Accelerating approximate string matching in heterogeneous computing platforms

João Rodrigues, Instituto Superior Técnico, Universidade de Lisboa

***Abstract*—The emergence of cheaper DNA sequencing tools lead to an exponential growth of sequenced DNA. Thus, the amount of computation involved in the area of bioinformatics, and in particular in the alignment of DNA sequences, has also grown. To face this challenge, the alignment can be parallelised using alternative computing platforms, such as GPUs, reducing the time required for the alignment. To further reduce computational costs, the alignment tools also adopt a heuristic model, where the queries are filtered using exact search, generating regions which are optimally aligned. The present work proposes a new approximate matching tool, BowMapCL, using heterogeneous multi-device computing platforms. The proposed tool was performed by extending the work of the exact search tool BowMapCL v1.0 [1]. This was performed by adding an filtration stage, using aforementioned exact search, and implementing the Smith-Waterman (SW) algorithm on GPUs. BowMapCL v1.0 (and thus BowMapCL) have been implemented using the OpenCL API, which means the execution is not restricted to NVIDIA GPUs, unlike existing alignment tools.**

When compared to state of the art tools, BowMapCL offers speedups of up to 4 times against the CPU-based tools. Compared to the existing GPU-based SOAP3-dp, it is faster for queries smaller than 100 bases. BowMapCL aligns fewer alignments correctly when compared to the evaluated state of the art tools.

***Index Terms*—GPU, OpenCL, Approximate string matching, SW, Smith-Waterman**

I. Introduction

THE emergence of cheaper DNA sequencing methods led to an exponential growth of sequenced DNA, particularly human DNA. The reduction in the cost of sequencing was achieved through the division of DNA into short fragments, smaller than previous technologies. These fragments are then sequenced in parallel, creating reads. A common usage for these reads is the re-creation of the original genome. This can be achieved either through *de novo* assembly, where no prior information is used, or by using a reference genome to find the final positions of each read, thereby speeding up the assembly process. In the latter method, each read is mapped to the reference genome, allowing some mismatches between the reference and the read, to find the position where the read and the reference have the best match. The mapping of the reads, also known as (short) read alignment, is an example of approximate string matching, a technique with multiple applications, including text retrieval and data mining [2].

The alignment of reads is a computationally expensive operation due to large number (tens of millions) of reads that can be generated in a single run of the DNA sequencing machine and the size of the reference. To reduce time required to perform the alignment, several different tools taking advantage of alternative computing platforms, such as Graphics Processing Units (GPUs), FPGAs or special purpose processors have been proposed. GPUs, in particular, are used by several state of the art DNA alignment tools [3], [4], owing to their increased performance and favorable memory capabilities. Furthermore, the development cost of GPUs tools are lower than other alternative computing platforms. There are two main frameworks for the development of GPU applications, CUDA and OpenCL. The majority of existing DNA alignment tools target the CUDA framework and are thereby restricted to run only on NVIDIA GPUs, whereas OpenCL can run on multiple devices, such as GPUs, CPUs, FPGAs and other accelerating devices, such as Xeon Phi.

Despite the increased performance offered by alternative platforms, the alignment of DNA data against the human genome still requires a great amount of time, which can be lowered by using heuristic techniques. These techniques use exact string search to reduce the area of the genome searched, by generating promising regions. These regions are then searched using optimal approximate string matching algorithms.

II. Algorithmic background

Approximate string matching is an operation where the objective is to find a pattern in a text, where one (or both) of them have suffered some corruption, and can not therefore be matched exactly. These corruptions can be deletions, insertions or substitutions between pattern and text. Approximate string matching is used in different applications, such as text retrieval, data mining[2] and bioinformatics. There are some approximate string matching algorithms capable of operating with a complexity of $\mathcal{O}(n)$, where n is the length of the pattern to be searched. However, bioinformatics applications require more flexibility, since the cost of replacing a base with another is dependent on the specific bases. Moreover, it is also necessary to consider gaps (serial deletions or insertions) to have lower cost than the equivalent serial operations. Hence, the considered optimal string matching algorithms use dynamic programming (DP), and possess a complexity of $\mathcal{O}(mn)$, where m is the size of the text.

A. Smith-Waterman algorithm

The considered optimal approximate string search uses the Smith-Waterman algorithm (SW), which is divided into two stages. In the first stage, a matrix H, as well as the auxiliary matrices E and F (which store the values required for the gaps) are filled according to (1) using the reference $S_q = q_1, \ldots q_n$ and the pattern $S_d = d_1, \ldots d_m$. It is possible

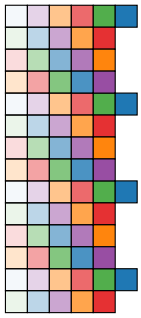

Fig. 1. Striped approach to intratask parallelisation

to adjust the parameters gap open α, gap extend β and substitution scores $\delta(q_i, d_j)$ to select optimal alignments with different characteristics.

$$
\begin{aligned}
H_{i,j} &= \max \begin{cases} H_{i-1,j-1} + \delta(q_i, d_j), \\ E_{i,j}, \\ F_{i,j}, \\ 0 \end{cases} \\
E_{i,j} &= \max \begin{cases} H_{i,j-1} - \alpha, \\ E_{i,j-1} - \beta \end{cases} \\
F_{i,j} &= \max \begin{cases} H_{i-1,j} - \alpha, \\ F_{i-1,j} - \beta \end{cases} \\
H_{i,0} &= E_{i,0} = F_{i,0} = 0 \\
H_{0,j} &= E_{0,j} = E_{0,j} = 0
\end{aligned}
\tag{1}
$$

In the second stage, a trace back occurs over the H matrix to extract the optimal alignment. The trace back begins at the cell from the matrix H with the highest value. The cell that originated the value from the current cell is then selected, iteratively, until a cell with a score of 0 is reached, and the trace back is concluded.

