Addis Ababa University
Addis Ababa Institute of Technology
School of Electrical and Computer Engineering

Accelaration of Preprocessors of the Snort Network Intrusion Detection System Using General Purpose Graphics Processing Unit

A thesis submitted to the School of Electrical and Computer Engineering in partial fulfillment of the requirements for the Degree of Master of Science in Computer Engineering

By Simegnew Yihunie Alaba

April, 2015

Addis Ababa University
Addis Ababa Institute of Technology
School of Electrical and Computer Engineering

Acceleration of preprocessors of the Snort Network Intrusion Detection System Using General Purpose Graphics Processing Unit

By
Simegnew Yihunie Alaba

Advisor: Fitsum Assamnew

Addis Ababa University
Addis Ababa Institute of Technology
School of Electrical and Computer Engineering

Acceleration of preprocessors of the Snort Network Intrusion Detection System Using General Purpose Graphics Processing Unit

By

Simegnew Yihunie Alaba

Approval By Board of Examiners

Dr. Yalemzewd Negash	_____
Dean, School of Electrical and Computer Engineering	signature
Fitsum Assamnew	_____
Advisor	signature
_____	_____
Internal Examiner	signature
_____	_____
External Examiner	signature

Abstract

Advances in networking technologies enable interactions and communications at high speeds and large data volumes. But, securing data and the infrastructure has become a big issue. Intrusion Detection Systems such as Snort play an important role to secure the network. Intrusion detection systems are used to monitor networks for unauthorized access. Snort has a packet decoder, pre-processor, detection engine and an alerting system. The detection engine is the most compute intensive part followed by the pre-processor. Previous work has shown how general purpose graphics processing units(GP-GPU) can be used to accellerate the detection engine. This work focused on the pre-processors of Snort, specifically, the stream5 pre-processor as profiling revealed it to be the most time consuming of the pre-processors. The analysis shows that the individual implementation of stream5 using Compute Unified Device Architecture(CUDA) achieved up to five times speed up over the baseline. Also, an over all 15.5 percent speed up on the Defense Advanced Research Projects Agency(DARPA) intrusion detection system dataset was observed when integrated in Snort.

Key words: Intrusion Detection System, Snort, Graphics Processing Unit, CUDA, Parallelization, Porting, Preprocessor.

Declaration

I,the undersigned, certify that research work titled "Acceleration of Preprocessors of Snort Network Intrusion Detection System using General Purpose Graphics Processing Unit " is my own work. The work has not been presented elsewhere for assessment. Where material has been used from other sources, it has been properly acknowledged.

Simegnew Yihunie Alaba signature ——————

Date of submission: April, 2015

place: Addis Ababa

This thesis has been submitted for examination with my approval as a university advisor.

Advisor: Fitsum Assamnew signature ——————————

Abbreviations

CPU	Central processing Unit
CUDA	Compute Unified Device Architecture
DARPA	Defense Advanced Research Projects Agency
DRAM	Dynamic Random Access Memory
FTP	File Transfer Protocol
GPU	Graphics processing Unit
GPGPU	General-Purpose Graphics processing Unit
HIDS	Host Intrusion Detection System
HTTP	Hypertext Transfer Protocol
ICMP	Internet Control Message Protocol
IDS	Intrusion Detection System
IIS	Internet Information Service
IP	Internet Protocol
IPS	Intrusion Prevention System
KB	Kilo Bytes
MB	Mega Bytes
Mbps	mega bits per second
MTU	Maximum Transfer Unit
NFS	Network File Sharing
NIDS	Network Intrusion Detection System
PCAP	Packet Capture

PCIe	Peripheral Component Interconnect Expresses
PTX	Parallel Thread Execution
RAM	Random Access Memory
RPC	Remote Procedure Call
SIMD	Single Instruction Multiple Data
SIMT	Single Instruction Multiple Thread
SMTP	Simple Mail Transfer Protocol
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
URL	Uniform Resource Locator

Acknowledgment

First and foremost I would like to thank the Almighty God for giving me the strength through long and challenging process of the thesis.

I would like to give special thanks to my Advisor Mr. Fitsum Assamnew for his help, the overall support, guidance,suggestions and encouragement throughout the thesis work.

Furthermore, I would like to thank my families, friends and colleagues who were around me by giving ideas,encouraging and commenting on the work throughout the thesis.

Lastly, I would like to thank Nvidia Developer team members specially Mr. Kevin for giving advice and support of new ideas and concepts.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Nowadays, there is a rapid development of network technologies and associated bandwidth. Though these developments enhance data communication, it also facilitates malicious activities against resources on the network. These malicious threats pose challenges to modern network security systems. Many methods have been developed to secure the network infrastructure and communication over the network. Examples of widely adopted security measures on the network are the use of firewalls and data encryption. In addition to these security measures Intrusions Detection and Prevention System (IDPS) [1] is a relatively new technique.

An Intrusion Detection System (IDS) is an application that monitors the network for any unauthorized accesses into it. The application monitors the network for violation of access permissions or other malicious activities. On the other hand, an Intrusion Prevention System blocks or prevents an intrusion. IDSs can be implemented in both hardware and software. Though hardware implementations are generally faster, they suffer from a couple of shortcomings that limit their usability. First, they are more expensive to implement and maintain. Second, since hardware modification is difficult, they are less flexible for improvement. Software implementations, on the other hand, can easily be modified with new

algorithms [2].

There are two types of IDSs: Network based and Host based. Network intrusion detection systems(NIDS) are intrusion detection systems that capture data packets traveling on the network media (cables, wireless) and match them to a database of signatures. Depending upon whether a packet is matched with an intruder signature, an alert is generated or the packet is logged to a file or database. Some examples of NIDS are Snort, Suricata and Bro. Among these, Snort is the most widely deployed Network Intrusion Detection Systems worldwide [3]. On the other hand, host-based intrusion detection systems(HIDS) are installed as agents on a host. These IDSs can look into system and application log files to detect any intruder activity. Some of these systems are reactive, meaning that they inform you only when something has happened. Some of the others are proactive; they can sniff the network traffic coming to a particular host on which the HIDS is installed and alert you in real time. Antivirus is one exapmle of HIDS.

Snort uses a set of clear text rules to instruct a detection engine on proper reactions to particular network traffic patterns. It is logically divided into multiple components: packet decoder, preprocessor, detection engine, logging and alerting system and output module [4]. These five major components are described as follows:

- **Packet Decoder**: Important for taking packets from different types of network interfaces and prepares the packets to be preprocessed.
- **Preprocessors**: Arrange or modify data packets before the detection engine does some operation. Some preprocessors also perform detection by finding anomalies in packet headers and generating alerts.
- **Detection Engine**: Detects if any intrusion activity exists in a packet. The detection engine employs Snort rules for this purpose.

- **Logging and Alerting Systems**: Log the activity or generate an alert. Logs are kept in simple text files, tcpdump-style files or some other form.
- **Output Module**: Controls the type of output generated by the logging and alerting systems.

The detection engine does pattern matching which takes large amount of processing time as compared to the other parts. The second time consuming part of Snort is the preprocessor. The Preprocessor is used to prepare data packets to be analyzed against rules in the detection engine [4]. The other three modules require less processing time as compared to the detection engine and the preprocessors [5]. Recently, with the coming of general purpose computing on Graphics processing units (GPUs), research to improve the speed of IDS is receiving attention.

1.1 Problem Statement

Network traffic has been increasing more rapidly as the available bandwidth increased. This makes it difficult the inspection of all network traffic for intrusion detection. These days, many researches have been carried out in the area of intrusion detection to increase the security of a network system. Some implementation of NIDS uses GPUs to accelerate the intrusion detection [1 and 5]. Though accelerating detection engine of snort increases the capability of detecting intrusions and decrease the number of packets dropped, it is not enough to protect the system effectively. This is because the preprocessor feeds the detection engine which effectively limits the speed of the detection engine as fast as the feed from the preprocessor. Therefore, accelerating it further increases the speed of detection.

1.2 Objectives

General Objective

The general objective of this thesis is to increase the performance of Snort IDS by identifying and porting compute intensive part of preprocessors.

Specific Objectives

- Understand the preprocessors of Snort.
- Identify compute intensive parts of the preprocessors.
- Port the identified preprocessor to GPUs using Compute Unified Device Architecture
- Optimize the GPU implementation of the preprocessor.
- Evaluate the performance of the GPU implementation.

1.3 Scope of the Thesis

Preprocessors are the possible bottle-neck for detection engine compared to other components. Since detection engine is most intensive, we tried to enhance the detection by solving the bottle-neck. Therefore, in this thesis we focused on the preprocessors of Snort.

1.4 Outline of the Thesis

The rest of the paper is arranged as follows. Chapter 2 presents related work which describes the combination of Snort NIDS and GPU. Different research papers in the area are summarized with the corresponding technologies and methodologies

used. Chapter 3 describes the theoretical background to the Graphics Processing Units and CUDA technology. Chapter 4 gives highlight on Snort NIDS architectures and preprocessors of Snort. Chapter 5 emphasis the methodology used for porting the stream5 preprocessor part of Snort using CUDA. Chapter 6 summarizes and discusses the results obtained during experiment. Finally, Chapter 7 concludes our work and lays out recommendations for future work.

Chapter 2

Literature Review

It was found out from the litrature review on the performance improvement of Snort IDS, the detection engine is the main focus for improvement using GP_GPUs. These works can be catagorized into two approaches; the first one that port the detection engine with out much modification and the second one that modify the detection engine when offloading it to the GPU. Section 2.1 presents the former approach while Section 2.2 describes the later.

2.1 Porting Detection Engine of Snort to GPU

The work in [5] focused on to accelerate the IDS by offloading the detection engine of Snort to run on GPUs using CUDA. Network traffic has been increasing more rapidly than the clock-speed of CPUs. The CPUs clock-speed are unable to cope with the bandwidth in high-tech network infrastructure . The massive flows of data packets overload the NIDS and lead to packet loss which makes them pass by unchecked for malware and intrusion attempts[5]. The main cause of this is the network packet inspection module in the detection engine of the NIDS. The detection engine consists of numerous functions and ultimately contains an

algorithm for string searching. The study experimented with implementations of known algorithms for string search; naïve string search, Knuth-Morris-Pratt and Aho-Corasick[6] on Nvidia GPUs using CUDA. Aho-corasick is multipattern where as the the other two algorithms are single pattern match algorithms. A static string of text is used as input where a known number of signatures can be placed to be used for detection. The results show that their GPU accelerated multi-pattern algorithm performs over 70 times more patterns than any of the single pattern algorithms executed on both GPU and CPU, including the CPU version of the multi-pattern algorithm. The work presents the speed up gained on the detection engine with out explaining how the over all Snort IDS performs with this improved detection engine.

The work done in [1] is similar to [5] in porting the detection engine of Snort to GPU but used real network data as input.This work experimented with aho-corasick algorithm only. 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.

In [7] the authors attempted to gain high performance in Snort NIDS using GPU. They observed that single pattern matching algorithms like Knuth-Morris-Pratt were not suited for GPU computing, and rather tried using multi-pattern matching algorithms. The authors chose to port the algorithm Aho-Corasick and concluded that it boosted performance of Snort by a factor of two.

The paper in [8] presents signature-matching IDS can experience significant decreases in performance when the load on the IDS-host increases.They propose a solution that off-loads some of the computation performed by the IDS to the Graphics Processing Unit (GPU) with graphics code(Cg). Cg runs as an embedded GPU language in conjunction with another graphics API, either OpenGL or Di-

rectX (Microsoft). The major operation in a signature-matching IDS is matching values seen operation to known black-listed values, as such, their solution implements the string-matching on the GPU. The results show that as the CPU load on the IDS host system increases, PixelSnort's performance is significantly more robust and is able to outperform conventional Snort by up to 40 percent. This work was implemented using graphics API before the programming models were developed.

2.2 Modified Detection Engine on the GPU

The work done in [9] proposes an efficient pattern matching algorithm to detect intrusions using GPUs. They proposed an efficient algorithm which is based on hierarchical hash table. In the hierarchical hash table architecture, the hash table in each level is stored as a one-dimensional array. The number of level is equal to the length of the pattern. Each cell of the hash table is the key to the entry of hash table in next level. In case the hash table is the final level, the corresponding cell will contain the unique identification number of the matching pattern, otherwise zero. A drawback of this structure is that hash tables are sparsely populated when the number of patterns is small or patterns can be classified in few groups by prefix or suffix of patterns. This algorithm was ported on GPU and achieved a maximum traffic processing throughput of 2.4 G bit/s.

