CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. (2012)
Published online in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/cpe.2970

SPECIAL ISSUE PAPER

Implementation and performance analysis of efficient index structures for DNA search algorithms in parallel platforms

Nuno Sebastião*,†, Gustavo Encarnação and Nuno Roma

INESC-ID/IST, Rua Alves Redol, 9, Lisboa, Portugal

SUMMARY

Because of the large datasets that are usually involved in deoxyribonucleic acid (DNA) sequence alignment, the use of optimal local alignment algorithms (e.g., Smith–Waterman) is often unfeasible in practical applications. As such, more efficient solutions that rely on indexed search procedures are often preferred to significantly reduce the time to obtain such alignments. Some data structures that are usually adopted to build such indexes are suffix trees, suffix arrays, and the hash tables of q-mers.

This paper presents a comparative analysis of highly optimized parallel implementations of index-based search algorithms using these three distinct data structures, considering two different parallel platforms: a homogeneous multi-core central processing unit (CPU) and a NVidia Fermi graphics processing unit (GPU). Contrasting to what happens with CPU implementations, the obtained experimental results reveal that GPU implementations clearly favor the suffix arrays, because of the achieved performance in terms of memory accesses. Furthermore, the results also reveal that both the suffix trees and suffix arrays outperform the hash tables of q-mers when dealing with the largest datasets.

When compared with a quad-core CPU, the results demonstrate the possibility to achieve speedups as high as 65 with the GPU when considering a suffix-array index, thus making it an adequate choice for high-performance bioinfomatics applications. Copyright © 2012 John Wiley & Sons, Ltd.

Received 21 December 2011; Revised 10 October 2012; Accepted 5 November 2012

KEY WORDS: GPGPU; indexed search; bioinformatics

1. INTRODUCTION

Bioinformatic applications have had an undisputed role in the discovery of new genetic information contained in the deoxyribonucleic acid (DNA) sequence. The emergence of the *next-generation sequencing technologies* [1], which generate large amounts of short DNA segments (*reads*), has led to the exponential growth of the amount of sequenced DNA. As an example, the December 15, 2011 release of the GenBank [2] database, one of the largest public databases of DNA sequences, includes over 135×10^9 base pairs. Such large datasets pose significant challenges to bioinformatics applications, both in terms of their complexity and in the time required to obtain the results.

One of the most used algorithms to extract information from biological sequences is the Smith–Waterman (S–W) algorithm [3]. It is a dynamic programming algorithm capable of finding the optimal local alignment between any two sequences with sizes n and m in $\mathcal{O}(nm)$ runtime, by performing an exhaustive computation of all possible alignments. The algorithm starts by building

*Correspondence to: Nuno Sebastião, INESC-ID/IST, Rua Alves Redol, 9, Lisboa, Portugal.
†E-mail: nuno.sebastiao@inesc-id.pt

Copyright © 2012 John Wiley & Sons, Ltd.

a dynamic programming matrix (of size $(n+1) \times (m+1)$) and afterwards performs the traceback, which starts at the cell with the highest score. However, for large sequences such as the human genome (with about 3×10^9 base pairs), the exhaustive computation has a runtime that is extremely large, which led to the development of faster but sub-optimal algorithms [4, 5]. Such algorithms typically start by finding an exact match between small subsequences of the query and the reference sequences (a seed). Afterwards, whenever such seed fulfills a given set of conditions (e.g., the minimum value of the score), the alignment is extended to the sides of such sub-sequence using a standard alignment algorithm (e.g. S–W), in which only the much smaller sequences participate instead of the full-sized sequences. This approach significantly reduces the computational requirements when compared with applying the S–W directly to the original sequences. Even when highly optimized and parallel implementations of the S–W algorithm are considered [6, 7], the usage of heuristics algorithms is still preferred when the runtime is a critical factor. On the other hand, whenever high sensitivity levels are required, the S–W algorithm must be used on the full sequences, because of the fact that the heuristic algorithms may miss some important alignments.

Many of the currently adopted heuristic algorithms make use of a pre-prepared index of the reference sequence to accelerate the search of the initial match (the seeds). Such index can be built using different data structures, such as the hash tables of q-mers (substrings of predefined length q) used in BLAST [4] or the suffix trees used in MUMmer [8]. Even though these index structures significantly accelerate the search for the initial match, these algorithms still present a high computational demand, mainly because of the large amount of data they must process. As such, several parallelization techniques have been considered to accelerate these algorithms [9]. On the other hand, with the most recent developments on high-performance computer architectures, a vast set of inexpensive parallel processing structures has emerged, such as multi-core central processing units (CPUs) with four or even more homogeneous processors, or graphics processing units (GPUs) with general purpose computation capabilities that may have as many as 512 processing cores. As a consequence, it has become imperative to adapt the implementation of the most demanding algorithms in order to take the maximum advantage of such processing capabilities [10].

Some previous work, mainly focused on the parallelization of the alignment algorithms in the several platforms that are currently available, has already been presented in the literature [6, 11–13]. The vector processing algorithm proposed by Farrar *et al.* [12] is integrated in the SSEARCH35 application of the FASTA framework and uses single instruction multiple data (SIMD) instructions to parallelize the S–W algorithm on the CPU. Other programs, such as MUMmer [8] and Bowtie [14], are also targeted at the CPU but mainly take advantage of data-level parallelism, because their processing model does not fit so well with SIMD parallelization approaches. MUMmer [8] uses a suffix tree as its index data structure, whereas Bowtie [14] uses the Burrows–Wheeler transform to reduce the memory footprint of its index structure.

Meanwhile, one common observation that has been retained is that the great number of processors that are presently provided in the GPU devices make them ideal for computationally intensive applications, such as bioinformatics algorithms. Nevertheless, their inherent restrictions on a strict *synchronous execution* scheme and on the *memory access* patterns often impose a significant constraint on the adopted programming model and limit the type of algorithms that can be efficiently parallelized on GPU devices. However, independently of the set of constraints that are imposed by the target architecture, it is still necessary to find among the several available algorithms for a given application, which is the best suited for parallelization.

In this scope, the research that is presented in this paper focuses mainly on the implementation and thorough comparison and evaluation of three optimized index structures commonly used in the initial search phase of heuristic DNA alignment algorithms as follows: *suffix trees*, *suffix arrays*, and *hash tables of* q-*mers*. These indexes have been widely adopted to perform DNA search operations of short query sequences against a large reference sequence in general purpose processors. In particular, the hash table index has the ability to more readily return a list of exact matching locations when compared with the other two index structures, because of the fact that it eliminates the need to traverse the tree or iterate through the array to find such locations. Although the hash-based index is mainly used to search for fixed size patterns (whereas the other two structures support the search of patterns of arbitrary length), such fact is not a major drawback when performing DNA

Copyright © 2012 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

sequence alignment, because the alignments can be easily detected using fixed length patterns that are afterwards filtered.

The presented algorithms were implemented using the CUDA API and executed in a NVidia GPU, providing significant speedups when compared with CPU implementations in C/C++. However, contrasting to what happens in conventional CPU implementations, where hash-based indexes usually offer better performances, it is demonstrated that suffix array index-based algorithms are able to offer greater processing rates than the other two data structures, mostly due to the lower data transfer times required to move data into and out of the GPU.

This paper is organized as follows: a brief overview of the three considered data structures that were used to create the index is given in Section 2, whereas in Section 3, a brief description of the GPU architecture is provided. Section 4 presents the implementation details of the three index structures and of the search algorithms in the GPU. The obtained performance results are presented and discussed in Section 5 and the conclusions are drawn in Section 6.

