International Journal of Communication Networks and Information Security (IJCNIS) Vol. 10, No. 2, August 2018

A New Multi-threaded and Interleaving Approach to Enhance String Matching for Intrusion Detection Systems

Ali Shatnawi, Bushra AlHajouj and Moath Jarrah

Department of Computer Engineering, College of Computer and Information Technology
Jordan University of Science and Technology, Irbid, 22110, Jordan

Abstract: String matching algorithms are computationally intensive operations in computer science. The algorithms find the occurrences of one or more strings patterns in a larger string or text. String matching algorithms are important for network security, biomedical applications, Web search, and social networks. Nowadays, the high network speeds and large storage capacity put a high requirement on string matching methods to perform the task in a short time. Traditionally, Aho-Corasick algorithm, which is used to find the string matches, is executed sequentially. In this paper, a new multi-threaded and interleaving approach of Aho-Corasick using graphics processing units (GPUs) is designed and implemented to achieve high-speed string matching. Compute Unified Device Architecture (CUDA) programming language is used to implement the proposed parallel version. Experimental results show that our approach achieves more than 5X speedup over the sequential and other parallel implementations. Hence, a wide range of applications can benefit from our solution to perform string matching faster than ever before.

Keywords: network intrusion detection systems, string matching, pattern matching, Aho-Corasick, GPUs/CUDA.

1. Introduction

String matching algorithms form an important class of algorithms in computer science. They are critical in applications such as network intrusion detection systems [1, 2, 3, 4], information retrieval [5], text editing applications [6], searching for records with state values in database systems [7, 8], spell-checkers [9], DNA sequence mapping [10], virus scanning [11], IP (Internet Protocol) lookup [12], and protocol parsers [13]. Applications use either exact or approximate string matching. In exact string matching, the exact pattern must be found in the text; whereas approximate string matching allows some maximum number of mismatches, and this number of allowed mismatches depends on the nature of application. In network intrusion detection systems (IDS), patterns of known network attacks are saved in a database (signatures) [14]. The IDS examines the network traffic streams to find if any pattern match is detected which alarms for an attack or a threat. Spell-checkers and DNA sequence mapping are examples of applications that use approximate string matching [15, 16]. Hence, string matching is an important problem for many applications, and researchers are still proposing new algorithms and mechanisms to enhance its speed and accuracy [17].

The algorithms of string matching can be classified into three classes depending on the number of patterns that are used. The classes are: single pattern algorithms, finite set of pattern algorithms, and infinite number of pattern algorithms. This paper investigates the Aho-Corasick (AC) string matching algorithm which belongs to the class of finite set of pattern algorithms. The AC algorithm is a string searching algorithm invented by Alfred V. Aho and Margaret J. Corasick [18]. It is an extension of the Knuth-Morris-Pratt algorithm for a set of patterns to match all patterns simultaneously.

The AC algorithm has several advantages over others because of the following reasons. First, it finds a match for multiple patterns by examining every text character only once during the searching process [18]. Hence, AC has a linear time complexity. It is a linear function of the sum of the lengths of patterns in the construction of finite-state pattern-matching machine stage; and the total length of the text strings to be processed in the matching stage [19]. Second, AC algorithm is not affected by the number of patterns, the size of the shortest, or the size of longest pattern in a group [20]. The major disadvantage of the AC algorithm is the high memory cost needed to store the transition rules of the underlying deterministic finite automaton [21]. However, modern computers come with large enough memory that can easily fulfill the algorithm memory cost.

The era of big data has put a challenge on string matching where the execution time can reach to an undesirable value. In order to overcome this challenge, graphics processing units (GPU) can be used to speedup intensive computational tasks. A typical GPU has hundreds or thousands of processor cores with specialized pipelines for graphics processing. Nowadays, because of the highly parallel capabilities of GPUs, the notion of General Purpose Graphics Processing Units (GPGPU) is emerging. GPUs are used in literature to improve the execution of many different applications such as medical application [22], image segmentation [23], and video processing [24]. The authors in [25] have utilized the computational power of GPUs to generate very large random numbers per second. Because a GPU has massively parallel computing capabilities, and since AC algorithm is scalable, GPUs can be efficiently utilized to perform the string matching problem. In this paper, we have designed and implemented a GPU-based AC algorithm using CUDA. The proposed approach achieved a speedup of more than 5X over other existing methods.

This paper is organized as follows. The first section provides a background and related work. The second section discusses the Aho Corasick (AC) algorithm and its deterministic finite state pattern matching machine (tire). A subsection is dedicated to discuss our GPU approach of the AC implementation. The evaluation and experimental results are discussed in the third section. Finally, we conclude our paper.

2. Background and Related work

Many important applications require that the execution time of string matching to be short. Hence, GPU accelerators are used to achieve this goal. The researchers in [26] developed a GPU based parallel implementation of the AC algorithm. Their solution carefully places and caches the input data and the pattern in the on-chip shared memories of the GPU. They implemented the AC algorithm as a Deterministic Finite Automata (DFA). The DFA represents all possible states of the machine along with information about the acceptable state transitions of the system. Input data is copied to the global memory and the state transition table is copied to the texture memory. The effective texture memory latencies for the random access of the state transition table are reduced because of the texture cache. Their approach suffers from the problem that patterns occurring at the boundaries of adjacent segments cannot be detected, as depicted in Figure 1. To overcome this problem, our approach uses overlapping as shown in Figure 2. This solution uses portions of the state machine on different threads. In [10], Tumeo and Villa used a technique that is similar to the one used in [26] with addition of GPU and message passing (MP) features. In this technique, they divided the state machine among a number of GPUs and then compared the performance with a system that uses a cluster of homogeneous processors.

