Accelerating Sequence Alignments Based on FM-Index Using the Intel KNL Processor

Jose M. Herruzo, Sonia González-Navarro, Pablo Ibáñez, Víctor Viñals, Jesús Alastruey and Oscar Plata

Abstract—FM-index is a compact data structure suitable for fast matches of short reads to large reference genomes. The matching algorithm using this index exhibits irregular memory access patterns that cause frequent cache misses, resulting in a memory bound problem. This paper analyzes different FM-index versions presented in the literature, focusing on those computing aspects related to the data access. As a result of the analysis, we propose a new organization of FM-index that minimizes the demand for memory bandwidth, allowing a great improvement of performance on processors with high-bandwidth memory, such as the second-generation Intel Xeon Phi (Knights Landing, or KNL), integrating ultra high-bandwidth stacked memory technology. As the roofline model shows, our implementation reaches 95% of the peak random access bandwidth limit when executed on the KNL and almost all the available bandwidth when executed on other Intel Xeon architectures with conventional DDR memory. In addition, the obtained throughput in KNL is much higher than the results reported for GPUs in the literature.

Index Terms—FM-index, short-read alignment, high-bandwidth memory, Intel Xeon Phi Knights-Landing

1 Introduction

The high demand for fast and low-cost genomic sequencing has pushed onward the rapid development of next-generation sequencing (NGS) technologies. As a result, a number of high-throughput sequencing systems have appeared in industry, including those from Illumina, Roche 454, Life Technologies and Pacific Biosciences. These systems are able to produce huge amounts of short reads (in the order of giga base-pairs) per day of operation. For instance, the Illumina NovaSeq 6000 sequencing system is able to produce up to 20 billion reads of 150 base pairs (bp) in less than two days. This represents up to 6 terabits of data which have to be processed as fast as possible.

Usually the first step in NGS corresponds to sequence alignment, where sequence reads must be aligned or compared to a genomic reference to identify regions of similarity [1]. Most popular alignment methods are based on two types of index structures: suffix trees and variants, and hash tables.

When dealing with large reference genomes, great efforts were devoted to reduce memory requirements for sequence alignment. As a result, a set of alignment algorithms based on the FM-index structure have been developed [2]. The FM-index uses Burrows-Wheeler transform (BWT), a method for rearranging a character string that is useful for data compression [3]. FM-index is well suited for fast exact matches of short reads to large reference genomes while keeping a small memory footprint. Many efficient sequence aligners are based on FM-index, such as Bowtie [4], BWA [5] (BWA-SW [6] for long reads), SOAP2 [7] and BWT-SW [8].

- *J.M. Herruzo, S. González, and O. Plata are with Department of Computer Architecture, University of Málaga, Andalucía Tech, 29071 Málaga, Spain. E-mail: {jmherruzo, sgn, oplata}@uma.es*
- *P. Ibañez, V. Viñals, and J. Alastruey are with Departamento de Informática e Ingeniería de Sistemas, University of Zaragoza, María de Luna 1, 50018 Zaragoza, Spain. E-mail: {imarin,victor,jalastru}@unizar.es*

Manuscript received MMMM DD, YYYY; revised MMMM DD, YYYY.

As high-throughput sequencing systems produce a massive amount of data, the usage of high-performance technologies is of crucial importance to deal with the computational challenge. In fact, many optimized exact matching algorithms have appeared recently in the literature for different architectures, like CPU clusters, GPUs and FPGAs [9].

Due to the data structure layout, the searching process using FM-index exhibits irregular memory access patterns. In addition, sequence aligners based on that index include support for inexact matching built on exact alignments, that causes the memory pattern to be even less predictable. These data access patterns cause a high cache miss rate on typical cache hierarchies of multicore processors. Besides, it is common for the exact matching algorithm to be memory bound due to the low arithmetic intensity (ratio of the computation to the memory traffic). Each step of the algorithm accesses a section of the index that it is not known in advance, making the cache hierarchy difficult to exploit.

This paper analyzes the exact matching based on the FM-index data structure. The study focuses on bottlenecks related to data access patterns and computational capabilities. The evaluation assumes a batch or offline setting, where a bulk of queries is issued to be processed as fast as possible. Our main contributions can be summarized as follows:

- Exact matching algorithms built upon FM-index are analyzed, focusing on those aspects related to computing costs and access to data, which have a great impact on performance.
- A new organization of the FM-index data structure layout and codification is proposed, which reduces the required traffic between memory and processor cores for the exact search process.
- An optimized exact matching algorithm has been implemented based on the proposed FM-index, exploiting the ultra high-bandwidth memory modules integrated in the KNL processor.

We have evaluated our proposal on a system with an Intel Xeon Phi 7210 processor [10], [11] (codenamed Knights Landing, or KNL) that includes 64 cores and 16 GiB of stacked 3D MCDRAM integrated on package. This memory offers a much higher memory bandwidth than the standard out-of-package DDR4 DRAM (∼400 vs. ∼ 90 GB/s peak). Our results show that performance on KNL reaches 95% of the peak random access bandwidth limit, outperforming other CPU and GPU solutions reported in the literature.

2 Background

2.1 FM-index

The FM-index is a data structure that allows fast substring searches over large texts [2]. In the following subsections we show some basic definitions concerning the FM-index.

In the rest of the paper, brackets are used to specify an entry in an array (e.g., $A[k]$ is k-th entry of the array A), and a character or a substring in a string (e.g., $A[k]$ is the character at position k in the string A, and $A[k..r]$ is the substring from position k to position r in A).

2.1.1 Suffix Array

Let $T[1..n]$ be a character string drawn from an alphabet Σ having σ symbols. The suffix array $SA[1, ..., n]$ of T is an array containing the starting positions of all suffixes of T in lexicographical order [12]. The suffix array can be used as an index to locate every occurrence of a pattern.

2.1.2 Burrows-Wheeler Transform (BWT)

The BWT is a permutation of a character string. Originally it was used in text compression algorithms, but it has other applications such as large text indexing.

The $BWT[1..n{+}1]$ of a n-character string T is another string obtained as follows: (1) append to the end of the original text T the symbol \$, which is lexicographically smaller than any symbol in Σ; (2) form a conceptual $(n{+}1){\times}(n{+}1)$ matrix M whose rows are the cyclic shifts of $T\$$ sorted in lexicographical order; (3) the last column of the matrix M is the BWT of T, denoted by L.

The suffix array of $T\$$ contains the starting positions in T of every row of the matrix M.

2.1.3 FM-index data structure

The FM-index is composed of two data structures derived from L (BWT of T): the C array and the Occ matrix. The C array stores in $C[c]$ the number of occurrences in L of the symbols lexicographical smaller than c. $Occ[c, i]$ contains the number of occurrences of the symbol c in the prefix $L[1..i]$, being $1{\leq}i{\leq}n{+}1$.

The FM-index was designed as a compressed structure such that the index size can be smaller than the original text. However, in the context of sequence alignment, it is usually not compressed in order to achieve better performance [1].

2.2 Exact Matching Using FM-Index

Given a pattern $Q[1..p]$, the FM-index allows to find all occurrences of Q in the text T [2]. The search process takes two steps: *Count* and *Locate*. The first step is a rank query process that calculates the number of occurrences of Q in T by identifying the first and last rows of matrix M (see sec. 2.1.2) prefixed by the query Q. The second step uses the indexes of these rows to access the suffix array, where it finds the position of every occurrence of Q in the text T.

The suffix array is usually a very large compressed data structure. However, its size for the human genome is about 12 GB (3 gigabases x 4 bytes), so it can be stored without compression in modern systems. This way, the *Locate* step is very simple, it only requires an access to the suffix array. For this reason, this paper focuses in the *Count* step.

2.3 Rank Query Implementations

To speed-up the *Count* process, FM-index uses the Occ matrix as a look-up table [2]. The main drawback of this solution is the large memory footprint of Occ. It is a matrix with σ rows and n+1 columns. Hence, its footprint is:

$$Fp(Occ) = \sigma \times (n + 1) \times Fp(Occ_{entry}), \quad (1)$$

where the size of Occ_{entry} depends on n. For the human genome (DNA), σ is 4 (A, C, G, and T) and n is around 3G (3 gigabases). Hence, each Occ entry fits in a 4-byte unsigned integer, and the footprint of Occ is about $4{\times}3\text{G}{\times}4 \approx 48$ GB. Several techniques have been developed to reduce this large footprint based on storing only a portion of Occ and calculating the rest of it [2], [13], [14]. These techniques are described and analyzed in detail in the next section. In this paper we propose a new organization of Occ that improves the performance of the *Count* algorithm, taking maximum advantage of the available memory bandwidth.

Another approach to reduce the memory usage uses wavelet trees to store the Occ data [15], [16]. These structures are specially space efficient when performing rank queries on large texts based on large alphabets. The efficiency of this solution also depends on the entropy of the text. A rank query on an alphabet of σ symbols is done by $log_2(\sigma)$ binary rank queries. Each binary query calculates several indexes, accesses several memory locations, and performs some arithmetic operations.

