Boosting the FM-index on the GPU: effective techniques to mitigate random memory access

Alejandro Chacón*, Santiago Marco-Sola†, Antonio Espinosa*, Paolo Ribeca‡†, and Juan Carlos Moure*

Abstract—The recent advent of high-throughput sequencing machines producing big amounts of short reads has boosted the interest in efficient string searching techniques. As of today, many mainstream sequence alignment software tools rely on a special data structure, called the FM-index, which allows for fast exact searches in large genomic references. However, such searches translate into a pseudo-random memory access pattern, thus making memory access the limiting factor of all computation-efficient implementations, both on CPUs and GPUs. Here we show that several strategies can be put in place to remove the memory bottleneck on the GPU: more compact indexes can be implemented by having more threads work cooperatively on larger memory blocks, and a k-step FM-index can be used to further reduce the number of memory accesses. The combination of those and other optimisations yields an implementation that is able to process about 2 Gbases of queries per second on our test platform, being about 8× faster than a comparable multi-core CPU version, and about 3× to 5× faster than the FM-index implementation on the GPU provided by the recently announced Nvidia NVBIO bioinformatics library.

Index Terms—GPGPU, Bioinformatics, Short Read Mapping, FM-index, Fine-Grain Parallelism, Memory-Level Parallelism

1 Introduction

The recent advent of high-throughput sequencing machines producing big amounts of short reads has boosted the interest in efficient string searching techniques. For instance, current Illumina HiSeq 2000/2500 sequencers can read out several billions of short DNA strings (usually of length 100-150) during a single run, which takes less than a week to complete. Each machine of the announced forthcoming Illumina HiSeq X Ten system will be able to generate 6 to 10 times more data per unit time than an Illumina HiSeq 2000. In most cases, the sequencing reads thus produced require subsequent alignment to a genomic reference, i.e. an error-tolerant search able to locate the positions in the genome the read might have originated from. Mammalian genomic references, like the human one, have a typical size of several Gbases.

As of today, the most effective sequence alignment software tools (like BWA [1], CUSHAW2 [2], SOAP3 [3], Bowtie [4], and GEM [5]) rely on a special data structure, called the *FM-index*, which allows for fast exact searches into large genomic references. The cost of the search is linear in the length of the searched pattern, and in theory does not depend on the size of the reference sequence (although it weakly does in practice). In addition, the FM-index can achieve high compression ratios, allowing to store the 3 Gbases of the human genome into 1-3 GB of memory space.

Low-cost GPU cards combine excellent performance with high energetic efficiency, offering at the same time thousands of computational cores and high memory access throughput (although, regrettably, not low latency); hence they are being increasingly used by big-data scientific applications to speed up computational-intensive algorithms. In particular, GPUs would look like ideal candidates to tackle the problem of short-read alignment: a situation where each read can be independently searched in the reference would seem to offer plenty of thread- and memory-level parallelism to hide the latencies of computation and memory access operations, thus showing clear potential for an efficient GPU implementation.

In practice, however, things are not so clear-cut. The main problem stems from the fact that exact searches in the FM-index framework translate into a pseudo-random memory access pattern, thus making memory access the limiting factor of all computation-efficient implementations, both on CPUs and GPUs. Straightforward parallelisation does not solve the problem, but rather exacerbates it. For instance, previous implementations of the FM-index on GPU ([6], [2] and [3] to name a few) typically assign one independent search task to each GPU thread; the result is an even more inefficient memory access, with too many threads requesting many relatively small data blocks spread across distant random locations.

*Universitat Autònoma de Barcelona, Bellaterra 08193, Spain.
†Centro Nacional de Análisis Genómico, Barcelona 08028, Spain.
‡The Pirbright Institute, Woking GU24 0NF, UK.
{alejandro.chacon, antoniomiguel.espinosa, juancarlos.moure}@uab.es
{santiagomsola, paolo.ribeca}@gmail.com

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.2014.2377716, IEEE/ACM Transactions on Computational Biology and Bioinformatics

In this paper we examine several strategies to remove the memory bottleneck on the GPU. Our contribution can be summarised as follows:

- In previous works we demonstrated that more compact FM-indexes can be implemented by having more threads work cooperatively on larger memory blocks [7], and a k-step FM-index can be used to further reduce the number of memory accesses [8]. Here we show that the *combination* of those and other optimisations yields an implementation that is able to process about 2 Gbases of queries per second on our test platform, being about 8× faster than a comparable multi-core CPU version, and about 3× to 5× faster than the FM-index implementation on the GPU provided by the recently announced Nvidia NVBIO bioinformatics library [9].
- We carry out a fully detailed performance analysis of the FM-index search algorithm executed on the GPU, for a wide range of implementations, indexing schemes and GPU platforms. Our analysis pinpoints the demanding requirements posed by random accesses to the memory system, and the crucial role played by the working set granularity. We also motivate why the effect of the optimisations we propose is synergistic, and why it scales well across very different GPU systems. We expect such in-detail study should be useful to optimise most of the applications using the FM-index on the GPU, in particular those employing a simple task-parallel strategy.

In detail, section 2 presents the FM-index data structure, and describes how several enhancements in its implementation (mainly the k-step strategy and our alternate-counters layout) lead to better performance. In section 3 we describe GPUs and the main determinants of their performance, focusing in particular on the relation between their memory system and the FM-index pseudo-random memory access pattern. In section 4 we discuss our proposal of increasing data locality by the use of a thread-cooperative approach. In section 5 we benchmark an implementation containing a combination of the optimisations proposed thus far, and present a comparison with the results of the FM-index implementation contained in the NVBIO library. Section 6 discusses related work and, finally, section 7 reports our conclusions and outlook.

2 FM-INDEX: BASICS AND OPTIMISATIONS

2.1 Basic definitions

2.1.1 Exact pattern matching

Let $R[1 \dots n]$ be a *reference* string over an alphabet Σ, where $R[i]$ is the i^{th} symbol of the string. $R[i \dots j]$ is a substring of R and $R[i \dots n]$ is a suffix of R starting at position i. Let $Q[1 \dots m]$ denote a *query* pattern, with $m \ll n$. Solving the *exact matching* problem is tantamount to finding all the occurrences of Q into R (i.e. the positions of all substrings of R that are equal to Q). Exact pattern search over a large reference string can be accelerated by making use of indexing data structures like the *suffix array* (SA) or the *FM-index*; the time spent by creating the index can be conveniently amortised whenever a large number of subsequent searches needs to be performed.

2.1.2 The Suffix-Array

The Suffix-Array of R, $SA[1 \dots n]$, stores the starting positions of all suffixes of R' in lexicographical order, R' being the original string R with an additional symbol \$ appended at the end. By convention \$ is taken to be lexicographically smaller than all other symbols in Σ. For example, if R'=$acaaacatat$\$ then SA=$[11, 3, 4, 1, 5, 9, 7, 2, 6, 10, 8]$.

We define the *SA interval* of a pattern Q as (l, h), being l and $h-1$ the ranks of the lexicographically-lowest and highest suffixes of R that contain Q as a prefix, respectively (the case $l=h$ indicates that Q does not occur in R). A binary search algorithm can compute the SA interval of $Q[1 \dots m]$ using $\log n$ steps of complexity $\Theta(m)$, and the $h-l+1$ occurrences of R can subsequently be obtained from SA.

2.1.3 Burrows-Wheeler Transform and FM-index

The *Burrows-Wheeler Transform* [10] of a string R, denoted BWT, is a permutation of the symbols of R'. Each value $BWT[i]$ stores the symbol immediately preceding the i^{th} smallest suffix: $BWT[i]$:=$R'[SA[i]-1]$. Hence if $R' = acaaacatat$\$, then BWT=tca\$$atcaaaa$.

BWT and two auxiliary data structures, $C[]$ and $Occ[]$, constitute the *Ferragina-Manzini or FM-index* [11] of R. $C[s]$ indicates the number of occurrences in BWT (or R) of symbols that are lexicographically smaller than symbol s. $Occ[s, p]$ counts the number of times symbol s appears in $BWT[1 \dots p-1]$.

