Accelerating String Matching Using Multi-threaded Algorithm on GPU


*National Taiwan Normal University, Taipei, Taiwan
**Dept. of Computer Science, National Tsing Hua University, Hsinchu, Taiwan

Abstract—Network Intrusion Detection System has been widely used to protect computer systems from network attacks. Due to the ever-increasing number of attacks and network complexity, traditional software approaches on uni-processors have become inadequate for the current high-speed network. In this paper, we propose a novel parallel algorithm to speedup string matching performed on GPUs. We also innovate new state machine for string matching, the state machine of which is more suitable to be performed on GPU. We have also described several speedup techniques considering special architecture properties of GPU. The experimental results demonstrate the new algorithm on GPUs achieves up to 4,000 times speedup compared to the AC algorithm on CPU. Compared to other GPU approaches, the new algorithm achieves 3 times faster with significant improvement on memory efficiency. Furthermore, because the new Algorithm reduces the complexity of the Aho-Corasick algorithm, the new algorithm also improves on memory requirements.

Keywords—string matching, graphics processing unit

I. INTRODUCTION

Network Intrusion Detection Systems (NIDS) have been widely used to protect computer systems from network attacks such as denial of service attacks, port scans, or malware. The string matching engine used to identify network attacks by inspecting packet content against thousands of predefined patterns dominates the performance of an NIDS. Due to the ever-increasing number of attacks and network complexity, traditional string matching approaches on uni-processors have become inadequate for the high-speed network.

To accelerate string matching, many hardware approaches are being proposed that can be classified into logic-based [1][2][3][4] and memory-based approaches [5][6][7][8][9]. Recently, Graphic Processor Unit (GPU) has attracted a lot of attention due to their cost-effective parallel computing power. A modified Wu-Manber algorithm [10] and a modified suffix tree algorithm [11] are implemented on GPU to accelerate exact string matching while a traditional DFA approach [12] and a new state machine XFA [13] are proposed to accelerate regular expression matching on GPU.

In this paper, we study the use of parallel computation on GPUs for accelerating string matching. A direct implementation of parallel computation on GPUs is to divide an input stream into multiple segments, each of which is processed by a parallel thread for string matching. For example in Fig. 1(a), using a single thread to find the pattern “AB” takes 24 cycles. If we divide an input stream into four segments and allocate each segment a thread to find the pattern “AB” simultaneously, the fourth thread only takes six cycles to find the same pattern as shown in Fig. 1(b).

However, the direct implementation of dividing an input stream on GPUs cannot detect a pattern occurring in the boundary of adjacent segments. We call the new problem as the “boundary detection” problem. For example, in Fig. 2, the pattern “AB” occurs in the boundary of segments 3 and 4 and cannot be identified by threads 3 and 4. Despite the fact that boundary detection problems can be resolved by having threads to process overlapped computation on the boundaries (as shown in Fig. 3), the overhead of overlapped computation seriously degrades performance.

In this paper, we attempt to speedup string matching using GPU. Our major contributions are summarized as follows.

1. We propose a novel parallel algorithm to speedup string matching performed on GPUs. The new parallel
algorithm is free from the boundary problem.

2. We also innovate new state machine for string matching, the state machine of which is more suitable to be performed in the parallel algorithm.

3. Considering special architecture properties of GPU, we have also described several speedup techniques.

4. Finally, we perform experiments on the Snort rules. The experimental results show that the new algorithm on GPU achieves up to 4,000 times speedup compared to the AC algorithm on CPU. Compared to other GPU approaches, the new algorithm achieves 3 times faster with significant improvement on memory efficiency. In addition, because the new Algorithm reduces the complexity of the Aho-Corasick (AC) algorithm, we achieve an average of 21% memory reduction for all string patterns of Snort V2.4.

II. PROBLEMS OF DIRECT IMPLEMENTATION OF AC ALGORITHM ON GPU

Among string matching algorithms, the AC algorithm [5][8][9][14][16][17] has been widely used for string matching due to its advantage of matching multiple patterns in a single pass. Using the AC Algorithm for string matching consists of two steps. The first step is to compile multiple patterns into a composite state machine. The second step is to use a single thread to recognize attack patterns by traversing the state machine. For example, Fig. 4 shows the state machine of the four patterns, “AB”, “ABG”, “BEDE”, and “EF”. In Fig. 4, the solid lines represent the valid transitions whereas the dotted lines represent the failure transitions.

