FM-index on GPU: a cooperative scheme to reduce memory footprint

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

Abstract—The FM-index is a data structure which is seeing more and more pervasive use, in particular in the field of high-throughput bioinformatics. Algorithms based on it show a pseudo-random memory access pattern. As a consequence, they are usually bound by memory bandwidth rather than CPU usage. Naive GPU implementations are no exception. Here we show that the combination of a compact design of the FM-index and a thread-cooperative approach can be used to restore a proper balance. The resulting solution is less memory-bandwidth intensive, and allows full exploitation of the computational resources of the GPU across several GPU architectures.

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

1 Introduction

The success of gaming and graphics video industry has boosted the widespread use of cheaper and cheaper GPU cards; they offer excellent performance with high energetic efficiency. The key of their success is an architecture designed for achieving high throughput (not low latency) by exploiting massive parallelism from applications.

The main reason of using a many-core GPU is to get the most from its multiple sources of computational power. Hence, on the GPU algorithms should be optimized in such a way that performance is bounded by computation. This is achieved by promoting data reuse on the relatively small on-chip memory of the GPU, shared by thousands of running threads. On the other hand, if execution is memory-bound the arithmetic intensity (i.e. the ratio of instructions executed versus the amount of data accessed from GDRAM) will be low, resulting in suboptimal performance.

In general, the relatively high memory bandwidth of the GPU should be effectively exploited by: (1) generating enough memory-level parallelism (lots of memory requests on the fly) to tolerate high memory access latencies, and (2) promoting spatial locality so that the in-flight memory requests are concentrated on a reduced subset of blocks of consecutive memory locations.

This paper illustrates the GPU optimisation trade-offs by means of a case example that is extremely relevant in the field of bioinformatics. Searching for substring exact matches is on the core of many problems like sequence alignment, *de-novo* sequence assembly, and so on. Those problems need to process large amounts of genomic sequenced data and demand fast and efficient pattern matching algorithms. The most effective sequence alignment software tools, like BWA [13], CUSHAW2 [16], SOAP3 [18], Bowtie [12], and GEM [19], rely on a special data-structure called the *FM-index* [8] to store a genomic reference and efficiently perform exact searches on it. The cost of the search is linear in the length of the searched pattern, and (theoretically) independent on the size of the reference sequence. In addition, the index achieves high compression ratios, allowing to store the full 3 GB of the human genome into 1 GB of memory space.

*Universitat Autònoma de Barcelona, Bellaterra 08193, Spain.
†Centro Nacional de Análisis Genómico, Barcelona 08028, Spain.
{alejandro.chacon, antoniomiguel.espinosa, juancarlos.moure}@uab.es
{santiagomsola, paolo.ribeca}@gmail.com

The FM-index is often used on scenarios where millions of independent pattern searches must be performed on a common large reference sequence. In principle, this offers plenty of thread and memory-level parallelism for GPU implementation. However, the straightforward strategy used by all previous work [11][16][18] was to assign one independent search task to each thread, in what we call a *task-parallel* scheme. Such a scheme is fundamentally limited by the inefficient reuse of the data stored in the L2 cache caused by the large number of pseudo-random relatively small data accesses.

The contribution of the present paper can be summarised in three main points:

- We present a detailed performance analysis of the task-parallel FM-index search algorithm executed on GPU. The limited locality of random memory accesses and the working set granularity play a crucial role on the performance obtained.
- We propose a novel cooperative scheme among threads for the GPU implementation of the algorithm. Together with a variable-size implementation of the FM-index, this approach achieves performance scalability along different GPUs and can be used to reduce the index size with negligible impact on performance.
- We compare the results obtained in a multicore CPU and in different GPUs from the Kepler and the new Maxwell family to find that less expensive, energy aware Maxwell cards are very suitable target platforms for algorithms with random memory accesses.

In section 2 we are going to describe the FM-index structure and operation. Section 3 introduces our implementation of the FM-index. In section 4 we discuss our proposal to increase data locality by a thread-cooperative approach. In section 5, we provide benchmarks. Section 6 discusses related work and, finally, section 7 summarises past and future work.

2 Background

2.1 Exact pattern matching

Let $R[1 \dots n]$ be a *reference* string over an alphabet Σ, where $R[i]$ is the ith 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 $m \ll n$. The *exact matching* problem consists of finding all the occurrences of

Q into R (the positions of each substring of R that are equal to Q). Exact pattern search over a large reference string is accelerated by turning it into data structures like the *suffix array* (SA) or the *FM-index*; the time spent on creating the index is amortised when a large number of searches is performed.

2.2 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' is the original string R with an additional symbol \$, lower than all symbols in Σ, appended at the end. For example, let 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 rank of the lexicographically-lower and higher suffix of R that contains Q as a prefix (the case $l{=}h$ indicates that Q does not occur in R). A binary search algorithm computes the SA interval of $Q[1 \dots m]$ with complexity $\Theta(m \log n)$. Afterward the $h{-}l{+}1$ occurrences of R can be obtained from SA.

