Burrows-Wheeler Transform based indexed exact search on a multi-GPU OpenCL platform

David Nogueira, Pedro Tomás, Nuno Roma
INESC-ID, Instituto Superior Técnico, Universidade de Lisboa, Portugal
e-mail: david.jacome.nogueira@tecnico.ulisboa.pt, pedro.tomas@inesc-id.pt, Nuno.Roma@inesc-id.pt

***Abstract*—A multi-GPU parallelization of exact string matching algorithms based on the backward-search procedure by using indexing techniques, such as the Burrows-Wheeler Transform and the FM-Index, is proposed in this paper. To attain an efficient execution on highly heterogeneous parallel platforms, the proposed parallelization adopted an unified OpenCL implementation that allows its execution either in CPUs and in multiple and possibly different GPU devices (e.g., NVIDIA and AMD GPUs) that integrate the targeted platform. Furthermore, the proposed implementation incorporates convenient load-balancing techniques, in order to ensure not only a convenient balance of the involved workload to minimize the resulting processing time, but also the possibility to scale the offered throughput with the number of exploited GPUs. The obtained experimental results showed that the proposed multi-GPU parallelization platform is able to offer significant speedups (greater than 10x, when using one single GPU) when compared to conventional mainstream multi-threaded CPU implementations (Bowtie - 8 threads), and between 5x and 30x when compared to other popular BWT-based aligners, namely BWA and SOAP2, using their multi-threading options. When compared with state of the art GPU implementations (e.g., SOAP3, HPG-BWT, Barracuda and CUSHAW2-GPU), the proposed implementation showed to be able to provide speedups between 2.5x and 5x. The execution of the proposed alignment platform when considering multiple and completely distinct GPU devices demonstrated the ability to efficiently scale the resulting throughput, by offering a convenient load-balancing of the involved processing in the several distinct devices.**

***Keywords*—Parallel Computing, OpenCL, Graphics Processing Unit (GPU), Burrow-Wheeler Transform, FM-index, exact string matching**

I. INTRODUCTION

The gradual decrease of the new sequencing technologies costs has led to a consequent increase of the amount of available biology data, raising important challenges to accelerate its processing. One of the most common and challenging problems is concerned with the mapping or alignment of sequence reads to a given reference DNA genome or chromosome, as most of the information is obtained by homology [1].

When no errors are allowed, this sequence alignment operation actually corresponds to an exact string matching problem. Nevertheless, exact string matching goes far beyond bioinformatics and it actually influences many other areas in the computer science domain, such as pattern recognition, document matching and text mining, intrusion detection systems and image and signal processing.

Exact string matching can be regarded as a sub-string matching problem, whose desired output is the list of all occurrences of a short sequence of characters (called *pattern* or *query* of length *n*) in a reference string of length *m*, such that $n \ll m$ [2]. The matching procedure is usually conducted with one of two approaches: sequentially searching for the query string directly on the reference text, usually implemented with dynamic programming schemes, without any additional data structure to support it; or through an indexed approach, which usually takes as input an auxiliary and previously computed data structure, called *index*.

Due to the significant computation time that is involved, the first approach is only adopted for very small strings, when its context is subject to changes or when it is not known *a priori*. On the contrary, the latter approach usually leads to a considerable reduction in the search time, as indexes reduce the search space to the positions where the string can occur. As a consequence, indexed based approaches are widely established whenever the reference text is large and constant, as changes on it would require the re-computation of the index. It should be noted, however, that the time for building the index is not usually an issue in what concerns the searching time, as it is supposed to be computed only once and before the actual search procedure.

Some of the most prominent index data structures are based on the Burrows-Wheeler Transform (BWT) [3] and FM-index [4], mostly due to the offered compression, when compared with other less efficient structures, such as suffix trees [5], suffix arrays [6], [7] and hash-tables [8]. By using these index data structures, several search and alignment algorithms are able to find a given query with a complexity proportional to $\mathcal{O}(n)$ (dependent only on the size of the pattern to be searched), as opposed to a $\mathcal{O}(m)$ execution time, if no index is used [9], [10] or $\mathcal{O}(mn)$ if a naive dynamic programming approach is considered [11].

Nevertheless, considering that current High Throughput Short Read (HTSR) sequencing technologies can produce, in a single run, hundreds of millions of short reads (DNA short sequences), each with 30 to 200 base pairs, the task of efficiently and accurately aligning these reads against a large reference sequence is still a computationally limiting factor, not only in the bioinformatics domain but also in several other research areas. As a consequence, the usage of high-performance computing resources becomes mandatory. Among the current offer, a cost-effective and commonly accessible option is the usage of GPUs, as they allow to significantly reduce the execution time by exploiting the available parallelism in the programs.

However, contrary to what happens with conventional dynamic programming alignment procedures, indexed sequence

978-1-4799-5313-4/14/$31.00

search is characterized by a highly irregular processing flow, which poses difficult challenges in order to be efficiently implemented in a GPU device. 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 [12], [13]. However, an important step forward still needs to be fulfilled, not only to extend such parallelization in order to efficiently exploit all the (possible distinct) devices (e.g. Intel and AMD CPUs, NVIDIA and AMD/ATI GPUs, etc.) that are currently available in the latest generation of heterogeneous platforms, but also to scale with the number of available devices, maximizing their utilization.