B. Extraction of parallelism in SW

In sequence alignment, several different reads are aligned against one reference. Furthermore, since each read can generate several promising sections of the reference to be searched against, it is possible to perform several alignments simultaneously in the GPU. This approach is known as intertask parallelism and presents a case of an embarrassingly parallel problem, since there are no dependencies between alignments. However, it is also possible to parallelise a single alignment between a read and the reference (or a reference section), an approach known as intratask parallelism. The chosen approach for intratask parallelism is a striped approached [5], where several values from one column are computed at once, marked in one color in Figure 1, without taking into consideration intracolumn dependencies, since the values of F often do not contribute to the values of H. After a full column is calculated, and if the values from H are high enough that the values of F contribute to the values of H, the values of the matrix H are recomputed. This recalculation is known as the lazy F loop.

C. Non-optimal alignment

Despite the increase of performance of optimal alignment tools, existent optimal alignment tools are not suited to the alignment of DNA generated by DNA alignment tools due to the high computational costs required to align millions of reads against a human genome through optimal alignment algorithms. Heuristic algorithms can lower the computational costs, by potentially missing some optimal alignments. The foundation of these heuristic algorithms is the fact that in every approximate occurrence of a query in a text there are some substrings of the pattern that match the text without any errors. The locations of theses strings, which can be found through exact search, indicate areas of text where the best possible approximate string matches can occur. To find the approximate string match, the area around each location can then be expanded by matching additional characters from the pattern and text as long as the pattern and text do not diverge significantly. It is also possible to use the areas around the locations of seeds as targets to perform optimal approximate string matching. The usage of seeds to select areas suitable for a more careful search is called filtration [6]. Another technique, called intermediate partitioning, divides the queries into seeds, which are searched in the text, although some errors are allowed. Since the number of errors in this search is smaller than the number of allowed for the whole query, the search procedure is less costly than optimal alignment. In fact, this procedure can be seen as searching all the neighbors of the seed, where the neighbors are mutated versions of the seed.

The exact search can be performed using several different algorithms, such as hash-tables, suffix trees or the Burrows-Wheeler Transform with FM-index [7]. Exact search using FM-index offers search times proportional to the length of the query and is specially amenable to GPU devices due to the lower memory costs when compared to other algorithms. Hence, it has been used for the implementation of several read alignment tools using GPUs [8], [9].

D. Brief presentation of BowMapCL v1.0

As previously stated, the present work is based upon the tool BowMapCL v1.0, proposed in [1]. BowMapCL v1.0 is an exact search tool targeting heterogeneous platforms, using BWT and FM-index to perform the alignment. This tool is capable of performing the exact search in DNA, Proteins or Text.

In contrast to existing alignment tools, BowMapCL v1.0 uses the OpenCL API to be able to execute in different GPUs from different vendors. Since the GPUs (and hosts) have varying quantities of memory, the tool is also capable of adjusting several parameters to limit the memory consumption of the data structures. Furthermore, the reference sequence can also be split into multiple blocks to further decrease the memory required, allowing the processing of any reference sequence, regardless of the size of the input data. Since the BWT is a data structure mostly used for offline exact string search, BowMapCL v1.0 has two operations modes. The first operation mode creates the index files for the search, taking into account the size of host and accelerating devices memory

to create data structures (index files) which fit into the available memory. The second mode is the principal operation mode. This mode reads the previously created index files and the file containing the reads/queries, which are then searched exactly in the reference text using the GPU(s), creating a range of positions. After the range of positions is transferred to the host memory, it is converted to absolute positions of the text, and the results are written to a file.

In order to hide the communication costs between CPU and GPU, [1] devised an architecture, see Figure 2, where multiple threads (BWT thread in the fig. 2), each with its own set of buffers, enqueue data and kernel executions to the device, and retrieve the range of positions. Another thread, known as the producer thread (thread #0 in the fig. 2), generates the queries which will be searched by reading the input file.

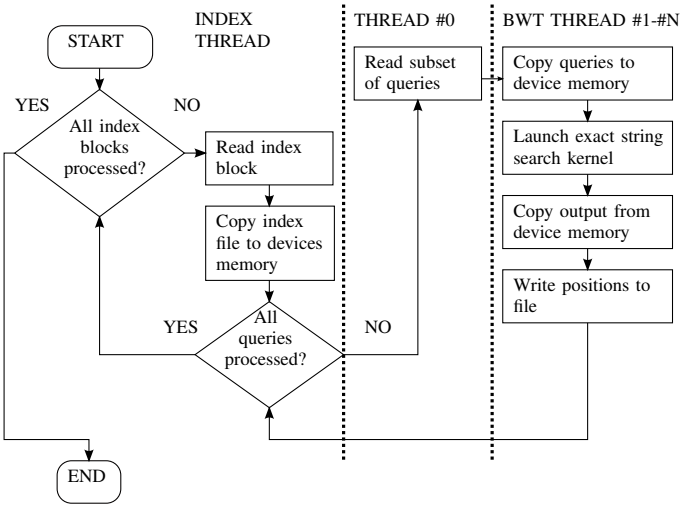

Fig. 2. Flowchart of the parallel solution BowMapCL v1.0 for exact string matching

This implementation was found to have a speedup of 10 over multi-threaded CPU-based exact search implementations, such as Bowtie, BWA and SOAP2, and speedups between 1.5 times and 5 when compared to existing state of the art GPU, such as SOAP3 and HPG-BWT.

III. Implementation

Similarly to its predecessor, the proposed tool has two modes of operation. The first mode creates the necessary data structures for the exact search, whereas the second (and main) mode is the approximate alignment mode. The main mode accepts the data structures previously created, the reference genome, and a query file, and aligns each query against the reference. The queries are divided into seeds to be searched using the exact search. Some positions from the exact search are selected to create potential sections of the reference for the optimal alignment, a procedure called filtration. Finally, the potential sections for each query are searched using SW.