2.3 Burrows-Wheeler Transform and FM-index

The *Burrows-Wheeler Transform* [3] of a string R, denoted BWT, is a permutation of the symbols of R . Each value $BWT[i]$ stores the symbol immediately preceding the ith smallest suffix: $BWT[i]{:=}R[SA[i]{-}1]$.

BWT and two auxiliary data structures, $C[]$ and $Occ[]$, constitute the *Ferragina-Manzini or FM- index* [8] of R. $C[s]$ indicates the number of occurrences in BWT (or R) of symbols that are lexicographically lower than symbol s. $Occ[s, p]$ counts the number of times symbol s appears in $BWT[0 \dots p]$.

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 FM-index. 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.

Algorithm 1: Exact pattern search using the FM-index

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

3 FM-INDEX DESIGN AND PERFORMANCE

This section describes the implementation of the FM-index considered for this work. We also present a preliminary performance analysis to explore the effects of size and structure of the index.

3.1 Implementation of the FM-index

The naive implementation of the FM-index presented in the last section is not sufficiently flexible for our purposes. To introduce a parametrised trade-off between memory footprint and time, we modify the representation of the $Occ[]$ arrays. In particular, in the scheme employed for this paper only a small fraction of $Occ[]$ is maintained; the 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 be reconstructed, at greater computational cost, from the sampled counters and BWT. This reduces the size of $Occ[]$ by d while the search algorithm still uses m steps, now of complexity $\Theta(d)$.

The sampled FM-index is divided into blocks of d consecutive BWT symbols, together with their associated $ROcc$ counters. We improve memory access performance by grouping the data of each block into a single entry, $FM[i]$, stored in a contiguous chunk of memory. FM contains $\lceil n/d \rceil$ entries; each entry contains $|\Sigma|$ counters, denoted $ROcc[]$, and the bitmap representation of d symbols, denoted BWT, encoded in $d{\times}\log_2 |\Sigma|$ bits.

A DNA string (with 4 bases `A`, `C`, `G` and `T`) of up to 4 Gbases requires $|\Sigma|$=4 counters per entry, or 4×4=16 Bytes. We select sampling distances d so that entry sizes are an exact multiple of 32 Bytes (the size of a cache line). For example, a 32-Byte entry contains d=64 symbols encoded with 2×64=128 bits= 16 Bytes. This single-level scheme of counters arranged into aligned entries has been already proposed for the GPU, for instance in [15].

Algorithm 2 illustrates the modified LF operation.

Algorithm 2: Modified LF operation

```
input : FM: FM-index, s: symbol, p: position in F,
        d: sampling distance
output: p': new position in FM
begin
    entry ← FM[p / d]
    cnt ← count(s, entry.BWT[0...p mod d]) return
    entry.ROcc[s] + cnt
end
```

3.2 Computational analysis of LF operations

Exact pattern searching performs recurrent LF operations. At each step, two F entries are read from positions calculated from the input SA interval and then some computation is done with the entry's contents to generate the output SA interval. Due to the characteristics of BWT and the search process, memory accesses are randomly spread along the whole F data structure; for large reference strings most of the accesses miss the on-chip cache and data needs to be read from main memory.

Here we show the performance of two implementations of the modified FM-index just described. The first one is a CPU version running on a Nehalem architecture, where memory access has been optimized by using prefetch instructions. The second one is a straightforward GPU version using a task-parallel scheme to overlap memory latency. In both cases the execution is memory-bound.

Along this paper performance will be expressed in terms of LF operations executed per unit of time. This metric is theoretically independent on the reference and query size. In practice, Fig. 1.a shows that performance is higher for small reference sizes, when data is reused inside on-chip caches, and drops as the reference size increases, both on CPU and GPU. We can conclude from Fig. 1.a that the GPU reuses data more effectively than the CPU for index sizes lower than 500 million symbols. However, in this paper we focus our analysis on bigger references, like the human genome, which are more important in real-life applications.

Figure 1.b shows that searching very small patterns ($m \leq 10$) provides a moderate performance advantage, especially on CPUs. This is due to the temporal locality of the small portion of the FM-index that is effectively accessed with small queries. As queries get longer, the SA interval becomes very narrow for most part of the search process and both ends of the interval tend to point to the same index entry. This provides additional memory locality that explains why performance slightly improves for growing query sizes. Since performance is very similar for a large range of query sizes, we set our experiments to use a query size of m=32 symbols.

The sampling distance, d, varies the compression ratio of the FM-index and defines a trade-off between memory and computation requirements: the larger d, the smaller the size of F but the higher the number of memory requests and counting operations. Figure 1.c illustrates the reduction of the index size as d is increased.

4 Designs for GPU FM-index search

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*.