To achieve this objective, the presented paper proposes a parallel implementation of a generic indexed search procedure based on the BWT, that efficiently exploits the computational resources available in state of the art heterogeneous platforms, composed by several GPUs. However, contrasting to other parallelizations that have been recently presented of the backward search procedure with the BWT [12], [13], the proposed approach was programmed by using a unified implementation based on the OpenCL API [14], making it the first that is able to be concurrently executed in several and possibly different GPUs coexisting in the considered heterogeneous platforms. The obtained experimental results showed that the proposed implementation is able to offer significant gains when compared with the current GPU-based state of the art tools, like SOAP3 [12] and HPG-BWT [13], providing speedups around 2.5x with a single GPU device and around 5x when using 2 GPUs. When compared with the mainstream CPU-based alignment tools, like Bowtie [15], BWA [16] and SOAP2 [17], the proposed GPU implementation was able to achieve speedup values between 5x and 15x (single GPU) and between 10x and 30x (two GPUs).

II. BWT BASED INDEXED SEARCH

Due to its reversible nature, the BWT has been widely adopted to compress data [3] (as an example, it is used in the well known file compression tool *bzip2*). In the particular domain of biological sequences processing, many of the indexes that have been adopted by several alignment and mapping tools are also based on the BWT, namely Bowtie, BWA and SOAP3.

The application of the BWT in these mapping and alignment tools is usually divided in two phases: i) the computation of the index; and ii) the actual search of a given sub-string or pattern. The following subsections present a brief description of these two phases.

A. Burrows-Wheeler Transform and FM-Index

The computation of the BWT index for a given reference string or text T starts by appending at the end of the text, a lexicographical smaller character that cannot be found in the original text. The *$* character is normally used for this purpose when using biological data. Hence, assuming T with m characters (characters with indexes from 0 to $m-1$), the new appended text T$ will have $m+1$ characters.

The first step of the algorithm is to create a $(m+1)\times(m+1)$ conceptual matrix with all the possible rotations of T$, by accommodating, in the rows, the several obtained cyclic shifts

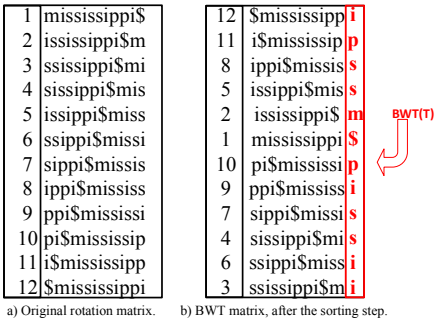


Fig. 1. Computation of the BWT index.

(see Fig. 1.a). Then, after this step, the rows of this matrix are lexicographically sorted (see Fig. 1.b). In practice, the obtained result is exactly the same that would be obtained with the sorted suffixes of a suffix array [6]. In fact, suffix arrays and the BWT matrix are very similar, with the relationship being formulated as $\mathrm{BWT}[i] = \mathrm{T}[\mathrm{SuffixArray}[i]-1]$.

The resulting BWT index is obtained by taking the last column of this matrix. For example, the resulting BWT for the "mississippi" string is "ipssm$pissii". The corresponding suffix array (12 11 8 5 2 1 10 9 7 4 6 3) can be obtained with the indexes of all suffixes occurrences in the original text, corresponding to the sorted permutations of the rows of the aforementioned matrix.

As it can be seen, the transformed text has the same size as the original text. Furthermore, there is no need to explicitly allocate the conceptual matrix, since the reordering of the rows can be efficiently done by simply changing the corresponding memory pointers, which can be also used to implement the comparisons.

Meanwhile, Paulo Ferragina and Giovanni Manzini proposed the FM-index as an alternative compressed full-text index, with significant advantages for search procedures, by noting the similarities between the BWT index and the suffix arrays [4]. Two additional data structures are used by their proposed backward search algorithm: the C vector and the OCC matrix (see Fig. 2). The C vector comprehends all the distinct characters that are presented in the text. For each character, it keeps the count of occurrences of all lexicographically smaller characters in the text. Hence, this vector can be seen as the first column of the BWT matrix, when it is represented in a compressed way. The OCC matrix counts the occurrences of each unique character c, for every prefix of the original text of the form $T[:k]$, with k from 0 to m. Therefore, the OCC(c,i) entry represents the number of occurrences of character c in the prefix $T[:i]$.

B. BWT-based Indexed Search

To search for any arbitrary sub-string in the reference text, the usually adopted backward search procedure is based on the 'last-to-first mapping' property, as described in [4] (see Algorithm 1). As an example, Fig. 2.c presents the application of this procedure to find the occurrences of the "ssi" sub-string 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

OCC	i	p	s	s	m	$	p	i	s	s	i	i
char	1	2	3	4	5	6	7	8	9	10	11	12
$	0	0	0	0	0	1	1	1	1	1	1	1
i	1	1	1	1	1	1	1	2	2	2	3	4
m	0	0	0	0	1	1	1	1	1	1	1	1
p	0	1	1	1	1	1	2	2	2	2	2	2
s	0	0	1	2	2	2	2	2	3	4	4	4

(a) OCC matrix.

char	C
$	0
i	1
m	5
p	6
s	8
total	12

(b) Cvector.

