5-2012

Parallelizing a network intrusion detection system using a GPU.

Anju Panicker Madhusoodhanan Sathik 1984-
University of Louisville

PARRELLELIZING A NETWORK INTRUSION DETECTION SYSTEM USING A GPU

By

Anju Panicker Madhusoodhanan Sathik
B.Tech. University of Kerala, India, 2006

A Thesis
Submitted to the faculty of the
J.B. Speed School of Engineering of the University of Louisville
In Partial Fulfillment of the Requirements
for the Degree of

Master of Science

Department of Computer Science and Engineering
University of Louisville
Louisville, Kentucky

May, 2012

PARRELLELIZING A NETWORK INTRUSION DETECTION SYSTEM USING A GPU

By

Anju Panicker Madhusoodhanan Sathik
B.Tech., University of Kerala, India, 2006

A Thesis Approved on

April 11, 2012

by the following Thesis Committee:

_____
Adviser - Ming Ouyang, Ph.D.

_____
Ahmed H. Desoky, Ph.D.

_____
Ibrahim Imam, Ph.D.

_____
Roman Yampolskiy, Ph.D.

_____
Jeff Hieb, Ph.D.

ACKNOWLEDGMENTS

I owe my deepest gratitude to my adviser Dr. Ming Ouyang for his immense help and guidance, without which it would have been impossible to complete this thesis. I like to thank all my professors, especially Dr. Desoky, Dr. Imam, and Dr. Yampolskiy for their invaluable support and faith that has helped me reached this far.

I thank Dr. Jeff Hieb for his timely advice and insights, which provided the momentum when it was most needed, as well as for agreeing to serve in my committee. I thank Mr. James Murphy for setting up the honey pot environment, which was crucial in obtaining the results.

I am most grateful to Martin Roesch, Nandakumar, and Russ Combs from Snort developers, and Giorgos Vasiliadis from Gnort developers, for helping me with my queries and doubts.

Above all, I would like to thank my husband Jyothish for his personal support and encouragement that has helped me to hold on during trying times. I also thank my family for their unequivocal support, as always, for which I still remain indebted to.

ABSTRACT

PARRELLELIZING A NETWORK INTRUSION DETECTION SYSTEM USING A GPU

Anju Panicker Madhusoodhanan Sathik

April 11, 2012

As network speeds continue to increase and attacks get increasingly more complicated, there is need to improved detection algorithms and improved performance of Network Intrusion Detection Systems (NIDS). Recently, several attempts have been made to use the underutilized parallel processing capabilities of GPUs, to offload the costly NIDS pattern matching algorithms. This thesis presents an interface for NIDS Snort that allows porting of the pattern-matching algorithm to run on a GPU. The analysis show that this system can achieve up to four times speedup over the existing Snort implementation and that GPUs can be effectively utilized to perform intensive computational processes like pattern matching.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

I. INTRODUCTION

Network intrusions are a real and serious threat to most organizations and hence have been a focus of study for over two decades. There have been numerous efforts to develop applications that detect intrusions or prevent such activities. However, majority of the widely available software packages suffer a serious defect: the time delay in detecting an intrusion after its onset. Recently, with the advent of CUDA enabled GPU computing, research to improve the speed of intrusion detection systems using GPUs is receiving a significant amount of attention. In this thesis an open source Network Intrusion Detection and Prevention software package, Snort®, is subjected to parallelization and ported to run on NVIDIA C2050/C2070 Tesla GPU. The performance of this GPU augmented Snort is evaluated under a variety of conditions and its performance are compared with the existing CPU serial implementation (using AMD Phenom II X4 965 processor), and the results are presented.

1.1 Network Intrusion Detection and Prevention System

A **Network Intrusion Detection System** (NIDS) is an application that monitors the network for any unauthorized accesses into the network. The application monitors the network for violation of access permissions or other malicious activities. An **Intrusion Prevention System** blocks or prevents an intrusion. Intrusion detection and prevention

are sometimes combined to form an Intrusion Detection and Prevention System (IDPS). Snort is one such IDPS software on which this thesis is based.

1.2 Snort

Snort is a signature based Intrusion Detection Prevention software package that performs real time network traffic analysis and logs the output. A signature is any pattern in the packet data that identifies a possible intrusion. The incoming/outgoing packets in the network are analyzed and the packet data are subjected to pattern search. Depending on the presence of a signature and the position of its occurrence inside the packet, appropriate actions like alert, log, pass, drop etc. are taken for the packet.

Snort can be configured to run in three different modes:

1. **Sniffer** mode: in this mode Snort reads the packets from the network and displays them on the screen. It can be configured to display just the protocol headers, or to display the entire packet including headers and the packet data.
2. **Packet Logger** mode: in this mode snort can be used to record all the network traffic into a file. It can be configured to log the network traffic to and from specific subnets or specific ports.
3. **Network Intrusion Detection System (NIDS)** mode: This is the most complex mode and allows matching packets against a user defined set of rules and performing several actions like drop, pass, alert etc. based on what it sees.

Every mode uses a configuration file *snort.conf* to set up its running environment. The configuration file is used to define the network addresses, a set of rules which Snort will apply to network packets, the desired type of output (such as: original ASCII coded

format or a binary log file), and several other run modes in which snort can be configured to work. Run modes can also be specified as command line options when starting Snort, and command line options override any of the options specified in the configuration file.

1.3 Fermi Architecture on NVIDIA's Tesla GPU using CUDA

Compute Unified Device Architecture (CUDA) is a massively parallel computing architecture that allows a heterogeneous co-processing computing model between a GPU (Graphics Processing Unit) and a CPU. The sequential parts of the application run on the CPU and the computation intensive parts are accelerated by the GPU. The GPU contains hundreds of processor cores, which are capable of running many thousands of parallel threads that work together to achieve high throughputs. Applications that leverage the CUDA architecture can be developed in a number of different languages including C, C++, Fortran, OpenCL, and DirectCompute.

The latest generation CUDA architecture is called **Fermi**, first released in 2010. Fermi is optimized for scientific applications with key features like over 500 gigaflops of IEEE standard floating point hardware support that provides a fused multiply-add instruction for both single and double precision arithmetic operations, L1 and L2 caches, coalesced memory access, local user managed data caches in the form of shared memory dispersed within the GPU and ECC or Error Checking and Correction that protects the memory from soft errors caused by external electromagnetic interferences.

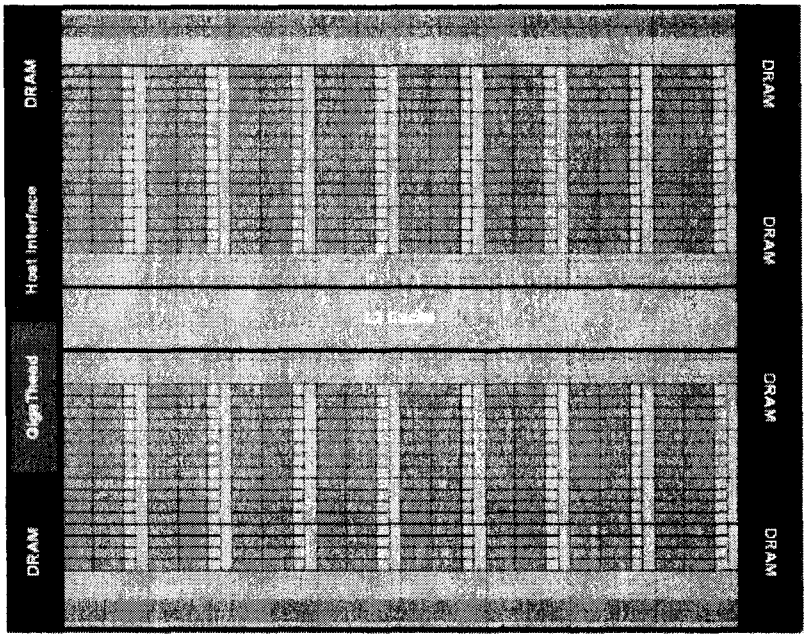

Figure 1: Block Diagram of Fermi Architecture [Source: Nvidia]

