International Journal of Modern Trends in Engineering and Research
www.ijmter.com
e-ISSN No.:2349-9745, Date: 2-4 July, 2015

Optimized Parallel Aho-Corasick Algorithm on GPU

Prachi S. Oke[1], Mrs. Archana S. Vaidya[2]
[1]*Department of Computer Engineering, G.E.S's R.H Sapat College Of Engineering, Nashik.*
[2]*Department of Computer Engineering, G.E.S's R.H Sapat College Of Engineering, Nasik.*

Abstract - Network Intrusion Detection Systems (NIDS) need to handle very computationally intensive operations like pattern matching, where huge amount of data needs to be matched against the known patterns. Storage capacity and link speed has increased with the advent in technology. Therefore there has been an increase in the amount of data that needs to be matched against the known patterns. The traditional algorithms cannot handle this increased amount of data. Therefore, we need such a hardware and software solution that would help to handle this increased amount of incoming data in NIDS to match it with the known patterns. We are using parallel multipattern matching algorithm that matches an input string with the known patterns (attack patterns or virus signatures) to check for the presence of any pattern in an input string and it would return the same if any. We are running this algorithm on NVIDIA Geforce GTX 680 GPU with CUDA 6.5 programming model. We have also introduced several optimization techniques for the Parallel AC algorithm that would result in the reduction of memory usage, time and cost required to execute Parallel AC algorithm on GPU.

Keywords- Pattern matching; KMP algorithm; Snort; AC algorithm; GPU.

I. INTRODUCTION

Network Intrusion Detection Systems (NIDS) have been widely adopted to protect computer systems from several network attacks such as denial-of-service attacks, port scans, or malwares. The most critical operation affecting the performance of NIDS is to inspect packet content against thousands of attack patterns. Due to the ever increasing number of attacks, traditional sequential pattern matching algorithms are inadequate to meet the throughput requirements for high-speed networks. So we need a multi-pattern matching algorithm to meet the throughput requirements for high-speed networks. Snort [2] a Network Intrusion Detection System, uses an Aho-Corasick (AC) algorithm to match the input data with the known attack patterns. The large amount of incoming data cannot be handled by this traditional algorithm and it drops the packet data. To meet the throughput requirements for high-speed networks this algorithm proves to be inadequate.
Here, we are using a Parallel Aho-Corasick algorithm which takes Snort virus signatures as input patterns and constructs a DFA to find multiple occurrences of patterns in an incoming packet data. We are 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 [3]. The Parallel Aho-Corasick algorithm run on GPU results in better throughput than running the traditional Aho-Corasick algorithm on CPU.

II.LITERATURE SURVEY

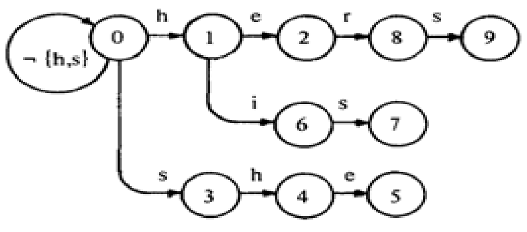
One of the most common approaches for solving the string and pattern matching problem is to compare the first element of the pattern 'P' to be searched, with the first element of the string 'S'. If the first element of 'P' matches with the first element of 'S', compare the second element of `P' with the second element of `S'. If match found continue the same process until entire 'P' is found. If a mismatch occurs at any position, shift ‘P’ one position to the right and beginning from first element of ‘P’, repeat the comparison. Elements of ‘S’ which were involved in the earlier comparisons are repeatedly involved in some future iterations. These unnecessary comparisons lead to the runtime of O(mn).

The Knuth-Morris-Pratt (KMP) Algorithm [4], [5] overcomes the drawback of this usual approach by avoiding comparisons with elements of ‘S’ that were previously involved in comparison with some elements of the pattern ‘P’ to be matched. i.e., backtracking on the string ‘S’ does not occur. The algorithm uses two functions, the prefix function π and the matcher function. The prefix function, π for a pattern encapsulates knowledge about how the pattern matches against shifts of itself. This information can be used to avoid unnecessary shifts of the pattern ‘P’. The matcher function takes string `S', pattern `P' and the prefix function π as an input and finds the occurrence of ‘P’ in ‘S’ and returns the number of shifts of ‘P’ after which occurrence is found. The prefix function takes O(m) running time while its matcher function takes O(n) running time.