(c) Backward search procedure for query string "ssi".

Fig. 2. OCC matrix and C vector corresponding to "mississippi" reference text to implement the Backward Search procedure.

search algorithm were represented by the F and L arrows, respectively. The procedure evolves backwardly, by selecting the previous character of the sub-string and by appending it to the searched suffix.

The resulting output of this procedure is an interval of text coordinates [FIRST, LAST], corresponding to all the occurrences of each sub-string. However, these coordinates are referred to the BWT matrix comprehending all the rotated versions of the reference text (see Fig. 1.b), instead of the original text. Hence, by combining the suffix array with the BWT+FM index (which can be easily created along with the BWT index) the desired indexes can be retrieved with $\mathcal{O}(1)$ time. In fact, the returned coordinates can be directly used to index the suffix array, and the values in those positions actually correspond to the desired output solution. An exception is raised whenever the LAST value happens to be lower than the FIRST value, meaning that the sub-string is not present in the text. As the 11th and 12th positions of the suffix array have the entries 6 and 3, these are the starting indexes of "ssi" in "mississippi".

However, although this algorithm only performs some simple arithmetic operations, it is characterized by an unpredictable pattern of memory accesses, which poses important challenges to search procedures implementations based on GPUs. This is particularly relevant when considering that the majority of the time that is spent on this procedure, ($\mathcal{O}(n)$ time), is mainly used to access the OCC matrix.

III. PROPOSED PARALLELIZATION

Usually, the BWT index and the corresponding suffix array, OCC matrix and C vector are computed only once for each reference sequence in a preliminary pre-processing phase. Given the index files and an input file comprehending the read sequences to align, the exact string matching program searches for each query in the original reference sequence, by using the generated index data structures. The output of the backward search procedure is then converted to the positions in the original text, by using the suffix array. Finally, the alignment solutions are written to one or more output files.

Algorithm 1 Backward search pseudo code.

```
procedure BACKWARD SEARCH
    for every query j do
        i := size of query j
        c := last character of query j
        pos := position of character c in matrix
        FIRST := C vector[pos]+1
        LAST := C vector[pos+1]
        while (FIRST ≤ LAST) AND (i ≥ 2) do
            c := get previous last character of query j
            pos := position of character c in matrix
            FIRST := Cvector[pos] + OCCmatrix[pos][FIRST-1] + 1
            LAST := Cvector[pos] + OCCmatrix[pos][LAST]
            i := i -1
        end while
        if LAST < FIRST then
            return not found
        else
            return values of indexes between FIRST and LAST
        end if
    end for
end procedure
```

GPU kernel implementation and mapping

Although highly irregular in terms of memory accesses, the exact string matching procedure using an indexed-based backward search (see Algorithm 1) is still parallelizable, since each sub-string can be independently processed. In fact, although the internal loop corresponding to the search of a possible match for each query is inherently sequential (*while* cycle), the outer loop is completely parallelizable (*for* cycle), as each query is independent and the same instructions can be executed over different data. Algorithm 2 presents the pseudo code of the corresponding kernel. As it can be observed, there is still a close similarity between the code of the single-threaded CPU implementation and the OpenCL GPU parallel kernel.

However, there are two major differences worth noting. One is concerned with the mapping of the queries 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). Since no data sharing is required between the working processing elements, there is no need for any complex synchronization scheme. Hence, 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 alignment, neither significantly affect the resulting performance.

There is, however, a specific synchronization point that is required to ensure the maximum efficiency of the kernel and the optimal usage of the local memory. In fact, not all types of memory have the same latency. Naturally, accesses to the local memory have a much lower penalty than accessing the global memory of the device. 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 from the global memory to the local memory in the beginning of the kernel

Algorithm 2 Backward search kernel pseudo code implemented in the GPU.

```
INPUT: numqueries, OCCmatrix, Cvector, queries, queries_sizes, character_Map, Data_Type_range
OUTPUT: queriesFIRST, queriesLAST
procedure BACKWARD SEARCH KERNEL
    global_index := get global id
    local_index := get local id
    group_size := get local size
    copy Cvector and character_Map from global memory to local memory
    wait at the OpenCL barrier //synchronize all threads in same work-group
    if global_index <numqueries then
        i := size of query[global_index]
        c := last character of query[global_index]
        pos := position of character c in matrix
        FIRST := localCvector[pos]+1
        LAST := localCvector[pos +1]
        while (FIRST ≤ LAST) AND (i ≥ 2) do
            c := get previous last character of query[global_index]
            pos := position of character c in matrix
            FIRST=localCvector[pos] + OCCmatrix[pos][FIRST-1] + 1
            LAST =localCvector[pos] + OCCmatrix[pos][LAST]
            i := i-1
        end while
        queriesFIRST[global_index] := FIRST
        queriesLAST[global_index] := LAST
    end if
end procedure
```

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). In contrast, the C vector is a very good candidate, since its size is lower than 1KB 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 (the input data type can be DNA, proteins or text). On each access to the OCC matrix, the thread only 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 exists in the text, it 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 row with a complexity of $\mathcal{O}(1)$. The character_Map data structure, as well as the C vector are both copied from the global memory to the local memory before the exact search begins. 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 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 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. Fig. 3 illustrates this profiling procedure, where the work-group size was varied for a given GPU device and benchmark. For this illustrated setup can be observed that 8 represents the best configuration for the work-group size, since smaller and larger work-groups will result in slower program executions. On the other hand, to achieve the best performance, the number of queries to be dispatched on each kernel launch should be large enough, as there is an inherent cost to every data transfer, and only with large chunks can that cost be mitigated.