This section details the implementation of the main mode, and is divided into three logical parts. In the first part the parallelism opportunities for the optimal alignment are described. The second part describes the proposed architecture for the program. The third part details how the SW algorithm was mapped to the GPUs.

A. Parallel processing paradigms in approximate string match

Since the number of queries is usually quite large, it is possible to harness the computational power of the GPUs by performing several optimal alignments simultaneously, improving the execution time of the tool. In general, the number of the queries precludes loading the entirety of the query file into the GPU memory in a single phase due to memory restrictions. Nevertheless, by loading chunks of queries to the GPU, it is possible to use a data-level parallelism approach, in the exact search step and in the optimal alignment step. Furthermore, since the exact search and the optimal alignment can be executed independently it is possible to take advantage of the spatial parallelism of GPUs and execute them in parallel, in a task level parallelism approach. It is also possible extract even more task level parallelism since there multiple tasks that can be performed in parallel to the computation of the optimal alignment, such as the I/O operations, namely reading the queries file and writing the results to a file, and computational tasks, such as extracting seeds from the queries and the filtration of the exact search results.

B. Proposed parallelisation architecture overview

As previously mentioned, the proposed tool is composed of a filtering phase, generating promising reference areas, and an optimal approximate string search. Each phase has an operation, in the case of filtering is the exact search, and in optimal alignment is the SW algorithm, which is offloaded to the GPU. In order to parallelise the two main phases of the alignment tool, a two stage pipeline is proposed, with each phase performed entirely by a single pipeline stage. The flowchart of the proposed solution is presented in Figure 3.

The program starts by initialising the OpenCL devices and loading into their memory the blocks of the index. In the initialisation step the several processing threads are also created.

The thread #0 reads the queries from the file to a buffer, grouping them into chunks of queries. When a buffer containing queries is filled, it is sent to circular queue to be consumed by the filtering stage. In the filtering stage, performed by the BWT threads, the seeds are extracted from the queries and sent to the GPU. The exact string matching kernel is launched, matching hundreds of seeds in parallel. When the kernel is complete, the result, containing the range of positions where each seed matches in the index block, is copied to the host memory. The results from each seed are used to select the most promising reference areas for each query, constituting the previously mentioned filtration step. The promising areas are sent to another circular queue. The SW threads in Figure 3 perform the next stage, the optimal alignment. In this phase, hundreds of promising areas are optimally aligned against the respective reference sections through the SW algorithm in parallel. After the kernel has finished its execution, the homology scores are loaded back to the host memory, to be written to the disk, along with the positions of the queries in the reference.

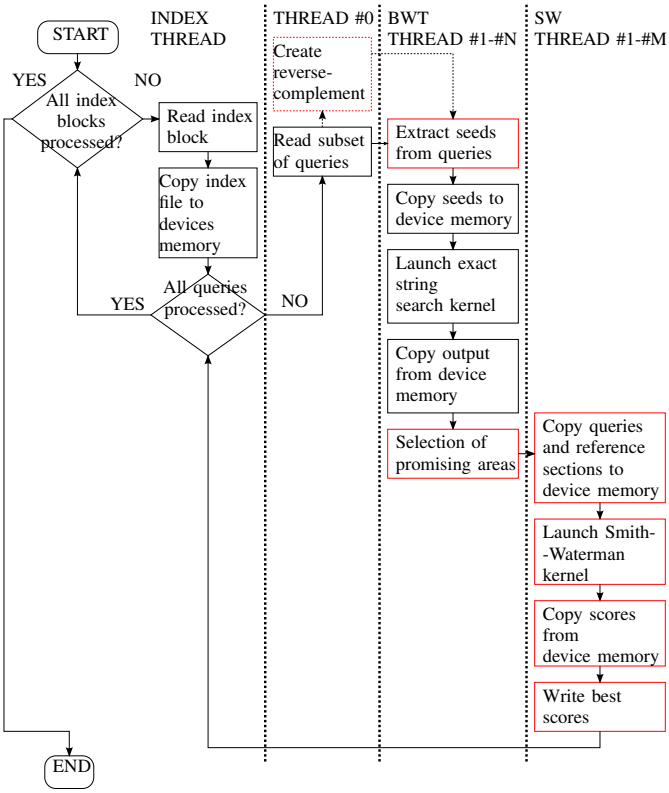

Fig. 3. Flowchart of parallel solution of non-optimal approximate string matching, new steps presented in red

C. Device management

To manage the accelerating devices, the proposed tool extends the architecture of its predecessor, BowMapCL v1.0, through the addition of a new stage, performing the optimal alignment, and the enlargement of the existing stage to perform the filtering step. In [1], two alternatives were evaluated for the management of multiple devices using the OpenCL devices: perform the management using exclusively OpenCL functions, managing several different devices from a single thread, or manage each device from one thread. The first alternative, and considering an execution with non-blocking (asynchronous) operations, the thread would manage the different devices. While the computation occurs in the devices, the thread would also execute the host side computations, such as filtration and seed extraction for the filtering stage and reference selection for the optimal alignment stage. However, if a device does not support non-blocking operations, the thread would be blocked waiting for the execution of the device side operations, including data transfers and kernel execution, preventing the host side from managing other devices or to concurrently prepare the data, leaving the efficiency of the proposed program vulnerable to the capabilities of the devices. The second alternative circumvents these problems by using several threads, each with its own command queue and managing a different device, allowing a concurrent execution of commands through the different devices regardless of the non-blocking capability. Moreover, for data which has a significant cost in processing in the host side, as it is the case, this architecture has the added benefit of extracting parallelism from the host side as well. Consequently, several threads process chunks of queries, where each thread has its own command queue. An initial thread, known as index thread, is responsible for setting up the required environment for the processing, such as the creation of the shared data structures in the host side and in the device side and the creation of the processing threads. Another thread, thread #0 is responsible for reading the input file, creating chunks of queries, and in the case of DNA, for the creation of the reverse-complement. The remainder threads, that effectively manage the devices through the command queues, are divided into two sets. One set (BWT threads in fig. 3) is responsible for the filtering stage, consuming the queries created by thread #0 and generating the promising areas. Each thread of the filtering set is responsible for fetching a chunk of queries from a pool of queries, extracting the seeds from the queries, enqueueing transfer of the seeds to the global memory of the correspondent device, ordering the execution of the exact search kernel, waiting for completion of the execution of the kernel, copying the kernel output to the host memory, selecting the positions of promising areas and, finally, storing those positions into a pool of promising query areas. Even though, at each kernel execution and data transfer from and to the device, a thread is blocked waiting for the completion of the command, the remainder threads can continue to operate in parallel, allowing a maximization of the usage of the CPU and the accelerating devices. The other set of threads (SW threads in fig. 3) is responsible for the optimal alignment, consuming the previously generated promising areas and generating the optimal positions of each query in the text, by fetching the promising areas from a chunk of queries from the pool of promising query areas, extracting the areas from the reference, enqueueing the transfer of the queries and the reference sections to the global memory of the correspondent device, ordering the execution of the SW kernel, copying the kernel output with the SW scores back to the host memory after the kernel execution, and, finally, writing the best scores to the disk.