Aho-Corasick [6] algorithm can match multiple patterns in a string at a time. It constructs a state machine to recognize the patterns in the input string. It introduces a failure transition, to map a state into another state. The Aho-Corasick algorithm uses three functions, Goto function g, Failure function f, and Output function output. Fig. 1 shows these three functions. Where, a Goto function g leads to a valid next state transition or reports fail. That is, if `s' is the current state of a machine and `a' is an input character from the input string, it decides whether g(s, a)= s' or g(s, a) = fail. Where, s' is the next state that machine enters when it makes a valid transition. When a Goto function g, reports fail, it indicates absence of valid next state transition and a Failure function f is consulted. If f(s) = s', the machine repeats the cycle with s' as the current state and `a' as the current input symbol. During the computation of a Failure function, an output function is updated. When we determine f(s) = s', output of state s is merged with the output of state s'. The best-case and the worst complexity of AC algorithm would be O(n). where, `n' is an input stream length. In AC algorithm a single thread is responsible for finding all the patterns in the string by traversing through the DFA. For example, Fig. 2 shows an AC state machine for the patterns ``AB,'' ``ABG,'' ``BEDE,'' and ``ED''. If at state 2, an input character other than `G' is taken, it makes a failure transition to state 4.

A hardware based string search engine for antivirus applications is presented in [7]. They have used ClamAV [8] virus database for their study. The method presented in this paper is for handling static strings, as majority of patterns in ClamAV pattern set are static strings.

Data Parallel Approach to AC algorithm [9], [1] parallelizes and improves the traditional Aho-Corasick algorithm. This approach divides an input stream into multiple chunks and a thread is allocated to each of these chunks. These threads individually traverse an AC state machine over those chunks to find and match the patterns as shown in Fig. 3. Data Parallel Approach to AC algorithm faces the problem of detection at the boundary. When a pattern occurs at the boundary of the two chunks, it cannot be identified by either of the two threads allocated to those two chunks. To resolve this problem each thread must scan an addition length across the boundary which is equal to longest pattern length – 1. If `n' is an input stream length, `s' is the number of chunks and `m' is the longest pattern length, the best-case and the worst-case complexity of Data Parallel Approach to AC algorithm is, O (n/s + m).



(a) Goto function.

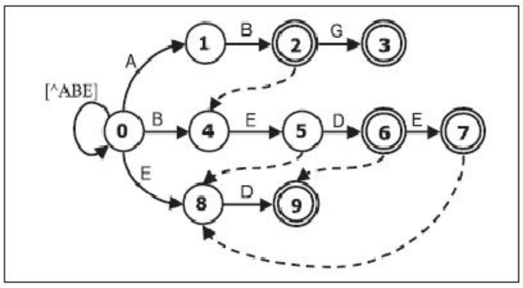
i	1	2	3	4	5	6	7	8	9
$f(i)$	0	0	0	1	2	0	3	0	3

(b) Failure function.

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

(c) Output function.

Fig. 1 Aho-Corasick Goto, Failure and Output function.



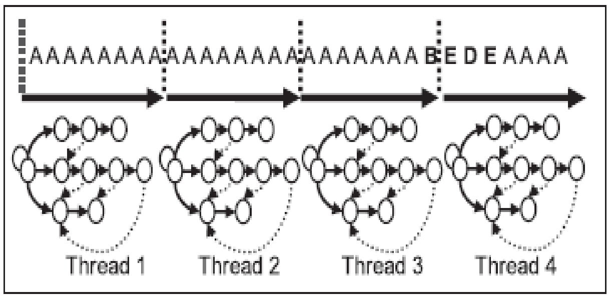


Fig. 2 Aho-Corasick state machine of the patterns "AB "ABG," "BEDE," and "ED".

Fig. 3 Data Parallel Approach with boundary detection problem

III. PROPOSED WORK

The system architecture is shown in the Fig.4. We can see from the figure, the Host and the device performing their tasks in order to make the proposed system work. It is always the CPU that starts executing the code until it detects an instruction to launch a kernel on GPU. When a kernel is launched the GPU comes into picture. And now it is the responsibility of a GPU to perform all the computationally intensive tasks for what it is called.

