DOI 10.12694/scpe.v16i2.1085
Scalable Computing: Practice and Experience ISSN 1895-1767
Volume 16, Number 2, pp. 121–137. http://www.scpe.org

MULTIPLE STRING MATCHING ON A GPU USING CUDA

CHARALAMPOS S. KOUZINOPOULOS*, PANAGIOTIS D. MICHAILIDIS† AND KONSTANTINOS G. MARGARITIS‡

Abstract. Multiple pattern matching algorithms are used to locate the occurrences of patterns from a finite pattern set in a large input string. Aho-Corasick, Set Horspool, Set Backward Oracle Matching, Wu-Manber and SOG, five of the most well known algorithms for multiple matching require an increased computing power, particularly in cases where large-size datasets must be processed, as is common in computational biology applications. Over the past years, Graphics Processing Units (GPUs) have evolved to powerful parallel processors outperforming CPUs in scientific applications. This paper evaluates the speedup of the basic parallel strategy and the different optimization strategies for parallelization of Aho-Corasick, Set Horspool, Set Backward Oracle Matching, Wu-Manber and SOG algorithms on a GPU.

Key words: multiple pattern matching, parallel computing, many-core computing, GPU, CUDA

AMS subject classifications. 68W32, 68W10, 68N19

1. Introduction. Multiple pattern matching is an important kernel in computer science and it occurs in many security and network applications including information retrieval, web filtering, intrusion detection systems, virus scanners and spam filters [3, 30]. More specifically, many well known tools utilize multiple pattern matching such as: Snort [36] to perform intrusion detection and GNU grep, fgrep and egrep programs support multi-pattern matching through the -f option [12, 8, 43, 44]. Moreover, in recent years there is an interest in string matching problems as a powerful tool to locate nucleotide or Amino Acid sequence patterns in biological sequence databases. The multiple pattern matching problem can be defined as [30]: Given an input string $T = t_0 t_1 \dots t_{n-1}$ of size n and a finite set of d patterns $P = p^0, p^1, \dots, p^{d-1}$, where each p^r is a string $p^r = p_0^r p_1^r \dots p_{m-1}^r$ of size m over a finite character set Σ and the total size of all patterns is denoted as $|P|$, the task is to find all occurrences of any of the patterns in the input string. More formally, for each p^r find all i where $0 \leq i < n - m + 1$ such that for all j where $0 \leq j < m$ it holds that $t_i = p_j^r$.

Numerous sequential algorithms to this problem have been proposed [30]. In general, most of the multiple string matching algorithms consist of two phases. The first phase is a preprocessing of the set of patterns and the second phase is searching of multiple patterns in the input string. During the preprocessing phase a data structure X is constructed, X is usually proportional to the length of the pattern set and its details vary in different algorithms. The search phase uses the data structure X and it tries to quickly determine if any pattern occurs in the input string. A naive solution to the multiple string matching problem is to perform d separate searches in the input string with a single string matching algorithm [30] leading to a worst-case complexity of $\mathcal{O}(|P|)$ for the preprocessing phase and $\mathcal{O}(n|P|)$ for the search phase. While frequently used in the past, this technique is not efficient, especially when a large pattern set is involved. There are some efficient and practical algorithms have been presented for multiple string matching in terms of preprocessing and searching time for different types of data, such as: Aho-Corasick [1], Set Horspool [30], Set Backward Oracle Matching [30], Wu-Manber [44] and SOG [34].

As multiple pattern matching algorithms are often used to process huge amounts of data with a size that is increasing almost exponentially in time, their sequential execution on conventional computer systems is often impractical. Multiple pattern matching algorithms have nearly all the characteristics that make them suitable for parallel execution on graphics processing units (GPUs); they involve sequential accesses to the memory, parallelism can easily be exposed through data partitioning while the device on-chip shared memory and caches can be utilized to further decrease their execution time. The use of GPUs to accelerate the performance of multiple pattern matching algorithms has been studied in the past [15, 14, 23, 24, 32, 38, 39, 40, 42, 45]. Similar parallelization efforts for multiple string matching have been presented for multi-core and cluster platforms using different programming frameworks (i.e. POSIX threads, OpenMP and MPI) [4, 21, 33, 37]. Most of these research studies have focused on parallelization of the Aho-Corasick and Wu-Manber algorithms.

*CERN, (charalampos.kouzinopoulos@cern.ch).
†Department of Balkan, Slavic and Oriental Studies, University of Macedonia, (pmichailidis@uom.gr).
‡Department of Applied Informatics, University of Macedonia (kmarg@uom.gr).

This paper focuses on the parallelization of the Aho-Corasick [1], Set Horspool [30], Set Backward Oracle Matching [30], Wu-Manber [44] and SOG [34] multiple pattern matching algorithms through their implementation on a GPU using the Compute Unified Device Architecture (CUDA) programming model. These multiple string matching algorithms were chosen since they are practical and are frequently encountered in other research papers. The performance of the parallel implementations is evaluated after applying different optimization techniques and their execution time is compared to the time of the sequential implementations. This paper differ from previous research studies in that the Set Horspool, Set Backward Oracle Matching and SOG algorithms have never been implemented before on a GPU using the CUDA API (Application Programming Interface).

The rest of the paper is organized as follows: in Section 2, we give an overview of the most significant sequential algorithms for multiple pattern matching, the GPU architecture and the CUDA programming model and the state-of-the-art multiple pattern matching algorithms on GPU. In Section 3, we present the basic parallel implementation and optimization techniques of multiple pattern matching algorithms on the GPU architecture. In Section 4 we show the performance results of the proposed implementations. Finally, in Section 5, we draw conclusions.

2. Literature review.

2.1. Sequential multiple pattern matching algorithms. The aim of all multiple pattern matching algorithms is scan the input string in a single pass to locate all occurrences of the patterns. These algorithms are often based on single string matching algorithms with some of their functions generalized to process multiple patterns simultaneously during the preprocessing phase. Based on the way the multiple patterns are represented and the search is performed, the algorithms can generally be classified in to one of the four following approaches.

- **Prefix algorithms** The prefix searching algorithms use a *trie* to store the patterns, a data structure where each node represents a prefix u of one of the patterns. For a given position i of the input string, the algorithms traverse the trie looking for the longest possible suffix u of $t_0...t_i$ that is a prefix of one of the patterns. One of the most well known prefix multiple pattern matching algorithms is Aho-Corasick [1], an efficient algorithm based on Knuth-Morris-Pratt algorithm [19] that preprocesses the pattern in time linear in $|P|$ and searches the input string in time linear in n in the worst case. Multiple Shift-And, a bit-parallel algorithm generalization of the Shift-And algorithm for multiple pattern matching was introduced in [30] but is only useful for a small size of $|P|$ since the pattern set must fit in a few computer words.
- **Suffix algorithms** The suffix algorithms store the patterns backwards in a suffix trie, a rooted directed tree that represents the suffixes of all patterns. At each position i of the input string the algorithms compute the longest suffix u of the input string that is a suffix of one of the patterns. Commentz-Walter [8] combines a suffix trie with the *good suffix* and bad character shift functions of the Boyer-Moore algorithm [6]. A simpler variant of Commentz-Walter is Set Horspool [30], an extension of the Horspool algorithm [13] that uses only the bad character shift function. Suffix searching is generally considered to be more efficient than prefix searching since on average more input string positions are skipped following each mismatch.
- **Factor algorithms** The factor searching algorithms build a *factor oracle*, a trie with additional transitions that can recognize any substring (or factor) of the patterns. Dawg-Match [10] and MultiBDM [11] were the first two factor algorithms, algorithms complicated and with a poor performance in practice [30]. The Set Backward Oracle Matching and the Set Backward Dawg Matching algorithms [30] are natural extensions of the Backward Oracle Matching [2] and the Backward Dawg Matching [9] algorithms respectively for multiple pattern matching.
- **Hashing algorithms** The algorithms following this approach use hashing to reduce their memory footprint, usually in conjunction with other techniques. Wu-Manber [44] is one such algorithm that is based on the Horspool algorithm. It reads the input string in blocks to effectively increase the size of the alphabet and then applies a hashing technique to reduce the necessary memory space. Zhou et al. [46] proposed an algorithm called MDH, a variant of Wu-Manber for large-scale pattern sets. Kim and Kim introduced in [18] a multiple pattern matching algorithm that also takes the hashing approach. The Salmela-Tarhio-Kytöjoki [34] variants of the Horspool, Shift-Or [5] and BNDM [29] algorithms can locate *candidate* matches by excluding positions of the input string that do not match

any of the patterns. They combine hashing and a technique called q-grams to increase the alphabet size, similar to the method used by Wu-Manber.

