Accelerating a BWT-based exact search on multi-GPU heterogeneous computing platforms

David Nogueira
INESC-ID, Instituto Superior Técnico, Universidade de Lisboa
Rua Alves Redol, 9, 1000-029 Lisbon, Portugal
e-mail: david.jacome.nogueira@tecnico.ulisboa.pt

Abstract—The computational demand of exact search and mapping procedures as well as the gradual decrease of the cost of new sequencing technologies have pressed the exploitation of parallel processing accelerators to reduce the execution time of many bioinformatics applications. However, this often imposes strict restrictions in terms of the problem size and the implementation efforts, mainly due to their possibly distinct architectures. The work herein presented describes the design of a solution for this problem, using an heterogeneous multi-device parallelization model. Circumventing these limitations, a new exact-search mapping tool (BowMapCL) based on the Burrows-Wheeler Transform and the FM-Index was implemented. Contrasting to other alternative solutions, BowMapCL is based on a unified implementation using the OpenCL API, allowing the exploitation of multiple and possibly different devices (e.g., NVIDIA, AMD/ATI, Intel GPUs/APUs). Some techniques and enhancements are investigated and implemented, such as multiple buffering, dynamic load-balancing, producer-consumer co-processing, index bit encoding and sampling, and index partitioning, in order to efficiently explore the processing power of such heterogeneous computing architectures. When compared with state-of-the-art tools, the attained results showed that BowMapCL, using a single GPU, offers more than a 10x speedup over mainstream multi-threaded CPU BWT-based aligners, like Bowtie, BWA and SOAP2; and speedups between 1.5x and 5x when compared with the best performing state-of-the-art GPU implementations (namely, SOAP3 and HPG-BWT). When multiple and completely distinct devices are considered, BowMapCL proved to efficiently scale the offered throughput, ensuring a convenient load-balance of the involved processing in the several distinct devices.

Index Terms—Exact String Matching, Burrow-Wheeler Transform (BWT), FM-index, Parallel Computing, Graphics Processing Unit (GPU), OpenCL.

I. INTRODUCTION

Bioinformatics is an extremely active research area where biological data, such as DNA and proteins, is processed using computing techniques. Bioinformatics applications often comprehend computationally intensive search and matching procedures, where a considerable amount of reads (e.g., short DNA sequences) have to be mapped or aligned to a longer reference or database sequence (e.g., species genome, chromosome), as most of the information is obtained by homology [1]. When no errors are allowed, this sequence mapping operation can actually be regarded as a regular exact string matching problem, which goes far beyond the bioinformatics domain and actually influences many other computer science fields, such as pattern recognition, document matching and text mining. Under this broader context, exact string matching is regarded as a pattern matching problem, in which the desired output is a list of all the occurrences of a short sequence of characters (called pattern or query) of length \( n \), in a long reference sequence of characters (string) of length \( m \), such that \( n \ll m \) [2].

Under this assumption, several alignment and mapping tools are already widely available, by making use of different algorithms. One of the most prominent approaches to perform exact string search with full-text indexes is based on the combination of the Burrows-Wheeler Transform (BWT) [3] and the Ferragina-Manzini Index (FM-index) [4], making use of a backward search that mimics the traversing of a tree data-structure without its underlying memory footprint. When compared to other full-text indexes algorithms and related data structures, this approach offers significant performance improvements, since features a considerable small memory footprint and its inherent computational complexity is only dependent on the length of the substring (query) that is being processed (\( O(n) \) search time) and does not depend on the length of the reference/database sequence.

To mitigate the significant demands that are imposed by these computational intensive procedures, the usage of high-performance computing resources is also mandatory and several bioinformatics applications have been accelerated by exploiting the inherent data and functional parallelism. For such purpose, multi-CPU, GPU or even special-purpose dedicated processing structures have been regarded as a highly viable alternative 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 and implementation procedures.