Initially, the task of CPU starts with accepting the two inputs for the system that are, the stream data or the packet payload data that has to be checked for the presence of attack patterns if any. For this, it needs another input which is the virus signatures or attack patterns with which the packet data will be matched. Once it has read all the patterns (virus signatures), it constructs a DFA from these patterns. This DFA is a parallel Aho-Corasick state machine that is traversed by the threads launched by the GPU to find and match the patterns in the input stream. Once this DFA is constructed, a CUDA kernel is launched. As said earlier a kernel is the part of the code that runs on GPU, which usually has computationally intensive operations.

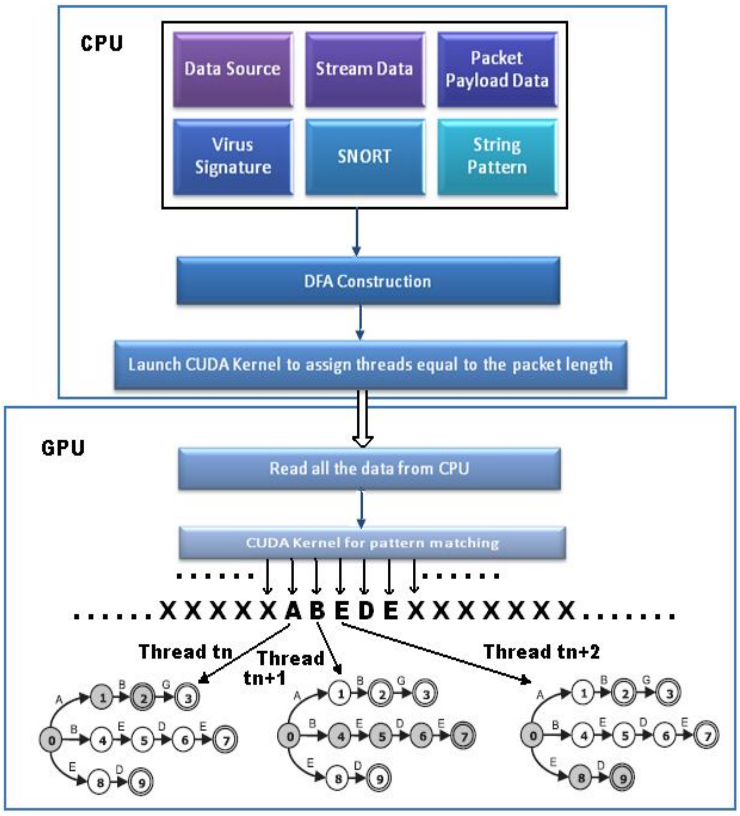

CPU
Data Source
Stream Data
Packet Payload Data
Virus Signature
SNORT
String Pattern
DFA Construction
Launch CUDA Kernel to assign threads equal to the packet length
GPU
Read all the data from CPU
CUDA Kernel for pattern matching
......XXXXXABEDEXXXXXXX.......
Thread tn
Thread tn+1
Thread tn+2

Fig. 4 System Architecture

Now the GPU starts its computation by first copying the data from the Host's memory to its own memory. Then it reads the number of packets and calculates the length of each packet so that it can assign the number of threads equal to the length of the data packet. That is, assigning threads to each and every character of the data packet. Once this is done a second kernel is launched which is responsible for pattern matching process. We can see from the figure that the threads that are assigned to each and every character of the data packet are responsible for finding the pattern that begins with the character they are pointing to. The threads t_n, t_{n+1}, t_{n+2} are allocated to characters `A', `B' and `E' and will traverse AC state machine without failure transitions, parallely [1]. Thread t_n 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 $t_{n+1}$ and $t_{n+2}$ 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.

3.1 Optimization Techniques for Parallel Aho-Corasick Algorithm on GPU

3.1.1 The Storage Model

The Parallel AC Algorithm needs to use a state transition table that gives the next state information. The number of rows corresponds to number of states in the state machine and there are 256 columns corresponding to 256 ASCII characters. Most of this table is sparse; therefore we are modifying it to banded row format. For example, Fig. 5 shows DFA implementation of existing AC algorithm. There are ten pointers in the state index table to the ten state nodes. At the state 0 and an input character as h, a 1 indicates a valid next state transition while the zeros indicate absence of valid next state transition. This approach of storage requires large amount of memory. Therefore we are using banded-row format to store data efficiently in parallel Aho-Corasick algorithm.