2. INDEXED SEARCH

To accelerate the processing of string matching algorithms, it is common to create an index of the reference (database) string and then use it to accelerate the match with a given query (pattern) string. Several different data structures are currently available to build such index, according to the specific requirements of the application. In the case of DNA alignment, the use of an index that is capable to find the match location of a given query of size n in linear time ($\mathcal{O}(n)$) is highly desirable. As an example, the well-known MUMmer framework [8] makes use of an index structure with such characteristics based on suffix trees. It uses this index to determine the Maximal Unique Matching subsequence between any two sequences.

2.1. Suffix trees

Let s be a sequence with n symbols ($s = s_1..s_n$). The suffix $\hat{s}_m$ of sequence s is defined as the subsequence that contains the last $n - m + 1$ symbols of sequence s, that is, $\hat{s}_m = s_m..s_n$. A suffix tree is a data structure that represents all the suffixes ($\hat{s}_0..\hat{s}_n$) of a given sequence [15, 16]. Each suffix tree is composed of a root node, several internal nodes, and leafs (see example in Figure 1). Each node is connected, by an edge, to at least two child-nodes or leafs and every edge is labeled with a subsequence of the original sequence. The sequence that results from concatenating all the edge labels in the path from the root node to a leaf represents a single suffix of the original sequence. Typically, the original sequence is padded with a special symbol ($) to assure that no suffix of the original sequence is a prefix of another suffix. Hence, an n symbol sequence has n suffixes and the corresponding suffix tree has n leafs. An internal node of the suffix tree represents a repeated subsequence of the original sequence, and the number of occurrences of this subsequence is equal the number of leafs below that node (see Figure 1).

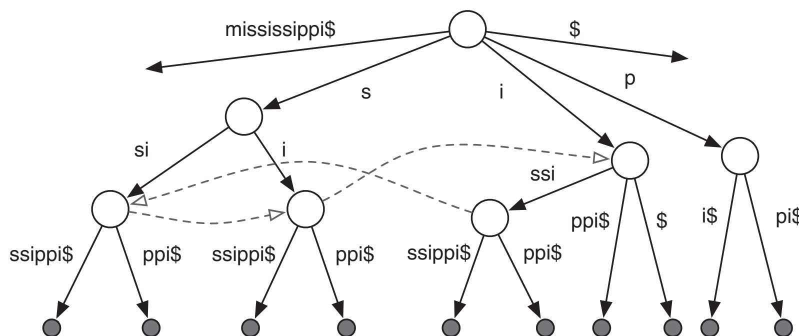

Figure 1. Example of a suffix tree for the string ‘mississippi’, including the suffix links (dashed lines).

Copyright © 2012 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

One typically used algorithm to build the suffix tree was presented in [16]. Such algorithm can build the suffix tree in linear time ($\mathcal{O}(n)$) by making use of a special pointer, denoted as suffix link. If it exists, the suffix link is a pointer between nodes z and $s(z)$ such that given the path label (L) of node z $L(z) = z_0..z_k$, node $s(z)$ has a path label that is $L(s(z)) = z_1..z_k$ (the label is identical except for the first symbol) [17]. Besides using the suffix links to build the suffix tree in linear time, these can also be used to speedup certain types of searches.

By using a data structure based on a suffix tree, it is possible to discover whether a particular query string exists within a larger reference string in a runtime proportional to $\mathcal{O}(n)$, where n is the size of the query string [17]. This is achieved by first creating a suffix tree that represents the reference sequence and then by following the tree edges that match the query string. If it is possible to match all query sequence characters with the characters encountered at an edge path while navigating down the tree, then the query exists somewhere in the reference. Furthermore, by performing a depth-first search from the point where the search stopped and finding all the leaf nodes from that point onwards, it is possible to exactly know *how many times* and *where* the query occurs in the reference sequence in linear time.

Nevertheless, all the significant features provided by suffix trees are offered at the cost of an important drawback, related to the amount of space that is required to store this index structure, which can be as high as 20 times the initial reference size.

2.2. *Suffix arrays*

When compared with suffix trees, suffix arrays [18] are regarded as a more space-efficient structure, typically requiring three to five times less space. This structure (illustrated in Figure 2) can be seen as an array of integers representing the start position of every lexicographically ordered suffix of a string. The improvement that allows suffix arrays to use less space than suffix trees comes from the fact that the array simply needs to hold a pointer to the start of the suffix (or an index of the corresponding text) to represent each of these suffixes. This means that the element of the suffix array that holds the value '0' points to the first character of the text (assuming the text is a zero indexed array of characters) and the suffix it represents corresponds to the whole text.

The most straightforward way to construct such data structure is to simply create an array with all the suffix elements placed in ascending order and then apply a sorting algorithm to properly sort the suffixes (see Figure 2). A more complex alternative is to start by creating a *suffix tree* and then find all the leafs in the tree. This approach, although being faster than the direct sorting method, has the important drawback of requiring much more space to hold the initial suffix tree.

The usage of a suffix array for string matching procedures (in this case for DNA sequence alignment) is similar to using any other sorted array to search for a given element. The only difference is the way that two items are compared with each other. Because the values in the suffix

Suffix	Index
mississippi	0
ississippi	1
ssissippi	2
sissippi	3
issippi	4
ssippi	5
sippi	6
ippi	7
ppi	8
pi	9
i	10

(a)

Suffix	Index
i	10
ippi	7
issippi	4
ississippi	1
mississippi	0
pi	9
ppi	8
sippi	6
sissippi	3
ssippi	5
ssissippi	2

(b)

Figure 2. Suffix array for the string 'mississippi': (a) unsorted suffix array and (b) lexicographically sorted suffix array.

Copyright © 2012 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

array are not strings but indexes (pointers) of a string, it is not the values themselves that must be compared but the text they point to. Hence, when comparing two given elements of the suffix array, not only the characters they point to must be compared but also, if they are equal, the subsequent characters.

The overall performance of the matching function will directly reflect the efficiency of the search algorithm used in the array. Binary search algorithms are generally used, where the array is repeatedly divided in half until the desired element is reached. Hence, suffix arrays solve the string matching problem with an $\mathcal{O}(n \log m)$ complexity, where n is the length of the query and m is the length of the reference.

2.3. Hash tables of q-mers

A hash table is a data structure used to map a given key (e.g., a sub-string) to its corresponding object (e.g., the start position in the string) [19]. The hash values are calculated using a specific and application dependent hash function, which maps each of the several considered keys with the corresponding objects. These hash values are typically stored in a predefined set of slots (table). Typically, the hash function does not guarantee that for every possible input (key), there is a different output (hash value), thus leading to the possible occurrence of *hash collisions* (two distinct keys having the same hash value). A typical solution to solve the collisions problem is to use several *buckets* that hold all the objects with the same key.

A q-mer is defined as any sub-string with length q of a given string. In a hash table of q-mers, all the objects have the same size (q) and the number of possible keys is limited to the set of possible combinations of the q elements (e.g., using an alphabet with four symbols, there are $4^3 = 64$ possible *3*-mers). Because of the fact that the whole set of keys is known, it is generally possible to design a hash function that is collision free (i.e., for every distinct q-mer (key), the hash function returns a different value), therefore avoiding the need to solve the hash collision problem. Furthermore, in this type of hash table, the hash value can be used to directly index the array that contains the data to be stored, therefore avoiding the need to store the key value in the table.

For the purpose of finding the occurrences of a given q-mer in a reference sequence, the object to which the hash table refers to is a list of positions on the reference sequence in which the corresponding q-mer occurs during the subsequent search phase. After building the hash table of q-mers for the considered reference sequence, it is possible to quickly find all the occurrences of a given string of size q, by simply calculating the hash value of such string and then using that value to perform a lookup on the hash table of the reference sequence. In terms of performance, the hash table can be used to find the locations of all q-mers of a query string in a runtime proportional to $\mathcal{O}(n)$, where n is the length of the query.