The FM-index *backward search* (see Algorithm 1) computes the SA interval of $Q[1 \dots m]$ using m steps of complexity $\Theta(1)$, and without requiring R or SA. This is a remarkable improvement on the SA. The operation of computing $LF := C[Q[i]]+Occ[Q[i], l]$ is conventionally named *LF mapping*, standing for "Last-to-First column mapping" after a fundamental property of the BWT.

2.1.4 Sampled FM-index

A more realistic implementation of the FM-index differs from what has been presented so far in several respects. First of all, one usually stores only a small fraction of $Occ[]$ [11]: a reduced table $ROcc[]$ holds the values for positions that are multiple of a *sampling distance* d, with $ROcc[s, i]$=$Occ[s, i \times d]$. The remaining counters can then be reconstructed from the sampled counters and BWT. Parameter d introduces a trade-off between memory footprint and computational complexity: while $ROcc[]$ will be d times smaller than Occ,

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.2014.2377716, IEEE/ACM Transactions on Computational Biology and Bioinformatics

Algorithm 1: Exact pattern search using FM-index

input : *FM*: FM-index of reference R, Q: query, n: $|R|$, m: $|Q|$
output: (l, h): *SA* interval of Q in R
begin
  $(l,\ h) \leftarrow (1, n+1)$
  for $i = m$ **to** 1 **do**
    $l \leftarrow LF(FM,\ Q[i],\ l)$
    $h \leftarrow LF(FM,\ Q[i],\ h)$
  end
  return $(l,\ h)$
end

Algorithm 2: LF operation on sampled FM-index

input : *FM*: sampled FM-index, s: symbol, p: position in FM, d: sampling distance
output: p': new position in FM
begin
  $idx \leftarrow p/d$
  $offset \leftarrow p\ mod\ d$
  $entry \leftarrow FM[idx]$
  $cnt \leftarrow count(s,\ entry.BWT[0\ldots offset-1])$
  return $entry.ROcc[s] + cnt$
end

the m steps of the search algorithm will now have complexity $\Theta(d)$ each.

Second, memory locality can be improved by splitting the sampled FM-index into $\lceil n/d \rceil$ blocks of d consecutive *BWT* symbols (see figure 1.a). In each block one finds both the bitmap representation of the symbols, encoded in $d \times \log_2 |\Sigma|$ bits (named *BWT* in the figure), and $|\Sigma|$ associated counters (*ROcc* in the figure). $C[]$ is memoised into *ROcc* to save operations. As in most implementations the string terminator \$ is not encoded; instead, its position in *BWT* is stored and checked whenever an LF mapping is performed.

Third, counters can be arranged into memory-aligned *index entries* (see for instance [12]). We select sampling distances d such that entry sizes are an exact multiple of 32 Bytes, i.e. the size of a typical cache line. In particular a DNA string (with 4 bases `A`, `C`, `G` and `T`) of up to 4 Gbases will require $|\Sigma|$=4 32-bit counters (or 16 Bytes) per entry. Then for instance a 32-Byte entry will contain 16 Bytes=128 bits=2×64 bits of bitmaps, that is d=64 encoded symbols. We mention in passing that production setups might require alphabets with more than 4 symbols and counters larger than 32 bits; such considerations, however, do not affect the conclusions presented in this paper.

Algorithm 2 illustrates an LF operation on a sampled FM-index. The *count*() function can usually be implemented in terms of the fast bit counting instructions available on current processors. As an additional optimisation, in our searches we obtain the *SA* interval for the first 8 symbols of each query via a single direct access to a table of modest size (256 KB). The usual search on the FM-index restarts from the 9^{th} symbol of the query on.

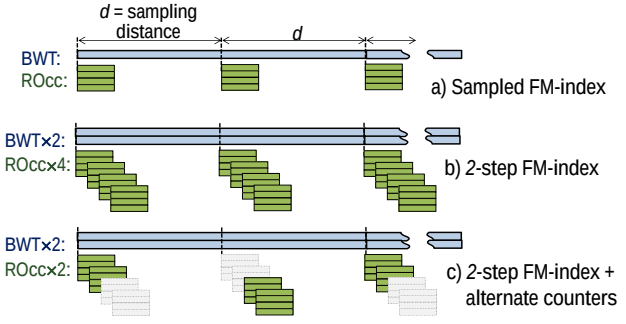

Fig. 1: Layouts of our FM-indexing strategies

2.2 *k*-step FM-Index

In [8] we proposed to use a generalised BWT, denoted *k-BWT*, that allows backward searches to be performed in steps of k symbols at a time. *k-BWT* is made of k strings, each string being the BWT of the text R' starting at a different offset: the i^{th} position of the j^{th} string is computed as *k-BWT*$[i][j]$:= $R'[SA[i]-j]$. For example, the 2-*BWT* transform of $R' = acaaacatat\$$ is $\{tca\$atcaaaa, aactaaa\$atc\}$.

A search step groups k consecutive symbols $s_1 \cdot s_2 \ldots s_k$ of Q from alphabet Σ and generates a new symbol $s_{1..k}$ from the alphabet Σ^k. The *k-FM* index includes the bitmap representation of *k-BWT* and the vector $ROcc[s_{1..k}, i]$ that counts the number of times that $s_{1..k}$ appears in *k-BWT*$[1 \ldots i \times d - 1]$. The LF operation on *k-FM* is exactly the same as that depicted in Algorithm 2, but this time using a symbol from a larger alphabet and larger data structures. A corner case happens when the last search step involves less than k symbols, say r. The solution is to aggregate all the *ROcc* counters matching with the r initial symbols of $s_{1..k}$, and counting occurrences on *k-BWT* ignoring the last k-r symbols.

Each entry of *k-FM* contains $|\Sigma|^k$ *ROcc* counters and $k \cdot log_2|\Sigma|$ bitmaps of size d. While the size dedicated to bitmaps still grows linearly with k, the size dedicated to counters now grows exponentially with k. Figure 1.b shows the memory layout of a 2-step FM-index.

The computational cost of each search step increases in a fashion proportional to k: at each step k times more bits are read from memory and need to be counted, and the \$ symbol needs to be checked in k positions. However, since the number of search steps is reduced by k the total amount of computational work performed per search, and that of data read, remains almost the same. The advantage of the proposal comes from the fact that in the case of an application

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.2014.2377716, IEEE/ACM Transactions on Computational Biology and Bioinformatics

that is already bounded by random memory accesses larger blocks can be read from memory almost for free.

This work analyses the k-step design only for k=2. For typical mammalian genomic references like the human one, larger values of k require memory footprints that exceed those typically available on current GPUs. However, future improvements in the technology might make the case k>2 practical.

2.3 Alternate counters

As we will point out in the next section, the performance of random accesses drops for excessively large memory footprints. Larger sampling distances reduce the size of the FM-index, but also increase the computational work. We propose another way of reducing the memory footprint at the cost of a small increase in computation, i.e. by dispensing with half of the counters. More in detail we use alternate counters as depicted in figure 1.c: odd FM entries contain *ROcc* counters for the first half of the symbols, while even FM entries contain counters for the second half of the symbols.

Algorithm 3 illustrates an LF operation on a k-step sampled FM-index with alternate counters. s is an input symbol that concatenates k original symbols. Depending on whether the identifier for the index entry is odd or even, and on whether s belongs to the first or second half of the symbols, the operation is performed as usual. Otherwise, the counters of the next FM entry must be used, and the symbols in the BWT bitmaps must be counted backward. Counting forward or backward has the same computational cost, and the extra access to a contiguous FM entry is often free, given the performance behaviour of random accesses.

