ISSN(Online): 2320-9801
ISSN (Print): 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering

(An ISO 3297: 2007 Certified Organization)

Vol. 3, Issue 6, June 2015

Optimization of Parallel Aho-Corasick Multipattern Matching Algorithm on GPU

Prachi S. Oke[1], Archana S. Vaidya[2]

Student, Dept. of Computer Engineering, G.E.S's R. H. Sapat College of Engineering, Nashik, India[1]

Assistant Professor, Dept. of Computer Engineering, G.E.S's R. H. Sapat College of Engineering, Nashik, India[2]

ABSTRACT: Network Intrusion Detection Systems (NIDS) need to handle 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 have used a 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 would return the state in the DFA and position in an input string where the pattern was found. We have run this algorithm on NVIDIA Geforce GTX 680 GPU with CUDA 6.5 programming model. We have also introduced several optimization techniques for the Parallel Aho-Corasick algorithm that would result in the reduction of memory usage, time and cost required to execute Parallel AC algorithm on GPU. Our results show that Optimized Parallel Aho-Corasick algorithm on GPU takes very less time for execution as compared to running the Serial Aho-Corasick algorithm on CPU, Parallel Aho-Corasick algorithm on CPU and Unoptimized Parallel Aho-Corasick algorithm on GPU.

KEYWORDS: Pattern Matching; KMP algorithm; Snort; AC algorithm; GPU; CUDA

I. Introduction

String and pattern matching problems are fundamental to any computer application involving text processing. Pattern matching can be used in finding similar DNA or protein sequences, in Intrusion Detection Systems to find the presence of virus signatures in the incoming packets. 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 [3] 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. Pattern matching routines in Snort account for up to 70% of total execution time [4]. To meet the throughput requirement for high-speed networks this algorithm proves to be inadequate.

Here, we have used a Parallel Aho-Corasick algorithm that takes Snort virus signatures as input patterns and constructs a DFA to find multiple occurrences of patterns in an incoming packet data. We have run this algorithm on GPU because, work by G. Vasiliadis et al. [5] shows that traditional CPU based techniques cannot match the speed at which GPUs perform heavy-duty anti-malware operations effectively 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 Parallel Aho-Corasick algorithm run on GPU results in better throughput than running the traditional Aho-Corasick algorithm on CPU. In the next section II we are presenting the literature survey over the related work. In the section III proposed work and its system architecture it depicted. Section IV shows the

ISSN(Online): 2320-9801
ISSN (Print): 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering

(An ISO 3297: 2007 Certified Organization)

Vol. 3, Issue 6, June 2015

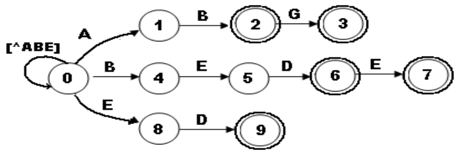
experimental setup. In the section V and VI the dataset used and the results are presented and section VII covers conclusion.

II. RELATED WORK

String and Pattern matching is used in numerous applications. Many String and Pattern matching algorithms have been developed and many optimizations for those algorithms have also been done till date. 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 [7], [8] over-comes the drawback of this usual approach by avoiding com-parisons 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 and the matcher function. The prefix function, 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 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 [9] 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) AC Goto Function

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

b) AC Failure Function

i	*output (i)*
2	{AB}
3	{ABG}
6	{BED, ED}
7	{BEDE}
9	{ED}

c) AC Output Function

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

ISSN(Online): 2320-9801
ISSN (Print): 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering

(An ISO 3297: 2007 Certified Organization)

Vol. 3, Issue 6, June 2015

A hardware based string search engine for antivirus applications is presented in [10]. They have used ClamAV [11] 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.

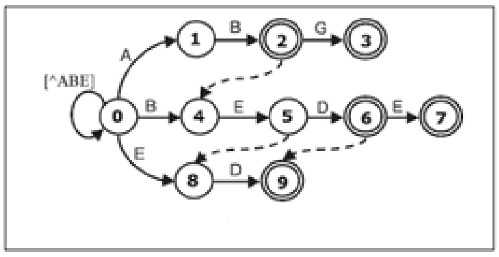


Fig. 2 Aho-Corasick state machine of the patterns “AB,” “ABG,” “BEDE,” and “ED.”

