Parallelization of DNA alignment algorithms using GPUs

Gustavo Encarnação
INESC-ID / IST-TU Lisbon
Portugal
gpfe@sips.inesc-id.pt

ABSTRACT

Since the discovery of Deoxyribonucleic Acid (DNA) significant technological advances were made, leading to very large amounts of data gathered for analysis. The tools for this analysis however have advanced at a slower pace and have become one of the limiting factors of new discoveries in this field of research.

Recently, from the 3D game market, a new generation of hardware has emerged. This hardware known as Graphics Processing Units (GPUs) has the capability for general purpose computing. With this new hardware high performance computing has become possible on cheap and readily available hardware which creates new opportunities to study and improve current tools and algorithms used for the study of DNA.

A study of sequence matching techniques using indexes and their adaptation to the DNA alignment problem is presented. The indexes used were suffix trees and suffix arrays. A heuristic DNA alignment algorithm which runs on the GPU was developed based on the indexes studied.

Tests conducted showed that the work lead to the creation of an algorithm that is capable of competing with current day techniques in both performance and quality of results.

Keywords: GPGPU, Indexed Search, Approximate Search, DNA Alignment, Bioinformatics

1. INTRODUCTION

Since the model for DNA was initially proposed in the early 20th century great advances have been made in the area, including the unveiling of new areas of scientific research such as genetics.
Nowadays, the ability to completely sequence the DNA of any organism is real and the methods for such task are constantly improving. As a result of these new and faster methods there are large amounts of data being generated [1]. The study and analysis of this data is mainly done by comparing very large numbers of DNA sequences, which is only possible through the use of computers and specialized algorithms. These algorithms however have not improved at the same rate as the sequencing platforms have.

The developed work is part of the growing focus on parallel processing as a means to solve time consuming problems in the shortest amount of time possible. It is also part of the growing interest in GPUs as platforms for efficient and affordable parallel processing.

One of the most used algorithms to extract information from biological sequences is the Smith-Waterman (S-W) algorithm. It is capable of finding the optimal local alignment between any two sequences with sizes n and m in O(nm) runtime. For large sequences, such as the human genome (with about 3 × 109 base pairs), this runtime can be extremely large which led to the development of other suboptimal algorithms that typically start by finding an exact match between small sub-sequences of the query and the reference sequences (a seed). Afterwards, if such seed fulfills a given set of conditions, the alignment is extended to the sides. To accelerate the search of the initial match, many of these heuristics make use of a pre-prepared index of the reference sequence. Such index can be built using different data structures, such as the hash tables of q-mers (substrings of pre-defined length q) used in BLAST [3] or the suffix trees used in MUMmer [4].

Even though these index structures significantly accelerate the search for the initial match, these algorithms still present a high computational demand, mainly due to the large amount of data they must process. As such, several parallelization techniques have been considered to accelerate these algorithms [5]. On the other hand, with the recent developments on high-performance computer architectures, a vast set of inexpensive parallel processing structures has emerged, such as multi-core CPUs with 4 or even more homogeneous processors or Graphics Processing Units (GPUs) with general purpose computation capabilities that 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.

Some previous work, focused on the parallelization of the alignment algorithms in the several platforms has already been presented [6,7,8]. The algorithm proposed by Farrar et al. [7] 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 at the algorithm level. Other programs, like MUMmer [4] and Bowtie [9], are also targeted at the CPU but mainly take advantage of data-level parallelism. While MUMmer [4] uses a suffix tree as its index data structure, Bowtie [9] uses the Burrows–Wheeler transform to reduce the memory footprint of its index structure.

One common observation that has been retained is that the great number of processors that are present in the GPU devices make them ideal for computationally intensive applications, like bioinformatics algorithms. Nevertheless, the inherent GPU restrictions on synchronous execution and on memory access often impose a significant constraint on the adopted programming model and limit the type of algorithms that can be efficiently parallelized on these devices. Nevertheless, independently of the set of constraints 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.

