ISSN:0975-9646

Review on Parallelized Multipattern Matching Using Optimized Parallel Aho-Corasick Algorithm on GPU

Prachi Oke, Prof. Mrs. A. S. Vaidya

Department of Computer Engineering, University of Pune
GES's RHSCOE, Nashik, INDIA

***Abstract*— Pattern Matching is a very computationally intensive operation in the Network Intrusion detection Systems where large amount of data has to be matched against the known patterns. With the advent in technology, storage capacity and link speed has increased, due to which there has been an increase in the amount of data that needs to be matched against the known patterns. But the traditional algorithms fail to handle this increased amount of data. So we need such a hardware and software solution that would help to handle this large amount of incoming data in the Network intrusion Detection Systems to match it with the known patterns or we can say virus signatures. So we will be using a parallel algorithm that matches an input string with the known patterns (virus signature) to check for the presence of any pattern in an input string and return the same if any. We will be running this algorithm on NVIDIA Geforce GTX 680 GPU with CUDA 6.5 programming model. And we will be introducing several optimization techniques for the Parallel AC algorithm that would eventually result in the reduction of time, cost, and memory usage required to execute Parallel AC algorithm on GPU.**

***Keywords*— Pattern Matching, Snort, KMP algorithm, AC algorithm, GPU.**

I. INTRODUCTION

A Network Intrusion Detection System like Snort [5], uses an Aho-Corasick algorithm to match the input data with the known attack patterns. This algorithm proves to be inadequate to meet the throughput requirements for high-speed networks and it tends to drop the packet data when it cannot handle the large amount of incoming data.

So, we will be using a Parallel Aho-Corasick algorithm which takes Snort virus signature as input pattern and constructs a DFA to find multiple occurrences of patterns in an incoming packet data. We will be running the algorithm on GPU because, GPUs have a highly parallel structure which makes them more effective than general purpose CPUs for algorithms where processing of large blocks of data is done in parallel [6]. The performance of Parallel Aho-Corasick algorithm run on GPU would result in higher throughput as compared to running a tradition Aho-Corasick algorithm on CPU.

Here, NVIDIA Geforce GTX 680 GPU with NVIDIA CUDA 6.5 programming model will be used. This GPU is considered to be a very fast and efficient GPU. TABLE I shows the specifications of NVIDIA Geforce GTX 680 GPU [7]. And NVIDIA CUDA 6.5 programming model is the latest version of most pervasive parallel computing platform and programming model [10]. It is available as a free download at www.nvidia.com/getcuda. It provides several new features. Reference [9] gives the features as:

1. CUDA 6.5 provides programmers with a robust, easy-to-use platform to develop advanced scientific, engineering, mobile and High Performance Computing applications
2. Support for Visual Studio 2013: CUDA 6.5 expands host compiler support to include Microsoft Visual Studio 2013 for Windows.
3. Double Precision Performance Improvements: The core math libraries in CUDA 6.5 introduce significant performance improvements for many double precision functions.
4. CUDA 6.5 significantly improves Multi-Process Service (MPS) performance: reducing launch latency from 7 to 5 microseconds, and reducing launch and synchronize latency from 35 to 15 microseconds.
5. CUDA programmers need to understand the constraints of the kernel and the GPU they will be working on and must decide about the block size to be used for the kernel launch, so that it must result in good performance. And a common way used to choose a good block size is to aim for high occupancy. Where high occupancy means the ratio of the number of active warps per multiprocessor to the maximum number of warps that can be active on the multiprocessor at once. With CUDA 6.5 calculating this occupancy is no more a tricky job as it was before, as it includes several new runtime features to aid in occupancy calculations.

TABLE II
NVIDIA GEFORCE GTX 680 GPU SPECIFICATION

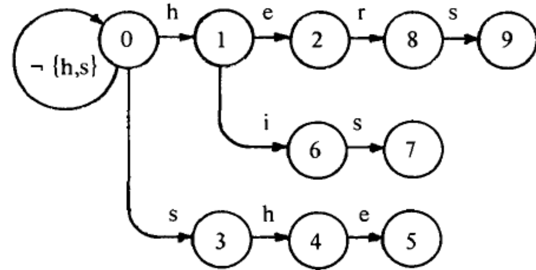
CUDA Cores	1536
Base Clock (MHz)	1006
Boost Clock (MHz)	1058
Texture Fill Rate (Billion/Sec)	128.8
Memory speed	6.0 Gbps
Standard Memory Config.	2048MB
Memory Interface Width	256-bit GDDR5
Memory Bandwidth (GB/sec)	192.2
Bus Support	PCI Express 3.0
Certified for Windows 7	Yes