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. 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, a dynamic scheduling is highly desired, as the performance of each device is only known during its execution, and it may even suffer from changes along the time. At this respect, the usage of simpler and static load balancing schemes could easily lead the system to a load imbalance, which is not desired. 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.

This dynamic load distribution was implemented by using a Producer-Consumer scheme (see Fig. 4), where a single producer fetches chunks of queries from the input file and puts them in a task queue (circular producer-consumer buffer). Then, as soon as each device finishes the processing of a given block, it accesses this circular buffer to get a new block of queries. This way, since the tasks are being dynamically assigned whenever the devices finish their current job, the faster devices will process more data to compensate for the slower ones. 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, reducing the imbalance penalty by fetching smaller blocks at the end of the program, trying to reduce the gap between each device execution time.

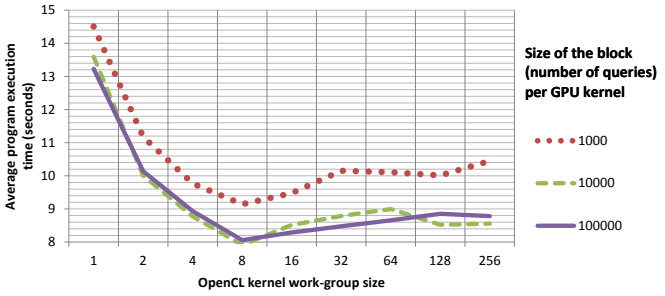


Fig. 3. Optimum work-group size estimation for a given kernel search (search of 10M queries of 75 nucleotides each).

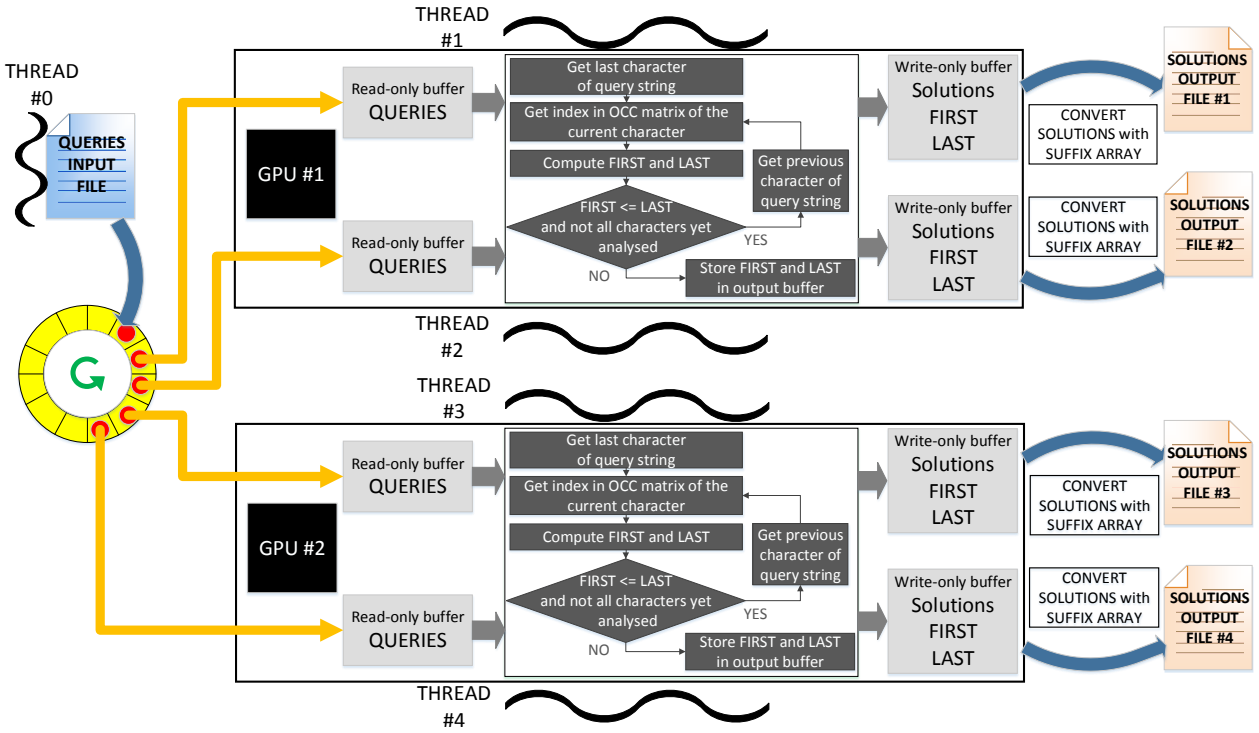


Fig. 4. Architecture of the proposed parallel implementation of the exact string matching procedure using multiple and possible different GPUs.