In Figure 3, an example of a hash table of *3*-mers is presented, where $f(s) = A(s[0]) * 10^0 + A(s[1]) * 10^1 + A(s[2]) * 10^2$ is the hash function used to build the table ($s[n]$ represents the symbol at position n of the *3*-mer string and $A(x)$ represents the ASCII value of character x). For the

(a)

3-mer	Index
mis	0
iss	1
ssi	2
sis	3
iss	4
ssi	5
sip	6
ipp	7
ppi	8

(b)

3-mer (key)	Hash Value	Index(es)
mis	12659	0
iss	12755	1, 4
ssi	11765	2, 5
sis	12665	3
sip	12365	6
ipp	12425	7
ppi	11732	8

Figure 3. q-mers and corresponding hash table for the string 'mississippi': (a) list of 3-mers and corresponding index and (b) hash table showing the 3-mer (key), its hash value, and the indexes (object).

Copyright © 2012 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

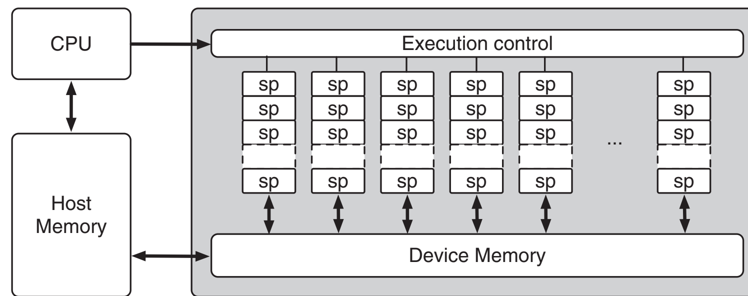

Figure 4. Simplified view of the graphics processing unit architecture.

example string, the ASCII values of the used characters are the following: $A(m) = 109$; $A(i) = 105$; $A(s) = 115$; $A(p) = 112$.

3. GPU ARCHITECTURE AND PROGRAMMING MODEL

The basic building blocks in NVidia's *Fermi* architecture [20] are the streaming multiprocessor (SM), which can be seen in Figure 4 together with a simplified diagram of the architecture and its interconnection with the host computing system. Each SM contains 32 processing cores (sp) and two warp schedulers. A warp is a set of 32 parallel threads which are concurrently executed in sets of 16 threads (a half-warp) on the same SM. The most important characteristic of these SMs is the followed single instruction multiple thread paradigm, which means that all the processing cores executing a half-warp always execute the same instruction on the different threads. Hence, if a thread within a warp performs a branch that diverges from code executed by other threads, the processing of the several threads of such warp will be serialized, thus presenting a major challenge to optimize the GPU code and avoid a significant loss of efficiency.

Besides providing a large number of processing elements, this GPU family also offers a large main memory with a high access bandwidth (six 64-bit wide memory interfaces). However, each access to this memory bank incurs in a high latency (in the order of hundreds of clock cycles). As a consequence, whenever possible, the memory management unit of the GPU tries to *coalesce* (join together 32 individual memory requests), in order to improve the resulting performance. Therefore, threads in the same warp should access neighboring memory positions so that a single memory transaction is capable of providing data to several threads of an individual warp. Moreover, the new *Fermi* architecture also provides a unified 768kB L2 cache for accessing the entire main memory, which might significantly improve the memory access time for memory access patterns which cannot be efficiently coalesced.

One particular aspect that must be taken into account is that the GPU threads can only access the device local memory. Thus, it is necessary to copy the data to be processed from the host memory to the GPU memory before any operations can be performed by the threads. At the end of processing, it is also required to copy back the results to the host memory for further processing.

4. ALGORITHM IMPLEMENTATIONS

To accelerate the execution of exact matching algorithms, the adoption of current high-performance parallel platforms, such as homogeneous multi-core CPU or GPU accelerators, has been regarded as highly promising.

In the particular case of indexed string matching algorithms, the index is firstly built, in a preliminary stage, by using the reference sequence data. Then, the most usual acceleration strategy adopts a pure and massive data-level parallelism approach, where the several queries that have to be matched with the same reference sequence are distributed by the several worker threads. Hence, the reference index data structure can be built offline and used afterwards for the search procedure.

Copyright © 2012 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

Because the reference sequence is the same for a large number of queries, the initial effort of building the index is widely amortized.

As mentioned in the previous section, it is necessary to copy the data into the local memory of the GPU before any processing can be performed. After processing, it is also necessary to transfer the results out of the GPU. Therefore, when determining the total processing time of an algorithm that executes in the GPU, it is necessary to take into account these memory transfers. These data transfers may adversely impact the total execution time, especially when dealing with large datasets.

4.1. Suffix trees

The first step in the search procedure for DNA sequences is to build the suffix tree of the reference/database sequence. This step is usually performed in the CPU, because it is an inherently irregular and sequential process. The suffix tree is then transferred to the GPU memory, to be accessed by the several concurrent threads, each one aligning a different query sequence to the same reference sequence.

The suffix tree is constructed from the reference/database string and is afterwards transformed into a flattened tree consisting of an array of edges. Each node is represented in the flattened tree, as seen in Figure 5 by its set of outgoing edges, where each edge contains the following: (i) its starting index in the reference sequence; (ii) the edge length; and (iii) the index (in the array) of the first edge of its destination node. Thus, each edge can be represented using three integers. However, to allow a perfect alignment of the memory accesses, the representation of each single edge is padded to hold exactly four integers (128 bits). Furthermore, each node always contemplates space for its four possible edges (representing an A, C, G, or T symbol), although it is possible that some of these may be filled out as *fake* edges. The need for flattening the tree arises from the fact that array indexes are more conveniently addressed in the memory space of the GPU than memory pointers. Furthermore, the traversal of the tree leads to an unpredictable access pattern that may significantly affect the performance of memory accesses, mostly due to the inability to coalesce them.

Because the suffix tree only includes references to the original sequence (position indexes), besides transferring the flattened tree to the GPU, it is also necessary to transfer the original reference sequence. Hence, to save space and to optimize certain parts of the matching function, the *reference* string is stored as a second array of integers. Each of these integers holds 16 nucleotides from the original sequence, each one represented using only two bits.

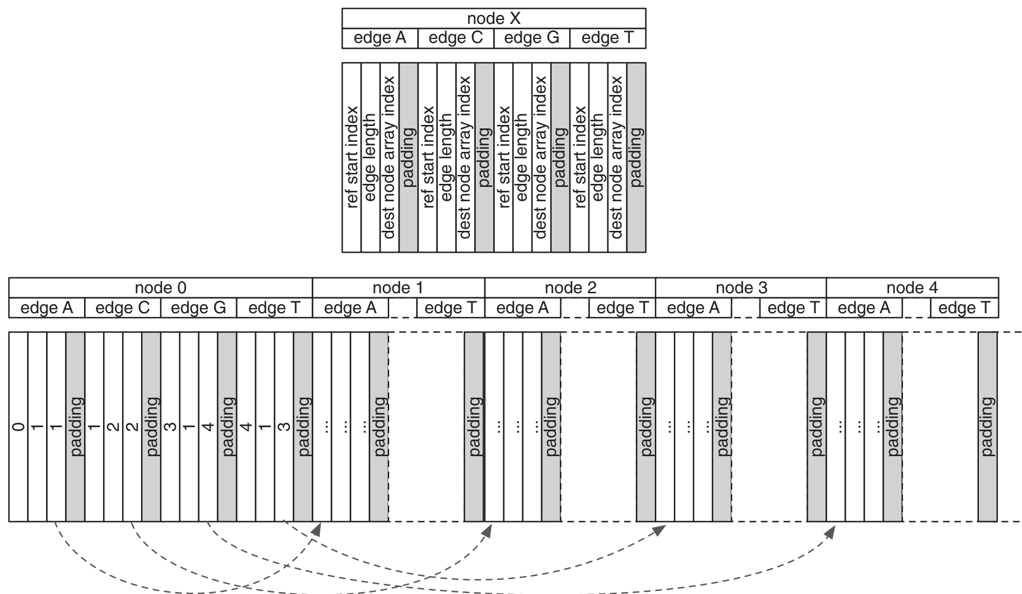

Figure 5. Example of a flattened suffix tree.

