BowMapCL: Burrows-Wheeler Mapping on Multiple Heterogeneous Accelerators

David Nogueira, Pedro Tomás, and Nuno Roma

Abstract—The computational demand of exact-search procedures has pressed the exploitation of parallel processing accelerators to reduce the execution time of many applications. However, this often imposes strict restrictions in terms of the problem size and implementation efforts, mainly due to their possibly distinct architectures. To circumvent this limitation, a new exact-search alignment tool (*BowMapCL*) based on the Burrows-Wheeler Transform and FM-Index is presented. Contrasting to other alternatives, *BowMapCL* is based on a unified implementation using OpenCL, allowing the exploitation of multiple and possibly different devices (e.g., NVIDIA, AMD/ATI, and Intel GPUs/APUs). Furthermore, to efficiently exploit such heterogeneous architectures, *BowMapCL* incorporates several techniques to promote its performance and scalability, including multiple buffering, work-queue task-distribution, and dynamic load-balancing, together with index partitioning, bit-encoding, and sampling. When compared with state-of-the-art tools, the attained results showed that *BowMapCL* (using a single GPU) is $2\times$ to $7.5\times$ faster than mainstream multi-threaded CPU BWT-based aligners, like Bowtie, BWA, and SOAP2; and up to $4\times$ faster than the best performing state-of-the-art GPU implementations (namely, SOAP3 and HPG-BWT). When multiple and completely distinct devices are considered, *BowMapCL* efficiently scales the offered throughput, ensuring a convenient load-balance of the involved processing in the several distinct devices.

Index Terms—Exact string matching, burrows-wheeler transform, parallel computing, graphics processing units, OpenCL

1 Introduction

Bioinformatics applications often comprehend computationally intensive search and matching procedures, where a considerable amount of reads have to be mapped (or aligned) to a longer reference or database sequence, as most of the information is obtained by homology [1]. To reduce the search space, many alignment pipelines include preliminary *exact* search steps to find and map the reads (seeds), and to identify the most probable matching regions in the reference sequence. This allows them to circumscribe the subsequent processing steps, where the mapped seeds are applied to *inexact* local aligners based on more sensitive algorithms that allow occasional mismatches, insertions and/or gaps.

Conceptually, such preliminary mapping through exact search (no errors are allowed) goes far beyond the strict domain of bioinformatics, influencing many other areas including pattern recognition, document matching and text mining. Under such broader context, this operation is regarded as a regular pattern matching problem whose output is a list of all the occurrences of each short sequence of characters (*pattern* or *query*) of length n in a longer reference sequence of characters (*string*) of length m, such that $n \ll m$ [2].

Under this assumption, several exact mapping tools are already widely available. One of the most prominent approaches uses full-text indexes based on the combination of the Burrows-Wheeler Transform (BWT) [3] and the Ferragina-Manzini Index (FM-index) [4], executing a backward search that mimics the traversing of a tree data-structure, but without its underlying memory footprint. When compared to other full-text indexed algorithms and related data structures, this approach offers significant performance improvements, since it features a considerable smaller memory overhead and its complexity is only dependent on the length of the substring (*query*) that is being processed ($\mathcal{O}(n)$ search time, with $n \ll m$).

To mitigate the significant demands that are still imposed by these computational intensive procedures, the usage of high-performance computing platforms has been regarded as a mandatory approach, and several bioinformatics applications have been accelerated by exploiting their inherent data and functional parallelism. For such purpose, multi-CPU, GPU or even special-purpose dedicated processing structures have been regarded as unavoidable alternatives to conventional sequential implementations. However, exploiting such vast set of different solutions often imposes strict restrictions and costly implementation efforts, mainly due to their distinct architectures.

Thanks to their programmable nature and attainable performance, GPUs have been widely exploited for exact string matching. However, contrary to conventional dynamic-programming alignments, characterized by a uniform data access pattern, indexed exact search is characterized by an unpredictable and highly irregular memory addressing, which poses difficult challenges for efficient implementation in GPUs. Nevertheless, the mapping efficiency offered by indexed approaches, allied with the vast parallel processing capabilities of GPUs, have deserved a considerable attention of the research community. As a result, a significant number of BWT-based tools using GPUs have been presented, with a particular predominance to CUDA-enabled GPUs, such as SOAP3 [5], HPG-BWT [6], CUSHAW [7] and BarraCUDA [8]. However, an important

- *The authors are with the INESC-ID, Instituto Superior Técnico, Universidade de Lisboa, Portugal. E-mail: david.jacome.nogueira@tecnico.ulisboa.pt, {pedro.tomas, Nuno.Roma}@inesc-id.pt.*

Manuscript received 17 Feb. 2015; revised 10 Oct. 2015; accepted 16 Oct. 2015. Date of publication 26 Oct. 2015; date of current version 4 Oct. 2016.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TCBB.2015.2495149

1545-5963 © 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

step still needs to be fulfilled in order to efficiently exploit the whole set of multiple and possibly different devices that are currently available in the latest generations of heterogeneous platforms (e.g., Intel and AMD CPUs, NVIDIA and AMD GPUs, and even FPGA-based accelerators) with a single and unified environment. Furthermore, efficiently scaling the exploited parallelization with the number of available devices still raises important challenges in terms of scheduling and load-balancing.

To achieve this objective, a new exact mapping tool is herein proposed: *BowMapCL*. It consists of a highly parallel and data-agnostic implementation of a BWT-based indexed search that allows an efficient execution on heterogeneous platforms based on multiple CPUs and GPUs. Moreover, contrasting to the conventional CUDA-based approaches, it adopts the unified OpenCL API, making it the first implementation that is able to be concurrently executed in several and possibly different accelerators coexisting in heterogeneous platforms. To maximize the execution efficiency, a broad set of techniques and enhancements were investigated and implemented, including multiple buffering, efficient data dispatching, dynamic load-balancing, and index bit-encoding, sampling and partitioning. Furthermore, the proposed implementation also allows the processing of arbitrarily large reference sequences, which contrasts with the restrictions of most state-of-the-art approaches.

2 BWT+FM Indexed Exact Search

Sequence mapping based on the BWT [3] and FM-index [4] comprehends two distinct steps, namely: *i*) off-line index creation, executed in a single preprocessing phase; and *ii*) a backward search procedure, to find and map the set of queries under consideration.

2.1 Burrows-Wheeler Transform and FM-Index

Let **A** be the alphabet of a length-m text **T** and \$ a symbol not present in **A** and lexicographically smaller than any other symbol. The computation of the BWT of **T** starts by defining an auxiliary string **T**′, obtained by appending \$ to **T**, and by computing the corresponding Suffix Array (SA). The SA can be represented as a vector of integers, corresponding to the indexed positions of all the suffixes in **T** after they have been lexicographically sorted. Accordingly, the ith smallest suffix will have its starting index on the ith entry of the SA. Hence, the relationship between the BWT and the SA can be formulated as $BWT[i] = T[SA[i] - 1]$ and the BWT can be computed with a complexity equal to the one of building the SA. In particular, the `libdivsufsort` library (version 2.0) can be used to construct the SA in $\mathcal{O}(m \times log(m))$ time, which is subsequently used to obtain the BWT by using the aforementioned relationship.

To obtain the FM-index, two additional data structures were proposed by Ferragina and Manzini [4]: the **C** vector and the **OCC** matrix, where $C(a)$ represents the occurrences count of all characters lexicographically smaller than a in the text (with $a \in \mathbf{A}$), while each $OCC(a, i)$ entry represents the number of occurrences of character a in the prefix $\mathbf{BWT}(\mathbf{T}')[: i]$.

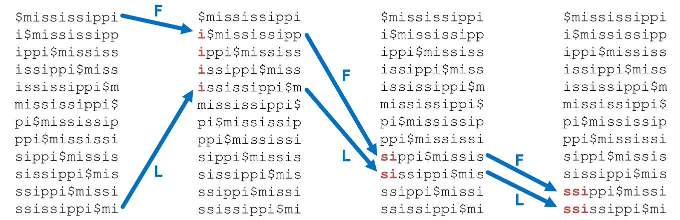

Fig. 1. Backward search of substring `ssi` on the `mississippi` text.

2.2 Backward Search

By using the precomputed **C** and **OCC** data structures, it is possible to search for any arbitrary substring in the reference text using the backward search procedure, based on the *last-to-first column mapping* property [4] (see Algorithm 1). As an example, Fig. 1 illustrates this procedure to find all occurrences of the `ssi` substring in the `mississippi` reference text. In each step, the **F** and **L** arrows represent the FIRST and LAST coordinates in the BWT matrix that were obtained after each iteration of the backward search algorithm. The procedure evolves backwardly, by selecting the previous character of the substring and by appending it to the searched suffix.