In [10] the authors presented a pattern matching library for GPU. They presented the design, implementation, and evaluation of a pattern matching library running on the GPU. The library supports both string searching and regular expression matching on the NVIDIA CUDA archi- tecture. The work discussed the performance impact of different types of memory hierarchies(texture , global and constant memories) in the GPU and presented possible solutions for memory congestion problems. The results of their performance evaluation using graphics

processors demonstrate that GPU based pattern matching can reach ten gigabits per second throughput.

As can be seen, there are many efforts that have been done in the offloading of detection engine of Snort to run on GPU. On the other hand, other components of Snort have not been given proper attention for acceleration even though they have impact in speed of detection. Preprocessors feed input to the detection engine and are the second compute intensive part of Snort. So, accelerating the preprocessor part of Snort also promises increasing the speed of detection.

2.3 Summary

In this Chapter the related work was presented in two categories based on the improvement made on the detection engine of Snort. The first category directly ported the detection engine to the GPU while the other category implemented a modified detection engine on the GPU. It was observed that Aho-corasick algorithm was presented as the suitable algorithm for pattern matching for the detection engine in Snort.

Chapter 3

General Purpose Computations on Graphics Processing Unit with CUDA

This section introduces general purpose computing on graphics processing units. Specifically, suitable applications on the GPU and programming tool CUDA and its architecture will be described.

3.1 Graphics Processing Unit

GPUs were originally designed for graphics processing. This includes things such as rendering of 3D models and simulations of motion for these models. The first company to develop such GPU was NVidia, Inc. in 1999. This first GPU (GeForce 256), can process 10 million polygons per second and has over 22 million transistors [11]. Through time these GPUs evolved to accomodate graphics computations that required flexibility. Many researchers investigated the application GPUs for general purpose computing(other than graphics functions). Hence, the general-

purpose GPU was born and programming tools such as CUDA and OpenCL were developed.

The GPU is specialized for compute-intensive, highly parallel computation; therefore, is designed in way more transistors are devoted to data processing rather than data caching and flow control as the more general CPU. Figure 3.1 shows the allocation of transistors in the CPU and GPU.

Figure 3.1: The allocation of transistors on GPU and CPU[12]

The GPU architecture focuses on the data processing unit much more than the control flow and caching unit as shown above. As a result, the applications suitable to be run on GPU should be data intensive, no complex control flows, and few relying on large caches. The data then can be mapped to each Arithimetic and Logic Unit(ALU) to be processed in single instruction multiple data (SIMD)fashion [13]. In SIMD computation, a single operation is execute on different input data.

ALUs in side a streaming multiprocessor of the GPU share the same instruction unit. Hence, if there are two or more different control flows to be processed, each control flow will be scheduled to execute in sequential manner. This will reduce the system utilization. As a result, applications with complex control flows are not suitable for GP-GPUs.

The GPU's caches are typically smaller than CPU's caches. Hence, the application that depends heavily on large caches to perform well is not appropriate for GPU.

3.2 Compute Unified Device Architecture

In November 2006, NVIDIA introduced CUDA TM , a general purpose programming model that leverages the parallel compute engine in NVIDIA GPUs to solve many complex computational problems in a more efficient way than on a CPU. CUDA comes with a software environment that allows developers to use C as a high-level programming language. Other languages, such as FORTRAN, Java etc are also supported.

3.2.1 CUDA architecture

The CUDA architecture consists of several multiprocessors as shown in Figure 3.2. The numbers of multiprocessors are different between GPU models. Each multiprocessor has 8 stream processors with a common instruction unit[13]. The shared instruction unit makes all stream processors within a multiprocessor work in the Single Instruction Multiple Data (SIMD) manner. In other words, executing many stream processors simultaneously within a particular multiprocessor requires the instruction to be the same.

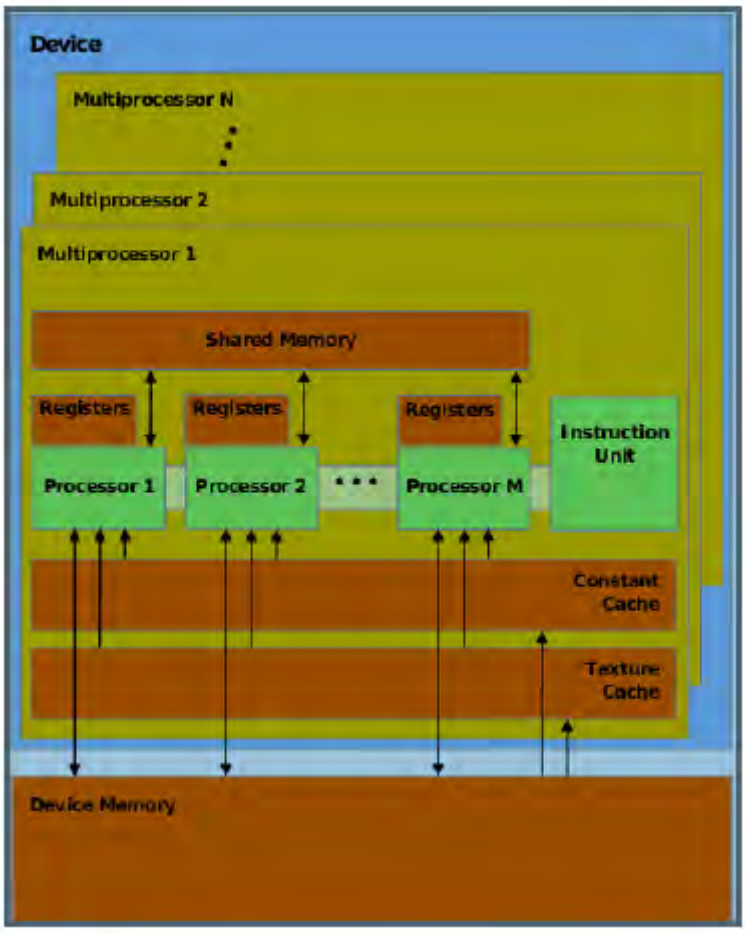

Figure 3.2: GPU CUDA Architecture [12]

Programming applications in CUDA architecture can be thought as a Single Instruction Multiple Thread (SIMT) programming [12]. In SIMT, the programmer defines the same code to be executed by multiple threads. As shown in Figure 3.3, the multiple threads are organized in a thread block and the thread blocks are organized in a grid. One thread executes concurrently and independently with other threads. Synchronization can be done only for the threads within the same block. Each thread has a thread ID namely *threadIdx* to identify the position within the block and block ID namely *blockIdx* to identify the position within the grid. Both the threads and blocks can be specified in 1, 2, or 3 dimensions. The decision on the dimensions depends totally on the applications to be implemented on the GPU.

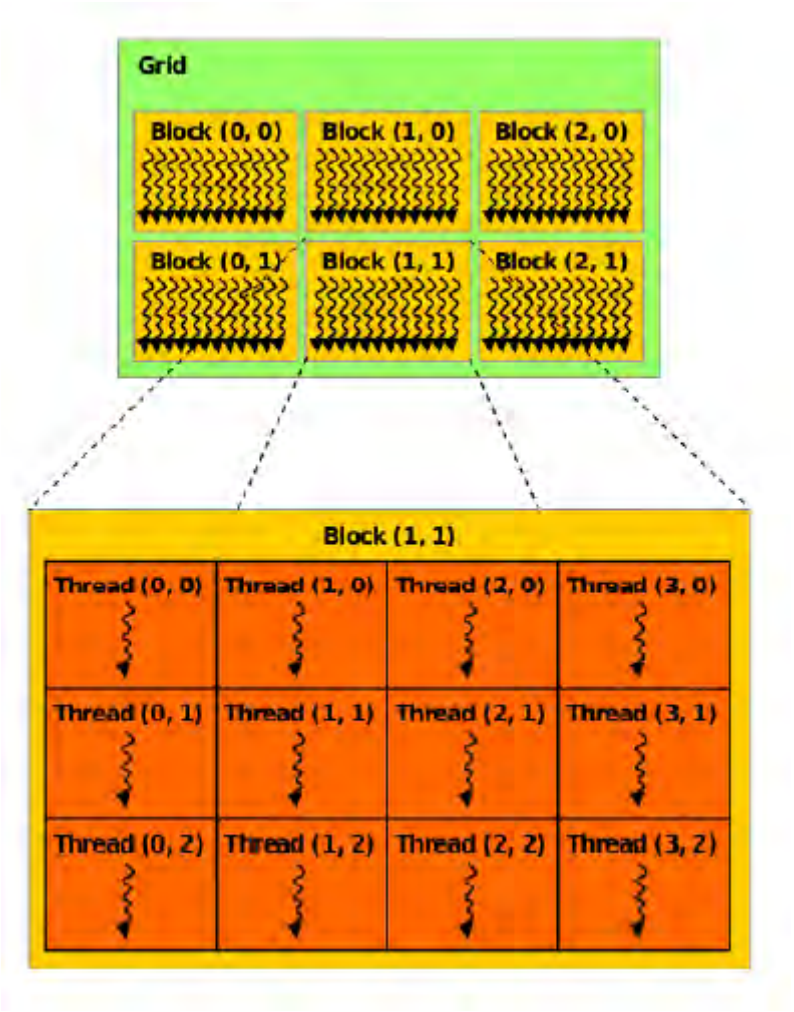

Figure 3.3: Grid of blocks in SIMT[12]

In CUDA, the executed code sent to the GPU is called a kernel. An example is provided below where we add two vectors A and B and store the result in C. The code looks like C code. However, there are identifiers that separate host-code from device-code, which is represented with __global__. The second important part is how the function VecAdd is called from main (). The difference from a normal C function call is "$\langle\langle\langle B, T\rangle\rangle\rangle$"which is an execution configuration syntax that tells the GPU to launch B-number of blocks, containing T-number of threads. The thread blocks scheduled to be executed in a multiprocessor are called active thread blocks . There is a limitation to the number of active thread blocks per multiprocessor.

This number depends on the GPU model and resources needed by the blocks. The

final number can be obtained by using CUDA profiler tools including CUDA visual profiler [14] and CUDA occupancy calculator [15]. Each multiprocessor executes a thread block by dividing the threads in the block into a smaller execution unit called warp. The warp always consists of warp size threads. The warp size for the current CUDA architecture is 32. As an illustration, each warp takes consecutive 32 threads by the thread IDs from the block. Then, when a multiprocessor is free, the SIMT scheduler schedules the next warp to be executed in the multiprocessor.

// Kernel definition

$\_\_global\_\_ \quad void \quad VecAdd\,(floatA, floatB, floatC)$

{

int i = threadIdx .x + blockIdx.x *blockDim.x;

C[i] = A[i] + B[i];

}

int main ()

{ ...

// Kernel invocation of B blocks with T threads.

VecAdd $\langle\langle\langle B, T\rangle\rangle\rangle$ (A, B, C);

}

Listing 3.1 Example of launching a kernel in CUDA with C code

3.2.2 Memory Hierarchy

A basic assumption within the CUDA programming model is that the threads are executed on a separate device from the host application running on the CPU. These devices hold their own separate memory spaces, accessible for the programmer.

CUDA GPUs contain two main types of memory storage, on-chip and off-chip. Registers and Shared memories are on-chip memories which is physically reside in GPU. On the other hand, global, constant, texture and local memories are off-chip

memories. The size of on-chip memories are limited. The stream processor has an option to keep data in a very larger memory namely device memory. Nonetheless, the device memory is off-chip and shared among all multiprocessors [12]. As a result, its latency is significantly higher than the on-chip memories.

Although the shared memory is fast, it has one major drawback over other memories. The drawback is that its life time is limited to a CUDA thread block. After the block finishes execution, the data in the memory is undefined [12]. Because of the limited life time, every new block has to load the data to the memory before execution. This loading is wasteful when the data is expected to be the same for all blocks. On the contrary, the device, constant, and texture memory do not have this limited life time. And, the data can be loaded only once and used by all blocks afterward.