II. LITERATURE SURVEY

String and Pattern matching is required in numerous applications. Many String and Pattern matching algorithms have been developed and many optimizations for those algorithms have also been done till date. The problem of string and pattern matching is, given a string 'S' and a pattern 'p', it deals with finding a pattern 'p' in string 'S' and if 'p' does occur in 'S', then returning the position in 'S' where 'p' occurs. And the most usual approach to solve this problem is to start with the first element in pattern 'p' and compare it with the first element in String 'S', if the elements match then proceed with comparing second element of 'p' with second element of 'S' and so forth. Now, if the mismatch occurs at any position, then it starts again by comparing first element of pattern 'p' with the second element of String 'S' and so forth. Thus, unnecessary shifts of pattern 'p' are done or in other words, backtracking on 'S' is done. These repetitive comparisons lead to the runtime of O(mn).

To overcome the drawback of this usual approach, an algorithm that matches a single pattern in the given string at a time, like the KMP algorithm [2], [8] was developed. KMP algorithm needs to use two functions, first to calculate the prefix function 'π' and the other is the matcher function. The prefix function 'π' encapsulates the information needed to avoid the unnecessary shifts of pattern 'p' or we can say backtracking on 'S' never occurs. The matcher function takes as an input the string 'S', pattern 'p' and the prefix function 'π' and finds the occurrence of 'p' in 'S' and returns the number of shifts of 'p' after which occurrence is found. The prefix function requires O(m) running time while its matcher function requires O(n) running time.

Then there are algorithms that match multiple patterns in the String at a time, like Aho-Corasick (AC) algorithm [1]. The AC algorithm uses a state machine to recognize the patterns in the input stream. Also it introduces a failure transition, to backtrack a state machine to recognize the patterns starting at any location of an input stream. The Aho-Corasick algorithm requires three functions, Goto function, Failure function, and Output function. Fig. 1 shows these three functions. Where a Goto function tells us that, in the state machine constructed from the known patterns, if at a state 's' an input 'a' is given, whether or not it leads to a valid transition to the next state. That is, whether g(s, a) = *validnextstate* or g(s, a) = *fail*. When the goto function reports fail, that is, when there is no valid transition to the next state, the Failure function tells us about the state at which the transition must be made in case of no valid next state transition. The failure function for all the states is calculated depth wise from depth 1 then depth 2 and so on. First the failure function f(s) of all the states of depth 1 are made 0. And for all the remaining states the failure function is calculated as, if f(s) = s', the machine repeats the cycle with s' as the current state and a as the current input symbol. The Output function is updated while computing the failure function. That is, when we are computing f(s) = s', outputs of 's' are merged with the output of s'. if 'n' is the input stream length then the best-case and worst complexity of AC algorithm would be O(n).



(a) Goto function.

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

(b) Failure function.

i	*output* (i)
2	{he}
5	{she, he}
7	{his}
9	{hers}

(c) Output function.

Fig. 1. Goto, Failure and Output function. [1].

In traditional Aho-Corasick algorithm, there are no threads running in parallel to find multiple patterns in the text or in the given string. There, a single thread is responsible for finding all the patterns in the string by traversing through the DFA. When for an input character there is no valid next state transition we say a failure has occurred. And as there is a single thread, it needs to jump to some other state from where it can continue matching the patterns again. So there was a need for failure transitions, which makes a thread to jump at some other state to continue finding the patterns, when a failure at a particular state occurs. For example for the patterns AB," "ABG," "BEDE," and "ED", an AC state machine would look like this as shown in Fig. 2. If at a state 2, an input character other than 'G' is taken, it would lead to a failure transition and would make a thread to jump at state 4.

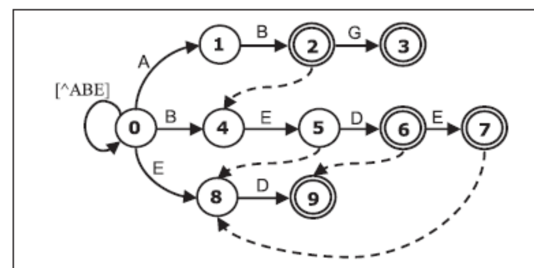


Fig. 2. Aho-Corasick state machine of the patterns "AB," "ABG," "BEDE," and "ED." [4]

Improvements to the traditional Aho-Corasick algorithm (AC) were done like, Data Parallel Approach to Aho-Corasick algorithm [3], [4], which tries to parallelize the AC algorithm. Data Parallel Approach to AC algorithm divides the input stream into multiple chunks and each of the chunks is allocated a thread. These individual threads perform an AC algorithm over those chunks.