Algorithm 1. Backward Search Procedure

1: **procedure** Backward search(queries[ ][ ], C[ ], OCC[ ][ ])
2: **for** every query j **do**
3: i := size of query j, i.e., i := sizeof(queries[j])
4: c := last character of query j, i.e., c := queries[j][i]
5: FIRST := C[c]+1
6: LAST := C[c+1]
7: **while** (FIRST $\leq$ LAST) AND (i $\geq$ 2) **do**
8: c := get previous character of query j, i.e., c := queries [j][i-1]
9: FIRST := C[c] + OCC[c][FIRST-1] + 1
10: LAST := C[c] + OCC[c][LAST]
11: i := i - 1
12: **end while**
13: **if** LAST $<$ FIRST **then**
14: return not found
15: **else**
16: return values of indexes between FIRST and LAST
17: **end if**
18: **end for**
19: **end procedure**

The resulting output is the set of intervals of text coordinates [FIRST, LAST] corresponding to all the occurrences of the considered substring in **T**′. This is consistent with the fact that whenever a substring occurs in the text, the indexes of those occurrences are contiguous in a SA interval, as those suffixes will have that substring as their prefix. Whenever the LAST value happens to be lower than the FIRST value, the substring does not occurs in the reference text.

Accordingly, by using both the SA and the BWT+FM index data structures, the desired mapping can be retrieved from the SA interval in $\mathcal{O}(1)$ time.

3 Related Work

A considerable number of sequence aligners has been developed in the past few years. To allow the presence of some

TABLE 1
Comparison between State-of-the-Art BWT-Based Tools and the Proposed One (*BowMapCL*)

Tool	CPU/GPU	Platform Heterogeneity	Scalability	Data Type Agnosticism	Unlimited Index Size	System Requirements
Bowtie	CPU	No	CPU-only	No	Yes	Few [a]
BWA	CPU	No	CPU-only	No	Yes	Few [a]
SOAP2	CPU	No	CPU-only	No	Yes	Few [a]
SOAP3	GPU	No (CUDA-based)	Single GPU	No	No [b]	Huge [c]
HPG-BWT	GPU	No (CUDA-based)	2 GPUs [d]	No	Yes	Huge [e]
CUSHAW	GPU	No (CUDA-based)	Up to 2 GPUs	No	No [f]	Few [g]
BarraCUDA	GPU	No (CUDA-based)	Multiple GPUs	No	Yes	Few [h]
BowMapCL	GPU	Yes (OpenCL-based)	Multiple GPUs	Yes	Yes	Few [i]

a) To index the complete human genome, Bowtie, BWA, and SOAP2 require 2 GB, 3.2 GB, and 8 GB of RAM, respectively.
b) SOAP3: the reference sequence can contain at most four billion characters (base pairs).
c) SOAP3: Linux workstation equipped with a multi-core CPU (default quad-core) with at least 20 GB of main memory; CUDA-enabled GPU (compute capability > 2.0) with at least 3 GB of memory (e.g., 2.5 GB is required for indexing a human genome).
d) HPG-BWT: Each GPU runs a different strand (forward and reverse) of the mapping kernel; the used GPUs cannot be used for parallel and concurrent execution of multiple and distinct mapping procedures.
e) HPG-BWT: the required host memory depends on the suffix array compression ratio and on the number of tolerated errors (e.g., for human genome index calculation with a 32× compression ratio and support for more than one error it needs about 7 GB of host memory and GPUs with at least 3 GB of local memory).
f) CUSHAW: individual chromosomes should not be longer than 2 GB.
g) CUSHAW: requires NVIDIA Fermi architecture or newer (no information regarding RAM nor GPU global memory size).
h) BarraCUDA: requires 4 GB of host memory and at least 20 GB of disk space when processing large genomes; the GPU local memory should have at least the size of the BWT-encoded genome (.bwt + .rbwt files), plus another 675 MB of buffering space to perform the alignments; only runs on GPUs with CUDA capability > 2.0.
i) BowMapCL: OpenCL version 1.1 (or higher).

mismatches and of small structural variations (InDels) in the sequences being aligned, many of these tools avoid the $\mathcal{O}(m)$ complexity of exhaustive dynamic programming approaches (m is the size of the reference sequence) by combining a preliminary *exact mapping* step with a subsequent *inexact alignment* [9], [10], [11], [12]. The first step frequently adopts offline search algorithms, as they pre-process the reference text (to create an index) before executing the actual mapping. Accordingly, although the whole query set has to be processed by the first step, only a significantly smaller subset (the mapped queries) has to be processed by the second step, leading to a substantial reduction of the execution time.

Most current short-read mappers adopt full-text indexes in their search procedure, by using a variety of data structures including hash-tables [13], suffix-arrays [14], [15], suffix-trees [16] or the BWT+FM index [3], [4], [17] to reduce the time-complexity of the search procedure to $\mathcal{O}(n)$ (n is the size of the query string). In particular, since the BWT+FM index incurs in a smaller memory footprint (when compared with other structures, such as suffix-trees), many currently prevailing tools have adopted this index (e.g., Bowtie [18], BWA [19] and SOAP2 [20]).

To further reduce the processing time, these implementations either rely on POSIX threads to take advantage of multi-core CPUs, or on Message Passing Interface (MPI) schemes to distribute the computation across clusters of CPUs [21]. Nevertheless, GPU-accelerated solutions are gradually being recognized as highly viable alternatives to provide added performance, by exploiting their higher parallelization capability. As a consequence, several BWT-based tools have already been developed for GPUs, namely SOAP3 [5] (based on SOAP2), HPG-BWT [6], CUSHAW [7] and BarraCUDA [8]. However, all these tools are implemented using CUDA programming environment for NVIDIA GPUs, preventing them from being executed in heterogeneous systems integrating GPUs from makers other than NVIDIA (e.g., AMD, Intel). Moreover, they often impose strict requirements in terms of the minimum host memory and GPU global memory, significantly limiting the considered application domain. Table 1 presents a comprehensive comparison of these CPU and GPU tools, including their most relevant limiting aspects (see footnotes).

To circumvent these constraints, an improved exact mapping tool (denoted as *BowMapCL*) is proposed. Such tool aims the mitigation of a broad set of limiting aspects existing in current state-of-the-art implementations, anticipating the achievement of greater processing performances by adding support to:

Platform heterogeneity. Contrasting to the state-of-the-art GPU-based tools, *BowMapCL* is not limited to CUDA-enabled GPUs. On the contrary, by using the platform-unified OpenCL API, it supports a larger variety of accelerators, such as NVIDIA and AMD/ATI GPUs (which constitute *BowMapCL* main focus, due to their higher processing performance) or any other accelerating device supporting the OpenCL API (e.g. ARM Mali GPUs, Intel Xeon Phi and Altera FPGAs). To the best of the authors' knowledge, this is the first BWT-based tool with such capability.

Scalability. *BowMapCL* also aims an efficient scaling of the offered throughput in terms of the number of used accelerating devices. Only a few state-of-the-art tools offer this capability (e.g., SOAP3 only runs on one GPU; HPG-BWT uses a maximum of two, but each GPU runs a different strand (forward and reverse) of the mapping kernel). To attain this objective, important complementary aspects have to be considered, namely:

- Dynamic load balancing procedures, to balance the workload across the multiple (and not necessarily equivalent) coexisting devices.

- Data-transfer masquerading, through complex buffering and kernel overlapping schemes, to mitigate the severe penalties incurred from the sequences data transfer between: *i*) hard-disk and host; *ii*) host and the target accelerators.

Data-type agnosticism. Contrasting to many hard-coded state-of-the-art tools, *BowMapCL* aims to be completely agnostic in what concerns to the input data type, allowing the processing of any generic sequence data (e.g., DNA, proteins, text). The user shall only provide the encoding of the input data and of the corresponding lexicon.

Unlimited index size. Contrasting to existing state-of-the-art alternatives, *BowMapCL* only requires the hard disk drive of the host machine to have enough space to hold the FM-Index and the set of query and reference sequences. It does not impose any RAM memory requirements related to the index size, such as those imposed by SOAP3 and HPG-BWT, which need at least 20 GB and 7 GB of host RAM, respectively, and a GPU with at least 3 GB of local memory to process the human genome. On the contrary, it incorporates:

- Index partitioning schemes, to conveniently divide the input reference text according to the amount of global memory available in each device, assuring its execution in any heterogeneous platform.
- Bit encoding and sampling schemes, to further reduce the memory footprint of the index data structures, conveniently devised to suit the GPUs architecture constraints and to avoid the introduction of any performance penalization.

Furthermore, the output interface of *BowMapCL* ensures an ample extensibility of this exact mapping tool through a straightforward connection with most alignment pipelines. In particular, it can be easily integrated with already existing inexact alignment frameworks that have been proposed for the implementation of the Smith-Waterman algorithm in GPUs (e.g., CUDASW++ [22]), envisaging the possibility to attain remarkable alignment performances through a massive exploitation of parallel processing heterogeneous platforms supporting OpenCL and CUDA.

4 Architecture and Implementation

The proposed tool is structured in two independent parts: *i*) the index generation, which receives the reference sequence file and creates all the necessary data structures for the subsequent sequence mapping (i.e., the BWT, the *OCC* matrix, the *C* vector and the suffix array); *ii*) the actual mapping procedure, which constitutes the main focus of the presented implementation, as described in the following sections.