A thread represents a single lane of a vector instruction. Warps are fixed size sets of threads (currently set to 32) that advance their execution in a lockstep synchronous way. Warps are the smallest scheduled work units. GPUs can execute all threads in a warp simultaneously, most times as a single vector operation. Control flow divergence among the threads in a warp causes the sequential execution of the divergent paths, so it is commonly avoided. A thread block contains warps that are executed independently but can cooperate via synchronisation operations. Warps from multiple blocks are scheduled for execution on each SIMD processing unit called streaming multiprocessor (SM).

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

The GPU memory is organised in 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 device and the reuse of accessed data is exploited via on-chip cache memory. The shared memory is accessible by all warps in a block, while the local memory is local to each thread and is mapped to a set of registers. The Kepler and Maxwell architectures provide 48 KB and 64KB per SM for the shared memory, respectively, and 256 KB per SM for the registers. The L1 and L2 cache memories hold 128-Byte and 32-Byte lines.

The communication between the threads in a block is carried out via the shared memory, whereas threads in a warp can communicate their register contents using the shuffle instructions. 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. The registers and the shared memory provide flexible accesses, while the accesses to the global memory must be coalesced to achieve the highest efficiency.

4.1 Task-parallel designs

4.1.1 Naive task-parallel design

Most published literature is based on a straightforward task-parallel approach, with each task independently processing independent queries on a shared FM-index. Fig. 2.a provides a representation of the task-parallel execution flow. Each GPU thread executes multiple memory load instructions to read a full FM-index entry from global memory and executes the corresponding intermediate data counting. Memory load instructions generate data transactions between main memory and the on-chip L2 cache, and between the L2 cache

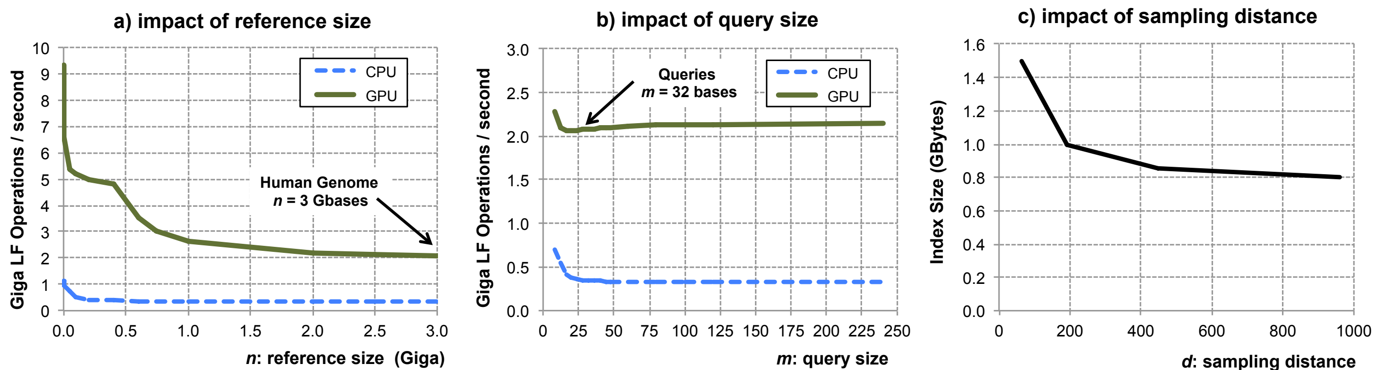

Fig. 1: Impact on performance –in Giga LF operations per second– of varying reference size n, and query size m; and impact on index size of varying sampling distance d

and the SM. For instance, as depicted in the figure, with d=448 the size of an index entry is 128 Bytes and requires a minimum of four main memory transactions of consecutive 32-Byte cache lines. Transactions between the L2 cache and the SM can be as large as 128 Bytes, the size of a L1 cache line. In addition, the 32 threads in a warp will be requesting data from very different memory locations; this non-coalesced access pattern wastes multiple execution cycles by re-issuing the load instruction.

Indeed our experimental results, presented in the next section, show that the main drawback of the task-parallel design comes from an inefficient reuse of the data accessed from GDRAM and stored in the L2 cache. The potential bandwidth of the L2 cache cannot be used efficiently with small requests and becomes the performance bottleneck.

4.1.2 Improved task-parallel design

Our first original proposal enhances the naïve solution 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. This design has 2 threads cooperating with a single query, and hence, as shown by our benchmarks, it already outperforms the straightforward implementation of the task-parallel scheme.

4.2 Memory-cooperative design

A more involved thread cooperation scheme can be used to improve data reuse even more, generating larger memory requests and better exploiting the scarce spatial locality of the algorithm. These memory-cooperative requests are called *coalesced* and are well-known and widely used for accessing large sequential portions of memory. The challenge for the FM-index search algorithm is to generalise coalescing for multiple, distant, relatively small blocks of memory; it requires a cooperation scheme that will be described in this subsection.

Figure 2.b shows the execution flow of the memory-cooperative design: the threads in the warp cooperate in groups to jointly request multiple complete index entries. The execution of a 16-Byte load instruction –the best performing option– simultaneously requests data from 32 memory addresses that accounts for 32×16= 512 Bytes. The example shown in Fig. 2.b requires 8 threads to retrieve a complete entry (128 Bytes) from memory. Therefore, a single warp requests 4 complete entries with a single load instruction. 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 accesses to the L2 cache.