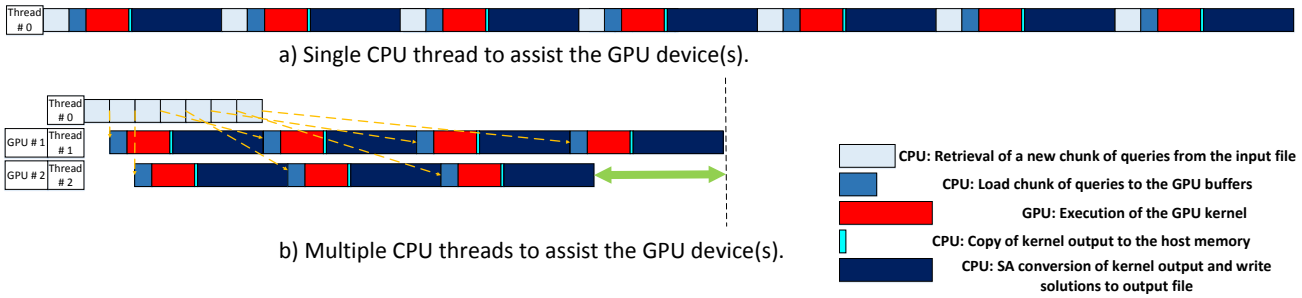


Fig. 5. Temporal diagram of the multiple GPU string matching procedure by considering: a) one single CPU helper thread; b) Multiple CPU helper threads.

Asynchronous multi-threaded execution

In a hypothetical GPU implementation supported and assisted by a single CPU thread (illustrated in Fig. 5.a), 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 GPU 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 GPUs, as the host thread would be blocked in each OpenCL operation (load of chunks of queries to GPU, waiting for the kernel execution and copy of results from GPU to host main memory). In fact, in such scenario it would be useless to use more than one GPU, since the previously used GPU would have already finished and would be already available to be used again by the time the other GPU was enqueued. As a consequence, the usage of multiple CPU threads (at least one per GPU) is required, in order to ensure a constant flow and processing of the sequences data.

To circumvent this problem, the proposed program architecture is comprised of several CPU threads to coordinate and orchestrate each GPU, as well as to perform the I/O operations, as it is illustrated in Fig. 4. One CPU thread (thread #0) is assigned with the task of reading the queries from the input file and of writing them into the host circular buffer. Although there are multiple consumers (one per buffer per GPU device, as explained in the following subsection), only one producer is created, as its operation is much less time expensive than the consumers operation. Each CPU thread that assists a GPU (threads #1 to #4 in Fig. 4) is responsible for: i) fetching a chunk of queries from the task queue buffer; ii) enqueueing the writing of those queries to its GPU 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 to the desired solution space by using the suffix array, and finally, for vi) outputting them to a file. Hence, although each thread waits for each kernel and data transfers to complete, the threads assigned to the GPUs are executing in parallel, allowing the program to scale with the number of GPUs with a

minor overhead (as can be seen in Fig. 5 and Fig. 7). Moreover, not only are the several alignments occurring in parallel in the various GPU devices, but the reading of the queries and the writing of the result to the output file are also simultaneously happening, which allows a combination of the CPU and GPU capabilities in an efficient way. 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 competition for a unique resource, which would arise if only one file had been used. This approach does not introduce any disadvantage, as the queries are independent and the output files can be easily merged, if necessary.

Communication and computation overlap

Besides the impossibility to fully exploit the concurrency between the CPU and the GPUs, the usage of only one thread per device would also not allow to overlap the communication (between the CPU and the corresponding GPU device) with the kernel computation on that same device. 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 the GPU could not start sending the queries of the next iteration to the GPU, as the kernel might still be running and the buffer with the input queries might be being used). To circumvent this problem, a double buffering technique was considered and more threads were created.

To implement such double buffering scheme, the memory allocation on the target device is divided in two sub-banks, to allow the usage of two distinct input buffers for the queries and two output buffers for the results (as depicted in Fig. 4). These two sets of buffers on each device are used concurrently, being each set managed by a single CPU thread. Hence, instead of having a single thread per device, the program assigns a number of threads equal to the number of sets of buffers for each device. As an example, in a platform with 2 GPUs and 2 buffers per GPU (such as the one in Fig. 4), it will assign 4 threads for the GPU management. Hence, while the GPU device is processing one chunk of queries (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 data transfers can be completely hidden, overlapping the communication and computation for each target device.

The OpenCL API allows the execution of the kernels on a wide range of devices, ranging from GPUs to CPUs. In this sense, it would be possible to dynamically distribute some of the workload to the CPU cores (schedule of kernel executions to both the GPUs and the CPU). However, this would lead to an increase in the execution time of CPU threads that assisted the GPU devices. Experimental results not presented in the paper showed that a better performance was achieved using the CPU cores to only orchestrate the execution on the GPUs and to control the buffer assistant threads (as depicted in Fig. 4).

