Exact Alignment with FM-Index on the Intel Xeon Phi Knights Landing Processor

ABSTRACT

The FM-index is an efficient data structure useful for searching strings in large reference texts. In fact, FM-index is used in many sequence aligner tools. Due to the data structure layout, the searching process based on FM-index exhibits irregular memory access patterns, causing a high cache miss rate. Besides, it is common for the search algorithm to be memory bound due to the low arithmetic intensity. As a result, different versions of FM-index and corresponding aligning algorithms were designed to minimize memory traffic due to those random accesses.

This paper describes various aligning algorithms based on FM-index and propose a modification of the FM-index data structure capable of reducing memory bandwidth and increase operational intensity, in order to attain a performance near the limit given by the peak random access memory bandwidth. Experiments on a Xeon Phi KNL processor obtain a throughput (Last-to-First operations per second) much higher than GPU and CPU implementations previously reported in the literature.

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 has appeared in industry. To support the progress of NGS technologies, new fast alignment tools and algorithms have recently been developed.

FM-index is a data structure well suited for fast exact matches of short reads to large reference genomes while keeping a small memory footprint [1]. Many efficient sequence aligners are based on FM-index, such as Bowtie2 [2, 3], BWA [4], BWA-SW [5] for long reads, SOAP2 [6] and BWT-SW [7].

As high-throughput sequencing systems produce a massive amount of data, despite the efficiency of the above aligner tools, the usage of high-performance techniques and platforms is of crucial importance to deal with the computational challenge. In fact, many optimized aligning algorithms have appeared recently in the literature for different architectures, like CPU clusters, GPUs and FPGAs [8], [9], [10], [11].

Due to the data structure layout, the searching process based on FM-index exhibits irregular memory access patterns. These data access patterns causes a high cache miss rate on typical cache hierarchies of multicore processors. Besides, it is common for the search algorithm to be memory bound due to the low arithmetic intensity. Since several reads can be searched concurrently in order to hide the memory latency, the limit is imposed by memory bandwidth rather than by latency.

This paper describes various aligning algorithms based on FM-index designed to deal with the data locality problem. We propose a modification of the FM-index data structure capable of reducing the memory bandwidth requirements in order to attain a performance near the limit given by the peak random access memory bandwidth. We have evaluated the proposed FM-index data structure and a corresponding search algorithm using an Intel Xeon Phi Knights Landing (KNL) processor. KNL includes AVX-512 vector processing units in each core and novel ultra high-bandwidth 3D MCDRAM memory modules integrated on package. These features provides both a remarkable computing power (up to 72 cores) and a 400 GB/s peak memory bandwidth, making it a good hardware platform for accelerating aligning algorithms.

The contributions of this paper can be summarized as follows:

- k-SFMbv, a new FM-index data structure layout and codification is proposed. k-SFMbv limits the required data traffic between memory and processor cores for the exact search algorithm.
- The new proposal is implemented on a KNL processor, exploiting the available ultra high-bandwidth memory modules.
- The roofline model [12] is used to show the experimental results and how we achieve near peak performance.

2. SEARCH ALGORITHMS WITH FM-INDEX

FM-index data structure uses the Suffix Array and Burrows-Wheeler transform (BWT) [13] to compress the input text.

The Suffix Array of a character string T is an array containing the starting positions of all suffixes of T in

Algorithm BS: Backward Search Based on FM-index

Input: FM-index of T text (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

Figure 1: Basic backward search algorithm based on FM-index

lexicographical order. For instance, if $T = [aacacbaa]$ then $SA = [8, 7, 1, 2, 4, 6, 3, 5]$. The suffix array requires $n \lceil log_2\ n \rceil$ bits (being n the length of the string T) and can be used as an index to locate every occurrence of a pattern in the string. Searching in a suffix array can be done using a binary search algorithm in $log_2\ n$ steps.

The BWT is a permutation of a character string and consists of three steps. First, the symbol \$ is appended to the end of the original text T, being \$ lexicographically smaller than any symbol of Σ (alphabet from where the text's characters are drawn from). Second, a conceptual $n \times n$ matrix M is formed, whose rows are cyclic shifts of T sorted in lexicographical order. Third, the result of applying BWT to T is L, the last column of the matrix M.

The FM-index of a text T is composed by two data structures derived from the BWT: the C array and the Occ matrix. The C array stores in $C[c]$ the number of occurrences in T of the symbols lexicographical smaller than c. $Occ[c, i]$ contains the number of occurrences of the symbol c in the prefix $BWT[1...i]$ of T.