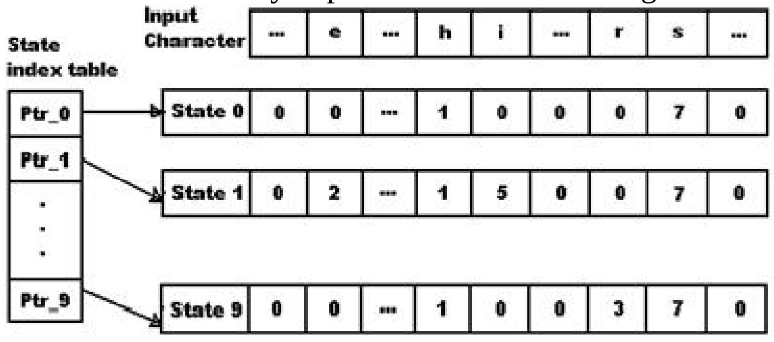

Input Character
... e ... h i ... r s ...
State index table
Ptr_0
Ptr_1
Ptr_9
State 0 0 0 ... 1 0 0 0 7 0
State 1 0 2 ... 1 5 0 0 7 0
State 9 0 0 ... 1 0 0 3 7 0

Fig. 5 DFA Implementation of Existing Aho-Corasick algorithm

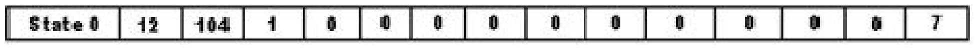


Fig. 6 Banded-row format

Fig. 6 shows a banded-row format. It stores the elements form the first non-zero value which in this example is 1 for state 0, to the last non-zero value which is 7. The first entry 12, in the banded-row format indicates the number of next-state values stored. And 104, the second entry which the ASCII value of character h, indicates the position of the first non-zero next-state in the original standard DFA node.

3.1.2 To reduce the latency of global memory access

When the data is stored to GPU's memory, the host transfers the data from host's memory to GPU's memory, which is usually the global memory. The global memory of GPU is the slowest memory. Therefore we are reducing the latency of global memory access. Storing the data in the texture memory instead, which is considered to be the fastest memory, would reduce the latency of data access.

3.1.3 Minimizing data transfer

Host (CPU) data allocations are in pageable memory by default. The GPU cannot access data directly from pageable memory of CPU, so when the data transfer starts from pageable host memory to device memory, the CUDA must first allocate a temporary space for page-locked memory, copy the host data to the pinned memory array, and then transfer the data from the pinned array to device memory. To avoid the cost of the transfer between pageable and pinned host arrays we are directly allocating our host arrays in pinned memory.

3.1.4 Overlap data transfer

We are also trying to achieve data transfer overlap where CPU and GPU work as a single entity. Kernel asynchronous behavior makes overlapping device and host computation possible. We can modify the code to add some independent CPU computation. The GPU kernel function say Aho-Corasick() will be launched on the device and CPU thread executes say ACMatch(), overlapping its execution on the CPU with the kernel execution on the GPU. This would lead to reduction in time and better throughput.

VI. EXPERIMENTAL SETUP

Here, Intel Ci3 CPU, NVIDIA Geforce GTX 680 GPU with NVIDIA CUDA 6.5 programming model are used. GTX 680 GPU has Kepler architecture [10] and is considered to be a very fast and efficient GPU. TABLE I shows the specifications of NVIDIA Geforce GTX 680 GPU [11]. And NVIDIA CUDA 6.5 programming model is the latest version of most pervasive parallel computing platform and programming model [12]. It is available as a free download at www.nvidia.com/getcuda. It provides several new features and are explained in detail in [13]. And the tools used are NVIDIA Nsight Eclipse and NVIDIA visual profiler [14] [15].

V. DATA SETS AND RESULTS

SNORT [2] is an open-source initiative with a large database of malware samples. Security researchers throughout the world are using many of these samples as a testbed when testing experimental algorithms or when proposing new solutions for malicious code detection. We are also using SNORT virus signature as a pattern dataset which is one of the two inputs and second input are the data packets that are to be checked against the attack patterns. These are manually generated text files of 32 bit string.
Table No. 1 shows the pattern matching time comparison of Serial and Parallel AC Algorithm on Intel Ci3 CPU done in Open MP [16] run using GCC compiler [17] and Unoptimized Parallel AC Algorithm on GPU for pattern lengths 2 and 32 of 500 words, 1000 words, 5000 words and 50000 words and input text size of 5000 words, 50000 words. We will be comparing the time taken by the Serial and Parallel AC algorithm to run CPU with the Unoptimized and Optimized Parallel AC Algorithm on GPU. Implementation of an Optimized Parallel AC Algorithm on GPU, results in reduced time, cost and memory usage and higher throughput.