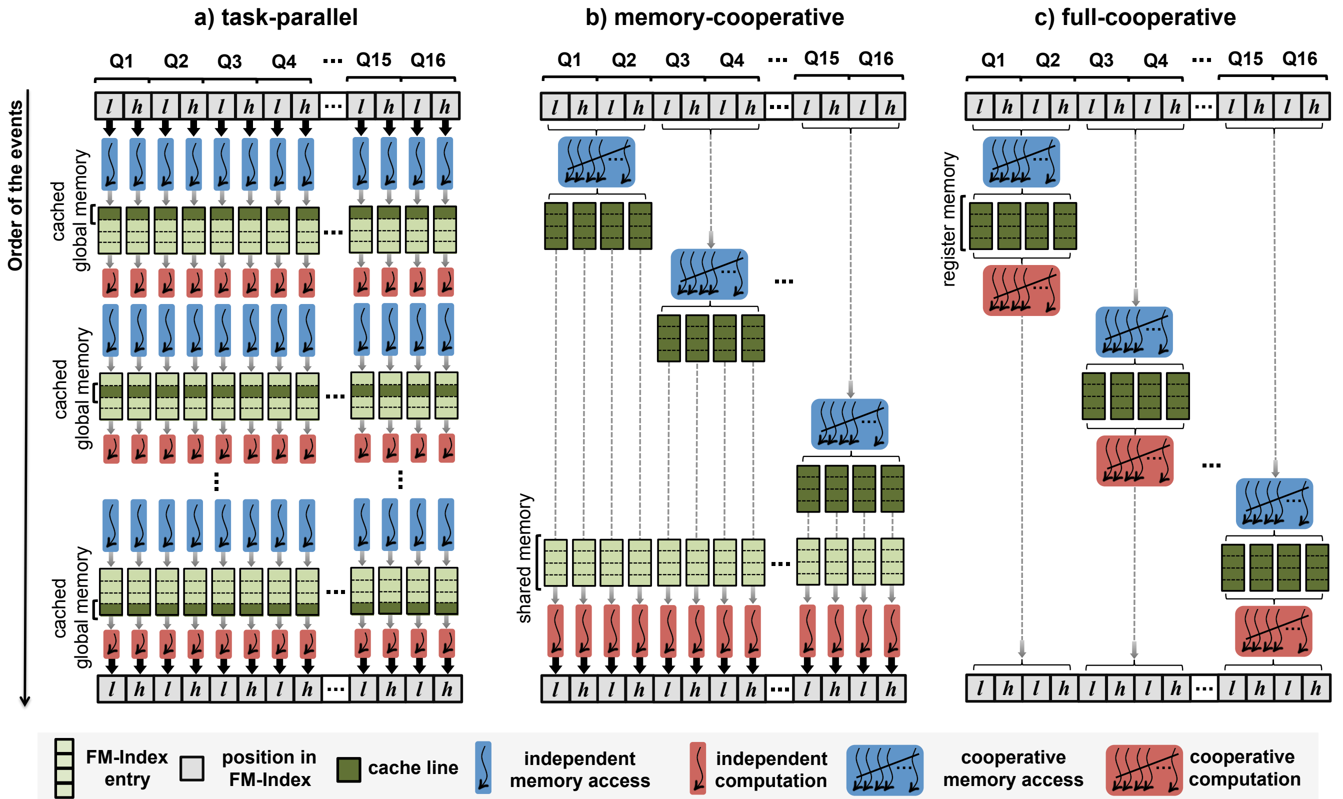

Fig. 2: GPU parallelisation alternatives: a) **task-parallel**: each thread performs independent LF operations; b) **memory-cooperative**: threads cooperate for reading data from index; and c) **full-cooperative**: threads cooperate both for reading data and for counting symbol occurrences. Each search 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.

Before the cooperative memory read, each thread must generate the index of its corresponding entry (l or h) and multicast this information among the other threads. This communication can also be done using the shared memory. Since all cooperation operations proposed in our design are performed at the warp level, there is no need of costly explicit synchronisation.

The main drawback of the memory-cooperative scheme is that all the FM-index entries read by a warp must fit simultaneously into shared memory. A relatively large sampling distance d 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 sampling distances larger than 200, corresponding to FM-index entries of 128 Bytes or larger.

4.3 Full-cooperative

Using register memory and shuffle instructions (available on Kepler and subsequent architectures) helps improving thread occupancy of the memory-cooperative, but is not the final solution. A much better solution can be achieved by reducing the working set of each thread (and hence of the full application) and making threads cooperate on the computational part of the algorithm too.