Due to its programmable nature and attainable performance, GPU solutions have been widely explored for exact string matching. As a result, a significant number of BWT-based tools actually exploit the computational power of GPUs, with a particular predominance of NVIDIA CUDA-enabled GPUs, such as SOAP3 [5], HPG-BWT [6], CUSHAW [7] and BarraCUDA [8]. However, contrary to conventional dynamic programming alignment procedures, which have a uniform data access pattern, this indexed exact string search is characterized by an unpredictable and highly irregular memory access pattern, which poses difficult challenges for efficient implementation in GPU devices. Nevertheless, the added alignment...
efficiency that is offered by these indexed approaches, allied with the vast parallel processing capabilities of GPUs, have deserved a considerable attention of the research community in the past few years. Furthermore, an important step forward still needs to be fulfilled to efficiently exploit multiple and possibly different devices, which are now currently available in the latest generation of heterogeneous platforms (e.g., Intel and AMD CPUs, NVIDIA and AMD/ATI GPUs, and even FPGA-based accelerators). However, scaling the exploited parallelization with the number of available devices raises important challenges in terms of scheduling and load-balancing.

Accordingly, this paper envisages the development of a new software tool, BowMapCL, a successful parallelization of an exact search BWT-based procedure, that allows an efficient execution on heterogeneous platforms based on multiple CPUs and GPUs. Some techniques and enhancements, such as multiple buffering, dynamic load-balancing, producer-consumer co-processing, index bit encoding and sampling, and index partitioning, were investigated and implemented, in order to efficiently explore the processing power of such heterogeneous computing architectures.

II. ALGORITHMIC BACKGROUND

The considered indexed search procedure is based on the BWT [3], since it is one of the prevailing full-text indexes with lower memory footprint and allows for fast exact string matching, by using the FM-index [4] enhancement.

A. Burrows-Wheeler Transform and FM-Index

Let A be the alphabet of a text T with length m and $ a $ symbol not present in A, which is lexicographical smaller than every other symbol. The Burrows-Wheeler Transform of text T will start by appending $ a $ to $ T $. $ T[i] $ is defined as the suffix of T that starts on the $ i^{th} $ position and ends on the last character of the appended text. The BWT can then be obtained after the computation of the corresponding suffix array (SA). The SA is a vector of integers, in such a way that they correspond to the indexed positions of all the suffixes in the text, after they have been lexicographically sorted. Accordingly, the $ i^{th} $ smallest suffix will have its starting index on the $ i^{th} $ entry of the SA. The relationship between the BWT and the SA is formulated as $ BWT[i] = T[S\cdot A[i] - 1] $.

With this approach, the BWT can be computed with a complexity equal to the one of building a suffix array. For such purpose, BowMapCL makes use of the libdivssect library [9], that constructs the suffix array in $ O(n \times \log(n)) $ time, and then uses the aforementioned relationship to construct the BWT.

To construct the FM-index, two additional data structures were proposed by Paulo Ferragina and Giovanni Manzini: the C vector and the OCC matrix. $ C(a) $ represents the occurrences count of all lexicographically smaller characters than a in the text, with $ a \in A $. $ OCC(a, i) $ entry represent the number of occurrences of character a in the prefix $ BWT(T')[i] $.

B. Backward Search algorithm

By using these two auxiliary 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, as described in [4] (see Algorithm 1). As an example, Figure 1 illustrates the application of this procedure to find all the occurrences of the ssi substring in the mississippi reference text. In each represented step, the FIRST and LAST coordinates in the BWT matrix that were obtained after the application of each iteration of the backward search algorithm were represented with the F and L arrows, respectively. The procedure evolves backwardly, by selecting the previous character of the substring and by appending it to the searched suffix.

The resulting output of this procedure is a set of intervals of text coordinates [FIRST, LAST], corresponding to all the occurrences of each substring. This is consistent with the fact that if a substring occurs in the text, the indexes of those occurrences appear contiguously in a suffix array interval, as those suffixes would have that substring as their prefix. Hence, by using both the suffix array and the BWT+FM index data structures, the desired solution indexes can be retrieved from the SA interval with $ O(1) $ time. Whenever the LAST value happens to be lower than the FIRST value, it means that the substring does not occurs in the reference text.

