)
8: **end for**
9: **return** sp, ep

1) Load $Offsets[s]$ and $Offsets[s+1]$ to obtain the start and end positions in the $Changes$ array of the column corresponding to symbol s.
2) Find in $Changes$ the first element of the column that is equal or greater than the values of *sp* and *ep*.
3) Return the indexes of the $Changes$ array for the found elements.

Algorithm 1 shows the pseudocode to perform a backward search for one sequence and $k = 1$. After initializing sp and ep in lines 1 and 2, two LF operations are performed, one for sp and one for ep, for each one of the remaining symbols. First, the start and end indexes of the column are read in lines 4 and 5. Then, the first element greater or equal than sp and ep is searched. The $find\_col\_index$ function in lines 6 and 7 returns the index of the found element in $Changes$. In the case of sp, this index is the starting position of the found sequences so far in matrix M (defined in Section 2.1), while ep indicates the first sequence that is not a match. Therefore, all sequences between sp and ep in M would currently be query matches. Finally, we return the pointers with the indexes that delimit found sequences in M (line 9).

As shown in Figure 8, the size of the columns represented in $Changes$ can be significantly large. Therefore, in order to perform the search operation quickly, the function $find\_col\_index$ implements a simple binary search algorithm. The current implementation takes advantage of the fact that columns are already sorted in ascending order. The function receives as parameters:

- $array$: array to search in.
- $left$: left limit of the binary search.
- $right$: right limit of the binary search.
- $pointer$: element to search.

As mentioned above, sp and ep keep track of the found sequences; therefore, after each iteration the distance between both pointers ($d = ep - sp$) will decrease or remain the same. Thus, we can modify the left and right limits in line 7 to perform the search over a subset of the column.

We can illustrate this optimization with the query shown in Figure 2. At the beginning of the second loop iteration, the values of sp and ep are 7 and 9 respectively (see Figure 2b). This indicates that $d = 2$ contiguous rows start with CC, a suffix of the pattern being queried. In the last iteration, once the new value of sp is calculated in line 6 of the algorithm ($sp = 10$, see Figure 2c), the $[left, right] = [9, 13]$ range for the ep search can be constrained: (i) we can set the left limit to the recent computed sp value, $left = sp = 10$, and (ii) we can set the right limit according to the maximum distance between sp and ep: $right = sp + d = 10 + 2 = 12$. This is possible because $Changes$ is sorted in ascending order and has no repeated elements. Note that this optimization is very effective because the distance between the left and right limits becomes smaller as the query progresses. This optimization can be implemented by changing line 7 of Algorithm 1 to:

$$ep = find\_col\_index(Changes, sp, sp + d, ep) \quad (5)$$

where d is calculated as $d = ep - sp$ at the beginning of the *for* loop.

4.4 COFI Example

We show how the previous example shown in Figure 4 for $bvSFM$ takes place in COFI. As a reminder, the initial values of the pointers are $sp = 7$ and $ep = 9$, and the next symbol to be processed by the backward search algorithm is $Q[i] = G$. Figure 6 shows COFI data structures with the pertinent information for column G. The operations followed in COFI are:

- Read the start and end indexes of the G column in $Changes$: $Offsets[G] = 9$ and $Offsets[G+1] = 13$.
- Apply binary search over the G column to find the elements greater or equal than sp and ep. The elements found have values 7 and 11 for sp and ep, respectively.
- Update sp and ep with the indexes of the found elements, i.e., 10 and 12, respectively.

Although the computational complexity of COFI is higher in the common case, with COFI we can perform larger *k-steps*. We can increase k to 15 in COFI while keeping a manageable FM-index size of 16 GB.

4.5 Performance Optimizations

The memory access pattern to perform an LF calculation does not present good locality. By inspecting Algorithm 1, we can observe that the LF loop first performs an access to $Offsets$, which is dependent on the current symbol, and then performs binary search over a subset (column) of the $Changes$ array. The accessed column is unlikely to be visited in the near future in the current or subsequent searched sequences, precluding temporal locality. In addition, the access pattern in binary search does not present spatial locality. Therefore, each memory access is likely to be long latency as it will miss in the caches.

Sequence interleaving: in order to hide memory latency, as previously implemented in the $bvSFM$ proposal, we search multiple sequences at the same time, overlapping long-latency memory requests. Figure 9 shows four overlapped searches in COFI, i.e., each iteration of the LF loop operates over four different query sequences. As can be seen in the figure, the size of the $Changes$ column determines the number of binary search steps for each pointer. Therefore, overlapped searches can finish at different steps in the binary search algorithm.

Software prefetching: with the same objective to reduce memory access latency, we also employ a simple software

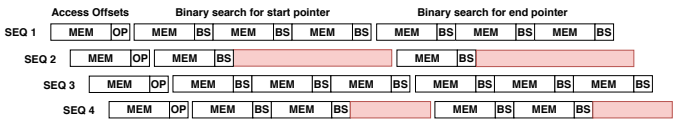


Fig. 9: Depicts four interleaved sequences to hide memory latency. *OP* denotes the time spent computing the limits of the columns, *MEM* is the time spent waiting for a response from memory, and *BS* stands for the time spent on binary search. *BS* depends on column size, therefore, some sequences finish this phase before others (red boxes).

prefetching scheme. When processing the current symbol, prefetch operations are issued for the next symbol to retrieve from memory the necessary *Offsets* elements. Additionally, the first pivot point for sp's binary search is prefetched once the column to be accessed is known.

Conditional moves: finally, COFI's LF computation is slightly more complex since it involves executing binary search. Binary search is known to exhibit poor branch prediction performance, as branches are difficult to predict, i.e., there is a 50% chance to take the branch. A branch misprediction flushes the pipeline and causes a significant performance penalty in processors with deep pipelines. Conditional move instructions can be an alternative to branches. These instructions write the contents of one register over another only if the defined predicate value is true, and can be executed speculatively on a processor's pipeline without the associated predication of branches. Therefore, in order to avoid branches, we have implemented our binary search algorithm with conditional moves.

Section 7.1 evaluates each optimization separately, and reports their contribution to performance improvements.

5 Experimental Methodology

5.1 Test Machines

To evaluate COFI, we use two different HPC platforms: (i) an Intel Xeon Phi 7230 (KNL), and (ii) an Intel Xeon Platinum 8160 Skylake-based (SKX). The main characteristics of these machines are specified in Table 1. In both machines we use the Intel C Compiler (version 18.0.1).