The work presented in this paper focuses on two sequence indexing techniques, their use to solve sequence matching problems and how they can be adapted to be used in a GPU. From these two indexes a heuristic algorithm for DNA alignment was developed and compared against common DNA alignment tools.

2. INDEXED SEARCH

To accelerate string matching problems, it is common to create an index of the reference string and then use it to accelerate the match with a given query 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 capable of finding the match location of a given query of size n in linear time (O(n)) is highly desirable. The well known MUMmer framework [4] makes use of an index with such characteristics based on suffix trees. It uses this index to determine the Maximal Unique Matching subsequences (MUMs) between any two sequences.

2.1. Suffix Trees

A suffix tree is a data structure that represents all the suffixes of a given sequence [10, 11]. It is composed of a root node, several internal nodes and leafs. 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. An n character 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 equals the number of leafs below that node.

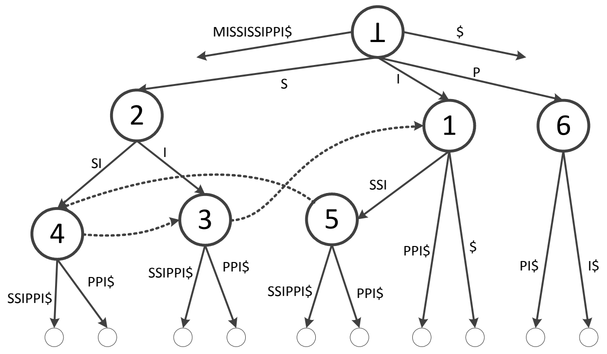

Figure 1. Example of a suffix tree for the string "mississippi", including the suffix links (dashed lines)

By using suffix trees, it is possible to discover whether a particular query string exists within a larger reference string in O(n), where n is the size of the query string. 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 leafs nodes from that point onwards, it is possible to exactly know how many times and where the query occurs in the reference 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 to suffix trees. suffix arrays [12] are regarded as a more space efficient structure typically requiring three to five times less space. This structure (illustrated in Fig. 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 come 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 store 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.

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

Figure 2. Suffix array for the string "mississippi".

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. 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 (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. Since the values in the suffix array are not strings but indexes (pointer) of a string, it is not the values themselves that must be compared but the text they point to. Hence, when comparing two elements of the suffix array, not only the character they point to must be compared but, if they are equal, the subsequent characters must also be compared. The overall performance of the alignment function will reflect the efficiency of the search algorithm used in the array.

By using a binary search algorithm the array is repeatedly divided in half until the desired element is reached. Hence, suffix arrays solve the string matching problem with an $O(n \log(m))$ complexity, where n is the length of the query and m the length of the reference.

3. GPU PROGRAMMING MODEL

The basic building blocks in NVIDIA's Fermi architecture [13] are the Streaming Multiprocessors (SMs). Each SM contains 32 processing cores 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 that they follow a Single Instruction Multiple Thread (SIMT) paradigm, which means that 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 different branch, 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 also offers a high capacity main memory with a high bandwidth access (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 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 provides an 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.

4. EXACT SEQUENCE MATCHING

4.1. Suffix Trees

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

The suffix tree is constructed from the reference string and is afterwards transformed into a flattened tree consisting of an array of edges. Each node is represented in the flattened tree by its set of outgoing edges, where each edge contains: 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 3 integers. However, to allow a perfect alignment of the memory accesses, the representation of a single edge is padded to hold exactly 4 integers. Furthermore, each node always contemplates space for its 4 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 due to the inability to coalesce them.

Since the suffix tree only includes references to the original sequence, besides transferring the flattened tree to the GPU it is also necessary to transfer the original reference sequence. To save space and to optimize certain parts of the alignment 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 two bits.

The original DNA query sequences are stored in the GPU global memory in their string format. However, to maximally exploit the available memory bandwidth, each set of 16 nucleotides is packed into a single 32-bit integer and the symbols of the different query sequences are interleaved. Due to the particular way 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 later characters are mapped in the highest order bits. Such *reverse* order is preferred since it allows obtaining the various characters by using a shift right instruction followed by a binary AND always with the same mask ('11').