Algorithm 3: LF operation using alternate counters

input : *FM*: k-step FM-index, s: symbol, σ: $|\Sigma|^k$, p: position in FM, d: sampling distance
output: p': new position in FM
begin
  $idx \leftarrow p/d$
  offset $\leftarrow p \bmod d$
  $entry \leftarrow FM[idx]$
  if $((s < \sigma/2) == even(idx))$ **then**
    $cnt \leftarrow$ *count-k*$(s, entry.BWT[0 \ldots offset\text{-}1])$
    return $entry.ROcc[s \bmod (\sigma/2)] + cnt$
  else
    $nextEntry \leftarrow FM[idx+1]$
    $cnt \leftarrow$ *count-k*$(s, entry.BWT[offset \ldots d\text{-}1])$
    return $nextEntry.ROcc[s \bmod (\sigma/2)] - cnt$
  end
end

Figure 2 compares the memory footprints of the different indexing schemes so far proposed for several values of the sampling distance d. More compact indexes come at the expense of additional computation; however, this is usually not a major concern on the GPUs, as there the FM-index search algorithms are typically memory-, and not computation-, bound.

3 Determinants of GPU performance

3.1 Background

Since its release in 2006, CUDA has become the most popular architecture for general-purpose GPU computing. The CUDA programming model defines a computation hierarchy formed by *kernels*, *thread blocks*, *warps*, and *threads*.

Warps are fixed size sets of threads (currently set to 32) that advance their execution in a lockstep synchronous way. They can be considered execution streams of vector (SIMD) instructions, where a thread represents a single lane of the vector instruction. Warp instructions are the smallest scheduled work units, and usually GPUs execute all the 32 operations in a warp simultaneously. However, control flow divergence among the threads in a warp causes the sequential execution of the divergent paths, and hence it must be avoided.

A thread block contains multiple warps that are executed independently. The warp instructions from multiple blocks are scheduled for execution on a vector processing unit called streaming multiprocessor (SM). The excess of parallelism expressed in terms of more warp instructions than the available computation resources helps alleviating the operation latencies.

The unit of work sent from the CPU to the GPU is called a kernel. The CPU can launch for parallel execution several kernels, each being composed by tens to millions of blocks. The blocks are scheduled for independent execution on multiple SMs.

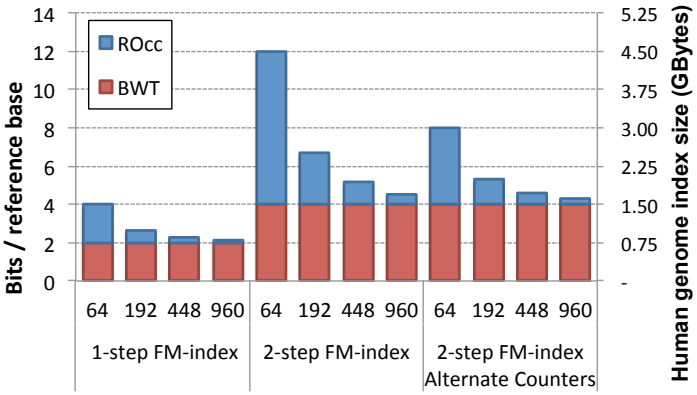

Fig. 2: Memory footprint of our FM-index implementations for different indexing schemes and sampling distances d= 64, 192, 448. The left Y-axis represents the bits/base ratio, while the right Y-axis shows the index size for the human genome.

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.2014.2377716, IEEE/ACM Transactions on Computational Biology and Bioinformatics

The GPU memory is basically organised as three logical spaces: global, shared, and local. The global memory is shared by all threads in a kernel and has a capacity of several GBs. It is located in the GDRAM of the GPU and the reuse of accessed data is exploited via on-chip cache memory. The shared memory is accessible by all warps belonging to the same block, while the local memory is private to each thread and mapped to a set of registers. Registers have the highest bandwidth and lowest latency. The shared memory is slower than the registers, whereas the GDRAM has very high access latency and limited bandwidth.

3.2 Why the task-parallel approach fails

Independently searching billions of short DNA strings in a large genomic reference is a problem that can be solved by resorting to the simplest parallel programming pattern, the `map` pattern [13]: an elemental function is applied in parallel to all the elements of the input set, usually producing an output set with the same shape as the input. A straightforward `map` GPU implementation would make each thread read its input data, perform the elemental function, and generate the output data. While this *task-parallel* approach is very effective on multicore CPUs, it can be problematic on GPUs due to some of their exclusive architecture features:

1) Accesses to global memory must be *coalesced* to achieve high efficiency. Coalesced accesses occur when all the threads in a warp address memory positions belonging to the same memory blocks.
2) The ratio of available on-chip memory per executing thread is very small; for example, Nvidia Kepler and Maxwell architectures provide a ratio of just 24-32 and 128 Bytes per thread for the shared memory and register storage, respectively.

A task *working set* is the aggregate active data set that must be kept in memory during the task execution. Due to feature (1), a simple task-parallel approach is inefficient on the GPU when each single task has to access a relatively large amount of input or output data. When a GPU task working set becomes large, on the other hand, due to feature (2) one has to face two possible performance problems. If the working set is placed on local registers or the shared memory, the excessive capacity requirements will ultimately reduce the maximum number of threads being executed in parallel (defined as the *GPU occupancy*), thus exposing the latencies of the computation. If the working set is placed in global memory, then the on-chip L2 cache will probably be overflown, and a high GDRAM traffic generated. While the latter effect is similar to what happens on the CPU, its relevance on the GPU is much bigger due to the much bigger number of threads involved.

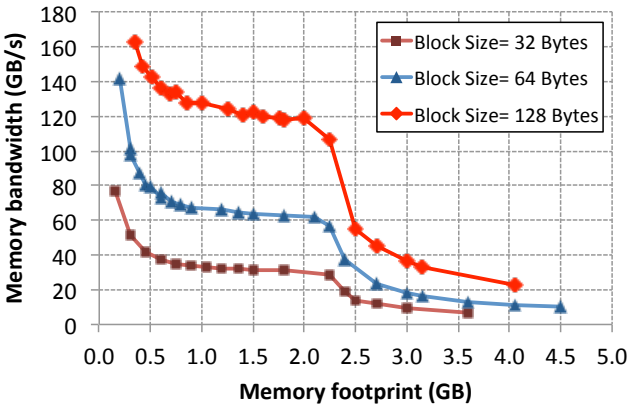

Fig. 3: Memory bandwidth for random accesses on the Titan GPU (6GB GDRAM)

3.3 Performance of random memory accesses

GPUs provide very high memory access bandwidth (more than 200 GB/s) for sequential coalesced accesses along relatively large contiguous portions of memory. However, performance suffers very much when a program accesses relatively small data blocks located at random memory addresses. Unfortunately, the FM-index search algorithm happens to show a pseudo-random hash table-like memory access pattern [14]. In fact, the FM-index search can be described as a loop that successively (1) loads a memory block from a given memory address, and then (2) calculates the address of the next needed memory block using the data just read. The generated set of memory addresses is fairly unpredictable, and uniformly distributed along the whole memory footprint.

On the top of that, the empirical GPU memory performance is far from being easily predictable. Figure 3 depicts the peak memory bandwidth achieved by our best GPU FM-index implementations (with coalesced accesses) on our test machine, for different block sizes and index sizes (memory footprint). Two main results can be read from the plot:

- *Large blocks are free*: accessing small blocks (32 Bytes) at random positions achieves suboptimal bandwidth; one can read larger blocks at the same cost without saturating the memory system.
- *Memory footprint size matters*: performance drops heavily as accesses are scattered along a large memory region; two clear inflection points exist for memory footprints of 0.5 GB and 2.5 GB. In particular, the 2.5 GB threshold cannot be easily explained by any architectural feature documented by the manufacturer, albeit it seems to appear on several GPUs (see section 5.4). A plausible explanation for this behavior might be the undisclosed existence of TLBs on the GPU, which has been put forth for instance in [15].