We deactivate hardware prefetchers in both systems, as done by the authors of *bv*SFM [16], since all evaluated algorithms do not benefit from them due to random memory access patterns. For all the evaluated proposals, disabling hardware prefetchers yield mild performance improvements between 1% and 2%. In addition, both machines are configured to use transparent huge pages of 1 GB.

5.2 Reference Genomes

We perform experiments using two human reference genomes: GRCh37 [10] and GRCh38 [11]. The column size distribution of GRCh38 can be seen in Figure 8. GRCh37 has a similar distribution. Since we extract some of the input sequences from the GRCh38 reference, as we detail in the next section, we use GRCh37 to cross-reference the input search sequences and show that similar results are achieved.

We note that GRCh37 contains around 2.86 Gbp, while GRCh38 is slightly larger with 3.05 Gbp. For $k = 15$, the size of the *Offsets* data structure is constant at 4 GB, while *Changes* occupies 10.67 GB and 11.35 GB, respectively. Therefore, all experiments performed in KNL allocate the main data structures in the MCDRAM memory region using flat mode [17]. We measure the index build time for GRCh38 on SKX and observe that it grows linearly when using k values of up to 12: for $k = \{2, 4, 8, 12\}$ it takes $\{13, 17, 24, 35\}$ minutes respectively. For $k = 15$, it takes 55 minutes, presenting a superlinear growth due to the increase in memory usage (see Figure 7). These index build times have low relevance as the index is built once and can be used across multiple experiments and projects.

TABLE 1: Test machines configuration.

		Intel Xeon Phi 7230 (KNL)	Intel Xeon Platinum 8160 (SKX)
Cores × Threads		64 × 4	24 × 2
Issue width		2	4
Frequency (GHz)		1.3	2.1
Last-level Cache (MB)		32	33
MCDRAM	capacity (GB)	16	N/A
	bandwidth (GB/s)	480	N/A
Main memory	capacity (GB)	192	96
	bandwidth (GB/s)	90	120

TABLE 2: Error rates for Mason inputs.

Inputs	Modifications	Insertions	Deletions
mason1	3%	0%	0%
mason2	1%	1%	1%
mason3	6%	0%	0%
mason4	0%	6%	0%
mason5	0%	0%	6%

5.3 Inputs

We use 11 representative sets of inputs sequences selected from different use cases; including reads from particular cell lines, as well as sequences generated using Mason [18], a read simulator.

- *sanger*: it has been extracted from the GRCh38 reference using Mason, simulating the Sanger process [19] without errors. This input has been also used to evaluate *bv*SFM [16].
- Five inputs coming from real reads made by an Illumina HiSeq 2000 machine, which outputs sequences with a length of 101 bases.
  - *ocily7-s*: reads from the cell line OCI-LY7 over RNAs smaller than 200 nucleotides [20].
  - *ocily7-1* and *ocily7-2*: reads from the same cell line OCI-LY7, but over RNAs greater than 200 nucleotides [21].
  - *a375-1* and *a375-2*: reads from the cell line A375 over RNAs greater than 200 nucleotides [22].
- *mason*$\{1..5\}$: we use Mason over GRCh38, simulating an Illumina machine to generate 5 inputs. We replicate the simulations described on the appendices of Alser et al. [23] with adjusted error rates. Table 2 shows the error introduced for each simulation.

TABLE 3: Number of sequences, length of each sequence and occurrences in GRCh37 and GRCh38 for each input.

Input	Nº seqs	Length	Occu. GRCh37	Occu. GRCh38
sanger	20M	200	15.49M	46.71M
ocily7-s	35.21M	101	13.09M	26.41M
ocily7-1	70.38M	101	46.04M	54.45M
ocily7-2	70.89M	101	34.27M	39.21M
a375-1	115.32M	101	61.54M	79.81M
a375-2	115.27M	101	57.11M	106.96M
mason1	10M	150	103,432	384,139
mason2	10M	150	109,470	385,207
mason3	10M	150	898	3,431
mason4	10M	150	758	3,402
mason5	10M	150	1,005	3,407

Table 3 shows the total number of sequences, their length, and the occurrences on each reference for all the inputs. The number of occurrences is higher for GRCh38, as it contains around two million bases more than GRCh37, and all inputs generated using Mason employ GRCh38 as input. In addition, we note that the number of occurrences is higher for *mason1-2* than for *mason3-5* because the error introduced is lower, 3% and 6% respectively (see Table 2).

Our inputs are based on short reads because FM-Index methods dominate in this scenario. However, COFI would perform similarly for any read length, as the algorithm and data structures are read length agnostic, and the performance characteristics would not change.

For each experiment, we perform 128 executions. For each execution, we search all sequences on the target reference. To measure the throughput, we use the number of LF operations performed per second (LFOPs/s). We discard the first execution in order to avoid cold start effects of hardware structures. We calculate the arithmetic mean with the rest of the executions.

6 Performance Analysis

A common metric used to measure the theoretical peak performance a workload can exhibit on a particular target machine is the *arithmetic intensity* [24]. This metric defines the number operations, typically floating-point operations, performed per byte brought from off-chip memory. In our study, we will employ *search intensity (SI)* for this purpose. Defined as the number of LFs performed per byte brought from memory [16]. Equation 6 computes the SI of a search for a k-step FM-index:

$$SI = \frac{2 \times k}{\alpha \times B} \tag{6}$$

where $2 \times k$ is the number of LFs performed per iteration, α is the average number of cache misses per iteration, which depends on the algorithm, and B is the cache block size (typically of 64 bytes).

Backward search algorithms are typically memory bound, i.e., little computation is done per byte brought from memory. Therefore, increasing search intensity is paramount in order to increase performance. In the case of $bvSFM$, it has $k = 2$ and computes four LFs per iteration. We need to bring two cache blocks, one for sp and another one for ep. The ep pointer could produce a cache hit if it is close enough to sp. For this algorithm, the reported α is 1.088 [16]. Therefore, $bvSFM$ has a search intensity of $\frac{2\times2}{1.088\times64} = 0.057$.

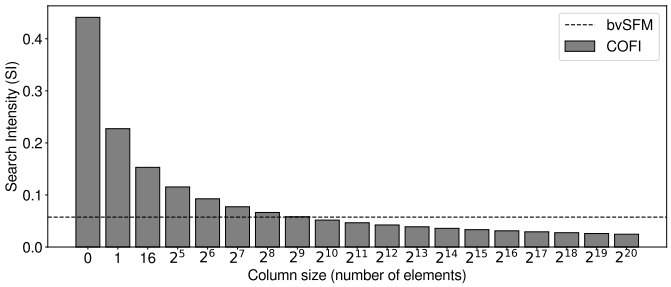


Fig. 10: Search intensity for different column sizes.