Kernel Output conversion

Due to its large space footprint (4 times the size of the original text) the suffix array is kept on the host memory, where the memory is not as constrained as it is in the GPUs. As a result, it is the host-side threads that convert each kernel output (corresponding to each query) to the actual positions in the original text. Besides the referred memory concern, it is worth noting that this conversion is a procedure that is not well balanced and therefore not suited for a GPU execution, as the range of solutions for each query may significantly vary (different number of occurrences for each query). Such situation would cause some GPU threads to idle whenever their work was finished before the work of the other threads. Since this is not desired on a GPU parallel execution, the conversion of the solution space was implemented on the CPU, during the periods when the host would be idle, waiting for the completion of the GPU kernels.

IV. EXPERIMENTAL EVALUATION

To assess and quantitatively evaluate the performance of the proposed implementation, it was compared against several mainstream mapping tools, like Bowtie [15], BWA [16] and SOAP2 [17] (CPU-based tools), and SOAP3 [12], HPG-BWT [13], Barracuda [18] and CUSHAW2-GPU [19] (CUDA-based GPU tools). The presented results were obtained on a platform equipped with a dual quad-core Intel Xeon E5-2609 CPU (eight cores) at 2.40GHz, 32GB of RAM and two NVIDIA GeForce GTX 680 GPUs with 4GB of memory. The used reference for the alignment experiments was the E.coli genome. The queries were obtained from real data.

A. Profiling

The different steps of the algorithm were initially individually profiled by considering a straightforward implementation with just one CPU thread to assist the GPU (see Fig. 5.a). In this case, overlap between steps does not occur, and the duration of each step can be easily obtained. With large datasets of reads to align, it was then observed that the I/O operations (which consists of reading of queries from the input file to the host main memory and writing of the attained solutions to the output file) were the most costly operations. The usage of a producer-consumer scheme allowed to overlap this steps with the core execution of the program, therefore preventing these operations from being a bottleneck.

Using the described implementation of the paper, and by examining only the OpenCL functions steps (namely, sending the queries to the GPU memory, executing the kernel on the GPU and retrieving the solutions from the GPU memory), it was observed that from these function steps the execution of the kernel is the most time consuming function (around 75%), while the transfer of queries to the GPU takes around 25%, and the retrieval of the solutions from the GPU memory is almost negligible. This shows that is possible to mitigate and even hide the OpenCL communications (sending the queries to the GPU and retrieving the solutions) under the period of time it takes for the GPU to complete the execution of the kernel, by making usage of a double buffering scheme.

B. Performance comparison

When compared with publicly available tools, the proposed implementation proved to offer a significantly better search performance in terms of execution time, as it can be seen in

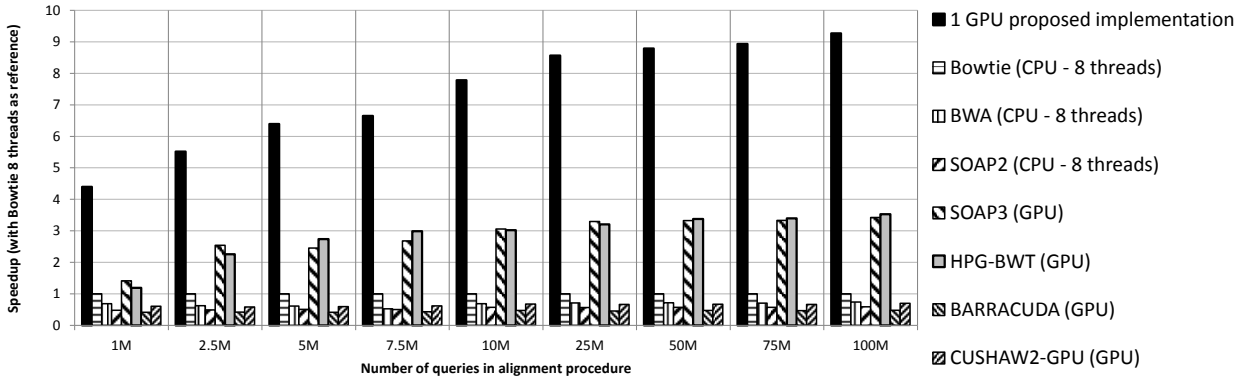


Fig. 6. Comparison of the execution times obtained with the proposed implementation using a single GPU and double buffering, and with all the considered alternative CPU and GPU implementations, for alignments with orders of magnitude between 1 million and 100 million queries. The speedups are presented having the Bowtie (8 CPU threads) performance as reference. All the tools were configured to perform the same operations and give the same output.

Fig. 6. The comparison was made by taking into account the overall execution time of each one of the different programs over a wide range of datasets. For such purpose convenient adaptations were introduced in these tools in order to ensure that the comparison was as fair as possible, and that the cost of outputting the solutions in different file formats did not benefit some tools in detriment of others. Moreover, all the tools were configured to perform 'exact search', whenever this parametrization was possible. The CUSHAW2-GPU and BARRACUDA did not allow such option, and therefore the measured execution times were considerably greater than the ones obtained with the other tools.