Copyright © 2012 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

The original DNA *query* sequences are stored in the GPU global memory in their string format. However, similarly to what was performed to the reference sequence, to maximally exploit the available memory bandwidth, each set of 16 nucleotides is packed into a single 32-bit integer. Furthermore, the symbols of the different query sequences are interleaved. In fact, due to the particular way that query characters are accessed using suffix trees (single character comparison, instead of 16 characters) these characters are stored in *reverse* order in each integer cell: the first character corresponding to the lowest order bits and the following characters being mapped in the highest order bits. Such *reverse* order is highly preferred, because it allows to obtain the various characters by using a simple and fast shift right instruction, followed by a binary AND logical operation, using always the same mask ('11').

Because of the adopted encoding of the queries by using only two bits, it might happen that the last memory position represents less than 16 nucleotides. This singularity and the fact that the queries might differ in size makes it necessary to create an auxiliary structure that specifies how many symbols each query actually has, so that their end can be determined during the matching process.

The implemented matching algorithm, which is executed by each thread in the GPU, is depicted in Algorithm 1. The first step in the matching process consists of reading the *query* sequence data. The first 16 nucleotides are read into a buffer and the number of valid nucleotides (in case the query is less than 16 nucleotides long) is calculated. After filling the query buffer, the first character is extracted from it and assigned to a 'test character'. Afterwards, the whole buffer is shifted by two bits, to prevent the same character from being used again.

Then, the test character is used to read the first edge that needs to be checked by calculating its position in the flattened tree using the character as an offset. Considering that the algorithm starts navigating the tree from the root node and the edges of the root node start at index 0, the edge leading out of the root node by the 'test character' will be at position `tree[0 + test char]`.

Once the query buffer is filled and there is an edge to follow, the matching becomes a cycle of comparisons. The cycle begins by comparing two characters: the test character and the first character in the edge. As soon as a mismatch is found, it is known that the *query* under processing does not exist within the *reference* sequence. On the other hand, if a point is reached where the query buffer is empty and there are no more characters to read, then the end of the match has been reached, the query exists within the reference sequence and all the leafs that can be reached from the destination node of the current edge represent one match.

4.2. *Suffix arrays*

When compared with suffix trees, the suffix arrays are usually regarded as a more space-efficient implementation of the index structure. Although their search time is asymptotically higher than the search time of suffix trees, in many applications, their smaller size leads to similar or even better performance levels [21], because of the attainable better cache performances.

The suffix array is a uni-dimensional array of integers and its access pattern is usually as unpredictable as in the case of suffix trees. Therefore, similar problems are encountered in terms

Algorithm 1 DNA matching using suffix trees

```
Read query[thread ID] to query buffer

Extract 'test character' Read edge[test character]

While (test character == reference[edge character]) {
 edge character++

 Refill the buffer if necessary
 Return the edge's destination if the buffer is empty

 Extract next 'test character'
 If necessary
  Read edge[edge destination + test character]
}
Return mismatch
```

Copyright © 2012 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

of coalescing the memory accesses. Just like in the case of the suffix tree implementation, it is also necessary to transfer the original reference sequence, as well as the query sequences, to the GPU memory. The data structure is the same as the one that was adopted to hold the query sequences for the suffix tree implementation.

The matching algorithm in the GPU was implemented by conducting a binary search in the pre-processed array. In each step, the whole query is compared against one entire suffix, contrasting to what happens in the suffix tree implementation, where a single edge is compared. The main consequence of this improved approach is that once the suffix to be considered is determined, the following memory accesses become sequential until it becomes necessary to re-pick the suffix. Therefore, by compacting the original reference sequence representation (8-bit characters vector) to an array of integers where, just as in the queries, each integer holds 16 2-bit nucleotides, the memory accesses can be reduced by 16 times. One additional (but also important) advantage that also arises is concerned with the possibility to simultaneously compare, in the best case scenario, 16 characters in a single instruction, leading to a rather efficient combination of the SIMD parallel programming model with the SIMT model, natively exploited by the GPU architecture.

The proposed matching algorithm, which is executed by the GPU thread, consists of two nested loops depicted in Algorithms 2 and 3, respectively. The first loop is executed until all possible search options have been exhausted. Because this implementation is based on a binary search algorithm, such situation happens whenever the left and right pointers are adjacent ($right - left = 1$). The first task of this loop is to pick the next suffix array element to be considered. This is performed by calculating the mid-point between the left and right pointers. After picking which suffix to use, it is necessary to read the *query* and *suffix* sequences into a buffer. The read of the first is straightforward, because it is always aligned. Nevertheless, a special care must be taken into account when reading the suffix, because it might not be aligned and thus the higher bits of the corresponding memory position will be invalid. Before the comparison cycle begins, it is also necessary to assure that the query buffer and the suffix buffer hold the same number of packed characters, because 16 symbols are compared at once.

The inner loop, depicted in Algorithm 3, is the comparison cycle ('==') that runs while the sequences are equal and there are more symbols to be compared in the sequences. When the algorithm enters into the inner loop, the buffers hold the same number of valid symbols. However, it is not required that the number of symbols in the buffers is always the maximum buffer capacity. Consequently, the smaller buffer will empty sooner than the larger one, which will still have some data waiting to be compared. The main task of the inner cycle is to read the data into any of the buffers that might have become empty after the last comparison in order to discard any previously used data and to make sure that both buffers always contain the same amount of symbols.

An interesting side effect that arises from using this comparison method is that the kernel is more computationally intensive, with more logic–arithmetic operations than memory accesses, which significantly benefits the efficiency of the parallel execution in the GPU.

Algorithm 2 Matching using suffix arrays - outer cycle

```
While (right - left != 1) {
 pivot = (left + right) / 2

 Read reference buffer
 Calculate reference buffer size

 Read query buffer = query[thread ID]
 calculate query buffer size

 Remove trailing characters from largest buffer

 < Inner cycle >
}
```

Copyright © 2012 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

Algorithm 3 Matching using suffix arrays - inner cycle

```
While (
   reference compare buffer == query compare buffer AND
   reference buffer size > 0 AND
   query buffer size > 0 ) {

 Set smallest buffer size to 0
 Remove leading characters from largest buffer
 Update largest buffer's size

 Read data into any buffer of size 0
 Calculate the size of updated buffers

 Remove trailing characters from largest buffer
}
```

4.3. *Hash-table of q-mers*

As it was previously mentioned in Section 2.3, the number of distinct keys in a hash table of q-mers used for exact string matching is limited to the number of possible combinations of the q-mers. In the particular case of DNA sequences (an alphabet of four symbols), the number of possible combinations is 4^q. In this specific application, the size of the q-mers is typically smaller than or equal to 13 ($q \leqslant 13$), which results in a hash table with a maximum number of entries of approximately 64×10^6 (2^{26}). By using a collision free hash function (perfect hash function), the data to be stored in the hash table is reduced to the positions in the reference sequence in which the corresponding q-mer occurs.