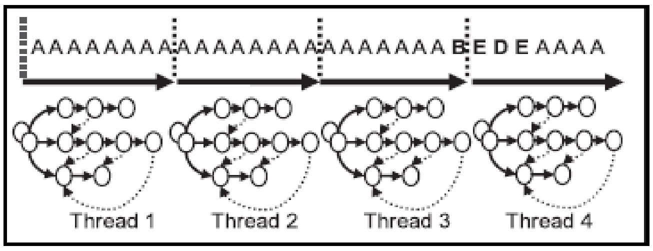


Fig. 3 Data Parallel Approach with boundary detection problem

Data Parallel Approach to AC algorithm [12], [1] [2] 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).

III. Proposed System

We can see the system architecture from the Fig. 4 where, the Host and the device perform their tasks in order to make the proposed system work. It is always the CPU that starts execution of 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.

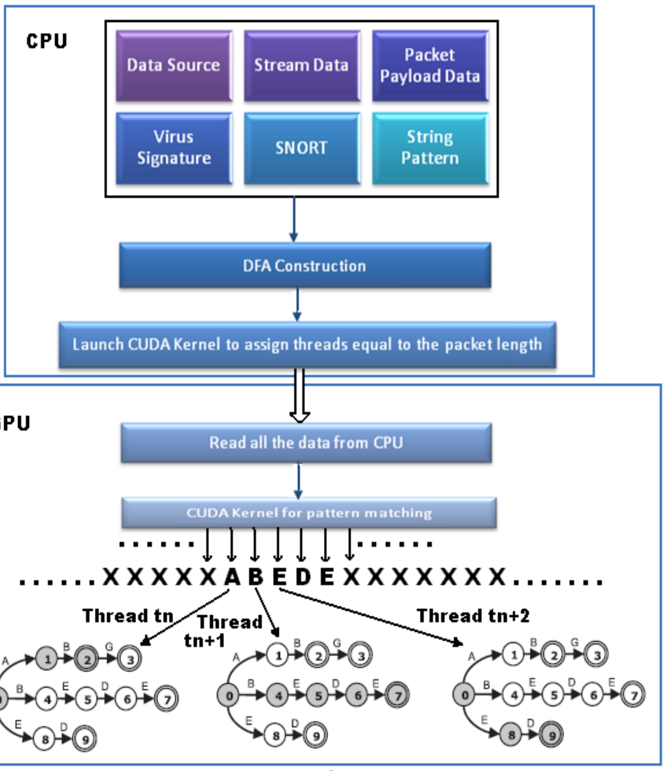
Now the GPU starts its computation by first copying the data from the Host’s memory to its own memory. Then it reads the number of packets and calculates the length of each packet so that it can assign the number of threads equal to the length of the data packet. That is, assigning threads to each and 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 tn, tn+1, tn+2 are allocated to characters ‘A’, ‘B’ and ‘E’ and will traverse AC state machine without failure transitions, in parallel [1][2]. 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.

ISSN(Online): 2320-9801
ISSN (Print): 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering

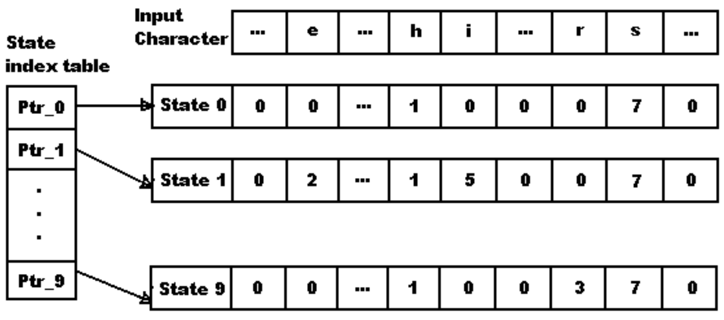
(An ISO 3297: 2007 Certified Organization)

Vol. 3, Issue 6, June 2015


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

A. Optimization Techniques for Parallel Aho-Corasick Algorithm on GPU

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 have used a banded-row format to store data efficiently in parallel Aho-Corasick algorithm. 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.