4.1 Parallelization Strategy

Due to the considerable amount of queries to be mapped, *BowMapCL* takes advantage of the large number of cores in the target GPU accelerators, where each query can be independently processed. However, since the whole dataset might not be accommodated in the host memory, chunks of queries are gradually fetched from the input file and subsequently dispatched for a mapping kernel in the GPU. Accordingly, all the processors of the GPU accelerator perform the same task (on different data).

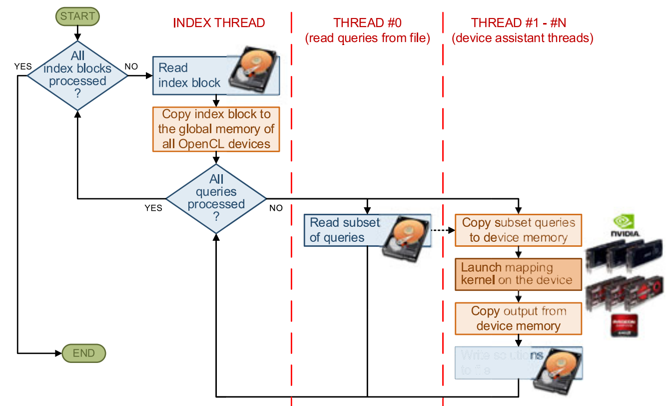

Fig. 2. Implemented parallel exact mapping work-flow.

Complementary to this data-level parallel processing model, task-level parallelism must also be exploited in order to ensure an efficient and simultaneous execution of the multiple tasks required by the search procedure [21]. Such tasks range from purely I/O operations (e.g., reading data/writing results from/to multiple files; data transfers between host-side memory and the target devices global memories) to computational ones (issuing of the mapping kernel in the GPUs and conversion of the backward search output to the desired solutions).

Accordingly, the proposed work-flow was especially devised to tackle the management of the multiple operations, as illustrated in Fig. 2. The program starts by loading the index under consideration (corresponding to the reference sequence) into the devices memory (see Section 4.4.1). Whenever this data-structure cannot be accommodated in the GPU local memory, the index is partitioned in multiple blocks, which are sequentially processed by the GPU. For each pair of index block and chunk of queries, an OpenCL mapping kernel is executed in one of the co-existent OpenCL accelerators, according to the pseudo-code presented in Algorithm 2. During this step, thousands of queries are simultaneously matched with the reference index. Finally, the kernels outputs are copied back to the host memory and the desired solutions are written to disk, after being converted with the precomputed suffix array.

To assist the issuing of all these operations (on the host side), the implemented tool aggregates several device-assistant threads for each OpenCL device (as further explained in Section 4.3.1).

4.2 Parallel Sequence Mapping

Despite the highly irregular addressing pattern of the backward-search procedure (see Algorithm 1) and its inherently sequential nature (there are data dependencies within each iteration of the internal *while* cycle loop, corresponding to the processing of each query), data-level parallelism can still be applied in the outer loop, as each query is independent and the same instructions can be executed over different data.

4.2.1 OpenCL Kernel Implementation

The pseudo-code presented in Algorithm 2 represents a simplified overview of the implemented OpenCL mapping

kernel. Two major aspects are worth noting when this kernel is compared with its sequential version (Algorithm 1): *i*) the mapping of the queries into the processing elements; and *ii*) the use of the GPU memory hierarchy to store the data structures.

Algorithm 2. OpenCL Backward-Search Kernel

```
1:  procedure BACKWARD SEARCH KERNEL(queries[ ][ ], queries_
    sizes[ ], C[ ], OCC[ ][ ], character_Map[ ], numqueries)
2:    global_index := get work-item global id
3:    local_index := get work-item local id
4:    group_size := get local size
5:    copy C vector and character_Map to local memory
6:    wait at the OpenCL barrier //work-group synchronization
7:    if global_index <numqueries then
8:      i := size of query (i := queries_sizes[global_index])
9:      c := last character of query (c := queries[global_index]
        [i])
10:     FIRST := C[c]+1
11:     LAST := C[c+1]
12:     while (FIRST ≤ LAST) AND (i ≥ 2) do
13:       c := get previous character (c := queries[global_in-
          dex][i-1])
14:       FIRST := C[c] + OCC(c,FIRST-1) + 1
15:       LAST := C[c] + OCC(c,LAST)
16:       i := i - 1
17:     end while
18:     queriesFIRST[global_index] := FIRST
19:     queriesLAST[global_index] := LAST
20:   end if
21: end procedure
```

Contrary to the sequential version, where the executing thread loops around the query dataset, each thread in the GPU parallel mapping (see Algorithm 2) processes a different query, based on its thread identifier (global thread ID). Hence, since no data sharing is required between the working processing elements, there is no need for complex synchronization schemes. As a consequence, the choice of which threads should be grouped together, as well as the size of the work-groups that map into the compute units of the device, do not compromise the correctness of the matching, neither significantly affect the resulting performance.

On the other hand, although the GPUs memory hierarchy typically integrates several memory elements, the usage of local memories is usually preferred, due to their much higher bandwidth and lower latency. Therefore, the main data structures should be copied to and accessed from the local memory, provided that there is enough space and the number of accesses to their entries justifies the inherent penalty of copying them to the local memory, before the actual execution. As an example, the **OCC** matrix tends to occupy a significant amount of memory, making an eventual copy to the local memory rather impracticable (it largely exceeds the available local memory of typical accelerating devices). In contrast, the **C** vector is a very good candidate, since its size (proportional to the lexicon size) is usually much smaller.

During the processing of every character of each query, the assigned thread needs to index the corresponding row in the **OCC** matrix. To rapidly identify such row, a *character_Map* data structure is used, which holds 256 (or 2^b) entries for an 8-bit (or b-bit) lexicon. Hence, whenever a given character exists in the text, this structure stores the number of its assigned row in the **OCC** matrix. Accordingly, since this structure is directly indexed with the character that is being searched for, it allows obtaining the required row with a complexity of $\mathcal{O}(1)$. Hence, the *character_Map* data structure, as well as the **C** vector, must be both copied from the global to the local shared memory before the exact search begins, thus requiring a synchronization point at the start of the mapping procedure, in order to make all the threads in the same work-group to wait for the others, and ensure the consistency of the local memory corresponding to these local data structures.

It is important to recall that the optimal performance can only be attained if each target device is configured with an optimum setup in terms of the adopted work-group size, which is intrinsically related to its own architecture. Accordingly, a preliminary performance modeling should be executed, in order to determine the most efficient configuration [23].

The adoption of the exact backward search algorithm is particularly suited for this parallel execution model since it ensures a homogeneous thread execution-time that is independent on the input data. This contrasts with other possible alternatives (e.g., non-exact search procedures based on backtracking algorithms), where a variable number of tentative matchings would have to be considered for each individual query in a much larger search space, preventing that all GPU threads finish at the same time.

4.2.2 Kernel Output Conversion

Due to its large memory footprint (e.g., about four times the size of the original text) the suffix array is kept on the host memory, where the memory usage is typically less constrained than in the accelerating devices. As a result, it is the set of host-side assistant threads that convert each kernel output (corresponding to each query) to the actual positions in the original text. Besides this memory concern, it is also worth noting that this final conversion is not well balanced and therefore not suited for execution in the accelerators, as the number of solutions for each query may significantly vary (different number of occurrences for each query). Such imbalance would cause some GPU threads to stall whenever their work was finished before the work of the other threads. Since this is not desired on a load balanced GPU parallel execution, the implementation of this conversion in the CPU, where the computational resources are not yet fully utilized by the assistant threads, is highly advantageous.

4.3 Scalability and Device Management

To ensure the scalability of the proposed implementation in a multiple-GPU platform, several important device-management aspects have to be taken into account in order to efficiently orchestrate the GPUs and all the involved data-flow.

4.3.1 Computation and Communication Overlap

In a hypothetical GPU implementation supported and assisted by a single CPU thread, the CPU would have no major operation besides sending the queries to the target devices and receiving their computed output, being kept

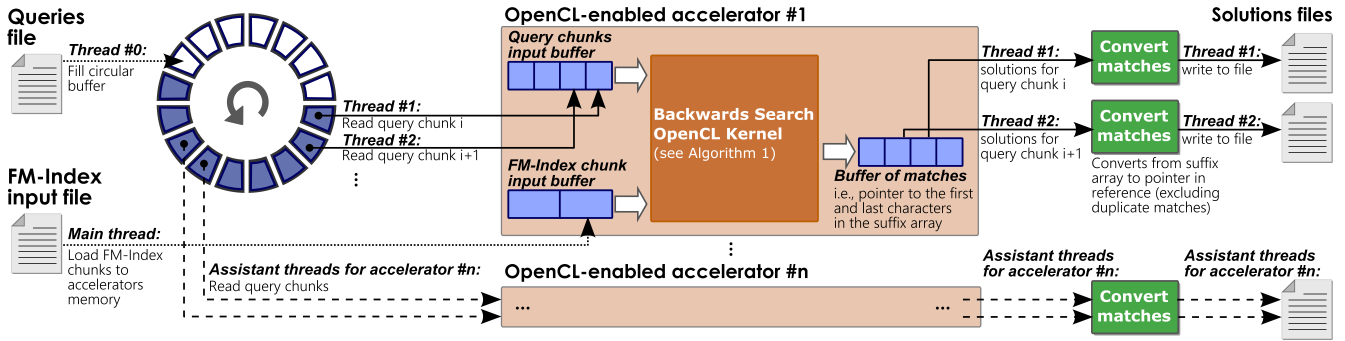

Fig. 3. Architecture of proposed OpenCL-based parallel implementation of the exact string matching procedure.

idle most of the time. Moreover, whenever the targeted accelerating device did not support OpenCL non-blocking I/O operations, the CPU would be blocked, waiting for the completion of the write and read of data. Such an usage of blocking operations using a single CPU thread would not even allow the exploitation of multiple accelerators, as the host thread would be blocked in each OpenCL operation (load of chunks of queries to the accelerator, waiting for the kernel execution, or copy of the results from the accelerator to the host main memory).