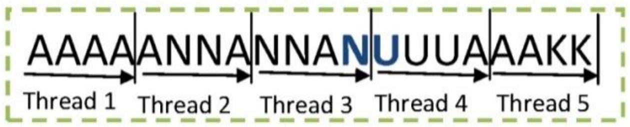

Figure 1. The boundary problem: pattern NU cannot be identified by Thread 3 or 4

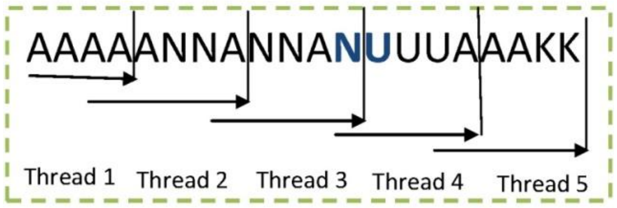

Figure 2. Thread overlapping: thread 4 can identify NU

Zha and Sahni in [27] have developed a GPU-based implementation of Aho-Corasick algorithm. They used DFA and divided the output array into blocks. A lockstep execution was used by all threads that are working on a single block. This is possible because there is no divergence in the execution paths of these threads. Each thread processes 16 characters at a time and writes the output results to the device memory. Asynchronous data transfer allows overlapping the time spent in data movements while the GPUs perform the computing tasks assigned to them. Kouzinopoulos, Assael, and Pyrgiotis in [28], presented a parallel implementation of the Aho Corasick and Wu Manber algorithms using NVIDIA CUDA and MPI on a hybrid GPU cluster. Their implementation was evaluated using a biological sequence dataset. A preprocessing phase is done on the host CPU and then the input string is split into chunks, where each chunk is assigned to a different node of the cluster. The input string and all preprocessing arrays in each node are copied to the global memory of the device. The input string is subsequently partitioned into chunks and then assigned to GPU threads. The utilization of the global memory bandwidth is increased by starting threads to read 16-byte words instead of a single character for each memory transaction. The researchers in [29] developed a GPU-based Parallel Failureless Aho-Corasick (PFAC) where they removed the *failure* function. Each byte in the input data is assigned to an individual thread that looks up into the state machine to find any match starting from that byte. A thread terminates when it does not find any valid next state transition. A MapReduce approach to speedup Myers algorithm for pattern matching was proposed in [30]. The researchers have integrated the MapReduce method in signature-based IDS. Oke and Vaidya in [31] used CPU and GPU to speed up Network Intrusion Detection Systems. They introduced an optimized PFAC that reduces the memory usage, time, and cost compared to the serial Aho-Corasick algorithm on a CPU. To reduce the latency of the global memory, they store the data in the texture memory. In addition, because the GPU cannot access data directly from a pageable memory of the CPU, CUDA must first allocate a temporary space for a page-locked memory and then copy the host data to the pinned memory array. A new model for IDS in a cloud environment has been proposed by the authors of [32]. The goal is to identify and detect attacks that occur in cloud computing environments. The work in [33] developed a CPU/GPU heterogeneous method for PFAC algorithm that was used for computational molecular biology applications. It uses a shared memory to store the transition table and to provide a segment of communication for the threads of each streaming processor. Hence, if a thread finds its next transition, it informs the next thread of the appropriate value so that the next thread continues its search. A thread terminates when that value is the same as its own starting position or when no valid next state transition is found. Other string matching methods and algorithms were proposed in the literature such as the work in [34], where the Smith-Waterman (SW) algorithm was implemented using a CPU/GPU heterogeneous system. The score matrix cells are computed in the coprocessor core. Then, the score matrix is divided into sub-matrices and each sub-matrix is stored in a memory space. A thread iteratively computes all cells that belong to a sub-matrix. The authors of [35] proposed a matching algorithm that works at the bit level to detect any anomaly in the network traffic. In addition, index-based searching algorithms such as suffix arrays and suffix trees are considered in research using GPUs to achieve higher performance [36]. For example, the suffix arrays technique outperforms the suffix trees technique, because it requires less number of memory accesses [37]. In [38] and [39], researchers used GPUs to produce seed generation for the FM-index. The search process is assigned to multiple processing units where each portion of the FM-index is allocated to one processing unit based on the input string size and the number of available GPUs.

3. Preliminary

3.1 Basic Aho-Corasick

The Aho Corasick algorithm constructs a deterministic finite state pattern matching machine (tire). An extra link between nodes is created to allow fast transitions between failed string matches without the need for backtracking. For example, a search for the word "door" in the tire that does not contain it, but contains the word "down", would fail at the node prefixed by the word "do". Given that the input string with size n is $S = s_0 s_1 \dots s_{n-1}$, the finite set of patterns with size k is $P = p_0 p_1 \dots p_{k-1}$, and each pattern pr is a string with size m, that is, $pr = pr_0 pr_1 \dots pr_{m-1}$. The alphabet size is denoted as $\sum$ and the size of all patterns is $|P|$. The problem is to find all occurrences of a pattern in the input string.