The hash value of string $m[0..q-1]$ (the q-mer) is calculated by using the expression presented in Equation (1), where function $B(x)$ converts each of the four distinct symbols used in the DNA sequence (A,C,G, and T) to, respectively, an integer value in the range (0..3). Such integers are represented using a 2-bit encoding. This hash function can be proven to be a perfect hash function and can be easily calculated using shift and logical operators, thus making it extremely efficient in terms of computation.

$$f(m,q) = \sum_{n=0}^{q-1} B(m[q-1-n]) \times 4^n. \tag{1}$$

The hash table for the considered reference sequence under analysis is constructed using the CPU. Afterwards, it is used to lookup the hash values of the several queries to directly obtain the indexes of the reference sequence where such q-mers of the query sequences occurred in the reference sequence.

To concurrently perform the hash calculations of the query sequences, the several sequences are concatenated into a single string, the *global query*, and afterwards transferred to the GPU, where the hash value of every q-mer of such string is calculated. At this point, the queries are still encoded using the usual 8-bit ASCII representation, thus avoiding the need to previously re-encode the query strings at the CPU. However, it is worth noting that some of the q-mers for which a hash value is calculated do not really occur in the original query sequences, because of the fact that they are the result of joining two distinct query sequences. These values are afterwards filtered out in a post-processing step. With such strategy, it is no longer necessary to check the bounds of every single query, thus making the computation of the hashes in the GPU more regular, which reduces thread divergence and improves the overall performance of the computation.

Considering a string s of length k, it is clear that it has $k-q+1$ q-mers. Hence, for the particular case of the hash function proposed in Equation (1), the computation of the hash value of the $(i+1)$-th q-mer (m_i) is carried out by reading the sub-string ($s[i..q-1+i]$) and by subsequently applying the mentioned hash function $f(m,q)$. Therefore, it is possible to present a complete definition of the hash values for all the q-mers of a string s, as defined in Equation (2).

Copyright © 2012 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

Algorithm 4 Hash values calculation

```
initialize i=0

While (workToBeDone) {
 let location = threadID + i x 'number of threads'

 read string globalQuery[location .. location + q - 1] to qmer

 apply hash function to qmer

 write hash value to hashValueArray[threadID + i x 'number of threads']

 i++
}
```

$$f'(i,s,q) = \sum_{n=0}^{q-1} B(s[q-1-n+i]) \times 4^n. \quad (2)$$

A particular aspect of the used GPU architecture is the fact that to optimize the memory access pattern, a total of 128 bytes should be read in a single coalesced transaction. Thus, the maximum performance is achieved when each of the 32 threads that compose a warp accesses a 4-byte word (an integer). Because of the fact that the adopted representation of the *global query* uses one byte for each DNA symbol, each of the threads reads, at each iteration, a 4-byte word, thus being able to process four symbols with a single memory read. In terms of write operations, each hash value is stored in a 4-byte word, thus being naturally optimized. This ensures an optimal memory access pattern, with maximally coalesced memory accesses (the threads of the same warp should read and write to adjacent memory positions) both in terms of reading and writing.

The adopted parallel algorithm, as seen in Algorithm 4, ensures that the 32 threads of the same warp calculate the hash value of consecutive q-mers of the *global query*. This ensures that memory write operations are coalesced. As a consequence, each thread will need to read all of the q symbols that constitute the q-mer. These read accesses made by the threads of a given warp can still be coalesced (in this case, the same memory position may be read by several threads), thus maintaining the performance levels.

5. EVALUATION OF INDEX-BASED SEARCH ALGORITHMS

To evaluate the conceived highly concurrent implementations of the considered index-based search algorithms, a set of real DNA sequence data obtained from the GenBank database was used. The reference sequence, which was used to build the indexes, corresponds to the first 10×10^6 nucleotides of the *Mus Musculus* Chromosome 1 (C_000067.5). The considered set of query sequences are 200 nucleotides long and came from a mix of the DNA sequences extracted from the *Mus Musculus* Chromosome 1 (C_000067.5) and the *Rattus Norvegicus* Chromosome 1 (RGSC3.4.64). Several collections of query sequences were used in the experiments, each one composed by a different number of elements, ranging from 1024 to 4,194,304 queries (each with 200 nucleotides).

The previously described algorithms were evaluated in a computational system composed of an Intel Core i7 950 quad-core processor, running at 3 GHz, with 6 GB of RAM. This platform also includes a NVidia GeForce GTX 580 GPU, with 512 processing cores running at 1.54 GHz and 1.5 GB of RAM. The system runs the Linux operating system with kernel version 2.6.34.8-0.2.

All the presented execution times were obtained by averaging the values of 10 executions of the same experience. The variations in the obtained runtimes are presented as the maximum deviation among all the datasets used in the specific execution context. These deviations are calculated as $d = \max_{i=1..n}\{\max_{j=1..10}\{|x_{ij} - \hat{x}_i|\}/\hat{x}_i\}$, where $\hat{x}_i$ represents the average of the time values obtained in the different executions (x_{ij}) when using the dataset i (the specific set of queries used for the experiment). This metric represents the maximum relative variation that occurred during the execution of the experiments.

Copyright © 2012 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

5.1. Sequential and parallel implementations in homogeneous CPUs

The first evaluation of the conceived parallel implementations of the considered algorithms was performed by comparing the performance provided by both sequential and concurrent executions of the DNA search algorithms based on the *suffix tree*, the *suffix array*, and the *q-mer hash table* indexes. The implemented algorithms were compiled to be executed in a homogeneous multi-core CPU, by making use of the POSIX threads API to support the parallel execution of two, four, and eight concurrent threads. The setup corresponding to eight concurrent threads was assessed by making use of Intel's Hyper-Threading technology, leading to a slight lower efficiency in what concerns the achieved acceleration. On the whole, this experimental procedure not only allowed to assess the scalability of the three index-based algorithms but also provided a comparative evaluation with other popular and highly efficient CPU-based software, namely, Bowtie and SSEARCH35 (an optimized CPU implementation of the S–W algorithm).

The obtained results for pure sequential executions are presented in Figure 6, whereas Table I (first column) shows the maximum relative deviations for these experiments. As it can be observed from those execution times, although the asymptotic runtime corresponding to the suffix arrays is slightly greater than that of the suffix trees, in practice, the performance of both implementations is quite similar. This result had already been observed in [21] and is mainly due to a more efficient usage of the cache memory by the suffix array, which is achieved because of its smaller and more regular structure. It is also possible to observe that the hash-table of q-mers, which has an asymptotic runtime equal to the suffix tree ($\mathcal{O}(n)$), presents the smallest runtime, mainly because of both the simplicity of the used hash function and its more regular memory access pattern, which improves the resulting cache performance. Furthermore, by comparing the obtained execution time results with those of the Bowtie and SSEARCH35 software frameworks (see Figure 6), it is possible to

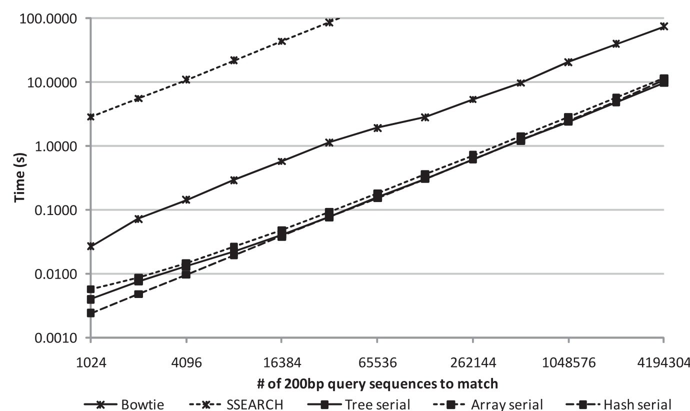

Figure 6. Performance comparison of the sequential implementations of the three index-based search algorithms with Bowtie and SSearch35.

Table I. Maximum relative deviation (%) of the execution time results obtained in the homogeneous multi-core CPU.