When compared with the three most popular CPU-BWT-based aligners, namely Bowtie, BWA and SOAP2, the proposed implementation presents speedups between 5x and 15x when using one single GPU and between 10x and 30x when using two GPUs. Regarding these particular CPU tools, Bowtie proved to be the fastest one, although it is 9x slower than the proposed implementation with a single GPU, even when using 8 threads, being therefore selected as the reference CPU implementation for all the performances that are compared in this chart. From the GPU-based versions, HPG-BWT and SOAP3 turn out to be similar in terms of execution time. Both tools proved to be around 2.5 times slower in comparison with the proposed tool when using a single GPU, and 5 times slower when using a dual-GPU system.

C. Scalability and load balancing

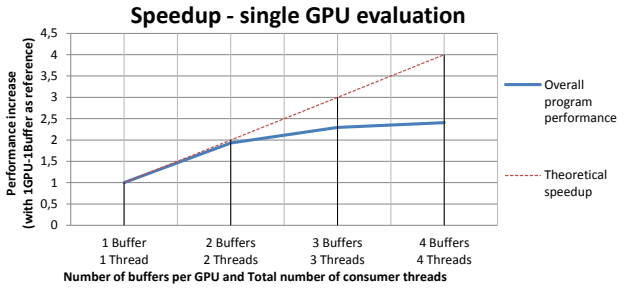
Fig. 7 presents the variation of the obtained performance with the number of buffers assigned to each GPU (which affects the total number of consumer threads that assist the GPUs). As it can be observed, increasing the number of buffers provides the necessary conditions to allow the execution of multiple operations in the same device. The scalability of the program is almost perfect when using two GPUs, comparing to the usage of just one device, with the speedup doubling its value. Both graphs show that using two buffers (double buffering) allows to double the speedup, which proves that the communication was perfectly hidden and even allowed to run both kernels at the same time, as the device was not yet at full utilization. Naturally, this gain is limited by an upper bound that is intrinsically related to the device and to the interconnection between the CPU and the GPU. As soon as the full utilization is reached, increasing the number of buffers per device it is not justified because the increase in performance is not significant.

Fig. 8 illustrates the performance that is offered by the tool when using two GPUs. Fig. 8.a presents the load balancing in a situation where the heterogeneous platform is composed by two GPUs from different manufacturers. Therefore, as their computation capabilities are not exactly the same, the dynamically assigned load during run-time is not equally distributed between the devices. Fig. 8.b depicts a particular situation where one of the devices (GPU 1) is also being used by another running instance of this tool. In the presence of this GPU contention, the task partitioning does not produce the same balancing as it would do when no competition for resources (shared device) exist. As a result, the second GPU was dynamically assigned with more chunks of queries to process than the other, as it processes each chunk faster than the first GPU. In the end, the observed difference in the execution time is lower than the time it takes for the slower device (GPU 1 on both situations) to process a single chunk of queries, which in the cases of Fig. 8 is between 200 and 400 ms (as represented by the green arrow in Fig. 5.b). This time was even reduced, since the last chunk of queries that was processed was intentionally partitioned in smaller chunks, so that the devices can fetch and process them with a finer granularity, completing their tasks with similar finishing times. This enhancement can be observed by the fact that the threads of each GPU (either in Fig. 8.a and Fig. 8.b) finish only with a difference of around 20 ms.

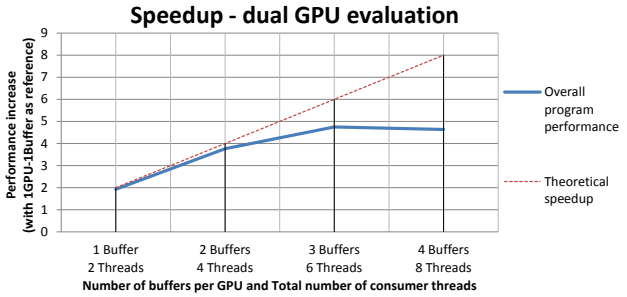
V. CONCLUSIONS

This paper described the design and implementation of a tool to perform exact string matching based on the Burrows-Wheeler Transform and the FM-Index, by exploiting heterogeneous platforms composed by multiple GPUs using the OpenCL API.

When compared with other tools based on BWT indexed search, the proposed tool provides speedups ranging from 2.5x-5x (when comparing with other GPU-based tools) to 5x-30x (when comparing with CPU-based tools using multi-threading). Such performance was achieved by using an highly optimized kernel and a producer-consumer scheme with multiple threads dedicated to each GPU device and to the involved



(a) Single GPU performance evaluation.



(b) Dual GPU performance evaluation.

Fig. 7. Performance evaluation with a different number of threads per GPU (being each one responsible for a set of buffers), on a single and dual GPU implementation, for alignments of 100M queries of 75 nucleotides.

I/O operations, together with the implementation of multiple buffering technique. This approach, not only allowed to overlap the I/O operations from disk to main memory with the sequence alignment procedure, but also to overlap the OpenCL data transfers between the host device and the target devices with the kernel execution on those same devices.