Figure 1 shows the block diagram of Fermi architecture of NVIDIA C2050/C2070 Tesla GPU. It consists of 448 CUDA cores which are organized into 14 Streaming Multiprocessors (SM), with each SM consisting of 32 cores. A CUDA core executes one floating point or integer instruction per clock cycle for a thread. It uses a two level distributed thread scheduler called the **GigaThread** thread scheduler. CUDA threads have access to multiple memory spaces during code execution. All threads have access to 3GB global memory space. All threads within the same block have access to the same configurable shared memory (up to 48KB per SM) during the lifetime of the corresponding block. The shared memory and L1 cache together is 64KB, and this 64KB can be configured as 48KB shared memory and 16KB L1 cache or 16KB shared memory and 48KB L1 cache. If shared memory is not used it automatically defaults to 16KB shared memory and 48KB L1 cache. Fermi supports a 768KB unified L2 cache that

services all load, store, and texture operations. It enables efficient high-speed data sharing across the GPU. Figure 2 demonstrates the Fermi memory hierarchy. The L1 cache enables high speedup in execution of programs whose memory accesses are not known beforehand.

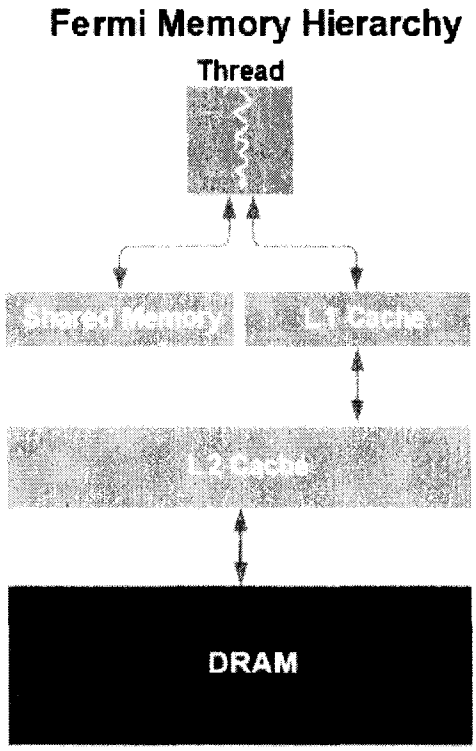

Figure 2: Fermi Memory Hierarchy [Source: Nvidia, Fermi Architecture White Paper]

The focus of this thesis is on accelerating the performance of Snort by porting Snort's string searching algorithm to run on a GPU. A GPU adaptation of the Aho-Corasick algorithm is implemented and incorporated into the Snort code. The rest of the thesis is organized as follows: Chapter 2 is a literature survey of other publications related to this work. Chapter 3 presents various methods and algorithms that form the basis of this work. Chapter 4 discusses the actual implementation details and presents results of the experiments conducted. Chapter 5 presents conclusions and directions future work.

II. LITERATURE SURVEY

The performance of intrusion detection systems is heavily dependent on pattern matching, as millions of packets must be examined at Mbps or Gbps. A large number of pattern matching algorithms have been developed which have found use in a variety of fields including bioinformatics, network security, and forensics where large amount of data have to be analyzed for pattern matches.

Pattern matching algorithms may be classified into single or multi pattern search algorithms. The Boyer Moore algorithm [23] is a single pattern search algorithm that searches for a pattern of length m in the text. The Boyer Moore algorithm uses some simple heuristics to improve performance and for a text of length n, it has an average performance of $O(n/m)$ comparisons. In the Knuth-Morris-Pratt single pattern search algorithm [11], the authors describe a method in which the performance can be marginally improved by relying on the information gained by previous symbol comparisons. By making use of the information gained by previous symbol comparisons, KMP avoids re-comparison of any text symbol that has matched a pattern symbol. The Knuth-Morris-Pratt algorithm has an average complexity of $O(m+n)$.

Multi pattern search algorithms search the text for a set of patterns simultaneously, and their performance is independent of the number of patterns being searched. This is achieved by building an automaton from all the patterns. The automaton can be a table, a tree or a combination of both. Each character in the text needs to be

examined only once for all the patterns together. Several algorithms have been developed for multi pattern searches. The Wu- Manber [22] algorithm makes use of the text shifting in the Boyer Moore algorithm, and proposes the use of a hash table and a prefix table to determine the candidate pattern for a match and to verify the match. The Aho-Corasick algorithm [1] makes use of a non-deterministic (NFA) or deterministic finite (DFA) automaton to perform simultaneous pattern matching, and thus its performance is independent of the number of patterns, and is linear in the lengths of the patterns plus the length of the test string.

Snort [36] uses the Aho-Corasick algorithm to perform a multi pattern search on the network packets. It first constructs an NFA, and then converts that NFA to a DFA with a reduced number of states. Several attempts have been made to improve the performance of Snort using a GPU [4, 6, 9, 10], by parallelizing the pattern matching algorithm.

2.1 Snort Architecture

Snort's intrusion detection functionality is achieved with the five main components, which is illustrated in Figure 3. Snort relies on an external packet capturing library *libpcap* to sniff the network packets. The raw packets are then fed to the **Packet Decoder**. The packet decoder can be considered as the first main component of the snort architecture. The packet decoder mainly segments the protocol elements of the packets to populate an internal data structure. These decoding routines are called in order through the protocol stack, from the data link layer up through the transport layer, finally ending at the application layer. Once the packet decoding is complete, the traffic is passed over

to the **Preprocessors** for normalization, statistical analysis and some non-rule based detection. Any number of preprocessor plugins can examine or manipulate the packets and then passes them over to the next component, the **Detection Engine**. The detection engine scrutinizes each packet data and search for intrusion signatures. The **Logging and Alerting** system either logs the packet information to a file or sends alerts through the output plugins. The last component of Snort is the **Output Plugins**, which generates the appropriate alerts to the present suspicious activity to the user.

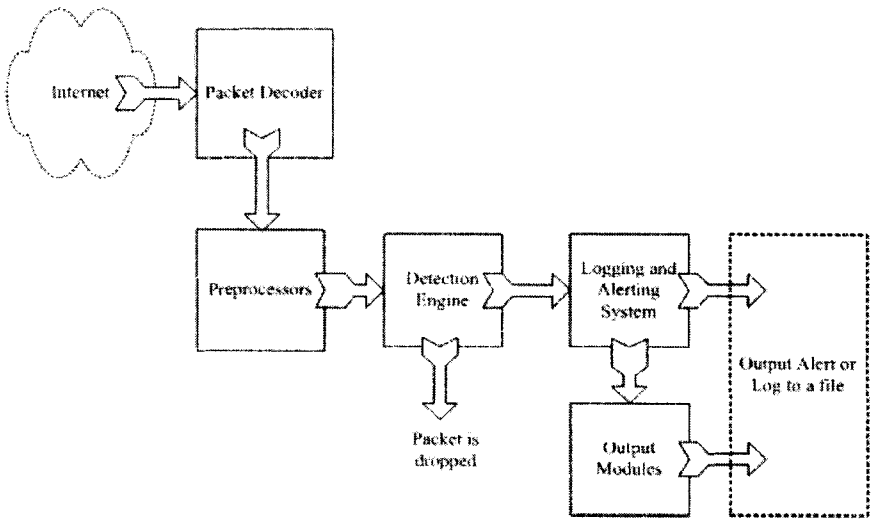

Figure 3: Block Diagram of Snort (Source: [35])

2.2 Programming in CUDA

CUDA SDK uses an extended C language that allows the user to program using the CUDA architecture. A user defined C function that is executed in the GPU is called a *kernel.* A set of parallel threads, which are organized into thread blocks and grids of thread blocks, execute the kernel concurrently. The programmer specifies the number of

times the kernel has to be executed by specifying the number of threads in the program. Each thread executes one instance of the kernel. So, if the user specifies the number of threads as N, the kernel will be executed N times by N different threads. CUDA follows a Single Instruction Multiple Thread (SIMT) programming model. The Fermi architecture also supports concurrent global *kernel* execution by allowing up to 16 kernels to execute simultaneously. The limitation with executing multiple kernels is that all kernels must belong to the same program, as CUDA cannot manage application level parallelism.