Function and Variable type qualifiers

The variable type qualifiers specify in which device memory a variable will be stored. The function type qualifiers specify the functions to be executed the device. CUDA supports automatic variable declaration without the use of qualifiers, and will generally be placed in the register. However, the compiler may choose to place it somewhere else as it feels fit. To force a location of a variable we therefore declare where we want it located. This is done by calling for example: *__constant__*, *__shared__* or *__global__* [16].

Device

The __device__ keyword indicates that the variable being declared is a CUDA device variable. This key word is also important for function type qualifiers. A device function executes on a CUDA device and can only be called from a kernel function or another device function. Device can be used together with any of the other memory target qualifiers to further declare where the variable should be stored. Using this qualifier alone, the variable will be stored in global memory by default and has a lifetime for the whole execution of the program or until it is released.

Global

Global Memory (global qualifier) is located off-chip also known as DRAM and is the biggest memory storage on the GPU card. The __global__ keyword indicates that the function being declared is a CUDA kernel function. This keyword also used as variable qualifier. The function will be executed on the device and can only be called from the host to generate a grid of threads on a device. The performance related to global memory resides in the fact that all global memory accesses have to be performed perfectly coalesced. This means that all memory requests sent from all threads needs to be combined into a single memory transaction. Efficient use of global memory is essential in a program by avoiding memory bandwidth limitations through data reuse.

Shared

Shared memory (shared qualifier) is located on-chip, a copy for each block launched on the GPU. A running thread has only access to the copy within its own block and cannot change or read from other blocks. The lifetime of shared memory is the same as the lifetime of the block it is created for. This memory is much faster than global memory, and should by all means be used whenever possible where many threads need access to the same memory.

Constant

Constant memory (*__constant__*qualifier) has a limited size. This memory is used to hold constant variables that will and cannot change during the execution of the kernel, and can only be set from the host. All threads on the GPU have access to constant memory.

Registers

Registers are the fastest cache memories on the GPU. To achieve peak performance in a program, registers are the best choice with the lowest latency and highest bandwidth.

Texture

Texture memory is read only on the device and can only be set from the host. It

is however relatively small. Texture memory is a hardware interpolation of the block, and will be used most effectively if it can read clustered blocks on the global memory. This memory is mostly used for graphics, when utilizing multi-sampling techniques and more.

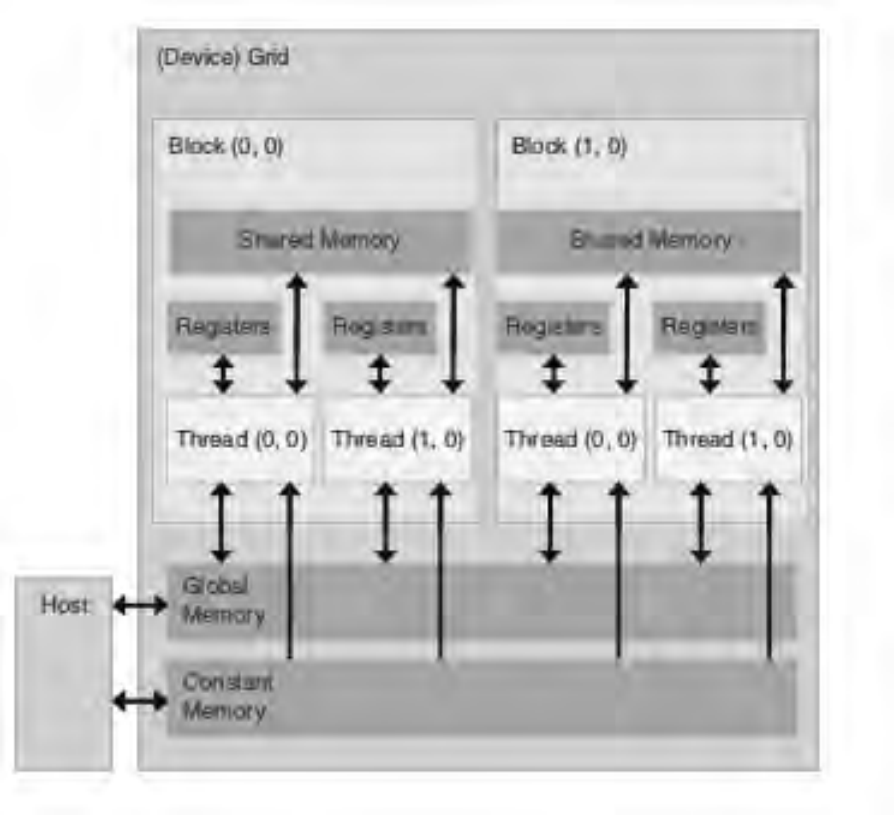

Figure 3.4: Overview of the CUDA device memory model[16]

3.2.3 Compilation and execution

A CUDA source code is a mix of code for both CPU and GPU [12]. The code then is compiled with the nvcc, CUDA C compiler, as shown in Figure 3.5. If the CUDA source code made a call to existing optimized library such as Basic Linear Algebra Subprograms (BLAS), and Fast Fourier Transform (FFT), the library would be included in the compilation as well.

After compilation two files are output which are the machine independent assembly (PTX) code [17] and CPU object code. Then, the PTX is loaded for execution in GPU with the CUDA driver. During the time of execution, the developer can perform runtime investigation with the debugger and profiler tool. For the CPU host code, it is compiled with standard C compiler such as GCC to be run on CPU.

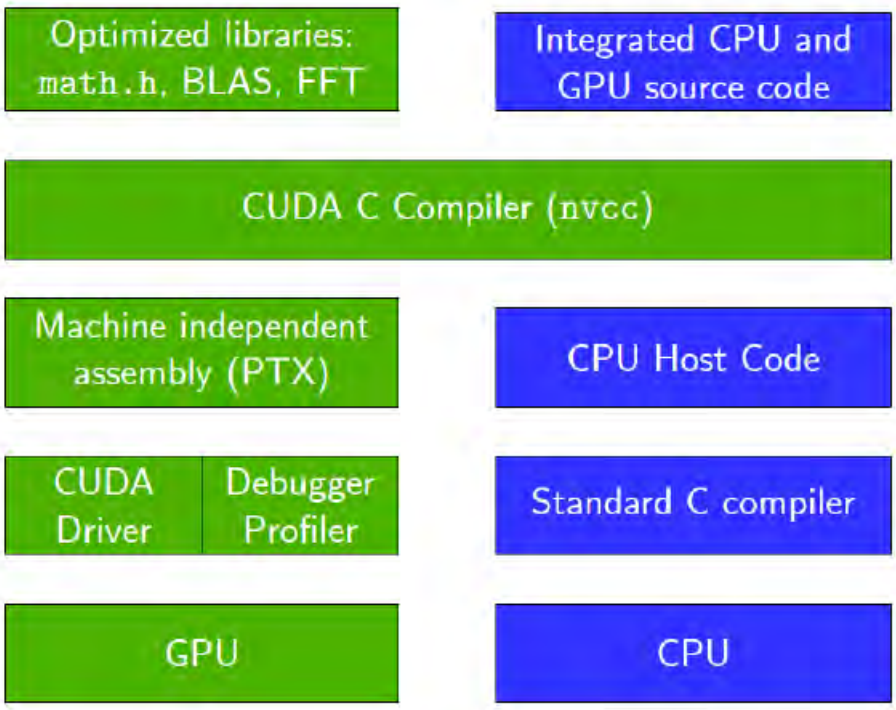

Figure 3.5: CUDA compilation and execution [18]

New revisions of CUDA have a compute capability flag within a given GPU, and are defined by a major and a minor revision number. Compute capability describes which features are supported by the CUDA hardware. At this moment of writing, the newest cards are based on the maxwell architecture and has compute capability 5.2[19]. Compute capability is set as an option flag. Example: arch=compute_21, -code=sm_21 will set the compiler to compute capability 2.1. The major revision number is 2.x, and the minor revision number is x.1.

3.3 Summary

We have seen that GPU is suitable for applications with data intensive, no complex control flow and a negligible dependence on large caches. Also, the CUDA architecture was discussed detailing the compute structure and memory hierarchy. Futhermore, programing constructs such as key words __device__,__global__,__shared__ and __constant__ variable type and function type qualifiers were explained.

Chapter 4

Snort

Snort is widely used network IDS and uses a rule based detection engine to detect malicious activities in the network. It has five components[4] as illustrated in Figure 4.1.

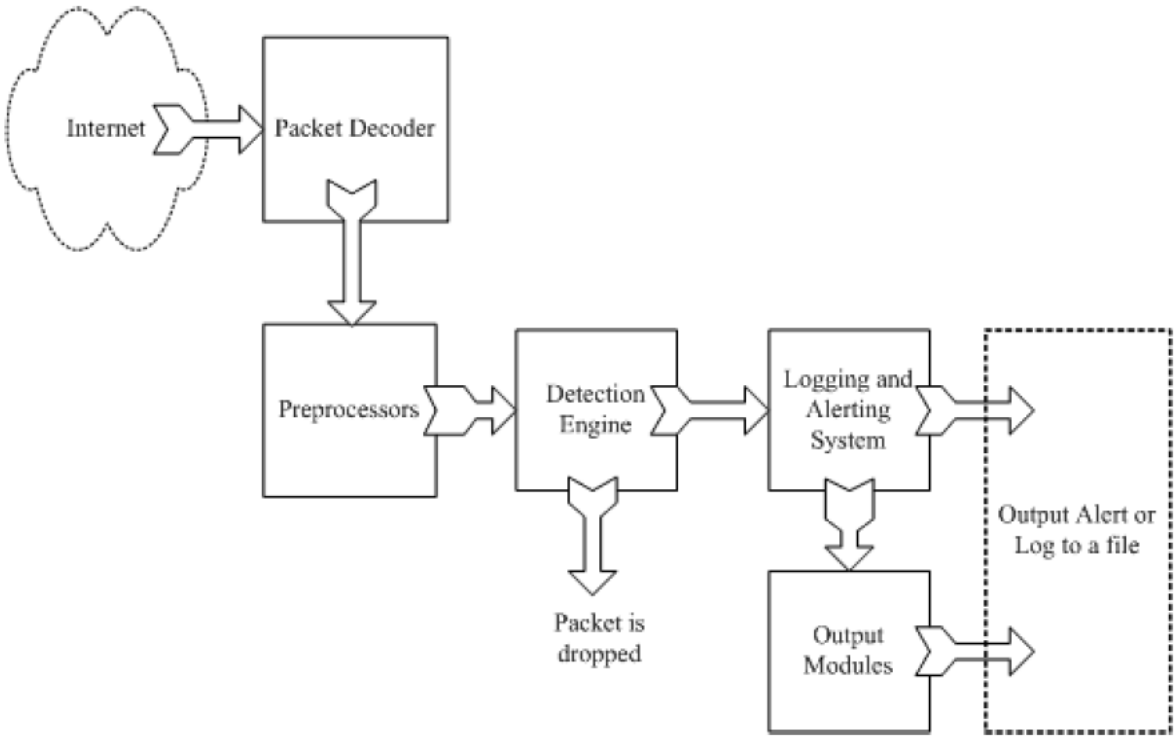

Figure 4.1: Block diagram of components of Snort[4]

Snort relies on an external packet capturing library libpcap to sniff the network packets (winpcap for windows). 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 pass 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.

4.1 Preprocessors of Snort

Preprocessors are components or plug-ins that can be used with Snort to arrange or modify data packets before the detection engine does some operation to find out if the packet is being used by an intruder. Some preprocessors also perform detection by finding anomalies in packet headers and generating alerts[20]. Preprocessors are very important for any IDS to prepare data packets to be analyzed against rules in the detection engine. Generally, preprocessors have three main purposes[20]. These are

- Reassembling packets
- Decoding protocols

- Nonrule or anomaly-based detection

4.1.1 Preprocessor Options for Reassembling Packets

Snort has two preprocessor plug-ins that assist rule-matching by combining data spread across multiple packets[20]: stream 5 and frag3. The stream preprocessor contained in spp stream5.c and improve Snort's handling of TCP sessions for selected traffic. Stream5 has two purposes. These are:

- TCP statefulness
- Session reassembly

TCP statefulness

Stateless devices only look at one packet at a time—they have no memory of the previous packets. This means that their only way of gauging the status of a session is to look at the combination of flags. Stateful devices remember what handshaking packets have been sent and can thus keep track of the state of the connection.