Figure 2.c presents the full-cooperative design. The most important difference is that threads cooperate for reading data and then *immediately* cooperate for counting symbol occurrences and generating the output SA intervals. In the example of the figure, threads in a warp cooperate to read 4 entries and then process the entries to generate 4 outputs. This approach allows adjusting the working set of each thread to a target size with the objective of maximising thread occupation. The working-set per thread can be as low as the size of a memory request (16 Bytes). Comparing figures 2.b and 2.c we notice that the full-cooperative scheme must simultaneously kept only four FM-index entries in local memory instead of 32. In other words, the granularity of the work assigned to each warp can be maintained constant even when the FM-index entry size is increased. This idea has been also applied in [4] to reduce the working set of the application computation with good results.

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. We used the provided $shuffle$ instructions to implement the different communication patterns needed for thread cooperation and corresponding to the following steps:

1) Multicast l and h values among threads
2) Threads generate a single cooperative memory load
3) Multicast symbol that must be applied on LF operation among threads
4) Parallel symbol counting by all threads
5) Parallel reduce of partial counters
6) Parallel gather of results

The drawback of this implementation is that we have increased the computational load of the full-cooperative scheme so that it will become the new bottleneck of the algorithm. However, this does not translate into a practical problem, due to the vast amount of so far unused computational power that the GPU can provide.

5 Experimental Results

We have run the implementations of backward search described so far on the CPU and on several GPU platforms. In this section we first assess the overall performance and then present a detailed analysis of each implementation, in order to identify the main architectural bottlenecks.

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 102.4GB/s. GPU results shown were done on a Nvidia GTX Titan with 2688 Kepler CUDA cores and 6GB main memory providing 288GB/s. For the case of comparative GPU analysis (fig. 10) we also used a Kepler GTX 680 (1536 CUDA cores; 192GB/s), a Kepler K20c (2496 CUDA cores; 208GB/s) and a Maxwell GTX750Ti (640 CUDA cores; 88GB/s).

The input of our tests was a set of 10 million input queries produced by well-known simulation tools [9] [20]; they were searched in the human genome reference GRCh37.

Before starting measurements we always made sure that the FM-index and the queries were already residing in the CPU and GPU memory. The multicore CPU implementation used 16×2 threads (OpenMP) to exploit hyperthreading. GPU implementations set the thread-block size value to provide the highest performance. The results are expressed in terms of the number of LF operations per time unit.

Our experimental methodology considers two kind of experiments: (a) scalability measures when increasing the number of active threads; and (b) performance measures without memory access penalties. For (a) we include a conditional statement that controls at run time the number of threads performing LF operations. For (b) we use the same query for all the backward search operations, forcing all threads to actually access the same piece of data in the local cache. The goal of the latter experiment is to estimate the performance of the computation part of the code isolated from the effect of memory performance.

5.2 Overall performance results

Fig. 3 shows the performance of the task-parallel approach on the CPU and that of the full-cooperative approach on the GPU. In both cases, we show the effect of increasing the FM-index sampling distance d from 64 to 960. Presented results correspond to the best-performing configuration for each sampling distance and for each system implementation. The figure shows a clear speed-up of the GPU version as compared to the multicore CPU version, in a range between 5.7× and 12.3× (corresponding in absolute terms to 1.3—2.1 Giga-LF operations per second).

The plotted dot line correlates performance and FM-index size as the sampling distance is increased. While the CPU suffers from a steady performance deterioration due to the increased computation work associated to a larger d, the GPU tolerates the index compression without any noticeable performance penalty up to d=960. Apart from enabling backward search on larger genomes, this parameter setup

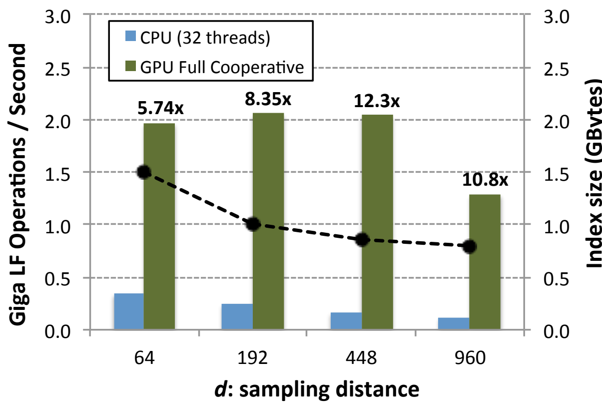

Fig. 3: CPU and GPU performance, measured in LF operations per second

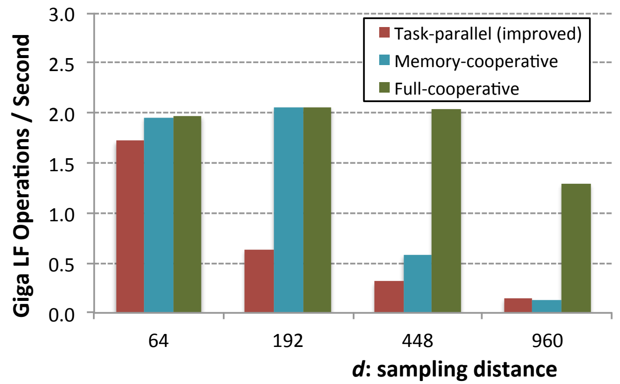

Fig. 5: Performance comparison of the task-parallel, memory and full-cooperative schemes

can be of special interest when using low-end GPUs that provide smaller amounts of memory.