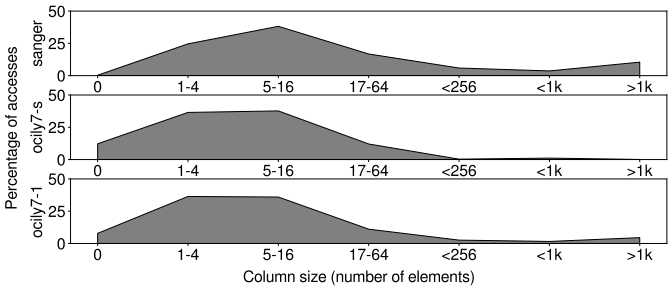


Fig. 11: Percentage of accesses per column size.

In COFI, SI is input dependent as iterations over large columns result in low SI values due to additional memory accesses. Contrarily, iterations over small columns result in high SI values. To calculate α, we have to take into account the number of memory accesses in $Offsets$ and $Changes$.

For $Offsets$, two 4-byte accesses are required, one for the left index and another one for the right index. Since both values are stored in contiguous memory positions, the probability of having a cache miss for the second element is $1/16$. Hence, an average of 1.0625 memory operations are performed due to the two $Offsets$ accesses. For $Changes$, the number of accesses depends on the number of 4-byte elements in the column, ne. When ne is 0 or 1, the number of $Changes$ accesses is 0 and 1, respectively. When ne is 16 or less, the worst case results in two cache misses, and each time we double the size of the column, one extra cache miss has to be added. Hence, when ne is larger than 16, the number of cache misses is $\lceil log_2(ne) - 2 \rceil$ for the worst case. Using $k = 15$, SI is $\frac{2\times15}{(1.0625+a)\times64}$, where a is the number of cache misses caused by the $Changes$ accesses, and it is defined in Equation 7.

$$a = \begin{cases} ne, & \text{if } ne = 0, 1 \\ 2, & \text{if } 2 \leq ne \leq 16 \\ \lceil log_2(ne) - 2 \rceil, & \text{otherwise} \end{cases} \tag{7}$$

Figure 10 shows the *bvSFM* SI (dotted black line) and COFI's SI (grey bars) for different column sizes. Note that the SI numbers for COFI represent worse case behaviour of the binary search algorithm, as the searched element could be found in one of the pivot points. We can see that

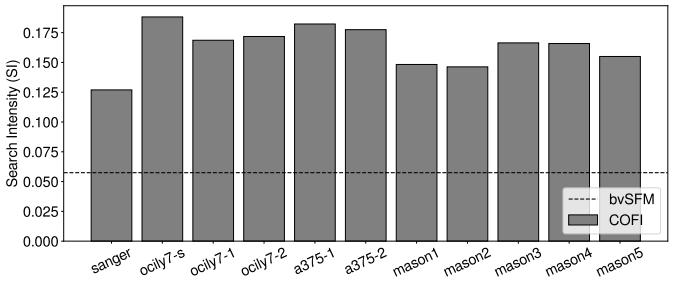
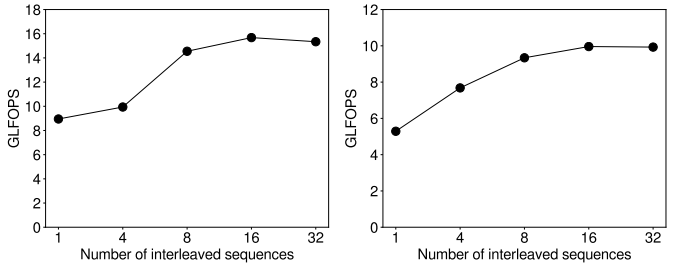
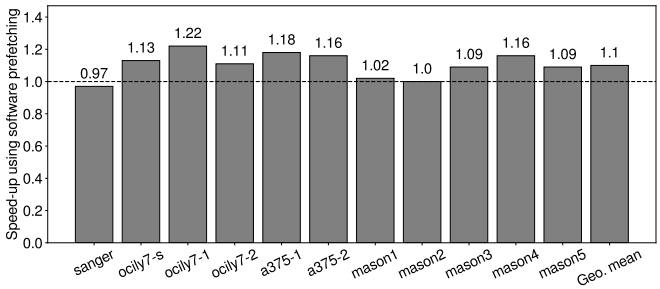


Fig. 12: Search intensity for each input.

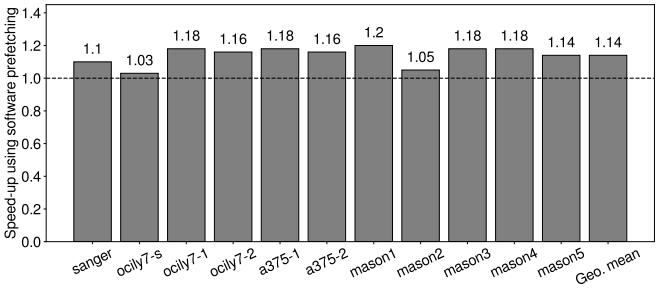


(a) KNL (b) SKX

Fig. 13: Average performance across all inputs for different numbers of interleaved sequences.



(a) KNL



(b) SKX

Fig. 14: Speed-up when using software prefetching.

COFI presents higher SI than *bvSFM* for values of ne lower than 1024 elements. This metric is directly correlated with performance.

In order to calculate SI for each input, we measure the size of the columns that the algorithm accesses during execution. We can see the percentage of accesses per column size in Figure 11 for the inputs *sanger*, *ocily7-s* and *ocily7-1* and the reference genome GRCh38. For *sanger*, we can observe that there are no accesses to empty columns, this is because *sanger* was extracted from GRCh38 without introducing errors. The percentage of accesses to columns bigger than 1024 elements is 10.54%, 0.1% and 4.55% for *sanger*, *ocily7-s* and *ocily7-1*, respectively.

By using the SI values from Figure 10 and the number of accesses per column type, as shown in Figure 11, we plot the calculated SI for each input in Figure 12. We can observe that the SI in COFI is significantly higher than that of *bvSFM*, up to 3.28× for *ocily7-s* input. This is likely to lead to better performance as the workload is memory bandwidth bound.