The AC algorithm consists of two phases: a preprocessing phase and a searching phase. The preprocessing phase constructs a trie data structure of pattern with a *goto* function, a *failure* function, and an *output* function. The *goto* function allows for the transition to the next state. The state in the trie is labeled as a prefix of a specific pattern $(pr \in P)$. $L(q)$ denotes the level of the path between the initial state and the state q. Hence, $L(q)$ is also a prefix of one of the patterns. Each pattern has a state q where $L(q)$ is equal to the pattern pr and the state is marked as a terminal state. An exact match is true when the search phase arrives at a terminal state. The *failure* function is used to visit a previous state when the *goto* function cannot find a transition from the current state to a next (child) state. The *goto* function is built during the construction of the trie. The trie is a depth-first traverse structure and uses a character of the patterns from the pattern set P for extension. For each extended process, the outgoing transitions are created in each state. The *failure link* is built for each state q when the longest suffix of $L(q)$ is also a prefix of any pattern in P. The start state usually is a 0 state. Figure 3 shows an example with 10 states from 0 to 9. The *goto* function g is responsible for mapping a pair of state and input symbol to another state or to the *failure* function f. The edge from state 0 to state 1 indicates that $g(0, a) = 1$, and the absence of an edge between states indicates that $g(0, Tc) = fail$; which means that each input symbol that is not b or e is a fail. At every machine cycle, one input symbol is processed by the machine. When the *goto* function reports a fail, the *failure* function f is consulted to map a state into another state. Specific states are designated as output states when patterns have been found and the *output* function formalized by using a set of patterns for each state. If a is a current state and s is the current character, then the pattern matching machine works as follows:

1. If $(a, s) = a'$, then a goto transition is done. It enters state a' and puts the next character of a as the current input character. If $output(a')$ is not empty and the operating cycle is not complete, the machine sends the set $output(a')$ with a position of the current input character.
2. If $(a, s) = fail$, the machine checks the *failure* function f to make a failure transition. If $f(a) = a'$, the cycle is repeated and a' is considered as the current state. The character s is marked as the current character.

The initial phase of the pattern matching machine is to set the current state of the machine as the start state, and the first character of the input text string as the current input character. Each character of the input text string is processed by making one operating cycle.

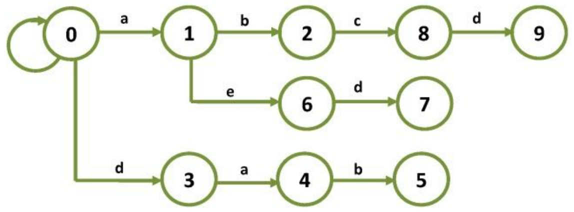

Figure 3. The pattern matching machine: *goto* function

i	1	2	3	4	5	6	7	8	9
f(i)	0	0	0	1	2	0	3	0	3

Figure 4. The pattern matching machine: *failure* function

i	Output(i)
2	{ab}
5	{dab,ab}
7	{aed}
9	{abcd}

Figure 5. The pattern matching machine: *output* function

Algorithm 1 shows the behavior of a pattern matching machine, where each iteration of the for-loop represents one cycle. The *goto*, *failure* and *output* functions are constructed in two phases (lines 5-8). In the first phase, the states and the *goto* function are constructed using a *goto* graph that consists of one vertex representing state 0. In the second phase the *failure* function is computed, while the *output* function spans both the first and second phases. The patterns expand the graph by adding directed edges starting from the initial state, and adding new vertices to the graph. Each path in the graph represents a pattern. A pattern is added to the *output* function of the state at which the path terminates (line 10, 11).

The *failure* function is constructed from the *goto* function. The depth of each state is defined as the length of the shortest path from the start state to that state. For example, in Figure 3 the depth of start state is 0, state 3 is 1, and so on. The *failure* function is computed for all states sequentially starting from all states with depth 1, then all states with depth 2. Computing the *failure* function starts by assigning $f(s) = 0$, where s is any state of depth 1. After computing f for all states of depth less than d, it is computed for the states of depth *d*. The states of depth d can be determined from the non-fail values of the *goto* function of the states of depth $d - 1$. The *failure* function for the states of depth d is calculated as follows:

1. If $g(r, a) = fail$, for all a, and r is a state of depth $d - 1$, stop.
   a. If $g(r, a) = s$, where r is a state of depth $d - 1$, $set\ ate = f(r)$.
   b. Execute for loop $(state \leftarrow f(state))$ until $g(state) \neq fail$.
   c. Set $f(s) = g(state, a)$.

Algorithm 1 Pattern matching machine

```
input: S, g, f, out /* A text string $S = s_0 s_1 .... s_{n-1}$ , $s_i$ is an input character for the pattern matching machine M, g is the goto function, f is the failure function, and out is the output function */
output: Locations at which patterns occur in $S$.
begin
    $state \leftarrow 0$
    for $i = 1$ to $n$
    do
        while $g(state, s_i) = fail$
        do
            $state \leftarrow f(state)$
        end
        $state \leftarrow g(state, s_i)$
        if $output(state) \neq empty$ then
            $result(i) = output(state)$
        end
    end
end
```

As an example, in Figure 3, $f(1) = f(3) = 0$; since the depth of these states is 1. To compute $f(2)$ of depth equal to 2, set $state = f(1) = 0$; and $f(2) = 0$ because $g(0, b) = 0$. Then, $f(4)$ which has a depth of 2, set $state = f(3) = 0$; and $f(4) = 1$, because $g(0, a) = 1$. This procedure is repeated for all states to obtain the *failure* function as shown in Figure 4. While computing the *failure* function, the *output* function gets updated. When $f(s) = s'$, the outputs of state s is merged with the output of state s'. For example, in Figure 3, $f(5) = 2$. Hence, the output set of state 2 which is {"ab"} is combined with the output set of state 5 producing a new output set which is {"ab", "dab"}. Figure 5 shows the *output* function.