2.3 GPU Thread Architecture

The massive parallelism in the CUDA programming model is achieved through its multi-threaded architecture. This thread parallelism allows the programmer to partition the problem into coarse sub problems that can be processed in parallel by blocks of threads, and each sub problem is further divided into finer pieces that can be solved cooperatively in parallel by all threads within a block. The CUDA threads are organized into a two-level hierarchy using unique coordinates called *block ID* and *thread ID*. Each of these threads can be independently identified within the *kernel* using its unique identifier represented by the built-in variable *blockIdx* and *threadIdx*.

The programmer can configure the number of threads required in a thread block, with a maximum of 1024 threads per block. An instance of the kernel is executed by each of these threads.

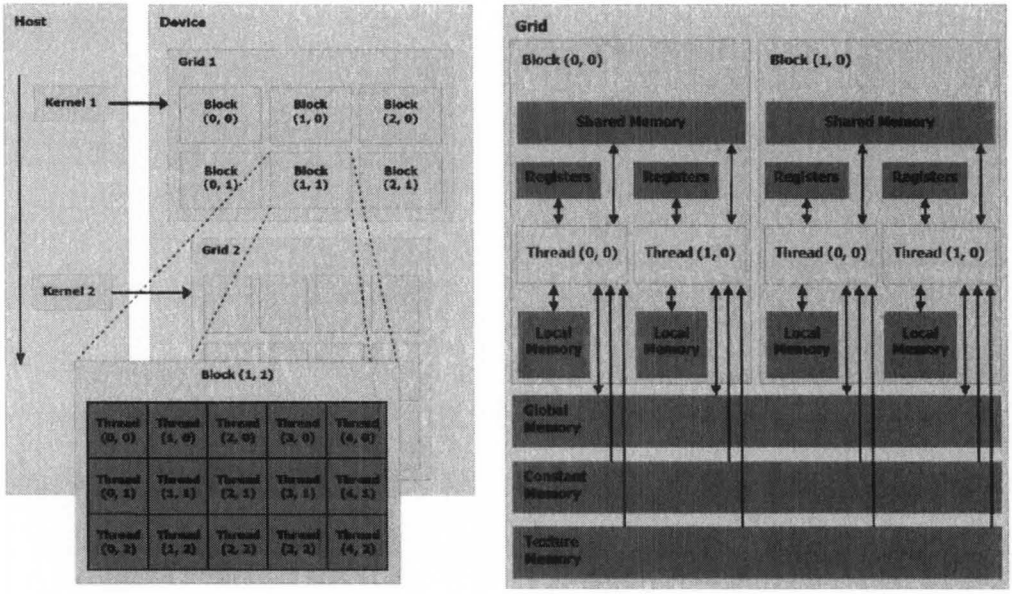

Figure 4: CUDA Thread Organization [Source: Nvidia]

A group of 32 threads with consecutive thread IDs is called a *Warp*, which is the unit of thread scheduling in SMs. The Fermi architecture supports 16 SMs each of which can track a total of 48 warps simultaneously resulting in a total of 24,576 (16 x 32 x 48) concurrent threads on a single chip.

Gnort [14] explores two methods of configuring the GPU threads, both of which achieve a speedup by a factor of two. One approach is to assign a single packet to each multiprocessor at a time, and the second approach is to assign a single packet to each stream processor at a time. In the first approach, each packet is divided into 32 equal chunks, which are concurrently processed by the 32 threads of a warp in parallel. Let X be the maximum pattern length in the state table. To handle correctly the patterns that span over consecutive chunks, each thread searches X bytes in addition to the chunk assigned to it. This chunk overlapping requires extra processing, which introduces overhead in execution. In the second approach, each packet is processed by a different

thread. Let Y be the total number of packets sent in a batch to the GPU. If the GPU has N multiprocessors, N thread blocks are created, and each thread block processes Y/N packets. In this thesis, Snort is subjected to parallelization along the lines of *Gnort*'s second approach, where a single packet is analyzed by a single thread.

In a later publication by Vasiliadis *et al.* [15], the performance was improved by 60% by implementing regular expression matching on the GPU. Regular expressions are more expressive and flexible than byte patterns, and several patterns can be combined to form a single regular expression. Similar to byte patterns, regular expression matching can also be parallelized using GPU. The GPU adaptation for pattern matching is applied to web pages to obtain 28 times peak performance as explained in [19]. Lin *et al.* [7] proposed a novel parallel algorithm *Parallel Failureless-AC* algorithm (PFAC) to speed up string matching, and is found to be 4,000 times faster than the existing Snort. In the PFAC algorithm, a trie similar to the Aho-Corasick algorithm is constructed but with the failure states removed. Each byte in the input packet is assigned a thread in the GPU, which searches for any signature patterns starting at that byte.

III. METHODS

This Chapter explains the methods used by Snort that are relevant to the implementation. These are the functionalities of Snort that are modified or dealt with in the new implementation.

3.1 Packet Capture and Preprocessing

The first phase of any network intrusion detection system is packet capturing. All data in the network are transmitted in the form of a packet, which comprises of a packet header, packet data, and sometimes, a trailer. The packet header consists of several Open Systems Interconnection (OSI) layer information, checksums, fragmentation flags and offsets, source and destination IP addresses, source and destination port numbers, etc.; the packet data consists of the payload [6]; the trailer contains end of packet and error checking codes. The OSI model is a 7 layer network architecture (Physical Layer, Data Link Layer, Network Layer, Transport Layer, Session Layer, Presentation Layer and Application Layer) model which standardizes the functions of a communication system in terms of abstraction layers. Packets in the network first reach the Network Interface Card (NIC) of a computer, which when operated in *promiscuous mode* passes all packet frames to the CPU rather than just those addressed to the NIC's MAC address. *Libpcap* is a platform independent open source library used to capture and process raw network packets. The raw packets thus captured are processed to extract the source and destination

addresses, source and destination ports, protocol information, and the packet payload all of which are essential for detecting intrusions. The information that is extracted is stored for comparison and reassembling the packet later. For IDS it is important to reassemble fragmented packets before detection because fragmentation can be used to hide attacks from signature based intrusions. One part of the signature may be in one fragment and the other part on another fragment. Hence the preprocessors play a vital role in de-fragmenting the packets and later reassembling the data before delivering it to the intended recipient.

3.2 Signature Detection

Signatures or Rules are vital to the efficiency of Snort as a Network Intrusion Detection System. Most known intrusions have a signature or pattern, and Snort uses them to identify whether the received packet is part of an intrusion or not. Snort has a set of attack signatures that are read line-by-line, parsed and loaded into an internal data structure when the service begins. Every incoming packet is then inspected and compared with these rules. When an intrusion is detected, appropriate actions are taken for the packet. Every time a new intrusion is reported, a rule that identifies that intrusion is created and added to the existing set. Every rule starts with an **Action**, which is the action to be performed if that rule is matched. Current rule actions are:

- Alert – Generate an alert and then log the packet.
- Log – Generate a log entry.
- Pass – Ignore the packet.
- Activate – Alert and turn on dynamic rules.