III. IMPLEMENTATION

The proposed tool was partitioned in two main modes of execution: the first one, the index generator, receives an input reference file and creates the necessary data structures for the string matching procedures. Its corresponding BWT is generated, as well as the OCC matrix, C vector and suffix array. The second main mode corresponds to the exact string matching procedure, and it is the mode that is the subject of the presented study and development efforts.

This section will be divided in three logical parts. The first part of this section describes the existing parallel opportunities in this matching procedure, and how the parallelization of the procedure is conducted. The second part concerns the implementation details of the exact search in GPU architectures, in order to maximize the attained throughput and scalability. Finally, the third part describes the conducted enhancements to the data structures of the index.

A. Parallel processing paradigms in the matching procedure

Since the number of queries to align is usually considerable large, the goal is to harness the computational power of GPUs

Fig. 1. Backward search of the substring ssi on the mississippi word.
Algorithm 1 Backward search algorithm (pseudo-code).

1: procedure BACKWARD SEARCH([[J], [C], OCC] [[I]])
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]
7:   while (FIRST < LAST) AND (i ≥ 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
10:    LAST := C[c] + OCC[c][LAST] + 1
11:   i := i - 1
12: if LAST < FIRST then
13:   return not found
14: else
15:   return values of indexes between FIRST and LAST
16: end if
17: end for
18: end procedure

Another parallelism paradigm to be taken into account is the task-level parallelism, as there are multiple different tasks that can be performed in parallel, not only on the same device, but across multiple ones (intra-device and inter-device task parallelism). These tasks range from purely I/O operations (namely reading data/writing solutions from/to multiple files and transferring data between host-side memory and targeted devices global memories) to computational ones (such as executing the string matching kernel and converting the backward search output to the desired solutions).

B. Proposed parallelization approach overview

As it was previously described, the goal of exact string matching is to find all the occurrences of a list of queries in a reference string. This search procedure is implemented by using a previously computed index structure for the considered reference input. To be able to execute such procedure in OpenCL enabled accelerators, the exact string matching algorithm must be defined in a kernel (kernels are functions that execute on OpenCL devices). Algorithm 2 presents this kernel pseudo-code.

The program starts by loading to the devices memory the blocks of the index and queries from disk. Then, it executes processing the queries in batches. For each pair of index block and chunk of queries, the exact string matching kernel is launched on each device, as seen in Figure 2. During this step, thousands of queries are matched to the reference string index in parallel, in the multiple processors of the targeted accelerator. The output of the kernels is copied back to the host memory and the desired solutions are written to the disk after being converted with the suffix array. To issue these operations on the host side for each OpenCL device, the proposed program will be composed of several device-assistant threads (as further explained in subsection III-C).

C. Computational and I/O task parallelism

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 idle most of the time. Moreover, whenever the targeted accelerating device does not support OpenCL non-blocking I/O operations, the CPU would be blocked, waiting for the completion of the write and read of data. In fact, the usage of blocking operations with a single 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 and copy of results from the accelerator to host main memory). As a consequence, the usage of multiple CPU threads (at least one per accelerator) is highly required, in order to ensure a constant data processing flow.

To circumvent this problem, the proposed program architecture is comprised of several CPU threads to coordinate and orchestrate each accelerator, as well as to perform the necessary I/O operations, as it is illustrated in Figure 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 task queue (i.e., a pool of queries) in the host memory. Although there are multiple consumers (at least one per accelerating device, as further explained in the following subsection), only one producer is created, as its operation tends to be much less time demanding than the consumers operation. Each CPU thread that assists each accelerator (threads #1 to #4 in Figure 3) needs to have its own OpenCL command queue and 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 finally, for 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 (as it can be seen in Figure 8). 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 a maximization of the usage of the computing resources of the CPU and of 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 only one 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.

D. On-device computation/communication overlap

Despite being able to scale the throughput with the number of GPUs, the usage of a single thread per device would not allow overlapping the data transfers with the kernel computations. The reason mainly lies in the fact that the execution is faced with a structural dependency, since the same buffers would be used in all iterations (e.g., the CPU thread that assists that OpenCL device can not send the queries of the next iteration to the accelerator global memory, while the kernel is still running, as the input buffer with the input queries is still being used). To circumvent this problem, a multiple buffering technique must be considered and more threads created. The proposed implementation of the 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 Figure 3).