Fig. 5 compares the performances of the proposed GPU parallelization schemes, displaying the best results for each case. As explained before, the full-cooperative design outperforms the other two. The performance of the task-parallel scheme is only competitive for d=64; similarly, the memory-cooperative scheme does not scale to d>192.

5.3 Detailed performance analysis

We'll now try to provide insight for the reasons of the performance behaviours shown here. We'll focus our discussion on two scenarios of interest: d=64 and d=448. The first one is a good case to describe how well each strategy works in favourable conditions; the second elicits the inefficiencies of the strategies described and shows how the full-cooperative solution is less affected than the rest.

Fig. 4 benchmarks three different implementations of a task-parallel scheme and the cooperative version, showing their performance as a function of the number of threads used. The expected result is that performance should first increase with more active thread (they issue more requests to saturate the memory system) and then eventually flatten as memory bandwidth limitations show up.

The performance results from Fig. 4 are complemented with main memory performance results shown in Fig. 6. The normalised number of Bytes requested to GDRAM should remain constant when increasing the number of running threads. An increase of the volume of data requested to GDRAM indicates that the L2 cache is not able to store data that should be reused by the program.

The performance behaviour of the cooperative version is very robust, scaling gracefully up to 4 thousand active threads. Then, memory bandwidth limits performance. The task-parallel schemes exhibit the performance anomalies anticipated in the previous section, which are discussed next.

The best performing task-parallel approach (labelled as "improved") uses 16-Byte memory loads and assigns a single LF operation per thread. The naive versions assign two LF operations per thread, both using 4-Byte and 16-Byte memory accesses. The overall improvement of the first version compared to the two naive versions is respectively 3.5× and 2.6×.

The naive task-parallel schemes overflow the L2 cache capacity and end up reading 3.8× and 2.0× more data from GDRAM than necessary. However, the performance advantage of the improved task-parallel version is not related with a better usage of the L2 cache capacity. The likely reason

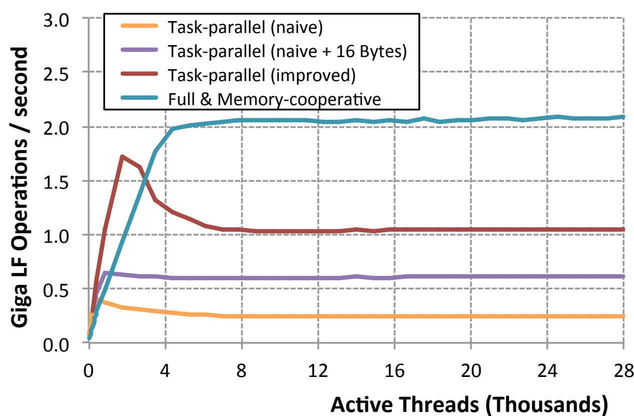

Fig. 4: Performance scalability (d=64)

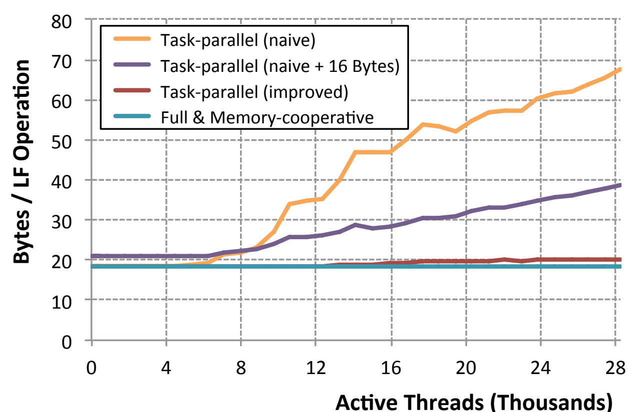

Fig. 6: Normalised amount of data read from GDRAM (d=64)

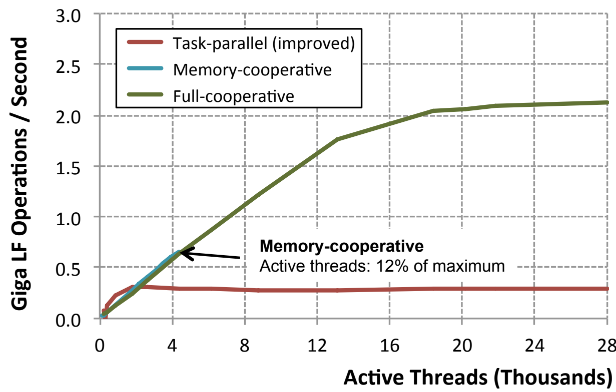

Fig. 7: Performance scalability (d=448)