State index table
Input Character
... e ... h i ... r s ...
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

ISSN(Online): 2320-9801
ISSN (Print): 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering

(An ISO 3297: 2007 Certified Organization)

Vol. 3, Issue 6, June 2015

State 0	12	104	1	0	0	0	0	0	0	0	0	0	0	7

Fig. 6 Banded-row format

2) To reduce the latency of global memory access: When the data is stored to GPUs memory, the host transfers the data from host's memory to GPU's memory, which is usually the global memory and then the accesses are performed on the global memory that is considered to be as is the slowest memory of GPU. Therefore we are reducing the latency of global memory access by storing the data in the shared memory instead, which is much faster than global memory, and this would reduce the latency of data access.

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 as shown in the Fig. 7.

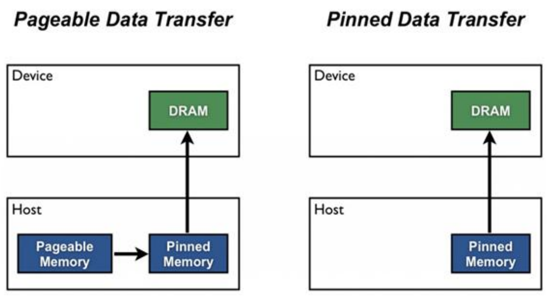


Fig. 7 Data transfer using pinned memory

4) Overlap data transfer: We have also achieved data transfer overlap where CPU and GPU work as a single entity. Kernel asynchronous behaviour makes overlapping device and host computation possible. We have modified the code further 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 better throughput.

B. Algorithm

Algorithm: Optimized Parallel AC Algorithm

Input:

- Input string S = [0, 1, ..., N-1] of length N, and
- Patterns P= [0, 1, ..., M-1] of length M.

Output:

- matchState i.e. the state at which a pattern is found in the state machine.
- numPatterns i.e. the total number of matched patterns.
- position i.e. an index position in the input string at which the pattern was matched.

Data Structures: Here, we will call state transition table as PAC_Table which means Parallel AC Table and an Output_Table which is a one dimensional array, is required to check for matched patterns. The number of elements in an Output_Table corresponds to the number states in the state machine. A positive element indicates a final state of matched pattern and 0 indicated all other internal states. The PAC_Table after optimization is called Brow.

ISSN(Online): 2320-9801
ISSN (Print): 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering

(An ISO 3297: 2007 Certified Organization)

Vol. 3, Issue 6, June 2015

Steps:

```
1: Read all the patterns P= [0, 1,..., M-1] (Virus Signatures)
2: Read number of input packets..
3: Construct PAC_Table and Output_Table.
4: Construct a Parallel Aho-Corasick State machine.
5: Reduce the sparse PAC_Table to Banded row format, called Brow.
6: Allocate GPU memory for Parallel Aho-Corasick state machine and the number of packets.
7: Allocate GPU memory for storing Matched pattern data and Brow by directly storing them in pinned memory.
8:  Copy DFA and input packet data to GPU.
9:  Launch a GPU kernel
10: Make an asynchronous call to CPU function to perform pattern matching instead of sitting idle.
11:  GPU Kernel
12:  Begin
13:  Read the packet data
14:  Calculate the length of the packet data, L.
15:  Assign each thread to one character of the packet data.
16:  currentState = 0
17:  patternlength=0
18:   numPatterns = 0
19:   position=0
20:  for cursor = beginning at the allocated character in the input string of a packet to the end of the string, L.
21:   if Brow[current state] [packet[cursor]] ≠ 0   then
22: currentState = Brow[current state] [packet [cursor]]. nextState
23:   patternLength = patternLength+l
24:   if Output_Table [current state]. isFinal = 0 then
25:   current state will not change
26:   else
27:   matchState = currentState
28:   numPatterns = numPatterns +1
29:   position= cursor
30:   else
31:   patternLength = 0
32:   currentState = 0
33:   End
34:   Read data from GPU
35:   Read data from CPU
36:   Display matchState, numPatterns, position.
```