These multiple sets of buffers on each device are concurrently used, each set being 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 number of sets of buffers created in each device. As an example, in a platform with 2 GPUs and 2 sets of buffers per GPU (i.e., using a double buffering technique, such as the one in Figure 3), it should assign 4 threads 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 be copying new queries to the other input buffer or retrieving the solutions of the previous kernel execution. With this approach, the cost of communication can be completely hidden, overlapping the data transfers and the computation for each target device.

E. Kernel output conversion

Due to its large space footprint (i.e., 4 times the size of the original text) the suffix array is kept on the host memory, where memory usage is typically less constrained than in the accelerating devices. As a result, it is the set of host-side 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 a procedure that 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 situation 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, this conversion of the solution space was implemented on the CPU, which has many processing units and whose computational resources are not yet fully utilized by the assistant threads.

F. 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 programs that indirectly affect their execution times (device contention), since the program can be competing for the resources with other running programs. 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 distribution is based on the usage of a Producer-Consumer scheme (see Figure 3), where a single producer fetches chunks of queries (of equal size) 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. By using such solution, 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 can also be 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, and thus reducing the imbalance penalty (i.e., trying to reduce the gap between each device execution time).

G. GPU kernel implementation and mapping

Despite the fact that the indexed-based backward search (see Section II) has a highly irregular pattern of memory accesses and is inherently sequential (the iterations of the while cycle present data dependencies), this exact string matching procedure can be easily parallelized by processing each substring independently. In fact, although the internal loop corresponding to the search of a possible match for each query is inherently sequential, the outer loop is completely parallelizable, as each query is independent and the same instructions can be executed over different data. As it can be observed, there is still a close similarity between the code of the single-threaded CPU implementation (see Algorithm 1 in Section II) and the OpenCL parallel kernel (see Algorithm 2).

However, there are two major differences worth noting. One is concerned with the mapping of the queries into the accelerator processing elements and the other with the usage of certain types of GPU memories to store the data structures.

Contrary to Algorithm 1, where the executing thread loops around every query, each thread in the GPU search procedure (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 any complex synchronization scheme. 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.

There is, however, a specific synchronization point that is required to be able to use the local memory and therefore, ensuring the maximum efficiency of the kernel. In fact, typical GPU accelerators have different memory elements, each with different characteristics. Naturally, local memories normally provide data to the processing elements with a much higher bandwidth and lower access latency than global memories. Therefore, whenever possible, the data structures should be copied to and accessed from the local memory, provided that the number of accesses to their entries justifies the inherent penalty of copying them to local memory before the actual execution. As an example, the OCC matrix is very expensive in terms of memory resources, making an eventual copy to the local memory rather impracticable (largely exceeds the available local memory size of typical accelerating devices). In contrast, the C vector is a very good candidate, since its size is usually lower than 1KB, even in worst case conditions. In practice, it only accommodates a number of integers equal to the number of distinct characters in the indexed text \( T \), it should be recalled once more that the input data type can be DNA, proteins or arbitrary text.

To compute the queries occurrence range, on each access to the OCC matrix, the thread needs to index the row corresponding to a certain character. To rapidly identify the row where the data corresponding to a specific character is stored, another data structure is used, herein denoted as character_map. This data structure has 256 entries (corresponding to all the possible 8-bit chars). Hence, whenever a character