On the other hand, even from the communication point of view, structural hazards on the device input and output buffers must also have to be avoided (e.g., the CPU thread that assists a given OpenCL device cannot send the queries corresponding to the next iteration to the accelerator global memory while the kernel is still running, as the input buffer with the current input queries is still being used).

To circumvent these issues, *BowMapCL* considers two complementary approaches: *i*) multiple device-assistant CPU threads, not only to ensure a constant input/output data flow, but also a simultaneous data processing in the devices; *ii*) a multiple buffering scheme, to support multiple input/output data-flows at each GPU device, promoting a complete overlapping between computation and communication.

The proposed multiple buffering scheme is based on the division of the allocated data structures on the target device in multiple sub-banks, allowing the usage of multiple and distinct input buffers for the queries and multiple output buffers for the results (as depicted in Fig. 3). These multiple sets of buffers on each device are concurrently used, where each set is managed by a single CPU thread. Hence, instead of having a single CPU thread per device, the program assigns a number of threads equal to the amount of buffers created in each device. As an example, in a platform with two GPUs and two sets of buffers per GPU (i.e., using a double buffering technique, such as the one in Fig. 3), 4 threads are assigned for GPU management. Hence, while the GPU device is processing one chunk of queries (i.e., the kernel is executing) over one set of buffers, the other CPU thread can copy new queries to the other input buffer or retrieve the solutions of the previous kernel execution. This allows a complete hiding of the communication overheads, by overlapping the data transfers with the computation performed in each device.

4.3.2 Task Parallelism

In a multiple-GPU implementation, the scalability of proposed mapper is ensured not only by the several device-assistant CPU threads that coordinate and orchestrate each accelerator, but also by another subset of threads that perform the necessary I/O operations, as it is illustrated in Fig. 3. Accordingly, one CPU thread (thread #0) is assigned with the task of reading the queries from the input file and of writing them to a shared task queue (i.e., a pool of queries) in the host memory. Although there are multiple consumers (at least one per accelerating device), one single producer is created, as its operation tends to be much less time demanding. Each CPU thread that assists each accelerator (threads #1 to #4 in Fig. 3) has its own OpenCL command queue and it is responsible for: *i*) fetching a chunk of queries from the pool of queries; *ii*) enqueueing the writing of those queries into the corresponding accelerator global memory; *iii*) waiting for the execution of the kernel; *iv*) writing the kernel output to the host memory; *v*) performing the conversion of the output occurrence range to the desired solution space, by using the previously computed suffix array; and *vi*) outputting the final results to a file. Hence, although each thread waits for the completion of each kernel execution and data transfer, all the threads assigned to the accelerators are executing in parallel, allowing the program to scale with the number of devices with a minor overhead. Moreover, not only are the several searches occurring in parallel in the multiple devices, but the reading of the queries and the writing of the results to the output file are also simultaneously happening, which allows maximizing the usage of the computing resources of both the CPU and the accelerating devices. In particular, the writing of the resulting solutions to an output file is done in parallel by using an individual file per thread, in order to prevent a direct competition for a unique resource, which would arise if one single file was used. This approach does not introduce any disadvantage, as the queries are independent and the output files can be easily merged, if necessary.

4.3.3 Dynamic Load Balancing

To balance and distribute the workload across multiple coexisting devices, OpenCL offers the capability to query the processing resources of the available platforms, as well as their devices and their specifications. This allows for a preliminary and approximate configuration of the program, in order to adjust its implementation to the target platform.

However, considering that the GPU devices that are present in a heterogeneous platform are not necessarily equivalent, the implemented load balancing scheme also has to take this level of heterogeneity into account.

Furthermore, besides the differences between the offered processing performances, the devices may also be subject to different loads from other concurrent programs that indirectly affect their execution times (device contention), since the mapper may be competing for the resources with other running applications. As a consequence, dynamic scheduling and load balancing are highly desired, as the performance of each device is only known during its execution, and it may even suffer from changes in runtime. Therefore, the division of the workload in the considered multi-GPU platform is done by dynamically assigning independent fractions of the queries dataset to each device, meaning that the work distribution among the devices is conveniently adjusted during the execution.

The proposed dynamic load balancing is based on the work-queue task-distribution scheme depicted in Fig. 3, where a single producer fetches equal-sized chunks of queries from the input file and stores them in a task queue (circular producer-consumer buffer). As soon as each device finishes the processing of a given block, it accesses this circular buffer to get a new block of queries. Since the tasks are being dynamically assigned whenever the devices finish their current task, the faster devices will process more queries (and therefore, more data) to compensate for the slower ones (without the need to assign load-varying chunks). At the end, the difference of the devices execution time is, in the worst case, equal to the time the slower device takes to process the last block of queries. To circumvent this penalty, a *guided* scheduling technique is also applied on top of this load balancing, so that larger blocks of queries are fetched in the beginning of the program and smaller blocks at the end of the program, thus reducing the final imbalance penalty (i.e., trying to reduce the gap between each device execution time).

4.4 Mitigation of the Index Memory Footprint

Although the BWT+FM-index is already one of the data structures with the lowest memory demands (when compared, for example, with the suffix trees [4] [24]), a special attention must still be addressed to reduce its memory footprint. To mitigate this problem, many CPU-based implementations take advantage of compression techniques, such as run-length and statistical encoders (e.g., Huffman or arithmetic coders) and dictionary coders (e.g., LZ77 and LZ78). However, although these algorithms may be faster than traditional scan-based methods, they depend on the scan of the whole compressed text, making them not recommended for large inputs [4].

As a result, many implementations have combined the BWT with move-to-front encoding (MTF), run-length encoding (RLE) and variable-length prefix code compression [4] of the index data. Nevertheless, despite the significant compression ratios that can be attained, these techniques can hardly be efficiently implemented in GPU architectures, due to the highly irregular addressing patterns that are involved, making them heavily memory bounded.

Bearing in mind the utmost importance to reduce the memory footprint overhead, implementation-driven solutions based on regular bit-encoding data structures for the **OCC** matrix, together with sampling procedures for the **OCC** matrix and SA vector, had to be considered in the proposed implementation. These type of encoding strategies have also been used in other BWT-based aligners, such as BWA [19] and CUSHAW [7]. Nevertheless, very large index structures can still not be accommodated in the internal memory of most accelerating devices (e.g., GPUs), which led to the introduction of complementary index partitioning approaches.

4.4.1 Index Partitioning

To attain a flexible and scalable solution, able to handle any input text, regardless of its size, the implemented index partitioning divides the index in smaller blocks. The size of the blocks can be either automatically computed, to adjust to the global memory size of the accelerating devices, or can be user-defined, giving the user the opportunity to define a maximum size for each block (as the size of the index-block also influences the size of the suffix array partition that is loaded into the host RAM at each iteration). When automatically computed, the number of partitions are set to the smallest possible: number of partitions = size of index/((total GPU global memory) $\times$ (% of GPU global memory reserved for index data)).

With this added feature, and independently of the size of the considered reference input, the proposed tool does not impose any particular requisite in terms of the GPU memory size or CPU RAM memory specification. This independence (in terms of the computational resources) is also applied to the queries, due to the fact that they are read on-the-fly, in chunks of queries, each one with a pre-defined size.

In what concerns the processing of the index-blocks, two approaches have been considered [25]: *i*) sequentially processing index blocks and matching all the queries against each block; or *ii*) sequentially processing the chunks of queries and searching them in all the index-blocks. Although the latter approach would allow storing the solutions of each query contiguously, in the same output file, the communication overhead would be much higher, as the index data structures need to be reloaded into the accelerating devices a greater number of times than in the first case.

On the other hand, in platforms with multiple and possible different accelerators, two block partitioning approaches were considered: *i*) the index is partitioned into a number of blocks equal to the number of devices, in such a way that each device gets a different part of the index and each CPU assistant thread is responsible for independently matching all the queries to its part of the index; or *ii*) the devices share the same index block and the CPU assistant threads concur to divide the queries that should be distributed and matched to their block. Although the first approach seems to be better in the sense that it may reduce the number of times the index-blocks must be transferred from the CPU (just once for each accelerator), the finishing times may differ quite substantially. On the contrary, in the second approach each index block is loaded into all devices and the queries are optimally distributed so that they will finish approximately at the same time, making it more suited for dynamic load-balancing.