Session Reassembly

The stream5 preprocessor reassembles the TCP stream so that Snort can try rule matches against the whole of the flowing data. The assembly may takes place in the side of client only / server only / or both . The first option tells stream5 how much of the stream it should reassemble. It can simply do reassembly on the client side (traffic going to HOME NET), when you set the client only option, reassembly only on the server side(traffic coming from HOME NET), when you set the server only option, or all traffic, when you set both.

frag3—Fragment Reassembly and Attack Detection

Fragmentation is a normal part of the Internet Protocol (IP). In essence, each type of networking hardware has a different Maximum Transfer Unit (MTU), a

number that quantifies how much data can be transferred in a single “chunk” on the medium. For example, Ethernet’s MTU is 1500 bytes, and it calls its data chunks “frames”. The sending IP stack in a communication generally puts as much data in a packet as it can, basically using the MTU of the outgoing network as a maximum size for the outgoing chunk. If the IP packet, as it goes through a router from one network to the next, is too large for the MTU of the next network, it gets broken into fragments. These fragments basically look like IP packets in their own right and can traverse the network. They are reassembled when they reach their destination. Frag3 is also useful in detecting fragment-based denial-of-service (DoS) attacks. These attacks will often send a series of well-designed fragments to take advantage of a host’s particular IP stack vulnerabilities. For example, some machines will re- boot, halt, or otherwise react negatively when they receive a fragment that has its offset configured to overwrite a previous fragment’s data.

4.1.2 Preprocessor Options for Decoding and Normalizing Protocols

Rule-based pattern matching can often fail on protocols for which data can be represented in many different ways. Web servers accept many different ways of writing a URL. For example, Internet Information Services(IIS) will accept backslash ”\” characters in place of forward-slash ”/” characters in URLs. Telnet, HTTP, and RPC protocols are the common Decoding and Normalizing preprocessors.

The Telnet protocol features are an in line negotiation protocol to signal what features the client and server can offer each other. The client and server intersperse this negotiation data with the normal payload data. Unfortunately, it’s usually the payload data that we want to match our rules against.

HTTP has become one of the most widely and diversely used protocols on the

Internet. Over time, researchers have found that Web servers will often take a number of different expressions of the same URL as equivalent. An attacker can use this flexibility in the Web server to attempt to hide his probes and attacks from the NIDS. Therefore, http inspect preprocessor normalize the web servers to protect network from intruders.

Applications such as Network File Sharing (NFS) and Network Information System (NIS) ride on Sun's Remote Procedure Call (RPC) protocol. Since RPC is intended to carry single messages, but can ride over the stream-based TCP protocol that doesn't distinguish between messages the way UDP does, Sun designed a "record" structure such that each RPC message is encapsulated in a "record."

4.1.3 Preprocessor Options for Nonrule or Anomaly-Based Detection

Anomaly based approaches are used to detect new attacks. These schemes construct statistical models of the typical behavior of a system and issue warnings when they observe actions that deviate significantly from those models. portscan and Bo (Back Orifice) are the common preprocessors of such purposes.

4.2 Summary

This chapter discussed Snort NIDS. Snort has five major components . These components are packet decoder, preprocessors, detection engine, logging and alerting system and output plugin. We have seen also that Preprocessors have three major purposes of reassembling packets, decoding protocols and anomally detection.

Chapter 5

Methodology

This section describes the tools,programming langues, libraries and the input data set used for the work. The experimental set up of the system and the methods used to accomplish the work are also described.

5.1 Developing tools

The tools, programming languages and libararies used for in this work are:

1. Programming Languages and tools:

   (a) CUDA 6.5: CUDA uses C language with additional features to perform general purpose calculations in the GPUs.

   (b) Snort-2.9.6.1: The Snort tool used for the work is version 2.9.6.1.

2. Libraries:

   Libpcap-1.5.3: is packet-capture library for retrieving packets from the network.

3. DARPA dataset

   There were lots of intrusion detection systems developed by researchers from all over the world. These systems needed to be evaluated for their detection capability. In 1998 Information Systems Technology group of MIT Lincoln Laboratory, under the sponsorship of Defense Advanced Research Projects Agency(DARPA ITO) and Air Force Research Laboratory (AFRL), started this evaluation process [21]. The result of this effort is the DARPA 1999 Intrusion Detection Evaluation data set. This data set contains three week of training data and two week long data for testing. The first and third week data set contain no attacks while the second week dataset has several attacks incidents. These three weeks of data is usaually used for training intrusion detection system. The last two weeks contain different types of attacks and are used as testing dataset. In addition, testing data of each day also contains outside and inside traffic data set. Outside and Inside data are collected by a sniffer located outside and inside the simulated network environment in MIT Lincoln laboratory respectively. In this work, week 4 and week 5 dataset are used to test our system.

4. Experiment setup

   A laptop computer with a discrete GPU was used in the experiments conducted in this work. Its specifications are given below in Table 5.1.

Table 5.1: Experimental set up of the experiment

processor	Intel(R) Core(TM) i5-4200M CPU @ 2.50GHz
RAM	4GB
HDD	500GB
GPU	Geforce 710m(Optimus technology)
Operating System	Ubuntu-14.04 LTS 64 bit

5.2 Profiling Snort

In order to identify where the Snort application spends the majority of the execution time, it was profiled using the DARPA data set with gprof. According to results shown in Table 5.2, it was confirmed that the detection engine is the most time consuming part of Snort. In addition, the preprocessor part was found out to the second most compute intensive part.

The results in Table 5.2 prove what we have mentioned in introduction of this work. Numerically explaining the result of the profiling, Snort spends an average 60 to 69 percent of processing time on the detection engine and while the preprocessors take from 30 to 40 percent of processing time. Moreover, it was observed for outside traffic in the dataset run time is higher than inside traffic. This probaly means, there are more attack in outside traffic than inside.

Table 5.2: Profiling Snort on the DARPA Data set

Layer	week4	week5
-	inside—outside	inside—outside
Detection	64.195—66.133	65.72—65.998
Preprocessors		
stream5	36.253—42.793	36.82—40.93
http inspect	6.24—7.82	5.74—6.18
smtp	0.983—0.98	0.55—0.665
telnet	0.335—0.268	0.146—0.185
flow	0.253—0.288	0.22—0.253
pop	0.154—0.168	0.158—0.198
Imap	0.103—0.113	0.102—0.13

Note: the numerical values in Table 5.2 are all percentages.

Among the preprocessors, stream5 is the most compute intensive part. Therefore, the sream5 preprocessor was selected to be accelerated on GPU using CUDA. The stream5 preprocessor was investigated further to see if the code was suitable for porting to GPU. The next step is to identify which components of the

stream5 preprocessors can be modified to run on the GPU. The list in Appendix A of sample profile shows this result. Among the componenets or functions, *Stream5AlertFlushStream* and *preprocess* are selected to run on GPU. These functions are selected based on the criteria of porting we have mentioned in Section 3.1. There are some other intensive functions but not suitable for porting. In most functions the problem is branching. Branching causes serial execution of each thread participating in the branch, and so it hinders performance. The *preprocess* function has branching but we run two concurrent kernels and we got the promosing result. Therefore, we can minimize the branching problem by dividing the branching portion in to different kernels to run concurrently.

After selecting the function to be ported to run on GPU, the next step was transferring packets from CPU to GPU. There is transfer overhead between CPU and device communication because of the PCIe bus. The bandwidth between CPU and the device is small compared to CPU to CPU and device to device communication. Therefore, a new technique is required to achieve the best result. To solve this problem and others the following methodology was proposed.

5.3 Proposed Solution

In order to overcome the communication overhead between CPU and the device we have to minimize the frequent interaction between the CPU and device . Therefore, we stored packets temporarily in a buffer and copied collected packets to the device so that accessing memory only at intervals.

First a buffer is created for temporarily storing of packets in memory on the host side. The size of the buffer depends on the number of packets processed. In our case, the number of packets are limited based on the capacity of shared memory the test computer. The size of shared memory for the test computer is 6144KB(i.e.

48KB per block*16 streaming processor*8 thread blocks). Based on this limitation up to 32768 number of packets can be processed without decrease in performance.

After packets are transferred to the device,the next step is doing the operation on the device. Then, the result is copied back to CPU . In this way of processing packets still there is an associated limitation on transferring to the device and accessing input from device memory. The following mechanisms are important to enhance the performance further.

In the initial port to the GPU, the memory on the host side is allocated using malloc() which allocates pageable memory (memory locations that can be swapped out to virtual memory on the disk). In order to use pinned memory located on the host memory which is accessible by the device(GPU), a page locked memor must be allocated on the host. This allocated memory will not be swapped to disk by the Operating System, but remains at the same location, with the same address space for the whole duration of the application[22]. The function cudaHostAlloc() is used to allocate a page-locked buffer on host memory.The GPU can then use direct memory access (DMA) to copy data to or from the host.

Pinned-memory use can be dangerous unless used properly. Specifically, the computer running the application needs to have available physical memory for every page-locked buffer, since these buffers can never be swapped out to disk. This mean that an application might start to fail on machines with smaller amounts of physical memory and/or the performance of other applications running on the system may be affected.

Similarly, in the initial port of the function to GPU the packets are stored in global memory. Each thread executes the application by accessing the inputs from global memory. This is due to the fact that global memory is accessible to all threads[16]. But, global memory is not in the chip and takes large time to access it. On the other hand, shared memory is located on-chip and is available for each block of

threads launched on the GPU. A running thread has access to shared memory within its own block and cannot change or read from other blocks. This memory is much faster than global memory. Using shared memory as a cache for global memory accesses increases the performance of applications.

5.4 Summary

In this chapter the methodologies followed were discussed. The developing tools and the experimental set up used were also described. The profiling result to identify which part of Snort is intensive is given. Then, batch processing of packets are proposed to effectively reduce the data transfer overhead between the host and GPU. Further optimizations to reduce of the effect of data transfer such as using pinned-memory and shared memory were proposed.

Chapter 6

Results and Discussion

This Chapter discusses the results obtained from the experiments conducted. First the *Stream5AlertFlushStream* function test results is given. Secondly, experiments conducted for *preprocess* function are detailed. Lastly, the overall system performance is described.

In all the experiments conducted, the output of the modified Snort has been verified against the orignal Snort. Week4 and Week5 data from the DARPA dataset was used as an input for the experiments. Testing is done using inside and outside testing data of the two weeks. Although, among the two weeks dataset Tuesday of week4 for inside and Friday of week5 for outside are not available in the dataset.

6.1 Testing Stream5AlertFlushStream

This function is used to flush packets or session. Based on the rules of Snort, this function is important to selectively drop/discard packets like packets with incorrect header length. Table 6.1 and Table 6.2 shows the result of original CPU

and initial port of GPU implementation for the two weeks data set(The raw results are available in Appendix B). From the experimental result, we can see that for small number of packets the execution time of GPU is much higher than CPU for both inside and outside test data. This is due to the latency to copy packets from main memory to global memory. On the other hand, when the number of packets processed increases, the execution time for the GPU is reduced respectively. This justfies the processing of large number of packets in batch by storing them in a buffer and copying them to the GPU. This increases performance by minimizing latency due to frequent interaction of CPU and GPU.

We have got a maximum speed up of 2.8 for Friday of week4 and 2.44 for Monday of week5 using outside data as input. Similarly, using inside data as input, the performance of the system is increased by 2.87 for Friday of week4 and 2.71 for Monday of week5. The minimum enhancements achieved are 2.34 times for week4 and 2.36 times for week5 both in Wednesday when outside test data is used. On the other hand, for inside traffic test data the minimum accelerations are 2.22 times for Wednesday of week4 and 2.24 times for Friday of week5. These results were achieved by the introduced batch processing of packets and gives an average speed up of 2.53. However, there are still other optimizations techniques that can offer further speed up.