2.1 Basic Search Algorithm

The FM-index data structures C and Occ can be used to locate the occurrences of a query $Q[1...p]$ in a text $T[1...n]$. The exact matching algorithm in [1], also called *backward search*, performs the search with a complexity of $\Theta(p)$, improving the suffix array binary search algorithm. Algorithm BS in Fig. 1 illustrates a version of the backward search algorithm.

The main operation in the FM-index search algorithm is a *Last-to-First* operation (LFop), defined as follows:

$$LF(Q[i], m) = C[Q[i]] + Occ[Q[i], m], \qquad (1)$$

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

The Occ memory access pattern is not predictable showing neither spatial nor temporal locality. Hence, each access to Occ misses in all the cache levels and results in a read request to main memory, severely restraining the algorithm performance.

The overall performance can be improved by overlapping the memory accesses of several queries in order to hide the main memory access latency. The high number of queries which are usually involved in genome mapping problems makes this approach feasible by increasing the sequence parallelism in each core.

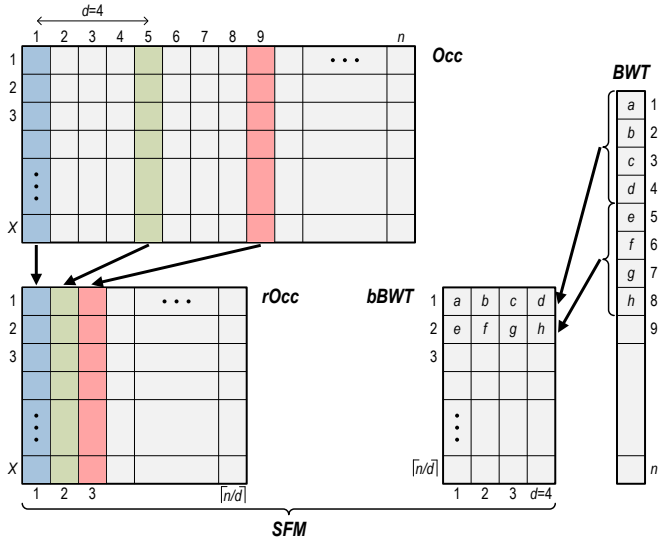

Figure 2: Sampling the Occ structure ($rOcc$) and blocking the BWT structure ($bBWT$), with $d = 4$

2.2 Sampled FM-index

Let us assume an alphabet Σ of X symbols. Therefore, the Occ matrix (of size $X \times n$) will require a large amount of storage space when indexing a long text (large n). For example, the Occ matrix for a human genome (Σ={A,C,G,T}) with 3 Gbases needs 48 GBytes of memory.

The storage requirements can be reduced by replacing the Occ structure with another smaller one denoted $rOcc$ [1]. $rOcc$ stores one column out of every d columns of Occ, that is, $rOcc[c, i] = Occ[c, 1+(i-1)\times d]$. In addition, in order to improve the memory locality the string BWT is rearranged in sub-strings of d consecutive symbols taken from BWT, called buckets [1]. These buckets ($bBWT$) are stored along with $rOcc$ giving a new data structure, *SFM*. Fig. 2 shows an example of this new data structure for $d = 4$.

In order to reconstruct the content of Occ, both data structures, $rOcc$ and $bBWT$, are needed. Although the memory footprint is reduced with the sampled FM-index, it introduces complexity in the computation of the LFops in the search algorithm since Occ has to be reconstructed from $rOcc$.

2.3 K-step Sampled FM-index

Data locality can be improved if several symbols are queried in a single LFop (see (1)) [14]. This solution basically replaces the original alphabet, Σ, by the set of k-tuples whose entries come from Σ (permutations with repetition). The new alphabet is denoted Σ^k and its size is X^k. This change in the alphabet requires modifying the sampled FM-index data structure. C is transformed into k-C, containing X^k entries, while $rOcc$

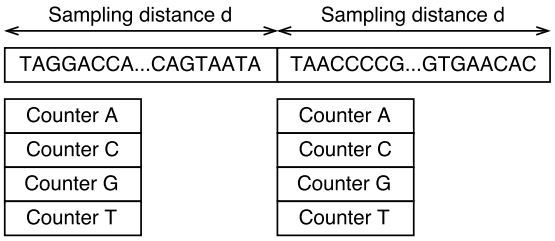

Figure 3: Original *k-SFM* data structure for DNA and 2-step sampled FM-index (k=2)