IV. EXPERIMENTAL SETUP

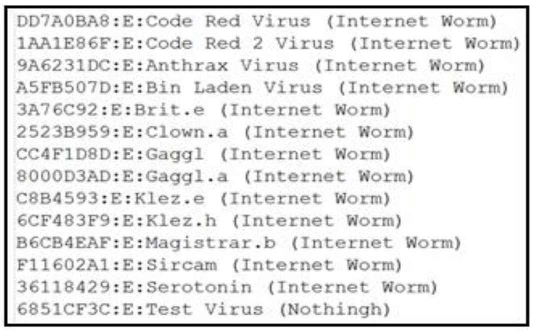
Here, Intel Ci3 CPU, NVIDIA Geforce GTX 680 GPU with NVIDIA CUDA 6.5 programming model are used. GTX 680 GPU has Kepler architecture [13] and is considered to be a very fast and efficient GPU. TABLE I shows the specifications of NVIDIA Geforce GTX 680 GPU [14]. And NVIDIA CUDA 6.5 programming model is the latest version of most pervasive parallel computing platform and programming model [15]. It is available as a free download at www.nvidia.com/getcuda. It provides several new features and are explained in detail in [16]. And the tools used are NVIDIA Nsight Eclipse and NVIDIA visual profiler [17] [18].

V. DATA SETS

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 have also used a SNORT virus signature whose format is, Format: Virus Signature: Code Name :Virus Name(Type) as shown in the Fig. 8 and the data set used for pattern matching is as shown in the TABLE II, 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 I
NVIDIA GFORCE GTX 680 GPU SPECIFICATIONS

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 Configuration	2048MB
Memory Interface Width	256-bit GDDR5
Memory Bandwidth (GB/sec)	192.2
Bus Support PCI Express	3.0
Certified for Windows 7	yes



```
DD7A0BA8:E:Code Red Virus (Internet Worm)
1AA1E86F:E:Code Red 2 Virus (Internet Worm)
9A6231DC:E:Anthrax Virus (Internet Worm)
A5FB507D:E:Bin Laden Virus (Internet Worm)
3A76C92:E:Brit.e (Internet Worm)
2523B959:E:Clown.a (Internet Worm)
CC4F1D8D:E:Gaggl (Internet Worm)
8000D3AD:E:Gaggl.a (Internet Worm)
C8B4593:E:Klez.e (Internet Worm)
6CF483F9:E:Klez.h (Internet Worm)
B6CB4EAF:E:Magistrar.b (Internet Worm)
F11602A1:E:Sircam (Internet Worm)
36118429:E:Serotonin (Internet Worm)
6851CF3C:E:Test Virus (Nothingh)
```

Fig. 8 Snort .hdb File Signature format

VI. RESULTS

In this section, the actual results obtained from the comparison of CPU, Unoptimized GPU and optimized GPU implementations are presented. The CPU used for the experiments was a 2.8 GHz Intel CORE I5 processor with 4 cores, 8 GB total memory. The GPU used for the implementation was a GTX 680 device with 1536 CUDA Cores. The core clock can be boosted up to 1058 MHz (from 1006 base clock) and the memory is clocked at 3004 MH. The performance of Network Intrusion Detection using GPU was measured using various benchmarks. Fig 9, 10 and 11 shows the Serial and Parallel implementation comparisons of Multipattern matching algorithm. It is observed that among all implementations of multipattern matching algorithm, Optimized Parallel Aho-Corasick Algorithm requires very less searching time for pattern matching.

TABLE II
DATA SETS USED IN PATTERN MATCHING

Name of Data Set	Description
P2by100	It Contain 100 patterns having length 2
P2by1000	It Contain 1000 patterns having length 2
P2by5000	It Contain 5000 patterns having length 2
P32by50000	It Contain 50000 patterns having length 32
Cuda500	It Contain 500 String as Data Packet
Cuda5000	It Contain 5000 String as Data Packet
Cuda50000	It Contain 50000 String as Data Packet
Cuda500000	It Contain 500000 String as Data Packet

ISSN(Online): 2320-9801
ISSN (Print): 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering

(An ISO 3297: 2007 Certified Organization)