Table 6.1: The performance comparison of CPU and initial port of GPU using week4 traffic testing data

Ratio	Traffic type	Monday	Tuesday	Wednesday	Thursday	Friday
CPU/GPU	outside	2.66	2.619	2.34	2.44	2.8
CPU/GPU	inside	2.77	–	2.22	2.62	2.87

Table 6.2: The performance comparison of CPU and initial port of GPU using week5 traffic testing data

Ratio	Traffic type	Monday	Tuesday	Wednesday	Thursday	Friday
CPU/GPU	outside	2.44	2.41	2.36	2.4	–
CPU/GPU	inside	2.4	2.71	2.7	2.62	2.24

6.1.1 Optimization using Page-locked Memory

In the initial port of the function, the memory on the host side is allocated using malloc() which allocates pageable memory. In pageable memory copy happens twice. The CUDA driver first copies from pageable memory to page-locked memory, then from page-locked system to the GPU. This mechanism decreases performance due to the high latency to transfer data to the GPU. Using page-locked memory instead solves this problem.

Table 6.3 and Table 6.4 show the performance of using page-locked memory for two weeks outside and inside input data .The maximum acceleration achieved is 2.85 from week4 of Friday and 2.47 from week5 of Monday for outside test data. On the other hand,the maximum acceleration achieved for inside test data are 2.89 for week4 of Friday and 2.72 for week5 of Monday . These test results are higher than what we have achieved in the initial port to GPU . From the whole experiments using page-locked memory the minimum speed ups from outside input data are 2.44 for week4 and 2.39 for week5, both for Wednesday. When inside data is used as input, the minimum performances are 2.23 times for Wednesday of week4 and 2.25 times for Friday of week5. In average we have got 2.57 speed up for using page-locked memory.

Table 6.3: The performance comparison of GPU and page-locked memory for week4 traffic testing data

Ratio	Traffic type	Monday	Tuesday	Wednesday	Thursday	Friday
CPU/pl	outside	2.69	2.67	2.44	2.45	2.85
CPU/pl	inside	2.81	–	2.23	2.63	2.89

Table 6.4: The performance comparison of GPU and page-locked memory for week5 traffic testing data

Ratio	Traffic type	Monday	Tuesday	Wednesday	Thursday	Friday
CPU/pl	outside	2.47	2.46	2.39	2.46	–
CPU/pl	inside	2.41	2.72	2.71	2.63	2.25

Note: *pl* stands for page-locked in above tables

6.1.2 Optimization using both Page-locked and Shared memories

In the initial port of the function to GPU the packets are stored in global memory. Each thread executes the application by accessing the inputs from global memory. This is due to the fact that global memory is accessible to all threads as discussed in Section 3.2.2. But, global memory is not in the chip and takes large time to access it. On the other hand, shared memory is located on the chip. Due to this reason threads access shared memory at a faster rate.

From Table 6.5 and From Table 6.6, we can see that, using both page-locked and shared memories at the same time increased the performance of the system higher than using page-locked memory only for all test data. From the experiments we have got maximum acceleration of 5.36 times from Friday of week4 and 3.88 from Monday of week5 using outside test data. While using inside test data gives

maximum acceleration of 5.1 times from Friday of week4 and 4.48 times from Tuesday of week5. On the other hand, the minimum accelerations acchieved using outside test data are 3.58 time for week4 and 3.44 for week5 both in Wednesday test data. Further more, the minimum enhancements achived from inside test data are 3.1 Wednesday of week4 and 3.14 Friday of week5. The average speed up achieved using both page-locked and shared memories is 4.02.

Table 6.5: The performance comparison of CPU,initial port,using page-locked memory and using both page-locked and shared memories together for week4 traffic testing data

Ratio	Traffic type	Monday	Tuesday	Wednesday	Thursday	Friday
CPU/GPU	out	2.66	2.619	2.34	2.44	2.8
CPU/GPU	in	2.77	–	2.22	2.62	2.87
CPU/pl	out	2.69	2.67	2.44	2.45	2.85
CPU/pl	in	2.81	–	2.23	2.63	2.89
CPU/pls	out	3.67	4.36	3.58	3.67	5.36
CPU/pls	in	4.91	–	3.1	4.2	5.1

Table 6.6: The performance comparison of CPU,initial port,using page-locked memory and using both page-locked and shared memories together for week5 traffic testing data

Ratio	Traffic type	Monday	Tuesday	Wednesday	Thursday	Friday
CPU/GPU	out	2.44	2.41	2.36	2.4	–
CPU/GPU	in	2.4	2.71	2.7	2.62	2.24
CPU/pl	out	2.47	2.46	2.39	2.46	–
CPU/pl	in	2.41	2.72	2.71	2.63	2.25
CPU/pls	out	3.88	3.68	3.44	3.63	–
CPU/pls	in	3.5	4.48	4.45	4.2	3.14

Note: In the above tables *pl* is for page-locked , *pls* for page-locked and shared, in for inside and out for outside.

From the experiments we have seen so far, the test results for inside data and outside data are enhanced. But, using inside test data as input gives a little larger enhancement than outside test data. This is due to the fact that, there is more attack in outside test data than inside. Therefore, it takes large amount of time to process more attacks than small attacks. In general, we can observe that the page-locked with shared memory version of our implementations gives the best speed up.

6.2 Testing Preprocess Function

The *preprocess* function has branching which hinders the utilization of the GPU's performance. As discussed in Section 3.2, if there are two groups of ALUs that would like to process different control flows, both control flows have to be taken sequentially. This will reduce the system utilization. Therefore, to reduce the performance hit the instructions in the different branches are executed as multiple kernels. This helps perfromance on devices that are capable of concurrent kernel execution, where streams used to execute multiple kernels simultaneously to take advantage of the device's multiprocessors. The instrucitons in the *preprocessor* are therefore separated in two kernels to be executed on the GPU simultaneously.

The result of original CPU and initial port of GPU implementation is shown in Table 6.7 and Table 6.8. Similar to the previous test data, the execution time of GPU is higher than CPU for small number of packets. However, when the number of packets processed increased, the performance of the GPU did not increase as in the previous shown acceleration of *Stream5AlertFlushStream*(See the raw result in Appendix B). This is because of the branching in the *preprocess* function. The maximum performance achived for large number of packets using outside input is 1.6 times for Friday of week4 and 1.54 for Monday of Week5. On the other hand, using inside input gives maximum performance of 1.66 times For Friday and 1.63 times for Monday of week4 and Week5 respectively. The average speed up for initial port is 1.57.

Table 6.7: The performance comparison of CPU and initial port to GPU week4 traffic testing data

Ratio	Traffic type	Monday	Tuesday	Wednesday	Thursday	Friday
CPU/GPU	out	1.59	1.52	1.55	1.56	1.6
CPU/GPU	in	1.66	–	1.49	1.63	1.67

Table 6.8: The performance comparison of CPU and initial port to GPU week5 traffic testing data of preprocess

Ratio	Traffic type	Monday	Tuesday	Wednesday	Thursday	Friday
CPU/GPU	out	1.5	1.53	1.54	1.51	–
CPU/GPU	in	1.55	1.67	1.67	1.62	1.46

Note: *in* stands for inside and *out* stands for outside in above tables

To minimize performance hit due to branching in *preprocess* function, it was split in to two kernels. Using these two kernels simultaneously gives maximum performance of 1.69 times for Friday and 1.59 times for Monday for Week4 and for Week5 respectively for outside input data. Similarly, performing the same experiment using inside input data the performance was 1.69 times for Friday and 1.67 times for Monday of Week4 and Week5 respectively as shown in Table 6.9 and Table 6.10. Therefore, using two kernels instead of one kernels increased the performance. Overall, using two kernels simultanously gives an average speed up of 1.61.

For further optimization, shared memory is used along with the two kernels. By doing so, performance is enhanced by maximum of 1.72 for Friday and 1.63 times for Monday of week4 and week5 respectively when outside data is used as input. While using inside test data as input, we have got maximum acceleration of 1.7 times for Friday and 1.68 times for Monday of week4 and week5 respectively. This gives an average performance improvement over the original using two kernels and shared memory is 1.62.

Table 6.9: The performance comparison of CPU,initial port,using two kernels and using both two kernels and shared memory together for week4 traffic testing data

Ratio	Traffic type	Monday	Tuesday	Wednesday	Thursday	Friday
CPU/GPU	out	1.59	1.52	1.55	1.56	1.6
CPU/GPU	in	1.66	–	1.49	1.63	1.67
CPU/twok	out	1.69	1.67	1.56	1.57	1.66
CPU/twok	in	1.67	–	1.492	1.64	1.68
CPU/tks	out	1.7	1.68	1.561	1.58	1.67
CPU/tks	in	1.68	–	1.5	1.643	1.69

Table 6.10: The performance comparison of CPU,initial port,using two kernels and using both two kernels and shared memory together for week5 traffic testing data

Ratio	Traffic type	Monday	Tuesday	Wednesday	Thursday	Friday
CPU/GPU	out	1.5	1.53	1.54	1.51	–
CPU/GPU	in	1.55	1.67	1.67	1.62	1.46
CPU/twok	out	1.59	1.58	1.55	1.57	–
CPU/twok	in	1.56	1.68	1.68	1.63	1.48
CPU/tks	out	1.6	1.59	1.56	1.58	–
CPU/tks	in	1.56	1.69	1.68	1.64	1.49

Note: *twok* stands for two kernel and *tks* stands for two kernel and shared in above tables .

6.3 Overall performance of Snort

So far, we have seen the individual perfromance improvement of the two functions identified as a bottleneck in the preprocessors of Snort. In this section the overall performance of Snort with the modified *Stream5AlertFlushStream* and the *preprocess* functions included is described.

As described in Section 3.2.2, CUDA device functions can only be called from a kernel function or another device function. This requires an integration strategy for the newly implemented functions on the GPU to work with the existing Snort code base.Therefore, wrapper function were developed to hide the details of the new implementation to make the integration seamless. Wrapper functions are used to allocate memory on the device part, transfer data to the device and from the device and launching of kernel.

Different cases are taken to test the overall performance of the system. The first case is testing the system by replaying the pcap file with the same speed as it was captured. The other cases are replaying the pcap file to the network at arbitrary speed including the top speed. In the individual improvement of the function, we have got up to 5 times acceleration. Since these functions are small part of Snort, the effect of accelerating these functions on the overall Snort application is presented.

To replayed captured pcap files to the network at different speed, we used tcpreplay. Tcpreplay is a suite of utilities for UNIX -like systems for editing and replaying network traffic which was previously captured by tools like tcpdump. The goal of tcpreplay is to provide the means for reliable and repeatable traffic for testing a variety of network devices such as switches, router, firewalls, network intrusion detection and prevention systems (IDS and IPS)[23].

When the pcap file is replayed to the network with the same speed as it was

captured, the maximum performance is 0.79 percent for week4 and 0.94 percent for week5. Next, packets are replayed with 1000 packets per second. At this time the maximum performance achieved is 0.77 percent for week4 and 0.4 perecent for week5 . Then, packets are replayed to the network at speed of 10 mega bits per second and enhances the overall performance by maximum of 1.64 percent for week4 and 1.38 percent for week5.

The final test was replaying the packets at high speed as much as possible. For the Orginal CPU implementation, packets were replayed/processed at maximum speed of 3552.2 Mbps while the GPU implementation reached 4101.17 Mbps. This shows a performance improvement of the overall system by 15.5 percent in how fast packets can be processed. The comparison among the scenarios are shown in Figure 6.1, Figure 6.2, Figure 6.3 and Figure 6.4 .

