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 show an existing throughput-memory trade-offs, showing that memory requirements limit implementation of large k-steps. We propose COFI, a Compressed FM-Index 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].

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 Compressed FM-Index 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
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

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 “AGATGCCAGGGCAT”. 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 a 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 \(\sigma\) is the number of symbols in the 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 \times\ and 1.39 \times in KNL and SKX, respectively.

![Figure 1](image1.png)

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.

![Figure 2](image2.png)

Fig. 2: Backward search example. (a) \(sp\) and \(ep\) initialization. (b) LF operation for \(sp\) and \(ep\). (c) Final state of the query.
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 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 \(T\)s until the 6th 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 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].

\[
sp = C[s] \quad \text{(1)}
\]

\[
ep = C[s + 1] \quad \text{(2)}
\]

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

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(d) 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.

\[
Occ[p, s] = rOcc[p/d, s] + occur(s, BWT[(p - p \mod d) .. p]) \quad \text{(4)}
\]

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
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

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

For \( \text{bvSFM} \), the \( 4^k \) bitmaps in a \( \text{bvSFMentry} \) have a constant number of set bits, equal to the sampling factor \( d \) (length of each bitmap). That is, if \( \text{BWT}[i] \) 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 \( \text{useful bits} \) 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 \( \text{bvSFMentry} \) contains 64 Mb \((64 * 4^{10} \text{ bits})\), of which only 64 are set.

Figure 5 shows how the size of a sampled \((d = 64)\) \( \text{bvSFM} \) structure increases with \( k \) (black line, right \( y \)-axis), quickly becoming intractable. Stacked bars show the percentage of the \( \text{bvSFM} \) size dedicated to counters (white background) and bitmaps (gray background). Moreover, we also show the percentage of the \( \text{bvSFM} \) size allocated to sampled counter to the result of the \( \text{popcount} \) operations: \( \text{rOcc}[p/d,s] + \text{popcount}(\text{masked bit vector}) \).

\[ 9 + \text{popcount}(100_2) = 9 + 1 = 10 \] for \( sp \), and
\[ 11 + \text{popcount}(12_2) = 11 + 1 = 12 \] for \( ep \).

The values of \( sp \) and \( ep \) will be 10 and 12, respectively.
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 structures 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| - 1]
2. ep = Offsets[|Q|]
3. for i = length - 2 to 0 do
   4. left = Offsets[i]
   5. right = Offsets[i + 1]
   6. sp = find_col_index(Changes, left, right, sp)
   7. ep = find_col_index(Changes, left, right, ep)
4. 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 \textit{find\_col\_index} function in lines 6 and 7 returns the index of the found element in Changes. In the case of sp, this 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 \textit{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:

- \textit{array}: array to search in.
- \textit{left}: left limit of the binary search.
- \textit{right}: right limit of the binary search.
- \textit{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:

\[
ep = \text{find}\_\text{col}\_\text{index}(\text{Changes}, sp, sp + d, ep) \tag{5}
\]

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 \textit{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:

- 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 \textit{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).

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 bsSFM [16], since all evaluated algorithms do not benefit from them due to random memory access patterns. For all the evaluated proposals, disabling hardware prefetchers yields 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.

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 bsSFM [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>
<thead>
<tr>
<th>TABLE 1: Test machines configuration.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel Xeon Phi</td>
</tr>
<tr>
<td>7230 (KNL)</td>
</tr>
<tr>
<td>Cores x Threads</td>
</tr>
<tr>
<td>Issue width</td>
</tr>
<tr>
<td>Frequency (GHz)</td>
</tr>
<tr>
<td>Last-level Cache (MB)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TABLE 2: Error rates for Mason inputs.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inputs</td>
</tr>
<tr>
<td>---------------------------------------------------------------</td>
</tr>
<tr>
<td>mason1</td>
</tr>
<tr>
<td>mason2</td>
</tr>
<tr>
<td>mason3</td>
</tr>
<tr>
<td>mason4</td>
</tr>
<tr>
<td>mason5</td>
</tr>
</tbody>
</table>
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:

\[
SI = \frac{2 \times k}{\alpha \times B}
\]

where \(2 \times k\) is the number of LFs performed per iteration, \(\alpha\) 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 \(\alpha\) is 1.088 [16]. Therefore, \(bvSFM\) has a search intensity of

\[
SI = \frac{2 \times 2}{1.088} \times 64
\]

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 \(\alpha\), 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 \(\left\lfloor \log_2(ne) - 2 \right\rfloor\) for the worst case. Using \(k = 15\), \(SI = \frac{2 \times 2}{(1.0625+a) \times 64}\) where \(a\) is the number of cache misses caused by the \(Changes\) accesses, and it is defined in Equation 7:

\[
a = \begin{cases} ne, & \text{if } ne = 0, 1 \\ 2, & \text{if } 2 \leq ne \leq 16 \\ \left\lfloor \log_2(ne) - 2 \right\rfloor, & \text{otherwise} \end{cases}
\]

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
COFI presents higher SI than bvSFM for values of $n_e$ 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, bvSFM [16]. bvSFM 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 accesses 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).
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 \textit{bvSFM} (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 \textit{Changes} data structure (see Figure 11 for examples). The obtained results correlate well with these profiles. For example, \textit{ocily7-s} presents the lowest amount of accesses to large columns and attains the best performance. On the other hand, \textit{sanger} has more accesses to larger columns and COFI has the same performance as \textit{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×.

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 inputs. 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., \textit{sanger} and \textit{ocily7-s}) are much narrower. In fact, COFI is 1.19× faster than \textit{bvSFM} with \textit{sanger} on SKX. Again, similar trends can be seen for the GRCh37 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 \textit{bvSFM} version maintains a constant performance among the inputs. However, in SKX, the \textit{ocily7-s} input shows a much higher throughput than the others. This is due to two reasons. Firstly, \textit{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 \textit{sanger}, \textit{ocily7-s} and \textit{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. \textit{sanger} is the input that spends more time in large columns, 37.5% of the time spent on columns larger than 1024 elements, while \textit{ocily7-s} and \textit{ocily7-1} spend 0.81% and 22.58%, respectively.
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 \( cp \), 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 HITSA [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 Compressed FM-Index 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 \( 2.14 \times \). On average, COFI obtains 1.46% and 1.39% improvements on KNL and Skylake-based systems for the GRCh38 reference, respectively.


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, parallel computer architecture, and 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.