	CPU threads			
Algorithm	1	2	4	8
Suffix tree	1.26	1.28	1.48	1.12
Suffix array	1.50	0.85	1.93	1.81
Hash table	1.45	1.62	1.83	1.71
SSearch35	2.14	–	–	–
Bowtie	3.87	–	–	–

CPU, central processing unit.

Copyright © 2012 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

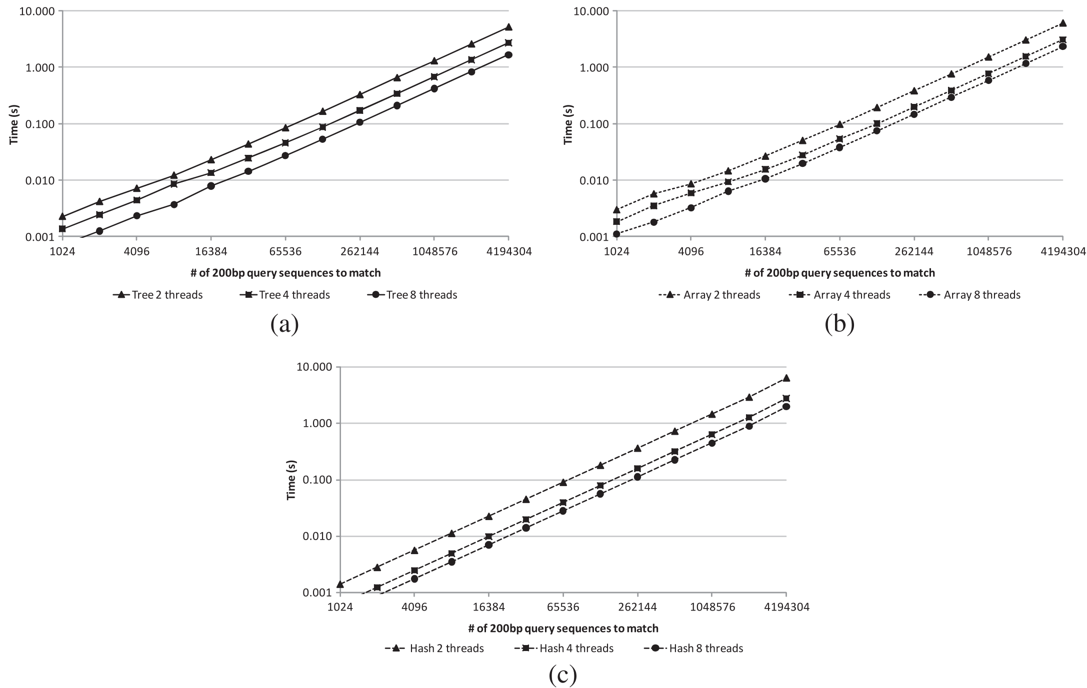

Figure 7. Performance evaluation of the multi-threaded implementations of the index-based search algorithms in an Intel i7 multi-core CPU: (a) suffix tree index; (b) suffix array index; and (c) hash table index.

Table II. Average speedup of the multi-threaded central processing unit implementations, executed in an Intel i7 quad-core CPU.

	CPU threads		
Algorithm	2	4	8
Suffix tree	1.98	3.69	5.91
Suffix array	1.99	3.60	5.44
Hash table	1.90	3.71	5.98

CPU, central processing unit.

observe that the implemented index-based algorithms are significantly faster, thus plenty justifying their adoption whenever high-performance DNA alignment is required.

The runtime results obtained from the parallel execution of these algorithms in the considered homogeneous Intel multi-core processor are presented in Figure 7, with the average speedup values presented in Table II and the corresponding maximum deviations presented in Table I. The presented results reveal that the algorithms scale almost linearly with the number of considered CPU cores. It can also be observed that there is an attenuation on the attained speedup (compared with the corresponding asymptotical value) that is most likely due to access contention on the shared resources (e.g., memories). Note that the eight-thread implementation makes use of the hyper-threading technology, which only improves the usage of the quad-core CPU, thus not yielding similar results as those obtained when executing with independent cores. The maximum deviation results, presented in Table I, demonstrate that there is a small dispersion of the execution times of the several experiments (below 2%) for the implemented index-based search algorithms.

5.2. Parallel implementations in a GPU accelerator

The performance of the conceived concurrent algorithms was then assessed in a state-of-the-art GPU platform, namely, an NVidia GeForce GTX 580. The obtained execution time results are presented

Copyright © 2012 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

in Figure 8 and the associated runtime deviation is shown in Table III. This chart also includes a comparison with another popular GPU-based DNA alignment framework, based on suffix trees: MUMmerGPU [9]. The presented results correspond to the *total* execution time of the algorithms while searching for the corresponding number of query sequences in the reference sequence. The *total* execution time considers all the required data transfers (host to GPU and GPU to host), as well as the kernel execution time. As it is possible to observe in Figure 9, the data input time is very significant in suffix tree and suffix array index-based search algorithms, because the large index data structure must always be transferred to the GPU device memory. In fact, when the number of query sequences to be searched is very small, this data input time is the main responsible for the modest performance values provided by the GPU implementations, when compared with the corresponding CPU implementations. As expected, the data output time is the same for the suffix tree and suffix

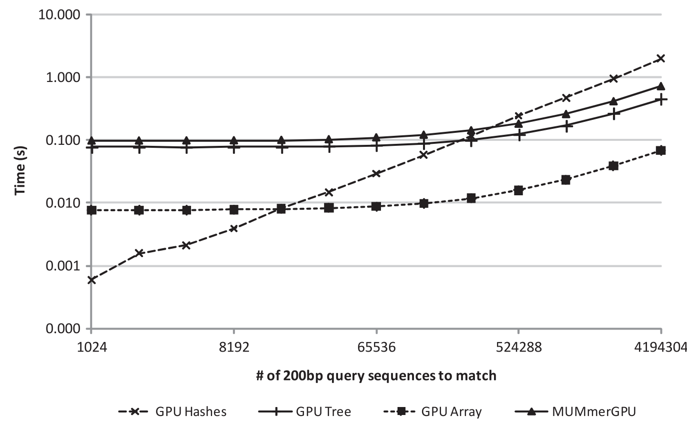

Figure 8. Performance evaluation of the considered index-based search algorithms in an NVidia GeForce GTX580 GPU.

Table III. Maximum relative runtime deviation (%) of the obtained results in the GPU.

Algorithm	Deviation (%)
Suffix tree	0.48
Suffix array	2.56
Hash table	1.07
MUMmerGPU	0.47

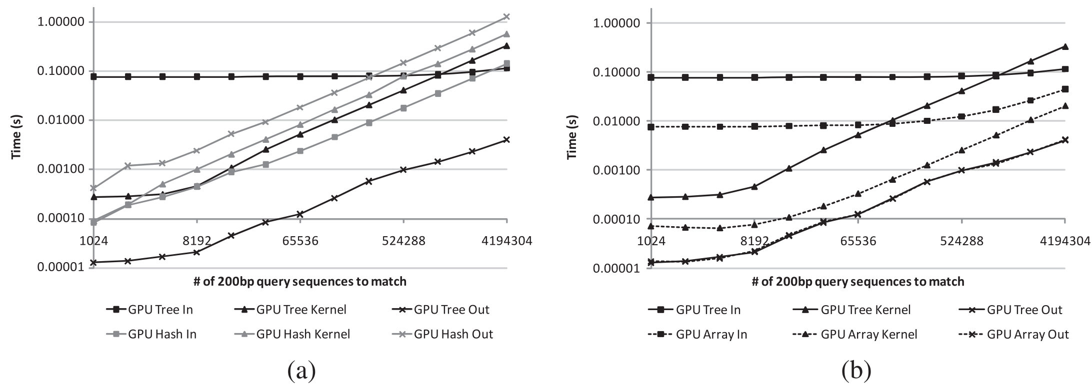

Figure 9. Comparison of the data transfer and kernel execution times for the three index implementations in the NVidia GeForce GTX 580 graphics processing unit: (a) suffix tree and hash table indexes comparison and (b) suffix tree and suffix array indexes comparison.