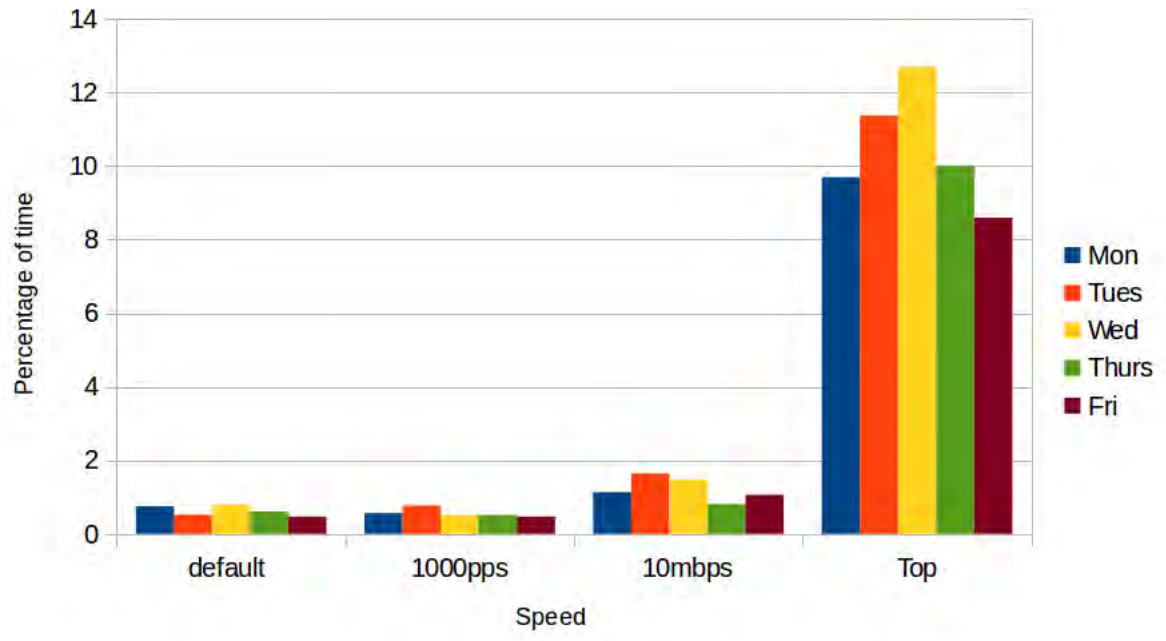

Figure 6.1: Speed up of overall performance for week4 outside dataset

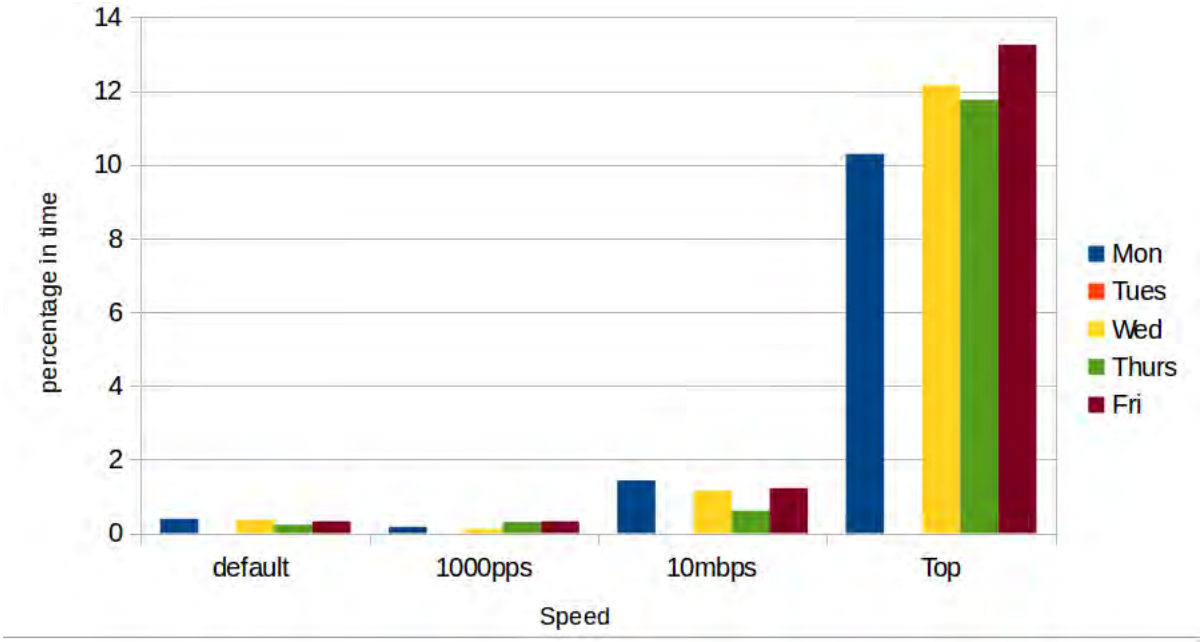

Figure 6.2: Speed up of overall performance for week4 inside dataset

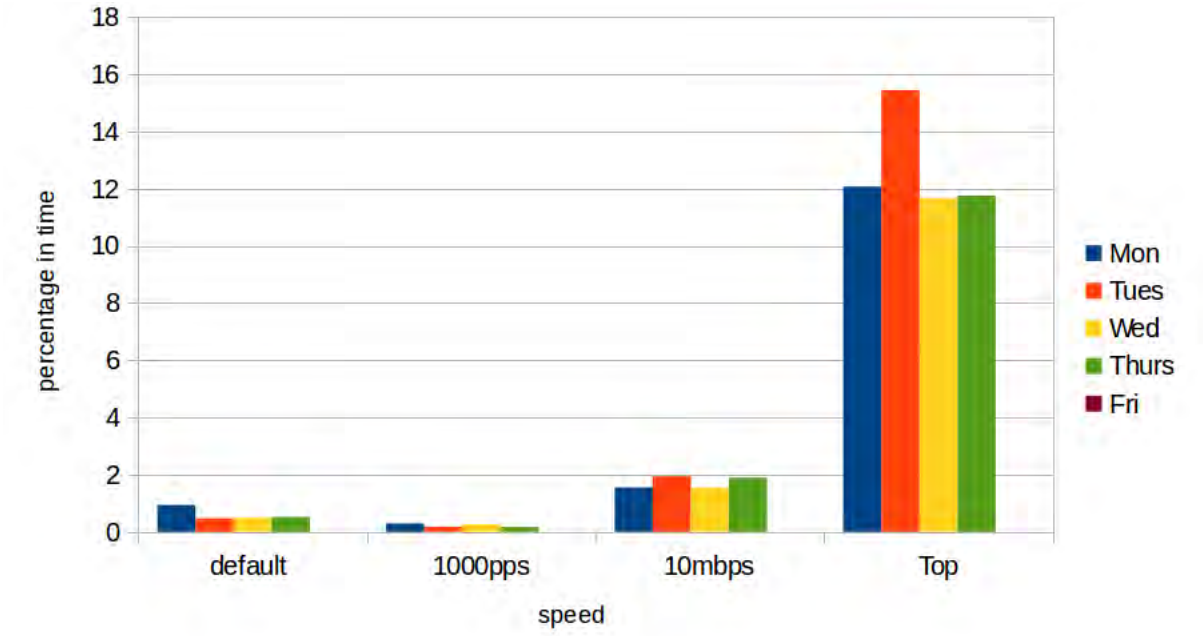

Figure 6.3: Speed up of overall performance for week5 outside dataset

Profile result of Snort after ported to GPU

The modified Snort implementations was profiled to compare with original. The profile result of Snort after porting chosen functions to GPU shows some improvement over the original especially for stream5 preprocessor. We can see that there is around maximum of 12 percent improvement in stream5 preprocessor as shown

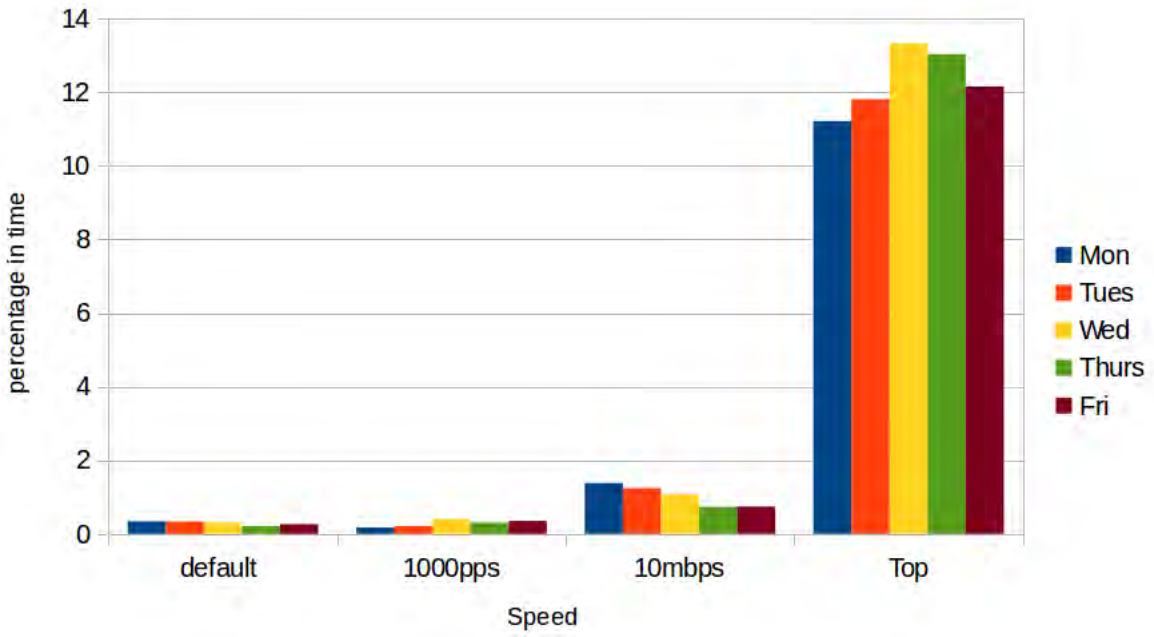

Figure 6.4: Speed up of overall performance for week5 inside dataset

in Table 6.11. For the other components there is slight difference in percentage among the two profile results on the positive side.

Table 6.11: DARPA Data set test result of Snort after porting

Layer	week4	week5
	inside—outside	inside—outside
Detection	67.25—69.58	68.96—69.23
Preprocessors	-	
stream5	32.03—38.87	33.12—37.48
http inspect	7.24—7.98	6.24—6.33
smtp	0.85—0.99	0.62—0.72
telnet	0.17—0.28	0.15—0.19
flow	0.26—0.29	0.2—0.25
pop	0.154—0.18	0.16—0.2
Imap	0.11—0.12	0.10—0.13

Note: inside and outside are all in percentages.

6.4 Summary

In this chapter we have discussed the implementation of the proposed design of processing packets, memory optimization and the analysis of it. From the results we have seen that using GPU for small number of packets is unrealistic. And GPU is suited for large number of packets to be processed. Using both page-locked and shared memories together gave performance improvement of up to five times for Stream5AlertFlushStream function and an average of 1.62 speed up for preproces functions. It was also observed how branching affects the performance on the GPU. The overall system performance was seen to improve by 15.5 percent when using GPUs for the preprocessors of Snort.

Chapter 7

Conclusions and Recommendations

7.1 Conclusion

The main goal of the thesis was to find out if the performance of the preprocessor of Snort NIDS could be enhanced by the use of GPUs. In order to achieve this goal the preprocessor of Snort were profiled. After profiling of Snort, further investigation was taken to identify whether the functions identified as bottlenecks were suitable to be ported to GPU. The functions found to be suitable *Stream5AlertFlushStream* and *Preprocess* were ported to the GPU and optimized.

The first optimization mechanism was using page-locked memory along with streams instead of using normal pageable memory. Using this method enhanced the performance of for Stream5AlertFlushStream function on average 2.57 times and 1.57 times for Preprocess function as compared to the original implementation. The second optimization used page-locked memory on the host side and shared memory on the device part. After the experiment, we have got up to 5 times improvement

from the original CPU implementation for Stream5AlertFlushStream function. By using two kernels and shared memory for Preprocess function speed up by 1.62. Overall, the modified Snort after integrating the functions ported to GPU was found out to be 15.5 percent faster in processing packets than the original CPU implementation.

From this work the following conclusions can be drawn:

- Packets should be processed in batch by collecting them in a buffer for GPU implementation in Snort
- For optimization of the programs on GPU page-locked memory is good but using both page-locked and shared memories are more advantageous for performance enhancement.
- GPU is good for processing large number of packets rather than small number of packets. Therefore, it is unrealistic to process small number of packets on GPU.
- Utilizing GPUs in Snort can actually increase the speed of packet processing.