2.1.1. Aho-Corasick algorithm. Aho-Corasick is an extension of the Knuth-Morris-Pratt algorithm [19] for a set of patterns P. It uses a deterministic finite state pattern matching machine; a rooted directed tree or *trie* of P with a *goto* function g and an additional *supply* function *Supply*. The *goto* function maps a pair consisting of an existing state q and a symbol character into the next state. It is a generalization of the *next* table or the *success* link of the Knuth-Morris-Pratt algorithm [19] for a set of patterns where a parent state can lead to its child states by σ where σ is a matching character. The *supply* function of Aho-Corasick is based on the *supply* function of the Knuth-Morris-Pratt algorithm [19]. It is used to visit a previous state of the automaton when there is no transition from the current state to a child state via the *goto* function. The *goto* function and the *supply* function are constructed during the preprocessing phase. In the searching phase of algorithm, each character in the input string is scanned from left to right with the trie that tracks partial matched patterns through state transitions. If one of terminal states is visited indicates that an occurrence of one of the pattern strings was found. The trie of P can then be built for all m patterns in $\mathcal{O}(|\Sigma|m^2)$ time, with a total size of $\mathcal{O}(|\Sigma|m^2)$. The time to pass through a transition of the *goto* function is $\mathcal{O}(1)$ in the worst and average case, while the search phase has a cost of $\mathcal{O}(n)$ in the worst and average case.

2.1.2. Set Horspool algorithm. The Set Horspool algorithm combines a deterministic finite state pattern matching machine with the *shift* function of the Horspool algorithm [13] to search for the occurrence of multiple patterns in the input string in sublinear time on average. The pattern matching machine used is a trie with a *goto* function g, created from each pattern $p^r \in P$ in reverse. The *goto* function and the *shift* function are computed during the preprocessing phase. The search for the occurrences of the patterns is then performed backwards similar to Horspool. When a mismatch or a complete match occurs, a number of input string positions can be safely skipped based on the bad character shift of the Horspool algorithm generalized for a set of patterns. The construction of the trie and the shift function requires $\mathcal{O}(|\Sigma||P|)$ time and space while the search phase of the algorithm is $\mathcal{O}(nm)$ worst case time or sublinear on average.

2.1.3. Set Backward Oracle Matching algorithm. The Set Backward Oracle Matching algorithm [2] extends the Backward Oracle Matching string matching algorithm to search for the occurrence of multiple patterns in the input string in sublinear time on average. It uses a *factor oracle*, a deterministic acyclic automaton created from each pattern $p^r \in P$ in reverse that is based on the notion of weak factor recognition. The automaton consists of a *goto* function g and at most m^2 additional external transitions such that at least any factor of a pattern can be recognized, similar to the *factor oracle* of Backward Oracle Matching. The *goto* function is constructed during the preprocessing phase from the set of the reversed patterns, similar to the automaton of the Set Horspool algorithm. During the search phase, the algorithm reads backwards with the factor oracle. If the oracle fails to recognize a factor at a given position, we can shift the pattern beyond that position. The oracle is created during the preprocessing phase in $\mathcal{O}(|\Sigma||P|)$ for all d patterns of the pattern set using a size of $\mathcal{O}(|\Sigma||P|)$. The search phase complexity of the algorithm is $\mathcal{O}(n|P|)$ worst case time or sublinear in average time.

2.1.4. Wu-Manber algorithm. Wu-Manber is a generalization of the Horspool algorithm [13] for multiple pattern matching. To improve the efficiency of the algorithm, Wu-Manber considers the characters of the patterns and the input string as blocks of size B instead of single characters. As recommended in [44], a good value for B is $log_\Sigma 2|P|$ although usually B could be equal to 2 for a small pattern set size or to 3 for a large pattern set size. During the preprocessing phase, three tables are built from the patterns, the *SHIFT*, *HASH* and *PREFIX* tables. *SHIFT* is the equivalent of the bad character shift of the Horspool algorithm and is used to determine the number of characters that can be safely skipped based on the previous B characters on each text position. The *PREFIX* table stores a hashed value of the B-characters prefix of each pattern while the HASH table contains a list of all patterns with the same prefix. During the searching phase, the algorithm is searching for the occurrences of all patterns in the input text with the assistant of the three tables that have been created by the previous state. Firstly, a hash value (h) for the block of B characters is calculated into the current search window and the shift value for that is checked ($SHIFT[h]$). If the shift value is greater than zero, then the current search window is shifted by $SHIFT[h]$ positions or else there is a candidate match and the

hashed value of the previous B characters of the input string is then compared with the hashed values stored at the *PREFIX* table to determine if an exact match exists. For the experiments of this paper, the algorithm was implemented with a block size of $B = 2$ and $B = 3$. To calculate the values of the *SHIFT*, *HASH* and *PREFIX* tables during the preprocessing phase, the algorithm requires an $\mathcal{O}(|P|)$ time. The worst case searching time of Wu-Manber is given in [7] as $\mathcal{O}(n|P|\log_{|\Sigma|}|P|)$. In [28] the lower bound for the average time complexity of exact multiple pattern matching algorithms is given as $\Omega(n/m \cdot \log_{|\Sigma|}(|P|))$ and according to [7] the searching phase of the Wu-Manber algorithm is optimal in the average case for a time complexity of $\mathcal{O}(n/m \cdot \log_{|\Sigma|}(|P|))$. In [25] the average time complexity of Wu-Manber was also estimated as $\mathcal{O}\big(n/\big[(m-B+1)\cdot(1-\frac{(m-B+1)\cdot d}{2|\Sigma|^B})\big]\big)$.

2.1.5. SOG algorithm. SOG extends the Shift-Or string matching algorithm [5] to perform multiple pattern matching in linear time on average. Similar to Shift-Or, SOG is a bit-parallel algorithm that simulates a nondeterministic automaton. Moreover, it acts as a character class filter; it constructs a generalized pattern that can simultaneously match all patterns from a finite set. The generalized pattern accepts classes of characters based on the actual position of the characters in the patterns. The searching phase of the algorithm consists of a filtering phase and a verification phase. To improve the efficiency of the verification, a two-level hashing technique is used as detailed in [27]. When a candidate match is found at a given position of the input string during the filtering phase, the patterns are verified using a combination of hashing and binary search to determine if a complete match of a pattern occurs. When the pattern set has a relatively large size, every position of the generalized pattern will accept most characters of the alphabet. In that case, false candidate matches will occur in most positions of the input string. To overcome this problem, SOG increases the alphabet size to $|\Sigma|^B$ by processing the characters of the input string and the patterns in blocks of size B, similar to the methodology used by the Wu-Manber algorithm. For the experiments of this paper, SOG was implemented using a block size of $B = 3$. The preprocessing time of the SOG algorithm is $\mathcal{O}(|P|)$ with an $\mathcal{O}(|\Sigma|^B + |P|)$ space. The combined filtering and verification phase is then $\mathcal{O}(n|P|)$ when all pattern rows have the same hash value and $\mathcal{O}(nm)$ otherwise in the worst case and linear in n in the average case.

For further details and pseudocode about the preprocessing and searching phase of the above sequential algorithms are presented in [20].

2.2. GPU architecture and CUDA programming model. Nvidia developed an unified GPU architecture that supports both graphics and general purpose computing. In general purpose computing, the GPU is viewed as a massively parallel processor array contains many processor cores. The GPU consists of an array of streaming multiprocessors (SMs). Each SM consists of a number of streaming processors (SPs) cores. Each SP core is a highly multithreaded, managing a number of concurrent threads and their state in hardware. Further, each SM has a SIMD (Single-Instruction Multiple-Data) architecture: at any given clock cycle, each streaming processor core performs the same instruction on different data. Each SM consist of four different types of on-chip memory, such as registers, shared memory, constant cache and texture cache. Constant and texture caches are both read-only memories shared by all SPs. On the other hand, the GPU has a off-chip memory, namely global memory (to which Nvidia refers as the device memory) in which has long access latency. More details for these different types of memory and GPU architecture can be found in [31].

For our numerical experimets, we take NVIDIA GTX 280 which is a compute capability 1.3 GPU from the GT200 series of NVIDIA's Tesla hardware. It has $1GB$ of GDDR3 global memory, $602MHz$ Graphics clock rate, $1.3GHz$ Processor clock tester rate and $1.1GHz$ memory clock rate. The GPU consists of 30 SMs. Each SM has 8 SPs for a total of 240 SPs, $16KB$ of on-chip shared memory, and a $16KB$ 32-bit register file. Each thread block can have a maximum of 512 threads while each SM supports up to 1024 active threads and up to 8 active blocks. Some of the related issues that affect the GPU performance such as non-coalesced memory accesses, shared memory bank conflicts, control flow divergence and occupancy are described in [20].

Nvidia's CUDA is one of the most widely adopted programming model that enables developing GPU-based applications using C programming language. A unified program C for CUDA in which consists of host code running on CPU and kernel code running on a device or GPU. The host code is a simple code C that implements some parts of an application that exhibit little or no data parallelism. The kernel code is a code using CUDA keywords that implements some parts of an application exhibit high amount of data parallelism. The kernel code implements the computation for a single thread and is usually executed on GPU by a set of parallel

threads. All parallel threads execute the same code of the kernel with different data. Threads are organized in a hierarchy of grids of thread blocks. A thread block is a set of concurrent threads that can cooperate via barrier synchronization and access to a shared memory which is only visible to the thread block. Each thread in a thread block has a unique thread ID number *threadIdx*. Thread ID can 1, 2, or 3 dimensional. Usually there can be 512 or 1024 threads per block, depending on the compute capability of the device. Thread blocks are grouped into a grid and are executed independently. Each block in a grid has a unique block ID number *blockIdx*. Block ID can be 1 or 2 dimensional. Usually there are 65535 blocks in a GPU. The programmer invokes the kernel, specifying the number of threads per block and the number of blocks per grid. We must note that prior the invoking the kernel, all the data required for the computations on GPU must be transferred from the CPU memory to GPU (global) memory using CUDA library functions. Invoking a kernel will hand over the control to the GPU and the specified kernel code will be executed on this data [26].