\begin{algorithm}[h]
\caption{OpenCL backward search kernel implementation (pseudo-code).}
\begin{algorithmic}[1]
\Procedure{backward search kernel}{queries[, ]}, queries\_sizes[, ], C[, ], OCC[, ], character\_Map[, ], numqueries}
\State \textbf{procedure Backward Search Kernel}(queries[, ], queries\_sizes[, ], C[, ], OCC[, ], character\_Map[, ], numqueries)
\State \textbf{global}_index \text{	extasciitilde= get work-item global id}
\State \textbf{local}_index \text{	extasciitilde= get work-item local id}
\If{\text{global}_index < \text{numqueries}}
\State \textbf{i} := size of query\[global\_index\]
\State \textbf{c} := last character of query\[global\_index\]
\State \textbf{FIRST} := \text{C}[\text{c}] + OCC(\text{c}, FIRST - 1) + 1
\State \textbf{LAST} := \text{C}[\text{c}] + OCC(\text{c}, LAST)
\State \textbf{i} := \text{c} - \text{C}[\text{c}]
\EndIf
\State \textbf{queries}\text{FIRST}[global\_index] \text{:= FIRST}
\State \textbf{queries}\text{LAST}[global\_index] \text{:= LAST}
\EndProcedure
\end{algorithmic}
\end{algorithm}
exists in the text, this data structure stores the number of its assigned row in the OCC matrix. Since this structure is directly indexed with the character that is being searched for, it allows to obtain the required row with a complexity of $O(1)$. Hence, the character Map data structure, as well as the $C$ vector, are both copied from the global to the local memory before the exact search begins. Also before the search procedure begins, all threads in the same work-group wait for the others, to ensure the consistency of the local memory corresponding to these local data structures.

At this point, it is also important to recall that the maximum resulting 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 modelling should be executed, in order to determine the most efficient configuration.

### H. Memory usage and index partitioning

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] [10]), a special attention must still be given to reduce the memory footprint. To attain such objective, many CPU-based implementations take advantage of compression techniques to mitigate this problem. Algorithms, such as run-length and statistical encoders (e.g., Huffman or arithmetic coders) and dictionary coders (e.g., LZ77 and LZ78), are commonly used to address this problem. As it was pointed out by Ferragina and Manzini [4], 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.

As a result, the majority of the implementations combine the BWT with move-to-front encoding (MTF), run-length encoding (RLE) and variable-length prefix code compression [4]. However, despite the huge compression factors of the index data that is provided by these techniques, they can hardly be efficiently implemented in GPUs, as the procedure of retrieving the data from the compressed text is highly irregular and not suited to the underlying GPU architecture, since the access to the compressed data usually incurs in significant processing overheads that also depend on the particular section of the index data being retrieved. In fact, the problem with algorithms like these, that present a significant amount of memory accesses with a very irregular pattern is that their implementation is easily memory bounded and hence, limited by the ability of the memory to provide data with complex patterns to the processing elements.

With the knowledge that it is of the utmost importance to reduce the memory footprint overhead, implementation-driven solutions are proposed, namely a bit-encoding data structure for the OCC matrix and sampling procedures for the OCC matrix and SA vector, that will be described in the next subsections. Nevertheless, besides these strategies to reduce the memory footprint, very large index structures can still not be accommodated in the internal memory system of most accelerating devices (e.g., GPUs). Accordingly, it was decided to implement an index partitioning scheme, where the index is divided in smaller blocks of indexes, creating a really scalable solution that can handle any input text, regardless of its size. However, as the queries are matched against each index block, although the search procedure complexity does not depend on the input reference size, the execution time increases linearly on the number of generated index blocks.

The size of the considered blocks can be either automatically computed by the tool, to adjust to the size of the global memory 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 to the host RAM at each iteration). With this added feature, the program does not require any particular GPU memory size nor any CPU RAM memory specification, independently of the size of the considered reference input. This computational resources independence is also applied to the queries (as already explained), as the tool allocated memory space is independent on the number of reads to be aligned, due to the fact that they are read on-the-fly, in chunks of queries, with a pre-defined size.

At this respect, two search approaches could have been considered (as pointed out in [11]): sequentially processing index blocks and matching all the queries against each block; or selecting chunks of queries, and searching them in all index blocks, repeating the process for every chunk of queries. Although the latter approach allows to store the solutions of each query contiguously in the same output file, the communication overhead is 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. As a result, the first option was adopted.

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 accelerator assistant thread is responsible for independently matching all the queries to its part of the index; ii) the devices share the same index block and the accelerator assistant threads concur to divide the queries that should be distributed and matched to their block. However, although the first approach appears 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 latter has the advantage of being more suited for a dynamic load-balancing application. In fact, since in the first case each device has its unique index block with which it maps every query, 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.