Independently of these partitioning strategies, it was preserved the possibility of searching substrings that are now being split across two adjacent blocks of the input text. To accomplish this, each block overlaps with the end of the

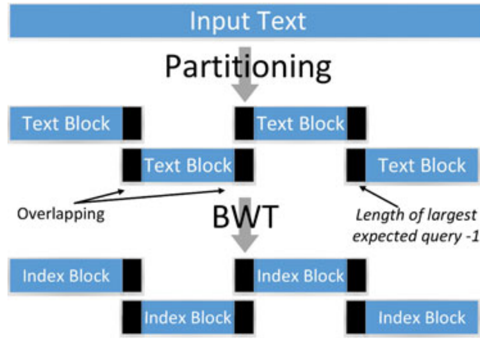

Fig. 4. Input text partitioning and posterior index creation for each block.

previous one by an amount given by $\texttt{MAX_QUERY_SIZE} - 1$, where `MAX_QUERY_SIZE` represents the maximum considered size for each input query. Hence, the last characters appear in both blocks, as depicted in Fig. 4. To avoid duplicate matches of the same query, the eventually found solutions are only reported if their occurrence index (after the suffix array conversion) are separated by at least the difference between `MAX_QUERY_SIZE` and the size of the query being processed.

4.4.2 OCC Matrix Bitmap Encoding

The most memory demanding data structure from the FM-index is the **OCC** matrix. Since each $OCC(c, i)$ entry (32-bit integer) represents the number of occurrences of character c in the prefix $\mathbf{BWT}(\mathbf{T}')[: i]$, it results in an overall memory footprint (in bytes) of:

$$\text{size}(\mathbf{OCC}) = m \times \text{alphabet_cardinality} \times \text{sizeof}(\text{int}).$$

Hence, in the simplest case corresponding to DNA data (A, C, T, G, \$), this memory space corresponds to $m \times 5 \times \text{sizeof}(\text{int}) = 20 \times m$ (i.e., the size of the index is 20 times greater than the original text).

Such a large footprint is hardly compatible with the strict memory restrictions of GPUs, thus imposing the introduction of a convenient data encoding. At the same time, the implementation of such encoding must be regular enough in order to be feasible and efficiently implemented in GPU architectures.

The adopted data structure consists of an **OCC** matrix bitmap ($\mathbf{OCC}_{\text{bitmap}}$), where each (integer) $OCC(c, i)$ entry is now represented as a boolean value (1 bit), indicating whether character c occurs in the ith position of the index (instead of storing the actual number of occurrences up to that position), leading to a memory footprint (in bytes) of:

$$\text{size}(\mathbf{OCC}_{\text{bitmap}}) = \frac{\text{size}(\mathbf{OCC})}{\text{sizeof}(\text{int}) \times 8}.$$

To prevent having to count all i bits when obtaining each $OCC(c, i)$ entry, a sampling strategy similar to the one proposed in [4] and also adopted by BWA [19], CUSHAW [7] and Bowtie [18] was considered. Accordingly, the $\mathbf{OCC}_{\text{bitmap}}$ matrix is complemented with an auxiliary data structure ($\mathbf{OCC}_{\text{sampled}}$) corresponding to the original **OCC** matrix sampled at certain specific positions of the index; the sampling interval corresponds to a multiple (sample_rate) of the amount of 1-bit $\mathbf{OCC}_{\text{bitmap}}$ elements that can be accommodated in a single 32-bits integer. Accordingly, for a sample rate r, the $\mathbf{OCC}_{\text{sampled}}$ matrix is composed of columns $0, 32r, 64r, \ldots$ of the original **OCC** matrix, resulting in a memory footprint (in bytes) of:

$$\text{size}(\mathbf{OCC}_{\text{sampled}}) = \frac{\text{size}(\mathbf{OCC})}{\text{sizeof}(\text{int}) \times 8 \times \text{sample_rate}}.$$

As a result, each $\mathbf{OCC}_{\text{sampled}}$ entry represents the accumulated occurrences of character c that were counted until the index range covered by the corresponding $\mathbf{OCC}_{\text{bitmap}}$ entry. Accordingly, whenever a given $OCC(c, i)$ value is required, the preceding $\mathbf{OCC}_{\text{sampled}}$ entry must be retrieved and the bits between that preceding sampled position and the desired one must be counted from the corresponding $\mathbf{OCC}_{\text{bitmap}}$ entry, thus involving two memory accesses. To perform such operation, a bit-mask is applied to the $\mathbf{OCC}_{\text{bitmap}}$ entry (in order to turn to zeros any ones that may exist in positions after the desired one) and a *popcount* function is used to count the number of ones in each word. Then, the two values (sample and popcount) are added and the number of occurrences is obtained. At this respect, it is worth noting that the popcount operation can either be invoked in the native OpenCL API (OpenCL version 1.2, or greater) or efficiently implemented with logic and shift operations [26]. Hence, the resulting memory footprint is equal to $\text{size}(\mathbf{OCC}_{\text{sampled}}) + \text{size}(\mathbf{OCC}_{\text{bitmap}})$.

At this point, it should be noted that the implemented data structure has some aspects in common with the one presented in [27]. They only differ from the selected bit-packing ratios (32 instead of 64, in order to better conform with GPU architectures), and in the ability of allowing the user to specify the sampling rate. By considering the above DNA data example (where $\text{size}(\mathbf{OCC}) = 20m$), with $sample_rate = 1$ and $\text{sizeof}(\text{int}) \times 8 = 32$ bits, it leads to a global footprint of $(20m)/32 + (20m)/(32 \times 1) = 1.25\, m$ bytes (i.e., 1.25 bytes for each element of the input text). When compared with the straightforward approach ($20\, m$), it represents a memory space reduction of 16 times.

In contrast, the BWA [19] stores the whole BWT of an m-sized DNA reference sequence (i.e., $2\, m$ bits) together with the **OCC** matrix. Since the considered structure represents each of the $4\, m$ existing integers with $\lceil \log_2 m \rceil$ bits and assumes a sampling rate of 128, it leads to a global memory footprint of $2\, m + 4\, m \lceil \log_2 m \rceil / 128$ bits. In particular, when large input references are considered (e.g., greater than one giga-nucleotides), the term $\log_2 m \approx 32$. In such a scenario, the BWA memory usage is $\approx 3m/8 = 0.375\, m$ bytes.

Hence, *BowMapCL* adopts a sampling scheme that is more conservative than BWA's. However, it is important to note that its memory usage can still be substantially reduced if greater sampling rates ($\text{sample_rate} > 1$) are considered, leading to FM-index data structures whose size may be even smaller than the original text size (at the cost of increasing the number of words that have to be processed whenever an $OCC(c, i)$ value is required). Nevertheless, since the access time to this data structure implicitly affects the performance of the whole backward-search procedure, an important compromise has to be established in order to

find the best balance between the resulting performance and memory footprint.

Another aspect worth noting refers to the data-type agnosticism of the adopted structure. Contrary to other hard-coded tools that only give support to DNA (e.g., BWA uses a 2-bit hard-hired encoding), the proposed implementation is able to process any alphabet (DNA, proteins, text, etc.), thus allowing for a better scalability when compared with other encodings that depend on the cardinality of the alphabet.

4.4.3 SA Sampling and On-the-Fly Computation

Due to its large memory footprint (i.e., $m \times \text{sizeof(int)}$ bytes), the suffix-array cannot be transferred into the GPU. Instead, it is kept and processed by the CPU. Even so, its significant size is still hardly manageable when very large reference sequences are processed. To circumvent this difficulty, a convenient sampling scheme was also considered in the suffix-array. Then, whenever the required value is not found in memory, the BWT's *Last (L) to First (F) column mapping* property (i.e. *LF-mapping* [4]) is used, by looping over the memory-resident BWT: $LF(i) = C[L[i]] + OCC(L[i], i)$.

The procedure starts by visiting the previous characters in the original reference text, till finding a stored sampled value. Since the last character of each row of the rotation matrix M precedes the first character of that row in the text, the previous letter of the desired index value in the text can be obtained by finding the row where it occurs in the first position (i.e., in vector F). The previous one would be the one in the last column of that row (i.e., in vector L).

Hence, by looping over the index positions and counting the number of hops (while visiting the characters that precede the one that was last visited), the required value is computed by adding the number of hops to the position of the first encountered sample. A special care must be taken to guarantee that such value indexes a position inside the vector (i.e., whenever the returning value is greater than the size of the vector, the remainder of the division is returned).

5 Experimental Evaluation

5.1 Datasets and Computing Platforms

To assess the performance offered by the proposed *BowMapCL* tool, a DNA dataset comprising the *E. Coli* complete genome (single chromosome, with around 4.6 Mbp), the *Homo Sapiens* complete genome (over 3 Gbp) and its Chromosome 1 (around 200 Mbp) was used. All the considered queries were randomly extracted from these species genomes.