Due to 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 particularity 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 alignment algorithm, is executed by each thread in the GPU. 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 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 ’test character’ will be at position `tree[0 + test char]`.

Once the query buffer is filled and there is an edge to follow, the alignment 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 there is a mismatch, 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 alignment 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 to 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 suffix trees, in many applications their smaller size leads to similar or even better performance levels [14, 15], due to the attainable better cache performances.

The suffix array is a one-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 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 alignment 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 memory accesses become sequential until it becomes necessary to re-pick the suffix. Therefore, by transforming 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 each of the GPU threads, consists of two nested loops. The first loop is executed until all possible search options have been exhausted. Since 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 done 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, since it is always aligned. Nevertheless, a special care must be taken when reading the suffix, since it might not be aligned and thus the higher bits of the memory position will be invalid.

Before the comparison cycle begins, it is necessary to assure that the query buffer and the suffix buffer hold the same number of packed characters, since 16 symbols are compared at once.

The inner loop, is the comparison cycle ('==') which runs while the sequences are equal and there are more symbols to be compared in the sequences. When the algorithm enters 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 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 parallel execution in the GPU.

5. APPROXIMATE SEQUENCE MATCHING

Approximate matching is closely related to exact matching. While in the later the sequence must matched completely, in the former a partial match or a match containing small differences is allowed.

As the problem becomes more complex the solutions used for exact matching are no longer suitable. The reference sequence index however, can still be applied to the problem.

Matching sequences with differences is a complex problem to address using indexed reference sequences. However, finding partial matches can be achieved with small changes to the search algorithms.

5.1 Suffix Trees

Suffix trees nodes represent a subsequence of the reference sequence the tree indexes. Each of this nodes possesses a connection to another node that represents the largest suffix of the subsequence represented (the sequence minus the first character). This connection is called the suffix link.

Using the suffix link it is possible to continue the matching effort even after a mismatch is found. When a mismatch is found, the search continues from the destination node of the suffix link. If the algorithm cannot match the sequence aNc it will try to match the sequence Nc by removing the first character.

Using the above method, a partial match is found for every mismatch encountered.

5.2 Suffix Arrays

Exact matching using suffix arrays uses binary search. Since suffix arrays do not possess any kind of additional information relating the array entries between them, performing approximate matching using these types of indexes relies on re-starting the search algorithm for each mismatch found.

One alternative would be to use auxiliary structures to hold the necessary information to reduce the amount of steps the binary search algorithm would have to do.

The simplest approach is to generate a hash table, using as hash key the first characters of the query sequence. Every entry in the table will hold the array index of the first element beginning with hash key. When a mismatch is found and the first character of the sequence is removed, the new hash value is computed and the appropriate entry (and the next one) in the hash table retrieved. This allows the algorithm to immediately reduce the search space to only the suffix array entries that begin with the same subsequence as the query.

The size of the hash key determines how large the search space for the binary search algorithm will be. A long hash key allows for a faster search at the cost of more memory and longer times to copy the auxiliary structure from the host memory to the device.

6. DNA ALIGNMENT

The optimal local alignment algorithm presented by Smith and Waterman is a dynamic programming algorithm with a runtime complexity of O(nm), where n and m represent the sizes of the sequences being aligned. However, with the huge amount of sequencing data that is currently available, the runtime of this algorithm quickly becomes a bottleneck. A way of using the Smith-Waterman algorithm for DNA sequence alignment is to reduce the size of the sequences being aligned. Therefore, other heuristic algorithms, such as BLAST [3], have been proposed to significantly reduce the alignment time.

The heuristic algorithms typically operate in three phases: i) an initial approximate match phase; ii) a filtering phase of the potential alignment locations; and iii) a refinement of the obtained score by considering gaps in the alignment.

6.1 Filtering the seeds

The initial approximate match phase might return a very high number of results (seeds) and these will be passed to the slower gapped alignment phase. It is important that the results computed in the first phase be filtered so that only the relevant ones be considered in the final phase.