Copyright © 2012 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

array index-based search algorithms, because of the fact that the output of both algorithms consists of the same data (note that in Figure 9(b), the lines overlap), whereas for the hash table index-based algorithm, such time is the highest.

On the other hand, the obtained results show that the GPU implementation of the hash table index-based algorithm performs better for datasets with fewer query sequences, mainly because of the reduced data input time. However, for a larger number of query sequences, not only is the kernel time of the hash table index-based algorithm higher than that of the other two indexes but also the data output time is significantly larger. Therefore, the observation that should be retained is that for a larger number of query sequences (commonly adopted by this application domain), the GPU implementations of the suffix tree, and the suffix array index-based algorithms offer a significantly better performance, with speedup values as high as 65 (*suffix array* implementation) when compared with homogeneous multi-core CPUs. From the repeated executions of the same experiments, it is possible to conclude that the maximum deviation of the runtime of the implemented algorithms is smaller than 2.6%, as shown in Table III.

The obtained execution results for the GPU implementations were also compared with two other software frameworks: MUMmerGPU [9] and CUDASW++ [22]. While the runtime values obtained with MUMmerGPU were consistently worst than those obtained with the considered implementations of the suffix tree and the suffix array, a comparison with CUDASW++ software was not possible because it has a strict limitation on the maximum reference size of about 64×10^3 base pairs.

The GPU L2 cache memory, available in the *Fermi* GPU architecture, may significantly improve the performance of those algorithms whose memory access patterns can not be efficiently coalesced. In fact, the improved cache efficiency of the GPU implementation of the suffix array is one important factor that contributes to its smaller execution time. In Figure 10, it is presented the L2 cache hit rate for the GPU implementations of the considered indexes. From these results it is possible to observe that the GPU implementation of the suffix array achieves a much higher cache hit rate, mainly because of the inherent smaller size of the index, which allows for a larger fraction of it to be available in the L2 cache for further use. On the other hand, the suffix tree tends to present a quite smaller cache hit ratio due to its larger space requirements, which reduce the fraction of the index structure that may be available in the cache, therefore reducing the probability of a cache hit.

The performance of the implemented indexes was also assessed when using reference sequences with different sizes. In Figure 11, it is possible to observe the execution times for aligning a set of 2,097,152 queries (each with 250 nucleotides) to a reference sequence with size ranging from 1.25×10^6 to 10×10^6 nucleotides. It is possible to observe that for the hash table implementation, the execution time is the same for all of the reference sequence sizes, because of the fact that only the hash values of the query sequences are performed in the GPU. On the other hand, the execution time for both the suffix array and the suffix tree implementation increases with the size of the reference

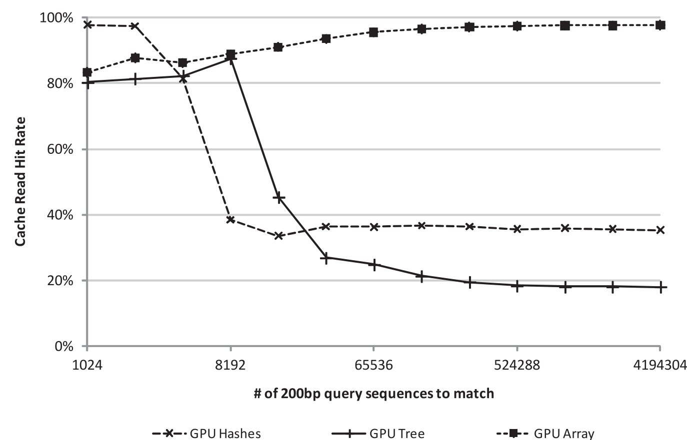

Figure 10. Cache hit rate for the three index implementations in the NVidia GeForce GTX 580 GPU.

Copyright © 2012 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

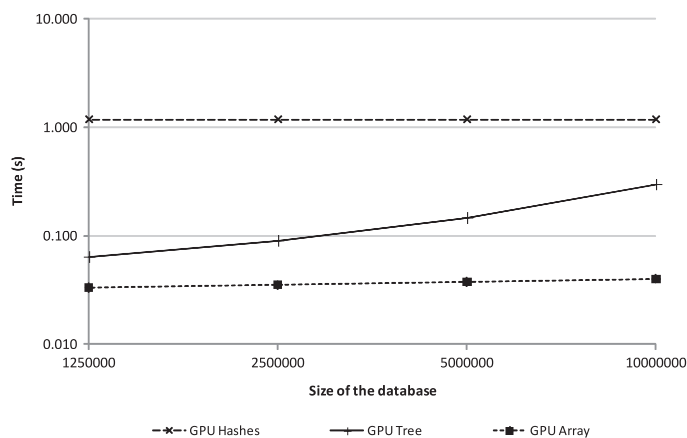

Figure 11. Execution time of the considered index implementations in the NVidia GeForce GTX 580 graphics processing unit for different reference sequence sizes.

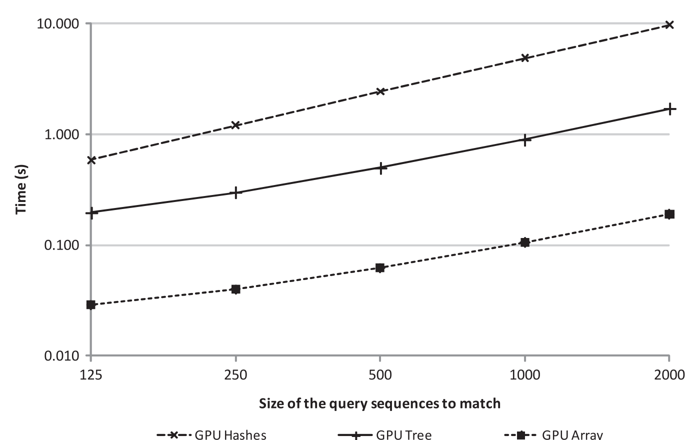

Figure 12. Execution time of the considered index implementations in the NVidia GeForce GTX 580 graphics processing unit for different query sequence sizes.

sequence. Nevertheless, for the suffix array, such increase is almost unnoticeable, because of the fact that the overall time is dominated by the data transfer operations.

Besides analyzing the execution time for various reference sequence sizes, the influence of the size of the query sequences in the execution time was also assessed. In Figure 12, the execution times for aligning a set of 2,097,152 queries is presented, with sizes ranging from 125 to 2000 nucleotides, to a reference sequence with 10×10^6 nucleotides. As expected, all of the considered index-based algorithms have an almost linear dependency with the size of the query sequences.

5.3. *Discussion*

The greater number of processing elements available in the used GPU device, when compared with the CPU, allows for a higher degree of parallelization of the algorithms. Such parallelization levels may lead to a potentially impressive speedup (over 100x). However, the architecture of the GPU, with its constraints on parallel memory accesses and thread execution model, significantly reduces the obtained performance when algorithms with inherently random access patterns, like those that were considered in this manuscript, are parallelized. Furthermore, the GPU accesses the data to be processed in its own local memory, which makes it necessary to copy/move all the data from

Copyright © 2012 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

the host main memory to the GPU local memory. With large datasets, such copy/move operations significantly impact the overall speedup, as it can be seen from the obtained CPU and GPU results. Nevertheless, the results presented in the previous section show that for very large datasets, as those commonly used in bioinformatics, the GPU implementations obtain a better performance than their CPU counterparts, thus making the GPU device a good candidate for the acceleration of bioinformatics algorithms.