7 Evaluation

In this section, we present the results obtained with COFI. First, we evaluate the performance impact for each of the optimizations described in Section 4.5. Second, we compare COFI against a state-of-the-art proposal, *bv*SFM [16]. *bv*SFM is a *2-step 64 sampled FM-index* that uses a bitmap to recover the original Occ values (see Section 2.4). Third, we discuss how inputs affect performance by relating time spent in large columns and SI with performance. Finally, we describe other experiments we performed and our findings.

7.1 Optimizations

We evaluate the performance impact when applying each of the three optimizations described in Section 4.5. Therefore, we analyse the impact on the number of interleaved sequences, the benefits of the software prefetching scheme, and the impact of conditional moves in binary search. We use software prefetching and conditional moves in these experiments except when evaluating their own impact. For this set of results, we just show numbers with the GRCh38 reference due to space constrains. Results with GRCh37 are similar and lead to the same conclusions.

7.1.1 Number of Interleaved Sequences

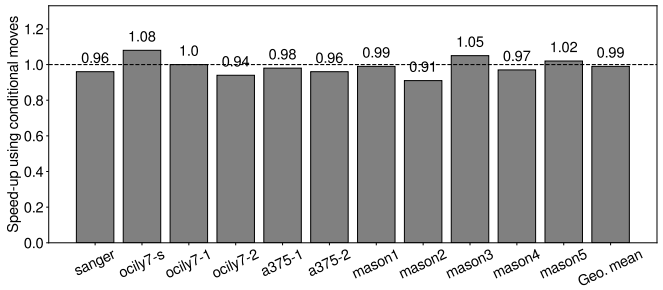
Figure 13 shows average performance across all inputs when changing the number of interleaved sequences from 1 to 32. In both test machines, KNL and SKX, we obtain the best average performance with 16 interleaved sequences. After 16 sequences, performance remains stable because the memory subsystem is already saturated. A few inputs behave slightly better with 8 or 32 interleaved sequences; however, to be consistent we employ 16 interleaved sequences for all inputs from now on.

7.1.2 Software Prefetching

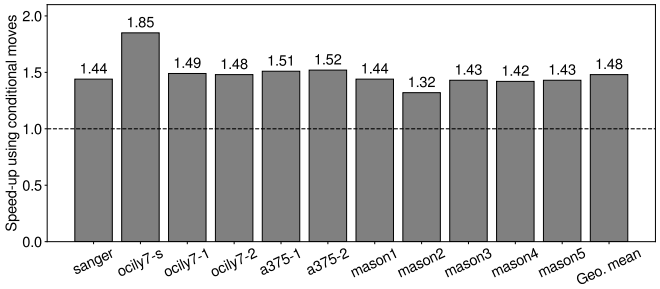
Figure 14 shows the speed-up when using the described software prefetching scheme. As previously specify, we are prefetching acceses to the $Offsets$ vector and the first pivot of the binary search, but not the remaining pivots. On average, we obtain a speed-up of 10% and 14% for KNL and SKX, respectively. Therefore, we consider this optimization useful and we will use it throughout the rest of the evaluation.

7.1.3 Conditional Moves

Figure 15 shows the speed-up of using conditional moves in the binary search algorithm. We can observe a significantly different impact on each of the test machines. For SKX, we obtain speed-ups of up to 1.85× (1.48× on average). This



(a) KNL



(b) SKX

Fig. 15: Speed-up when employing conditional moves in the binary search algorithm.

is because the Skylake core is heavily penalized by branch mispredictions due to its aggressive speculative execution on predicted branches, which in this case have a high probability of being mispredicted. These mispredictions are likely to perform additional memory accesses on the wrong execution path, further hindering performance. Prior work already showed that conditional moves are an effective way to speed-up binary search algorithms for the mentioned reasons [25].

For KNL, however, performance is not altered significantly, leading to a marginal 1% slowdown on average. Branch mispredictions do not impose such a large penalty in KNL due to a shallower pipeline and less aggressive speculative execution. Since this optimization fails to deliver performance improvements in KNL we chose not to enable it from now on in KNL experiments. It will be used only on SKX experiments.

7.2 Throughput

Figure 16 shows the speed-up of COFI with respect to the best performing *bv*SFM (*2-step 64-sampled*), for references GRCh37 and GRCh38. For KNL, we observe speed-ups of up to 1.81× (1.46× on average) when using the GRCh38 reference. Performance differences across inputs come from the different column access profiles in the *Changes* data structure (see Figure 11 for examples). The obtained results correlate well with these profiles. For example, *ocily7-s* presents the lowest amount of accesses to large columns and attains the best performance. On the other hand, *sanger* has more accesses to larger columns and COFI has the same performance as *bvSFM*. We present further details on input sensitivity in Section 7.3. The results for the GRCh37 reference show the same trends, with an average performance improvement of 1.5×.

TABLE 4: Raw throughput in GLFOPs/sec using GRCh38.

	KNL		SKX	
Input	***bv*SFM**	**COFI**	***bv*SFM**	**COFI**
sanger	10.26	10.25	5.65	6.73
ocily7-s	11.69	21.16	12.55	17.81
ocily7-1	11.25	17.45	7.58	9.59
ocily7-2	11.11	17.60	7.56	9.74
a375-1	11.46	18.93	7.97	10.39
a375-2	11.52	19.16	7.98	10.44
mason1	9.61	12.51	5.86	8.57
mason2	10.00	13.59	5.87	7.86
mason3	9.92	14.10	5.83	9.42
mason4	10.05	15.07	5.87	9.65
mason5	10.01	14.25	5.88	9.36
geo. mean	10.60	15.51	6.94	9.68

For SKX, we obtain speed-ups of up to 1.64× (1.39× on average) when using the GRCh38 reference. Here we also find similar trends for the different inpus. However, since the binary search component of the algorithm performs significantly better in SKX due to the conditional move optimization, we can see that performance differences between inputs with different column access profiles (i.e., *sanger* and *ocily7-s*) are much narrower. In fact, COFI is 1.19× faster than *bvSFM* with *sanger* on SKX. Again, similar trends can be seen for the CRCh37 reference, with an average improvement of 1.44×.