7.2 Recommendation

There is a room for improvement for future work. As future work, this application can be ported to GPU clusters(multiple GPU devices) that will run in parallel. When the number of GPU cards used increases, a corresponding performance enhancement is expected. Using unified memory of the current devices may increase the performance. Unified memory is currently introduced after CUDA-6.0 and can be available devices with compute capabilities greater than or equal to three.

An other research can be conducted to switch between CPU for low traffic and GPU for high traffic. There are also preprocessor functions which are intensive but have branches. By devoting more effort on multiple kernel execution these functions may boost the performance of Snort.

We transferred packets to GPU when the buffer is full. But, there might be occasions in real network that the buffer may not be full in certain time when the speed is slow. In this case packets may drop because of its life span. Therefore, the life span of packets should be considered and transferred to GPU when the buffer is not full within a certain time limit to prevent dropping of packets.

Bibliography

[1] Anjou Panicker Madhusoodhanan Sathik *Parallelizing a network intrusion detection system using GPU.* A Master's Thesis Submitted to Department of Computer Science and Engineering, University of Louisville, Louisville, Kentucky May 2012

[2] Mazen Kharbutli, Monther Aldwairi, and Abdullah Mughrabi. *Function and Data Parallelization of Wu-Manber Pattern Matching for Intrusion Detection Systems,*Jordan University of Science and Technology, Irbid, Jordan, July 2012

[3] Mauno Pihelgas. *A comparative analysis of open-source intrusion detection systems.* A master's thesis submitted to Tallinn University of Technology, Department of Computer Science, 2012

[4] Rafeeq Ur Rehman. *Intrusion Detection Systems with Snort, Advanced IDS Techniques Using Snort, Apache, MySQL, and PHP, and ACID,*pages 12-15, 2003

[5] Kristian Nordhaug *GPU Accelerated NIDS Search.* A Master's Thesis Submitted to Department of Computer Science and Media Technology for Master of Science in Information Security Gjøvik University College, 2012

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

[7] Giorgos Vasiliadis, Spiros Antonatos, Michalis Polychronakis, Evangelos P. Markatos, and Sotiris Ioannidis. *Gnort: High performance network intrusion detection using graphics processors.* In Proceedings of the 11th international symposium on Recent Advances in Intrusion Detection, RAID '08, pages 116–134, Berlin, Heidelberg, 2008. Springer-Verlag.

[8] Che-Lun Hung, Hsiao-hsi Wang, Chin-Yuan Chang and Chun-Yuan Lin. *An Efficient Packet Pattern Matching for Gigabit Network Intrusion Detection*

*using GPUs.*IEEE 14th International Conference on High Performance Computing and Communications, 2012

[9] Nigel Jacob and Carla Brodley *Offloading IDS Computation to the GPU*, 2004

[10] Giorgos Vasiliadis, Michalis Polychronakis, and Sotiris Ioannidis. *Parallelization and characterization of pattern matching using GPUs.* In Workload Characterization (IISWC), 2011 IEEE International Symposium on, pages 216 –225, nov. 2011

[11] Graphics Processing Unit Definition *http://techterms.com/definition/gpu*Last Accessed December,2014

[12] cuda_c programming guide *http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf, August 1, 2014*

[13] Rerngvit Yanggratoke, *GPU Network Processing,*Stockholm, June 15, 2010

[14] Nvidia,Cuda visual profiler version 6.5 *http://docs.nvidia.com/cuda/profiler-users-guide/index.html*, 2014

[15] Nvidia CUDA Occupancy calculator*http: //developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls*. Last Accessed April,2015

[16] David B. Kirk and Wen-mei W. Hwu, *Programming Massively Parallel Processors.* A Hands-on Approach,2010

[17] CUDA development team, CUDA enabled GPUs *https://developer.nvidia.com/cuda-gpus*,January,2015

[18] J.Seland CUDA Programming *//heim.ifi.uio.no/ knutm/geilo2008/seland.pdf*, 2008

[19] Nvidia. Nvidia compute ptx: Parallel thread execution*http://www.cs.cmu.edu/afs/cs/academic/class/15668s11 /www/cuda-doc/ptx_isa_1.4.pdf, August 23 ,2010.*

[20] Caswell, Brian. *Snort 2.1 Intrusion Detection,*Second Edition, Syngress 2004.

[21] DARPA Intrusion Detection Data Sets *http://www.ll.mit.edu/ideval/data/* ,1999

[22] JAson sAndersedwArd KAndrot,cuda by example. *An Introduction General-Purpose Programming,*July 2010

[23] Control and replay network traffic with tcpreplay *https:// tournasdimitrios1.wordpress.com/ 2011/02/14/control-and-replay-network-traffic-with-tcpreplay/*, 2014

Appendix

Appendix A profile result of Snort

Flat profile:

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
8.98	0.65	0.65	1273177	0.00	0.00	in_chksum_tcp
4.42	0.97	0.32	1300257	0.00	0.00	DecodeIP
4.28	1.28	0.31	1376598	0.00	0.00	Preprocess
3.45	1.53	0.25	1301254	0.00	0.00	PrintIPHeader
3.18	1.76	0.23	1376598	0.00	0.00	PacketCallback
2.97	1.98	0.22	2600514	0.00	0.00	_dir_sub_lookup
2.83	2.18	0.21	1376598	0.00	0.00	ProcessPacket
2.76	2.38	0.20	1273177	0.00	0.00	DecodeTCP
2.35	2.55	0.17	3902976	0.00	0.00	SnortSnprintf
2.35	2.72	0.17	1273177	0.00	0.00	CreateTCPFlagString
1.80	2.85	0.13	2753196	0.00	0.00	sfNetworkGetBinding
1.66	2.97	0.12	1300257	0.00	0.00	PrintIPPkt
1.52	3.08	0.11	1376598	0.00	0.00	DecodeEthPkt
1.52	3.19	0.11	1300257	0.00	0.00	ts_print
1.52	3.30	0.11	1300257	0.00	0.00	sfiph_build
1.45	3.41	0.11	1376598	0.00	0.00	PrintPacket
1.38	3.51	0.10	17066950	0.00	0.00	Event_Enabled
1.38	3.61	0.10	2600514	0.00	0.00	sfxhash_count
1.38	3.71	0.10	1374530	0.00	0.00	Detect
1.31	3.80	0.10	3997538	0.00	0.00	PushLayer
1.24	3.89	0.09	1300257	0.00	0.00	in_chksum_ip
1.10	3.97	0.08	2600514	0.00	0.00	sfrt_lookup
1.10	4.05	0.08				orig_ip4_ret_id
1.10	4.13	0.08				pcap_process_loop
1.04	4.21	0.08	1273177	0.00	0.00	PrintTCPHeader
0.97	4.28	0.07	2602508	0.00	0.00	sfip_set_raw
0.97	4.35	0.07	829	0.08	0.08	orig_ip4_ret_proto
0.97	4.42	0.07				SFAT_ReloadAttributeTableThre
0.97	4.49	0.07	2753196	0.00	0.00	SnortEventqReset
0.97	4.56	0.07	2602508	0.00	0.00	inet_ntoax
0.97	4.63	0.07	1301254	0.00	0.00	PrintIpAddrs

Figure 7.1: Top profile results of Snort

Appendix B Raw test results of the two weeks dataset

Test results of stream5FlushAlertStream Function

Table 7.1: Week4 Monday outside and inside input data test result in ms for stream5Flushalertstream function.

No. of packets	CPU	GPU	using Pinned M.	both pinned and shared
for outside traffic-	-	-	-	
128	8.553	162.322	157.02	152.703
256	13.932	178.544	1169.35	161.527
512	23.024	196.524	191.326	183.587
1024	221.414	208.178	203.580	191.845
2048	503.834	319.288	309.296	278.368
4096	1234.346	584.924	552.893	356.482
8192	2859.582	1271.589	1233.656	798.878
16384	6895.963	2589.363	2563.154	1879.356
32768	12563.042	5562.897	5538.985	4097.587
for inside traffic	-	-	-	
128	9.383	176.338	174.259	171.043
256	18.851	184.11	182.529	178.325
512	27.525	197.407	188.13	182.719
1024	220.798	209.259	198.472	191.137
2048	618.5	275.778	255.362	216.992
4096	1445.589	549.371	521.053	294.368
8192	3139.676	1132.502	1115.322	659.975
16384	6715.238	2517.611	2503.421	1558.372
32768	13747.308	5130.314	5109.359	3181.766

Table 7.2: Week4 Tuesday outside and inside input data test result in ms for stream5Flushalertstream function.

No. of packets	CPU	GPU	using Pinned M.	both pinned and shared
for outside traffic-	-	-	-	
128	13.517	206.695	162.761	157.921
256	16.43	212.109	167.576	164.278
512	23.885	218.976	173.398	167.589
1024	225.549	229.814	186.507	175.113
2048	655.139	316.994	285.922	224.808
4096	1539.921	668.678	618.287	376.206
8192	3221.172	1269.157	1223.968	741.291
16384	6752.756	2578.106	2525.511	1548.393
32768	14196.193	5633.579	5567.825	3641.119
for inside traffic	not available	-	-	

Table 7.3: Week4 Wednesday outside and inside input data test result in ms for stream5Flushalertstream function.

No. of packets	CPU	GPU	using Pinned M.	both pinned and shared
128	13.256	206.722	168.219	164.675
256	18.547	214.206	173.615	170.132
512	38.778	232.032	180.311	177.042
1024	271.92	243.827	202.949	196.987
2048	688.587	385.2	348.736	245.232
4096	1641.307	754.948	717.733	491.411
8192	3463.393	1507.799	1418.867	966.169
16384	7227.593	3085.921	3033.901	2061.201
32768	15209.703	6589.626	6562.063	4609.307
for inside traffic	-	-	-	
128	12.954	225.667	222.751	219.166
256	23.247	229.188	227.922	225.502
512	48.75	247.032	242.078	236.177
1024	278.514	272.716	269.166	263.855
2048	709.508	402.47	391.254	308.698
4096	1655.385	807.704	792.33	574.667
8192	4076.617	2125.949	2107.715	1628.768
16384	7809.43	3652.283	3632.388	2698.332
32768	15638.817	7044.925	7023.661	5094.872

Table 7.4: Week4 Thursday outside and inside input data test result in ms for stream5Flushalertstream function.

No. of packets	CPU	GPU	using Pinned M.	both pinned and shared
for outside traffic-	-	-	-	
128	17.382	349.145	317.164	313.913
256	20.96	355.471	324.988	319.48
512	32.34	361.462	331.536	324.218
1024	265.899	392.934	347.576	342.21
2048	721.366	538.235	498.374	400.773
4096	1645.359	942.082	893.695	632.782
8192	3475.673	1671.653	1610.527	1119.245
16384	7104.982	3094.887	3042.948	2063.403
32768	14338.815	5872.075	5843.151	3903.951
for inside traffic	-	-	-	
128	12.791	420.176	419.82	416.989
256	20.375	428.475	426.18	422.945
512	25.25	434.221	433.913	427.119
1024	253.075	471.809	466.871	460.99
2048	708.369	605.99	592.216	501.544
4096	1637.481	972.474	961.558	746.582
8192	3331.291	1598.281	1585.324	1125.305
16384	6739.27	2783.17	2764.833	1783.937
32768	13600.914	5186.939	5167.46	3241.778

Table 7.5: Week4 Friday outside input data test result in ms for stream5Flush-alertstream function.

No. of packets	CPU	GPU	using Pinned M.	both pinned and shared
128	13.628	189.94	145.499	142.851
256	16.89	191.096	154.147	149.081
512	28.004	197.82	159.224	153.891
1024	233.105	216.013	175.395	168.843
2048	625.886	301.319	238.247	186.989
4096	1444.571	578.097	544.536	289.729
8192	3079.699	1090.454	1073.951	574.104
16384	6523.611	2394.871	2340.046	1375.519
32768	13741.461	5151.65	5082.28	3188.069

Table 7.6: Week4 Friday inside input data test result in ms for stream5Flush-alertstream function.

No. of packets	CPU	GPU	using Pinned M.	both pinned and shared
128	8.42	218.226	216.333	213.118
256	13.95	223.149	220.843	218.803
512	22.54	234.685	225.708	221.025
1024	199.751	249.703	240.762	228.011
2048	611.640	316.301	311.862	265.348
4096	1408.692	576.436	568.666	332.732
8192	3046.441	1087.607	1080.829	612.75
16384	6372.86	2218.662	2206.877	1258.779
32768	13656.122	5089.543	5063.129	3183.522

Table 7.7: Week5 Monday outside input data test result in ms for stream5Flush-alertstream function.