In the DNA context, the alphabet is very small and the text is relatively small and with high entropy. Therefore, optimized versions of the Occ matrix fit in contemporary memory systems, so as the advantage of using wavelet trees regarding the space usage is not so relevant. However, its computational cost is much higher than those implementations based on a table-based Occ.

In the experimental evaluation section we compare our proposal with an FM-index implementation based on wavelet trees using the *sdsl-lite* library [17].

3 Analysis of Exact Matching

This section presents a computational analysis, in terms of memory and performance, of the *Count* algorithm for different versions of the FM-index proposed in the literature using a table-based Occ structure. This analysis serves to introduce the main characteristics of the searching algorithm and to justify the our proposal.

Algorithm BS: Backward Search Based on FM-index
Input: FM-index of *T* (*C* & *Occ*), *Q* query, *n*=|*T*|, *p*=|*Q*|
Ouput: (*sp*,*ep*): Interval pointers of *Q* in *T*
begin
1: *sp* = *C*[*Q*[*p*]]
2: *ep* = *C*[*Q*[*p*]+1]
3: **for** *i* **from** *p*-1 **to** 1 **step** -1
4: *sp* = *LF*(*Q*[*i*],*sp*)
5: *ep* = *LF*(*Q*[*i*],*ep*)
6: **end for**
7: **return** (*sp*+1,*ep*)
end

2 LFM-chains

Fig. 1. Basic backward search algorithm based on FM-index

3.1 Full FM-index

The *full FM-index* assumes that Occ is a pre-computed full look-up table, that is, it stores counts for each possible symbol and index. It can be used to quickly locate the occurrences of a pattern (query) $Q[1..p]$ in a text $T[1..n]$, being $p \ll n$. An exact matching algorithm using FM-index has been presented in [2]. Fig. 1 illustrates the *Count* step of this algorithm, called *backward search* (BS). At the end of the algorithm, the sp and ep variables contain the start and the end indices in the suffix array of T that contains Q as a prefix, respectively.

The main operation in the search algorithm is a *Last-to-First Mapping* (LFM), which is performed by calling the function $LF()$, defined as follows:

$$LF(Q[i], u) = C[Q[i]] + Occ[Q[i], u], \tag{2}$$

where i is the index of the loop and u is either sp or ep.

Each iteration of the loop 3−6 in the BS algorithm accesses the Q string and makes two calls to the $LF()$ function, one with sp and the other with ep. Note that in every loop iteration, sp (ep) is updated using the value computed in the previous iteration. That constitutes two dependency chains of calls to $LF()$, one for sp and the other for ep. We denote these chains *LFM-chains* (see Fig. 1).

3.2 Sampled FM-index

The full FM-index requires a large amount of memory space (see (1)) but the BS algorithm exhibits low computing cost. The *sampled FM-index* is a variant of the full version that introduces a trade-off between memory footprint and computing cost [2], [13].

The storage requirements can be reduced by replacing the Occ structure with a smaller one ($rOcc$). $rOcc$ stores one column out of every d columns of Occ, that is, $rOcc[c, i] = Occ[c, 1+(i-1)\times d]$. Fig. 2 depicts this new data structure (sampled, k=1).

In order to reconstruct the content of Occ, both $rOcc$ and BWT are needed. Let us see an example for d=5. Consider that we need to know the number of occurrences of the symbol s up to the entry 8 of the BWT text, that is, the value of $Occ[s, 8]$. The nearest entry in the row s of Occ previous to $Occ[s, 8]$ that is stored in $rOcc$ corresponds to $rOcc[s, 2]$, because $\lfloor(8\text{-}1)/d{+}1\rfloor$=2. So, we can obtain $Occ[s, 8]$ by adding the value $rOcc[s, 2]$ (which is equal to $Occ[s, 6]$) and the number of occurrences of the symbol s in the sub-string of BWT from position 7 to 8. This equivalence can be expressed in general as follows:

$$\begin{aligned} Occ[s, p] &= Occ[s, q] \ + \ occur(s, BWT[(q+1)..p]) \\ &= rOcc[s, \lfloor(p-1)/d+1\rfloor] \ + \ occur(s, BWT[(q+1)..p]), \end{aligned} \tag{3}$$

being q=1+$d\times\lfloor(p-1)/d\rfloor \leq p$, and *occur(s,str)* the number of occurrences of the symbol s in the string str.

A way of improving data locality consists of placing next in memory columns of $rOcc$ and the blocks of BWT required to reconstruct Occ. This is accomplished in two steps (see Fig. 2). Firstly, rearranging the BWT text in an array of substrings of d consecutive symbols taken from BWT, called *buckets* [2]. The new data structure, named $bBWT$, is defined as $bBWT[u, v]$=$BWT[d\times(u-1)+v]$, representing the symbol v of the bucket u. Secondly, combining both $rOcc$ and $bBWT$ data structures into a new one denoted by *SFM* (*Sampled FM-index*). *SFM* associates the column j from $rOcc$ with the row j from $bBWT$. Specifically, a *SFM* row refers to a $rOcc$ column (σ counters) and the $bBWT$ bucket required to reconstruct the discarded Occ counters up to the next $rOcc$ column.

A search algorithm based on the sampled FM-index follows a similar structure as the BS algorithm, but it requires to re-write the calculation of an LFM (see (2)) as follows:

$$\begin{aligned} sLF(Q[i], m, d) &= C[Q[i]] + rOcc[Q[i], \lfloor(m-1)/d+1\rfloor] \\ &+ occur(Q[i], bBWT[\lfloor(m-1)/d\rfloor+1, [1..(m \bmod d)]]). \end{aligned} \tag{4}$$

Note that each sampled LFM uses $rOcc$ instead of Occ, which is d times smaller, but at the cost of performing more computation (the higher the value of d, the higher the computational cost).

3.3 K-step Sampled FM-index

The *k-step sampled FM-index* searches k symbols in a query in a single step [14]. This version reduces computing cost and improves data locality of the sampled version but increases slightly memory footprint.

To search k symbols in a single query, the original alphabet, Σ, is replaced by the set of k-tuples whose elements come from Σ (permutations with repetition). The new alphabet is denoted Σ^k and its size is σ^k. This change in the alphabet implies modifications in the sampled FM-index data structure. C is transformed into C_k, which is indexed by k-tuples in Σ^k (hence, its size is σ^k). $rOcc$ becomes $rOcc_k$, whose first dimension is also indexed by k-tuples (thus, its size is $\sigma^k\times\lceil(n{+}1)/d\rceil$). BWT is transformed into BWT_k, which is composed of k $(n{+}1)$-symbol strings, namely the last k columns of the M matrix (see sec. 2.1.2). These k strings, however, can be encoded as only one $(n{+}1)$-symbol string, where each symbol is now the concatenation of the k symbols of each row from the M matrix. Similarly to $bBWT$, BWT_k, encoded as a single string of k-tuples of symbols, can be blocked into buckets of size d. This new structure is denoted $bBWT_k$, an array of sub-strings composed by the concatenation of d k-tuples of symbols. As with *SFM*, the extended data structures, $rOcc_k$ and $bBWT_k$, are combined into a new one denoted by SFM_k (*k-step*

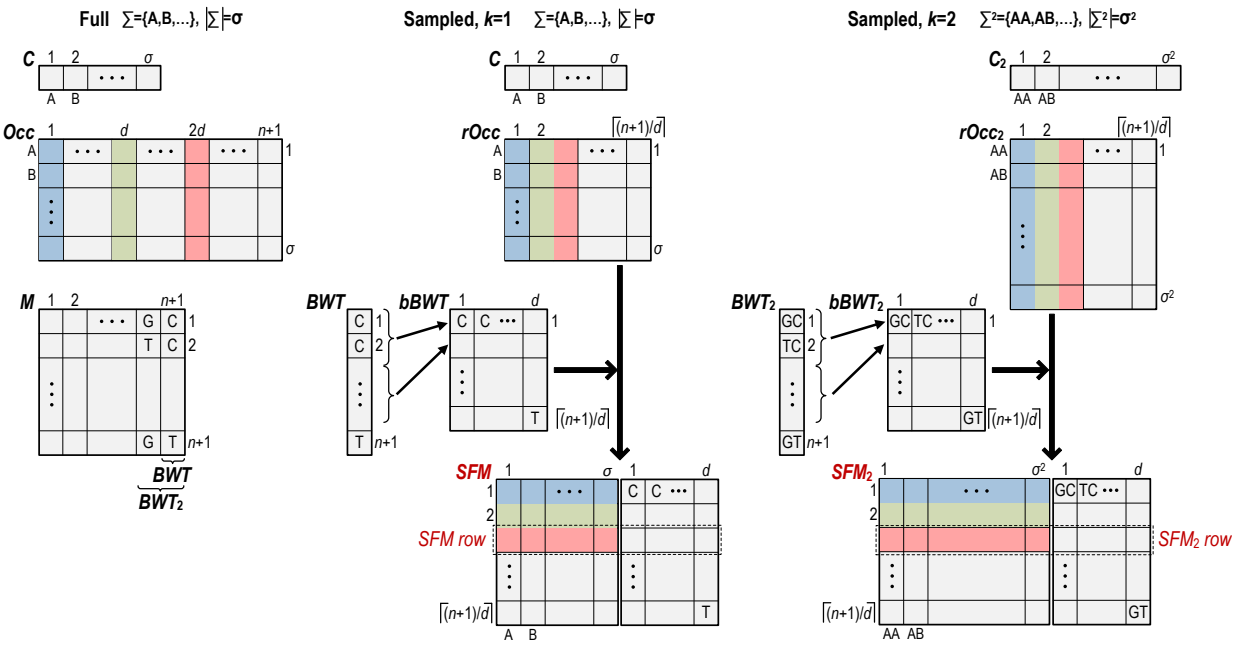

Fig. 2. Full, sampled, and 2-step sampled FM-index data structures

Sampled FM-index). Fig. 2 shows these new data structures for k=2.