This evaluation considered three different platforms:

A- dual quad-core (8 cores) Intel Xeon E5-2609 CPU @2.40 GHz (32 GBytes) + 2× NVIDIA GeForce GTX 680 GPUs (4 GBytes);

B- quad-core Intel Core i7-3820 CPU@3.6 GHz (16 GBytes) + 1× AMD Radeon R9 290X GPU (3 GBytes) + 1× NVIDIA GeForce GTX 560 Ti GPU (1 GByte);

C- quad-core Intel Core i7-4770K CPU@3.5 GHz (32 GBytes) + 1× NVIDIA GeForce GTX 780 Ti GPU (3 GBytes) + 1× NVIDIA GeForce GTX 660 Ti GPU (2 GBytes).

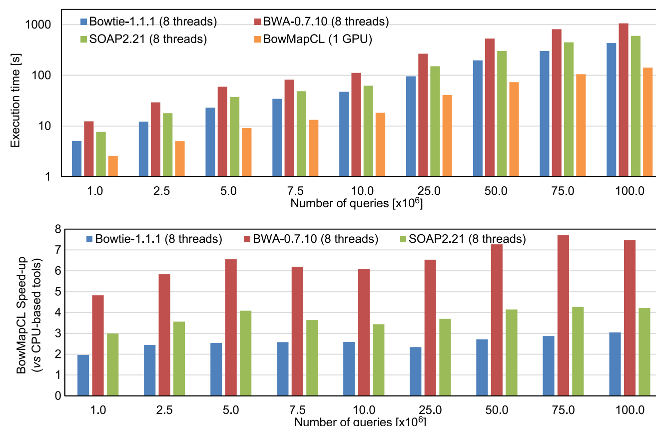

Fig. 5. Performance provided by *BowMapCL* when compared with CPU-based alternative tools (Platform C, using a single NVIDIA GeForce GTX 780Ti GPU, and the Human Chromosome 1 as reference).

Platform A, holding two identical GPUs, will be used to evaluate the tool scalability on the number of devices; *platform B*, with two GPUs from different brands (different performances), will be used to evaluate the load balancing on heterogeneous systems; *platform C*, comprising a state-of-the-art CPU and a latest generation GPU, will be used for performance comparisons with state-of-the-art alternative implementations.

5.2 Performance Evaluation

To quantitatively evaluate the performance offered by *BowMapCL*, its performance was compared against several mainstream mappers, including Bowtie [18], BWA [19] and SOAP2 [20] (CPU-based tools), as well as SOAP3/SOAP3-db [5], [28], HPG-BWT [6] and CUSHAW [7] (CUDA-based GPU tools). When compared with all these state-of-the-art tools, the proposed implementation proved to offer a significantly better exact string matching performance, as it is shown in Figs. 5 and 6. The comparisons were conducted by considering an **OCC** sample-rate of 1 and two-buffers assigned to the GPU device. The exact matching operation was executed over the Human Chromosome 1 using a wide range of query set sizes, each one with 100 characters (base pairs). All the considered tools were configured to perform the 'exact search' operation and to produce their results using the Sequence Alignment/Map (SAM) output format, by conveniently selecting the appropriate input parameters. They were also configured to use all the available CPU logical cores, thus using 8 threads when considering hyper-threading on an 4-core processor.

As it can be observed in Fig. 5, when the proposed *BowMapCL* tool makes use of one single GPU (NVIDIA GeForce GTX 780 Ti) and it is compared with the three most popular CPU BWT-based mappers (Bowtie, BWA and SOAP2) it presents speedups ranging from 2-5× (reduced amount of queries) and 3-7.5× (greater number of queries). The usage of greater amounts of queries allows to masquerade the starting time overhead for creating the OpenCL environment, as well as to fill the processing pipeline with operations of the exact string matching procedure.

Among the GPU-based mappers, SOAP3, HPG-BWT and CUSHAW were the selected tools for the conducted evaluation. Other mainstream tools (e.g., BarraCUDA) were

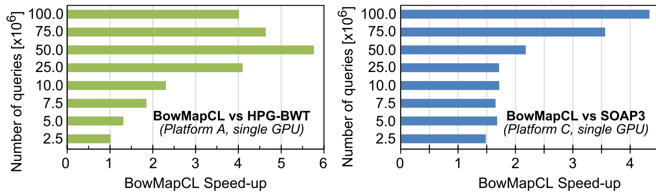

Fig. 6. Performance speedup provided by *BowMapCL* when compared with the GPU-based alternative tools (using the Human Chromosome 1 as reference).

excluded from this comparison, since they do not allow a pure exact mapping option (i.e., they do not offer the possibility of selecting the maximum number of mismatches). Moreover, since CUSHAW bears a considerable worse performance when compared to SOAP3 and HPG-BWT (approximately 12× slower, for queries datasets over 25 M), it was not considered in the performance chart depicted in Fig. 6. Furthermore, the comparison with HPG-BWT considered its intermediate output format (`"QUERY_ID: OCC_ID"`), since this tool does not output in the SAM format.

Two different platforms were used in this evaluation. Since HPG-BWT requires two equal GPU devices (although only one is used to implement the exact mapping in the forward strand version of the kernel), it was executed in platform A, where it was compared with *BowMapCL* (using one GPU). SOAP3 only supports a single GPU and was executed in platform C, where it was compared with *BowMapCL* (also using a single GPU). From the obtained results, it can be observed that *BowMapCL* is up to 4× and 5× faster, regarding SOAP3 and HPG-BWT, respectively.

To analyze the scalability of the proposed *BowMapCL* when varying the query length, its performance was evaluated and compared with SOAP3 on platform C. The results (presented in Fig. 7) allow to conclude that the peak performance difference occurs for queries with 25 or 35 characters (base pairs), where performance improvements of up to 7× are observed.

5.3 Performance Scalability

Fig. 8 presents the variation of the obtained performance with: *i*) the number of GPU devices, and *ii*) the number of buffers assigned to each GPU device (affecting the total number of consumer threads that assist the accelerators), when aligning 100 M queries against the *E. Coli* genome in platform A.

From the obtained results, it can be observed that the performance scalability of *BowMapCL* is almost perfect when using two GPU devices, with a speedup very close to 2×. In what respects the number of allocated buffers, the obtained results also show that by using more than one set of buffers

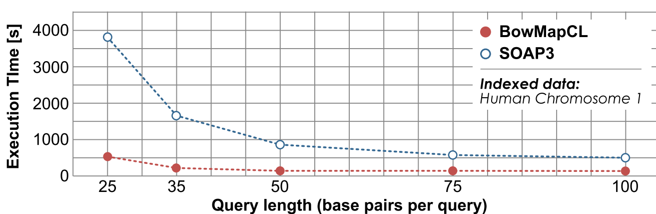

Fig. 7. BowMapCL performance scalability with the query length (using 100 M queries).

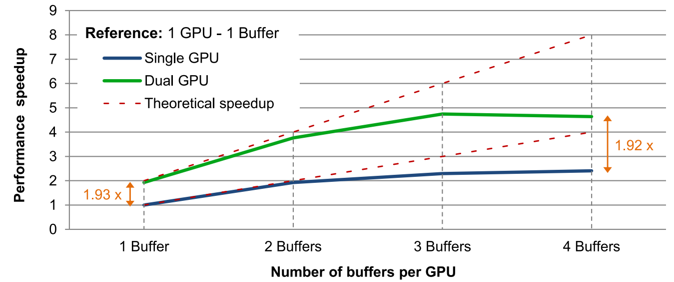

Fig. 8. Performance scalability with the number of GPU devices and with the number of buffers per GPU.

it is possible to overlap multiple concurrent operations in the same device (*e.g.*, host-device communication and kernel computation), resulting in a speed-up of around 2× when using two buffers. By using a higher number of buffers it is possible to further exploit the GPU spatial resources, guaranteeing that most of the stream-multiprocessors of the device have their blocks and threads working at maximum performance. Naturally, this gain is limited by an upper bound that is intrinsically related to the device. In fact, as soon as the full utilization is reached, increasing the number of buffers per device is no longer justified, because the attained performance increase is not significant (or can even be slightly degraded).

5.4 Load Balancing

The upper bars of Fig. 9 present an evaluation of *BowMapCL* load balancing capabilities when using two distinct accelerators from different manufacturers (NVIDIA and AMD). As it would be expected, due to their rather different intrinsic characteristics and architectures, they achieve different matching performances. To compensate for this difference, *BowMapCL* automatically adapts and unevenly distributes the workload (chunks of queries), in order to minimize the resulting processing time. To achieve such load balancing, *BowMapCL* further partitions the last chunk of queries with a finer granularity, guaranteeing that the devices terminate at the same time. As it can be seen in the obtained results, this is achieved with an insignificant difference, as low as 14 ms.