2.3. Existing approaches on GPU. Several implementations of multiple pattern matching algorithms running on GPUs have been introduced during the last years, offering a substantial performance increase compared to their sequential versions.

The Aho-Corasick algorithm was implemented in [40] using the CUDA API to perform network intrusion detection. The Aho-Corasick trie was represented by a two-dimensional *state_transition* array; each row of the array corresponded to a state of the trie, each column to a different character of the alphabet Σ while the cells of the array represented the *next* state. The array was precomputed by the CPU and was then stored to the texture memory of the GPU. The input string (in the form of network packets) was stored in buffers allocated using *pinned* host memory using a double buffering scheme and was copied to the global memory of the GPU as soon as each buffer was full. Since the input string was in the form of network packets, two different parallelization approaches were considered; with the first, each packet was processed by a different warp of threads while with the second, each packet was processed by a different thread. A speedup of 3.2 comparing to the respective sequential implementation was reported of the parallel implementation of Aho-Corasick when executed using an NVIDIA GeForce G80 GPU. A similar implementation of the Aho-Corasick algorithm was used in [41] to perform heavy-duty anti-malware operations.

The Parallel Failureless Aho-Corasick algorithm (PFAC), an interesting variant of the Aho-Corasick algorithm on a GPU was presented in [24]. It is significantly different than the rest of the parallel implementations of the same algorithm in the sense that each character of the input stream was assigned to a different thread of the GPU. Since each thread needs only to examine if a pattern exists *starting* at the specific character and no back-tracking takes place, the *supply* function of Aho-Corasick was removed. The *goto* function was mapped into a two-dimensional *state transition* array. The *state transition* array was then stored in shared memory by grouping the patterns based on their prefixes and distributing these groups into different multiprocessors. The original paper claimed a 4000 speedup of the PFAC algorithm when executed using an NVIDIA GeForce GTX 295 GPU comparing to the sequential implementation of Aho-Corasick.

In [39], the Aho-Corasick algorithm was implemented using the CUDA API. The Aho-Corasick trie was represented using a two-dimensional *state transition* array that was precomputed on the CPU and stored to the texture memory of the GPU. Instead of storing a map of the final states in another array, the final states that corresponded to a complete pattern were flagged directly in the *goto* array by reserving 1 bit. Bitwise operations were then used to check its value. The parallelization of the algorithm was achieved by assigning different characters of the input string to different threads of the GPU and letting them perform the matching by accessing the shared *state transition* array.

The work presented in [45] focused on the implementation of the Aho-Corasick algorithm on a GPU. The Aho-Corasick trie was precomputed on the CPU and was stored in the texture memory of the GPU but it is not clear the way it was represented. The input string was stored in the global memory of the GPU, partitioned to blocks and assigned to different threads in order to achieve parallelization. The threads were then responsible to process the input string and create an output array of the states of the Aho-Corasick trie that corresponded to each character position. The paper utilized a number of optimizations in order to further improve the performance of the algorithm implementation; casting the input string from *unsigned char* to *uint4* to ensure that each thread will read 16 input string characters from the global memory instead of 1. To further improve the bandwidth utilization, the accesses of multiple threads inside the same half-warp were coalesced by reading

the required data to process an input string block to the shared memory of the device. Finally, to avoid shared memory bank conflicts, the threads of a half-warp were accessing memory from different banks. The experimental results for the parallel implementation of the Aho-Corasick algorithm on an NVIDIA Tesla GT200 GPU reported a speedup of up to 9.5 relatively to the sequential implementation of the same algorithm.

An implementation of the Aho-Corasick algorithm was also presented in [14] using the CUDA API. Similar to the methodology used in previously published research papers, the trie of the algorithm was represented using a two-dimensional *state transition* array and was stored compressed in the texture memory of the GPU while the input string was stored in global memory. Kargus was introduced in [16], a software that utilizes the Aho-Corasick algorithm to perform intrusion detection on a hybrid system consisting of multi-core CPUs and heterogeneous GPUs. The trie of Aho-Corasick was represented in the GPU using a two-dimensional *state transition* array. The *state transition* array was created in the CPU and was stored in the GPU. The network packets that comprise the input string were stored in texture memory. To increase the utilization of the memory bandwidth of the GPU, the input string was cast to *uint4* using a technique similar to [45].

In [32], the Aho-Corasick and Commentz-Walter algorithms were used to perform virus scanning accelerated through a GPU. The Aho-Corasick and Commentz-Walter tries were represented in the GPU using stacks. The *goto* and *supply* functions were substituted with offsets, essentially serializing the trie in a continuous memory block. The stacks were precomputed in the CPU and were then transferred to the GPU. Parallelization was achieved using a data-parallel approach.

Simplified Wu-Manber (SWM), a variant of the Wu-Manber algorithm was implemented in [42] using the OpenCL API for network intrusion detection. The algorithm was modified by using smaller values of B as well as avoiding hashing and direct comparison of the patterns to the input string. With these changes, the shift tables became smaller and thus they were able to fit in the shared memory of the GPU cores. Moreover, multiple pattern comparisons didn't occur and thus thread divergence between cores was avoided. Finally, the hash calculations were entirely removed, further improving the performance of the algorithm implementation. The shift tables of the Wu-Manber variant were calculated by the CPU and were then transferred to the shared memory of the GPU. The input string in the form of network packets was stored in *pinned* host memory and was mapped to the global memory of the GPU. To achieve parallelization, different threads were assigned to non-adjacent sections of the network packets.

A group of researchers proposed a GPU version of the Wu-Manber multiple pattern matching algorithm [15]. The algorithm was to be used in network intrusion detection, and was implemented under OpenGL in order to take the advantage of an NVIDIA GeForce 7600 GT card. The experiments proved to be two times faster than the existing optimized version that was used in Snort [36].

An optimized version of the agrep algorithm was presented [23], which is based on Wu-Manber algorithm using the CUDA API and taking advantage of a GeForce GTX285, for approximate nucleotide sequence matching. The performance of the implementation was evaluated for sequences of genomes, comparing an OpenMP implementation to the CUDA implementation of the algorithm, and proved to exhibit 70-fold and 36-fold performance speedups, for pattern sizes of 30 and 60 respectively.

Moreover, a modified version of the Wu-Manber algorithm for approximate matching was presented in [38]. The implementation was simplified to run on a NVIDIA GeForce 480 using the OpenCL API, and managed to achieve 62-fold speedups.

3. Parallel implementations using CUDA. In this section, we present a basic data-parallel implementation strategy for the Aho-Corasick, Set Horspool, Set Backward Oracle Matching, Wu-Manber and SOG multiple pattern matching algorithms, analyzes the characteristics of the algorithm implementations that leverage the capabilities of the device, discusses the flaws that affect their performance and addresses them using different optimization techniques.

3.1. Basic parallelization strategy. Table 3.1 lists the notation for the parallel implementation of the algorithms with the CUDA API that is used for the rest of this paper.

To expose the parallelism of the multiple pattern matching algorithms, the following basic data-parallel implementation strategy was used. The preprocessing phase of the algorithms was performed sequentially on the host CPU. The input string and all preprocessing arrays, including the arrays that represent the trie of the Aho-Corasick, Set Horspool and Set Backward Oracle Matching algorithms as discussed later in this