Before concluding, an important remark should be made concerning the partitioning of the index and of the input text. In fact, a special care must be taken in order to preserve the possibility to search for substrings that are now being split across two contiguous blocks of the input text. To accomplish
this, each block overlaps with the end of the previous one, in
such a way that the last characters appear in both blocks, as
depicted in Figure 4.

I. OCC matrix bitmap encoding

The most memory demanding data structure from the FM-
index is the OCC matrix (see Section II), whose access
time to obtain \( OCC(c, i) \) significantly affects the backward
search procedure. Each \( OCC(c, i) \) entry of this matrix con-
tains the number of occurrences of character \( c \) in the prefix
\( BWT(T')[:i] \). Hence, in a straightforward implementation,
this value could be obtained with just one memory access,
therefore resulting in a \( O(1) \) complexity. However, this sim-
plicity would be offered at the cost of a significant amount
of required memory: a matrix of integers of size
\( (m+1) \times 5 \times m = 20 \times m \), with \( 5 \) being the number
of distinct characters plus the appended symbol $ at the end
of the string.

By taking the memory restrictions of the GPUs into account,
an alternative to this very large data structure was considered.
It consists of an OCC matrix bitmap, where each \( OCC(c, i) \)
entry is a boolean variable indicating whether character \( c \)
occurs on the \( i \)th position of the index (instead of storing the
actual number of occurrences up to that position). These boolean
values are then packed in 32-bit words (integers). To prevent
counting \( i \) bits when obtaining \( OCC(c, i) \), the OCC
matrix is sampled at certain specific positions of the index
and a subset of the entries is also stored. Thus, whenever a
given \( OCC(c, i) \) value is required, the previous sample must
be retrieved and the bits between that sampled position and
the desired one must be counted. To perform such operation,
a \texttt{popcount} function is used to count the number of ones in
each 32-bit word.

To maximize the performance, a sample should be stored
for each integer, i.e., for each 32-bit word (that holds 32
boolean values). Thus, to obtain any \( OCC(c, i) \) value, only
two memory accesses to the data structures are needed: one
to get the previous sample value and one to get the bit-encoded
word. A mask is then applied to the latter one, in order to turn
to zeros any ones that may exist in positions after the desired
one, so that the \texttt{popcount} operation can be performed over
the whole 32-bit word. Then, the two values (sample and \texttt{popcount}) are added and the number of occurrences is obtained.
Such approach reduces memory space by 16 times, when
compared with the straightforward approach. When dealing
with DNA, the memory space cost is of 1.25 bytes for each
byte of the input text, as opposed to the cost of 20 bytes per
original byte of text in a straightforward approach, as already
discussed.

J. SA sampling and on-the-fly computation

By definition, the suffix array is a vector of integers with
length equal to the input reference text. Therefore, with no
modifications, it would occupy four times the size of the
input text (due to the conversion of chars to integers, i.e.,
the information of each byte is now stored in four bytes). Due
to its memory footprint, this data structure is stored in host
memory and processed by the CPU (and not in the GPUs).
However, for large input data files its size can still be a
problem. Thus, as it happens for the OCC matrix, the suffix
array is also sampled.

If, when trying to obtain a given value of the sampled
suffix array, that desired index is not found, a loop must
be performed, visiting every previous letters in the original
reference text, till one of them is a stored sampled value. To
obtain the previous letter in the text, two properties of the
BWT are necessary:

- On each row of the rotation matrix \( M \) \((m+1) \times (m+1)
  \) conceptual matrix with all the possible rotations of the
  text \( T' \), its last character precedes its first character
  in the text \( T' \). The first column of \( M \) will be designated as
  vector \( F \) and the last one \( L \). Considering the \( i \)th row, it is
  known that at some point in \( T' \), the sequence \( ...L[i]F[i]...
  \) must occur.

- Considering the \( i \)th row of \( M \), let \( c \) be the character in the
  position \( L[i] \). Let \( rank_c \) be the rank of the \( i \)th row
  of \( M \) among all the rows that also end with the character
  \( c \). Let the \( j \)th row be the \( rank_c \) row of \( M \) that starts
  with the character \( c \). Then, the character in the position
  \( L[i] \) corresponds to the same character in the position
  \( F[j] \) (last element of the \( i \)th row and first element of the
  \( j \)th row are the same). This property is called \textit{last-to-
  first column mapping} or just \textit{LF-mapping} (as described in [4]).