- Dynamic – After activated by the Activate rule, act as a log rule.
- Drop – Make iptables drop the packet and log the packet.
- Reject – Make iptables drop the packet, log it, and then send an unreachable message if the protocol is User Datagram Protocol (UDP).
- Sdrop – Make iptables drop the packet but do not log it.

A typical Snort rule consists of two main components: the rule header and the rule options. The rule header comprises of protocols, variables, and ports. The rule options include parameters like rule title, flow, content, depth, offset, etc. Figure 5 shows an example of a Snort rule.

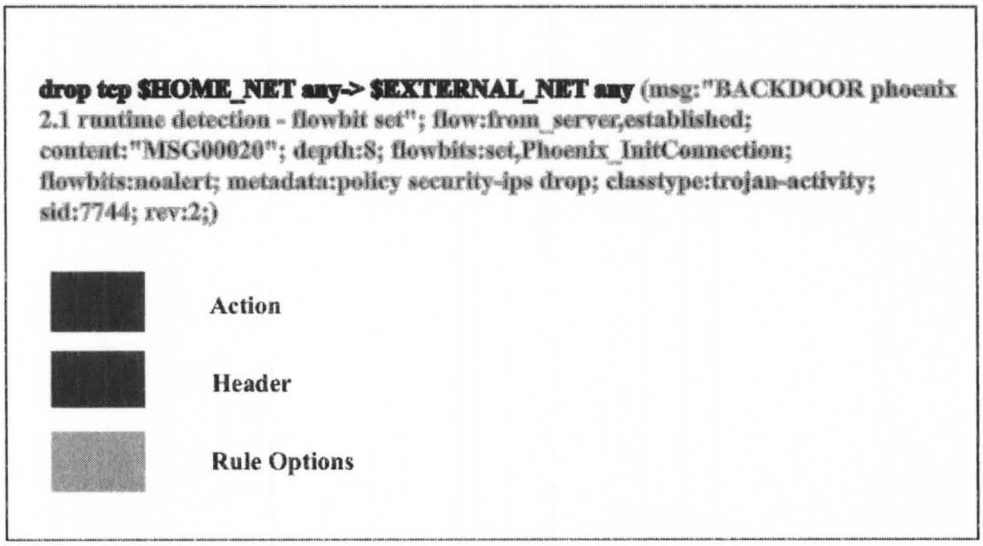

Figure 5: Signature or Rule

In the rule in the example illustrated by Figure 5, the action to be taken is 'drop'. The protocol here is '*tcp*'. Other protocols identified by the NIDS are '*ip*', '*udp*', and '*icmp*'. Next part of the header is the Source and Destination IP addresses. In the above

example the source IP is HOME_NET and the destination IP is EXTERNAL_NET. EXTERNAL_NET and HOME_NET are variables, the values of which can be set in a configuration file. The next parameter in the rule is the source and destination port numbers. In the above example, the Source port and Destination port values are set as *'any'*. This means that the rule can be applied to all packets irrespective of the port numbers to which it is sent or received, provided the remaining parts of the header match. The *direction* in the signature tells in which way the signature has to match. This means that only packets with the same direction as that of the rule can match. The direction of traffic in which the above rule will be active is from source to destination. The direction can be '*left to right* (->)', '*right to left* (<-)' or '*both* <>'.

The second part of the rule is the rule options. The options in a rule may include '*msg*', '*sid*', '*content*', '*uricontent*', '*flow*', '*depth*', '*offset*', '*within*', etc. Each of these keywords is supplied with a value. The value for '*msg*' will be the rule title that will be logged if a packet is matched with that rule; '*sid*' will have the unique rule id for each rule; '*content*' denotes the pattern that is to be searched for in the payload and '*uricontent*' is the pattern that is to be searched for in the request-uri; '*flow*' helps to control load by limiting the search to a certain type of stream; '*depth*', '*offset*', and '*within*' specify the location of the particular pattern inside the payload.

3.3 Snort's Multi Pattern Search (Aho-Corasick)

Snort requires a pattern matching system that can search for thousands of patterns in relatively small packets at very high speeds. This can be achieved with multi pattern search algorithms like Wu-Manber or Aho-Corasick. The latest version of Snort uses the

Aho-Corasick algorithm, as it is slightly faster and less sensitive to the size of the pattern being searched. The Aho-Corasick algorithm uses a Deterministic Finite Automata (DFA) for performing the multi pattern search.

The Aho-Corasick (AC) algorithm was developed by Alfred V. Aho and Margaret J. Corasick in 1975 [1]. The AC algorithm works by constructing a finite state pattern-matching machine from the set of keywords to be searched. This machine is then used to process the input text string in a single pass. The finite state pattern-matching machine is basically a finite automaton that is built from the keywords. Figure 6 shows an example of a pattern matching machine that is built from the keywords P = {he, she, his, hers}. The final states will be 2, 5, 7, and 9.

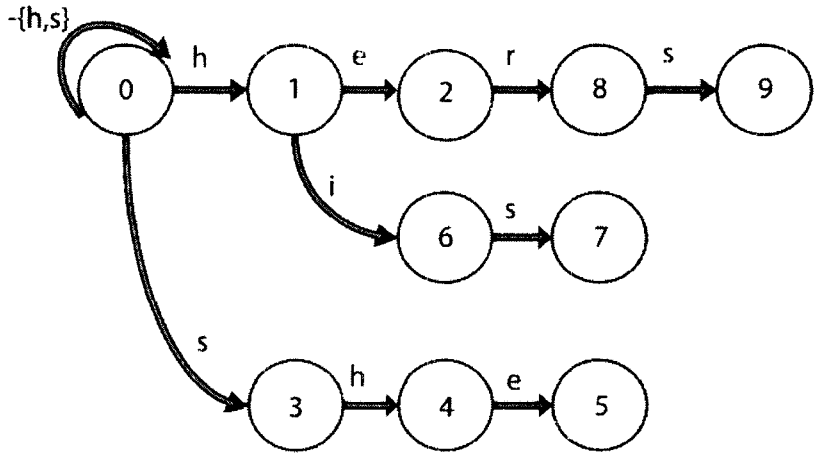

Figure 6: Example of a DFA (From [1])

The pattern machine is constructed by starting at the root node and inserting each pattern one after the other. The algorithm works as follows:

- Start at the root node.
- For each pattern in P

- If the path ends before the pattern, continue adding edges and nodes for the remaining characters in the pattern.
- Once the pattern is identified mark it as the final state.

The time taken for the search is linearly proportional to the length of the pattern being searched. The search algorithm is similar to the above one.

- Start at the root node.
- For each character in the text, follow the path led by the trie
  - If it is a final state node, the pattern is present in the text.
  - If the path terminates before the text, the pattern is not present in the text.

In the Aho-Corasick automaton the actions are determined by three functions:

1. The *goto* function g(q,a) is the next state from the current state q, on receiving symbol 'a'.
2. The *failure* function f(q), for $q \neq 0$, is the next state in case of a mismatch.
3. The *output* function out(q) gives the set of patterns found at state q.

The Aho-Corasick algorithm as explained in [1] is illustrated below.

Input: A text string $x = a_1 a_2 \ldots a_n$ where each a_i is an input symbol and a pattern matching machine M with *goto* function *g*, *failure* function *f*, and *output* function *out*, as described above.

Output: Locations at which keywords occur in x.