TABLE 3.1
GPGPU implementation notation

$numBlocks$	The number of thread blocks
$blockDim$	The size in threads of each block
$threadId$	The unique ID of each thread of a block
$blockId$	The unique ID of each block
S_{chunk}	The size in characters of each chunk
S_{thread}	The number of characters that each thread processes
$S_{memsize}$	The size in bytes of the shared memory per thread block

paper, were copied to the global memory of the device. The input string was subsequently partitioned into $numBlocks$ character chunks, each with a size S_{chunk} of $\frac{n}{numBlocks}$ characters. The chunks were then assigned to $numBlocks$ thread blocks. Each chunk was further partitioned into $blockDim$ sub-chunks, that in turn were assigned to each of the $blockDim$ threads of a thread block. Each thread uses two auxiliary variables, *start* and *stop* were used to indicate the input string positions where the search phase begins and ends respectively. More specifically, the variable *start* is $(blockId \times n)/numBlocks + (n \times threadId)/(numBlocks \times blockDim)$ and the variable *stop* is $start + n/(numBlocks \times blockDim)$. To ensure the correctness of the results, $m-1$ overlapping characters were used per thread. These characters were added to the *stop* variable of Aho-Corasick and SOG, algorithms that perform forward searching and to the *start* variable of Set Horspool, Set Backward Oracle Matching and Wu-Manber, algorithms that scan the input string backwards. Therefore, each thread processed $S_{thread} = \frac{n}{numBlocks \times blockDim} + m - 1$ characters for a total of $(m-1)(numBlocks \times blockDim - 1)$ additional characters. In other words, each thread executes the corresponding search phase of the multiple string matching algorithms on the device as a CUDA kernel. Furthermore, an array *out* with a size of $numBlocks \times blockDim$ integers was used to store the number of matches per thread. To avoid extra coding complexity it is assumed that n is divisible by both $numBlocks$ and $blockDim$. Since the character chunks have to overlap, the fewer possible thread blocks should be used to reduce the redundant characters as long as the maximum possible occupancy level is maintained per SM. A pseudocode of the host function for the parallel implementation of the algorithms is given in Figure 3.1.

```
Main procedure
main()
{
    1. Execute the preprocessing phase of the algorithms;
    2. Where relevant represent tries using arrays;
    3. Allocate one-dimensional and two-dimensional arrays in the global memory
       of the device using the cudaMalloc() and cudaMallocPitch() functions respectively;
    4. Copy the data from the host arrays to the device arrays using
       the cudaMemcpy() and cudaMemcpy2D() functions;
    5. Launch CUDA kernel;
    6. Copy the results array back to host memory;
    7. Calculate the results;
}
```

FIG. 3.1. *Pseudocode of the host function*

Three tables were constructed during the preprocessing phase of the Aho-Corasick algorithm implementation. *State_transition* that corresponds to the *goto* function is a two-dimensional array where each row corresponds to a state of the trie, each column to a different character of the alphabet Σ and the cells of the array represent the next state. To ensure that alignment requirements are met on each row, *state_transition* was allocated as pitched linear device memory using the *cudaMallocPitch()* function. *State_supply* that corresponds to the *supply* function is one-dimensional array, allocated using the *cudaMalloc()* function. Each column of the array corresponds to a different state of the trie while the cells represent the supply state of a given state and the information whether that state is final or not respectively. Since the Aho-Corasick trie can have a maximum of $m \times d + 1$ trie states, a value that for the experiments of this paper was typically equal to more than 65536, each state was represented using a 4-byte integer. In this implementation, there can be divergence among the execution paths of the threads when previous states are visited using the *supply* function of the algorithm.

The Set Horspool implementation used identical version to the *state_transition* table of the Aho-Corasick implementation to locate all the occurrences of any pattern in the input string during the search phase in reverse. Moreover, *state_shift* was used, a one-dimensional array allocated using the *cudaMalloc()* function with a size of $|\Sigma|$ that represents the bad character shift of the Horspool algorithm. Each row of the array corresponds to a different state of the trie, each column to a different character of the alphabet Σ while each cell holds the number of input string character positions that can be safely skipped during the search phase. There can be divergence in the execution path of the threads in the basic implementation of Set Horspool when the next states are visited using the *goto* function of the algorithm.

To represent the factor oracle of the Set Backward Oracle Matching algorithm, a modified version of the *state_transition* array was used with an equal size to the versions used by the Aho-Corasick and Set Horspool implementations. The cells of *state_transition* correspond not only to the next state for each character of the alphabet Σ but also contain information for the external transitions of the trie. The *state_transition* array was allocated as pitched linear device memory using the *cudaMallocPitch()* function to ensure that alignment requirements are met in the global memory of the device. There is the possibility for the threads of every half-warp to diverge from their execution path, causing individual threads to branch out and execute independently. This can happen when there are potential matches of some patterns from the pattern set since they have to be verified directly to the input string.

The Wu-Manber algorithm uses the one-dimensional *SHIFT* table, allocated using the *cudaMalloc()* function and the two-dimensional *HASH* and *PREFIX* tables allocated as pitched linear memory using *cudaMallocPitch()*. Each row of the *HASH* and *PREFIX* represents a different pattern from the pattern set while the columns represent different hash values. The relevant data structures were copied to the global memory of the device with no modifications. As with the implementation of the Aho-Corasick, Set Horspool and Set Backward Oracle Matching algorithms, there can be divergence among the threads of the same-half warp during the verification of potential matches. The value of the *out* array is set during the verification phase.

Finally, SOG computes the V, *hs_array* and *hs'_array* one-dimensional tables during the preprocessing phase, allocated as linear memory using the *cudaMalloc()* function. V is a bit vector that can store the hash value h for each different B-character block, *hs_array* holds the hash value hs for each pattern from the pattern set while *hs'_array* stores the two-level hs' hash values. The implementation of the SOG algorithm exhibits a significant level of thread divergence that is expected to affect its performance. As with Wu-Manber, the value of the *out* array is also set during the verification phase.

3.2. Implementation Limitations and Optimization Techniques. We know that accesses to global memory for compute capability 1.3 GPUs by all threads of a half-warp are coalesced into a single memory transaction when all the requested words are within the same memory segment. The segment size is 32 bytes when 1-byte words are accessed, 64 bytes for 2-byte words and 128 bytes for words of $4, 8$ and 16 bytes. With the basic implementation strategy, each thread reads a single 1-byte character on each iteration of the search loop; in this case the memory segment has a size of 32 bytes. When $S_{thread} > 32$, each thread accesses a word from a different memory segment of the global memory. This results to uncoalesced memory transactions, with one memory transaction for every access of a thread. Since the maximum memory throughput of the global memory is 128 bytes per transaction, the access pattern of the threads results in the utilization of only the $\frac{1}{128}$ of the available bandwidth.

3.2.1. Coalescing Memory Accesses. To work around the coalescing requirements of the global memory and increase the utilization of the memory bandwidth, it is important to change the memory access pattern by reading words from the same memory segment and subsequently store them in the shared memory of the device. This involves the partition of the input string into $\frac{n}{S_{memsize}}$ chunks and the collective read of $S_{memsize}$ characters from the global into the shared memory by all $blockDim$ threads of a thread block. For each 16 successive characters from the same segment then, only a single memory transaction will be used. This technique results in the utilization of the $\frac{1}{8}$ of the global memory bandwidth, improved by a factor of 16. The threads can subsequently access the characters stored in shared memory in any order with a very low latency.

Using the shared memory to increase the utilization of the memory bandwidth has two disadvantages. First, a total of $\frac{n}{S_{memsize}} \times (blockDim - 1) \times (m - 1)$ redundant characters are used that introduce significantly more work overhead when compared to the basic data-parallel implementation strategy. Second, using the shared

memory effectively reduces the occupancy of the SMs. As the size of the shared memory for each SM of the GTX 280 GPU is $16KB$, using the whole shared memory would reduce the occupancy to one thread block per SM. Partitioning the shared memory is not an efficient option since it was determined experimentally that it further increases the total work overhead.

3.2.2. Packing Input String Characters. The utilization of the global memory bandwidth can also increase when the threads read 16-byte words instead of single characters on every memory transaction. For that, the built-in *uint*4 vector can be used, a C structure with members x, y, z, and w that is derived from the basic integer type. This way, each thread accesses an 128-bit *uint*4 word that corresponds to 16 characters of the input string with a single memory transaction while at the same time the memory segment size increases from 32 to 128 bytes. By having each thread read 128-bit *uint*4 words from different memory segments results in the utilization of the $\frac{1}{8}$ of the global memory bandwidth similar to the coalescing technique above. The input string array stored in global memory can be casted to *uint*4 as follows:

```
uint4 *uint4_text = reinterpret_cast < uint4 * > ( d_text );
```

The two previous techniques can be combined; reading 16 successive 128-bit words or 256 bytes in the form of 16 *uint*4 vectors from global to shared memory can be done with just two memory transactions, fully utilizing the global memory bandwidth. The input string characters are then extracted from the uint4 vectors as retrieved from the global memory and are subsequently stored in shared memory on a character-by-character basis. To access the characters inside a uint4 vector, the vector can be recasted to *uchar*4:

```
uint4 uint4_var = uint4_text[i];

uchar4 uchar4_var0 = *reinterpret_cast < uchar4 * > ( &uint4_var.x );
uchar4 uchar4_var1 = *reinterpret_cast < uchar4 * > ( &uint4_var.y );
uchar4 uchar4_var2 = *reinterpret_cast < uchar4 * > ( &uint4_var.z );
uchar4 uchar4_var3 = *reinterpret_cast < uchar4 * > ( &uint4_var.w );
```

The drawback of casting input string characters to *uint*4 vectors and recasting them to *uchar*4 vectors is that it can be expensive in terms of processing power.

3.2.3. Texture Binding. The preprocessing arrays of the algorithms are relatively small in size while at the same time they are frequently accessed by the threads. In the case of the Set Backward Oracle Matching, Wu-Manber and SOG algorithms where a character-by-character verification of the patterns to the input string is required, the pattern set array is also accessed often. The performance of the parallel implementation of the algorithms should then benefit from the binding of the relevant arrays to the texture memory of the device. The linear memory region was bound to the texture reference using *cudaBindTexture()* for one-dimensional arrays and *cudaBindTexture2D()* for two-dimensional arrays. The textures were then accessed in-kernel using the *tex1Dfetch()* and *tex2D()* functions. Arrays accessed via textures not only take advantage of the texture caches to minimize the memory latency when cache hits occur but also bypass the coalescing requirements of the global memory.