Vol. 3, Issue 6, June 2015

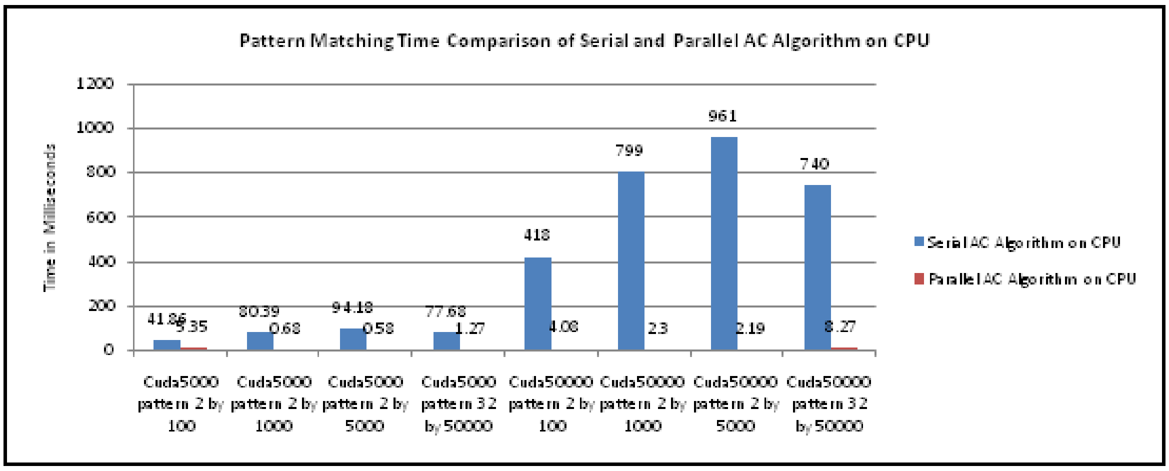


Fig. 9 Pattern Matching Time Comparison of Serial and Parallel AC Algorithm on CPU.

Fig. 9 shows the variation in total run time for Serial and Parallel CPU implementation of AC Algorithm for a variable number of packets and variable number of patterns. We can observe that, by parallelizing the AC algorithm on CPU itself results in much less running time than its serial implementation.

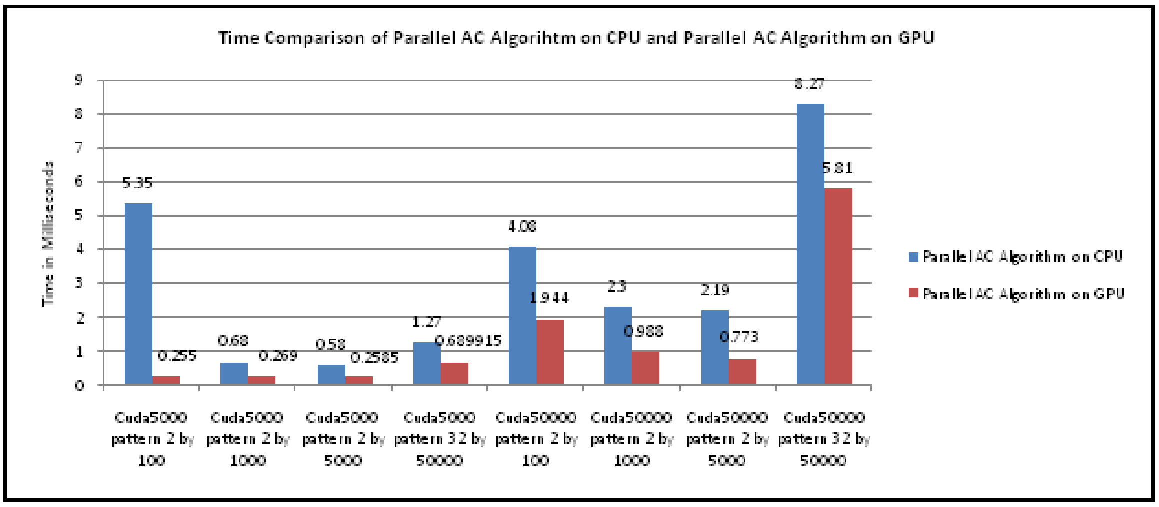


Fig. 10 Pattern Matching Time Comparison of Parallel AC Algorithm on CPU and Parallel AC Algorithm on GPU.