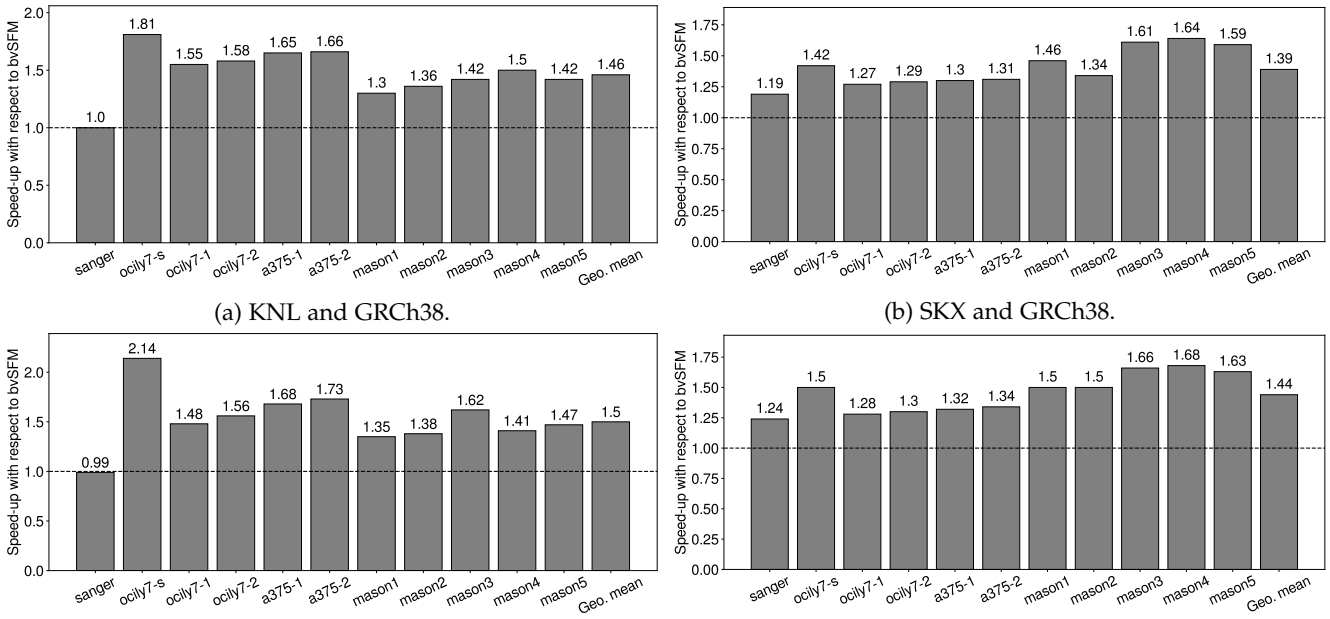
Table 4 shows the raw throughput measured in GLFOPs/sec for each input and the GRCh38 reference genome. In KNL, the *bvSFM* version maintains a constant performance among the inputs. However, in SKX, the *ocily7-s* input shows a much higher throughput than the others. This is due to two reasons. Firstly, *ocily7-s* contains a high number of repeated sequences. Secondly, SKX has larger last-level cache slices per core and it can exploit data locality for these repeated sequences.

7.3 Input Sensitivity

There is a noticeable difference on performance depending on the input. We have previously mentioned that performance is correlated with the SI associated to an input, and consequently, with the number of accesses for each column size.

In Figure 17, we can see the percentage of time spent on each column size for *sanger*, *ocily7-s* and *ocily7-1* inputs. In order to obtain representative results of the cost to perform a search for a given column size, we perform this experiment using 1 thread and 1 interleaved sequence on the KNL using the GRCh38 reference. Multiple interleaved sequences would slowdown searches over small columns when they interleave with searches over big columns.

As expected, the time spent searching on large columns is directly correlated to the performance differences seen across inputs. That is, inputs that spend more time on larger columns obtain lower performance. *sanger* is the input that spends more time in large columns, 37.5% of the time spent on columns larger than 1024 elements, while *ocily7-s* and *ocily7-1* spend 0.81% and 22.58%, respectively.



(a) KNL and GRCh38.

(b) SKX and GRCh38.

(c) KNL and GRCh37.

(d) SKX and GRCh37.

Fig. 16: Speed-up of COFI with respect to *bv*SFM for all inputs, two references, and two test machines.

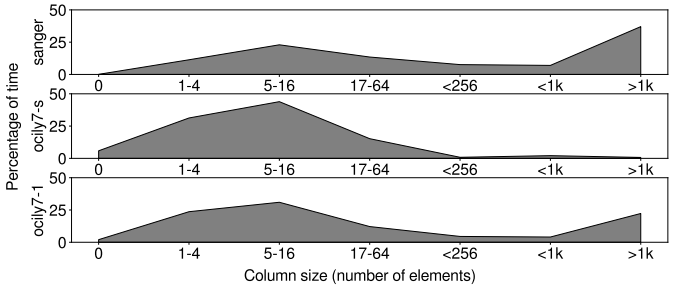
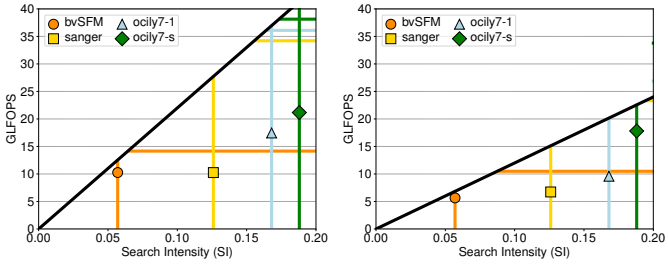


Fig. 17: Execution time spent on each column size.



(a) KNL roofline.

(b) SKX roofline.

Fig. 18: Roofline models for *bvSFM* (input independent) version and three inputs of COFI.

An important observation is that some exact search algorithms avoid or even filter-out large columns [26]. The reason is that these columns contain sequences that appear a lot of times, and they provide redundant locations that other columns already find. It is also the reason our inputs based on real read data from cell lines (e.g., *ocily7-s* and *ocily7-1*) access less large columns in percentage than *sanger*, which is produced using a read simulator over the entire genome reference. Therefore, COFI could further benefit from this fact if columns with small SI are filtered-out.

Finally, to illustrate how higher SI improves performance in COFI, we show the roofline models [24] for both test machines in Figure 18. Our roofline model ties together throughput, SI, and memory performance in a two-dimensional graph. The Y-axis is GLFOPS per second (throughput) and the X-axis is SI, i.e., LF per byte of off-chip memory traffic. Theoretical ceilings can be derived using hardware specifications. The diagonal black line (memory ceiling) depends on memory bandwidth available and determines the maximum performance achievable for a given SI value. Horizontal lines denote compute ceilings for each input, calculated by dividing the number of executed instructions to perform all LF operations and the number of instructions each machine can retire per cycle. Note that the compute ceilings are input dependent.