With that knowledge, it is known that the last character of
a row of \( M \) precedes the first character of that row in the text.
Therefore, to obtain the previous letter of the desired index
value in the text, it is only needed to find 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 (i.e., vector \( L \)). This
operation is performed by recurring to the \textit{last-to-first column
mapping}, whose equation is presented in equation 1:

\[
LF(i) = C[L[i]] + OCC(L[i], i)
\]  

A cycle is then created, looping over the index positions,
visiting the previous character of the one last visited, and
counting the number of hops. At the end, the number of hops is
added to the position of the first sample encountered, and that
value is returned. A special care must be devised to guarantee that such value indexes a position inside the vector. Whenever the returning value is greater than the size of the vector, the remainder of the division is returned.

IV. EXPERIMENTAL EVALUATION

A. Datasets and computing platforms

To experimentally assess the performance offered by the proposed BowMapCL tool, a set of DNA datasets was used, comprising the E. Coli complete genome (single chromosome, with around 4.6 MB), the Homo Sapiens complete genome (over 3 GB) and its Chromosome 1 (around 200 MB). The considered queries were randomly extracted from these species genomes, against which they were matched.

The presented results were obtained on three different platforms. Platforms (A) and (B) are used due to their GPU specifications, namely: (A) holds two NVIDIA GPUs of the same model, making it best suited for testing the tool scalability on the number of devices, and (B) holds two GPU from different brands, with different performances, which allows to better test the load balancing of heterogeneous systems. The last one, (C), including a state-of-the-art CPU and a latest generation GPU, is the one used for the most part of testing and results. Their specifications are as presented: (A) dual quad-core Intel Xeon E5-2609 CPU (8 cores) at 2.40GHz, 32GB of RAM and two NVIDIA GeForce GTX 680 GPUs with 4GB of memory; (B) quad-core Intel Core i7-3820 CPU at 3.6 GHz, 16GB of RAM, equipped with an AMD Radeon R9 290X GPU with 3GB of memory and a GeForce GTX 560 Ti GPU with 1GB of memory; (C) quad-core Intel Core i7-4770K CPU at 3.5 GHz, 32GB of RAM and two NVIDIA GeForce GTX 660 Ti GPU with 2GB of memory.

B. Performance comparison

To quantitatively evaluate the performance of the BowMapCL tool, it was compared against several mainstream mappers, including Bowtie [12], BWA [13] and SOAP2 [14] (CPU-based tools), as well as SOAP3 [5], 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 can be seen in Figures 5, 6 and 7. The comparisons were conducted by executing the exact matching operation over the Human Chromosome 1 using a wide range of query set sizes. Naturally all the considered tools were configured to perform ‘exact search’, through the selection of the appropriate parameters. They were also configured to maximally use the available number of CPU threads for their execution (8 threads was the selected value).

As it can be observed in Figure 5, when using one single GPU and compared with the three most popular CPU BWT-based mappers, namely Bowtie, BWA and SOAP2, the proposed implementation presents speedups starting on $5 \times$ (for a reduced number of queries) up to $10 - 15 \times$ (for a considerable amount of queries). The usage of such a considerable amount of queries allows to masquerade and make irrelevant the starting time overhead for creating the OpenCL environment, as well to fill the pipeline of operations of the exact string matching procedure.

Among the GPU-based versions, SOAP3, HPG-BWT and CUSHAW were the selected tools for the comparison evaluation. From these three tools, CUSHAW bears a worse performance when comparing to SOAP3 and HPG-BWT (obtained speedup is approximately $12 \times$ for queries sets over 25M). HPG-BWT tool, as it required the usage of 2 equal GPUs, could not be ran in the same platform as the other tools (platform (C)), and therefore, is executed in platform (A). HPG-BWT proved to be around $5 \times$ slower when compared to BowMapCL (as it can be observed in Figure 6). Among these 3 tools, SOAP3 is the one with the best performance, such that the speedup of BowMapCL over SOAP3 is approximately
1.5× (depicted in Figure 7). Although both the SOAP3 and the CUSHAW were ran in platform (C) (just like every CPU-based tool), their results are not depicted in the same graph as they offer very different computing performances. Other mainstream state-of-the-art applications (e.g., BarraCUDA) were excluded from this comparison evaluation, as it would not be fair, since they do not allow such exact matching option (do not offer the possibility to select the maximum number of mismatches).