Fig. 10 shows Search Time Comparison of Aho-Corasick algorithm, implemented in CUDA and implemented in OPEN MP. OPEN MP is used for multi-threaded parallel processing and work on shared-memory multi-processor (core) computers. It has been observed that searching time of CUDA implemented Aho-Corasick algorithm is much less than OPEN MP implemented Aho-Corasick algorithm, for same number of patterns and data packets. It is observed that GPU is twice as fast as CPU on an average.

Fig. 11 shows Search Time Comparison of Optimized Parallel Aho-Corasick algorithm using CUDA, Unoptimized Aho-Corasick algorithm using CUDA, Parallel AC implementation using OPEN MP. It has been observed that Optimized Parallel Implementation of Aho-Corasick Algorithm further reduces the running time required by the Parallel Un-Optimized Aho-Corasick algorithm and is thus faster than its Unoptimized CUDA and OPEN MP implementations. Searching time is shown in Microseconds for variable number of packet and variable number of patterns.

ISSN(Online): 2320-9801
ISSN (Print): 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering

(An ISO 3297: 2007 Certified Organization)

Vol. 3, Issue 6, June 2015

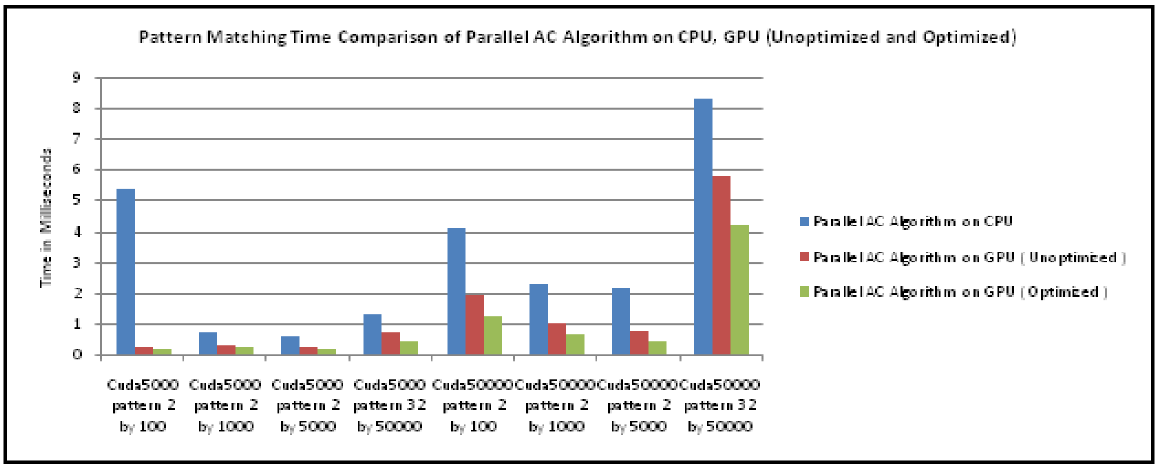


Fig. 11 Pattern Matching Time Comparison of Parallel AC Algorithm on CPU and Parallel AC Algorithm on GPU (Unoptimized and Optimized).

TABLE III shows the pattern matching time comparisons of four different versions of AC Algorithm implemented on CPU and on GPU for a variable number of packets and variable number of patterns. We can see that Optimized Parallel AC Algorithm takes less time for execution than its other implementations.