The k-step version of the calculation of a LFM (denoted by $sLF_k()$) is an extension of the single-step version, $sLF()$ (see (4)), but using the extended Σ^k alphabet and the extended data structures: C_k, $rOcc_k$ and $bBWT_k$. A single $sLF_k()$ call resolves k LFMs, that is, it is equivalent to k calls to $sLF()$. A call to $sLF_k()$ reads one piece of main memory containing the data to resolve k LFMs, exploiting in this way data locality. Moreover, the computational cost of $sLF_k()$ is slightly higher than that of $sLF()$ and therefore the cost per LFM is much lower.

3.4 Memory and Performance Analysis

3.4.1 Memory footprint

The most memory consuming data structure is Occ, for full FM-index, and SFM_k, for the sampled variants. In the case of DNA, the full Occ matrix is a big data structure (see (1)). The sampled versions, however, reduces largely this size depending on the parameters d (sampling factor) and k (symbols searched in a single step). Specifically, the memory footprint for the SFM_k data structure is:

$$Fp(SFM_k) = \lceil (n+1)/d \rceil \times (\sigma^k \times R + d \times \lceil log_2 \sigma^k \rceil) \text{ bits}, \quad (5)$$

where R is the size of the $rOcc_k$ entry.

Taking the human genome example (n=3 Gbases, σ=4) and d=64, the memory footprint for SFM_k is 1.5 GBs, for k=1, and 4.5 GBs, for k=2, considering that a $rOcc$ entry fits in 32 bits. These sizes are much lower than the footprint of the full Occ structure (48 GBs).

Values of k greater than 2 require a large amount of memory due to the exponential dependency of $Fp(SFM_k)$ on the number of steps (k). Taking the same example as above but for k=4, the size of SFM_k increases to 51 GBs, greater than the original Occ structure.

3.4.2 Memory access pattern

One of the main performance limitations of the BS algorithm is related to the memory system. When executing this algorithm in an out-of-order processor, two LFM-chains are issued for each backward search query, overlapping their execution. Each of these LFMs obtains the Occ entry address (sp or ep) using the address from the previous iteration. Given how the BS algorithm works, the Occ memory access pattern for an LFM-chain is not predictable and it is distributed along the whole Occ matrix, showing neither spatial nor temporal locality. Hence, accesses to Occ result in a high cache miss rate.

However, computations from both LFM-chains are partially correlated. When a part of the query has already been performed, there are usually few matches in the reference text, and the start and end pointers (sp and ep) may have similar values. In that case, the pair of LFMs executed in a loop iteration likely access two Occ entries that are stored in the same cache block. We have measured the ratio of these cache block correlations for different query lengths and text sizes, assuming 64-byte cache blocks (see Fig. 3). It can be noted a high degree of cache block correlation between the two $LF()$ calls within the same loop iteration.

To summarize, each pair of LFMs in a loop iteration reads two different cache blocks from main memory at the beginning of a query, but they are likely to access only one cache block when the query moves forward. Let α denote the average number of cache blocks read from main memory by each LFM pair in the same loop iteration, that is,

$$\alpha = 1 + (1 - r), \quad (6)$$

being r the probability of sp and ep referring data from the same cache block (see Fig. 3). The value of α depends on factors such as the index size and the query length.

For the sampled FM-index variants, an LFM reads one $rOcc_k$ entry and a sub-string from $bBWT_k$ (see (4) for k=1),

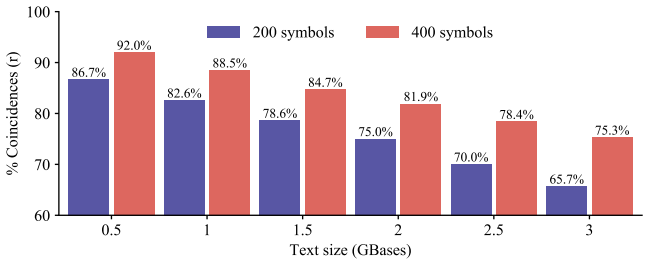

Fig. 3. Fraction of $LF(c, ep)$ calls that access the same cache block as companion $LF(c, sp)$ calls for different text and query sizes

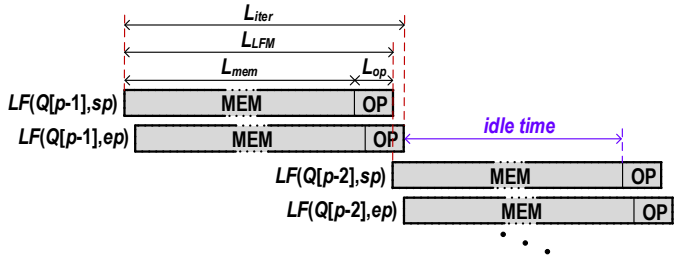

Fig. 4. Backward search timing model, where L_x represents latencies for the different phases.

TABLE 1
Full and sampled FM-index properties (B (GB) stands for bytes (Giga bytes), CB for cache blocks and P for padding)

Full			r	Occ column size (CB)	Occ size (B)	Occ size (GB)	α (CB)	SI (LFMs/B)
			0.657	1	16	48	1.34	0.0232
Sampled				SFM_k row size		SFM_k size	α_{dk}	SI_{dk}
k	d	Δ_{dk}	r_{dk}	(CB)	(B)	(GB)	(CB)	(LFMs/B)
1	32	1	0.916	1	24+8P	3	1.08	0.0288
1	192	1	0.934	1	64	1	1.07	0.0293
1	448	1.57	0.943	2	128	0.857	1.66	0.0188
1	704	2.09	0.947	3	192	0.818	2.02	0.0142
2	16	2	0.860	2	72+56P	13.5	2.28	0.0274
2	128	2	0.924	2	128	3	2.15	0.0290
2	256	2.5	0.932	3	192	2.25	2.67	0.0234
2	384	3	0.936	4	256	2	3.19	0.0196

both belonging to the same SFM_k row. In order to minimize the number of cache blocks read from main memory, the SFM_k row has to be properly aligned to cross the minimum number of cache block boundaries. Therefore, either d has a suitable value or the SFM_k row has to be padded.

Equation (6) has to be adapted for the sampled FM-index versions because d and k appear as new parameters. In particular, the average number of cache blocks read from main memory for each query step (performing $2k$ LFMs) is:

$$\alpha_{dk} = \Delta_{dk} \times (1 + (1 - r_{dk})), \tag{7}$$

where Δ_{dk} is the average number of accessed cache blocks for a pointer access (either sp or a ep), and r_{dk} is the probability that both sp and ep refer to elements stored in the same SFM_k row.

Table 1 shows the footprint of Occ and α for the full FM-index (first row in table), as well as the footprint of SFM_k and α_{dk} for the sampled versions, using different values of k and d, and 64-byte cache blocks. r and r_{dk} were obtained experimentally searching 20M sequences of DNA strings with an average length of 200 symbols in a full human genome reference (σ=4 and n=3G). Δ_{dk} was calculated assuming random accesses to $rOcc_k$. Note that, for k=1 and d=32, the SFM_k row size is 24 bytes, which is padded with 8 extra bytes in order to store two complete rows in a single cache block (similar situation for k=2 and d=16). It is worth noting that the huge memory footprint of Occ for the full version may result in frequent TLB misses for most of the modern processors.

Our design goal is to reduce α_{dk} for a given k value. On the one hand, Δ_{dk} increases with growing values of d, since the SFM_k row size increases. On the other hand, the greater the value of d, the higher the probability r_{dk}. For k=1 and k=2, this trade-off results in a minimum value of α_{dk} with d=192 and d=128, respectively. For 64-byte blocks, these are the sampling values that minimize the average number of cache blocks read from main memory for each query step. Lower d values do not exploit sp and ep locality. For higher values, the SFM_k size increase in terms of cache blocks outweighs the effect of pointers locality.