ACKNOWLEDGMENT

This work was partially supported by national funds through Fundação para a Ciência e a Tecnologia (FCT), under projects: HELIX (PTDC/EEA-ELC/113999/2009), TAGS (PTDC/EIA-EIA/112283/2009), Threads (PTDC/EEA-ELC/117329/2010), P2HCS (PTDC/EEI-ELC/3152/2012) and project PEst-OE/EEI/LA0021/2013.

REFERENCES

[1] L. Wang and T. Jiang, "On the complexity of multiple sequence alignment," *Journal of computational biology*, vol. 1, no. 4, pp. 337–348, 1994.

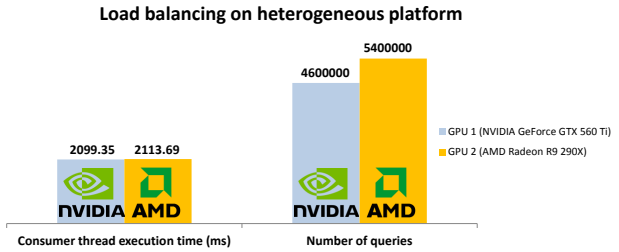
[2] M. J. Fischer and M. S. Paterson, "String-matching and other products." DTIC Document, Tech. Rep., 1974.

[3] M. Burrows and D. J. Wheeler, "A Block-sorting Lossless Data Compression Algorithm," Digital Equipment Corporation, Tech. Rep. 124, May 1994.

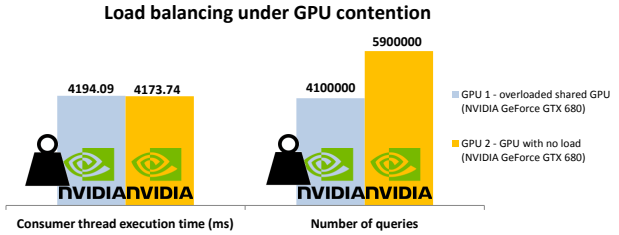
[4] P. Ferragina and G. Manzini, "Opportunistic data structures with applications," in *Proceedings of the 41st Annual Symposium on Foundations of Computer Science*, 2000, pp. 390 –398.

[5] P. Weiner, "Linear pattern matching algorithms," in *Proceedings of the 14th Annual Symposium on Switching and Automata Theory (swat 1973)*, ser. SWAT '73. Washington, DC, USA: IEEE Computer Society, 1973, pp. 1–11.

[6] U. Manber and G. Myers, "Suffix arrays: a new method for on-line string searches," in *Proceedings of the first annual ACM-SIAM symposium on Discrete algorithms*, ser. SODA '90. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 1990, pp. 319–327.



(a) Load balancing evaluation on a heterogeneous system with two different GPUs.



(b) Load balancing evaluation on a system with two equal GPUs, but where one of them (GPU 1) is under considerable contention.

Fig. 8. Load balancing evaluation in systems composed by two GPU cards, for an alignment of 10M queries of 75 nucleotides.

[7] S. J. Puglisi, W. F. Smyth, and A. H. Turpin, "A taxonomy of suffix array construction algorithms," *ACM Comput. Surv.*, vol. 39, no. 2, July 2007.

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

[9] L. Colussi, Z. Galil, and R. Giancarlo, "On the exact complexity of string matching," in *Foundations of Computer Science, 1990. Proceedings., 31st Annual Symposium on*. IEEE, 1990, pp. 135–144.

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

[11] R. N. Horspool, "Practical fast searching in strings," *Software: Practice and Experience*, vol. 10, no. 6, pp. 501–506, 1980.

[12] C.-M. Liu, T. Wong, E. Wu, R. Luo, S.-M. Yiu, Y. Li, B. Wang, C. Yu, X. Chu, K. Zhao *et al.*, "Soap3: ultra-fast gpu-based parallel alignment tool for short reads," *Bioinformatics*, vol. 28, no. 6, pp. 878–879, 2012.

[13] J. Salavert Torres, I. Blanquer Espert, A. Tomas Dominguez, V. Hernendez, I. Medina, J. Terraga, and J. Dopazo, "Using gpus for the exact alignment of short-read genetic sequences by means of the burrows-wheeler transform," *IEEE/ACM Trans. Comput. Biol. Bioinformatics*, vol. 9, no. 4, pp. 1245–1256, July 2012.

[14] Khronos, "Khronos opencl home page," http://www.khronos.org/opencl/ (Dec. 2013).

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

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

[17] R. Li, C. Yu, Y. Li, T.-W. Lam, S.-M. Yiu, K. Kristiansen, and J. Wang, "Soap2: an improved ultrafast tool for short read alignment," *Bioinformatics*, vol. 25, no. 15, pp. 1966–1967, 2009.

[18] P. Klus, S. Lam, D. Lyberg, M. S. Cheung, G. Pullan, I. McFarlane, G. S. Yeo, and B. Y. Lam, "Barracuda-a fast short read sequence aligner using graphics processing units," *BMC research notes*, vol. 5, no. 1, p. 27, 2012.

[19] Y. Liu and B. Schmidt, "Cushaw2-gpu: empowering faster gapped short-read alignment using gpu computing," 2013.