As hinted previously, with the proposed architecture it is possible to have multiple command queues per device. Hence, in addition to computation and transfer overlap over multiple devices, it is also possible to overlap host side computation and kernel execution/data transfers in a single device, reducing the cost of communication and maximizing the occupancy of the accelerating devices.

D. Memory management

The size of the genomes present a challenge to the parallelisation on GPUs. BowMapCL v1.0 implemented memory efficient data structures in order to minimize the memory occupied by the BWT index blocks. Nevertheless, for human-sized genomes the full index may not fit entirely into the memory of the majority of GPUs. To solve this problem and make the tool capable of handling texts of any size, BowMapCL v1.0 also implemented a technique where the text is partitioned into index blocks that fit into the available device memory.

The complete alignment is conducted by processing sequentially the index blocks and matching all the queries against

each block, since this approach reduces the communication overhead for the data structures. To be able to report only the best result of the optimal alignment, it is necessary to have all the optimal results for all the index blocks simultaneously. This could be achieved by saving all promising areas and conducting the optimal search only after the filtration of all index blocks is finished, or by saving all the optimal scores from the optimal alignment and conducting a post-processing step selecting the best scores. Both methods impose an heavy penalty in the host memory required to store the intermediate results, hence the best alignment inside each index block is stored directly to the output file, possibly resulting in a duplication of results for the best location.

E. Management of multiple devices

The proposed tool has two tasks, the exact search kernel and the optimal alignment, which are executed in parallel. Hence, in the presence of multiple devices, there are two approaches that can be used: differentiate the devices and execute only one of the tasks in each device, or execute both tasks simultaneously in each device, similar to the single device management. The first approach reduces possible contention between tasks and reduces the memory usage in each device. Since the characteristics of the work load are unknown before hand, and can even vary in the course of the execution, it is impossible to distribute the tasks between the devices in such a way as to eliminate bottlenecks, preventing the full utilisation of the device(s). Consequently, it was chosen to execute both tasks simultaneously in all devices.

F. Filtration

As previously stated, BowMapCL v1.0 only performs exact search and does not search for mismatches of any kind. The only existent read alignment tool sharing this restriction is bowtie2 [10]. Thus, BowMapCL uses an algorithm inspired by bowtie2 since the choice of the algorithm heavily impacts the quality of the results as well as the execution time, and the analysis of the trade-offs involved (such as execution time vs. alignment quality) are outside of the scope of the present work.

The filtration happens in two steps, the creation of the seeds for the exact search and the selection of the promising reference sections. The seeds are created by extracting overlapping seeds at regular intervals. By default, the seed length is independent of the query length. In [10] it was reported that it is advantageous to set the interval length to a sub-linear function of the read length. The default function used in bowtie2, also used in the current tool, is to set the interval length between consecutive seeds $I(x)$ to $I(x) = \lfloor 1 + 1.15 \times \sqrt{x} \rfloor$, where x is the length of the query.

After the seeds are searched using exact search in the GPU, the results are copied back to the host memory. The results from each seed are a range of transformed positions, each corresponding to a position in the original text. The number of results have a great variation, with some queries not present in the text, some are present only a few times, and some seeds generate ranges with thousands of results. These latter seeds are not very selective, thus are not very interesting to analyse. Moreover, the conversion procedure from transformed positions to text positions is time consuming. Thus, in the selection of the promising areas, the proposed tool orders the seeds of a single query according to increasing range and then selects the positions from the seed with the smallest range, until a configurable quantity of search regions are reach. If this quantity is not reached, the positions from the next seed are selected, until the quantity is reached or until the program runs out of seeds for the current query. The selected positions are converted to effective text positions, which in turn generate search regions around these positions. If two search regions from a read overlap they are joined and more positions are converted to search regions. Both steps of filtering are performed on the CPU since they do not map efficiently to GPUs due to the memory access patterns and irregularity of the code. Moreover, the conversion of the transformed positions to regular positions requires a very large data structure that can not be easily stored in the GPU.

G. Mapping of SW to GPU

As previously stated, the execution of Smith-Waterman algorithm for each promising area is a computationally expensive procedure. However, it is also suited for parallelisation using GPUs. There are two main approaches: intratask parallelism, where parallelism is extracted from a single optimal alignment, and intertask parallelism, where parallelism is extracted by performing several alignments in parallel. In order to select the best approach, the two approaches were implemented and evaluated.

1) Intratask parallelism: In intratask parallelism parallelism is extracted from the computation of a single alignment using the SW algorithm. In OpenCL the computation is divided into two levels of granularity. On a lower level, all work items execute the same operation, but on different data, extracting data level parallelism. However, work items are grouped into work groups, inside which they can share data and flow control, although work items belonging to different work groups can not share data.