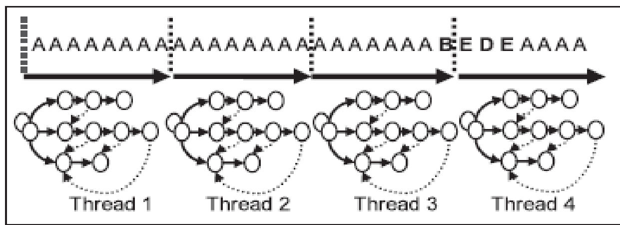


Fig. 3. Data Parallel Approach with boundary detection problem. [4]

But the problem with the Data Parallel Approach to AC algorithm is detecting the boundary. That is, problem arises when pattern occurs at the boundary of the two chunks. In this case the pattern cannot be identified by either of the two threads allocated to those two chunks. So to resolve this problem each thread must scan an addition length across the boundary which is equal to longest pattern length - 1. The best-case and the worst-case complexity of Data Parallel Approach to AC algorithm is, O(n/s + m), where n is the input stream length, s is the number of chunks and m is the longest pattern length.

III. PROPOSED SYSTEM

In Parallel Aho-Corasick algorithm, the failure transitions of the AC state machine are all removed also, the self-loop transition of the initial state has been removed. Because, in Parallel Aho-Corasick algorithm a thread is allocated to every byte of an input stream which is responsible for finding the pattern beginning at its starting position. Which means, as shown in the Fig. 4, a thread located at ‘A’ is responsible for finding a pattern that begins with ‘A’ by traversing an AC state machine without failure transitions, similarly a thread located at ‘B’ is responsible for finding a pattern that begins with ‘B’ and so on.

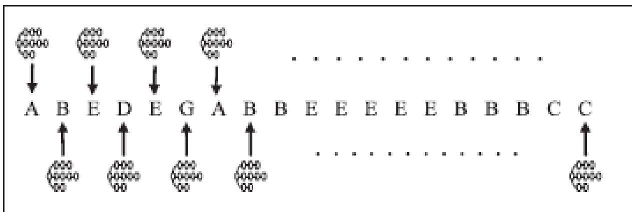


Fig. 4. Parallel Aho-Corasick Algorithm [4].

So if the patterns that begin at the position located by their respective threads do not exist, then such threads terminate immediately. The Fig. 5 shows an AC state machine without failure transition for the patterns “AB,” “ABG,” “BEDE,” and “ED.”

We can see from the Fig. 5, that the patterns either begin with ‘A’, ‘B’ or ‘E’. Thus any such threads that are allocated

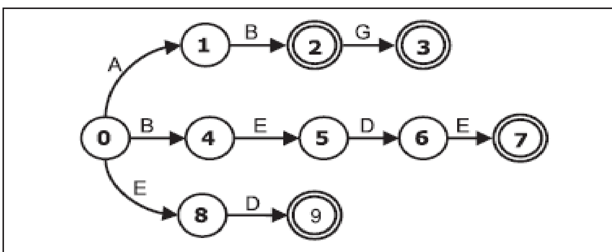


Fig. 5. Parallel AC state machine without failure transitions for the patterns “AB,” “ABG,” “BEDE,” and “ED” [4]

to the characters like, ‘G’, ‘D’, ‘C’ (form Fig. 4) or basically other than the characters ‘A’, ‘B’ or ‘E’ would terminate immediately. So, all the three threads that are allocated to characters ‘A’, ‘B’ and ‘E’ would all run in parallel. Let us look at an example below.

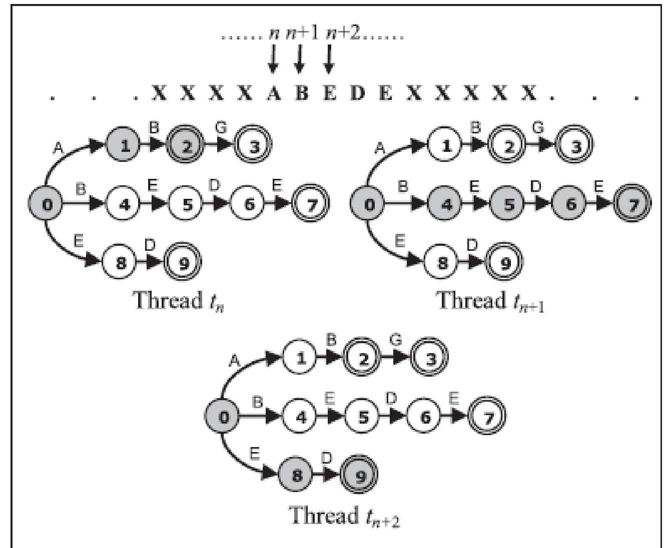


Fig. 6 Example of Parallel Aho-Corasick Algorithm without failure transition [4].