3.2.4. Avoiding Bank Conflicts. The shared memory of the GTX 280 GPU consists of 16 memory banks numbered $0 - 15$. The banks are organized in such a way that successive 32-bit words are mapped into successive banks with word i being stored in bank i mod 16. Bank conflicts occur when two or more threads of the same half-warp try to simultaneously access words $i, j, \ldots z$ when $i \bmod 16 = j \bmod 16 = \ldots = z \bmod 16$. When the memory coalescence optimizations described above are used, it is challenging to avoid bank conflicts when the 16 characters of a *uint*4 vector are successively stored to the shared memory and when are retrieved from shared memory by the threads of the same half-warp during the search phase. Storing the input string characters in shared memory results in a 4-way bank conflict. An alternative would be to cast each *uint*4 vector to 4 *uchar*4 vectors and store them in shared memory in a round-robin fashion:

```
int tid16 = threadIdx.x % 16;

if ( tid16 < 4 ) {

        uchar4_s_array[threadIdx.x * 4 + 0] = uchar4_var0;
```

```
            uchar4_s_array[threadIdx.x * 4 + 1] = uchar4_var1;
            uchar4_s_array[threadIdx.x * 4 + 2] = uchar4_var2;
            uchar4_s_array[threadIdx.x * 4 + 3] = uchar4_var3;

    } else if ( tid16 < 8 ) {

            uchar4_s_array[threadIdx.x * 4 + 1] = uchar4_var1;
            uchar4_s_array[threadIdx.x * 4 + 2] = uchar4_var2;
            uchar4_s_array[threadIdx.x * 4 + 3] = uchar4_var3;
            uchar4_s_array[threadIdx.x * 4 + 0] = uchar4_var0;

    } else if ( tid16 < 12 ) {

            uchar4_s_array[threadIdx.x * 4 + 2] = uchar4_var2;
            uchar4_s_array[threadIdx.x * 4 + 3] = uchar4_var3;
            uchar4_s_array[threadIdx.x * 4 + 0] = uchar4_var0;
            uchar4_s_array[threadIdx.x * 4 + 1] = uchar4_var1;

    } else {

            uchar4_s_array[threadIdx.x * 4 + 3] = uchar4_var3;
            uchar4_s_array[threadIdx.x * 4 + 0] = uchar4_var0;
            uchar4_s_array[threadIdx.x * 4 + 1] = uchar4_var1;
            uchar4_s_array[threadIdx.x * 4 + 2] = uchar4_var2;
    }
```

s_array represents an array stored in the shared memory of the device with a size of $S_{memsize}$. This technique was not used since in practice the performance of the implementations did not improve. Although it was conflict-free when storing the vectors, it resulted in a 4-way thread divergence that caused the serialization of accesses to shared memory, the same effect that the example code was trying to avoid. The modulo operator is very expensive when used inside CUDA kernels and had a significant impact therefore in the implementations' performance.

For further details and pseudocode about the basic and optimized parallel implementations of the Aho-Corasick, Set Horspool, Set Backward Oracle Matching, Wu-Manber and SOG algorithms are presented in [20].

4. Experimental results. The performance of the parallel implementations of the algorithms was evaluated by comparing the running time of the GPU implementations to that of the sequential version. The computing platform in which the experiments of these implementations were executed is composed by an Intel Xeon CPU with a $2.40GHz$ clock speed and $2GB$ of memory which was used as a host and a GTX 280 GPU which was used as the device. The sequential algorithms were implemented using the ANSI C programming language and were compiled using the GCC compiler with the "-O2" and "-funroll-loops" optimization flags. The parallel GPU implementations of the algorithms were compiled using the NVCC 5.0 compiler of the CUDA API with the "-O2" optimization flag.

For the comparison of the parallel implementations we used the running time and speedup as measures. Running time is the total execution time of an implementation including the preprocessing time and the searching time. Preprocessing time is the time in seconds an algorithm uses to preprocess the pattern / pattern set while searching time is the total time in seconds an algorithm uses to locate all occurrences of any pattern in the input string. The running time of the CUDA implementations was measured using the CUDA event API. To decrease random variation, all time results were averages of 100 runs. Speedup is defined as (Sequential running time on a specific CPU) / (Parallel running time on a specific GPU).

The parameters that affect the performance of parallel multiple pattern matching algorithms are the size of the input string n, the size of the pattern set d, the size of the patterns m and the size $|\Sigma|$ of the alphabet used.

The data set used for the experiments was similar to the sets used in [17, 22, 35]. It consisted of randomly generated input strings of a binary alphabet, the genome of Escherichia coli from the Large Canterbury Corpus, the Swiss Prot. Amino Acid sequence database, the FASTA Amino Acid (FAA) and FASTA Nucleic Acid (FNA) sequences of the A-thaliana genome and natural language input strings of the English alphabet:

- Randomly generated input strings of size $n = 4,000,000$ with a binary alphabet. The alphabet used was $\Sigma = \{0, 1\}$.
- The genome of Escherichia coli from the Large Canterbury Corpus with a size of $n = 4,638,690$ characters and the FASTA Nucleic Acid (FNA) of the A-thaliana genome with a size of $n = 116,237,486$

characters. The alphabet $\Sigma = \{a, c, g, t\}$ of both genomes consisted of the four nucleobases of the Deoxyribonucleic Acid (DNA).

- The FASTA Amino Acid (FAA) of the A-thaliana genome with a size of $n = 10,830,882$ characters and the Swiss Prot. Amino Acid sequence database with a size of $n = 177,660,096$ characters. The alphabet $\Sigma = \{a, c, d, e, f, g, h,$
  $i, k, l, m, n, p, q, r, s, t, v, w, y\}$ used by the databases consisted of 20 different characters.
- The CIA World Fact Book from the Large Canterbury Corpus. The input string had a size of $n = 1,914,500$ characters and an alphabet of size $|\Sigma| = 128$ characters.

The pattern set for each execution was created from subsequences of the corresponding input string, consisting of $1,000$ and $8,000$ patterns with each pattern having a size of $m = 8$ and $m = 32$ characters.

The performance of the algorithm implementations is evaluated when a number of optimizations are applied for different sets of data and is compared to the performance of the sequential implementations. Each implementation stage also incorporates the optimizations of the previous stages.

1. The first stage of the implementation was unoptimized. The input string, the pattern set, the preprocessing arrays of the algorithms and the *out* array that holds the number of matches per thread, were stored in the global memory of the device. Each thread accessed input string characters directly from the global memory as needed and stored the number of matches directly to *out*. At the end of the algorithms' search phase, the results array was transferred to the host memory and the total number of matches was calculated by the CPU.
2. The second stage of the implementation involved the binding of the preprocessing arrays and the pattern set array in the case of the Set Backward Oracle Matching, Wu-Manber and SOG algorithms to the texture memory of the device.
3. In the third stage, the threads worked around the coalescing requirements of the global memory by collectively reading input string characters and storing them to the shared memory of the device.
4. The fourth stage of the implementation was similar to the third but in addition, each thread read simultaneously 16 input string characters from the global memory by using a *uint*4 vector. The characters were extracted from the vectors using *uchar*4 vectors and were subsequently stored to the shared memory.

For the experiments of this paper, 30 thread blocks were used, one per each SM, with 256 threads per block. As discussed in section 3, the shared memory was utilized to work around the coalescing requirements of the global memory during the third implementation stage and it was determined experimentally that further partitioning the shared memory was not an efficient option as the total work overhead increases.

Figures 4.1 to 4.5 depict the speedup of different implementations of the Aho-Corasick, Set Horspool, Set Backward Oracle Matching, Wu-Manber and SOG algorithms for randomly generated input strings, biological sequence databases and English alphabet data on a GTX280 GPU. As can generally be seen, the performance of the algorithms increased significantly when implemented in parallel on a GPU using the CUDA API, even before applying any optimization. When the presented optimization techniques were used though, the implementations had a significantly higher speedup over the sequential versions, although different algorithms were affected in different ways.

When the preprocessing arrays and the pattern set in the case of some algorithms were bound to the texture memory of the device, the performance of the implementations increased considerably. Also in most cases, the implementations exhibited a faster running time when the accesses to the input string for the threads of the same half-warp were coalesced with the use of the shared memory. On the other hand, it can be observed that the performance of the algorithm implementations did not improve significantly in the fourth implementation stage due to the high cost involved in casting the input string characters to *uint*4 vectors and recasting them to *uchar*4 vectors as discussed in section 3.