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.2014.2377716, IEEE/ACM Transactions on Computational Biology and Bioinformatics

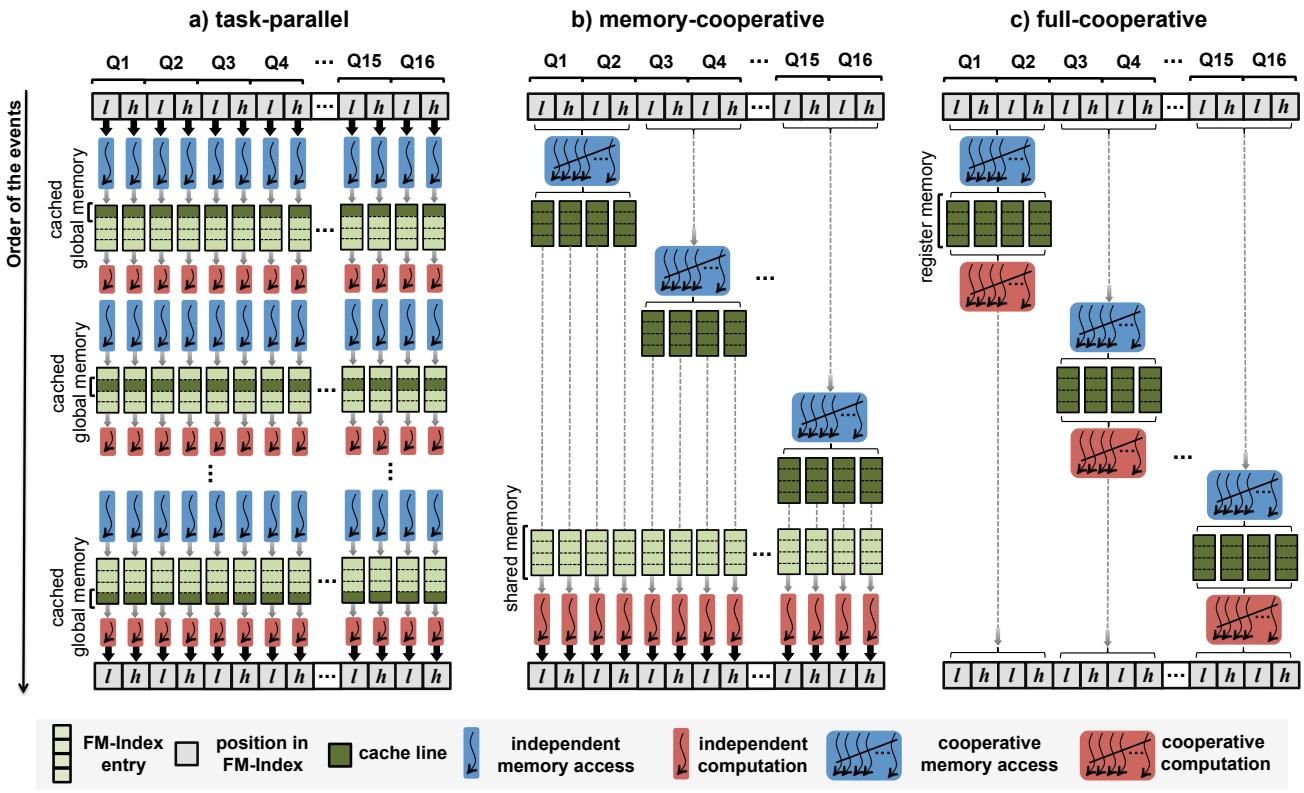

Fig. 4: GPU parallelisation alternatives: a) **task-parallel**: each thread performs independent LF operations; b) **memory-cooperative**: threads cooperate on reading data from index; and c) **full-cooperative**: threads cooperate both on reading data and on counting symbol occurrences. Each search step comprises 16 queries; the case d=448 is considered. We depict all the 32 threads in a warp participating in the execution of 32 LF operations. Memory read operations are shown in blue, and computation on the data (basically, counting symbols) in red.

4 Designs for GPU FM-index search

4.1 Task-parallel designs

As mentioned before, most published GPU FM-index implementations are based on straightforward task-parallel approaches, where each task corresponds to searching a different query in a shared FM-index. Each GPU thread independently processes a task, and operates on a complete SA interval, both on the l and h positions. The performance of the task-parallel scheme is suboptimal due to the 32 threads of a warp requesting data words from different scattered memory locations; this access pattern forces the GPU to re-issue the load instruction for each non-coalesced memory block, making the L2 cache the main performance bottleneck.

As shown in [7] and in the benchmark section, a simple way of enhancing this design can be achieved by using two separate threads to operate on each SA interval; each thread applies LF operations to either the previous l or the previous h position of the interval. Most of the time the extrema of the SA interval of a query are mapped to the same index entry and hence half of the threads in a warp are requesting the same data as the other half. This improves the coalescing process performed by the GPU when handling the memory requests of a warp. Figure 4.a provides a representation of the improved task-parallel execution flow. In the figure, each thread executes multiple memory load instructions to read a full index entry from global memory and then executes the corresponding counting instructions.

In spite of those improvements, the task-parallel design is bound to become worse when each thread needs to read a larger memory block, as is the case for large sampling distances d or the k-step approach.

4.2 Memory-cooperative design

Again along the lines of [7], one can obtain some further improvement by using thread cooperation in order to coalesce multiple data requests of different distant blocks of memory. Figure 4.b shows the execution flow of a *memory-cooperative* design: the threads in the warp jointly request multiple complete index entries. The example depicted in the figure uses a warp of 4×8 threads to retrieve from memory 4 complete entries (of 128 Bytes each) with a single 16-Byte load instruction (the best performing option), for a total of 32×16= 512 Bytes. The process iterates (8

times in the example) to copy the 32 entries from main memory into shared memory. Finally, each thread can efficiently access the shared memory to read the data corresponding to its entry to perform the LF operation, avoiding the costly non-coalesced accesses to the GDRAM and L2 cache.

The main drawback of the memory-cooperative scheme is that all the index entries read by a warp must fit simultaneously into shared memory. A relatively large sampling distance d coupled with a k-step strategy puts pressure on the capacity of the shared memory, and may ultimately lead to a significant reduction of thread occupancy. Experiments shown in the next section reveal a severe performance degradation for index entries of 128 Bytes or larger.

4.3 Full-cooperative design

Using registers instead of shared memory helps improving the thread occupancy, but is not a scalable solution. Instead, following [7], a much better approach is to reduce the working set of each thread (and hence of the whole application) by making threads also cooperate on the computational part of the algorithm (counting symbol occurrences and generating the output *SA* intervals), not only on reading data. Figure 4.c presents the full-cooperative design: in the example shown there the threads belonging to a warp cooperate to read 4 index entries, and then process the entries to generate 4 outputs. As in [16], this approach allows adjusting the working set of each thread to a given target size with the objective of maximising actual GPU occupation. Comparing figures 4.b and 4.c we notice that the full-cooperative scheme must simultaneously keep only four index entries (512 Bytes) in fast memory instead of 32 (4096 Bytes). In other words, the granularity of the work assigned to each warp can be maintained constant even when the entry size is increased.

Since all cooperative operations proposed in our design are performed at the warp level, there is no need of costly explicit synchronisation: through *shuffle* instructions Kepler and later CUDA architectures provide support for cooperating at the register level, which is faster and more efficient than cooperating using the shared memory. In detail, based on *shuffle* instructions we implemented the following communication patterns:

1) Multicast among threads l and h values
2) Generate a single cooperative memory load
3) Multicast among threads the symbol that must be applied to an LF operation
4) Parallel symbol counting by all threads
5) Parallel reduction of partial counters by groups of threads
6) Parallel gather of results.

A drawback of the cooperative design with respect to a task parallel one is that it increases the amount of executed instructions. As explained before, however, the choice of diverting part of the vast amount of computational power provided by the GPUs into solutions designed to improve memory performance happens to pay off in terms of the overall computational efficiency of the implementation.

5 Experimental Results

In this section we benchmark the exact searches performed with our implementations of the FM-index on one CPU and several GPU platforms. After presenting the experimental methodology, we describe the overall performance results. Then we present a detailed performance analysis of all considered solutions (task-parallel design, thread-cooperative design, 2-step approach and NVBIO library). Finally we examine how performance and energetic efficiency vary with the GPU model.