Method:

```
begin
        state ← 0
        for i ← 1 until n do
          begin
             while g(state, a_i) = fail do state ← f(state)
               state ← g(state, a_i)
               if out(state) ≠ empty then
               begin
               print i
               print out(state)
               end
          end
end
```

IV. IMPLEMENTATION

The Detection Engine that performs the signature matching handles the most computationally intensive process in Snort. Around 75% of the total execution time is spent in signature matching process [16]. Therefore the speed of execution can be considerably increased if the signature matching process is accelerated through parallelization. We aim to achieve this by porting the string-matching algorithm used in Snort, Aho-Corasick, to run on a GPU. The AC algorithm relies on a set of DFAs for the string comparison. These DFAs also need to be transferred to the GPU memory for the string comparison.

The new design is incorporated into the existing Snort source code. The basic components of Snort can now be re-organized to three main components: Packet Capture and Buffering, Transferring the DFAs and Packets to the GPU, and Perform Pattern Matching and Obtain the Outputs.

4.1 Packet Capture and Buffering

As mentioned in section 3.1, Snort uses the external packet capturing library *libpcap* to sniff the packets in the network. These packets are processed by the preprocessing component of Snort before any analysis. The network addresses that need

to be monitored are explicitly stated in the Snort configuration file. Snort captures and analyses the packets one by one serially. In this implementation, the same library is used for capturing the network packets. After a number of packets are captured, the parallelized pattern-matching algorithm is applied to all of these packets simultaneously in the GPU. To achieve this, the incoming packets have to be buffered. A separate packet buffering scheme is implemented and incorporated in Snort that groups the incoming packets into buffers.

Snort reads the entire set of rules and classifies them into different groups based on their source and destination IP addresses and port numbers. The rule contents and uricontents are then extracted to construct the DFAs that are used by the Aho-Corasick algorithm to perform string matching. Snort does not assign an identifier to a rule group and the associated DFA. The different rule groups in the present implementation are assigned unique group identifiers. The source and destination IP addresses and port numbers of the incoming packets are observed and the rule group to which it belongs is determined. A separate buffer is created for each rule group. The buffer size is made to vary from 32, 64, ..., to 4096 for different numbers of input packets. Packets that fall in the same group are copied to the corresponding packet buffer. The buffers are operated based on a timer. When the buffer is full, the packets are transferred to the GPU. If the buffer is still not full after a prescribed time threshold (100ms in the present implementation), the contents of the buffer are transferred anyway, such that there is minimal latency introduced by buffering.

4.2 Transferring the DFAs and Packets to the GPU

Snort uses rule contents and uricontents of all the rules in a rule group to construct one DFA, which is implemented using a hash table. In this implementation the DFA is represented in the form of a table or a two dimensional array. This table has 256 columns, each of which represents the corresponding ASCII character (0-255); the number of rows is equal to the number of states in that DFA [14]. Each cell in this table is a data structure containing two integers. The first integer represents the next state for that particular row (row represents the current state) and column (which represents the current symbol), which corresponds to the *goto* function of the AC algorithm [1]. The second integer denotes whether that is a final state or not. If it is a final state this integer will have a value 1, and 0 otherwise.

Table 1:Table Representing a DFA for the String 'black'

	0	1	2--	97	98	99	--	107	108	--	255
State 0					1, 0						
State 1									2, 0		
State 2				3, 0							
State 3						4, 0					
State 4								4, 1			

Table 1 shows a simple example of how the DFA table for the string 'black' would look like. The ASCII values of characters 'b', 'l', 'a', 'c' and 'k' are 98, 108, 97, 99, and 107 respectively. State '0' is the starting state. At State 0, it goes to State '1' only when it encounters the character 'b' represented by ASCII value 98. For all other characters it remains on State '0'. At State '1', it goes to State '2' on receiving the character 'l', which

has an ASCII value of 108 and so on. On receiving any character other than 'l' in State '1', it goes back to the starting state or State '0'. In this DFA State '4' is the final state. Hence at State '4' there is no transition and the second integer has the value '1' indicating that it is the final state.

In this implementation, the rule contents and uricontents of Snort are used to construct the DFAs in the tabular format. These tables are then rearranged to form a single one-dimensional array of cells, which are copied to the GPU global memory. An additional array of offsets is constructed so as to retrieve the correct table for comparison when a set of packets is received.

The packets are transferred either when the buffer is full or if the timer has timed out. In either case, the DFA table that represents that group is identified and the packets along with the table offset are transferred to the GPU.

4.3 Perform Pattern Matching and Obtain the Output

The Aho-Corasick multi-pattern search algorithm was ported to work with the GPU parallel architecture. The GPU implementation of the algorithm is slightly different from the original AC algorithm.

```
Input: DFA Table, Set of packets {P1, P2, ..., Pn}, data structures for
storing the output
Output: Locations where the patterns occur in each packet

begin
Declare n threads; one for each packet
currentState ← 0
patternLength ← 0
numPatterns ← 0
for cursor ← beginning of packet to end of packet
  if DFATable[current state][packet[cursor]].nextState ≠ 0 then
    if DFATable[current state][packet[cursor]].isFinal = 0 then
      currentState = DFATable[current state][packet[cursor]].nextState
      patternLength = patternLength+1
```

```
    else
      matchPostion = cursor - patternLength
      matchState = currentState
      numPatterns = numPatterns +1
  else
    patternLength ← 0
    currentState ← 0
end
```

A data structure is created to record all the match instances for each packet. The position at which the pattern was found, the DFA state at which the pattern was found, and the total number of instances of pattern matches found in the packet can be recorded in this data structure. An array of such structures, one for each packet, is copied to the GPU global memory along with the packets. After the string comparison, any match found in a packet is recorded into the corresponding data structure.

After pattern matching, the data structures containing the results are copied back to the CPU RAM. This output can directly be logged or can be used to raise an alert in case of a match.

4.4 Results

In this section, the actual results obtained from the comparison of CPU and GPU implementations are presented. The CPU used for the experiments was a 2.8 GHz AMD Phenom II X4 965 processor with 4 cores, 16 GB total memory and 512 KB cache. The GPU used for the implementation was a Tesla C2050 device with 14 multiprocessors and 32 cores per multiprocessor. It has a GPU clock speed of 1.15 GHz and 2.68 GB global memory.

The performance of Network Intrusion Detection using GPU was measured using various benchmarks. Initial analysis was made on sample *pcap* files obtained from the websites [37, 38]. Later, a Honey Pot was set up so as to attract actual intrusion packets into the system, and these packets were analyzed by the new application.

"Honey Pots are any security resource whose values lies in being probed, attacked, or compromised. They can be real operating systems or virtual environments mimicking production systems" [17]. They create fake working environments so as to attract intruders such that the signatures left by them can be studied and analyzed.

Figure 7 shows the variation in total run time for CPU and GPU for a fixed number of packets. It is observed that GPU is twice as fast as CPU on average. It is independent of the buffer size for small numbers of packets.