The bottom bars of Fig. 9 depict a different case study. Although the two used GPUs are entirely equivalent, one of them is also being used by another independent application kernel. As a consequence, the task partitioning algorithm perceives the performance of this GPU as being much lower and compensates the device contention, by dynamically distributing more chunks of queries to the other GPU. The last chunk is further divided in fine-grained partitions to

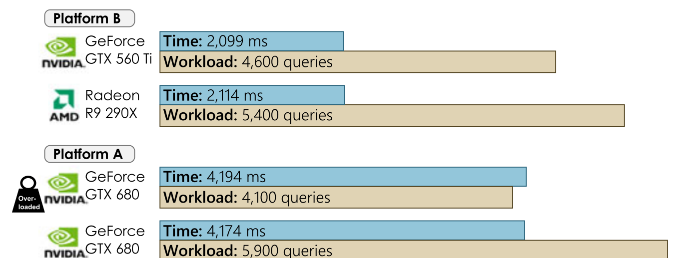

Fig. 9. Load balancing evaluation in heterogeneous and homogeneous platforms.

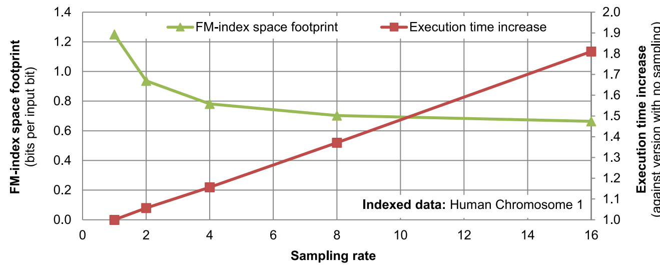

Fig. 10. FM-index data structure size and resulting performance variation of the string matching procedure for a variable OCC matrix sampling rate.

achieve an overall load balancing. As a result, both devices finish within a time interval of 20 ms.

5.5 OCC Index Sampling

As it was referred before, *BowMapCL* uses a conservative encoding of the BWT index data structure to reduce the resulting memory footprint in the GPU accelerators. In fact, although other alternative and more sophisticated schemes have been proposed in the literature, the adopted encoding allows a particularly regular reconstruction procedure that better suits the execution constraints imposed by GPU architectures.

However, despite its regularity, the adopted encoding still offers the user with the conventional parameterization and configuration means in terms of the sampling rate that is applied to the index structure. While the best performance is obtained without any sampling (i.e., by storing a sample value for each 32-bit word), this incurs in a considerably large FM-index data structure size (e.g., when processing DNA data (5 chars), it occupies 1.25 bits per original input bit). As a result, greater sampling rates are usually adopted. As depicted in Fig. 10, with a sampling rate equal to or greater than 2, the resulting ratio is already below 1 (the index takes less space than the input text).

Naturally, increasing the sampling rate has a negative effect on the kernel performance, as the GPU threads need to perform more operations for the computation of each $OCC(c, i)$ entry (see Fig. 10). Since the OCC bitmap always takes the same space ($m \times 5 \times 4/32 = 0.625 \times m$, for DNA) regardless of the adopted sampling rate (the extreme case will be to not store any sample), the minimum index size is limited by this value. Naturally, other data types (e.g. text) will incur in greater memory footprints.

As a comparison example, BWA creates an index data structure of 2.3 GB for a 3 GB genome [19], corresponding to a compression ratio of around 0.75. To attain a similar memory footprint, *BowMapCL* would use a sampling rate between 4 and 8, which would correspond to a consequent increase of the execution time by a factor between 1.15 and 1.37, thus still providing a significantly better matching throughput.

5.6 Native Popcount Function

As it was referred in section 4.4.2, whenever the target GPU (or the OpenCL API) does not feature a native *popcount* function, *BowMapCL* automatically considers an alternative implementation based on logic and shift operations. To evaluate the performance of such alternative, it was devised an experiment to compare the two *popcount* implementations. The obtained results (see Fig. 11) show that the achieved performance is very similar, leading to the conclusion that *BowMapCL* is always efficiently executed, independently on the availability of a native *popcount* function.

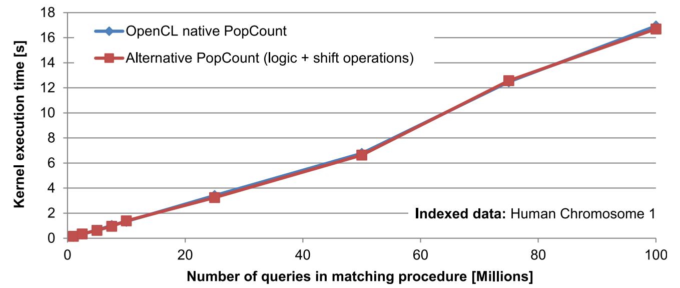

Fig. 11. Comparison between the OpenCL 1.2 native *popcount* function and the alternative implementation using logic and shift operations (platform C)—the two lines almost overlap.

5.7 SA Sampling

As it was described in Section 4.4.3, the suffix array value used for each occurrence previously identified by the GPU kernel is obtained in the CPU-side by recurring to the *last-to-first column mapping* property of the BWT. With such approach, the SA memory footprint can be reduced by r, when using a sampling rate r (see Fig. 12). However, as the sampling rate increases (the sampled SA becomes more sparse), the number of steps required for the computation of an absent sampled value introduces an increased overhead (linear with r), implying a convenient balance between memory footprint and execution time.

5.8 Tool Usability Benchmarks

To evaluate the usability of the proposed *BowMapCL* tool from a data-type agnostic perspective, a collection of datasets from a wider variety of application domains was also analyzed, including not only DNA, but also proteins and regular text. Such *BowMapCL* feature (i.e., the possibility to operate over a wide range of input data types) contrasts with most state-of-the-art tools, since the majority of them only process DNA or both DNA and proteins sequences, but not arbitrary text. Furthermore, the considered data sources were selected to cover a wide range of input sequence lengths and data type cardinalities. The corresponding parameters and resulting execution times are presented in Table 2. Each index was computed off-line, by using the `libdivsufsort` library (version 2.0) to construct the SA, which was subsequently used to obtain the BWT.

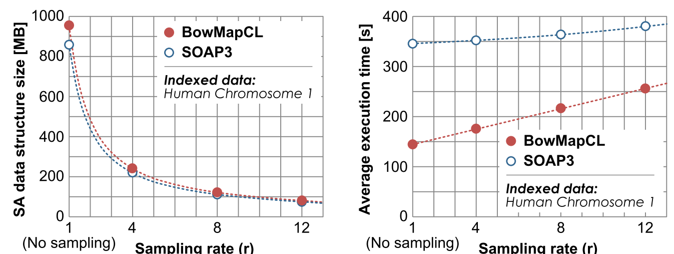

Fig. 12. SA memory footprint for a variable sampling rate, and resulting performance variation (mapping of 50 M queries of 100 characters each, in platform C).

TABLE 2
BowMapCL Benchmarking when Processing Different Input Data-Types, in Platform C

Parameter	E.coli genome	Human chromosome 1	Human genome	Titin protein (Human source)	Shakespeare play	Bible
Data type	DNA	DNA	DNA	Protein	Text	Text
Data type cardinality (including $)	4+1	4+1	4+1	20+1	127+1	127+1
Size of data [MBytes]	4.42	238	2958	0.03	0.12	4.19
Size of FM-index [MBytes]	5.53	297	3698	0.17	3.82	134
Resulting index partitions	1	1	3	1	1	1
Size of SA index [MBytes] @ SA sampling rate	17.70 @ 1	951 @ 1	2958 @ 4	0.13 @ 1	0.48 @ 1	16.77 @ 1
Index build time [s]	1.56	83.34	988.70	0.72	0.94	3.22
Execution time [s] for 10 M mapped queries	3.82	5.06	53.65	3.38	2.50	4.24
Execution time [s] for 100 M mapped queries	34.50	37.05	217.18	28.01	16.15	41.17

The considered queries for the protein and DNA sequences have a fixed length of 100 characters (base pairs) and were obtained by randomly extracting them from the reference sequences. Such option provides the worst-case performance evaluation of *BowMapCL*, since the implemented *early termination* feature of the considered backward search algorithm will result in shorter processing times for queries not occurring in the reference index. For the text inputs, the queries were also obtained from those same reference sequences but with variable lengths, ranging from 30 to 50 characters (base pairs), as finding considerable larger strings does not make that much sense, considering that their number of occurrences would be extremely low.

Among the several considered datasets, the Human genome is the one that presents more distinguishable results, and therefore, deserves a further discussion. Firstly, since its input size is in the order of the gigabytes, the resulting index cannot be stored in one single block in the target GPU. As a result, the index had to be partitioned in the smallest amount of partitions, such that the resulting equal-sized text blocks can be accommodated in the space reserved for the index data structure in the target GPU global memory. As a result, the matching operation must be repeated three times (one for each partition). Furthermore, due to its large memory footprint, the suffix array was sampled with a sampling rate of 4, leading to a sampled data structure with exactly the same memory footprint as the original reference sequence. One consequence of such partitioning and sampled SA structure is an observed overhead in the execution time (when compared with the matching of the same number of queries against the other considered DNA indexes). Naturally, such overhead is aggravated by the larger index structure of this dataset (takes more time to be loaded into the host main memory).