The failure transitions are used to back-track the state machine to recognize patterns in different locations. Given a current state and an input character, the AC machine first checks whether there is a valid transition for the input character; otherwise, the machine jumps to the next state where the failure transition points. Then, the machine regards the same input character until the character causes a valid transition. For example, consider an input stream which contains a substring “ABDE”. The AC state machine first traverses from state 0, state 1, to state 2 which is the final state of pattern “AB”. Because state 2 has no valid transition for the input “E,” the AC state machine first takes a failure transition to state 4 and then regards the same input “E” leading to state 5. Finally, the AC state machine reaches state 7 which is the final state of pattern “BEDE”.

One approach to increase the throughput of string matching is to increase the parallelism of the AC algorithm. A direct implementation to increase the parallelism is to divide an input stream into multiple segments and to allocate each segment a thread to process string matching. As Fig. 5 shows, all threads process string matching on their own segments by traversing the same AC state machine simultaneously. As discussed in introduction, the direct implementation incurs the boundary detection problem. To resolve the boundary detection problem, each thread must scan across the boundary to recognize the pattern that occurs in the boundary. In other words, in order to resolve the boundary detection problem and identify all possible patterns, each thread must scan for a minimum length which is almost equal to the segment length plus the longest pattern length of an AC state machine. For example in Fig. 5, suppose each segment has eight characters and the longest pattern of the AC state machine has four characters, each thread must scan a minimum length of eleven (8+4-1) characters to identify all possible patterns. The minimum length is calculated by adding the segment length and the length of the longest pattern, and then subtracting one character. The overhead caused by scanning the additional length across the boundary is so-called overlapped computation.

On the other hand, the throughput of string matching on GPU can be improved by deeply partitioning an input stream and increasing threads. However, deeply partitioning will cause the probability of the boundary detection problem to increase. To resolve the boundary detection problem, the overlapped computation increases tremendously and leads to throughput bottleneck.

III. PARALLEL FAILURELESS-AC ALGORITHM

In order to increase the throughput of string matching on GPU and resolve the throughput bottleneck caused by the overlapped computation, we propose a new algorithm, called Parallel Failureless-AC Algorithm (PFAC). In PFAC, we allocate each byte of an input stream a GPU thread to identify any virus pattern starting at the thread starting location.

The idea of allocating each byte of an input stream a thread to identify any virus pattern starting at the thread starting location has an important implication on the efficiency. First, in the conventional AC state machine, the failure transitions are used to back-track the state machine to identify the virus patterns starting at any location of an input stream. Since in the PFAC algorithm, a GPU thread only concerns the virus pattern starting at a particular location, the GPU threads of
PFAC need not back-track the state machine. Therefore, the failure transitions of the AC state machine can all be removed. An AC state machine with all its failure transitions removed is called Failureless-AC state machine. Fig. 6 shows the diagram of the PFAC which allocates each byte of an input stream a thread to traverse the new Failureless-AC state machine.

![Figure 6. Parallel Failureless-AC algorithm which allocates each byte of an input stream a thread to traverse the Failureless-AC state machine.](image)

We now use an example to illustrate the PFAC algorithm. Fig. 7 shows the Failureless-AC state machine to identify the patterns “AB”, “ABG”, “BEDE”, and “EF” where all failure transitions are removed. Consider an input stream which contains a substring “ABEDE”. As shown in Fig. 8, the thread \( t_n \) is allocated to input “A” to traverse the Failureless-AC state machine. After taking the input “AB”, thread \( t_n \) reaches state 2, which indicates pattern “AB” is matched. Because there is no valid transition for “E” in state 2, thread \( t_n \) terminates at state 2. Similarly, thread \( t_{n+1} \) is allocated to input “B”. After taking input “BEDE”, thread \( t_{n+1} \) reaches state 7 which indicates pattern “BEDE” is matched.