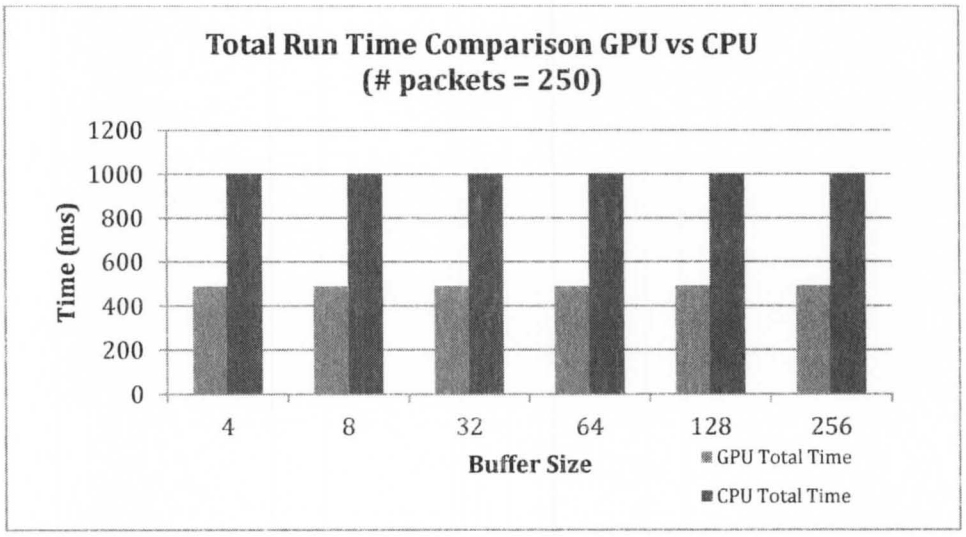

Figure 7: Total Run Time Comparison

However, as shown in Figures 8 and 9, when the total time taken for the search process alone is compared, it is found that for small fixed numbers of packets, the CPU

outperforms the GPU by a factor of two. This variation is due to the buffering scheme in the new implementation. For fewer numbers of packets, the buffering scheme introduces a delay while waiting for 100ms for the buffer to be full, in case of large buffer size; or frequent GPU memory accesses in case of smaller buffer sizes.

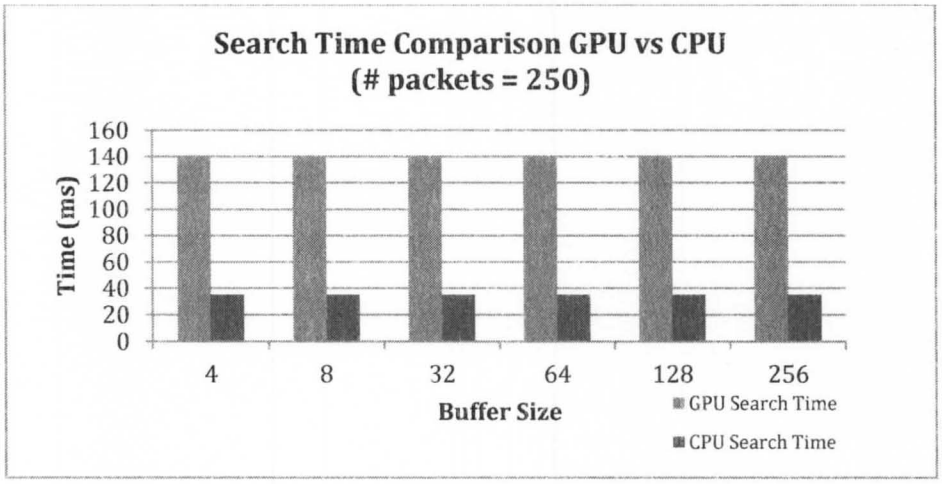

Figure 8: Search Time Comparison

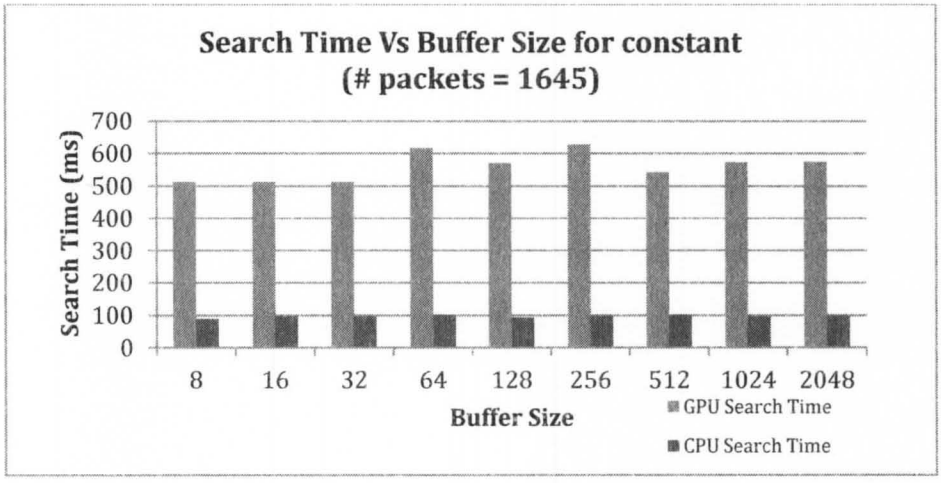

Figure 9: Variation of Search Time vs Buffer Size

It can be observed that for hundreds of thousands of packets the performance of GPU is at least twice as fast as the CPU in the case of total search time, as can be seen

from Figure 10. A maximum performance of four times the speed was observed as can be seen from the graph. The values are recorded for different number of packets for the time being.

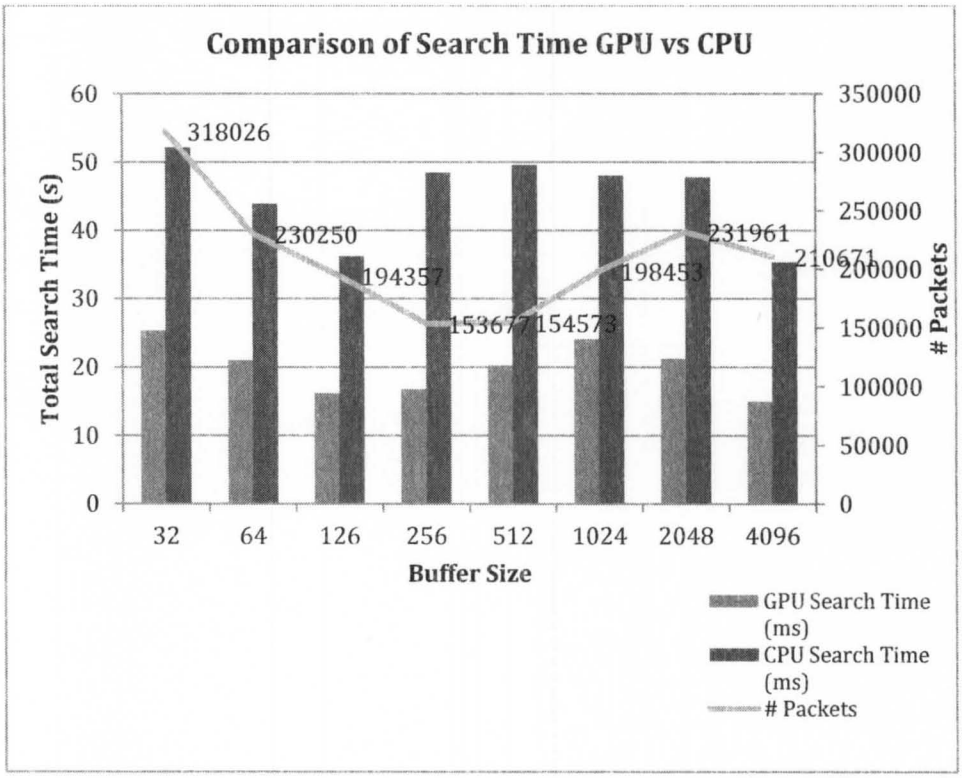

Figure 10: Comparison of Search Time

The speed of the GPU augmented Snort is clearly increased by offloading the pattern-matching algorithm to the GPU. The performance improvement shows a steady rise as the number of packets received per second increases. It can also be observed that the GPU search time shows a very gradual rise as the number of packets increases. Therefore, it can be concluded that for real attacks like Denial Of Service attack, when a large number of packets need to be analyzed, GPUs exhibit a consistent performance while the CPUs tend to get slower.

V. CONCLUSION AND FUTURE WORK

The 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.

Gnort [2] was a prototype implementation of Snort that claimed to have a performance of twice the speed of Snort. This thesis presented the implementation of an actual application that runs like Snort but with twice to four-fold the speed.