3.4.3 Search intensity

The *arithmetic intensity* is the ratio of the number of operations (work) to the amount of data traffic (in bytes) [18]. In the case of the FM-index backward search, we use the number of LFMs performed per transferred byte. Consequently, we name this metric *search intensity* (*SI*).

For the full FM-index, an LFM pair needs to retrieve, in average, α cache blocks from main memory. Hence, the SI of the BS algorithm for a B-byte cache block is:

$$SI = \frac{2}{\alpha \times B} \text{ LFMs/byte.} \tag{8}$$

For the sampled versions, search intensity is calculated in a similar way, but replacing α with α_{dk}, that is:

$$SI_{dk} = \frac{2 \times k}{\alpha_{dk} \times B} \text{ LFMs/byte,} \tag{9}$$

considering that $2k$ LFMs are searched per query step.

Table 1 shows the search intensity for the full and sampled versions of FM-index. Search intensity is maximized for the pairs (k=1,d=192) and (k=2,d=128), obtaining similar values (0.0293 and 0.0290) and slightly better than that for the full version (0.0232). For k=2, SFM_k occupies three times more memory than for k=1 (3 GB vs. 1 GB). The impact of searching k symbols in a query step on search intensity is compensated by the increase in the average number of blocks read from memory (α_{dk}). For example, for k=2 and d=128, the matching algorithm performs 4 LFMs in a query step (two $sLF_{k=2}()$) instead of the 2 LFMs performed with k=1 and d=192. However, each step loads from memory an average of 2.15 cache blocks instead of 1.07, leaving the search intensity almost unchanged.

3.4.4 Throughput

Latency. Considering the full FM-index, the execution of the pair of LFMs issued in a iteration of a search query can be modeled with two logical phases (see Fig. 4). The first phase, *MEM*, corresponds to the memory operations associated to an LFM, which is mainly due to the access to

Occ. The second phase, OP, corresponds to the computing operations of an LFM. Basically, this phase comprises the processing of three memory and one add instructions.

In an out-of-order processor with a non-blocking cache, the access to Occ with ep can be initiated while the cache is still servicing the access to Occ with sp in the same iteration (see Fig. 4). However, the next access to Occ must wait to finish the execution of the corresponding LFM of the previous iteration (due to the data dependence in the LFM-chain). As the MEM phase requires typically hundreds of cycles (L_{mem}), while the OP phase lasts few cycles (L_{op}), the processor is idle most of the time (*idle time* in Fig. 4).

Assuming that a single hardware thread per core performing a complete query, the throughput is:

$$Th^{L}_{core} = \frac{2}{L_{iter}} \approx \frac{2}{L_{LFM}} \approx \frac{2}{L_{mem}} \text{ LFMs/s}, \quad (10)$$

where L_{iter} is the latency of the iteration (see Fig. 4), and L_{LFM} is the latency of a complete computation of an LFM.

Bandwidth. Equation (10) determines an upper bound of single-thread throughput in terms of latencies. Memory bandwidth, on the other hand, also limits the maximum achievable throughput. Given the search intensity (SI), the throughput of a core (single thread) can be calculated as:

$$Th^{BW}_{core} = \frac{Th^{BW}_{system}}{N_{cores}} = \frac{SI \times BW_{system}}{N_{cores}} \text{ LFMs/s}, \quad (11)$$

being Th^{BW}_{system} the throughput of the complete system, N_{cores} the number of cores executing independent queries in parallel, and BW_{system} the main memory bandwidth for the complete system.

Sampled versions. With the sampled FM-index, the throughput upper bounds determined by query latencies and memory bandwidth are calculated by equations (10) and (11), respectively, but replacing SI with SI_{dk}.

Regarding Th^{L}_{core}, the LFM latency (L_{LFM}) increases compared with the full version because of the larger instruction count in the computing phase (OP). Hence, the maximum throughput per core is lower in the sampled versions. Regarding Th^{BW}_{core}, throughput bound is maximum for k and d values that maximize SI_{dk} (see Table 1).

3.5 Optimizing Throughput: Overlapped FM-index

The full and sampled FM-index variants have different characteristics in terms of memory footprint and data locality exploitation. However, their impact on throughput is much less important as the search intensity remains almost unchanged (see Table 1).

The exact matching algorithm (BS algorithm) is typically query latency bound, since many cycles are lost waiting for data (*idle time* in Fig. 4), wasting part of the available memory bandwidth. However, the memory latency responsible of the *idle time* can be hidden by issuing a given number of different independent queries, that is, by overlapping the memory accesses of several queries (batch or offline processing). This way, the throughput upper limit given query latencies is increased. The high number of queries which are usually involved in solving genome mapping problems makes this approach feasible.

Algorithm OBS: Query-Overlapped Backward Search
Input: FM-index of T text (C & Occ), $Q[]$ array of queries
Input: n:$|T[]|$, N_q:$|Q[]|$, p:$|Q[k]\{\}|$, k=1...N_q
Ouput: ($sp[k]$,$ep[k]$): Interval array of pointers of $Q[k]$ in T

```
    begin
1:  sp[k] = C[Q[k]{p}], k=1...N_q
2:  ep[k] = C[Q[k]{p}+1], k=1...N_q
3:  for i from p-1 to 1 step -1
4:    for k from 1 to N_q step 1
5:      sp[k] = LF(Q[k]{i},sp[k])
6:      ep[k] = LF(Q[k]{i},ep[k])
7:      prefecth(Occ[Q[k]{i},sp[k]])
8:      prefecth(Occ[Q[k]{i},ep[k]])
9:    end for
10: end for
11: return (sp[k]+1,ep[k]), k=1...N_q
    end
```

2N_q LFM-chains

Fig. 5. Backward match algorithm overlapping N_q queries

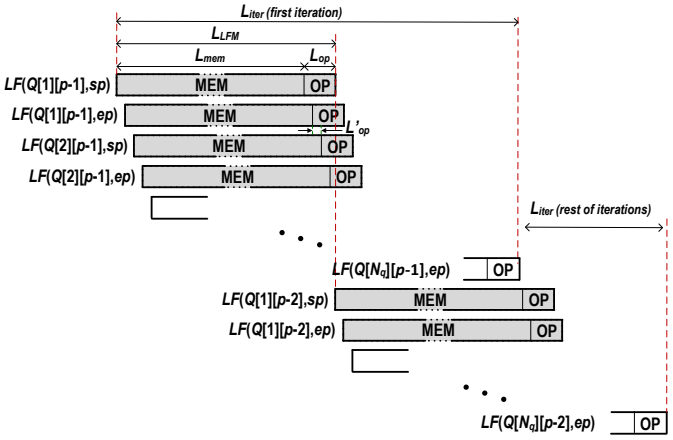

Fig. 6. Backward search timing model with N_q overlapped queries

The resulting algorithm is shown in Fig. 5 for the full FM-index. The OBS algorithm executes a total of $2N_q$ LFM-chains for each iteration of the outer loop 3−9, corresponding to an array of N_q different queries that are searched concurrently. Note that after computing the two LFMs required for a given query, two prefetch operations are issued to retrieve from memory the two Occ entries needed for computing the next two LFMs of the same query. The latencies of these memory reads are hidden by computing LFMs from other queries (see Fig. 6). By overlapping independent queries, the processor should be busy most of the time.

Assuming a single hardware thread per core, the minimum number of LFMs that must be overlapped in order to nullify the idle time is L_{iter}/L'_{op}, where L'_{op} is the latency of the fraction of the OP phase that is not overlapped with the same phase of other LFMs (see Fig. 6). So, to reach such situation, N_q must take a value such that $2 \times N_q$=L_{iter}/L'_{op}. We calculate L'_{op} for several implementations of the algorithm and for several processors in Section 5.

Using the above expression, the maximum throughput obtained by a core with the OBS algorithm is:

$$Th^{C}_{core} = \frac{2 \times N_q}{L_{iter}} = \frac{1}{L'_{op}} \text{ LFMs/s}. \quad (12)$$

Current architectures support simultaneous multithread-

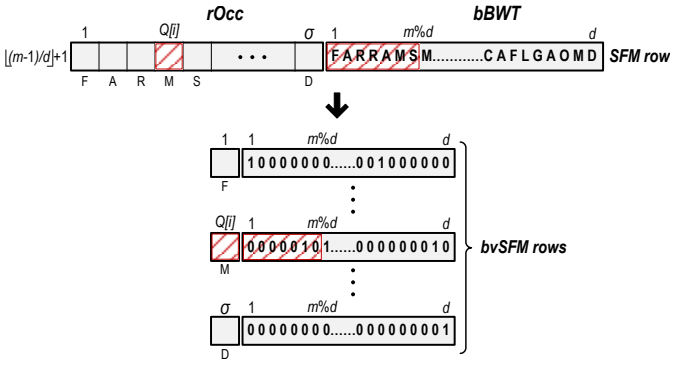

Fig. 7. Original SFM_k (k=1) data structure (top) and new $bvSFM_k$ data structure (bottom). The accessed data during the computation of an LFM is marked in red (see (4)).

ing (SMT) [19], a technique that allows a single core to execute several interleaved independent execution flows (hardware threads). In this situation, the N_q queries can be distributed among the hardware threads of a core.