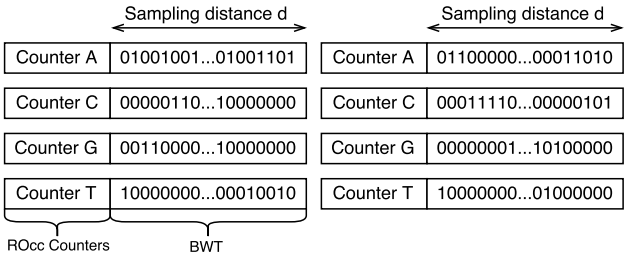

Figure 4: Proposed *k-SFMbv* data structure for DNA and 1-step sampled FM-index (k=1)

is transformed into *k-rOcc* with size $X^k \times \lceil n/d \rceil$. The $bBWT$ size is also affected, but linearly instead of exponentially. $bBWT$ is transformed into *k-bBWT*, composed of k n-symbol strings, namely the last k columns of the M matrix in the definition of the BWT.

The backward search steps, $LF_k()$, are computed like the one-step version, but with an extended Σ^k alphabet and with larger data structures. A single $LF_k()$ function execution resolves k basic LFops. Hence, each $LF_k()$ function call reads a set of consecutive locations of main memory (exploiting data locality). As a result, the main memory bandwidth required is reduced. However, it is worth mentioning that this approach is, in general, unfeasible for values of k greater than 2 due to the great amount of memory occupied by *k-rOcc* ($X^k \times n$).

3. BIT-VECTOR K-STEP SAMPLED FM-INDEX

The number of cache blocks that a LFop requires to read from main memory negatively impacts the performance of the aligning algorithm. As an example, let us consider a DNA string with 4 different symbols in the alphabet (X=4), a 2-step sampled FM-index and 4-byte values stored in 2-C and 2-*rOcc*. The entries (counters) of 2-*rOcc* required for the 2 LFops computed in $LF_2()$ occupies a complete 64-byte cache block ($X^k \times counter\text{-}size = 4^2 \times 4\ bytes$). Hence, the 2-$bBWT$ bucket must be stored in a different cache block. As a result, each $LF_2()$ needs to read two cache blocks from memory to perform two LFops, as in the case of the 1-step sampled version, although in two $LF()$ calls. This fact leaves performance essentially unchanged in both versions.

Burrows et al. [13] proposed the use of a new level of counters (super-buckets) which makes possible to use shorter counters in *k-SFM*. However, for big texts, the super-buckets data would be too big to be stored on cache and would increase the amount of main memory traffic.

This lead us to focus on reducing the number of cache blocks that each call $LF_k()$ has to load from main memory. In order to reduce memory traffic, we propose to change the *k-SFM* layout and codification (see an example of the original *k-SFM* data structure for a DNA alphabet in Fig. 3). *k-SFM* is re-arranged in order to fit all the data needed by a call to $LF_k()$ in a minimum number of cache blocks. In the case of $LF_2()$, only in a single cache block. We denote the new structure by *k-SFMbv* (bit-vector k-step sampled FM-index). Our solution comes from observing that in the aligning algorithms described in the previous section, the computation of a LFop reads more data than it is required. Specifically, the counters (*k-rOcc* entries) of all the alphabet symbols are read (e.g., 16 counters for 4 DNA symbols and 2-step sampled FM-index), when just one counter is really necessary.

We propose to separate the counters of a *k-SFM* entry in several cache blocks and place the counter of each symbol together with the required data to count the number of occurrences of that symbol in the bucket.

With this aim, we have designed a new data structure which represents each *k-BWT* bucket together with several bitmaps, each one related to a symbol of the alphabet. This is shown in Fig. 4 for the 1-step sampled FM-index. The 2-step sampled FM-index requires 16 bitmaps for DNA. Each bit in a bitmap corresponds to a symbol of the *k-BWT*, being 1 if the *k-BWT* symbol equals the one associated to the bitmap, and 0 otherwise.

The *k-SFMbv* entry is an array with the same number of elements as different symbols contains the alphabet. Each element stores the bitmap and the counter associated to a given symbol. This reduces to only one element of the *k-SFMbv* entry the amount of memory required to be read for every LFop, at the cost of increasing the memory footprint of the whole *k-SFMbv* structure. Since most high-performance hardware platforms have enough memory to allocate an actual genome, this up-sized data structure should not be a problem.