One way of filtering the results is to merge seeds close together into a single seed. This will allow the reduction of the total number of seeds while at the same time preserving all the information gathered from the matching phase.

To consider which seeds to merge it is important that they are sorted by starting location so as not to induce the algorithm in error. If one query sequence yields two different partial matches it is crucial that the order by which these appear in the query sequence and in the reference sequence be the same.

To sort the seeds, an insertion-sort algorithm [16] is used with binary search trees. Since the smaller the tree the faster the algorithm is, multiple trees are used and the seeds are assigned a tree based on the seed’s starting position.

Also, to ensure the trees are as balanced as possible an initial dummy node is inserted into the tree, representing the average value the tree can hold. If a tree is assigned to hold the seeds that begin in positions 0 through 1000 then the dummy node will have the value 500.

Once the seeds have been sorted, the trees are traversed and converted into a linked list of seeds. The final step of the filtering algorithm is to walk the list and merge any seeds that are not spaced apart more than a specified number of positions.

6.2 Gapped alignment

Once the seeds have been filtered they are passed to the final phase that consists of a modified version of the Smith-Waterman algorithm.

The version of the Smith-Waterman algorithm used, is based on CUDA SW++[2]. The algorithm was modified and optimized for DNA alignment, using a simplified scoring matrix (holding only four values: match, mismatch, gap open, gap extend) and using 2-bit encoding for the query and reference sequences.

All the subsequences sent to the Smith-Waterman algorithm have the same length. This length is specified by a parameter. After the seeds have been filtered a location is created for each seed. This location is created by extracting a subsequence of the query that contains the seed. The exact start position of this subsequence depends on the specified length for the location, the start position of the seed and is restrained to positions that are multiples of 16.

The locations are aligned to 16 characters to ensure that, using 2-bit encoding, all the elements are fully used (the elements are integers which hold 16 bases).

7. RESULTS

The tests conducted with the algorithms developed were done using real DNA data. The reference sequence was extracted from the Homo Sapiens Chromosome 1 (NT_167186.1). The query sequences were extracted from the Homo Sapiens Chromosome 1 (NT_167186.1) and from the Mus Musculus Chromosome 1 (NT_039170.7). The query sequences were grouped into sets of 1024 to 4194304 queries where half the sequences came from Homo Sapiens and half from Mus Musculus.

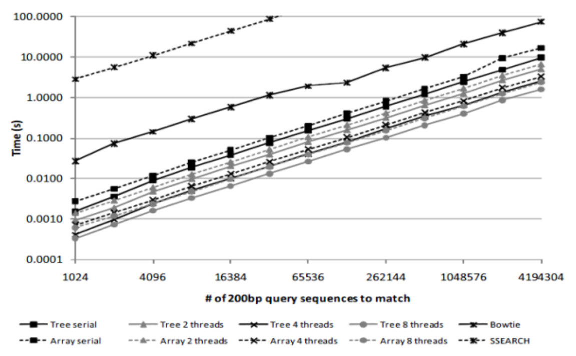

Figure 3. Performance evaluation of suffix tree and suffix array index based search algorithms in multi-core CPUs.

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

To evaluate the performance provided by the indexing methods on exact matching problems the algorithms developed were compared in a homogeneous multi-core CPU by making use of POSIX threads.

From the obtained results (see Fig. 3) it can be observed that 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 was already observed in [14, 15], and is mainly due to a more efficient usage of the cache memory by the suffix array, which is achieved due to its smaller and more regular structure. Furthermore, by comparing the execution time results with the Bowtie and SSEARCH35 programs, it is possible to observe that the implemented suffix tree and suffix array algorithms are significantly faster, thus plenty justifying their adoption whenever high performance DNA alignment is required.