TABLE IIII
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 (Time in ms)**	**Parallel AC Algorithm on CPU (Time in ms)**	**Parallel AC Algorithm on GPU (Unoptimized) (Time in ms)**	**Parallel AC Algorithm on GPU (Optimized) (Time in ms)**
P2by100	5000 words	41.86	5.35	0.255	0.2
P2by1000	5000 words	80.39	0.68	0.269	0.242
P2by5000	5000 words	94.18	0.58	0.258	0.157
P32by50000	5000 words	77.68	1.268	0.689	0.435
P2by100	50000 words	418	4.078	1.944	1.2
P2by1000	50000 words	799	2.30	0.988	0.648
P2by5000	50000 words	961	2.186	0.773	0.428
P32by50000	50000 words	740	8.273	5.81	4.2

VII. CONCLUSION AND FUTURE WORK

Importance of Network Intrusion Detection Systems is increasing as new threats and viruses invade the network each day and more intrusion signatures are added to the existing rule set. The speed of the pattern matching algorithm is therefore one of the main concerns in the Network Intrusion Detection Systems. With the advent of CUDA several attempts have been made to parallelize the existing algorithms as well as to develop other new algorithms that work best with CUDA architecture. 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 and memory usage required to run the algorithm on GPU. Serial and parallel implementation results of AC algorithm on CPU and parallel Unoptimized and Optimized implementations of AC algorithm on GPU are provided. Our results show that Optimized Parallel AC Algorithm run on GPU takes less time for execution than Unoptimized Parallel AC algorithm on GPU and with twice to four-fold the speed than serial and parallel algorithm on CPU.

There is a room for improvement in this work. Every time a new GPU card is released with improved computational features, the horizon further advances. As future work, this application can be ported to multiple GPU devices that will run in parallel.

ACKNOWLEDGMENT

We are glad to express our sentiments of gratitude to all who rendered their valuable guidance to us. We would like to express our appreciation and thanks to Prof. Dr. P. C. Kulkarni, Principal, G. E. Ss. R. H. Sapat College of Engg., Nashik. We are also thankful to Prof. N. V. Alone, Head of Department. Computer Engg., G. E. Ss. R. H. Sapat College of Engg., Nashik.

I would also like to thank Prof. A. S. Vaidya my project guide for her invaluable guidance. We thank the anonymous reviewers for their reviews and suggestions.

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, pp. 1906-1916, 2013.
2. C.H. Lin, S.Y. Tsai, C.H. Liu, S.C. Chang, and J.M. Shyu, “Accelerating String Matching Using Multi-threaded Algorithm on GPU”, Proc. IEEE GLOBECOM, 2010.
3. Snort, https://www.snort.org/
4. J. Yu and J. Li, “A Parallel NIDS Pattern Matching Engine and Its Implementation on Network Processor”, Proc. Int’l Conf. Security and Management (SAM), 2005.
5. G. Vasiliadis and S. Ioannidis, “GrAVity: A Massively Parallel Antivirus Engine”, Proc. 13th Int’l Conf. Recent Advances in Intrusion Detection (RAID ’10), 2010.
6. Graphics Processing Unit Definition, http://en.Wikipedia.org/wiki/Graphicsprocessingunit
7. DONALD E. KNUTH, JAMES H. MORRIS, AND VAUGHAN R. PRATT, “Fast Pattern Matching in Strings”, SIAM J. COMPUT. Vol. 6, No. 2, pp. 323-350, 1977.
8. KMP algorithm by example, http://www.cs.utexas.edu/users/moore/best-ideas/string-searching/kpmexample. htmlstep01
9. 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.
10. 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, pp.1596-1609, 2011.
11. ClamAV antivirus software, http://www.clamav.net
12. A. Tumeo, O. Villa, and D. Sciuto, “Efficient Pattern Matching on GPUs for Intrusion Detection Systems”, Proc. Seventh ACM Intl Conf. Computing Frontiers, 2010.
13. NVIDIA Kepler Architecture, http://www.nvidia.com/object/nvidia-kepler.html
14. Geforce GTX680 Specification, http://www.geforce.Com/hardware/desk top-gpus/geforce-gtx-680/specifications.
15. About CUDA 6.5, http://www.scientific-computing.com/press releases/productdetails.php? productid=1935/
16. CUDA 6.5 features, http://devblogs.nvidia.com/parallelforall/10-ways-cuda- 6-5-improves-performance-productivity/
17. Performance Analysis Tools, https://developer.nvidia.com/performance-analysis-tools
18. CUDA Toolkit Documentation, http://docs.nvidia.com/cuda/index.html cudacbestpractices
19. OpenMP, http://openmp.org/wp/
20. GCC, http://gcc.gnu.org/