In the Fig. 6 given above the threads *tn, tn+1, tn+2* are allocated to characters ‘A’, ‘B’ and ‘E’ and will traverse AC state machine without failure transitions, parallely. Thread *tn* will traverse an AC state machine and will find a pattern “AB” and the moment it takes an input “E”, it would terminate as there is no valid next state transition for input “E” at state 2. Similarly, threads *tn+1* and *tn+2* would find and match the patterns “BEDE” and “ED” and would terminate. The other threads that are allocated to characters “X” would terminate immediately as there is no valid next state transition for this character at state 0. Thus, even though this algorithm allocates a very large amount of threads, many of them can terminate at a very early stage. Thus, instead of using one thread to find all the patterns in the string, the threads can run in parallel and thus reduce the time required in finding and matching the patterns in the string. So, we can say, each thread of Parallel AC algorithm without any failure transitions would run in the best time of O(1) and the worst time of O(m) where m is the longest pattern length.

A. *Contribution*

In this research work, we will be introducing several optimization techniques for Parallel Aho-Corasick algorithm on GPU, including, reducing the memory requirement needed to store the Parallel AC algorithm table which is a state transition table. The Parallel AC algorithm table gives the next state information where each row contains 256 columns storing the next state information corresponding to each ASCII alphabet. As the Parallel AC algorithm removes most transitions, the corresponding state transition table is very sparse. Thus the state transition table, which is sparse, can be compressed and can be stored as

one dimensional array. In this way memory usage will be reduced. Then, we can reduce the latency of global memory access as Global memory of GPU is the slowest memory. While storing the data to GPU's memory, the host transfers the data from host's memory to GPU's memory, which is usually the Global memory. Thus storing the data in the texture memory instead, which is considered to be the fastest memory, would reduce the latency of data access. Then, we can reduce the time and cost incurred in transferring the data from host's memory to GPUs memory by directly storing the data in the GPUs memory without first requiring it to store it in the Host's memory and then transferring it to the device's memory. Also, we can achieve a data transfer overlap where CPU and GPU can work as a single entity. Which means all the computationally intensive operation will be done on GPU and at the same time less computationally intensive operation can be made to run on CPU. This again would lead to reduction in time and more throughputs.

IV. CONCLUSIONS

We have seen single pattern and multiple pattern matching algorithms and comparisons between the same. Also we saw how several optimizations for the Parallel Aho-Corasick algorithm can be done. The Parallel Aho-Corasick algorithm will run on NVIDIA Geforce GTX 680 GPU with CUDA 6.5 programming model. By parallelizing the traditional Aho-Corasick algorithm we can reduce the time required for its execution. Further, Running the Parallel Aho-Corasick algorithm on GPU would accelerate its speed as compared to running the same algorithm on CPU. The proposed optimizations for the Parallel Aho-Corasick algorithm would further reduce the time, cost, and memory usage required to run the Parallel Aho-Corasick algorithm on GPU.

REFERENCES

[1] A.V. Aho and M.J. Corasick, Efficient String Matching: An Aid to Bibliographic Search, Comm. ACM, vol. 18, no. 6, pp. 333-340, 1975.

[2] DONALD E. KNUTH, JAMES H. MORRIS, AND VAUGHAN R. PRATT, Fast Pattern Matching in Strings, SIAM J. COMPUT. Vol. 6, No. 2, June 1977.

[3] A. Tumeo, O. Villa, and D. Sciuto, E_cient Pattern Matching on GPUs for Intrusion Detection Systems, Proc. Seventh ACM Intl Conf. Computing Frontiers, 2010.

[4] Cheng-Hung Lin, Lung-Sheng Chien, and Shih-Chieh Chang, Accelerating Pattern Matching Using a Novel Parallel Algorithm on GPUs, IEEE Transactions On Computers, Vol. 62, No. 10, October 2013.

[5] Snort, https://www.snort.org/.

[6] GPU, http://en.wikipedia.org/wiki/ Graphics_processing_unit

[7] Geforce GTX680 Specification, http://www. geforce.Com / hardware/ /desktop-gpus/geforce-gtx-680/specifications.

[8] KMP algorithm by example, http://www.cs.utexas.edu/users/moore/ best-ideas/string-searching/kpm example.htmlstep01.

[9] CUDA 6.5 features, http://devblogs.nvidia.com/parallelforall/10-ways-cuda-6-5-improves-performance-productivity/

[10] About CUDA 6.5, http://www.scientific-computing.com/press-releases/product_details.php?product_id=1935/