3.2 The proposed GPU Methodology

We have designed a highly optimized implementation of parallel AC (PAC) pattern matching kernel using CUDA. By using various optimization strategies, we were able to increase the throughput of the string matching machine and overcome the boundary problem without the need for overlapping. In addition, the lifetime of threads is much shorter than the time needed for the straightforward implementation. Our implementation consists of two phases. The first phase is a preprocessing phase that is done by the CPU. The second phase is a stretching phase that is deployed on the GPU device. Algorithm 2 summarizes the CPU phase which is a data preparation step.
The function *buildmatchingmachine* (line 5) in Algorithm 2 reads the input patterns and constructs the *goto*, *failure*, and *output* functions. Algorithms 3 and 4 show the steps of generating the aforementioned three functions. Experimental results show that a chunk size of $10k$ gives the best performance (Algorithm 2, line 6). All data are exported to the GPU memory along with the results of execution of the function *buildmatchingmachine* (Algorithm 2, line 10-15). Three arrays are computed that represent the following: a two dimensional array (g[][]) which represents the *goto* function, a one dimensional array (f[]) that represents the *failure* function, and a one dimensional array (out[]) that represents the *output* function. In order to transfer the three arrays and the input string to the GPU, GPU buffers are allocated using the CUDA-specific APIs cudaMalloc() and cudaMemcpy(). After that, the second phase starts by executing a parallel search on the GPU. One kernel thread searches for each keyword/pattern.

Algorithm 2 CPU version

```
input: file, keywords/* An input string (file), set of patterns */
output: string buffer ($str$), goto table ($g$), failure table ($f$), out table ($out$).
begin
    for $i = 1$ to $k$
    do
        enter(Keyword )
        $buildmatchingmachine(Keyword)$
        $chunk \leftarrow 10k$
        $numofchunk \leftarrow ((size\ of\ file) \div chunk)$
        for $i = 1$ to $numofchunk$
        do
            $str \leftarrow file[chunk]$
            $dev\_str \leftarrow str$
            $dev\_g \leftarrow g$
            $dev\_f \leftarrow f$
            $dev\_out \leftarrow out$
            $result \leftarrow kernel(dev\_str, dev\_g, dev\_f, dev\_out)$
        end
    end
end
```

Algorithm 3 *goto* function construction.

```
input: $P$ /* Set of patterns $P = p_0 p_1 .... p_{k-1}$ */
output: partially computation of the output function
begin
    $newstate \leftarrow 0$
    for $i = 1$ to $k$
        do
            enter($p_i$ )
        end
        for all $s$ such that $g(0, s) = fail$ do
            $g(0, s) \leftarrow 0$
        end
        procedure enter($s_0 s_1 \dots s_{n-1}$):
        state ← 0
        j ←1
        while $g(state, sj) \neq fail$ do
            $state \leftarrow g(state, s)$
            $j \leftarrow j + l$
        end
        for $q = j$ to $m$
        do
            $newstate \leftarrow newstate + 1$
            $g(state, sq) \leftarrow newstate$
            $state \leftarrow newstate$
        end
        $output(state) \leftarrow \{s_1 s_2 .... s_m\}$
end
```

Algorithm 4 *failure* function construction

```
input: the output from Algorithm 2 (g function and output function).
output: failure function (f) and output function (output).
begin
    $queue \leftarrow empty$
    if $g(0,s) = a \neq 0$ do
      then
        $queue \leftarrow queue \cup \{a\}$
        $f(a) \leftarrow 0$
      end
    while $queue \neq empty$
      do
        $r \leftarrow the\ next\ state\ in\ queue$
        $queue \leftarrow queue - \{r\}$
        if $g(r,s) = a \neq fail$
          then
            $queue \leftarrow queue \cup \{a\}$
            $state \leftarrow f(r)$
            while $g(state,s) = fail$
            do
              $state \leftarrow f(state)$
              $f(a) \leftarrow g(state,s)$
$output(a) \leftarrow (output(a) \cup$
              $output(f(a)))$
            end
          end
      end
end
```

Algorithm 5 GPU version

```
input: $dev\_str, dev\_g, dev\_f, dev\_out, patt\_size$ /*An input chunk ($dev\_str$), the arrays ($dev\_g$),( $dev\_f$)( $dev\_out$), pattern size ($patt\_size$) */
output: Locations at which patterns occur in str.
begin
    $th \leftarrow threaded$
    $currentstate \leftarrow 0$
    for $i = th$ to $(th + k)$
      do
        $currentstate \leftarrow g(currentstate, dev\_str[i])$
        for $j = 0$ to $patt\_size$ do
          if $currentstate = 0$
          then
              break
          end
          else if $currentstate \neq 0$
            then
              $result[i] \leftarrow +1$
          end
        end
      end
end
```

3.3 Thread assignment methodology

In order to improve the string matching performance, we have parallelized the searching phase of the algorithm by allocating each character of an input stream to a CUDA thread. This is feasible because the used GPU hardware is powerful and the lifetime of the threads in our implementation is very short.

The threads searching tasks are as follows. Figure 6 shows the AC state machine that identifies the patterns “ab”, “abcd”, “dab”, and “aed”. If the input stream contains the substring “dabcd”, the thread "tn" is assigned to the input d and traverses the AC state machine. When processing the input “dab”, thread "tn" reaches state 5 which indicates that the pattern “dab” is found. Since there is no valid transition for c in state 5, thread t_n terminates at state 5. Similarly, thread t_{n+1} is assigned to the input a. When processing the input “abcd”, thread t_{n+1} reaches state 9 which indicates that the pattern “abcd” is found. As shown in Figure 6, all threads ($t_0, t_1, \ldots, t_{n+2}, t_{n+3}, t_{n+5}$) terminate early at state 0 because there are no valid transitions for x, b, and c in state 0. Thread t_{n+4} terminates early at state 3 without a valid transition for x in state 3.