For the Aho-Corasick algorithm, the experimentally measured speedup for the basic parallel implementation of the Aho-Corasick algorithm over the sequential implementation ranged from 3.2 to 10.9. For the second and third implementation stages, the overhead that was caused by the frequent accesses of the threads to the global memory of the device was partially reduced with an additional performance increase of up to 3.5 times. The speedup of the final, optimized kernel ranged between 10 and 18.5 comparing to the sequential implementation,

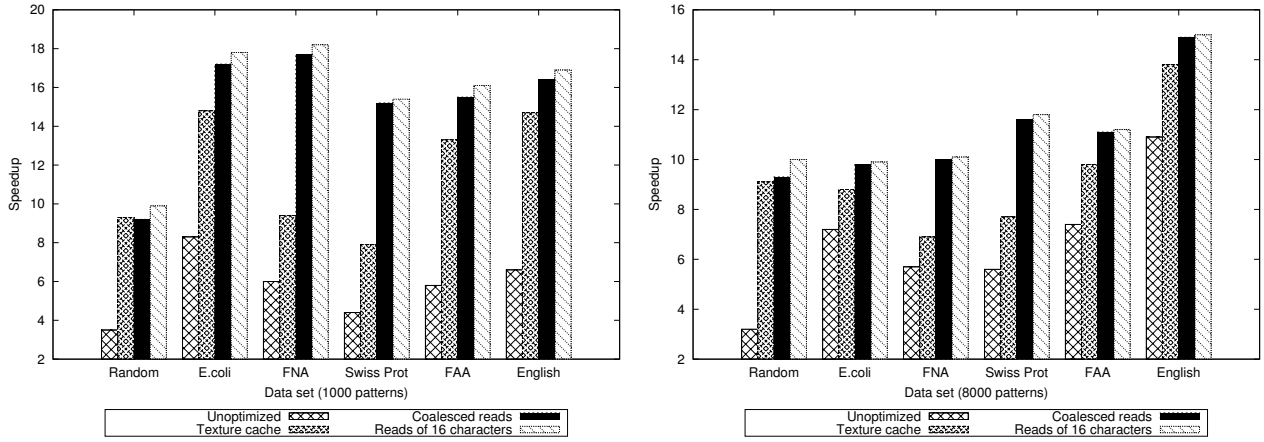

FIG. 4.1. *Speedup of different Aho-Corasick implementations for randomly generated input strings, biological sequence databases and English alphabet data*

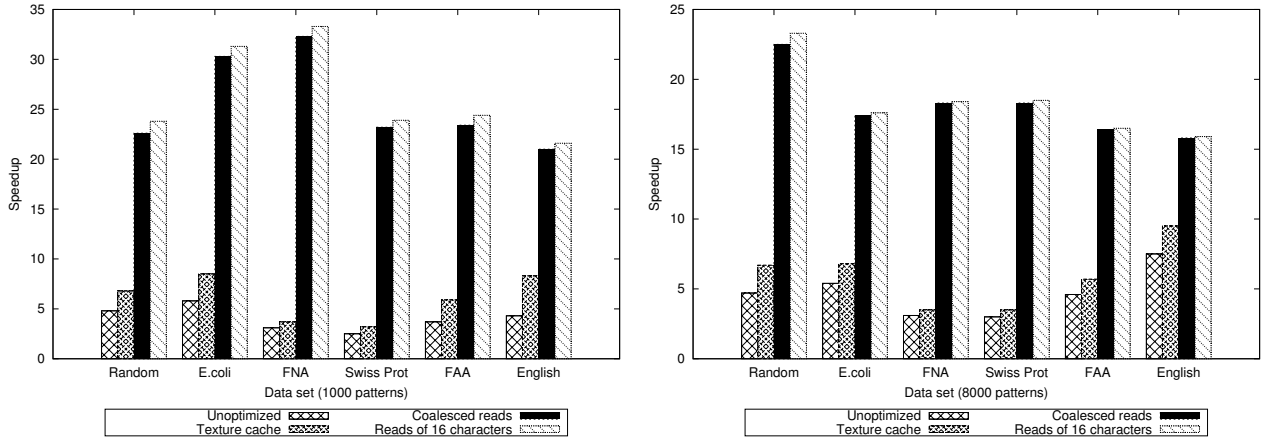

FIG. 4.2. *Speedup of different Set Horspool implementations for randomly generated input strings, biological sequence databases and English alphabet data*

slightly faster than the reported speedup in [40, 45]. This was expected since for the experiments of this paper a more recent GPU was used to evaluate the performance of the algorithm implementations. The highest speedup was generally observed when the E.coli genome and the FASTA Nucleic Acid (FNA) of the A-thaliana genome were used, together with sets of $1,000$ patterns.

The unoptimized implementation of the Set Horspool algorithm had a speedup between 2.5 and 7.5 over the sequential version. As with the implementation of the Aho-Corasick algorithm, the parallel searching time of Set Horspool also benefited when the preprocessing arrays were bound to the texture memory of the device in the second stage of the implementation, although to a lesser degree; the speedup increased between 1.2 and 1.9 times comparing to the basic parallel implementation for sets of $1,000$ patterns and between 1.1 and 1.4 times for sets of $8,000$ patterns. The performance of the implementation increased to a much greater extent during the third implementation stage, with a speedup increase over the second implementation stage of up to 8.7. The speedup of the optimized kernel was between 22.1 and 33.7, significantly higher than the corresponding speedup of the Aho-Corasick algorithm. Again, the highest speedup was achieved when the E.coli genome and the FASTA Nucleic Acid of the A-thaliana genome were used, together with sets of $1,000$ patterns.

As can be seen in Figure 4.3, the performance of the Set Backward Oracle Matching algorithm had similar

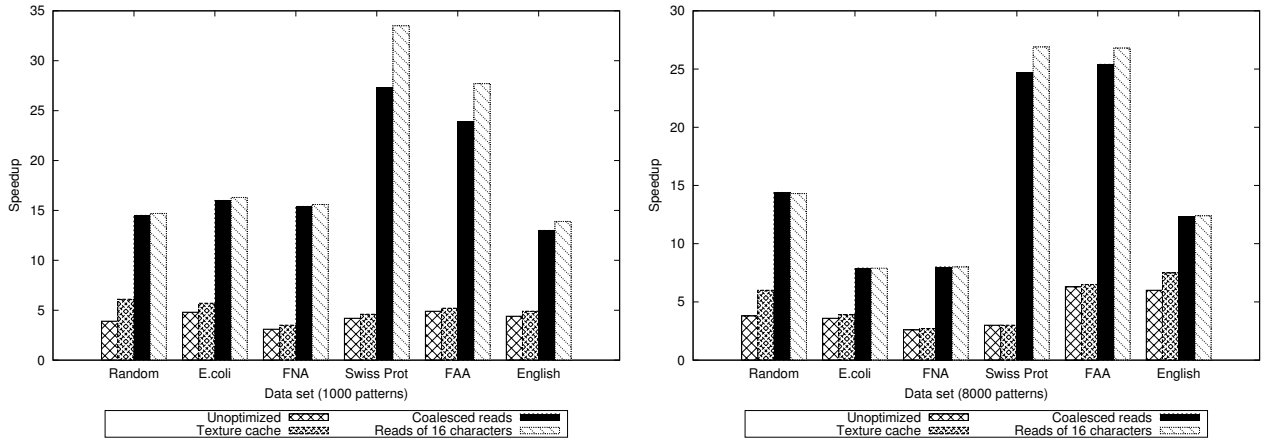

FIG. 4.3. *Speedup of different Set Backward Oracle Matching implementations for randomly generated input strings, biological sequence databases and English alphabet data*

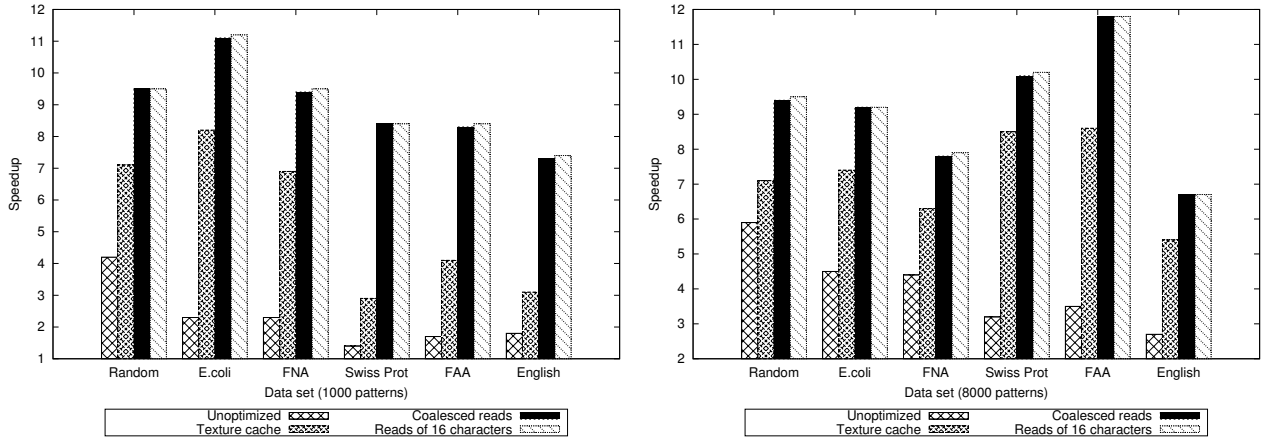

FIG. 4.4. *Speedup of different Wu-Manber implementations for randomly generated input strings, biological sequence databases and English alphabet data*

characteristics to that of Set Horspool. The speedup of the implementation ranged between 2.6 and 6.3 over the sequential version of the algorithm and further improved by up to 1.6 times when the preprocessing arrays and the pattern set were bound to the texture memory of the device. When the accesses for the threads of the same half-warp were coalesced, the performance of the algorithm implementation improved by up to an additional 8.2 times. The speedup of the optimized kernel of the algorithm ranged between 16.1 and 33.7. The highest speed was achieved when the Swiss Prot. Amino Acid sequence database and the FASTA Amino Acid (FAA) of the A-thaliana genome were used, for sets of either $1,000$ or $8,000$ patterns.