Since intratask parallelism requires communication between the vector elements/work items, it can only be applied at the work group level. Consequently, the intratask parallelism mode combines intratask and intertask parallelism by distributing a single alignment to a work group, but performing several different alignments across different work groups.

Inside each work group, the intratask parallelism was implemented using the previously described striped layout approach, since it has been successfully ported to a GPU architecture and offers the best performance. The pseudo code is presented in Figure 4. The algorithm fills the matrix column by column. In each column, the cells are computed without taking into account inter-column dependencies. If the inter-column dependencies affect the final result, the results are corrected in the lazy F loop.

Even though work items inside a same work group share some resources, such as local memory, and control flow, the targeted version of OpenCL, 1.1, does not have any functions

Fig. 4. OpenCL Intratask SW algorithm using striped layout (pseudo-code)

```
1: function SW(char ref[][], profile_query[][][][], result[],
   interval_small, vHE[][][], database[])
2:   group := get_group_id()
3:   local_size := get_local_size()
4:   tid := get_local_id()
5:   length_big := database[group].reference_length
6:   length_small := database[group].query_length
7:   seg_len = ⌈length_small/local_size⌉
8:   max_score := 0
9:   for i := 1, 2, ..., length_big do
10:    regF := 0
11:    query_char[][] = profile_query[group][big[group][i]]
12:    regH := 0
13:    if tid > 0 then
14:      regH := vHE[group][seg_len][tid − 1].H
15:    end if
16:    Sync_threads
17:    for j := 1, 2, ..., seg_len do
18:      regH_next := vHE[group][j][tid].H
19:      regE := vHE[group][j][tid].E
20:      regH := max(regH + query_char[j][tid], 0)
21:      max_score := max(max_score, regH)
22:      regH := max(regH, regE, regF)
23:      regE := max(regE - GE, regH - GO)
24:      vHE[group][j][tid] := (regH, regE)
25:      regF := max(regF - GE, regH - GO)
26:      regH := regH_next
27:    end for
28:    for i := 1, 2, ..., local_size do          ▷ Lazy F loop
29:      shift_aux[tid] := regF
30:      regF := -GO
31:      Sync_threads
32:      if tid > 0 then
33:        regF := shift_aux[tid - 1]
34:      end if
35:      Sync_threads
36:      for j := 1, 2, ..., seg_len do
37:        vHE[group][j][tid].H := max(vHE[group][j][tid].H,
   regF)
38:        if AnyElement(regF > regH − GO, tid) = false then
39:          Break all
40:        end if
41:        regF := regF - GE
42:      end for
43:    end for
44:  end for
45:  res[gid] := max_score
46: end function
```

to make the control flow dependent on values from all the work items, required for the AnyElement function (used in line 38 of Figure 4), nor does it have any functions which transfer values from a work item to another, required for the vector shift left. For the latter problem, the transference of values from a work item to another is performed by storing the values of each work item in particular location of the local memory and then fetch them from another local memory location, see line 29 until line 35 of Figure 4.

In relation to the AnyElement function, presented in Figure 5, which determines if a given condition is true for any of the elements of a work item, it is also necessary to use local memory to store the values of the condition of each work item inside a work group. Then, each work item traverses the array to find if the condition is true for any of the work items. Since every work item will see the same values, the control flow is performed equally in all work items.

Fig. 5. OpenCL AnyElement function (pseudo-code)

```
function ANYELEMENT(condition, tid, local_size)
    local cmp[local_size]
    cmp[tid] := cond
    Sync_threads
    decision := false
    for k := 1, 2, ..., local_size do
        if cmp[k] == true then
            decision := true
        end if
    end for
    return decision
end function
```

2) Intertask parallelism: In intertask parallelisation each work item performs a complete alignment between a read and reference section. This arrangement, unlike intratask parallelisation, does not have dependencies between each work item, even inside the same work group. However, since there are more matching procedures occurring simultaneously, the memory requirements are higher.

In intertask parallelisation, the matrix can be built row by row (or equivalently, column by column) or anti-diagonal by anti-diagonal. The advantages of the former include the regularity of the memory accesses pattern, since every row has the same amount of columns, unlike the anti-diagonal, and the number of columns is smaller than the maximum size of the cells in the anti-diagonal, resulting in less required memory. The choice of building row by row or column by column is important for the memory usage when the sizes of the read (n) and the reference (section) (m) are very dissimilar, since the memory usage can be proportional to either n or m, which is not the case here, since we are only interested in a reference section enveloping the read.

Since the substitution matrix has memory accesses which can not be coalesced and has a relatively small size, it is an ideal candidate to be loaded onto the local memory. Therefore, before the execution of the optimal alignment all work items load the substitution matrix to the local memory.

In order to reduce memory accesses to the global memory for the intermediate values of H and E, the intertask kernel implements a tiling approach similar to CUDASW++ v2.0 [11]. The matrix is computed in stripes with the same length as the reference section, and with a width adjustable at compile time in the kernel, RFULL in Figure 6. Thus, the width can be adjusted at run time due to the architecture

of OpenCL, where the kernels are compiled each time the program executes. Inside this stripe, the matrix is filled row-wise until the width of the stripe is reached, and only then is the following row computed. Using this technique, the number of memory accesses per stripe is reduced from $RFULL \times n$ to n. The intermediate values H and F of the stripe are stored in registers.