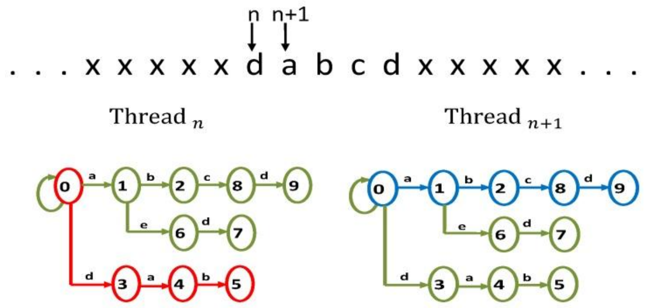

Figure 6. Parallel AC algorithm allocation

3.4 Optimization of Data Transfers

The GPU threads cannot directly access the data in a pageable host memory. Hence, it must first allocate a pinned host array to copy the data. Subsequently, the data is transferred from the pinned array to the device memory as shown in Figure 7 (left). In order to improve data transfers, our implementation allocates the host arrays into a pinned memory to avoid the cost of the transfer between pageable and pinned host arrays, as shown in Figure 7. Using pinned data transfers allows us to exploit the high computing capability of the GPU device using the overlap feature. In order to interleave the data transfers and the computations on the device, we used the CUDA asynchronous function *cudaMemcpyAsync()* instead of the synchronous function *cudaMemcpy()*.

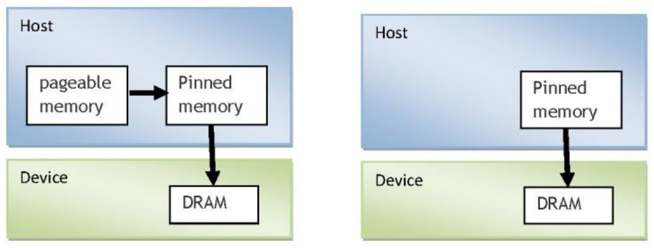

Figure 7. Pageable and pinned data transfers

3.5 Threads in a Group

Our implementation divides the input data into chunks of size 10×2^{10} characters. A choice of having 10×2^{10} threads to process each chunk in parallel seems to be preferable so that each thread works on one character. However, our experimental results and other research efforts such as in [39] show that a high performance is achieved by setting the number of threads on each block to be 256 threads. This is because managing many threads could result in performance degradation in terms of the overall execution time. The threads of a block are identified using a two-dimensional structure. The GPU architecture is a 32 bit-wide SIMT (Single Instruction Multiple Threads). Hence, the number of threads in a block must be a multiple of 32. These threads are organized as 32 in the $x - axis$ and 8 in the $y - axis$. Thus, the number of blocks is calculated by dividing the chunk size by 256.

3.6 Kernel Overlaps Execution

The concurrent transfer and execution capability of the GPU drive us to dispatch more than one kernel concurrently in order to achieve high performance improvement. Hence, two kernels are dispatched and overlapped in execution for multiple input chunks as shown in Figure 8. Both kernels perform asynchronous data transfer and execution for each chunk on the GPU.

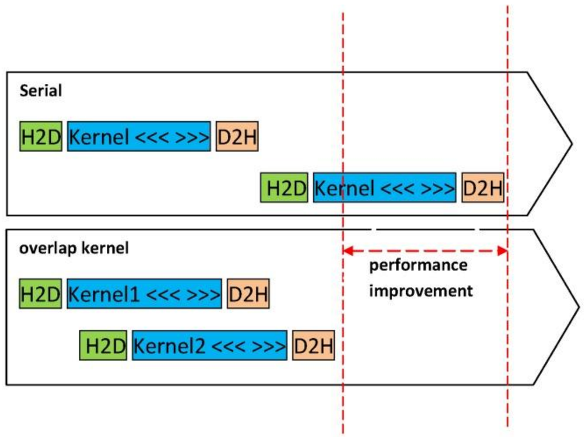

Figure 8. Synchronous and asynchronous kernel execution

The main disadvantage of the AC is the high memory cost. It requires a large space to store the tables when the patterns and the input text contain human-readable alphabets, numbers, and signs. In order to overcome this problem, we have converted the patterns and the input data into hexadecimal representation.

The main disadvantage of the AC is the high memory cost. It requires a large space to store the tables when the patterns and the input text contain human-readable alphabets, numbers, and signs. In order to overcome this problem, we have converted the patterns and the input data into Hexadecimal representation.

4. Evaluation and Experimental Results

This section compares and evaluates the performance of our proposed method against the traditional sequential CPU approach and the hybrid parallel implementation of the Aho-Corasick proposed in [28]. All experiments were performed on a system with the following specifications:

- Host machine: Intel Core2 i5-4460 CPU, four CPU cores at 3.20GHz, 6 MB Cache, main memory is 8 GB DDR3.
- GPU device machine: 5.2 NVIDIA GeForce GTX 960 1279 MHz with 1024 CUDA cores and 2GB GDDR5 memory.
- Patterns: Snort v2.9 [40].

Different trace files were used in the experiments as in the work by Aldwairi and Alansari [41]. The files contain different numbers of intrusions. Wireshark Network Protocol Analyzer is used to extract packet traces as in [42]. We used a C++ code to extract the contents from the database of Snort rules.

CUDA version 7.5 was used to implement the parallel version of the algorithm. The following two metrics were used to evaluate the performance of the three methods:

1- Execution time: This represents the kernel searching process. The preprocessing phase is excluded from the execution time because it is computed only once. The three implementations share the *I/O* operations and hence it is also excluded from the execution time.