![Figure 7. Failureless-AC state machine of the patterns “AB”, “ABG”, “BEDE”, and “EF”.](image)

![Figure 8. Example of PFAC](image)

There are three reasons that the PFAC algorithm is superior to the straightforward implementation in Section II. They are described as follows. First, there is no boundary detection problem, as with the straightforward implementation. Second, both the worst-case and average life times of threads in the PFAC algorithm are much shorter than the time needed for the straightforward implementation. As shown in Fig. 9, threads \( t_n \) to \( t_{n+3} \) terminate early at state 0 because there are no valid transitions for “X” in state 0. The threads \( t_{n+6} \) and \( t_{n+8} \) terminate early at state 8 because there are no valid transitions for “D” and “X” in state 8. Although the PFAC algorithm allocates a large number of threads, most threads have a high probability of terminating early, and both the worst-case and average life-time of threads in the PFAC algorithm are much shorter than the direct implementation. Third, the memory usage of the PFAC algorithm is smaller, due to the removal of failure transitions.

![Figure 9. Most threads terminate early in PFAC](image)

IV. GPU IMPLEMENTATION

We adopt the Compute Unified Device Architecture (CUDA)[19] proposed by NVIDIA[20] for GPU implementation. In this section, we will discuss several issues to optimize the performance of our algorithm on GPU including group size regulation, thread assignment methodology, and thread number adjustment.

A. Group size regulation

There are two main principles to improve throughput on GPU. One principle is to employ as many threads as possible. The other principle is to utilize the shared memory. In our implementation, 512 threads, the maximum number of threads of a block are employed to process string matching at the same time. The 512 threads traverse the same Failureless-AC state machine. Because multiple threads traverse the same state machine, using shared memory to store the corresponding state transition tables is the most efficient method to improve the latency of accessing state transition tables. However, the size of shared memory is limited and cannot accommodate all virus patterns. In order to utilize the shared memory, we need to divide all virus patterns into several groups and compile these groups into small Failureless-AC state machines to fit into the shared memory.

There are two steps to divide all virus patterns into small groups. The first step is to group these virus patterns by their prefix similarity. By sharing prefixes, the corresponding state machine can be reduced. The second step is to determine the size of a Failureless-AC state machine to fit into the shared memory of a multiprocessor. Given the Snort rules and the size restriction of the state machine, our algorithm can determine the number of the state machine needed to

978-1-4244-5637-6/10/$26.00 ©2010 IEEE
implement all Snort rules. We parameterize the size of the state machine as 1, 2, 3, 4, 5, 6, and 7 KB and find the corresponding number of state machines which are 315, 136, 83, 63, 49, 41, and 35. We find the throughput goes up when the size of a state machine increases. We would like to mention that when the size of a state machine is over 8KB, the corresponding state table cannot fit into the shared memory of our GPU.

B. Thread assignment methodology
As described in section 3, all threads of PFAC traverse the same Failureless-AC state machine and terminate when no valid transition exists. Therefore, it is likely that many threads will terminate very early. To utilize the stream processors, we evaluate two different approaches of thread assignment.

The first approach of the thread assignment is called Average thread assignment which equally distributes the bytes of an input stream to each thread of the same block. Supposing we allocate 512 threads to process an input stream of 4,096 bytes, each thread is in responsible for processing eight locations. For example, the first thread processes the bytes starting at locations 0, 512, 1,024, 1,536, 2,048, 2,570, 3,072, and 3,584 of the input stream.

The other approach of the thread assignment is called Boss thread assignment. The Boss thread assignment declares a global variable called Boss to assign new jobs to the threads which have already finished their jobs. However, the Boss variable can be accessed by only one thread at a time, which is known as the critical section problem. Therefore, if many threads try to access the Boss variable at the same time, the throughput slows down significantly.

Finally, we choose the average thread assignment to implement our algorithm because the experimental results demonstrate the average thread assignment is much better than the Boss thread assignment.