Table No. 1. Pattern Matching Time Comparisons of Serial and Parallel AC Algorithms

Serial and Parallel Algorithms Performance					
Pattern Size	Text Packet Size	Serial AC Algorithm on CPU	Parallel AC Algorithm on CPU	Parallel AC Algorithm on GPU (UnOptimized)	Parallel AC Algorithm GPU (Optimized)
Patterns of 100 words of length 2	5000 words	41.86ms	5.35ms	0.255ms	0.2ms
Patterns of 1000 words of length 2	5000 words	80.39ms	0.68ms	0.269ms	0.242ms
Patterns of 5000 words of length 2	5000 words	94.18ms	0.58ms	0.2585ms	0.157ms
Patterns of 50000 words of length 32	5000 words	77.68ms	1.2685ms	0.689915ms	0.435ms
Patterns of 100 words of length 2	50000 words	418ms	4.07882ms	1.944ms	1.2ms
Patterns of 1000 words of length 2	50000 words	799ms	2.30ms	0.988ms	0.648ms
Patterns of 5000 words of length 2	50000 words	961ms	2.1863249999ms	0.773ms	0.428ms
Patterns of 50000 words of length 32	50000 words	740ms	8.273ms	5.81ms	4.2ms

CONCLUSION

GPUs have a massive parallel computation power than CPUs. As the traditional algorithms are inadequate to meet the throughput requirements for high-speed networks, with the Parallel implementation of the traditional Aho-Corasick algorithm, we reduce the time required for its execution. The Parallel Aho-Corasick algorithm run on GPU results in better throughput than running the traditional Aho-Corasick algorithm on CPU. We have introduced several optimization techniques for the Parallel Aho-Corasick algorithm. The algorithm runs on NVIDIA Geforce GTX 680 GPU with CUDA 6.5 programming model. Proposed optimizations for the Parallel Aho-Corasick algorithm further reduces the time, cost, and memory usage required to run the algorithm on GPU. Serial and parallel implementation results of AC algorithm on CPU and parallel implementation of AC algorithm without optimizations on GPU are provided.

REFERENCES

[1] 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.
[2] Snort, https://www.snort.org/.
[3] Graphics Processing Unit Definition, http://en.wikipedia.org/wiki/Graphicsprocessingunit
[4] DONALD E. KNUTH, JAMES H. MORRIS, AND VAUGHAN R. PRATT, "Fast Pattern Matching in Strings", SIAM J. COMPUT. Vol. 6, No. 2, June 1977.
[5] KMP algorithm by example, http://www.cs.utexas.edu/users/moore/bestideas/string-searching/kpmexample. htmlstep01
[6] 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.
[7] Derek Pao, Xing Wang, Xiaoran Wang, Cong Cao, and Yuesheng Zhu, "String Searching Engine for Virus Scanning", IEEE Transactions on Computers, Vol. 60, No.11, November 2011.
[8] ClamAV antivirus software, http://www.clamav.net
[9] A. Tumeo, O. Villa, and D. Sciuto, "Efficient Pattern Matching on GPUs for Intrusion Detection Systems", Proc. Seventh ACM Intl Conf. Computing Frontiers, 2010.
[10] NVIDIA Kepler Architecture, http://www.nvidia.com/object/nvidiakepler.html
[11] Geforce GTX680 Specification, http://www.geforce.Com/hardware/desktop-gpus/geforce-gtx680/ specifications.
[12] About CUDA 6.5, http://www.scientific-computing.com/pressreleases/productdetails.php? productid=1935/
[13] CUDA 6.5 features, http://devblogs.nvidia.com/parallelforall/10-wayscuda-6-5-improves-performance-productivity/
[14] Performance Analysis Tools, https://developer.nvidia.com/performanceanalysis-tools
[15] CUDA Toolkit Documentation, http://docs.nvidia.com/cuda/index.htmlcudacbestpractices
[16] OpenMP, http://openmp.org/wp/
[17] GCC, http://gcc.gnu.org/