We can observe that *bvSFM* performs close to the memory ceiling in both machines, and it is therefore memory bound rather than compute bound. However, COFI manages to increase SI significantly, and for the *ocily7-s* input we are able to break the computational ceiling of *bvSFM* on both machines.

Even though SI has increased, the achieved performance with COFI is far from the theoretical compute ceilings. Therefore, we can conclude that the main performance limitation has now shifted from memory bandwidth to the algorithm itself, due to branches and data dependencies in the code.

7.4 Discussion

We have tried to apply other ideas to improve performance even further. Among these ideas, we highlight two: (i) to rearrange the columns of $Changes$ in order to be cache-friendly, and (ii) the use of larger *k-steps* using SKX.

When performing binary search we can imagine the columns of $Changes$ as trees. When the column is large, cache misses arise when traversing the first levels of the tree. In order to avoid this problem, we thought of two approaches. Firstly, to place the children together as in

the Eytzinger layout [25]. This would allow us to apply software prefetching to several levels in advance. The main drawbacks that made this implementation not feasible are irregular column sizes, and that the algorithm to recover the original index becomes too costly. Secondly, to group several levels of the tree together [27]. We place n levels of the tree at the center of the column, and perform binary search (in this case n-ary search) for n pivots knowing that they are close in memory. Then, we have to search into one of the 2^n subtrees applying recursion. We have a threshold from which we do not rearrange the column, at which point we use the normal binary search algorithm. The problem is that with this layout we can not limit the end pointer search, because the columns are not sorted in ascending order. Therefore, we have to repeat the binary search for the whole column for ep, precluding any performance improvements.

We also did experiments using $k = 16$ and $k = 17$ for SKX, we already fill up completely the MCDRAM of the KNL with $k = 15$. In both versions, we obtain a small slowdown due to the size of the FM-index causing a large amount of TLB misses, as the amount TLB entries for huge pages is exceeded.

8 Related work

There are many software libraries available that perform sequence alignment. Some examples targeting conventional computing infrastructures include HITSAT [28], BWA [5, 6, 7], Bowtie [4] or SOAP [8, 29]. Additionally, in order to cope with the increasing demand to post-process read data, significant efforts have been devoted to efficiently exploit other architectures. For GPGPUs, examples are CUSHAW2 [30], Arioc [31], BarraCUDA [32], SARUMAN [33], and NVIDIA. The latter includes an FM-index implementation in its bioinformatics library NVBIO [34]; for clusters, CUSHAW3 [35]; and for cloud computing, BigBWA [36]. All of these libraries make extensive use of exact search algorithms and would benefit from COFI.

With a focus on exact search, Xin et al. [37] propose an algorithm to find the best way to select the seeds from the sequences. They also propose FastHASH [26], an algorithm that applies several techniques using hash tables by filtering seeds with occurrences above a threshold, i.e., filtering the equivalent of large columns in COFI. Optimizations to sample structures and increase k have also been evaluated using GPGPUs [14, 38, 39]. In addition, multiple accelerators based on FPGAs have been proposed, such as Gatekeeper [23], Shouji [40], Darwin [41], FHAST [42], a proposal by Olson et al. [43], and GenCache [2].

With a focus on FM-index optimizations, Chacón et al. also propose to extend the alphabet to have a k-step algorithm [15], and a way to compress even further a sampled FM-Index for GPU execution [14]. As in bvSFM, the employed data structures become too large for $k > 4$, as these techniques do not solve the problem of exponential memory footprint growth. In addition, Chacón et al. also propose a cooperative scheme for GPU threads that enables good performance scaling on different GPU architectures, and that can be used to reduce the index size with negligible impact on performance. GPU architectures cannot efficiently hide random access memory latencies, which penalizes performance significantly. However, both KNL and SKX have out-of-order capabilities that help hide such latencies, especially when employing the sequence interleaving optimization.

Wavelet trees [44] have also been used to navigate the BWT structure. Prior work [16] evaluates wavelet tree implementations using the Succinct Data Structure Library (SDSL) [45]. In spite of requiring less memory, its performance results cannot match those obtained by our baseline and COFI. For example, when using $k = 1$ a wavelet tree would have two levels, requiring two memory accesses to traverse it plus one access for the base counter. For $k = 2$ the wavelet tree would have four levels, requiring four plus one accesses. As a comparison point, our baseline needs two accesses, one for the counter and one for the bitmap, and both are in the same cache block. Therefore, while wavelet trees are memory efficient, their performance is several times worse than *bvSFM* and COFI.

Hash-based indexing methods are also extensively used, especially when dealing with long reads obtained by Oxford Nanopore and PacBio machines. A single read can contain hundreds of thousands of bases. Hashing has been present in sequence alignment tools for decades, for example in BLAST [46] [47]. A more recent proposal, minimap2 [48], also employs hashing to find seeds for long reads. A number of proposals have proposed improvements over hashing methods. Ma et al. [49] proposed to use non-consecutive matches in order to increase the sensitivity. Lin et al. [50] proposed an optimal way to choose the minimum number of seeds given read length, sensitivity and memory usage. Finally, Homer et al. [51] proposed a two-level hash table in order to reduce the memory usage.

Finally, there have been efforts to improve seed-and-extend algorithms based on Smith-Waterman [52], which is the other time consuming part of sequence alignment. Gotoh proposed improvements over the original algorithm [53], and others focused on exploiting available SIMD hardware [54, 55, 56]. Also, a recent work by Park et al. [57] uses the new vector extensions from Arm (SVE) to vectorize the Smith-Waterman algorithm.

9 Conclusions

This paper analyses the memory footprint of state-of-the-art FM-index data structures and makes the key observation that the amount of useful data stored remains constant as the k-step size increases. Based on this insight, we propose COFI, a **CO**mpressed **F**M-**I**ndex for large K-steps. Contrary to prior proposals, COFI's main data structure has constant size with respect to the value of k. This enables large k-step searches that present better trade-offs in terms of throughput per unit of moved data.

We evaluate COFI on two different high performance hardware platforms, using two reference genomes and a representative set of inputs. We show that COFI consistently outperforms a state-of-the-art proposal with improvements of up to 2.14×. On average, COFI obtains 1.46× and 1.39× improvements on KNL and Skylake-based systems for the GRCh38 reference, respectively.

REFERENCES

[1] D. Fujiki, A. Subramaniyan, T. Zhang, Y. Zeng, R. Das, D. Blaauw, and S. Narayanasamy, "Genax: a genome sequencing accelerator," in *Proceedings of the 45th Annual International Symposium on Computer Architecture*. IEEE Press, 2018, pp. 69–82.