The unoptimized implementation of Wu-Manber was up to 5.9 times faster than its sequential implementation. When the pattern set as well as the *SHIFT*, *HASH* and *PREFIX* tables that were created during the preprocessing phase of the algorithm, were bound to the texture memory of the device and the shared memory was used to bypass the coalescing requirements of the global memory during the second and third implementation stages, the speedup of the implementation increased by up to an additional 6.1 times. The performance of the final, optimized implementation of Wu-Manber was up to 11.8 times faster than the sequential implementation and was not significantly affected by the type of the dataset used.

The parallel implementation of SOG exhibited the lowest speedup comparing to the rest of the presented

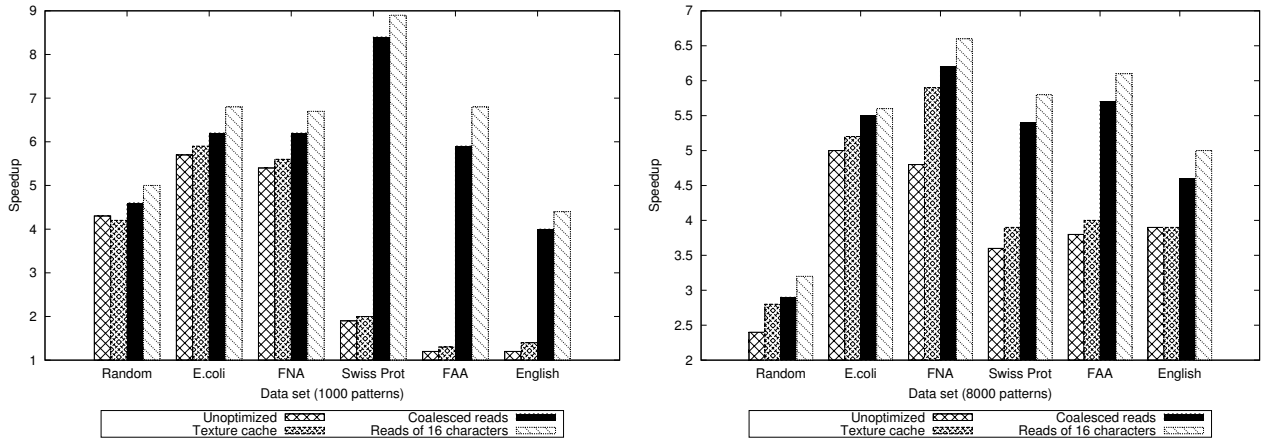

FIG. 4.5. *Speedup of different SOG implementations for randomly generated input strings, biological sequence databases and English alphabet data*

multiple pattern matching algorithms. The performance of the unoptimized kernel increased by up to 5.7 times, comparing to the sequential implementation. As can be seen in Figure 4.5, the algorithm implementation had a significant speedup increase when the preprocessing arrays were bound to the texture memory and the input string was copied to the shared memory of the device. The increase was more evident for the Swiss Prot. Amino Acid sequence database, the FASTA Amino Acid of the A-thaliana genome and for the English language datasets, all of them data with a large alphabet size. The optimized algorithm implementation had a speedup of up to 8.9 that was observed when the Swiss Prot. Amino Acid sequence database was used together with sets of $1,000$ patterns.

From the results, we can also see that there are two trends in the performance of the CUDA implementations when the number of patterns is scaled. The speedup of the Aho-Corasick, Set Backward Oracle Matching, Wu-Manber and SOG algorithms had a rising tendency as the patterns increased for Random, FAA, FNA/Swiss Prot and FNA/FAA/English databases respectively, in contrast with the other cases where the speedup of the algorithms had a decreasing tendency. This decreasing tendency for large pattern sets is due to the fact that the preprocessing phase of the algorithms was performed sequentially on the CPU. More specifically, the preprocessing phase of the algorithms imposes a relatively high cost in terms of time to setup the necessary data structures and the time to preprocess the pattern set increases linearly with the number of patterns d. Furthermore, the searching phase on the GPU becomes slower for large pattern sets because of the unfavorable GPU memory hierarchy access times. This fact has been alleviated in more recent versions of GPU devices.

Although the performance of the CUDA implementations increased significantly when different optimization techniques applied, the speedup of the implementations is not proportional to the number of cores in GPU. This fact is due to three reasons: One reason is that the implementations of the algorithms require the frequent access of the threads to the GPU memory hierarchy of the device. Another limiting factor is the significant overhead of the thread divergence, the serial execution of threads within the same warp that execute different instructions as a result of *if-else* and *while* statements in the kernel. Furthermore, the optimized implementations of each algorithm require increased register usage per thread to execute out of the 16KB registers per block that the GTX 280 provides. More specifically, the algorithms Aho-Corasick, Set Horspool, Set Backward Oracle Matching, Wu-Manber and SOG require 18, 19, 22, 22 and 21 registers per thread, respectively. The number of registers per thread is a limiting factor on the number of concurrent threads the hardware is capable of executing. The proposed implementations could be scaled up by using a large number of GPUs by means of cuda-aware MPI infrastructure and by applying suitable modifications of the data structures used in preprocessing phase of the algorithms so that pattern set can be partitioned and the GPU memory hierarchy is better utilized.

5. Conclusions. In this paper, we presented details on the design and the implementation of the Aho-Corasick, Set Horspool, Set Backward Oracle Matching, Wu-Manber and SOG multiple pattern matching algorithms in parallel on a GPU using the CUDA API. Different optimizations were used to further increase the performance of the parallel implementations. Based on the experimental results, it was concluded that the speedup of the basic implementation of the algorithms ranged between 2.5 and 10.9 over the equivalent sequential version of the algorithms. It was also shown that by applying the optimizations discussed in this paper, the parallel implementation of the Aho-Corasick algorithm was up to 18.5 times faster than the corresponding sequential implementations, the implementations of the Set Horspool and the Set Backward Oracle Matching algorithms were up to 33.7 times faster, the parallel implementation of the Wu-Manber algorithm was up to 11.8 times faster while the implementation of SOG was up to 8.9 times faster.

It was concluded that the parallel implementations of the Set Horspool and the Set Backward Oracle Matching algorithms exhibited higher speedup than other three parallel implementations. Note that the experimental results presented herein for Aho-Corasick algorithm achieve a high speedup of 18.5, which compares favorably with the results presented in the previous research studies [40, 45]. Further, the speedup presented herein for parallel version of the Wu-Manber (WM) algorithm is relatively better than the parallel version of the WM algorithm that has been implemented using OpenGL and – achieved twice the performance of the corresponding WM algorithm used in Snort [15]. The speedups reported in those articles are comparable to the speedups of this paper because the aforementioned research proceed in a fashion similar to our work, i.e. they parallelize the same original Aho-Corasick and WM algorithms. On the other hand, two approximate string matching implementations, based on a modified Wu-Manber algorithm [23] and the bit-parallel BPR algorithm [38] achieve better results than those presented herein. The speedups reported are 76 and 30, in the former reference and 62, in the latter. Furthermore, some other GPU implementations [24, 39, 14, 42] seem to perform better than those presented herein. These studies have incorporated significant modifications of the original Aho-Corasick and Wu-Manber algorithms, such that the resulting GPU implementations avoid important problems such as memory accesses of threads, thread divergence between cores and hashing comparisons/calculations. Therefore, the resulting speedups are not directly comparable to the reported speedups of ours or other approaches, since the sequential algorithm, i.e. the basis of comparison has been modified significantly. Furthermore, in all the previous cases, the direct comparison of absolute speedups is not possible, since the experimental setups used in different experiments, that is host machines and GPU devices, are not directly comparable.

Acknowledgments. The authors are thankful to the reviewers for the useful comments and suggestions, which improved the presentation of this paper.