5.1 Experimental setup and methodology

The experimentation platform is a heterogeneous CPU-GPU node. The CPU is a dual-socket Intel Xeon E5-2650, with eight 2-way hyperthreaded cores per socket providing a memory bandwidth of 102 GB/s. Unless explicitly noted the GPU results shown in the following figures were obtained on our best card, a Nvidia GTX Titan with 2688 Kepler CUDA cores and 6 GB of main memory providing 288 GB/s. In order to perform comparative GPU analysis (section 5.4) we also used a Kepler K20 card and a Maxwell GTX750Ti card. Table 1 gathers all platforms hardware specs as declared by the manufacturer.

The input of our tests was a set of 10 million input DNA queries produced with widely used simulation tools ([17] and [18]) following standard procedures; input queries were searched in the human genome reference GRCh37. Experiments that test the effect of smaller reference sizes use a trimmed down version of the same genome. Before starting measurements we always made sure that the FM-index and the reads were already residing in the CPU or GPU memory. Performance results are expressed in terms of the number of query bases searched in the index per time unit.

Our multicore CPU implementation uses 16×2 threads (via OpenMP) to exploit hyperthreading, and memory access is optimised by using prefetching instructions. The sampling rate d is set to 64 for the CPU. Our GPU implementations set thread -number and -block sizes to values providing the highest performance.

The Nvidia NVBIO library [9] contains a suite of components to build new bioinformatics applications for massively parallel architectures. It offers methods for performing exact searches (via the `match` primitive) on a sampled FM-index both for GPU and CPU. On the GPU a different search will be

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.2014.2377716, IEEE/ACM Transactions on Computational Biology and Bioinformatics

TABLE 1: Hardware specifications of the experimentation platforms

	Architecture	Cores	Hardware threads	Frequency (Ghz)	Bandwitdh (GB/s)	Main Memory (GBytes)	TDP (watts)
Nvidia Kepler K20	2nd Kepler	2496	26624	0.71	208	5	225
Nvidia GTX Titan	2nd Kepler	2688	28672	0.84	288	6	250
Nvidia GTX 750Ti	1st Maxwell	640	10240	1.02	88	2	60
2×Intel Xeon E5-2650	Sandy Bridge	16	32	2.00	102	256	190

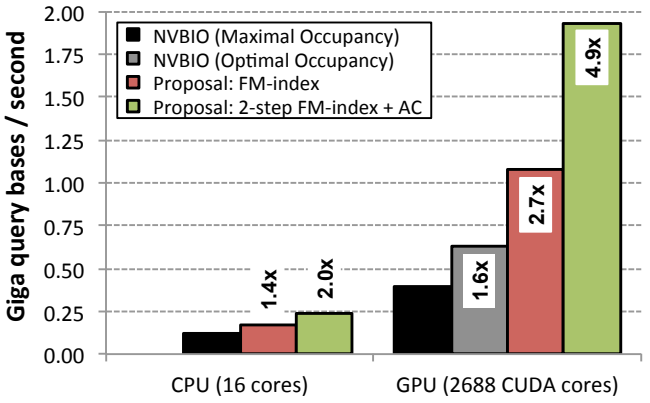

Fig. 5: Best FM-index search performance results on CPU and GPU

executed by each of the threads in a kernel using a task-parallel approach. For the NVBIO code we have selected the best-performing FM-index configuration, with a sampling distance d=64 and a decoupled SA (partial FM-index). In some tests we also adapted the GPU implementation of NVBIO to control the thread occupancy, as explained below. The NVBIO library version 0.9.7 used in our experiments was compiled with release-mode settings. Code for CPU was generated with GCC version 4.8, and code for GPU with Nvidia compiler v6.0.

5.2 Overall performance results

Figure 5 summarises the main results of our experiments, where we benchmark exact searches in the human genome (size 3 Gbases). The results presented correspond to the best-performing configuration for each implementation, both in the case of NVBIO and of our proposals (1-step, and 2-step with alternate counters). For comparison purposes we also include the timings achieved by the NVBIO code after the improvements we obtained by tweaking it: configuring NVBIO with a surprisingly low thread occupancy (9%) improves performance about 1.6× with respect to a configuration with maximal occupancy. We'll show next that this is due to its underlying task-parallel design.

The figure shows a clear speedup of our best proposals compared to the NVBIO library, both on the CPU (2.0×) and on the GPU (3.1× when considering the tweaked NVBIO code, and 4.9× versus the stock version of NVBIO distributed by NVIDIA).

The 2-step design outperforms the simple FM-index by 1.8× on the GPU and by 1.4× on the CPU. As previously discussed, the moderate speedup on the CPU is due to the higher computational cost of the new design. However, such increased cost has a very limited impact on the GPU, where excess computational power is available to be used.

Finally, our best-performing implementation on the GPU (which in absolute terms delivers almost 2 Gigabases of query searched per second) is 8.1× faster than our best implementation on the CPU. Interestingly, this speed-up is higher than the ratio of the raw bandwidths for sequential memory access delivered by the two platforms (which is around 3 times faster on the GPU than on the CPU). This fact confirms that our implementation strategies aimed at obtaining a better performance for random memory accesses are particularly effective on the GPU.

5.3 Performance analysis

5.3.1 Task-parallel versus cooperative schemes

In this section we analyse the inefficiencies of the task-parallel (section 4.1) and memory-cooperative strategies (section 4.2) as opposed to the full-cooperative solution (section 4.3). Since the 2-step FM-index implementation exhibits a similar behaviour, we restrict our analysis to the classical sampled FM-index.

Performance versus number of active threads

In figure 6 we benchmark the cooperative version, three different implementations of the task-parallel scheme, and the NVBIO implementation, showing their performance as a function of the number of threads used. The naive task-parallel versions assign two LF operations to each thread, both using 4-Byte ("naive") and 16-Byte memory accesses ("naive+16 Bytes"). The task-parallel approach labelled as "improved" uses 16-Byte memory loads and assigns a single LF operation per thread, which is a limited form of cooperation (see section 4.1). In all cases we use a sampling distance d=64, which is the most favourable for the task-parallel strategy.

The task-parallel schemes exhibit the problems anticipated in the previous section: performance first increases with more active threads, and then suddenly drops and flattens. The performance peak is

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.2014.2377716, IEEE/ACM Transactions on Computational Biology and Bioinformatics

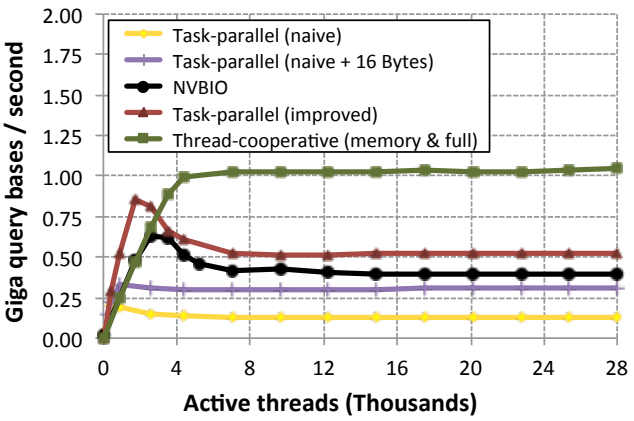

Fig. 6: Thread scalability for our proposals (sampled FM-index with d=64) and the Nvidia NVBIO implementation.

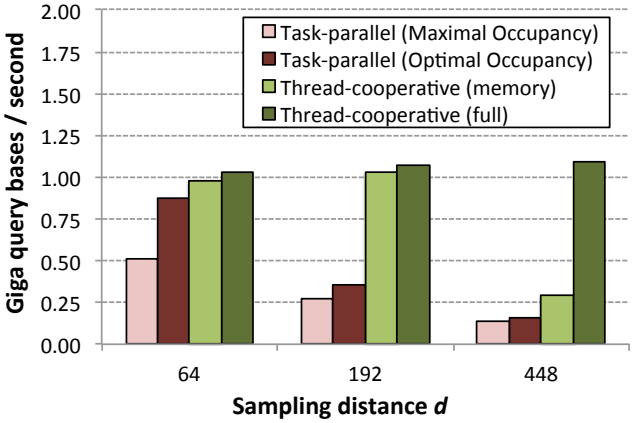

Fig. 7: Performance of task-parallel (both with maximal and optimal thread occupancy) and thread-cooperative designs for increasing sampling distance d.

located at some specific, relatively small number of active threads. The NVBIO implementation, which also uses a task-parallel approach, suffers from the same problem. On the other hand the performance behaviour of our cooperative version is very robust, scaling gracefully up to 4 thousand active threads. Eventually, as more threads are executed, a larger number of requests are issued that end up saturating the memory system.