4 Split Bit-Vector Sampled FM-Index

4.1 Approach

In order to increase search intensity to improve bandwidth throughput α_{dk} must be reduced. The upper part of Fig. 7 shows a row of the SFM_k data structure, for k=1. All entries accessed in the computation of an LFM are marked in red, according to (4). Note that only a single entry in $rOcc$ is accessed, so that if σ is large enough the substring accessed in $bBWT$ may be stored in a different cache block. This occurs, for example, in the case of DNA (σ=4) and k=2. Since the alphabet size is 16 (σ^k), and each entry in $rOcc$ has a size of 4 bytes, a single $rOcc_2$ column occupies a complete 64-byte cache block (16 counters $\times$ 4 bytes/counter).

We propose to change the SFM_k layout and data codification so as all data needed to compute an LFM is stored in a minimum number of cache blocks.

4.2 Description

We denote the new data structure by $bvSFM_k$, called *split bit-vector sampled FM-index*. Our solution comes from the observation that only one out of the σ^k $rOcc_k$ entries is read for each LFM computation (see upper part of Fig. 7).

The $bvSFM_k$ structure is obtained from SFM_k through two transformations: (1) partition each row of SFM_k into σ^k rows, where each of them consists of a single $rOcc_k$ entry combined with the complete bucket. Specifically, the row t of SFM_k, that is, $SFM_k[t,*] \equiv rOcc_k[*,t] \mid bBWT_k[t]$ (a concatenation of the column t of $rOcc_k$ and the bucket t), is transformed into σ^k rows of $bvSFM_k$, of the form, $bvSFM_k[(t\text{-}1)\sigma^k{+}i,*] \equiv rOcc_k[i,t] \mid bBWT_k[t]$, for i=1,...,σ^k; (2) compression of each bucket using a bitmap where each symbol is represented by a single bit. This representation is as follows: given the row $bvSFM_k[(t\text{-}1)\sigma^k{+}i,*]$ ($1 \leq i \leq \sigma^k$), the corresponding bucket ($bBWT_k[t]$) is replaced by a bitmap of length d, where a symbol in the bucket is represented by a set bit (1) if it is equal to the one associated to the entry $rOcc_k[i,t]$, and by a unset bit (0) otherwise.

TABLE 2
Split bit-vector k-step sampled FM-index parameters, for $k = 2$

d	$bvSFM_k$ row size (CB)	$bvSFM_k$ size (B)	$bvSFM_k$ size (GB)	α_{sk} (CB)	SI_{sk} (LFMs/B)
32	1	8	12	1.108	0.0564
64	1	12+4P	12	1.088	0.0574
96	1	16	8	1.081	0.0578
224	1	32	6.86	1.069	0.0585
480	1	64	6.4	1.061	0.0589

The lower part of Fig. 7 shows, as an example, the σ^k rows of $bvSFM_k$ for k=1. Note that now, all data required to calculate an LFM (in red) are placed together in memory and in a compact way, minimizing memory bandwidth consumption. The transformation also allows the *occur()* function in (4) to be simplified. Now it has to count the number of set bits (1) in the accessed bucket, operation that can be efficiently performed by the *popcount* instruction.

4.3 Memory footprint

The new $bvSFM_k$ data structure has σ^k rows for each row of the original SFM_k structure, as shown in Fig. 7. Therefore, the memory footprint of $bvSFM_k$ is:

$$Fp(bvSFM_k) = \sigma^k \times \lceil (n+1)/d \rceil \times (R+d) \text{ bits}, \quad (13)$$

where R is the size of the $rOcc_k$ entry. Note that the row size does not depend on the alphabet size (σ^k).

Table 2 shows the size of a $bvSFM_k$ row and of the complete structure for the human genome for k=2 and different values of d. For all the selected d values, a $bvSFM_k$ row fits in a cache block. Compared to the corresponding SFM_k values (see Table 1), the size of the whole data structure increases. For instance, with d=64, the $bvSFM_{k=2}$ and $SFM_{k=2}$ footprints are 12 GB and 4.5 GB, respectively. Current computing systems have enough memory to allocate this up-sized $bvSFM_{k=2}$ data structures.

4.4 Search Intensity and Throughput

To minimize memory traffic, a value must be chosen for d such that a $bvSFM_k$ row fits in a single cache block. In addition, these rows must be cache-aligned in order to not splitting a row between two consecutive cache blocks. That means that the cache block size must be an integer multiple of the row size. Otherwise, the rows must be padded accordingly.

Table 2 shows α_{dk} and SI_{dk} for different values of the sample distance d, assuming 64-byte cache blocks and k=2. Search intensity is calculated using (9). These values have been obtained in the same way as those of Table 1. Different from the $SFM_{k=2}$ rows, which require two or more cache blocks to be stored, the rows of $bvSFM_{k=2}$ fit in a single cache block. As a result, the value of α_{dk} is lower for the bit-vector version, and consequently its search intensity is about twice larger than that of SFM_k. In addition, the sensitivity of SI_{dk} with d is minimum for *bvSFM*, as its row size does not depend on d for the selected values.

The throughput upper bounds are calculated by (10) and (11). The proposed split bit-vector version improves SI_{dk} of the exact matching algorithm and, hence, increases the throughput upper bound given by memory bandwidth.

TABLE 3
Features of processors used in the evaluation

	Xeon Phi 7210 (KNL)	Xeon E5-2630V4 (Broadwell)	Xeon Gold 5120 (Skylake)
Cores	64× @ 1.3 GHz	10× @ 2.2 GHz	14× @ 2.2 GHz
IPC	2	4	4
HW Threads	4	2	2
Vector Unit	AVX-512	AVX2	AVX-512
Memory Peak BW	400 GB/s (MCDRAM) 95 GB/s (DDR4)	- 68 GB/s (DDR4)	- 107 GB/s (DDR4)
Memory Size	16 GiB (MCDRAM) 192 GiB (DDR4)	- 256 GiB (DDR4)	- 48 GiB (DDR4)

TABLE 4
Instruction count and computation latency per $sLF_k()$ call

Algorithm	Instr. count	Broadwell L'_{op} (cycles)	Broadwell L'_{op} (ns)	Skylake L'_{op} (cycles)	Skylake L'_{op} (ns)	KNL L'_{op} (cycles)	KNL L'_{op} (ns)
k1d32-SFM	33	8.25	3.8	8.25	3.8	16.5	12.7
k1d192-SFM	77.5	19.38	8.8	19.18	8.8	38.75	29.8
k2d16-SFM	37.5	9.38	4.3	9.38	4.3	18.75	14.4
k2d128-SFM	98.5	24.62	11.2	24.63	11.2	49.25	37.9
k2d64-bvSFM	23.5	5.88	2.7	5.88	2.7	11.75	9.0
k2d96-bvSFM	38	9.5	4.3	9.5	4.3	19	14.6

In addition, as with the full and sampled versions, it is feasible to combine this data structure with the overlapping of independent queries (OBS algorithm). The resulting exact matching algorithm also improves the throughput limited by query latencies (see (12)).

5 Throughput Bounds Analysis

In this section, we assess the throughput bounds of the exact matching algorithms based on the analyzed versions of FM-index. In particular, we consider the sampled versions with pairs (k=1,d=32) (k1d32-SFM), (k=1,d=192) (k1d192-SFM), (k=2,d=16) (k2d16-SFM) and (k=2,d=128) (k2d128-SFM), and the split bit-vector version with pairs (k=2,d=64) (k2d64-bvSFM) and (k=2,d=96) (k2d96-bvSFM). All versions use the query-overlapped technique (OBS) to maximize throughput. (k,d) values have been selected so that a SFM_k row occupies the minimum number of cache blocks. k1d32-SFM, k2d16-SFM and k2d64-bvSFM have 64-bit buckets, which matches the maximum data size of the processor.

The assessment is carried out in three processor architectures from Intel (see relevant features in Table 3).

5.1 Instruction count

Table 4 shows the average number of instructions in the *OP* phase (see Fig. 4) of the calculation of an LFM for the different versions of the exact matching algorithm, as well as, the time spent by the processor to execute those instructions. We have analyzed optimized x86 machine codes. The Intel IACA tool [20] was used to analyze the hardware resource occupation in Broadwell and Skylake processors, while a similar analysis was made manually in KNL because the IACA tool does not support this architecture. In all processors, the resource that limits most the computation of the LFMs is the front-end unit, in charge of processing 2 (KNL) or 4 (Broadwell, Skylake) instructions per cycle.