[2] A. Nag, C. Ramachandra, R. Balasubramonian, R. Stutsman, E. Giacomin, H. Kambalasubramanyam, and P.-E. Gaillardon, "Gencache: Leveraging in-cache operators for efficient sequence alignment," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 2019, pp. 334–346.

[3] T. Smith and M. Waterman, "Identification of common molecular subsequences," *Journal of Molecular Biology*, vol. 147, no. 1, pp. 195 – 197, 1981.

[4] 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, no. 3, p. R25, 2009.

[5] H. Li and R. Durbin, "Fast and accurate short read alignment with burrows–wheeler transform," *bioinformatics*, vol. 25, no. 14, pp. 1754–1760, 2009.

[6] ——, "Fast and accurate long-read alignment with burrows–wheeler transform," *Bioinformatics*, vol. 26, no. 5, pp. 589–595, 2010.

[7] M. Vasimuddin, S. Misra, H. Li, and S. Aluru, "Efficient architecture-aware acceleration of bwa-mem for multicore systems," in *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2019, pp. 314–324.

[8] 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.

[9] P. Ferragina and G. Manzini, "Opportunistic data structures with applications," in *Proceedings 41st Annual Symposium on Foundations of Computer Science*. IEEE, 2000, pp. 390–398.

[10] "Genome reference consortium human reference 37," http://hgdownload.cse.ucsc.edu/goldenPath/hg19/bigZips/, accessed: 2019-07-23.

[11] "Genome reference consortium human reference 38," http://hgdownload.cse.ucsc.edu/goldenPath/hg38/bigZips/, accessed: 2019-07-23.

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

[13] U. Manber and G. Myers, "Suffix arrays: a new method for on-line string searches," *siam Journal on Computing*, vol. 22, no. 5, pp. 935–948, 1993.

[14] A. Chacón, S. Marco-Sola, A. Espinosa, P. Ribeca, and J. C. Moure, "Fm-index on gpu: a cooperative scheme to reduce memory footprint," in *2014 IEEE International Symposium on Parallel and Distributed Processing with Applications*. IEEE, 2014, pp. 1–9.

[15] A. Chacón, J. C. Moure, A. Espinosa, and P. Hernández, "n-step fm-index for faster pattern matching," *Procedia Computer Science*, vol. 18, pp. 70–79, 2013.

[16] J. M. Herruzo, S. G. Navarro, P. Ibáñez, V. Viñals-Yufera, J. Alastruey, and O. Plata, "Accelerating sequence alignments based on fm-index using the intel knl processor," *IEEE/ACM transactions on computational biology and bioinformatics*, 2018.

[17] A. Sodani, R. Gramunt, J. Corbal, H. Kim, K. Vinod, S. Chinthamani, S. Hutsell, R. Agarwal, and Y. Liu, "Knights landing: Second-generation intel xeon phi product," vol. 36, no. 2, pp. 34–46, Mar 2016.

[18] "Mason simulator," http://www.seqan.de/apps/mason/, accessed: 2019-07-23.

[19] 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.

[20] "Homo sapiens oci-ly7," https://www.encodeproject.org/experiments/ENCSR740DKM/, accessed: 2019-07-23.

[21] "Homo sapiens oci-ly7," https://www.encodeproject.org/experiments/ENCSR001HHK/, accessed: 2019-07-23.

[22] "Homo sapiens a375," https://www.encodeproject.org/experiments/ENCSR504VXC/, accessed: 2019-07-23.

[23] M. Alser, H. Hassan, H. Xin, O. Ergin, O. Mutlu, and C. Alkan, "Gatekeeper: a new hardware architecture for accelerating pre-alignment in dna short read mapping," *Bioinformatics*, vol. 33, no. 21, pp. 3355–3363, 2017.

[24] S. Williams, A. Waterman, and D. Patterson, "Roofline: An insightful visual performance model for multicore architectures," *Commun. ACM*, vol. 52, no. 4, pp. 65–76, Apr. 2009.

[25] P.-V. Khuong and P. Morin, "Array layouts for comparison-based searching," *J. Exp. Algorithmics*, vol. 22, pp. 1.3:1–1.3:39, May 2017.

[26] H. Xin, D. Lee, F. Hormozdiari, S. Yedkar, O. Mutlu, and C. Alkan, "Accelerating read mapping with fasthash," in *BMC genomics*, vol. 14, no. 1. BioMed Central, 2013, p. S13.

[27] T. M. Chilimbi, M. D. Hill, and J. R. Larus, "Cache-conscious structure layout," in *ACM SIGPLAN Notices*, vol. 34, no. 5. ACM, 1999, pp. 1–12.

[28] D. Kim, B. Langmead, and S. L. Salzberg, "Hisat: a fast spliced aligner with low memory requirements," *Nature methods*, vol. 12, no. 4, p. 357, 2015.

[29] 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.

[30] Y. Liu and B. Schmidt, "Cushaw2-gpu: empowering faster gapped short-read alignment using gpu computing," *IEEE Design & Test*, vol. 31, no. 1, pp. 31–39, 2013.

[31] R. Wilton, T. Budavari, B. Langmead, S. J. Wheelan, S. L. Salzberg, and A. S. Szalay, "Arioc: high-throughput read alignment with gpu-accelerated exploration of the seed-and-extend search space," *PeerJ*, vol. 3, p. e808, 2015.

[32] P. Klus, S. Lam, D. Lyberg, M. S. Cheung, G. Pullan, I. McFarlane, G. S. Yeo, and B. Y. Lam, "Barracuda-a fast short read sequence aligner using graphics processing units," *BMC research notes*, vol. 5, no. 1, p. 27, 2012.

[33] J. Blom, T. Jakobi, D. Doppmeier, S. Jaenicke, J. Kalinowski, J. Stoye, and A. Goesmann, "Exact and complete short-read alignment to microbial genomes using graphics processing unit programming," *Bioinformatics*, vol. 27, no. 10, pp. 1351–1358, 2011.

[34] "Nvbio," http://nvlabs.github.io/nvbio/, accessed: 2019-08-08.

[35] J. Gonzalez-Dominguez, Y. Liu, and B. Schmidt, "Parallel and scalable short-read alignment on multi-core clusters using upc++," *PloS one*, vol. 11, no. 1, p. e0145490, 2016.