The origin of the observed behaviour cannot lie in the GDRAM system, since searches in small indexes that fit into the L2 cache and require no data transfer from GDRAM show the same performance anomaly (data not shown). Instead the reason is due to the fact that the data transfer mechanism between the L2 cache and the executing units is strongly optimised to favour spatial locality and coalesced accesses, and its scarce temporary storage becomes easily saturated when many threads compete to request data from the L2 cache. This is why the implementation issuing 16-Byte loads performs better than the one issuing 4-Byte loads. Consistently, the "improved" version achieves better results because it issues less load instructions.

Performance versus sampling distance d

Figure 7 shows the performance of the proposed parallelisation schemes as a function of the sampling distance d, displaying the best results for each case. The task-parallel scheme is only competitive for d=64; larger values of d worsen the problem of non-coalesced accesses. Similarly, the memory-cooperative scheme does not scale to d>192, as the shared memory capacity becomes exhausted by the requirements of too many threads. For d=448, the GPU occupancy is 12% of the maximum number of active threads, and there is not enough parallelism to hide memory latencies. As expected, the full-cooperative design outperforms the other two in all cases.

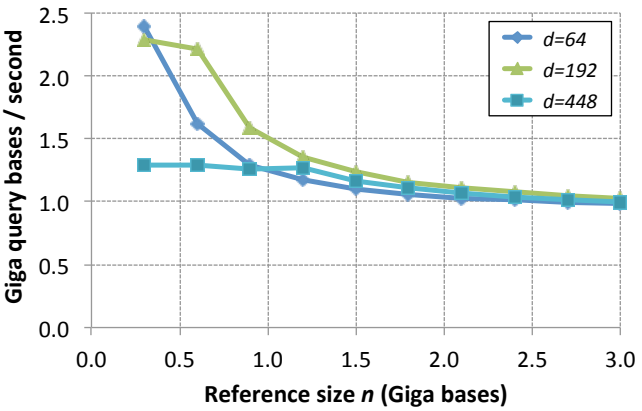

Fig. 8: Performance effect of varying reference size n and sampling distance d on FM-index.

5.3.2 *Performance versus reference size*

In this section we analyse the empirical dependence of the FM-index on the size of the reference (which in turn stems from the empirical performance of random memory accesses on the GPU seen in section 3.3). Then we examine how the combination of our thread-cooperative design (section 4.3) and our k-step indexing strategy (section 2.2) can lead to the best performance results shown in section 5.2.

Classical 1-step sampled FM-index

As mentioned in section 2.1.3, in theory the complexity of the FM-index search is independent of index size n. Quite to the contrary, figure 8 shows that in practice index size is a relevant parameter for the classical *1*-step sampled FM-index of section 2.1.4: when the index size grows, performance decreases for all the values of sampling distance d. This effect is due to the underlying performance of random memory accesses on the GPU, and is a direct consequence of figure 3.

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.2014.2377716, IEEE/ACM Transactions on Computational Biology and Bioinformatics

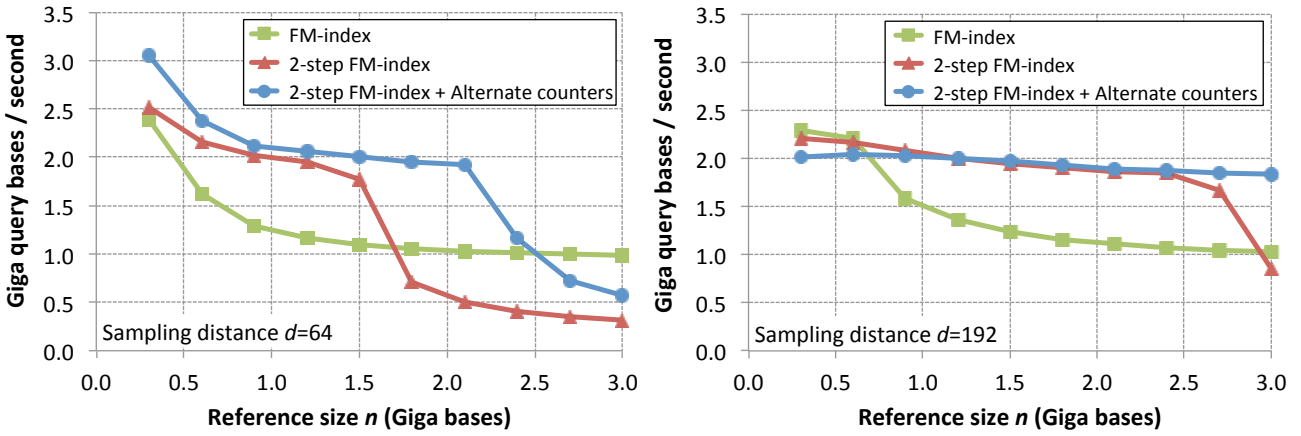

Fig. 9: Performance effect of varying reference size n and sampling distance d on different indexing schemes

For large indexes (n bigger than 1.5 Gbases) performance reaches saturation due to random memory accesses, leading to similar results for all the three compression ratios analysed. In other words, one can compress the index (as per figure 2) without any performance penalty, as described in more detail in [8]. In fact, for some genome sizes a more compressed index also provides better performance: for instance, for n=0.7 Gbases, the choice d=192 is better than d=64. In contrast, for smaller indexes where n is below 0.5 Gbases the performance is always better for smaller values of d. This happens because in this case the search is always computation-bound (data not shown).

2-step FM-index and alternate counters

The k-step strategy trades reading less blocks for reading bigger blocks, and also benefits from the first performance principle seen in section 3.3: large blocks are free for random-access memory patterns. Its drawback, though, is a larger memory footprint that can be detrimental if the index size goes beyond the empirical limit of 2.3 GB (see fig. 3). For instance, this is what happens in the case of the human genome when a 2-step approach with distances d=64 or 192 is used: the indexes thus generated will require 4.5 and 2.5 GB, respectively. However, the use of alternate counters reduces the index sizes to 3 GB and 2 GB, respectively, thus restoring the efficiency of the choice k=2, d=192.

Figure 9 allows us to compare performance for different reference sizes, sampling distances and FM-index configurations. According to our previous observation, the performance drop of the 2-step configurations occurs when the index size exceeds the 2.3 GB limit. In addition, the configuration k=2, d=192 and alternate counters turns out to be the best option for references larger than 1 Gbases and smaller than 3.5 Gbases. For bigger references, the 2-step design generates an index that is too large. For references smaller than 1 Gbases the GPU provides higher memory bandwidths and the execution may become computation-bound, similar to what happens for the case k=1; the most effective solution for such reference sizes is to reduce the compression rate in order to reduce the computational burden.

5.3.3 Computational cost

Table 2 measures the computational cost of our indexing schemes. The number of executed instructions collected from our benchmarks confirms that indeed the computational cost of the cooperative design on the GPU is significantly (2.5×) higher than that of the task-parallel design. Also, the cost of compressing the FM-index (higher d) grows as expected from the definition of section 2.2: doubling the entry size doubles the number of instructions (and the amount of Bytes read from memory). Finally, the last two rows of table 2 show that with respect to the 1-step strategy the 2-step strategy incurs only a moderate computational overhead (10% to 18%), which is negligible for large indexes but can be detrimental for short indexes. Overall, the complete lack of correlation between the entries of this table and the corresponding performance values confirms the predominant role played by memory effect when exact searches are performed on the FM-index.