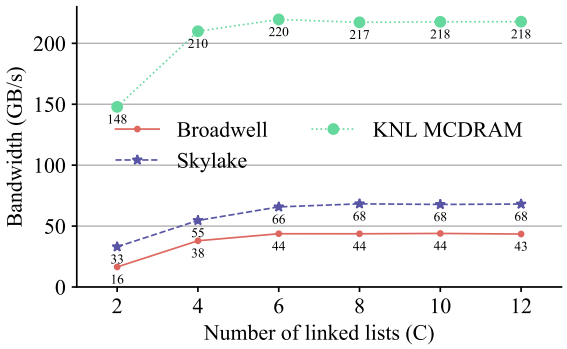

Fig. 8. Memory bandwidth for Broadwell, Skylake and KNL obtained with the RANDOM benchmark for various values of C

As expected, k1d32-SFM, k2d16-SFM and k2d64-bvSFM versions (those with 64-bit buckets) have low instruction counts because the operations in *occur(s,str)* (see (3)) translate into a few processor instructions if the bucket size is equal or smaller than the processor word (64 bits). Likewise, those versions with larger d values have higher instruction counts because they have to loop through d-symbol buckets.

5.2 Random Memory Access Benchmark

In order to have a more accurate value of memory bandwidth when issuing random memory accesses, we have developed a benchmark, called RANDOM, that performs memory operations following a random pattern similar to that in the different versions of FM-index. RANDOM uses C randomly generated linked lists with no access locality.

All the bandwidth tests have been conducted using the maximum number of hardware threads supported by cores and for a different number of linked lists (C). For C values beyond 6, the bandwidth reaches a peak and remains stable (see Fig. 8). For lower values, bandwidth is under the peak value because the memory latency cannot be completely hidden by the prefetch operations.

The maximum bandwidth corresponds to KNL MCDRAM, able to provide about 219 GB/s. This value is much lower than the peak 400 GB/s reported for the STREAM benchmark [21]. On the other hand, the peak bandwidth provided by the DDR4 DRAM memory is about 68 GB/s (Skylake) and 44 GB/s (Broadwell).

The memory access latencies were also evaluated using the RANDOM benchmark. Three load tests were used: *low load*, *medium load* and *high load*. In the low load test, only one hardware thread in the complete processor executes the benchmark that runs over a single linked list (C=1). All the others threads remain idle. In the medium load test, the maximum number of hardware threads in each processor runs over a single linked list. In the high load test, the maximum number of hardware threads (same as medium load) execute the benchmark. Every thread, except one, runs over several linked list in order to have a high load in the system. The remaining thread runs just over one linked list in order to accurately measure the memory access latency. The number of linked lists was selected to be the one that achieves the best memory bandwidth (see Fig. 8).

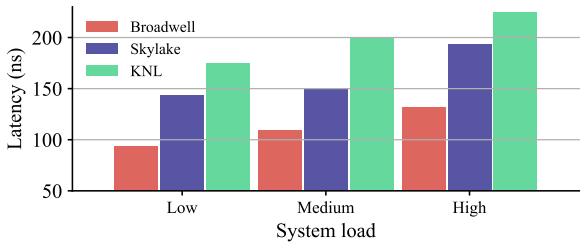

Fig. 9. Comparison of RANDOM memory latencies

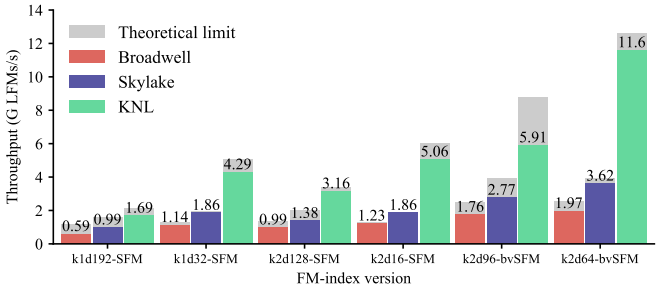

Fig. 10. Throughput for different FM-index versions

Fig. 9 shows the RANDOM latency results for the three processor architectures. It can be noted that the latency increases significantly with the load in the system. This behaviour is expected as, when the amount of simultaneous queries increases, accesses to hardware shared resources are much more likely to conflict.

5.3 Throughput Bounds

Table 5 shows the throughput upper bounds determined by query latencies (processor computing capacity) and main memory bandwidth for all FM-index versions. These values have been obtained through the expressions (12) for computing time and (11) for memory bandwidth, taking as BW_{system} those values shown in Fig. 8.

As expected, the system throughput bounds imposed by memory bandwidth are much higher in KNL than in Broadwell or Skylake. The reason is that, in KNL, the FM-index structure is stored in its MCDRAM banks which provides a much higher bandwidth than DDR4 DRAM.

KNL cores have a lower computing capacity than Broadwell/Skylake cores but a much higher memory bandwidth. This fact results in more versions of the matching algorithm being compute bound for KNL than for Broadwell/Skylake.

The split bit-vector version performs best, mainly due to its higher search intensity compared to the other versions. Specifically, the memory bandwidth throughput bound for *bvSFM* is around twice as large as that of the *SFM* versions.

In summary, the best result for Broadwell is 2.52G LFM/s and it is achieved with the k2d96-bvSFM version, with a memory footprint of 8 GB. For Skylake, the best result is 3.94G LFMs/s, obtained with the same version. Finally, for KNL, this version is compute bound, achieving the best throughput with the k2d64-bvSFM version (12.61G LFM/s), with a memory footprint of 12 GB. In general, the split bit-vector data structure strongly improves the throughput for all processor architectures.

6 Experimental Evaluation

6.1 Experimental Setup and Methodology

The evaluation was conducted on three different platforms: (1) a system with an Intel Xeon Phi 7210 processor (KNL), 16 GiB of MCDRAM and 192 GiB of DDR4 running Ubuntu 16.04.1 Linux; (2) a system with an Intel Xeon Gold 5120 processor (Skylake) and 48 GiB of DDR4 running CentOS Linux 7; and (3) a system with an Intel Xeon E5-2630v4 (Broadwell) and 256 GiB of DDR4 running Ubuntu 16.04. We used the Intel C Compiler (ICC version 17.0.4) with common flags, `-O3 -qno-opt-prefetch`, and architecture dependent flags, `-xMIC-AVX512`, `-xCORE-AVX512`, and `-xCORE-AVX2`, for the KNL, Skylake and Broadwell systems, respectively.

Thread-level parallelism has been exploited by using all the available threads in all the physical cores. For the KNL system, all FM-index data structures were placed in the MCDRAM, and it was configured in memory flat mode and in quadrant clustering mode [22]. In addition, we used 1 GiB huge TLB pages to avoid TLB misses. The overlapping factor, N_q, was set to 4 for KNL, and to 20 for Broadwell and Skylake, being the minimum number of queries to be overlapped to keep busy the processor for all the versions.

A set of 20M queries generated by the Mason simulation tool [23] was used as input data. These queries (200 symbols in average) have been searched in the human genome reference GRCh38 (3 gigabases). All experiments were conducted after loading the sequences into memory.

6.2 Throughput

Fig. 10 shows the throughput achieved by different FM-index versions. Each bar is split into two parts. The colored part (bottom of the bar) shows the value obtained experimentally in each processor. The gray part (top of the bar) shows the theoretical value achievable according to the models presented in the previous sections.

In general, the experimental values reasonably approximate the theoretical ones. The differences are due to processor features not taken into account in the models, specially, the penalty caused by branch miss-predictions. The actual throughput is about 95% of the theoretical limit for exact matching algorithms with 64-bit buckets, and it drops to around 80% for larger bucket sizes. In these cases, the count of coincidences in a variable number of 64-bit pieces forces to execute more branches, which also have unpredictable behavior because they are dependent on the input data.

The branch miss-prediction penalty of those versions with buckets larger than 64 bits adds to the substantial increase in the instruction count (see Table 4). As a result, throughput for versions with large buckets (more than 64 bits) is lower than that of versions with shorter buckets.

The exact matching algorithm based on the proposed split bit-vector data structure outperforms by about 60% and 90% the best of previous implementations, executed in Broadwell and Skylake processors, respectively. In KNL, our

TABLE 5
Throughput limits imposed by query latencies and memory bandwidth (in LFMs/s). Underlined entries show the minimum throughput

FM-index version	Broadwell				Skylake				KNL			
	Computing		Bandwidth		Computing		Bandwidth		Computing		Bandwidth	
	Core	System	Core	System	Core	System	Core	System	Core	System	Core	System
k1d32-SFM	267M	2.67G	126M	1.26G	267M	3.73G	140M	1.96G	79M	5.04G	99M	6.33G
k1d192-SFM	114M	1.14G	128M	1.28G	114M	1.59G	143M	2G	34M	2.15G	101M	6.44G
k2d16-SFM	469M	4.69G	128M	1.28G	469M	6.57G	133M	1.87G	139M	8.87G	94M	6.02G
k2d128-SFM	179M	1.79G	135M	1.35G	179M	2.50G	141M	1.98G	53M	3.38G	100M	6.38G
k2d64-bvSFM	749M	7.49G	251M	2.51G	749M	10.49G	279M	3.91G	221M	14.16G	197M	12.61G
k2d96-bvSFM	463M	4.63G	251M	2.52G	463M	6.48G	281M	3.94G	137M	8.76G	198M	12.70G

TABLE 6
Match operation performance

Implementation	Performance (GLFM/s)	Index Size (GB)
sdsl-lite library on Broadwell	0.122	1.25
sdsl-lite library on Skylake	0.147	1.25
sdsl-lite library on KNL	0.455	1.25
2-Step + AC on CPU [25]	0.5	3
2-Step + AC on GPU [25]	3.8	3
NVBIO on Tesla P100*	2.7*	0.23*
*Test performed using a reduced 950 MiB reference file		

proposal outperforms by about 135% the best of previous solutions adapted to this processor. In addition, the best throughput in KNL, obtained for k2d64-bvSFM version, is about 6x and 3x that achieved by Broadwell and Skylake, respectively. This improvement is mainly due to the ultra high-bandwidth provided by the MCDRAM memory.