Furthermore, the conducted experimental observations revealed that unlike the obtained CPU performance results, presented in Figure 7 and Table II, the GPU implementations clearly favor the suffix array index structure. In fact, compared with the suffix tree, the suffix array presents a more regular execution flow and is more efficient in terms of the cache memory usage. Furthermore, the space occupied by the suffix array index is much smaller than that of the suffix tree index, which makes the suffix array implementation to always present a much lower transfer time from the host to the GPU device. On the other hand, when compared with the hash table of q-mers, the suffix array achieves a better performance mainly due to the much smaller amount of data that it needs to output its results and to the consistently smaller kernel execution time. Overall, the proposed suffix array implementation achieves the best results for high-performance GPU implementations. Moreover, the obtained results suggest that it would be advantageous to implement a hybrid algorithm that takes advantage of the hash tables for the smaller datasets and of the suffix arrays for the larger datasets.

Finally, it is worth noting that further improvements could still be obtained by parallelizing the processing of these algorithms on a large cluster by using a processing model entirely similar to the one presented in [7]. This approach would be particularly advantageous in data-level parallelization schemes, when considering the alignment of a large number of queries to a large number of sequences. This would allow the partitioning of the data to be processed among the several cluster nodes, thus significantly improving the parallelism efficiency of the execution, with the consequent increase in the resulting throughput.

6. CONCLUSIONS

This paper presented the implementation and a comparative evaluation of three index data structures that are especially suited for accelerating DNA sequence alignment in bioinformatics applications: the *suffix tree*, the *suffix array*, and the *hash table of* q-*mers*. These three indexes were thoroughly compared in terms of performance, when implemented using two different state-of-the-art parallel platforms: a multi-core CPU system and a NVidia GeForce GTX580 GPU (Fermi architecture).

From the obtained results, it was observed that although the asymptotic search time of the suffix array ($\mathcal{O}(n \log m)$) is higher than those corresponding to the suffix tree and to the hash table of q-mers ($\mathcal{O}(n)$), practical implementations in a multi-core CPU revealed that the performance of the hash-table of q-mers is the highest among the three and that the performance of the array is very similar to the one corresponding to the trees. However, the same is not true when these algorithms are implemented in GPUs. As opposed to the asymptotic analysis, the obtained experimental results show that for this specific parallel architecture, the suffix arrays clearly outperform both the suffix trees and the hash tables when dealing with large datasets, mainly because of their smaller amount of required memory space and to the improved usage of cache.

ACKNOWLEDGEMENTS

This work was partially supported by national funds through Fundação para a Ciência e a Tecnologia (FCT) under project "*HELIX: Heterogeneous Multi-Core Architecture for Biological Sequence Analysis*" (reference number PTDC/EEA-ELC/113999/2009), project "*TAGS: The Power of the Short — Tools and Algorithms for next Generation Sequencing Applications*" (reference number PTDC/EIA-EIA/112283/2009), project PEst-OE/EEI/LA0021/2011, and by the PhD grant with reference SFRH/BD/43497/2008.

Copyright © 2012 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

REFERENCES

1. Shendure J, Ji H. Next-generation DNA sequencing. *Nature Biotechnology* 2008; **26**(10):1135–1145.
2. Benson DA, Karsch-Mizrachi I, Lipman DJ, Ostell J, Sayers EW, GenBank. *Nucleic Acids Research* 2010; **38**(Database):D46–51. DOI: 10.1093/nar/gkp1024.
3. Smith TF, Waterman MS. Identification of common molecular subsequences. *Journal of Molecular Biology* 1981; **147**(1):195–197.
4. Altschul SF, Gish W, Miller W, Myers EW, Lipman DJ. Basic local alignment search tool. *Journal of Molecular Biology* 1990; **215**(3):403–410.
5. Pearson WR, Lipman DJ. Improved tools for biological sequence comparison. *Proceedings National Academy of Sciences of the United States of America* 1988; **85**(8):2444–2448.
6. de O Sandes EF, de Melo ACMA. Smith-Waterman alignment of huge sequences with GPU in linear space. *IEEE International Parallel Distributed Processing Symposium. IPDPS*, Anchorage, Alaska, USA, May 2011; 1199–1211, DOI: 10.1109/IPDPS.2011.114.
7. Hamidouche K, Mendonca FM, Falcou J, Etiemble D. Parallel biological sequence comparison on heterogeneous high performance computing platforms with BSP++. *23rd International Symposium on Computer Architecture and High Performance Computing*, SBAC-PAD, Vitoria, Espirito Santo, Brazil, October 2011; 136 –143, DOI: 10.1109/SBAC-PAD.2011.16.
8. Kurtz S, Phillippy A, Delcher A, Smoot M, Shumway M, Antonescu C, Salzberg S. Versatile and open software for comparing large genomes. *Genome Biology* 2004; **5**(2):R12.
9. Schatz M, Trapnell C, Delcher A, Varshney A. High-throughput sequence alignment using graphics processing units. *BMC Bioinformatics* 2007; **8**(1):474.
10. Encarnação G, Sebastião N, Roma N. Advantages and GPU implementation of high-performance indexed DNA search based on suffix arrays. *International Conference on High Performance Computing and Simulation. HPCS 2011*, Istanbul, Turkey, July 2011; 49 –55, DOI: 10.1109/HPCSim.2011.5999806.
11. Rognes T, Seeberg E. Six-fold speed-up of Smith-Waterman sequence database searches using parallel processing on common microprocessors. *Bioinformatics* 2000; **16**(8):699–706.
12. Farrar M. Striped Smith-Waterman speeds database searches six times over other SIMD implementations. *Bioinformatics* 2007; **23**(2):156–161.
13. Liu Y, Schmidt B, Maskell D. CUDASW++2.0: enhanced Smith-Waterman protein database search on CUDA-enabled GPUs based on SIMT and virtualized SIMD abstractions. *BMC Research Notes* 2010; **3**(1):93. DOI: 10.1186/1756-0500-3-93.
14. Langmead B, Trapnell C, Pop M, Salzberg S. Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. *Genome Biology* 2009; **10**(3):R25.
15. Weiner P. Linear pattern matching algorithms. *Proceedings 14th Annual Symposium on Switching and Automata Theory. SWAT '08.*, Iowa City, Iowa, USA, October 1973; 1–11, DOI: 10.1109/SWAT.1973.13.
16. Ukkonen E. On-line construction of suffix trees. *Algorithmica* 1995; **14**(3):249–260. DOI: 10.1007/BF01206331.
17. Gusfield D. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press: Cambridge, New York, USA, 1997.
18. Manber U, Myers G. Suffix arrays: a new method for on-line string searches. In *Proceedings First Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '90. Society for Industrial and Applied Mathematics: Philadelphia, PA, USA, 1990; 319–327.
19. Knuth DE. *The Art of Computer Programming. vol. 3: Sorting and searching*. Addison-Wesley Pub. Co: Reading, Mass, 1998.
20. Nickolls J, Dally WJ. The GPU computing era. *IEEE Micro* 2010; **30**(2):56–69. DOI: 10.1109/MM.2010.41.
21. Abouelhoda MI, Kurtz S, Ohlebusch E. Replacing suffix trees with enhanced suffix arrays. *Journal of Discrete Algorithms* 2004; **2**(1):53 –86. DOI: 10.1016/S1570-8667(03)00065-0.
22. Liu Y, Maskell D, Schmidt B. CUDASW++: optimizing Smith-Waterman sequence database searches for CUDA-enabled graphics processing units. *BMC Research Notes* 2009; **2**(1):73.

Copyright © 2012 John Wiley & Sons, Ltd.