5.4 Comparison of GPU architectures

In this section we want to describe how the performance of the proposed algorithms varies on three different GPUs: two Kepler cards (GTX Titan and K20c) and a recent Maxwell card (GTX 750Ti).

TABLE 2: Warp instructions executed per query base

Sampling distance	d=64	d=192	d=448
Task-parallel FM-index	3.08	6.15	12.14
Full-cooperative FM-index	7.68	15.63	31.43
2-step FM-index	8.49	17.70	35.10
2-step FM-index + alt. counters	8.89	17.10	37.10

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.2014.2377716, IEEE/ACM Transactions on Computational Biology and Bioinformatics

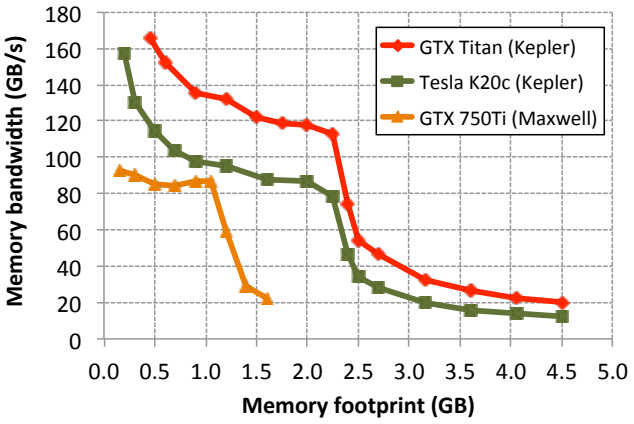

Fig. 10: Performance of random memory accesses for different GPUs (index entry size is 128 Bytes)

First of all, in figure 10 we extend figure 3 and compare the random memory access bandwidth of the three cards. Quite surprisingly the largest bandwidth is provided by the commodity GTX Titan; the professional Tesla K20c shows a similar performance profile, but with about 30% less performance. In particular, at 2.3 GB the two cards share the same sweet spot that maximises the product of bandwidth and memory footprint. The low profile Maxwell card gets its maximum throughput with 1GB memory footprints. As its cost and power consumption are only a fraction of those of the other professional GPUs, this card can still be appropriate for small genomes, or to process bigger genomes on multiple cards.

In figure 11 left panel we compare the performance of the proposed algorithms for the case of the human genome. The Titan and K20c GPUs show similar performance profiles for the different algorithms; since the search algorithm is memory-bound, the observed throughput reflects well the memory bandwidth profile of each GPU depicted in figure 10. On the other hand, on the GTX 750Ti the human genome can be indexed only with k=1 due to the smaller memory size available. In spite of this fact, the performance is still quite good (only 40% worse than that of the k=1 version on the Titan, and about 2.5× worse than that of the best 2-step version on the Titan). However, when comparing the nominal energetic efficiencies (figure 11 right panel, obtained from the performances and table 1) one notes that the GTX 750Ti stands out among all other platforms in terms of the number of queried bases/joule. Compared with the CPU, the GTX 750Ti has 8.5× better energetic efficiency, while still providing a 2.6× better performance. All the GPUs considered in this study are far more energetically efficient than the CPU (from 4.8× to 8.5× if the best implementations are considered).

We conclude that while the profile that correlates index footprint size and memory bandwidth for random accesses varies on different GPUs, it will anyway be one of the strongest determinants of the performance of our best FM-index search implementation. Hence it will be necessary to adapt the indexing scheme to the target GPU and the reference genome of interest. Luckily, in our framework performance can be easily optimized by selecting suitable values for parameters k and d. We anticipate that this feature of our algorithm is going to be more and more relevant for the forthcoming GPU systems.

6 Related Work

Over the last few years many CPU short-read mappers have been developed, like Bowtie [4], BWA [1] or GEM [5] to name a few. There have been many attempts to use GPUs for computational genomics (reviewed in [19]) and read mapping in particular (see for instance [20], [21] or [22]). In this context, the impact of pseudo-random memory access patterns on the GPU was well described in [14].

Several GPU implementations of the FM-index have been developed as the core component of fully fledged short-read mappers. Some examples are provided by CUSHAW (see [2], [12], [23]), BarraCUDA [6] and SOAP3 (see [24], [3]). Apart from the fact that they all use a task parallel approach to query the index, those implementations are quite sophisticated: for instance, SOAP3-dp [3] is based on a one-level sampling FM-Index with 64-Bytes aligned block entries, has an interleaved layout of input sequences to improve coalescing memory accesses, implements an overlapped CPU/GPU algorithm and determines at runtime the sampling rate to take full advantage of the memory available.

It is important to note that performing meaningful comparisons between a "pure" FM-index implementation and implementations embedded into more complicated applications is extremely difficult. When considering realistic short-read mapping setups, not only is it very hard to determine the time that each program spends querying the index; one should also consider that index access will be coupled with other calculations, which makes benchmarks inherently blurry. This is why in this paper we do not directly compare our method with FM-index implementations used by short-read mapping programs. However, all such implementations published so far share the same task-parallel approach, and hence they are all likely to benefit from the ideas and techniques presented in this paper.

Finally, the work described here stems from several preliminary studies we performed in the past. We had already proposed and evaluated the k-step generalisation of the FM-index algorithm for the CPU in [8]. How to engineer GPU performance by reducing the working set of the application and selecting the appropriate granularity of the work assigned to each thread in a warp had been described in [16], and

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.2014.2377716, IEEE/ACM Transactions on Computational Biology and Bioinformatics

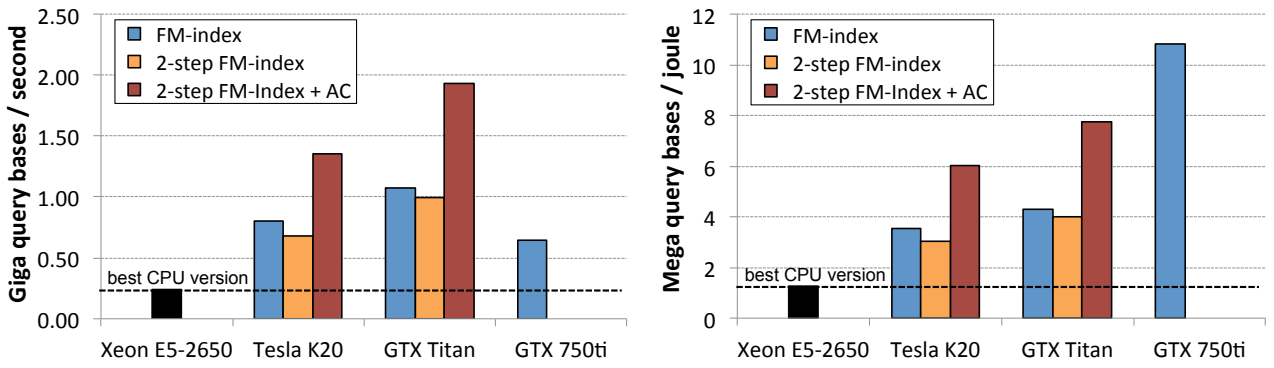

Fig. 11: Performance comparison (left) and energetic efficiency (right) of our thread-cooperative strategy on different CPU/GPU architectures

we had presented an early version of a cooperative scheme to reduce the memory footprint of the FM-index in [7].

7 Conclusions and Outlook

Technological improvements in memory performance are mostly achieved by incrementing the size of the data transfer bursts between main memory and the CPU/GPU. While this feature can greatly improve the performance of algorithms accessing large blocks of sequential data, it is neutral for algorithms requesting relatively small data blocks spread across distant random locations. In fact we are expecting to see that, in terms of efficiency, pseudo-random memory access patterns like those shown by straightforward FM-index implementations will steadily lag behind sequential access patterns even in upcoming next-generation memory systems. In such a scenario, the performance cost is determined by the total number of blocks accessed and not by the amount of data accessed. Therefore, we must favour algorithmic variations that access similar amounts of data but concentrated on less and bigger data blocks, even at the expense of more computation. This is precisely what our k-step FM-indexing strategy does: it trades reading less blocks for reading bigger blocks.