6.3 Roofline Model

The roofline model [18], [24] is a simple and intuitive visual method that provides performance upper bounds for an application running in a given architecture. This model is based on the concept of *arithmetic intensity*. However, since the backward search algorithm does not perform floating-point operations, the *search intensity* is used instead.

Fig. 11 shows the roofline model of different FM-index matching algorithms on the three processor architectures. The model considers the main memory peak bandwidth and the experimental results obtained when performing random memory accesses (described in section 5.2) as the memory bandwidth bounds. This random access bandwidth is, in fact, the hardware resource that really limits the algorithm performance for the best FM-index implementation (k2d64-bvSFM) in all processor architectures. This algorithm version is able to use up to 95% of the peak bandwidth for the KNL processor (with vector extensions, AVX-512, extensively used as intrinsics in the computation of an LFM) and almost all the available bandwidth for the Broadwell and Skylake processors.

6.4 Comparison with Other Implementations

Table 6 shows the performance of different FM-index implementations presented in the literature. *sdsl-lite* results are achieved using Huffman shaped wavelet trees with no compression [26][17]. 2-step with alternate counters (AC) performance is reported in [25]. These results are achieved on dual Intel Xeon E5-2650 and a NVIDIA Kepler GTX Titan,

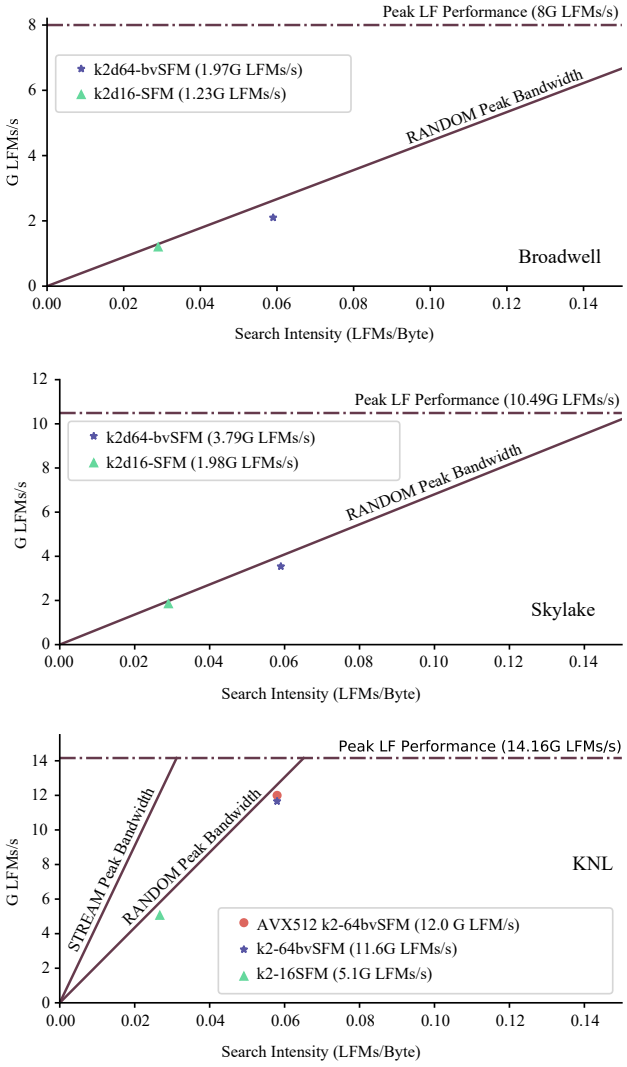

Fig. 11. Broadwell, Skylake and KNL roofline models

respectively. The NVIDIA Tesla Pascal P100, a modern GPU that includes HBM2 high-bandwidth memory technology, reports a throughput of 2.7G LFMs/s, corresponding to the execution of a NVBIO match function.

6.4.1 *Double-level bucket FM-index*

$rOcc$ counters can be organized as a two-level structure to reduce memory usage [27]. The BWT of T (L) is divided into s-symbol superbuckets, each one divided into d-symbol buckets. A first-level (L1) of counters stores, for each super-

bucket, the number of occurrences of each symbol in Σ^k from the beginning of L up to the beginning of the bucket. Similarly, a second-level (L2) of counters stores, for each bucket, the number of occurrences of every symbol from the beginning of its superbucket.

Considering 32-bit L1 and 16-bit L2 counters, and the human genome (n=3G, σ=4), L1 counters take up 2.9 MB of memory space (for k=2) and, thus, they can be stored in the last-level cache (LLC) of both Broadwell and Skylake processors (25 and 19.25 MB shared among all cores, respectively), but not in the KNL LLC cache (1 MB private for the two cores in a tile). Our results confirm this analysis. With d=16, this version reaches 1.75G, 3.13G, and 4.32G LFM/s for the Broadwell, Skylake and KNL, respectively. Those values are 89%, 86%, and 37% of the k2d64-bvSFM throughput.

7 Related work

Many sequence alignment applications based on FM-index have emerged recently, such as HISAT [28], Bowtie [4], BWA [6], [5] and SOAP [7], [29]. Furthermore, implementations for specific architectures or accelerators have been published, including GPUs (Arioc [30], CUSHAW2 [31], BarraCUDA [32], [33]), Clusters (CUSHAW3 [34]), Clouds (BigBWA [35]) and FPGAs (FHAST [36]).

Several works focus on improving the performance of the exact matching algorithm (FM-index) for GPUs, like Chacon et al. [25] and Chen et al. [37]. FM-index is also included in the NVBIO [38] library, developed by Nvidia to speed up bioinformatics using GPUs and CUDA technology.

The most relevant operation in the FM-index backward search algorithm is the $rank$ operation [39]. This operation, together with the $select$ one, has been addressed in numerous papers which focus on optimizing both the memory footprint and the pattern search time [40]. Most of this papers are based on succinct data structures [41], [42], [26] and wavelet trees [43], [44].

Unlike mentioned previous works, our paper focuses on improving the pattern search time for genomic data on CPU, specially those with many cores and high bandwidth memory. Unlike CPUs, GPUs exploit fine-grained massive parallelism, but the performance drops when the control flow diverges or the data access pattern is irregular. However, while improving pattern search time, we increase memory footprint, getting close to other fast exact matching algorithms, like suffix array binary search [45].

Even if these techniques can also be very fast, FM-Index can be used for non-exact matching. Several aligment tools are based on it and could be benefited from the ideas and techniques described in this paper.

8 Conclusions

This paper presents a new data layout organization of FM-index that boosts throughput thanks to an increase in the search intensity. Basically, our optimized data structure packs all relevant data needed in a query step within a single cache block, minimizing the memory bandwidth demand.

We have experimentally evaluated an exact search algorithm based on the proposed FM-index structure using three multi-core processors. Our proposal outperforms by about 60%, 90% and 135% the best of previous implementations.

The best performance was obtained in the Intel Xeon Phi (KNL) architecture, mainly because of the high peak random access memory bandwidth. Our implementation is able to obtain a throughput of 12G LFM/s, being about 3x faster than previous GPU implementations and about 4.4x faster than the GPU version implemented in the NVIDIA NVBIO bioinformatics library executed on a Tesla Pascal P100.

References

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

[2] P. Ferragina and G. Manzini, "Opportunistic data structures with applications," in *41st Ann. Symp. on Foundations of Computer Science*, 2000, pp. 390–398.

[3] M. Burrows and D. J. Wheeler, "A block-sorting lossless data compression algorithm," Digital Equipment Corporation, Tech. Rep. 124, 1994.

[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, pp. R25.1–R25.10, 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] H. Li and R. Durbi, "Fast and accurate long read alignment with Burrows-Wheeler transform," *Bioinformatics*, vol. 5, no. 26, pp. 589–595, 2010.

[7] 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.

[8] T. W. Lam, W. K. Sung, S. L. Tam, C. K. Wong, and S. M. Yiu, "Compressed indexing and local alignment of DNA," *Bioinformatics*, vol. 6, no. 24, pp. 791–797, 2008.