Fig. 6. OpenCL Intertask SW algorithm

```
function SW(char ref[][][], query[][], result[], vHE[][][],
delta[], database[])
  gid := get_group_id()
  length_big := database[gid].reference_length
  length_small := database[gid].query_length
  Copy delta to local memory
  max_score := 0
  for i := 1, 1 + RFULL, ..., length_big do
    pack_ref[] = reference[gid][i]
    for k := 1, 2, ..., RFULL do
      regH[k] := 0
      regF[k] := 0
    end for
    regH_prev_diag = 0
    for j := 1, 2, ..., length_query do
      regHE := vHE[gid][j]
      query_char = multiple_query[gid][j]
      sub := delta[pack_ref[j]][query_char]
                    ▷ regH and regF from previous iteration
      regF[1] := max(regH[1] - GO, regF[1] - GE)
      regE := max(regHE.H - GO, regHE.E - GE)
      regP := regH[1]
      regH[1] := regH_prev_diag + sub
      regH[1] : = max(regH[1], regE, regF[1], 0)
      max_score := max(max_score, regH[1])
      regH_prev_diag := regP
      for k = 2, 3, ..., RFULL do
        sub := delta[pack_ref[j]][query_char]
        regF[k] := max(regH[k] - GO, regF[k] - GE)
        regE := max(regH[k-1] - GO, regE - GE)
        regP := regH[k]
        regH[k] := regH_prev_diag + sub
        regH[k] := max(regH[k], regE, regF[k], 0)
        max_score := max(max_score, regH[k])
        regH_prev_diag := regP
      end for
      regH_prev_diag = regHE.H
      vHE[gid][j] = (regH[RFULL], regE)
    end for
  end for
  res[gid] := max_score
end function
```

IV. Experimental evaluation

A. Testing framework

In order to experimentally assess the performance of the proposed tool, a set of human DNA read files from was selected, with varying lengths and number of reads. The read files are presented in table I. These reads were aligned against the human reference genome GRCh37.75 [12], with an approximate size of 3 GB, since it represents a widely used reference against which reads are aligned [13].

Accession reference	Length of reads	Number of reads
SRR001115	47	10M
SRR3317506	51	26M
SRR211279.1	100	25M
ERR1344794	302	35M

TABLE I
Considered sequenced reads

The evaluation of the independent optimal alignment step was conducted using protein data, since the existing gaped SW implementations using GPUs, calculating only the score, can only operate on proteins. The database of proteins used is the simulated simdb, with 585 MB, available from [11]. This databases represents an optimal case for parallelism, since every sequence in the database has a length of 3000, giving an indication of the maximum performance attainable by the optimal alignment tools. A set of proteins of various sizes, varying from 144 amino acides until 5478 amino acides, also available from [11], served as the proteins to be aligned against the database.

The platform used to obtain the results is a quad core Intel Core i7-4770K operating at 3.5 GHz with 32 GB of RAM, and two GPUs, namely: a Nvidia GeForce GTX 780 Ti GPU with 3 GB of graphics RAM and a Nvidia GeForce GTX 680 GPU with 2 GB of graphics RAM. The tests were conducted using only Nvidia GeForce GTX 780 Ti, unless noted otherwise.

B. Performance comparison

1) Optimal alignment: To quantitatively evaluate the performance of the stand-alone optimal alignment, the proposed tool was compared against CUDASW++ v2.0 [11], a state of the art optimal alignment tool which implements the Smith-Waterman algorithm in CUDA-enabled GPUs. The two parallelisation approaches were also evaluated, with the results presented in Figure 7. In relation to the two parallelism approaches offered by the proposed tool, intratask parallelism has the smallest performance, reaching a maximum of 6.72 GCUPS. Intertask parallelism, in contrast, reaches a maximum of 97.43 GCUPS, a speedup of over 14 times when compared to intratask parallelism. When compared to CUDASW++ 2.0, the proposed tool operating in intertask parallelism is up to 1.70 times faster. The speed advantage of BowMapCL decreases as the size of the queries increases, but even for the biggest queries BowMapCL has a speedup of 5% in relation to CUDASW++ 2.0.

For the complete alignment tool, the SW algorithm is performed using intertask parallelism due to its superior performance characteristics.

2) Non-optimal alignment: The proposed tool was quantitatively evaluated by comparing it against state of the art alignment tools, such as the CPU-based bowtie2 and the GPU-based SOAP3-dp. The comparison was performed by aligning

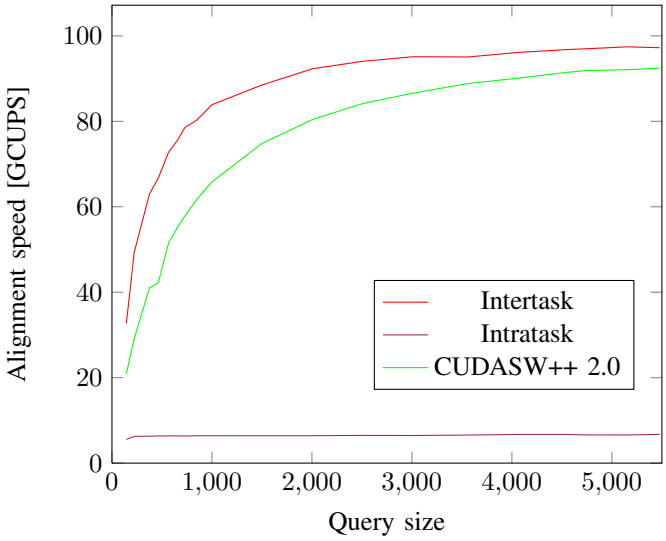

Fig. 7. Comparison of different tools using database simdb, scored with BLOSUM62 substitution matrix

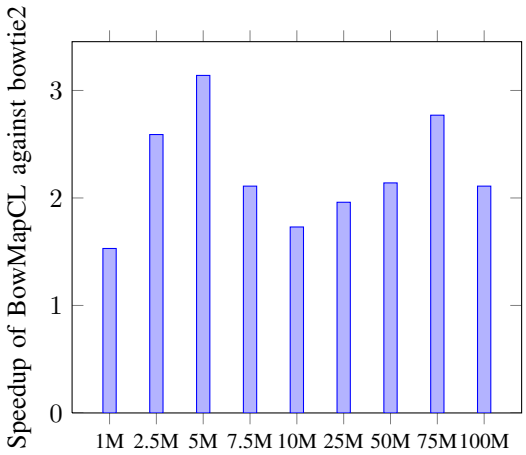

Fig. 8. Speedup of proposed tool in relation to bowtie2

a varying number of queries taken from SRR001115 against the reference genome.