4. EXPERIMENTAL EVALUATION

We have evaluated the performance of the aligning algorithm using the described sampled FM-index data structures in two different processor architectures:

- Xeon Phi 7210 (KNL) [15]: 64 cores at 1.3 GHz with four hardware threads and two AVX-512 vector units. The MCDRAM memory has a peak bandwidth of about 400 GB/s and the six DDR4 memory channels can provide up to 90 GB/s [1].
- Xeon E5-2630V4 (Broadwell) [16]: 10 cores at 2.2 GHz with two hardware threads and three AVX2 vector units. The four DDR4 memory channels have a peak bandwidth of 68 GB/s [1].

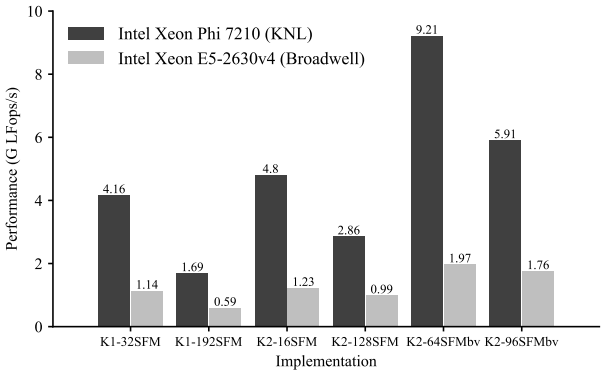

Figure 5: Performance results

Two data sets of 20M 200-symbol queries were used, generated by the Mason [18] simulation tool. These queries have been searched in the human genome reference GRCh38, composed of 3G symbols. Throughput (LFops per second) was used as the metric to measure performance and compare the different implementations. Fig. 5 compares the performance achieved by the aligning algorithm using different versions of the original *k-SFM* and *k-SFMbv* data structures. Specifically, we have implemented the 1-step sampled FM-index with d=32 and d=192 (K1-32SFM and K1-192SFM), the 2-step sampled FM-index with d=16 and d=128 (K2-16SFM and K2-128SFM), using the original *k-SFM* data structure. We have also developed a 2-step sampled FM-index with d=64 and d=96 using our proposed *k-SFMbv* data structure. Note that the version which obtains higher performance in both processor architectures is K2-64SFMbv. The values of d in the experiments were chosen because they minimize memory traffic and optimize computational performance.

4.1 Random Memory Access Benchmark

In order to obtain a more accurate view of the memory performance when issuing random memory accesses, we have developed a benchmark, that we call RANDOM, that performs memory operations following a random access pattern similar to that when computing LFops using FM-index, but with no computing at all.

RANDOM uses one or several randomly generated linked lists with no data locality. The size of the data structure is 12 GB, so it can be fitted in the KNL high-bandwidth MCDRAM memory. The kernel of our benchmark is a thread that traverses a configurable number of linked lists.

All the memory bandwidth tests have been done using the maximum number of threads per core but with different number of linked lists being simultaneously traversed by each thread.

The bandwidth that the KNL MCDRAM is able to provide when each thread traverses 6 lists in parallel is 176.4 GB/s (accessing randomly single cache blocks) and 188.7 GB/s (accessing randomly pieces of two contiguous cache blocks). This value is much lower than the 400 GB/s reported for the STREAM benchmark [17]. On the other hand, the bandwidth provided by the Broadwell processor is 42.24 GB/s (single cache blocks) and 48.07 GB/s (two contiguous cache blocks), far below the KNL performance.

These RANDOM benchmark results are used to calculate the practical bandwidth limits for the roofline model.

4.2 Roofline Model

The roofline model [12, 19] is a method that provides the upper bound of performance for an application running in a given architecture. It is very useful to show which implementation adapts best to a specific architecture. The roofline model is based on the concept of *arithmetic* or *operational intensity*, defined as the ratio of the number of operations (work) to the amount of data traffic (in bytes). In the case of the aligning algorithms using FM-index, we use the number of LFops performed per transferred byte. Consequently, we name this metric *search intensity* (*SI*).

Figures 6 and 7 show the roofline model of the aligning algorithms using different versions of FM-index on the KNL and the Broadwell processors. In the model, we have considered the main memory peak bandwidth and the experimental bandwidth results obtained when performing random block accesses (described in the previous section). This random access bandwidth is, in fact, the resource that really limits the algorithm performance for the best FM-index implementation (K2-64SFMbv) in both processor architectures. Performance in KNL reaches about 93% of peak bandwidth while Broadwell reaches about 83% of peak bandwidth. Note that our proposed FM-index data structure increases SI compared to the k-step sampled one. This fact results in an important increase of throughput.