On the other hand, the working set granularity plays a crucial role in GPU performance. In fact, a simple task-parallel approach to FM-indexing is inefficient because the addition of more threads will turn into a larger and larger working set. However, when threads cooperate on a single task the working set is distributed among the cooperating threads. This allows us to efficiently process the bigger index entries produced by the k-step strategy. The increase in computational cost due to cooperation has a limited impact on the GPU, where excess computational power is available to be used, and overall our solution turns out to be successfully trading more work for less memory accesses.

The combination of all those optimisations yields an easily tunable implementation that is able to process about 2 Gbases of queries per second on our test platform, being about 8× faster than a comparable multi-core CPU version, and about 3× to 5× faster than the FM-index implementation on the GPU provided by the recently announced Nvidia NVBIO bioinformatics library.

Some relatively straightforward future developments of our technique can be foreseen. First, we plan to build an automatic system that characterises a target GPU memory system, and then tunes the FM-indexing strategies in order to achieve a specified goal (either improving performance or achieving a given memory limit). Second, we intend to analyse the performance of our implementation on multi-GPU platforms, exploring the potential use of several energy-effective low-end GPUs to replace a high-end GPU. Finally, we will integrate the techniques described in the present article into the GEM mapper [5], leveraging the power of GPUs in order to provide a hopefully faster and more energy-efficient solution to the approximate string matching problem.

Acknowledgments

This research has been supported by MICINN-Spain under contract TIN2011-28689-C02-01. We would like to thank NVidia for donating K20c GPU cards used in this work.

References

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

[2] Y. Liu and B. Schmidt, "CUSHAW2-GPU: empowering faster gapped short-read alignment using GPU computing," *Design and Test of Computers*, 2013.

[3] R. Luo, T. Wong, J. Zhu, C.-M. Liu, X. Zhu, E. Wu, L.-K. Lee, H. Lin, W. Zhu, D. W. Cheung *et al.*, "SOAP3-dp: Fast, accurate and sensitive GPU-based short read aligner," *PloS one*, vol. 8, no. 5, p. e65632, 2013.

[4] B. Langmead, C. Trapnell, M. Pop, S. Salzberg *et al.*, "Ultrafast and memory-efficient alignment of short dna sequences to the human genome," *Genome Biol*, vol. 10, no. 3, p. R25, 2009.

[5] S. Marco-Sola, M. Sammeth, R. Guigo, and P. Ribeca, "The GEM mapper: fast, accurate and versatile alignment by filtration," *Nature Methods*, vol. 9, no. 12, pp. 1185–1188, 2012.

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

[7] A. Chacón, S. Marco-Sola, A. Espinosa, P. Ribeca, and J. C. Moure, "FM-index on GPU: a cooperative scheme to reduce memory footprint," in *Proceedings of the International Symposium on Parallel and Distributed Processing with Applications (ISPA 2014)*. IEEE, 2014.

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

[9] J. Pantaleoni and N. Subtil. (2014, Jun.) NVBIO: a library of reusable components designed by NVIDIA corporation to accelerate bioinformatics applications using CUDA. [Online]. Available: http://nvlabs.github.io/nvbio/

[10] M. Burrows and D. Wheeler, "A block-sorting lossless data compression algorithm," in *Technical Report 124, palo Alto, CA*, 1994.

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

[12] Y. Liu and B. Schmidt, "Evaluation of GPU-based seed generation for computational genomics using burrows-wheeler transform," in *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2012 IEEE 26th International*, May 2012, pp. 684–690.

[13] M. McCool, J. Reinders, and A. Robison, *Structured parallel programming: patterns for efficient computation*. Elsevier, 2012.

[14] C. Kim, J. Chhugani, N. Satish, E. Sedlar, A. Nguyen, T. Kaldewey, V. Lee, S. Brandt, and P. Dubey, "Fast: fast architecture sensitive tree search on modern cpus and gpus," *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, 2010.

[15] H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos, "Demystifying GPU microarchitecture through microbenchmarking," in *Performance Analysis of Systems & Software (ISPASS), 2010 IEEE International Symposium on*. IEEE, 2010, pp. 235–246.

[16] A. Chacón, S. Marco-Sola, A. Espinosa, P. Ribeca, and J. C. Moure, "Thread-cooperative, bit-parallel computation of Levenshtein distance on GPU," in *Proceedings of the 28th ACM International Conference on Supercomputing*, ser. ICS '14. New York, NY, USA: ACM, 2014, pp. 103–112.

[17] M. Holtgrewe, "Mason - A read simulator for second generation sequencing data," *Technical Report FU Berlin*, 2010.

[18] Y. Ono, K. Asai, and M. Hamada, "PBSIM: PacBio reads simulator—toward accurate genome assembly," *Bioinformatics*, vol. 29, no. 1, pp. 119–121, 2013.

[19] S. Aluru and N. Jammula, "A review of hardware acceleration for computational genomics," *Design and Test of Computers*, 2013.

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

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

[22] A. Drozd, N. Maruyama, and S. Matsuoka, "A multi GPU read alignment algorithm with model-based performance optimization," in *High Performance Computing for Computational Science - VECPAR 2012*, ser. Lecture Notes in Computer Science, M. Daydé, O. Marques, and K. Nakajima, Eds. Springer Berlin Heidelberg, 2013, vol. 7851, pp. 270–277.

[23] Y. Liu, B. Schmidt, and D. Maskell, "CUSHAW: a CUDA compatible short read aligner to large genomes based on the burrows–wheeler transform," *Bioinformatics*, vol. 28, no. 14, pp. 1830–1837, 2012.

[24] C.-M. Liu, T.-W. Lam, T. Wong, E. Wu, S.-M. Yiu, Z. Li, R. Luo, B. Wang, C. Yu, X. Chu *et al.*, "SOAP3: GPU-based compressed indexing and ultra-fast parallel alignment of short reads," in *3th Workshop on Massive Data Algorithms*, 2011.

Alejandro Chacón received from the Universitat Autònoma de Barcelona (UAB) a BSc degree in computer science in 2011, and a MSc in high performance computing and information theory in 2012. He is currently a PhD student in high performance computing at the UAB, working to implement on GPUs several algorithms that are the building blocks for a number of bioinformatic applications. His research interests include computer architecture and parallel optimizations for heterogeneous HPC systems.

Santiago Marco Sola received his BSc degree in computer engineering from the Universidad de Zaragoza in 2010, and a MSc degree in computer science from the Universitat Politècnica de Catalunya in 2012. In 2010 he joined the CNAG (the Spanish national sequencing center) as a PhD student in the Algorithm Development group. Currently he is also a lecturer at the Universitat Autònoma de Barcelona. His research interests include pattern matching algorithms, data compression, parallel and high performance computing.

Antonio Espinosa received his BSc degree in computer science in 1994 and his PhD degree in computer science in 2000. He is a post-doctoral researcher at the Computer Architecture and Operating Systems department at the Universitat Autònoma de Barcelona. During the last 10 years he has participated in several european and national projects related to bioinformatics, life sciences and high-performance computing, in collaboration with a number of biotechnology companies and research institutions.

Paolo Ribeca received his PhD in theoretical physics from Université de Paris Sud XI in 2003. His research interests always focused on the application of high-performance scientific computing to open research problems in physics, mathematics and biology. Since the inception of high-throughput DNA sequencing techniques he specialized on algorithms for short-read processing. Since 2010 he is leader of the Algorithm Development team at the CNAG (the Spanish national sequencing center) and since 2014 he is leader of the Integrative Biology group at the Pirbright Institute (the UK national animal virology center).

Juan Carlos Moure is an assistant professor at the University Autónoma of Barcelona (UAB), where he teaches computer architecture and parallel programming. He has a MS and a PhD on Computer Architecture from the UAB. His current research interests include massive parallel architectures, programming, and algorithms, mainly focused on bioinformatics applications. He is the author of more than 20 papers, including international journals and conferences.