2- Speedup: This represents the amount of improvement of one method compared to another. The speedup of our parallel implementation is compared with the CPU version using:

$$Speedup = \frac{execution\ time\ of\ serial}{execution\ time\ of\ parallel}$$

Figure 9 shows the execution time of the three methods which are: our GPU implementation, the CPU serial, and the parallel hybrid version. The experiments were performed using the good (av, gd, hm, lc), the bad (1, 22) and the ugly (51, 58) traces. Ugly traces are the worst-case scenario in terms of intrusions while the good traces are the best or the norm scenario. The results show that our implementation outperforms the other two methods for all trace files. The hybrid parallel implementation of [28] has the worst performance. The reason for that is that intrusion detection systems (IDSs) work on small input data especially those that process the data on-the-fly as they receive data (stream traffic). In order to prove that, we have varied the data chunk size and calculated the execution time as shown in Figure 10. In the figure, the hybrid method outperforms the CPU method when the size becomes more than $10K$. However, it was shown that it is slower than our method in all cases. Figure 11 shows the speedup for the GPU implementations over the CPU and hybrid implementations using the six trace files. The speedup is always greater than one.

5. Conclusion

String matching problem is an important step for the success of many applications in areas such as network intrusion detection systems, information retrieval, text editing applications, searching in database systems, spell-checkers, DNA sequence mapping, virus scanning, IP (Internet Protocol), and protocol parsers. This paper has addressed the Aho-Corasick (AC) string matching algorithm. The AC algorithm has several advantages over other algorithms in terms of the time complexity. Also, it finds the match for multiple patterns with one pass over the text. In addition, the AC algorithm is not affected by the number of patterns, the size of the shortest, or the size of longest pattern in a group. The aforementioned advantages make the AC a preferred choice by different applications. The paper has proposed a new GPU-based parallel implementation to improve the execution time of the AC algorithm. GPU hardware has been used to execute the parallel implementation and compared against the sequential CPU.

In addition, a comparison of the proposed GPU method against other parallel implementations is given. We have used patterns from snort and network traffic in conducting our experiments. Experimental results show that our method outperformed the CPU and the other parallel implementations. We have achieved a speedup of up to 5X. Finally, we have presented different optimization steps in order to enhance the execution time of the AC algorithm.

References

[1] Y.H. Kim, W.H. Park, “A study on cyber threat prediction based on intrusion detection event for apt attack detection,“ Multimedia Tools and Applications, Vol. 71, No. 2, pp. 685-698, 2014.

[2] R.T. Liu, N.F. Huang, C.H. Chen, C.N. Kao, “A fast string-matching algorithm for network processor-based intrusion detection system,” ACM Transactions on Embedded Computing Systems, Vol. 3, No. 3, pp. 614-633, 2004.

[3] N. Wattanapongsakorn, C. Charnsripinyo, “Web-based monitoring approach for network-based intrusion detection and prevention,” Multimedia Tools and Applications, Vol. 74, No. 16, pp. 639-641, 2015.

[4] C. Yin, L. Ma, L. Feng, “Towards accurate intrusion detection based on improved clonal selection algorithm,” Multimedia Tools and Applications, Vol. 76, No. 19, pp. 19397-19410, 2017.

[5] J. Zobel, P. Dart, “Phonetic string matching: Lessons from information retrieval,” The Nineteenth annual international ACM SIGIR conference on Research and Development in Information Retrieval, Switzerland, pp. 166-172, 1996.

[6] K. Kukich, “Techniques for automatically correcting words in text,” ACM Computing Surveys, Vol. 24, No. 4, pp. 377-439, 1992.

[7] W.T. Balke, U. GÜntzer, “Multi-objective query processing for database systems,” The Thirtieth International Conference on Very Large Data Bases VLDB Endowment, Canada, pp. 936-947, 2004.

[8] D. Brown, P. Beckingham, “System and methods for searching and matching databases,” US Patent no. 6,026,398, 2000.

[9] P.O. Kristensson, S. Zhai, “Relaxing stylus typing precision by geometric pattern matching,” The Tenth International Conference on Intelligent User Interfaces, USA, pp. 151-158, 2005.

[10] A. Tumeo, O. Villa, “Accelerating DNA analysis applications on GPU clusters,“ The Eighth IEEE Symposium on Application Specific Processors, USA, pp. 71-76, 2010.

[11] X. Zhou, B. Xu, Y. Qi, J. Li, “MRSI: A fast pattern matching algorithm for anti-virus applications,” The Seventh International Conference on Networking, Mexico, pp. 256-261, 2008.

[12] B. Lampson, V. Srinivasan, G. Varghese, “IP lookups using multiway and multicolumn search,” IEEE/ACM Transactions on Networking (TON), Vol. 7, No. 3, pp. 324-334, 1999.

[13] R.D. Graham, P.C. Johnson, “Finite state machine parsing for internet protocols: faster than you think,” Security and Privacy Workshops, USA, pp. 185-190, 2014.

[14] P. Garcia-Teodoro, J. Diaz-Verdejo, G. Maciá-Fernández, E. Vázquez, “Anomaly-based network intrusion detection: techniques, systems and challenges,” computers & security, Vol. 28, No. 1, pp. 18-28, 2009.

[15] L.L. Cheng, D.W. Cheung, S.M. Yiu, “Approximate string matching in DNA sequences,” The Eighth International Conference on Database Systems for Advanced Applications, Japan, pp. 303-310, 2003.

[16] D. Gusfield, “Algorithms on strings, trees and sequences: computer science and computational biology,” Cambridge University press, 1997.

[17] L. Zhao, K. Zhou, J. Guo, S. Wang, T. Lin, “A universal string matching approach to screen content coding,” IEEE Transactions on Multimedia, Vol. 20, No. 4, pp. 796-809, 2018.

[18] A.V. Aho, M.J. Corasick, “Efficient string matching: an aid to bibliographic search,” Communications of the ACM, Vol. 18, No. 6, pp. 333-340, 1975.

[19] S. Vidanagamachchi, S. Dewasurendra, R.G. Ragel, “Hardware software co-design of the Aho-Corasick algorithm: Scalable for protein identification?,” The Eighth IEEE International Conference on Industrial and Information Systems, Sri Lanka, pp. 321-325, 2013.