The horizontal dashed lines appearing in figures 6 and 7 are shown for reference and comparison purposes. Previous GPU and CPU performance is reported in [9]. Tesla Pascal P100 performance corresponds to the execution of a NVBIO [20] match function on that GPU. Peak LF performance is a theoretical calculation considering an unlimited memory bandwidth.

5. CONCLUSIONS

In this work a modification of the k-step sampled FM-index data structure is proposed with the aim of reducing memory traffic when executing the main operation (LFop) in the aligning algorithm.

We have evaluated our proposal in two different processor architectures, one of them offering an ultra high-bandwidth memory (KNL). The experimental results show that our proposal obtains a much higher throughput than the previous versions due to the fact that the available effective memory bandwidth (for random accesses) is better exploited, as the roofline model shows.

[1]All peak memory bandwidth measurements have been performed using the STREAM benchmark[17]

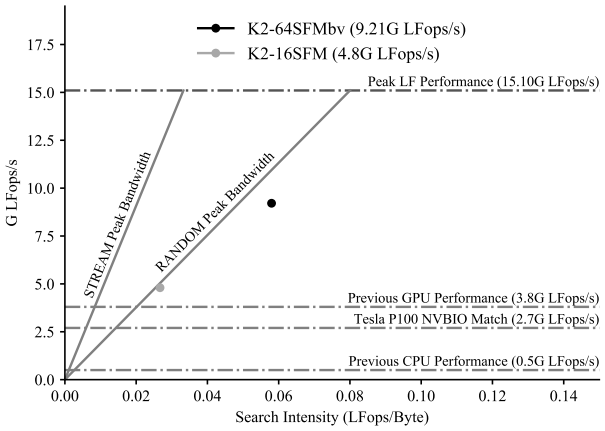

Figure 6: KNL roofline model

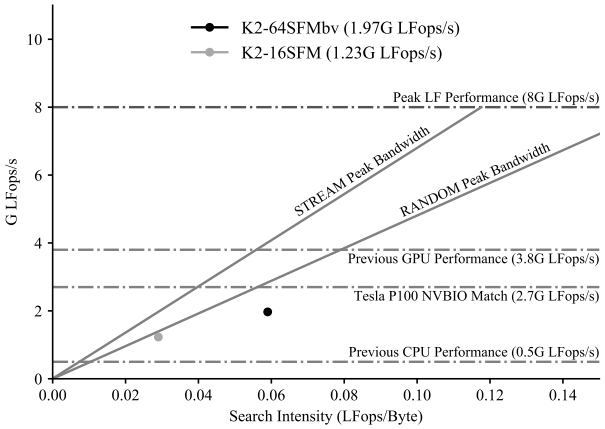

Figure 7: Broadwell roofline model

Our implementation is able to obtain a throughput of up to 9.21G LFops/s, being about 2.5x faster than previous GPU implementations and about 3.3x faster than the GPU version implemented in the NVIDIA NVBIO bioinformatics library executed on a NVIDIA Tesla P100.

6. REFERENCES

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

[2] 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.

[3] B. Langmead and S. L. Salzberg, "Fast gapped-read alignment with Bowtie 2," *Nature Methods*, vol. 9, no. 4, pp. 357–359, 2012.

[4] H. Li and R. Durbin, "Fast and accurate short read alignment with Burrows-Wheeler transform," *Bioinformatics*, vol. 25, no. 14, pp. 1754–1760, 2009.

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

[6] 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.

[7] 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.

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

[9] 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*, pp. 1048–1059, 2015.

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

[11] W. Tang, W. Wang, B. Duan, C. Zhang, G. Tan, P. Zhang, and N. Sun, "Accelerating millions of short reads mapping on a heterogeneous architecture with fpga accelerator," in *Field-Programmable Custom Computing Machines (FCCM), 2012 IEEE 20th Annual International Symposium on*, pp. 184–187, IEEE, 2012.

[12] "Roofline performance model."

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

[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] "Intel xeon phi processor 7210 (16gb, 1.30 ghz, 64 core) product specifications."

[16] "Intel xeon processor e5-2630 v4 (25m cache, 2.20 ghz) product specifications."

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

[18] M. Holtgrewe, "Mason - a read simulator for second generation sequencing data," Tech. Rep. 962, Freie Universitaet Berlin, 2010.

[19] S. Williams, A. Waterman, and D. Patterson, "Roofline: An insightful visual performance model for multicore architectures," *Communications of the ACM*, vol. 52, pp. 65–76, Apr. 2009.

[20] "NVBIO: A library of reusable components designed by NVIDIA corporation to accelerate bioinformatics applications using CUDA."