C. Scalability

Figure 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 (affecting the total number of consumer threads that assist the accelerators). From the obtained results, it can be observed that, with a significant number of queries, the scalability of BowMapCL is almost perfect when using two GPUs, with a speedup very close to 2×. In what respects the number of allocated buffers, it can also be observed that by using more than one set of buffers 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. 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 degraded).

D. Load balancing

Figure 9(a) presents the BowMapCL performance and the offered load balancing capabilities when using two heterogeneous 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 distributes the workload (chunks of queries) unevenly, in order to minimize the resulting overall processing time. To achieve such load balancing, BowMapCL further partitions the last chunk of queries with a finer granularity, such as to guarantee that the devices terminate at the same time. As it can be seen in Figure 9(a), this is achieved with an insignificant difference, as low as 14ms.

Figure 9(b) depicts a different case study. Although the two used GPUs are equivalent, one of the accelerators (GPU 1) is also being used by another independent application kernel. As a consequence, the task partitioning algorithm perceives the performance of the first GPU as being much lower than the second. To compensate for the device contention, the tool dynamically distributes more chunks of queries to the second device. The last chunk of queries is further divided in fine-grained partitions, to achieve an overall load balancing and as a result, both devices finish within a time interval of 20ms.

E. Index sampling

As it was referred before, a conservative encoding scheme was adopted in the presented BowMapCL tool in order to reduce the memory footprint required to accommodate the index data structure in the GPU accelerators. In fact, although other alternative and more sophisticated schemes have already been proposed in the literature, it was decided to adopt an encoding algorithm as regular as possible, that better suits the constraints imposed by GPU architectures.

Nevertheless, BowMapCL also offers the user the capability to configure the sampling rate that is adopted in the processing of the index data 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 FM-index data structure that occupies 1.25 bits per original input bit, when processing DNA data. On the other hand, by increasing the sampling rate,
it is possible to reduce the resulting FM-index data structure size. As an example, with a sampling rate equal 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 threads need to perform more operations for the computation of each OCC($c, i$) entry (see Figure 10). Since the OCC bitmap always take the same space ($m \times 5 \times 4/32 = 0.625 \times m$), 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. Obviously, other data types will incur in other memory footprints, as instead of 5 distinct characters, values up to 128 can be necessary when dealing with general text.

F. Suffix array index sampling

When sampling the suffix array, this data structure that by definition is four times larger than the original data (due to the conversion of characters to integers) becomes less memory demanding. By recurring to the last-to-first column mapping property (see equation 1), the SA value for each occurrence returned by the kernel can be computed. As the sampling increases the number of steps till a sampled value is found (as they become more sparse), it results in a program execution time overhead. As shown in Figure 11, such overhead presents a linear behaviour. On the other hand, a SA sampling rate $r$ reduces the SA memory footprint by $r$. Such result is invariant for different data types and it is not DNA data type-dependent.

V. DISCUSSION AND CONCLUSIONS

This paper presented a study and characterization of a heterogeneous multi-device parallelization model for an exact string matching problem. The envisaged and designed solution was subject to a programming development using the OpenCL API and a tool (BowMapCL) was implemented. Such software is defined as a fast OpenCL-based exact string matching tool, based on the Burrows-Wheeler Transform and the FM-Index, targeting highly heterogeneous platforms composed by multiple OpenCL-enabled devices. The tool also offers 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 almost any machine, not presenting any relevant restrictions, which contrasts with the majority of the currently available tools. Furthermore, due to the implemented index partitioning scheme, 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 provides speedups ranging from $1.5 \times 5$, when compared with the best performing GPU-based tools, and from $10 \times 15 \times$ when compared with CPU-based tools, using a single GPU device.

REFERENCES