[20] M. Norton, “Optimizing pattern matching for intrusion detection,” Sourcefire, Inc., Columbia, 2004.

[21] V. Bhardwaj, V. Garg, “A comparative study of Wu-Manber string matching algorithm and its variations,” International Journal of Computer Applications, Vol. 132, No. 17, pp. 34-38, 2015.

[22] Y. Jararweh, M. Jarrah, S. Hariri, “Exploiting GPUs for compute-intensive medical applications,” International Conference on Multimedia Computing and Systems, Morocco, pp. 29-34, 2012.

[23] M. Shehab, M. Al-Ayyoub, Y. Jararweh, M. Jarrah, “Accelerating compute-intensive image segmentation algorithms using GPUs,” The Journal of Supercomputing, Vol. 73, No. 5, pp. 1929-1951, 2017.

[24] L. Deligiannidis, H.R. Arabnia, “Parallel video processing techniques for surveillance applications,” International Conference on Computational Science and Computational Intelligence, USA, pp. 183-189, 2014.

[25] M.A.S. Al-khatib, A.H. Lone, “Acoustic lightweight pseudo random number generator based on cryptographically secure LFSR,” International journal of Communication Networks and Information Security, Vol. 10, No. 2, pp. 38-45, 2018.

[26] N.P. Tran, M. Lee, S. Hong, J. Choi, “High throughput parallel implementation of Aho-Corasick algorithm on a

GPU," Parallel and Distributed Processing Symposium Workshops & PhD Forum, USA,pp. 1807-1816, 2013.

[27] X. Zha, S. Sahni, "Multipattern string matching on a GPU," IEEE Symposium on Computers and Communications, Greece, pp. 277-282, 2011.

[28] C.S. Kouzinopoulos, J.A.M. Assael, T.K. Pyrgiotis, K.G. Margaritis, "A hybrid parallel implementation of the Aho-Corasick and Wu-Manber algorithms using NVIDIA CUDA and MPI evaluated on a biological sequence database," arXiv:1407.2889, 2014.

[29] C.H. Lin, C.H. Liu, L.S Chien, S.C. Chang, "Accelerating pattern matching using a novel parallel algorithm on GPUs," IEEE Transactions on Computers Vol. 62, No. 10, pp. 1906-1916, 2013.

[30] M. Aldwairi, A.M. Abu-Dalo, M. Jarrah, "Pattern matching of signature-based IDS using Myers algorithm under MapReduce framework," EURASIP Journal on Information Security, Vol. 2017, No. 1, pp. 9-19, 2017.

[31] P.S. Oke, A.S. Vaidya, "Optimization of parallel Aho-Corasick multipattern matching algorithm on GPU," Optimization, Vol. 3, No. 6, pp. 5191-5200, 2015.

[32] A. Ahmad, N.B. Idris, M.N. Kama, "CloudIDS: Cloud intrusion detection model inspired by Dendritic cell mechanism," International journal of Communication Networks and Information Security, Vol. 9, No. 1, pp. 67-75, 2017.

[33] S. Soroushnia, M. Daneshtalab, J. Plosila, T. Pahikkala, P. Liljeberg, "High performance pattern matching on heterogeneous platform," Journal of Integrative Bioinformatics, Vol. 11, No. 3, pp. 88-98, 2014.

[34] L. Ligowski, W. Rudnicki, "An efficient implementation of Smith Waterman algorithm on GPU using CUDA for massively parallel scanning of sequence databases," IEEE International Symposium on Parallel & Distributed Processing, Italy, pp.1-8, 2009.

[35] M. Zeeshan, H. Javed, S. Ullah, "Discrete R-contiguous bit matching mechanism appropriateness for anomaly detection in wireless sensor networks," International journal of Communication Networks and Information Security, Vol. 9, No. 2, pp. 157-163, 2017.

[36] G. Encarnação, N. Sebastião, N. Roma, "Advantages and GPU implementation of high performance indexed DNA search based on suffix arrays," International Conference on High Performance Computing and Simulation, Turkey, pp. 49-55, 2011.

[37] Y. Liu, B. Schmidt, "Evaluation of GPU-based seed generation for computational genomics using Burrows-Wheeler transform," Parallel and Distributed Processing Symposium Workshops & PhD Forum, China, pp. 684-690, 2012.

[38] D. Zhang, Y. Zhang, S. Liu, X. Huang, "Parallelization of FM-index," The Tenth IEEE International Conference on High Performance Computing and Communications, China, pp. 169-173, 2008.

[39] S. Cook, "CUDA programming: a developer's guide to parallel computing with GPUs," Newnes, 2012.

[40] Snort, "Snort rules", Available in URL http://www.snort.org, May, 2018.

[41] M. Aldwairi, D. Alansari, "Exscind: Fast pattern matching for intrusion detection using exclusion and inclusion filters," The Seventh International Conference on Next Generation Web Services Practices (NWeSP), Spain, pp. 24-30, 2011.

[42] Wireshark, "Wireshark v1.4.4," Available in URL http://www.wireshark.org, May, 2018.