When compared against bowtie2, as can be seen in Figure 8, the proposed implementation can achieve speedups of up to 3 times. The reduced speedup for a small number of queries is caused by the initialisation and GPU-computation overhead, which have a significant impact there are not many queries to dilute this cost.

In relation to the GPU-based SOAP3-dp, the proposed tool is can be up to 4 times faster, as shown in Figure 9, for fewer than 10 million queries. For more than 10 million queries, BowMapCL achieves a speedup of up to 2 times. The disparity between speedups is explained by the size of the data involved, which prevents it from being cached in the memory.

3) Alignment sensitivity: To quantitatively assess the sensitivity of the proposed tool, several real datasets were aligned against the human genome. The Figure 10 shows the percentage of reads from each file successfully aligned against the genome for all the evaluated tools. The execution times of the tools are presented in table II. The proposed tool has,

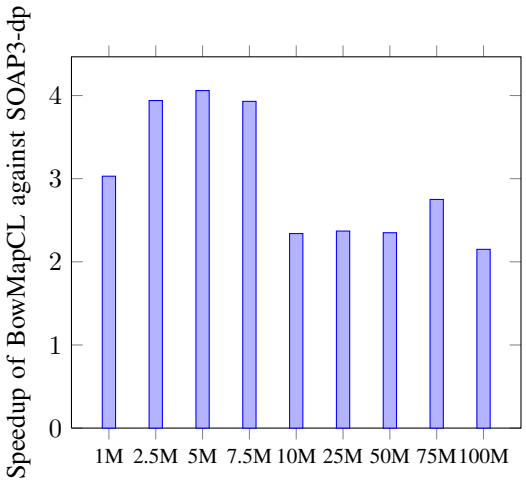

Fig. 9. Speedup of proposed tool in relation to SOAP3-dp

on average, a sensitivity, i.e., percentage of reads successfully aligned with the reference, inferior to the bowtie2 and SOAP3-dp, with an sensitivity of 79.7%, whereas bowtie2's sensitivity is 85.4% and SOAP3-dp is 91.2%.

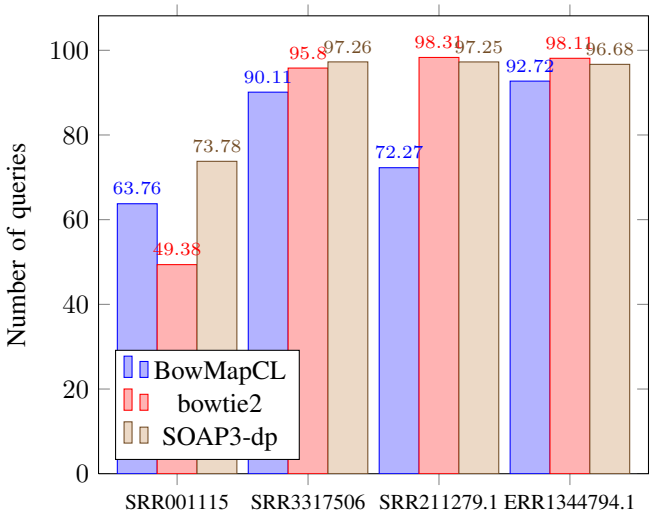

Fig. 10. Alignment sensitivity comparison

In terms of execution times, BowMapCL is faster than bowtie2 for all the evaluated datasets, with a maximum speedup of 4.00 times, with both tools presenting an marked increase in execution times with the increase of the read length. In contrast, SOAP3-dp has a small variation with the size of the read length, and BowMapCL is only competitive in read files composed of reads with a length inferior to 100.

Accession reference	BowMapCL	bowtie2	SOAP3-dp
SRR001115	48.79	195.325	264.299
SRR3317506	338.42	482.031	230.182
SRR211279.1	488.01	1414.65	181.353
ERR1344794.1	793.36	2885.727	249.036

TABLE II
COMPARISON OF EXECUTION TIMES, IN SECONDS

4) Alignment quality: An important characteristic of an alignment tool is the percentage of reads correctly aligned to the reference. To quantitatively evaluate the quality, 3 sets of reads with differing mutation rates were created from the human genome. Figure 11 presents the results for the three evaluated tools, with the lower bar representing the correctly aligned results, the higher bar represents the incorrectly aligned. Thus, the complete bar represents the aforementioned sensitivity. The alignments created by the proposed tool have a lower quality than the alignments produced by bowtie2. For reads with highest (10 %) mutation rate, the filtering approach adopted by BowMapCL can align more reads correctly than SOAP3-dp.

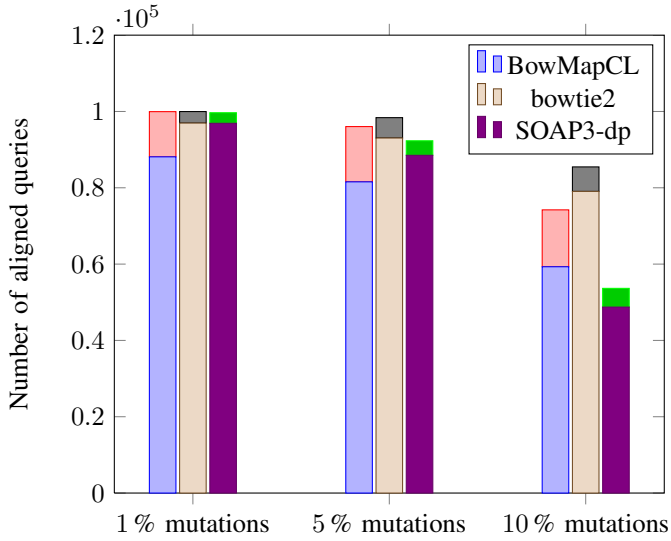

Fig. 11. Alignment quality comparison for 100000 simulated reads of length 100

5) Multiple GPU scaling: The scaling in regards to the number of buffers, with one and two GPUs, was also evaluated. As can be seen in Figure 12, it is possible to increase to the performance by using two buffers in single GPU by overlapping several tasks, such as communication and computation. Despite the increased performance offered by 2 GPUs due to the reduced contention of computation, the performance is limited by I/O costs such as writing the results to the disk.