There is a huge 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. As the number of GPU cards used increases, a proportional speed up of the application is expected. Presently, this implementation performs only the content matching, which can be extended to regular expression matching that will give a tremendous boost to the performance. Research can also be conducted to improve the performance of the application by coupling the use of serial CPU during low traffic hours and switching to GPU computation during high traffic hours.

The idea of parallelizing the pattern matching algorithm can be extended to parallelizing the packet preprocessing part. The preprocessing component of Snort that examines packets for suspicious activity or process packets to provide appropriate input to detection engines, can be ported to the GPU, for further improvement in speed. This process is expected to produce enormous speed as all the costly computations can be offloaded to the GPU.

The accuracy of detection of intrusion packets is not measured in the current implementation as it was built over Snort and Snort does post processing of the packets, which further filters them into intrusion and non – intrusion packets. This is one area which can be worked on to implement all post processing activities similar to Snort and compare the accuracy.

REFERENCES

[1] A. V. Aho, and M. J. Corasick (1975). Efficient string matching: an aid to bibliographic search. *Communications of the ACM, 18(6)*, pp. 333-340.

[2] Ando, K., Okada, M., Shishibori, M., Jun-Ichi Aoe (1997). Efficient multi-attribute pattern matching using the extended Aho-Corasick method. *Systems, Man, and Cybernetics, 1997. Computational Cybernetics and Simulation., 1997 IEEE International Conference, 4*, pp. 3936-3941.

[3] Anithakumari, S., Chithraprasad, D (2009). An Efficient Pattern Matching Algorithm for Intrusion Detection Systems . *Advance Computing Conference, 2009. IACC 2009. IEEE International*, (pp. 223-227).

[4] Antonino Tumeo, O. V., Donatella Sciuto (2010). Efficient pattern matching on GPUs for intrusion detection systems. *Proceedings of the 7th ACM international conference on Computing frontiers CF 10*, (p. 87).

[5] Charalampos S Kouzinopoulos, Konstantinos G Margaritis (2009). String Matching on a Multicore GPU Using CUDA. *13th Panhellenic Conference on Informatics* .

[6] Cheng, Y. W. (Feb. 2010). Fast Virus Signature Matching Based on the High Performance Computing of GPU. *Communication Software and Networks, 2010. ICCSN '10. Second International Conference*, (pp. 513-515).

[7] Cheng-Hung Lin, Chen-Hsiung Liu, Shih-Chieh Chang (Dec. 2011). Accelerating Regular Expression Matching Using Hierarchical Parallel Machines on GPU. *Global Telecommunications Conference (GLOBECOM 2011), 2011 IEEE* (pp. 1-5). IEEE.

[8] Cheng-hung Lin, Sheng-yu Tsai, Chen-hsiung Liu, Shih-chieh Chang, Jyuo-min Shyu (2010). Accelerating String Matching Using Multi-threaded Algorithm on GPU. *Communications Society* .

[9] Chengkun Wu, Jianping Yin, Zhiping Cai, En Zhu, Jieren Chen (2009). A Hybrid Parallel Signature Matching Model for Network Security Applications Using SIMD GPU. *APPT '09 Proceedings of the 8th International Symposium on Advanced Parallel Processing Technologies, 5737*, pp. 191-204.

[10] Chih-chiang Wu, Sung-hua Wen, Nen-fu Huang, Chia-nan Kao. (2005). A Pattern Matching Coprocessor for Deep and Large Signature Set in Network Security System. *GLOBECOM 05 IEEE Global Telecommunications Conference*, (pp. 1791-1795).

[11] D. E. Knuth, J. Morris, and V. Pratt (1977). Fast pattern matching in strings. *SIAM Journal on Computing , 6(2)*, 127-146.

[12] Daniel Luchaup, Randy Smith, Cristian Estan, Somesh Jha (2011). Speculative Parallel Pattern Matching. *IEEE Transactions on Information Forensics and Security*, (pp. 438-451).

[13] Fechner, B. (Feb. 2010). GPU-Based Parallel Signature Scanning and Hash Generation. *Architecture of Computing Systems (ARCS), 2010 23rd International Conference*, (pp. 1-6).

[14] G. Vasiliadis, S. Antonatos, M. Polychronakis, E. P. Markatos, and S. Ioannidis (2008). Gnort: High performance network intrusion detection using graphics processors. *In Proceedings of 11th International Symposium on Recent Advances in Intrusion Detection (RAID)*, (pp. 116-134).
[15] G. Vasiliadis, S. Antonatos, M. Polychronakis, E. P. Markatos, and S. Ioannidis (2009). Regular Expression Matching on Graphics Hardware for Intrusion Detection. *12th International Synopsium on Recent Advances in Intrusion Detection (RAID)*.
[16] J. B. D. Cabrera, J. G., W. Lee, R. K. Mehra (Dec. 2004). On the Statistical Distribution of Processing Times in Network Intrusion Detection. *43rd IEEE Conference on Decision and Control*, (pp. 75-80).
[17] Jammi Ashok, Y. Raju, S.Munisankaraiah (2010). Intrsion Detection Through Honey Pots. *InternationalJournal of Engineering Science and Technology , 2(10)*, 5689-5696.
[18] Jiangfeng Peng, Hu Chen, Shaohuai Shi (May 2010). CUgrep: A GPU-based high performance multi-string matching system. *Future Computer and Communication (ICFCC), 2010 2nd International Conference*.
[19] Jiangfeng Peng, Hu Chen, Shaohuai Shi (2010). The GPU-based string matching system in advanced AC algorithm. *IEEE International Conference on Computer and Information Technology*, (pp. 1158-1163).
[20] Lee, T.-H. (2007). Generalized Aho-Corasick Algorithm for Signature Based Anti-Virus Applications . *Computer Communications and Networks, 2007. ICCCN 2007. Proceedings of 16th International Conference*, (pp. 792-797).
[21] Lei Wang, Shuhui Chen, Yong Tang, Jinshu Su (2011). Gregex: GPU Based High Speed Regular Expression Matching Engine. *Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS), 2011 Fifth International Conference* , (pp. 366-370).
[22] S. Wu and U. Manber (1994). A fast algorithm for multi-pattern searching.
[23] R. S. Boyer and J. S. Moore (1977). A fast string searching algorithm. *Communications of the Association for Computing Machinery, 20(10)*, pp. 762-772.
[24] Nen-Fu Huang, H.-W. H., Sheng-Hung Lai, Yen-Ming Chu, Wen-Yen Tsai (2008). A GPU-Based Multiple-Pattern Matching Algorithm for Network Intrusion Detection Systems. *Advanced Information Networking and Applications - Workshops, 2008. AINAW 2008. 22nd International Conference*, (pp. 62-67).
[25] Nigel Jacob, Carla Brodley (2006). Offloading IDS Computation to the GPU. *2006 22nd Annual Computer Security Applications Conference ACSAC06*, (pp. 371-380).
[26] *NVIDIA CUDA manual reference.* (n.d.). Retrieved from http://developer.nvidia.com/object/gpucomputing.html
[27] Randy Smith, Neelam Goyal, Justin Ormont, Karthikeyan Sankaralingam, Cristian Estan (2009). Evaluating GPUs for network packet signature matching. *2009 IEEE International Symposium on Performance Analysis of Systems and Software*, (pp. 175-184).
[28] Rehman, R. U. (2003). Intrusion Detection Systems with Snort: Advanced IDS Techniques with Snort, Apache, MySQL, PHP, and ACID. Prentice Hall PTR.
[29] Sunho Lee, Dong Kyue Kim (2009). Efficient multiple pattern matching algorithms for Network Intrusion Detection Systems. *2009 IEEE International Conference on Network Infrastructure and Digital Content*, (pp. 609-613).