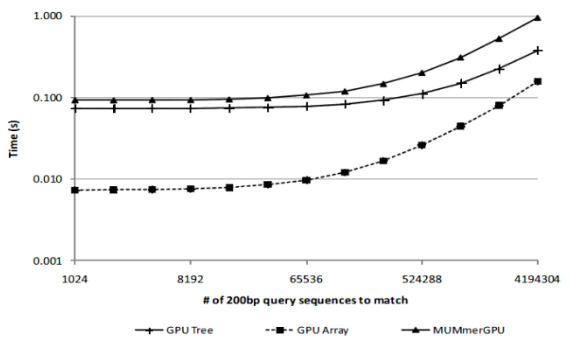

Figure 4. Performance evaluation of the considered index based search algorithms in the GPU.

Then, the performance of the conceived concurrent algorithms was assessed in a GPU platform, namely the NVIDIA GeForce GTX 580. The obtained results are presented in Fig. 4. This chart also includes a comparison with another DNA alignment framework, based on suffix trees, executed in the GPU: MUMmerGPU [5]. These 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 Fig. 5, the data input time is very significant in all these index-based search algorithms, since 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 to the corresponding CPU implementations. However, for a larger number of query sequences (commonly adopted by this application domain), the GPU implementations offer a significantly better performance, with speedup values as high as 85 for the suffix array implementation and 25 for the suffix tree implementation.

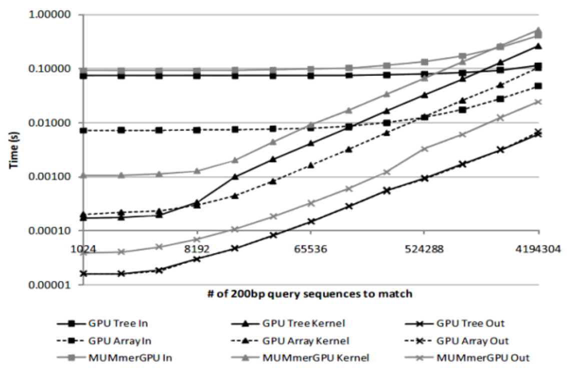

Figure 5. Communication and kernel execution times for the implementations in the GeForce GTX580 GPU.

These observations reveal that contrary to what is stated by the asymptotical complexity analysis of these algorithms (and unlike the obtained CPU performance results), the GPU implementation clearly favors the suffix array index structure. The justification for this fact is not only the more regular execution flow of this algorithm and its more efficient use of the cache memory, but is also the fact that 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.

Since the algorithm developed for DNA alignment would be an heuristic algorithm with the first phase being approximate matching, the two possible indexes were compared when solving the approximate matching problem.

The results show, that suffix trees with the use of the suffix link have the best performance when large number of query sequences are matched. This is due to the fact that, unlike the binary search algorithm used by suffix arrays, the search using suffix trees does not need to start from the beginning for every mismatch found. Even when using auxiliary structures, the suffix array cannot outperform suffix trees.

Finally, the implemented DNA alignment solution was compared with other DNA alignment tools (BLAST). The execution time was measured as well as the error (difference between the score reported by the algorithm and the score calculated by the Smith-Waterman algorithm) for BLAST.

Table 1. Comparison between BLAST and the GPU algorithm using a reference sequence with 1.000.000 BP.

Alignment tool	BLAST				GPU algorithm		
Seed size	5	8	10	default	5	8	10
Time	416.93	220.94	140.06	3.62	36.27	3.09	2.58
# alignments	1000	986	890	226	869	973	1000
Sum of errors (BLAST subset)	1920	1934	1934	4917	2083	2808	5473
Sum of errors (reported)	25817	32062	47904	4917	35466	64847	84068

The results in Table 1 show that the GPU algorithm executed with a seed of size 8 has a smaller error than BLAST with the default parameters, a smaller execution time (considering the larger reference) and a much higher number of query sequences for which an alignment is reported.

The presented results allow to conclude that the new algorithm is capable of outperforming BLAST both in terms of speed and quality of results. Furthermore, the developed algorithm also allows the user to select the parameters to exploit the balance between quality of results and execution time, thus being flexible enough to be used in a wide range of scenarios.