V. Conclusion

This paper proposes a new non-optimal approximate string matching model using heterogeneous multi-device parallelisation. The model was implemented using OpenCL, by extending the work developed in [1], to create a alignment tool capable on operating multiple types of data, such as DNA or text. The optimal alignment stage of the tool was implemented in GPUs using intratask and intertask parallelism. We found the proposed intertask parallelism to be up to 14 times faster than intratask parallelism. Moreover, the intertask parallelism is also up to 1.7 times faster than CUDASW++ v2.0. When compared to state of the art tools, BowMapCL offers speedups of up to 4.0 times against the CPU-based

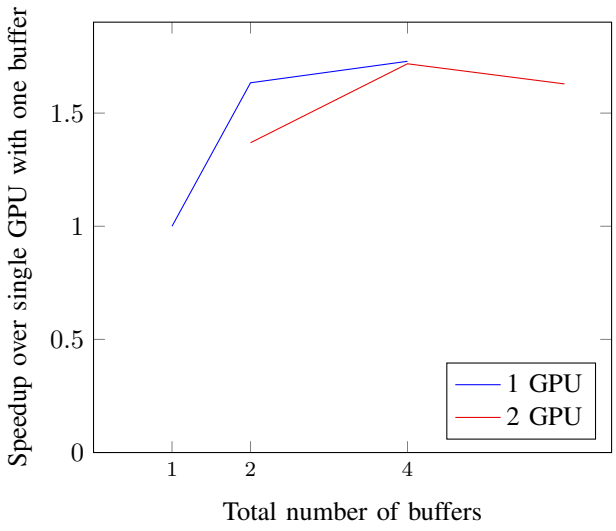

Fig. 12. Scalability in regards with the number of buffers, in the alignment of SRR3317506

bowtie2. Compared to SOAP3-dp, it is up to 2 times faster for smaller queries (less than 100 bases long). In terms of quality of the alignments, BowMapCL has a lower sensitivity and quality than the evaluated state of the art tools.

References

[1] D. A. B. d. C. J. Nogueira, "Accelerating a BWT-based exact search on multi-GPU heterogeneous computing platforms," Master's thesis, Instituto Superior Técnico, Lisboa, Portugal, 2014.

[2] G. Navarro, "A Guided Tour to Approximate String Matching," *ACM Comput. Surv.*, vol. 33, no. 1, pp. 31–88, Mar. 2001.

[3] Y. Liu, D. L. Maskell, and B. Schmidt, "CUDASW++: optimizing Smith-Waterman sequence database searches for CUDA-enabled graphics processing units." *BMC research notes*, vol. 2, p. 73, 2009.

[4] D. Razmyslovich, G. Marcus, M. Gipp, M. Zapatka, and A. Szillus, "Implementation of Smith-Waterman Algorithm in OpenCL for GPUs," in *Parallel and Distributed Methods in Verification, 2010 Ninth International Workshop on, and High Performance Computational Systems Biology, Second International Workshop on*, Sept 2010, pp. 48–56.

[5] M. Farrar, "Striped Smith-Waterman speeds database searches six times over other SIMD implementations." *Bioinformatics*, vol. 23, no. 2, pp. 156–61, Jan 2007.

[6] G. Navarro, R. Baeza-yates, E. Sutinen, and J. Tarhio, "Indexing Methods for Approximate String Matching," *IEEE Data Engineering Bulletin*, vol. 24, no. 4, pp. 19–27, 2001.

[7] H. Li and R. Durbin, "Fast and accurate long-read alignment with Burrows-Wheeler transform," *Bioinformatics*, vol. 26, no. 5, pp. 589–595, Mar 2010.

[8] R. Luo, T. Wong, J. Zhu, C. M. Liu, X. Zhu, E. Wu, L. K. Lee, H. Lin, W. Zhu, D. W. Cheung, H. F. Ting, S. M. Yiu, S. Peng, C. Yu, Y. Li, R. Li, and T. W. Lam, "SOAP3-dp: fast, accurate and sensitive GPU-based short read aligner," *PLoS ONE*, vol. 8, no. 5, p. e65632, 2013.

[9] Y. Liu and B. Schmidt, "CUSHAW2-GPU: Empowering Faster Gapped Short-Read Alignment Using GPU Computing," *Design Test, IEEE*, vol. 31, no. 1, pp. 31–39, Feb 2014.

[10] B. Langmead and S. L. Salzberg, "Fast gapped-read alignment with Bowtie 2." *Nat. Methods*, vol. 9, no. 4, pp. 357–9, Apr 2012.

[11] Y. Liu, B. Schmidt, and D. L. Maskell, "CUDASW++2.0: enhanced Smith-Waterman protein database search on CUDA-enabled GPUs based on SIMT and virtualized SIMD abstractions." *BMC research notes*, vol. 3, p. 93, 2010.

[12] E. S. Lander, L. M. Linton, B. Birren, C. Nusbaum, and M. C. Zody, "Initial sequencing and analysis of the human genome," *Nature*, vol. 409, no. 6822, pp. 860–921, Feb 2001.

[13] G. R. Abecasis, D. Altshuler, A. Auton, L. D. Brooks, R. M. Durbin *et al.*, "A map of human genome variation from population-scale sequencing," *Nature*, vol. 467, no. 7319, pp. 1061–1073, Oct 2010.