Too alleviate the overhead introduced by the read and write operations, the benefits of using a Solid State Drive (SSD) to accommodate the sequences data were also assessed. When processing the smaller genomes, the index reading overhead is negligible, resulting in an insignificant effect. However, when larger genomes are processed (e.g., Human genome), the reading of the index partitions can be substantially reduced by more than 15× (from about 45-60 seconds to only about 3 seconds), thus completely mitigating the overhead of reading the index. The same benefit was also observed for the writing operations that are required in the final step (SA conversion and writing of solutions to file). Hence, the usage of SSD technology has shown to be a highly viable means to make the resulting execution time almost solely dependent on CPU/GPU computing and communication (mainly when large datasets are considered).

6 Discussion and Conclusions

This paper proposes *BowMapCL*, a fast OpenCL-based exact string mapping tool, based on the Burrows-Wheeler Transform and FM-Index, and targeting highly heterogeneous platforms composed by multiple OpenCL accelerators. The tool is offered with a set of user-defined parameters that enable an extensive customization of the desired trade-off between performance and index memory footprint reduction. As a result, it can be executed in a broad range of GPU accelerators, not presenting any relevant restrictions on the host memory size (RAM) nor on the global memory of the GPUs, which contrasts with the majority of the currently available tools. Furthermore, due to the implemented index partitioning and sampling schemes, the proposed tool allows the processing of any reference text, independently of its dimension. When compared with the current state-of-the-art BWT-based mapping tools, the proposed *BowMapCL* is up to 4× faster than the best performing GPU-based tools, and from 2× to 7.5× faster than mainstream CPU-based tools, using a single GPU device.

Acknowledgments

This work was partially supported by national funds through Fundação para a Ciência e a Tecnologia (FCT), under projects Threads (ref. PTDC/EEA-ELC/117329/2010), P2HCS (ref. PTDC/EEI-ELC/3152/2012) and project UID/CEC/50021/2013.

References

[1] L. Wang and T. Jiang, "On the complexity of multiple sequence alignment," *J. Comput. Biol.*, vol. 1, no. 4, pp. 337–348, 1994.
[2] M. J. Fischer and M. S. Paterson, "String-matching and other products," MIT, Cambridge, MA, USA, Tech. Rep. MIT-LCS-TM-041, 1974.
[3] M. Burrows and D. J. Wheeler, "A block-sorting lossless data compression algorithm," Digital Equip. Corp., Maynard, MA, USA, Tech. Rep. 124, May 1994.
[4] P. Ferragina and G. Manzini, "Opportunistic data structures with applications," in *Proc. 41st Annu. Symp. Found. Comput. Sci.*, 2000, pp. 390–398.

[5] C.-M. Liu, et al., "SOAP3: Ultra-fast GPU-based parallel alignment tool for short reads," *Bioinf.*, vol. 28, no. 6, pp. 878–879, 2012.

[6] J. Salavert Torres, I. B. Espert, A. T. Dominguez, V. Hernendez, I. Medina, J. Terraga, and J. N. Dopazo, "Using GPUs for the exact alignment of short-read genetic sequences by means of the Burrows-Wheeler transform," *IEEE/ACM Trans. Comput. Biol. Bioinf.*, vol. 9, no. 4, pp. 1245–1256, Jul. 2012.

[7] Y. Liu, B. Schmidt, and D. L. Maskell, "CUSHAW: A CUDA compatible short read aligner to large genomes based on the Burrows-Wheeler transform," *Bioinf.*, vol. 28, no. 14, pp. 1830–1837, 2012.

[8] P. Klus, et al., "BarraCUDA-a fast short read sequence aligner using graphics processing units," *BMC Res. Notes*, vol. 5, no. 1, p. 27, 2012.

[9] L. Colussi, Z. Galil, and R. Giancarlo, "On the exact complexity of string matching," in *Proc. 31st Annu. Symp. Found. Comput. Sci.*, 1990, pp. 135–144.

[10] D. E. Knuth, J. H. Morris, Jr, and V. R. Pratt, "Fast pattern matching in strings," *SIAM J. Comput.*, vol. 6, no. 2, pp. 323–350, 1977.

[11] A. Hume and D. Sunday, "Fast string searching," *Softw.: Practice Experience*, vol. 21, no. 11, pp. 1221–1248, 1991.

[12] S. F. Altschul, et al., "Gapped blast and PSI-blast: A new generation of protein database search programs," *Nucleic Acids Res.*, vol. 25, no. 17, pp. 3389–3402, 1997.

[13] H. Li, J. Ruan, and R. Durbin, "Mapping short DNA sequencing reads and calling variants using mapping quality scores," *Genome Res.*, vol. 18, no. 11, pp. 1851–1858, 2008.

[14] U. Manber and G. Myers, "Suffix arrays: A new method for On-line string searches," in *Proc. 1st Annu. ACM-SIAM Symp. Discr. Algorithms*, 1990, pp. 319–327.

[15] S. J. Puglisi, W. F. Smyth, and A. H. Turpin, "A taxonomy of suffix array construction algorithms," *ACM Comput. Surveys*, vol. 39, no. 2, pp. 1–31, Jul. 2007.

[16] P. Weiner, "Linear pattern matching algorithms," in *Proc. 14th Annu. Symp. Switch. Autom. Theory*, 1973, pp. 1–11.

[17] A. Chacon, et al., "FM-Index on GPU: A cooperative scheme to reduce memory footprint," in *Proc. IEEE Int. Symp. Parallel Distrib. Process. Appl.*, Aug. 2014, pp. 1–9.

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

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

[20] R. Li, et al., "SOAP2: An improved ultrafast tool for short read alignment," *Bioinf.*, vol. 25, no. 15, pp. 1966–1967, 2009.

[21] H. Lin, X. Ma, W. Feng, and N. F. Samatova, "Coordinating computation and I/O in massively parallel sequence search," *IEEE Trans. Parallel Distrib. Syst.*, vol. 22, no. 4, pp. 529–543, Apr. 2011.

[22] Y. Liu, A. Wirawan, and B. Schmidt, "CUDASW++ 3.0: Accelerating Smith-Waterman protein database search by coupling CPU and GPU SIMD instructions," *BMC Bioinf.*, vol. 14, no. 1, p. 117, 2013.

[23] J. Guerreiro, A. Ilic, N. Roma, and P. Tomás, "Multi-kernel Auto-tuning on GPUs: Performance and Energy-aware optimization," in *Proc. Euromicro Int. Conf. Parallel, Distrib., Netw.-Based Process.*, Mar. 2015, pp. 438–445.

[24] M. I. Abouelhoda, S. Kurtz, and E. Ohlebusch, "Replacing suffix trees with enhanced suffix arrays," *J. Discr. Algorithms*, vol. 2, no. 1, pp. 53–86, Mar. 2004.

[25] A. Drozd, N. Maruyama, and S. Matsuoka, "A multi GPU read alignment algorithm with model-based performance optimization," in *Proc.10th Int. Meet. High-Perform. Comput. Comput. Sci.*, 2013, pp. 270–277.

[26] H. S. Warren, *Hacker's Delight*. New York, NY, USA: Pearson Edu., 2003.

[27] D. Kopczynski and S. Rahmann, "A memory-efficient data structure for pattern matching in DNA with backward search," in *Proc. 17th German Conf. Bioinf. (GCB'2013)—Poster*, 2013.

[28] R. Luo, et al., "SOAP3-dp: Fast, accurate and sensitive GPU-based short read aligner," *PLoS ONE*, vol. 8, p. e65632, May 2013.

David Nogueira received the BSc and MSc (electrical and computer engineering) degrees in 2012 and 2014, respectively, from the Instituto Superior Técnico (IST), Universidade de Lisboa, Lisbon, Portugal. His current research interests include parallel processing and high-performance computing, bioinformatics, machine learning, artificial intelligence, robotics, optimization, and algorithms.

Pedro Tomás received the five-year engineering degree, and the MSc and the PhD degrees in electrical and computer engineering from the Instituto Superior Técnico (IST), Technical University of Lisbon, Portugal, in 2003, 2006, and 2009, respectively. He is currently an assistant professor in the Department of Electrical and Computer Engineering, IST and a researcher at the Signal Processing Systems Group (SiPS), Instituto de Engenharia de Sistemas e Computadores R&D (INESC-ID). His research activities include specialized computer architectures, high-performance computing and signal processing for biomedical applications, bioinformatics, and computational chemistry. He is a member of the IEEE Computer Society and has contributed to more than 45 papers to international peer-reviewed journals and conferences.

Nuno Roma received the PhD degree in electrical and computer engineering from the Instituto Superior Técnico (IST), Universidade de Lisboa, Lisbon, Portugal, in 2008. He is currently an assistant professor in the Department of Electrical and Computer Engineering of IST and a senior researcher at the Signal Processing Systems Group (SiPS) of Instituto de Engenharia de Sistemas e Computadores R&D (INESC-ID). His research interests include computer architectures, specialized and dedicated structures for digital signal processing (including image and video coding and biological sequences processing), parallel processing, and high-performance computing systems. He contributed to more than 80 papers to journals and international conferences. He is a senior member of the IEEE Circuits and Systems Society and a member of the ACM.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.