[30] Tuck, N., Sherwood, T., Calder, B., Varghese, G (2004). Deterministic memory-efficient string matching algorithms for intrusion detection. *INFOCOM 2004. Twenty-third AnnualJoint Conference of the IEEE Computer and Communications Societies, 4*, pp. 2628-2639.
[31] Vasiliadis, G., Polychronakis, M., Ioannidis, S (Nov. 2011). Parallelization and characterization of pattern matching using GPUs. *Workload Workload Characterization (IISWC), 2011 IEEE International Symposium*, (pp. 216-255).
[32] Vespa, L. J., Ning Weng (Oct. 2011). GPEP: Graphics Processing Enhanced Pattern-Matching for High-Performance Deep Packet Inspection. *Internet of Things (iThings/CPSCom), 2011 International Conference on and 4th International Conference on Cyber, Physical and Social Computing*, (pp. 74-81).
[33] Xinyan Zha, Sahni, S. (March 2012). GPU-to-GPU and Host-to-Host Multipattern String Matching on a GPU. *Computers, IEEE Transactions.*
[34] Xinyan Zha, Sahni, S. (2011). Multipattern string matching on a GPU . *Computers and Communications (ISCC), 2011 IEEE Symposium*, (pp. 277-282).
[35] Zha, X., & Sahni, S. (2008). Highly compressed Aho-Corasick automata for efficient intrusion detection. *Computers and Communications, 2008. ISCC 2008. IEEE Symposium*, (pp. 298-303).
[36] http://www.snort.org/
[37] http://www.ll.mit.edu/mission/communications/ist/corpora/ideval/data/index.html
[38] http://wiki.wireshark.org/SampleCapture
[39] Jack Koziol, Intrusion Detection with Snort , ISBN-10: 157870281X ISBN-13: 9781578702817 Publisher: Sams Publishing Copyright:2003 Format: Paper; 360 pp Published: 05/20/2003

CURRICULUM VITAE

Anju Panicker Madhusoodhanan Sathik
2241 Arthur Ford Ct. Apt #4,
Louisville, KY 40217.
E-mail:- anjupanicker.ms@gmail.com

Personal Profile:-

- Received the **Highest Cumulative Scholastic Standing** award from University of Louisville for the Computer Science Master's program.
- Worked as Teaching Assistant for Information Security, Performance Evaluation and Algorithms courses. Performed duties like grading tests and projects and has given lectures.
- Performed research in performance improvement of Boolean Satisfiability problem (SAT) using GPU intensified **Parallel Programming** with the use of **CUDA** architecture.
- Current research is on **Network Intrusion Detection System** using parallel programming with CUDA-enabled gpu.
- Hands on experience in **Software Development** and **Information Technology.**
- .Net: Experience in developing Windows Mobile application for mobility enterprise solutions.
- Experience in Embedded C programming for POS applications.
- Involved in analysis, coding, testing, client communication and knowledge transfer activities.
- Exceptional ability to motivate others and help provide a highly productive development environment.

Experience Summary:-

- Worked as **Teaching Assistant** at University of Louisville for 12 months. (2011)
- Worked at **REACH** (Resources for Academic Achievements), an undergraduate association at University of Louisville, as a student tutor for 4 months.
- **2.8** years of experience in Software Development at **Infosys Technologies Ltd.** as a **Software Engineer**

Educational Qualification:-

- Pursuing **Master of Science** degree at University of Louisville with a current GPA of **3.93/4**. (2010-2012)
- **Bachelor of Technology** in **Electronics and Communication** from University of Kerala with an aggregate of **78.8%.** (2002-2006)
- ISC with subject aggregate of 90%. (2000-2002)
- ICSE with an aggregate of 90%. (2000)

Publications:-

- Ahmed H. Desoky, Anju P. Madhusoodhanan. *Bitwise Hill Cipher Crypto System*. IEEE International Symposium on Signal Processing and Information Technology (ISSPIT), December 14-17, 2011 - Bilbao – Spain. http://www.isspit.org/isspit/2011/prog.pdf

Abstract:

This paper describes a modification to the conventional Hill Cipher system. The purpose of this paper is to explore the adeptness of Hill Cipher for binary data. The plaintext is any binary data such as images, audio, video, etc. The plaintext is subjected to scrambling by dividing it into 8 planes. Each of these planes is encrypted using a different key. From the study conducted on Bitwise Hill Cipher, it is found that it has enough security required for commercial applications.

Technical Expertise:-

Programming Languages	:	CUDA, Matlab, C, C++, Visual Basic, VB.Net, C#.Net
Operating Systems	:	Windows NT, Windows 2000/XP, Linux
Database	:	Microsoft SQL Server 2000.

PROJECTS:-

1. BootsPDT Replacement:

Client : Boots (UK)
Role : Developer.
Duration : Sept 2008 to July 2009.

Project Introduction:

The Boots PDT replacement project dealt with developing a windows mobile application for the MC70 scanner for the store's inventory stock management. Boots store initially used a Symbol device which was to be replaced by MC70 in order to improve performance.
The core functionalities delivered by the project are:

- Active file download to the MC70 scanner from an EPOS controller during the Start of Day process and Export data upload to the EPOS controller at End of Day.

- Maintaining a local database which would help the application to maintain the Total Stock Figure for the store.

- Applications like Shelf Management, Goods In and Goods Out which are used for the day to day maintenance of the goods in and out of the store.

Responsibilities:

1. Involved in the project from Requirement Analysis phase.

2. Design of the Shelf Management features.
3. Development of the Shelf Management application which includes the features like:
    - Shelf Monitor: To manage the count of products inside the store
    - Excess Stock: To manage the goods in back shop.
    - Item Info: Allows viewing the details of an item which is scanned using the scanner.
    - Price Check: To determine an increase/decrease in price of any item.
    - Space Planning: Obtain the Plano gram details of the scanned item.

4. Done complete testing for the project including the regression cycles.

2. Nordstrom POS application:

Client	: Nordstrom (US)
Role	: Developer
Duration	: Mar 2006 to Sept 2007.

Project Introduction:

Enhancements on the Nordstrom Point Of Sale application , GlobalSTORE developed by Fujitsu which include multi-vendor infrastructure management services and point-of-sale hardware and software.

The key enhancements were:

- MCR1 or Multi Channel Retailing which introduced online transactions in Global STORE.

- Developing server side services for Register Alerting which would update the POS application with updates from a central controller.

Responsibilities:

1. Analysis & Design, Coding.
2. Ensure implementation of application, verification & validation activities to achieve the quality of deliverables.
3. Review the design, code, unit test plan, test cases & test results.

3. Mobile POS:

Client : Infosys Technologies Ltd.
Role : Developer
Duration : Mar 2006 to July 2009.

Project Introduction:

The mobile POS application will allow the transaction in a store to be performed using the mobile phone. The image of the barcode of a particular product will be captured using the mobile phone camera. This will be decoded and the product details will be fetched from a dedicated server using Wi-Fi connectivity. The transaction is completed by making a credit card payment. The credit card details entered in the mobile phone will be validated by a credit server.

Responsibilities:

1. Analysis & Design, Coding of the Customer Interest Tracker module which would fetch the sales record from the database within a particular period for a selected number of items and would provide a graphical display of the same.
2. Review the design, code, unit test plan, test cases & test results

Personal Profile

Date of Birth	:	24-08-1984
Sex	:	Female
Marital Status	:	Married
Languages Known	:	English, Malayalam and Hindi
Nationality	:	Indian
Permanent Address	:	#108, Chayakkudi Lane Pettah P.O. Trivandrum – 695024. Kerala, India.

.

Declaration:-

I hereby declare that the information presented above is correct and complete to the best of my knowledge and belief.

Anju Panicker M.S.