8. CONCLUSIONS

After comparing the two index data structures especially suited for accelerating DNA sequence alignment in bioinformatics it was observed that the optimal index varied with the problem and the platform used. While exact matching problems can be efficiently solved by suffix arrays, approximate matching problems cannot. As such, in exact matching suffix arrays based algorithms designed to run on GPUs have the advantage of the index structure being smaller and requiring less time to copy to the device memory. On the other hand, the solution to approximate matching is more complex and requires more information which is readily available for suffix trees but not for suffix arrays.

From the results obtained with the heuristic algorithm for DNA alignment it was observed that GPUs present a practical and efficient platform for high performance computing and algorithms optimized for these platforms can obtain better perform than current options for the same problem that run on the CPU.

ACKNOWLEDGMENTS

This research was performed in the scope of project “HELIX: Heterogeneous Multi-Core Architecture for Biological Sequence Analysis”, funded by the Portuguese Foundation for Science and Technology (FCT) with reference PTDC/EEA-ELC/113999/2009, and project “TAGS: The power of the short - Tools and Algorithms for next Generation Sequencing applications”, funded by FCT with reference PTDC/EIA-EIA/112283/2009.

REFERENCES

[1] J. Shendure and H. Ji, “Next-generation DNA sequencing,”Nature Biotechnology, vol. 26, no. 10, pp. 1135–1145, October 2008.

[2] Y. Liu, B. Schmidt, and D. 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, no. 1, p. 93, 2010.

[3] S. Altschul, W. Gish, W. Miller, E. Myers, and D. Lipman, “Basic local alignment search tool,” Journal of Molecular Biology, vol. 215, no. 3, pp. 403–410, 1990.

[4] S. Kurtz, A. Phillippy, A. Delcher, M. Smoot, M. Shumway, C. Antonescu, and S. Salzberg, “Versatile and open software for comparing large genomes,” Genome Biology, vol. 5, no. 2, p. R12, 2004.

[5] M. Schatz, C. Trapnell, A. Delcher, and A. Varshney, “Highthroughput sequence alignment using Graphics Processing Units,” BMC Bioinformatics, vol. 8, no. 1, p. 474, 2007.

[6] T. Rognes and E. Seeberg, “Six-fold speed-up of Smith-Waterman sequence database searches using parallel processing on common microprocessors,”Bioinformatics, vol. 16, no. 8, pp. 699–706, 2000.

[7] M. Farrar, “Striped Smith-Waterman speeds database searches six times over other SIMD implementations,” Bioinformatics, vol. 23, no. 2, pp. 156–161, 2007.

[8] Y. Liu, B. Schmidt, and D. Maskell, “CUDASW++2.0: enhanced Smith-Waterman protein database search on CUDAenabled GPUs based on SIMT and virtualized SIMDabstractions,” BMC Research Notes, vol. 3, no. 1, p. 93, 2010.

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

[10] P.Weiner, “Linear pattern matching algorithms,” in Proceedings 14th Annual Symposium on Switching and Automata Theory. SWAT ’08., October 1973, pp. 1–11.

[11] E. Ukkonen, “On-line construction of suffix trees,” Algorithmica, vol. 14, no. 3, pp. 249–260, September 1995.

[12] U. Manber and G. Myers, “Suffix arrays: a new method for on-line string searches,” in Proceedings First annual ACMSIAM Symposium on Discrete algorithms, ser. SODA ’90, Philadelphia, PA, USA, 1990, pp. 319–327.

[13] J. Nickolls and W. Dally, “The GPU Computing Era,” IEEE Micro, vol. 30, no. 2, pp. 56–69, March 2010.

[14] M. I. Abouelhoda, S. Kurtz, and E. Ohlebusch, “Replacing suffix trees with enhanced suffix arrays,” Journal of Discrete Algorithms, vol. 2, no. 1, pp. 53 – 86, 2004.

[15] G. Navarro and R. Baeza-yates, “A hybrid indexing method for approximate string matching,” Journal of Discrete Algorithms, vol. 1, p. 2000, 2000.

[16] N. Sebastião, G. Encarnação, and N. Roma, “A New Seed Filtering Approach for Heuristic DNA Alignment using GPUs,” INESC-ID, Tech. Rep. 44/2011, October 2011.