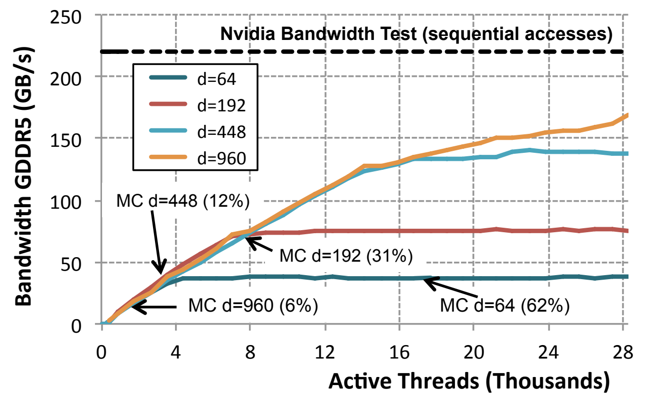

Fig. 8: Memory Bandwidth evolution for different d values

is that the L2 cache design is optimised to perform 32-Byte and 128-Byte data transfers, and lower-size data transfers are strongly penalised. I.e., coalescing is a strong requirement for performance even for L2 cache accesses.

We now analyse the scenario for a more compressed index (d=448) in order to bring light to the performance differences of the three basic approaches (see Fig. 7). The performance of the task-parallel version is limited by the small size of the L2 cache memory accesses: i.e. the non-coalesced access pattern. The memory-cooperative performance scales well until the shared memory capacity is exhausted by the requirements of too many threads. With only 12% of the maximum number of active threads, there is not enough parallelism to hide memory latencies and increase memory bandwidth usage.

After verifying that the full-cooperative scheme scales gracefully in most scenarios, the next question is how well it is exploiting the GPU available memory bandwidth and computation resources. For this purpose we depict in fig. 8 and 9 both the effective memory bandwidth and the instructions per cycle (IPC) rate achieved by the proposal.

For reference, fig. 8 includes the peak empirical bandwidth of our target GPU for sequential accesses (220 GB/s) as measured by the Nvidia bandwidth test tool. As anticipated, pseudo-random memory access patterns, as expressed in our algorithm, are well below the peak bandwidth. Increasing the sampling distance creates more spatial locality (larger FM-index entries) and this is reflected in a higher effective bandwidth (two times more bandwidth as entry size is duplicated).

For very large entries (d=960), the performance limit is not memory bandwidth anymore but the amount of computation. The application has to execute more instructions per FM-index entry, including the overhead due to thread cooperation, while reading a large entry has almost the same performance cost as reading a smaller entry (because of the characteristic access pattern of the algorithm).

Fig. 9 confirms that performance is not bounded by computation until d=960. The shaded bars represent the IPC obtained when a computation-only version is executed, while the solid bar indicates the actual IPC. An IPC measured figure of 3.5 is very close to the IPC=4 value achieved by several computation-bound applications published by NVidia.

5.4 Comparative performance analysis

This final subsection compares the performance achieved by the FM-index search algorithm on different Kepler GPU cards and the recent Maxwell GTX 750Ti. We expect that the memory performance of each GPU architecture will be the major factor to determine the overall performance. We also expect differences in the point where the cooperative scheme becomes computation-bound, which will be correlated with the ratio of computation and memory bandwidth offered by each GPU. Performance results are shown in fig. 10.

The right side of the chart shows the case where the performance of all the GPUs is computation-bound. In this case, the performance achieved correlates very well with the potential performance offered by each GPU. Notice that the low-end Maxwell GPU becomes computation-bound before all Kepler GPUs, for d=448.

The left-side of the chart shows the case where the performance of all the GPUs is memory-bound. In this case, performance does not clearly follow the potential memory bandwidth offered by GPUs, which is measured for sequential memory accesses. For random memory access patterns, memory performance is not as different on the range of GPUs analysed as could be inferred from the published bandwidth figures. An interesting case happens for d=192, where a GTX 680 performs as well as a Titan GPU (with 0.67× the potential bandwidth) as well as a Maxwell GTX 750ti performs like a Tesla K20c (with 0.42× the potential bandwidth).