[36] J. M. Abuín, J. C. Pichel, T. F. Pena, and J. Amigo, "Bigbwa: approaching the burrows–wheeler aligner to big data technologies," *Bioinformatics*, vol. 31, no. 24, pp. 4003–4005, 2015.

[37] H. Xin, S. Nahar, R. Zhu, J. Emmons, G. Pekhimenko, C. Kingsford, C. Alkan, and O. Mutlu, "Optimal seed solver: optimizing seed selection in read mapping," *Bioinformatics*, vol. 32, no. 11, pp. 1632–1642, 2015.

[38] A. Chacón, S. Marco-Sola, A. M. Espinosa Morales, P. Ribeca, and J. C. Moure López, "Boosting the fm-index on the gpu: effective techniques to mitigate random memory access," *IEEE/ACM Transactions on computational biology and bioinformatics*, vol. 11, no. 6, 2015.

[39] S. Chen and H. Jiang, "An exact matching approach for high throughput sequencing based on bwt and gpus," in *2011 14th IEEE International Conference on Computational Science and Engineering*. IEEE, 2011, pp. 173–180.

[40] M. Alser, H. Hassan, A. Kumar, O. Mutlu, and C. Alkan, "Shouji: a fast and efficient pre-alignment filter for sequence alignment," *Bioinformatics*, 2019.

[41] Y. Turakhia, G. Bejerano, and W. J. Dally, "Darwin: A genomics co-processor provides up to 15,000 x acceleration on long read assembly," in *ACM SIGPLAN Notices*, vol. 53, no. 2. ACM, 2018, pp. 199–213.

[42] 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*, vol. 12, no. 5, pp. 973–981, 2015.

[43] 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*. IEEE, 2012, pp. 161–168.

[44] R. Grossi, A. Gupta, and J. S. Vitter, "High-order entropy-compressed text indexes," in *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, ser. SODA '03, 2003, p. 841–850.

[45] 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)*, 2014, pp. 326–337.

[46] 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.

[47] S. F. Altschul, T. L. Madden, A. A. Schäffer, J. Zhang, Z. Zhang, W. Miller, and D. J. Lipman, "Gapped blast and psi-blast: a new

generation of protein database search programs," *Nucleic acids research*, vol. 25, no. 17, pp. 3389–3402, 1997.
[48] H. Li, "Minimap2: pairwise alignment for nucleotide sequences," *Bioinformatics*, vol. 34, no. 18, pp. 3094–3100, 2018.
[49] B. Ma, J. Tromp, and M. Li, "Patternhunter: faster and more sensitive homology search," *Bioinformatics*, vol. 18, no. 3, pp. 440–445, 2002.
[50] H. Lin, Z. Zhang, M. Q. Zhang, B. Ma, and M. Li, "Zoom! zillions of oligos mapped," *Bioinformatics*, vol. 24, no. 21, pp. 2431–2437, 2008.
[51] N. Homer, B. Merriman, and S. F. Nelson, "Bfast: an alignment tool for large scale genome resequencing," *PloS one*, vol. 4, no. 11, p. e7767, 2009.
[52] M. S. Waterman, T. F. Smith, and W. A. Beyer, "Some biological sequence metrics," *Advances in Mathematics*, vol. 20, no. 3, pp. 367–387, 1976.
[53] O. Gotoh, "An improved algorithm for matching biological sequences," *Journal of molecular biology*, vol. 162, no. 3, pp. 705–708, 1982.
[54] A. Wozniak, "Using video-oriented instructions to speed up sequence comparison," *Bioinformatics*, vol. 13, no. 2, pp. 145–150, 1997.
[55] T. Rognes and E. Seeberg, "Six-fold speed-up of smith–waterman sequence database searches using parallel processing on common microprocessors," *Bioinformatics*, vol. 16, no. 8, pp. 699–706, 2000.
[56] M. Farrar, "Striped smith–waterman speeds database searches six times over other simd implementations," *Bioinformatics*, vol. 23, no. 2, pp. 156–161, 2006.
[57] D.-H. Park, J. Beaumont, and T. Mudge, "Accelerating smith-waterman alignment workload with scalable vector computing," in *2017 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2017, pp. 661–668.

Rubén Langarita received the BS degree in computer science from Universidad de Zaragoza in 2018. He spent one academic year at the University College Cork as an Erasmus student. His final degree project was about optimizing molecular dynamics applications. He is currently doing his MS degree at UPC while working at the BSC as a research student. His research interests include processor microarchitecture and HPC applications.

Adrià Armejach is a researcher at Universitat Politècnica de Catalunya (UPC) and an associate researcher at the Barcelona Supercomputing Center (BSC). His research interests include computer architecture, parallel computing, memory systems, and performance evaluation. He received M.Sc. and Ph.D. degrees from UPC in 2009 and 2014, respectively. He has lead the technical contributions of multiple FP7 and H2020 projects.

Javier Setoain obtained his PhD. in Computer Science from Universidad Complutense de Madrid working on high performance computing applications in GPGPU. As a post-doc, he worked on bioinformatics in the National Centre for Biotechnology, focusing on genomics and transcriptomics for drug repositioning. After that, he worked on memory hierarchy architectures for genomics applications, as part of a collaboration between UCM and Imec. For the last few years, Javier Setoain has been part of Arm Research, as a Research Engineer in the Software and Large Scale Systems Group.

Pablo Ibáñez-Marín received the MS degree in computer science from the UPC in 1989, and the PhD degree in computer science from Universidad de Zaragoza in 1998. He is an associate professor in the Computer Science and Systems Engineering Dept. at the University of Zaragoza, Spain. His research interests include processor microarchitecture, memory hierarchy, parallel computer architecture, and HPC applications.

Jesús Alastruey-Benedé received the MS degree in Telecommunication and the PhD degree in Computer Science from Universidad of Zaragoza in 1997 and 2009, respectively. He is a professor in the Computer Science and Systems Engineering Department (DIIS), University of Zaragoza, Spain. His research interests include processor microarchitecture, memory hierarchy, and High Performance Computing (HPC) applications.

Miquel Moretó Miquel Moreto is a Ramon y Cajal at UPC Barcelona. Prior to joining UPC, he spent 5 years as a senior researcher at the Barcelona Supercomputing Center (BSC), and 15 months as a post-doctoral fellow at the International Computer Science Institute (ICSI), Berkeley, USA. He received the B.Sc., M.Sc., and Ph.D. degrees from UPC. His research interests include studying shared resources in multithreaded architectures and hardware-software co-design for future massively parallel systems.