C. Thread number adjustment
A GPU comprises multiple multiprocessors and an off-chip global memory. Each multiprocessor has 15 KB shared memory, eight stream processors with their register files, an instruction unit, constant cache and texture cache. Multiple blocks are allowed to execute on a multiprocessor which maps each thread to a stream processor. Because the registers and shared memory are split among the threads of blocks, the number of blocks which can be executed simultaneously depends on the number of registers per thread and the number of shared memory per block required for a given kernel function. In order to utilize the eight stream processors of a multiprocessor, we parameterize the thread number of a block as 64, 128, 192, 256, 320, 384, 448, and 512. The experimental results show that 512 threads can achieve the best throughput. Therefore, we choose the thread number of 512 to implement our algorithm.

V. EXPERIMENTAL RESULTS
We have implemented the proposed algorithm on a commodity GPU card and compare with the recent published GPU approaches. The experimental configurations are as follows:

- CPU: Intel® Core™2 Duo CPU E7300 2.66GHz
- System main memory: 4,096 DDR2 memory
- GPU card: NVIDIA GeForce GTX 295 576MHz
- 480 cores with 1,792 MB GDDR3 memory
- Patterns: string patterns of Snort V2.4

In order to evaluate the performance of our algorithm, we implement three approaches described in this paper for comparisons. As shown in Table 1, the CPU_AC denotes the method of implementing the AC algorithm on CPU, which is the most popular approach adopted by NIDS systems, such as Snort. The Direct_AC approach denotes the direct implementation of the AC algorithm on GPU. The PFAC denotes the Parallel Failureless-AC approach on GPU. Table 1 shows the results of these three approaches for processing two different input streams. Column one lists two different input streams, the normal case denotes a randomly generated sequence of 219Kbytes comprising 19,103 virus patterns, whereas the virus case denotes a sequence of 219 Kbytes comprising 61,414 virus patterns.

Column 2, 3, 4, and 5 list the throughput of the three approaches, CPU_AC, Direct_AC, and PFAC, respectively. For processing the normal case of input streams, the throughput of CPU_AC, Direct_AC, and PFAC are 997, 6,428, and 3,963,966 KBps (Kilo Bytes per second), respectively. The experimental results show that the PFAC performs up to 4,000 times faster than the CPU_AC approach while the Direct_AC can only perform 6.4 times as fast. In other words, the PFAC also achieves up to 600 times faster than Direct_AC approach on GPU.

Furthermore, because the new algorithm removes the failure transitions of the AC state machine, the memory requirement can also be reduced. Table 2 shows that the new algorithm can reduce the number of transitions by 50%, and therefore achieve a memory reduction of 21% for Snort patterns. Table 3 compares with several recent published GPU approaches [10][11][12][13]. In Table 3, columns 2, 3, 4, and 5 shows the character number, memory size, throughput, and the memory efficiency which is defined as the following equation.

\[
\text{Memory efficiency} = \frac{\text{Throughput}}{\text{Memory}} \quad (1)
\]

As shown in Table 3, our results are faster than all [10][11][12][13] with efficient memory usage. We would like to mention that the memory-based approach requires the design and fabrication of a dedicated hardware whereas the GPU approach is more general in the sense that both software and the virus patterns can be easily updated.
addition, with significant improvement on memory efficiency. In GPU approaches, the new algorithm achieves 3 times faster compared to the AC algorithm on CPU. Compared to other new algorithm on GPU can achieve a significant speedup matching by GPU. The experimental results show that the proposed a novel parallel algorithm to accelerate string massive data parallel computing. In this paper, we have attention due to their cost-effective and dramatic power of memory requirements.

VI. CONCLUSIONS

Graphics Processor Units (GPUs) have attracted a lot of attention due to their cost-effective and dramatic power of massive data parallel computing. In this paper, we have proposed a novel parallel algorithm to accelerate string matching by GPU. The experimental results show that the new algorithm on GPU can achieve a significant speedup compared to the AC algorithm on CPU. Compared to other GPU approaches, the new algorithm achieves 3 times faster with significant improvement on memory efficiency. In addition, because the new algorithm reduces the complexity of the AC algorithm, the new algorithm also improves on memory requirements.

REFERENCES