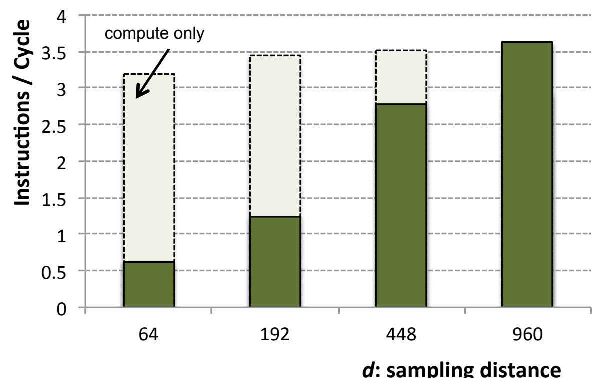

Fig. 9: Instructions per Cycle for full-cooperative scheme

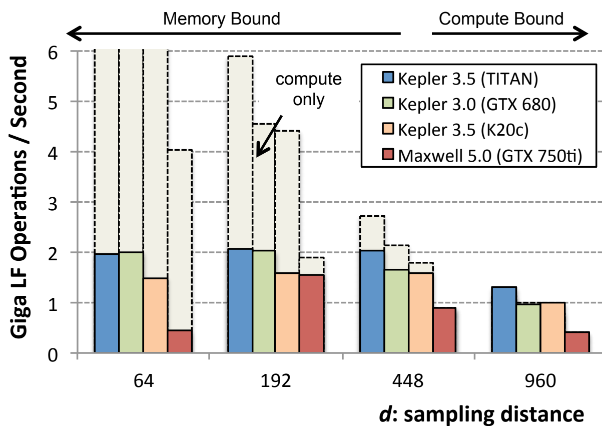

Fig. 10: Comparison among different GPUs (Full Cooperative)

For pseudo random memory access algorithms where the performance is far from computational capabilities offered by hardware, the relatively low cost and low energy consumption offered by Maxwell card provides a good target platform. That is, we can get a relatively good performance with a small fraction of the cost and energy consumption of higher-end GPU cards as the Tesla K20c or the GTX Titan.

6 Related Work

Over the last few years many CPU short-read mappers had been developed, like GEM [19], BWA [13] or Bowtie [12] to name a few. There have been many attempts to use GPUs for computational genomics [1] and read mapping in particular [6] [4] [2] [7]. In this context, the impact of pseudo-random dependent memory accesses on GPU was well described in [10] and many techniques have been developed to overcome this problem. For example, in [5] the authors point out a technique toward augmenting the FM-index locality and reducing the memory bandwidth required per LF-mapping operation.

Some GPU implementations of the FM-index have been developed as the core component of short-read mappers like CUSHAW [15][17][16], SOAP3 [14][18] or BarraCUDA [11]. Currently, SOAP3-dp [18] represents the most elaborated and efficient implementation. Based on a one-level sampling FM-Index with 64-Bytes aligned block entries, it uses a task parallel approach to query the index. Among other improvements, it 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 advantage of the memory available.

It is important to note that performing meaningful comparisons between different implementations is difficult. Not only it is very hard to determine the time that each program spends querying the index; one should also consider that in a realistic setup index access will be coupled with other calculations, which makes benchmarks blurry. However, all the implementations published so far share the same task-parallel approach; hence they can all benefit from the ideas and techniques presented in this paper.

7 Conclusions

Current GPUs (and CPUs) increase their memory bandwidth capabilities with wider data access pipelines. While this feature can be used to greatly improve the performance of algorithms accessing memory in a sequential fashion, it is not very useful for algorithms exhibiting pseudo-random access patterns, like the one analysed in this paper.

In this paper we have shown that the combination of a compact, size-tunable FM-index, and a novel thread-cooperative approach, can be used to tip the algorithmic bottleneck away from memory access. While the index size cannot be reduced too much on the CPU due to the excessive computational costs entailed, the same operation has a very limited impact on the GPU, where excess computational power is available to be used. Thanks to our results, one might use a few cheap and energy-effective low-end GPUs to replace a high-end GPU.

In the future, we plan to improve our GPU FM-index algorithm with the n-step mechanism described in [5]. Combined with the index compaction capabilities of our proposal, the n-step method will provide more data locality and better performance. We also plan to integrate this results into the GEM mapper [19] as to enhance the approximate string matching search on GPUs.

Acknowledgements

This research has been supported by MICINN-Spain under contract TIN2011-28689-C02-01. Thanks to NVidia for the donation of K20c GPU cards used in this work

References

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

[2] 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*, 27(10):1351–1358, 2011.

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

[4] A. Chacón, S. Marco-Sola, A. Espinosa, P. Ribeca, and J. C. Moure. Thread-cooperative, bit-parallel computation of levenshtein distance on gpu. In *International Conference on Supercomputing (ICS)*. (to be published) ACM, 2014.

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

[6] 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*, pages 173–180, Aug 2011.

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

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

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

[10] 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.

[11] 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*, 5(27), 2012.

[12] 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*, 10(3):R25, 2009.

[13] H. Li and R. Durbin. Fast and accurate short read alignment with burrows–wheeler transform. *Bioinformatics*, 25(14):1754–1760, 2009.

[14] 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.

[15] 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*, pages 684–690, May 2012.

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

[17] Y. Liu, B. Schmidt, and D. Maskell. CUSHAW: a CUDA compatible short read aligner to large genomes based on the burrows–wheeler transform. *Bioinformatics*, 28(14):1830–1837, 2012.

[18] 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*, 8(5):e65632, 2013.

[19] S. Marco-Sola, M. Sammeth, R. Guigo, and P. Ribeca. The GEM mapper: fast, accurate and versatile alignment by filtration. *Nature Methods*, 9(12):1185–1188, 2012.

[20] Y. Ono, K. Asai, and M. Hamada. PBSIM: PacBio reads simulator—toward accurate genome assembly. *Bioinformatics*, 29(1):119–121, 2013.