[9] B. Schmidt and A. Hildebrandt, "Next-generation sequencing: Big data meets high performance computing," *Drug Discovery Today*, vol. 22, no. 4, pp. 712–717, Apr. 2017.

[10] A. Sodani, R. Gramunt, J. Corbal, H.-S. Kim, K. Vinod, S. Chinthamani, S. Hutsell, R. Agarwal, and Y.-C. Liu, "Knights Landing: Second-generation Intel Xeon Phi product," *IEEE Micro*, vol. 36, no. 2, pp. 34–46, Mar. 2016.

[11] J. Jeffers, J. Reinders, and A. Sodani, *Intel Xeon Phi Processor High Performance Programming, Knights Landing Edition*. Morgan Kaufmann, 2016.

[12] 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.

[13] A. Chacon, S. M. Sola, A. Espinosa, P. Ribeca, and J. C. Moure, "FM-index on GPU: A cooperative scheme to reduce memory footprint," in *IEEE Int. Symp. on Parallel and Distributed Processing with Applications (ISPA 2014)*, 2014.

[14] A. Chacon, J. C. Moure, A. Espinosa, and P. Hernandez, "n-step FM-index for faster pattern matching," *Procedia Computer Science*, vol. 18, pp. 70–79, 2013.

[15] R. Grossi, A. Gupta, and J. S. Vitter, "High-order entropy-compressed text indexes," in *14th Ann. ACM-SIAM Symp. on Discrete Algorithms (SODA 2003)*, 2003, pp. 841–850.

[16] L. Foschini, R. Grossi, A. Gupta, and J. S. Vitter, "When indexing equals compression: Experiments with compressing suffix arrays and applications," *ACM Transactions on Algorithms*, vol. 2, no. 4, pp. 611–639, 2006.

[17] "Succinct data structure library 2.0." [Online]. Available: https://github.com/simongog/sdsl-lite

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

[19] D. M. Tullsen, S. J. Eggers, and H. M. Levy, "Simultaneous multithreading: Maximizing on-chip parallelism," in *22nd Ann. Int. Symp. on Computer Architecture (ISCA 1995)*, Jun. 1995.

[20] "Intel Architecture Code Analyzer," 2012. [Online]. Available: https://software.intel.com/en-us/articles/intel-architecture-code-analyzer

[21] J. D. McCalpin, "Memory bandwidth and machine balance in high performance computers," *IEEE Technical Committee on Computer Architecture Newsletter*, pp. 19–25, Dec. 1995.

[22] R. Asai, "MCDRAM as High-Bandwidth Memory (HBM) in Knights Landing processors: Developer's guide," 2016. [Online]. Available: https://colfaxresearch.com/knl-mcdram
[23] M. Holtgrewe, "Mason - A read simulator for second generation sequencing data," Freie Universitaet Berlin, Tech. Rep. 962, 2010.
[24] D. Doerfler, J. Deslippe, S. Williams, L. Oliker, B. Cook, T. Kurth, M. Lobet, T. Malas, J.-L. Vay, and H. Vincenti, "Applying the roofline performance model to the Intel Xeon Phi Knights Landing processor," in *Int. Conf. on High Performance Computing*, 2016.
[25] A. Chacon, S. Marco-Sola, A. Espinosa, P. Ribeca, and J. C. Moure, "Boosting the FM-index on the GPU: Effective techniques to mitigate random memory access," *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, vol. 12, no. 5, pp. 1048–1059, 2015.
[26] S. Gog, J. Kärkkäinen, D. Kempa, M. Petry, and S. J. Puglisi, "Faster, minuter," in *Data Compression Conf. (DCC 2016)*, 2016.
[27] P. Ferragina and G. Manzini, "An experimental study of an opportunistic index," in *12th Ann. ACM-SIAM Symp. on Discrete Algorithms (SODA 2001)*, 2001, pp. 269–278.
[28] D. Kim, B. Langmead, and S. L. Salzberg, "Hisat: a fast spliced aligner with low memory requirements," *Nature Methods*, vol. 12, 2015.
[29] R. Luo, T. Wong, J. Zhu, C.-M. Liu, X. Zhu, E. Wu, L.-K. Lee, H. Lin, C. D. W. Zhu, W., H.-F. Ting, S.-M. Yiu, S. Peng, C. Yu, Y. Li, R. Li, and T.-W. La, "SOAP3-dp: Fast, accurate and sensitive GPU-based short read aligner," *PLoS One*, vol. 8, 2013.
[30] 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:e808, 2015.
[31] Y. Liu and B. Schmidt, "CUSHAW2-GPU: Empowering faster gapped short-read alignment using GPU computing," *IEEE Design and Test*, vol. 31, no. 1, pp. 31–39, 2014.
[32] P. Klus, S. Lam, D. Lyberg, M. S. Cheung, G. Pullan, I. McFarlane, G. S. H. Yeo, and B. Y. H. Lam, "BarraCUDA - a fast short read sequence aligner using graphics processing units," *BMC Research Notes*, vol. 5, no. 27, 2012.
[33] J. Blom, T. Jakobi, D. Doppmeier, S. Jaenicke, J. Kalinowski, J. Stoye, and G. A., "Exact and complete short-read alignment to microbial genomes using Graphics Processing Unit programming," *Bioinformatics*, vol. 27, no. 10, pp. 1351–1358, 2011.
[34] 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, 2016.
[35] J. M. Abun, 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.
[36] E. B. Fernandez, J. Villarreal, and S. Lonardi, "FHAST: FPGA-based acceleration of Bowtie in hardware," *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, vol. 12, no. 5, pp. 973–981, 2015.
[37] S. Chen and H. Jiang, "An exact matching approach for high throughput sequencing based on BWT and GPUs," *IEEE 14th Int. Conf. on Computational Science and Engineering*, 2011.
[38] "NVBIO: A library of reusable components designed by NVIDIA corporation to accelerate bioinformatics applications using CUDA." [Online]. Available: http://nvlabs.github.io/nvbio
[39] G. Jacobson, "Space-efficient static trees and graphs," in *30th Ann. Symp. on Foundations of Computer Science*, 1989.
[40] V. Mäkinen and G. Navarro, "Rank and select revisited and extended," *Theoretical Computer Science*, vol. 387, no. 3, 2007.
[41] R. Raman, V. Raman, and S. R. Satti, "Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets," *ACM Transactions on Algorithms*, vol. 3, no. 4, 2007.
[42] S. Gog and M. Petri, "Optimized succinct data structures for massive data," *Software – Practice & Experience*, vol. 44, no. 11, 2014.
[43] T. Gagie, G. Navarro, and S. J. Puglisi, "New algorithms on wavelet trees and applications to information retrieval," *Theoretical Computer Science*, vol. 426–427, pp. 25–41, 2012.
[44] A. Golynski, J. I. Munro, and S. S. Rao, "Rank/select operations on large alphabets: A tool for text indexing," in *17th Ann. ACM-SIAM Symp. on Discrete Algorithm (SODA 2006)*, 2006, pp. 368–373.
[45] U. Manber and G. Myers, "Suffix arrays: A new method for on-line string searches," *1st Ann. ACM-SIAM Symp. on Discrete Algorithms (SODA 1990)*, pp. 319–327, 1990.

Jose M. Herruzo received his B.S. and M.S. degrees in Computer Engineering from University of Cordoba in 2014 and from the University of Malaga in 2015, respectively. He is currently working toward the PhD degree at the University of Malaga. His current research interests include processing in memory, near-data processing, stacked memory architectures, high-performance computing and high-throughput sequence alignment.

Sonia Gonzalez-Navarro received the B.S. and M.S. degrees in Mathematics in 2000 and the Ph.D. degree in Computer Science in 2006, both from the University of Malaga, Spain. She is currently an assistant professor in the Computer Architecture Department at the University of Malaga. Her research interests include computer arithmetic, floating point number computation, high-performance architectures for data-intensive applications and near-data processing.

Pablo Ibáñez-Marín received the MS degree in computer science from the UPC in 1989, and the PhD degree in computer science from the University of 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.

Víctor Viñals-Yúfera received the MS degree in Telecommunications and the PhD degree in Computer Science from the UPC in 1982 and 1987, respectively. He was associate professor in the UPC from 1983 to 1988. Currently, he is full professor in the Informática e Ingeniería de Sistemas Dept. at the University of Zaragoza. His research interests include processor microarchitecture, memory hierarchy, and parallel computer architecture.

Jesús Alastruey-Benedé received the MS degree in Telecommunication and the PhD degree in Computer Science from the University 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.

Oscar Plata earned a M.S. in Physics in 1985 and a Ph.D. in Physics in 1989, both from the University of Santiago de Compostela, Spain. He started as Assistant Professor in the same University where he became Associated Professor in 1990. He moved to the University of Malaga in 1995, where he became Full Professor in the Computer Architecture Dept. in 2002. His research interests are related to high performance computing and parallel architectures.