REFERENCES

[1] A. Aho and M. Corasick, *Efficient String Matching: An Aid to Bibliographic Search*, Communications of the ACM, 18 (1975), pp. 333–340.
[2] C. Allauzen, M. Crochemore, and M. Raffinot, *Factor Oracle: A New Structure for Pattern Matching*, SOFSEM99: Theory and Practice of Informatics, 1725 (1999), pp. 758–758.
[3] A. Apostolico and Z. Galil, *Pattern Matching Algorithms*, Oxford University Press, 1997.
[4] S. Arudchutha, T. Nishanthy, and R. Ragel, *String matching with multicore cpus: Performing better with the aho-corasick algorithm*, in Industrial and Information Systems (ICIIS), 2013 8th IEEE International Conference on, Dec 2013, pp. 231–236.
[5] R. Baeza-Yates and G. Gonnet, *A New Approach to Text Searching*, Communications of the ACM, 35 (1992), pp. 74–82.
[6] R. Boyer and J. Moore, *A Fast String Searching Algorithm*, Communications of the ACM, 20 (1977), pp. 762–772.
[7] X. Chen, B. Fang, L. Li, and Y. Jiang, *WM+: An Optimal Multi-pattern String Matching Algorithm Based on the WM Algorithm*, Advanced Parallel Processing Technologies, (2005), pp. 515–523.
[8] B. Commentz-Walter, *A String Matching Algorithm Fast on the Average*, in Proceedings of the 6th Colloquium, on Automata, Languages and Programming, 1979, pp. 118–132.
[9] M. Crochemore, A. Czumaj, L. Gasieniec, S. Jarominek, T. Lecroq, W. Plandowski, and W. Rytter, *Speeding Up Two String-matching Algorithms*, Algorithmica, 12 (1994), pp. 247–267.
[10] M. Crochemore, A. Czumaj, L. Gasieniec, T. Lecroq, W. Plandowski, and W. Rytter, *Fast Practical Multi-pattern Matching*, Information Processing Letters, 71 (1999), pp. 107 – 113.
[11] M. Crochemore and W. Rytter, *Text Algorithms*, Oxford University Press, Inc., 1994.
[12] G. Grep, *Webpage containing information about the gnu grep search utility.* Website, 2012. http://www.gnu.org/software/grep/.

[13] R. Horspool, *Practical Fast Searching in Strings*, Software: Practice and Experience, 10 (1980), pp. 501–506.
[14] L. Hu, Z. Wei, F. Wang, X. Zhang, and K. Zhao, *An Efficient AC Algorithm with GPU*, Procedia Engineering, 29 (2012), pp. 4249–4253.
[15] N.-F. Huang, H.-W. Hung, S.-H. Lai, Y.-M. Chu, and W.-Y. Tsai, *A GPU-based multiple-pattern matching algorithm for network intrusion detection systems*, in Advanced Information Networking and Applications - Workshops, 2008. AINAW 2008. 22nd International Conference on, March 2008, pp. 62–67.
[16] M. Jamshed, J. Lee, S. Moon, I. Yun, D. Kim, S. Lee, Y. Yi, and K. Park, *Kargus: a Highly-scalable Software-based Intrusion Detection System*, in Proceedings of the ACM Conference on Computer and Communications Security (CCS), 2012.
[17] P. Kalsi, H. Peltola, and J. Tarhio, *Comparison of Exact String Matching Algorithms for Biological Sequences*, Communications in Computer and Information Science, 13 (2008), pp. 417–426.
[18] S. Kim and Y. Kim, *A Fast Multiple String-pattern Matching Algorithm*, Proceedings of the 17th AoM/IAoM Inernational Conference on Computer Science, (1999), pp. 1–6.
[19] D. Knuth, J. Morris, and V. Pratt, *Fast Pattern Matching in Strings*, SIAM Journal on Computing, 6 (1977), pp. 323–350.
[20] C. Kouzinopoulos, *Parallel and Distributed Implementations of Two-Dimensional and Multiple Pattern Matching Algorithms*, PhD thesis, Department of Applied Informatics, University of Macedonia, 2013.
[21] C. Kouzinopoulos, P. Michailidis, and K. Margaritis, *Performance Study of Parallel Hybrid Multiple Pattern Matching Algorithms for Biological Sequences*, in International Conference on Bioinformatics-Models, Methods and Algorithms, 2012, pp. 182–187.
[22] T. Lecroq, *Fast Exact String Matching Algorithms*, Information Processing Letters, 102 (2007), pp. 229–235.
[23] H. Li, B. Ni, M.-H. Wong, and K.-S. Leung, *A fast CUDA implementation of agrep algorithm for approximate nucleotide sequence matching*, in Application Specific Processors (SASP), 2011 IEEE 9th Symposium on, June 2011, pp. 74–77.
[24] C. Lin, S. Tsai, C. Liu, S. Chang, and J. Shyu, *Accelerating String Matching using Multi-Threaded Algorithm on GPU*, in Global Telecommunications Conference (GLOBECOM 2010), 2010 IEEE, 2010, pp. 1–5.
[25] P. Liu, Y. Liu, and J. Tan, *A Partition-based Efficient Algorithm for Large Scale Multiple-strings Matching*, in String Processing and Information Retrieval, Springer, 2005, pp. 399–404.
[26] P. D. Michailidis and K. G. Margaritis, *Accelerating kernel density estimation on the GPU using the CUDA framework*, Applied Mathematical Sciences, 7 (2013), pp. 1447–1476.
[27] R. Muth and U. Manber, *Approximate Multiple String Search*, in Combinatorial Pattern Matching, Springer, 1996, pp. 75–86.
[28] G. Navarro and K. Fredriksson, *Average Complexity of Exact and Approximate Multiple String Matching*, Theoretical Computer Science, 321 (2004), pp. 283–290.
[29] G. Navarro and M. Raffinot, *A Bit-parallel Approach to Suffix Automata: Fast Extended String Matching*, Lecture Notes in Computer Science, 1448 (1998), pp. 14–33.
[30] ———, *Flexible Pattern Matching in Strings: Practical On-line Search Algorithms for Texts and Biological Sequences*, Cambridge University Press, 2002.
[31] NVIDIA, *NVIDIA CUDA Compute Unified Device Architecture Programming Guide, version 5.5*, 2013. http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf.
[32] C. Pungila and V. Negru, *A Highly-Efficient Memory-Compression Approach for GPU-Accelerated Virus Signature Matching*, Information Security, (2012), pp. 354–369.
[33] Y. Qi, Z. Zhou, Y. Wu, Y. Xue, and J. Li, *Towards high-performance pattern matching on multi-core network processing platforms*, in Proceedings of the Global Communications Conference, 2010. GLOBECOM 2010, 6-10 December 2010, Miami, Florida, USA, 2010, pp. 1–5.
[34] L. Salmela, J. Tarhio, and J. Kytöjoki, *Multipattern String Matching with q-grams*, Journal of Experimental Algorithmics, 11 (2006), pp. 1–19.
[35] S. Sheik, S. Aggarwal, A. Poddar, B. Sathiyabhama, N. Balakrishna, and K. Sekar, *Analysis of String-searching Algorithms on Biological Sequence Databases*, Current Science, 89 (2005), pp. 368–374.
[36] Snort, *Webpage containing information on the snort intrusion prevention and detection system.* Website, 2010. http://www.snort.org/.
[37] G.-M. Tan, P. Liu, D.-B. Bu, and Y.-B. Liu, *Revisiting multiple pattern matching algorithms for multi-core architecture*, Journal of Computer Science and Technology, 26 (2011), pp. 866–874.
[38] T. Tran, M. Giraud, and J.-S. Varr, *Bit-parallel multiple pattern matching*, in Parallel Processing and Applied Mathematics, R. Wyrzykowski, J. Dongarra, K. Karczewski, and J. Waniewski, eds., vol. 7204 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2012, pp. 292–301.
[39] A. Tumeo, S. Secchi, and O. Villa, *Experiences with String Matching on the Fermi Architecture*, Architecture of Computing Systems-ARCS 2011, (2011), pp. 26–37.
[40] G. Vasiliadis, S. Antonatos, M. Polychronakis, E. Markatos, and S. Ioannidis, *Gnort: High Performance Network Intrusion Detection using Graphics Processors*, Proceedings of RAID, 5230 (2008), pp. 116–134.
[41] G. Vasiliadis and S. Ioannidis, *Gravity: A Massively Parallel Antivirus Engine*, in Recent Advances in Intrusion Detection, Springer, 2010, pp. 79–96.
[42] L. Vespa and N. Weng, *SWM: Simplified Wu-Manber for GPU-based Deep Packet Inspection*, in Proceedings of the 2012 International Conference on Security and Management, 2012.
[43] S. Wu and U. Manber, *Agrep - A Fast Approximate Pattern-Matching Tool*, In Proceedings of USENIX Technical Conference, (1992), pp. 153–162.
[44] ———, *A Fast Algorithm for Multi-pattern Searching*, (1994), pp. 1–11. Technical report TR-94-17.

[45] X. Zha and S. Sahni, *Multipattern String Matching on a GPU*, in Computers and Communications (ISCC), 2011 IEEE Symposium on, IEEE, 2011, pp. 277–282.
[46] Z. Zhou, Y. Xue, J. Liu, W. Zhang, and J. Li, *MDH: A High Speed Multi-phase Dynamic Hash String Matching Algorithm for Large-Scale Pattern Set*, Information and Communications Security, 4861 (2007), pp. 201–215.

Edited by: Marian Gusev
Received: Dec 21, 2014
Accepted: Mar 30, 2015