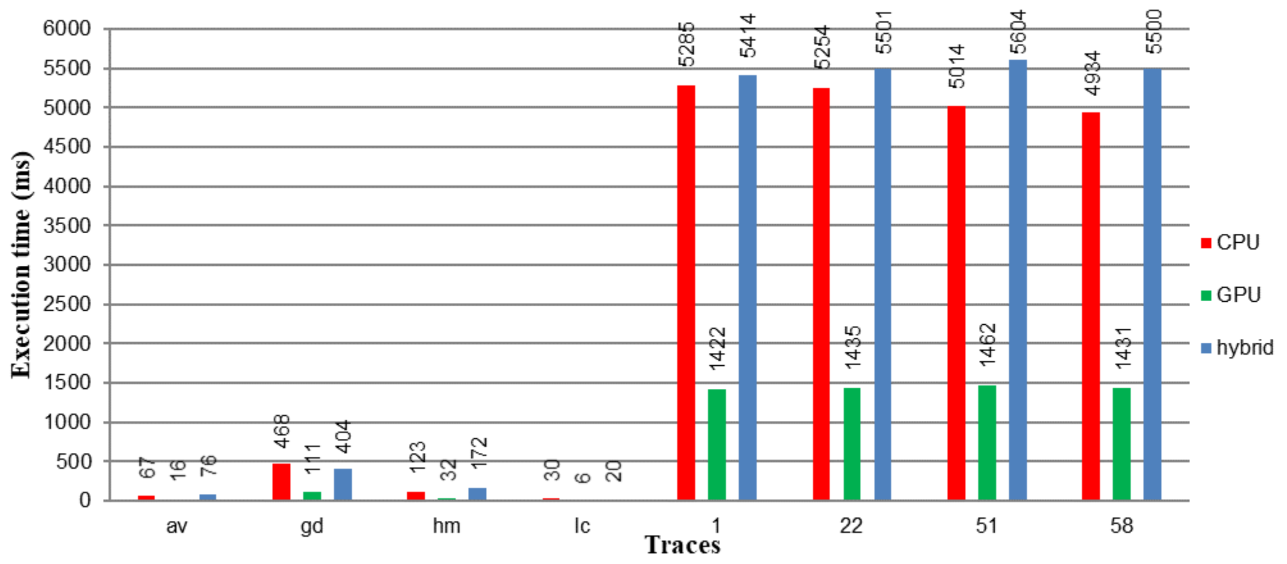

Figure 9. Execution time in milliseconds

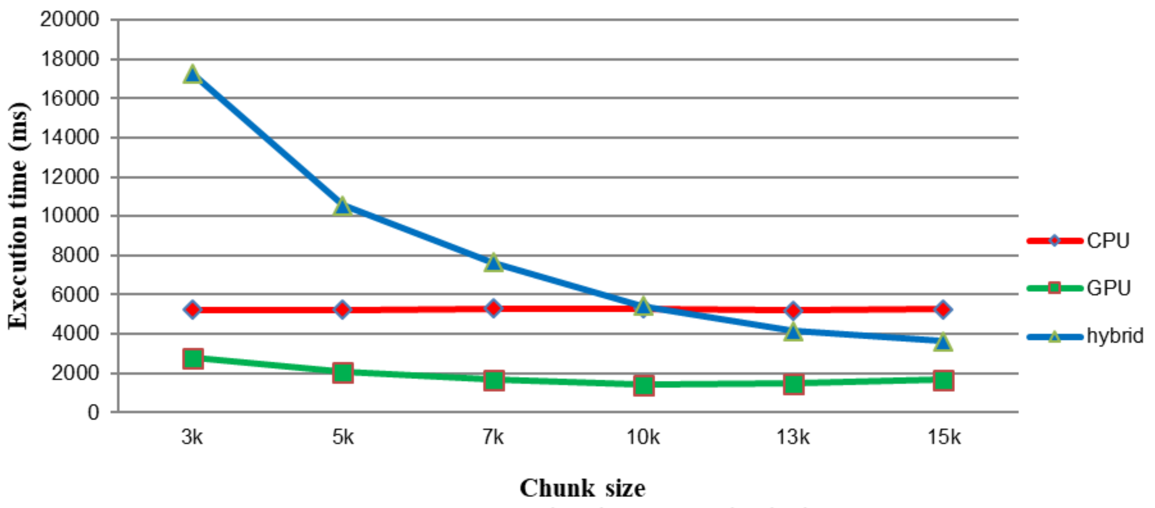

Figure 10. Execution time versus chunk size

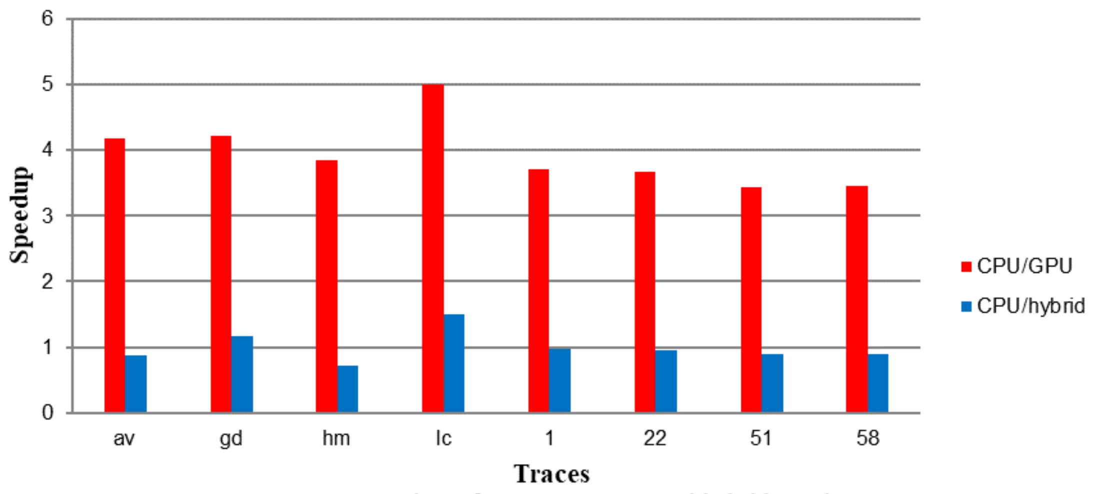

Figure 11. Speedup of GPU over CPU and hybrid versions