No. of packets	CPU	GPU	using Pinned M.	both pinned and shared
for outside traffic	-	-	-	
128	12.55	186.016	181.459	176.894
256	21.038	192.058	187.795	182.046
512	42.353	206.808	198.579	192.756
1024	257.543	227.814	217.855	207.855
2048	684.572	364.449	319.759	245.198
4096	1538.129	661.036	625.249	396.426
8192	3337.146	1401.364	1346.407	866.676
16384	7042.513	2895.997	2842.002	1903.165
32768	14697.645	6079.579	6034.245	4127.815
for inside traffic	-	-	-	
128	14.75	276.75	273.085	271.177
256	27.689	288.877	285.851	277.269
512	61.982	296.471	291.046	285.282
1024	278.868	322.078	315.663	308.329
2048	694.076	431.356	420.549	349.456
4096	1559.194	748.679	733.204	488.458
8192	3419.509	1524.551	1511.205	1034.718
16384	7165.336	3046.168	3027.277	2071.222
32768	14646.219	6105.253	6093.667	4188.301

Table 7.8: Week5 Tuesday outside and inside input data test result in ms for stream5Flush- alertstream function.

No. of packets	CPU	GPU	using Pinned M.	both pinned and shared
for outside traffic	-	-	-	
1287	13.839	291.794	258.192	253.92
256	16.88	292.629	263.416	256.022
512	26.267	302.027	272.791	263.541
1024	258.28	323.282	294.022	286.263
2048	746.849	509.309	474.956	352.787
4096	1647.19	911.017	860.713	594.25
8192	3334.788	1470.917	1466.448	967.003
16384	6946.035	2932.44	2919.91	1939.223
32768	14314.31	5861.416	5825.176	3889.043
for inside traffic	-	-	-	
128	10.996	336.845	333.674	330.694
256	15.903	338.936	336.466	334.224
512	39.721	354.885	351.206	347.36
1024	267.091	387.205	381.637	374.646
2048	713.599	523.322	515.329	426.451
4096	1563.044	833.176	827.834	592.862
8192	3246.043	1419.793	1413.09	939.243
16384	6612.21	2578.898	2570.809	1593.232
32768	13452.24	4966.059	4945.502	2999.591

Table 7.9: Week5 Wednesday outside input data test result in ms for stream5Flush- alertstream function.

No. of packets	CPU	GPU	using Pinned M.	both pinned and shared
128	13.353	196.177	181.061	177.247
256	19.467	211.977	190.458	182.834
512	39.07	213.614	202.478	195.481
1024	270.397	257.394	218.914	208.79
2048	736.234	419.559	395.416	270.789
4096	1662.126	801.494	745.703	522.958
8192	3571.734	1622.933	1577.989	1117.035
16384	7241.154	3085.363	3062.602	2102.006
32768	14886.682	6293.445	6236.619	4344.195

Table 7.10: Week5 Wednesday inside input data test result in ms for stream5Flush- alertstream function.

No. of packets	CPU	GPU	using Pinned M.	both pinned and shared
128	8.241	266.966	262.993	259.344
256	24.002	279.656	269.639	266.334
512	35.177	295.541	293.948	276.106
1024	281.536	319.971	313.751	306.903
2048	766.34	481.054	478.44	372.794
4096	1652.206	817.776	813.859	575.063
8192	3301.451	1414.29	1408.663	895.516
16384	6694.71	2592.414	2577.223	1611.044
32768	13609.258	5031.766	5019.894	3061.505

Table 7.11: Week5 Thursday test result in ms for stream5Flush- alertstream function.

No. of packets	CPU	GPU	using Pinned M.	both pinned and shared
for outside traffic	-	-	-	
128	15.408	316.846	308.258	304.872
256	18.52	328.071	316.033	310.869
512	26.079	334.828	329.398	317.417
1024	251.565	356.404	351.574	344.371
2048	696.447	505.392	482.727	386.547
4096	1622.974	883.242	870.126	645.893
8192	3407.563	1640.625	1573.372	1112.962
16384	7016.728	3055.527	3032.574	2051.177
32768	14224.031	5887.772	5793.713	3913.5
for inside traffic	-	-	-	
128	9.034	424.476	418.64	416.183
256	16.389	434.754	430.002	428.548
512	24.056	443.368	439.995	433.339
1024	267.857	470.483	464.784	457.053
2048	707.638	611.591	606.547	519.69
4096	1641.799	1001.272	966.766	746.702
8192	3334.546	1579.043	1570.075	1095.642
16384	6726.681	2779.291	2772.834	1797.144
32768	13559.626	5183.075	5146.895	3236.925

Test results of preprocess Function

Table 7.12: Week4 Monday result in ms for preprocess function.

No. of packets	CPU	GPU	using twok	both twok and shared
for outside traffic	-	-	-	
128	8.584	216.243	213.033	211.925
256	13.163	223.561	216.352	213.601
512	23.018	227.23	223.123	219.875
1024	93.516	252.305	239.209	232.685
2048	424.138	422.222	393.829	368.084
4096	1017.120	804.546	790.937	764.446
8192	2055.874	1393.499	1381.421	1354.828
16384	4528.105	2882.872	2838.179	2823.157
32768	9247.055	5822.168	5761.4367	5728.292
for inside traffic	-	-	-	
128	7.567	151.804	145.416	143.675
256	13.931	157.706	151.085	147.089
512	20.459	166.968	158.553	153.561
1024	60.166	172.338	162.181	157.175
2048	305.394	254.872	239.773	226.481
4096	798.075	533.987	517.797	509.787
8192	1848.384	1106.389	1094.477	1086.002
16384	4139.002	2507.407	2476.182	2457.291
32768	8548.917	5132.051	5104.617	5075.618

Table 7.13: Week4 Tuesday test result in ms for preprocess function

No. of packets	CPU	GPU	using twok.	both twok and shared
for outside traffic	-	-	-	
128	6.778	134.587	133.617	132.733
256	12.121	137.286	135.284	134.702
512	19.316	141.442	140.239	139.234
1024	98.313	157.826	151.577	145.447
2048	342.869	266.318	251.693	243.372
4096	895.474	573.979	559.884	548.33
8192	1928.392	1218.092	1165.438	1152.692
16384	4133.517	2596.738	2467.325	2458.574
32768	9026.615	5649.248	5536.653	5529.922
for inside traffic	data not available	-	-	

Table 7.14: Week4 Wednesday test result in ms for preprocess function

No. of packets	CPU	GPU	using twok	both twok and shared
for outside traffic	-	-	-	
128	9.532	136.935	134.123	132.034
256	16.558	146.519	141.406	139.443
512	29.976	152.779	147.845	144.495
1024	95.481	178.083	167.739	158.112
2048	370.36	291.52	278.792	270.214
4096	984.619	695.297	676.379	652.126
8192	2152.31	1435.29	1419.855	1409.183
16384	4625.545	2989.146	2974.101	2963.337
32768	9979.557	6554.191	6499.422	6467.501
for inside traffic	-	-	-	
128	5.803	182.302	177.945	174.677
256	12.111	188.469	183.437	181.642
512	23.306	204.476	193.699	185.13
1024	99.01	225.571	216.03	209.655
2048	388.579	347.51	336.2070	330.52
4096	1007.872	775.306	766.369	755.146
8192	2757.869	2086.326	2070.436	2046.731
16384	5216.641	3622.007	3611.962	3598.946
32768	10422.217	7001.877	6986.17	6955.766

Table 7.15: Week4 Thursday test result in ms for preprocess function.

No. of packets	CPU	GPU	using twok	both twok and shared
for outside traffic	-	-	-	
128	8.844	250.085	247.151	245.335
256	14.885	255.093	252.882	249.467
512	21.614	263.749	257.585	254.666
1024	94.469	280.089	276.895	267.856
2048	396.621	441.699	415.39	402.725
4096	1005.015	812.775	815.994	796.073
8192	2168.206	1567.249	1525.96	1511.029
16384	4501.769	2991.228	2956.942	2941.966
32768	9121.17	5834.744	5794.0161	5759.157
for inside traffic	-	-	-	
128	6.375	345.001	337.939	332.852
256	11.183	353.937	342.901	336.609
512	19.874	359.345	348.302	344.547
1024	102.041	374.085	365.381	358.183
2048	388.596	536.47	524.613	503.573
4096	996.812	933.527	897.58	882.56
8192	2042.508	1520.823	1505.208	1488.593
16384	4143.892	2724.208	2702.564	2689.616
32768	8357.996	5121.266	5108.52	5087.465

Table 7.16: Week4 Friday outside input data test result in ms for preprocess function

No. of packets	CPU	GPU	using twok	both twok and shared
128	9.121	133.568	126.245	123.981
256	15.402	137.818	132.625	128.126
512	22.441	144.787	139.025	135.591
1024	34.958	153.812	147.345	143.915
2048	310.796	206.144	193.156	177.258
4096	827.612	502.433	487.811	461.256
8192	1781.204	1013.403	1006.869	978.680
16384	3853.433	2396.329	2360.839	2337.255
32768	8213.051	5132.207	5090.167	5065.797

Table 7.17: Week4 Friday inside dataset input test result in ms for preprocess function

No. of packets	CPU	GPU	using twok	both twok and shared
128	5.717	188.67	180.313	175.201
256	10.536	197.723	185.456	178.247
512	18.3279	196.858	188.35	185.916
1024	32.838	202.2	196.65	192.324
2048	284.397	284.183	263.892	245.268
4096	768.898	528.408	515.242	501.646
8192	1745.572	1067.478	1043.09	1027.48
16384	3773.699	2183.009	2170.051	2152.96
32768	8468.73	5072.04	5050.108	5027.841

Table 7.18: Week5 Monday test result in ms for preprocess function.

No. of packets	CPU	GPU	using twok	both twok and shared
for outside traffic	-	-	-	
128	6.875	149.587	143.75	138.52
256	12.56	163.807	154.742	148.36
512	17.267	168.413	161.028	157.653
1024	98.968	197.684	183.896	170.445
2048	342.571	291.547	275.670	240.69
4096	888.992	613.577	596.54	580.364
8192	2046.173	1337.809	1313.16	1289.84
16384	4262.5	2823.45	2809.674	2793.356
32768	9237.442	6025.063	6011.38	5992.832
for inside traffic	-	-	-	
128	10.271	224.441	221.693	217.34
256	19.468	228.265	227.674	221.911
512	35.24	235.803	236.036	230.371
1024	96.305	272.002	256.028	248.021
2048	356.954	375.478	360.838	355.336
4096	889.006	692.256	679.061	662.647
8192	2129.184	1467.506	1452.396	1444.536
16384	4589.153	3004.278	2987.091	2970.129
32768	9416.48	6063.387	6052.956	6037.13

Table 7.19: Week5 Tuesday test result in ms for preprocess.

No. of packets	CPU	GPU	using twok	both twok and shared
for outside traffic	-	-	-	
128	5.46	226.054	215.609	210.12
256	10.718	233.52	221.76	215.167
512	18.638	238.079	229.322	221.545
1024	102.834	258.654	247.196	236.45
2048	405.989	406.596	397.444	383.179
4096	1016.0349	809.638	792.396	778.993
8192	2058.708	1393.76	1380.258	1363.205
16384	4419.358	2865.288	2839.333	2819.433
32768	8954.623	5812.57	5756.12	5732.069
for inside traffic	-	-	-	
128	8.735	283.72	277.242	275.964
256	14.55	292.398	286.42	280.222
512	22.339	298.716	291.037	285.799
1024	117.697	316.008	310.068	301.647
2048	400.954	470.656	472.533	459.36
4096	923.661	782.362	776.15	760.346
8192	1962.655	1370.376	1348.423	1336.524
16384	4027.739	2527.996	2518.228	2511.358
32768	7985.71	4908.564	4875.032	4864.561

Table 7.20: Week5 Wednesday outside input data test result in ms for preprocess function.

No. of packets	CPU	GPU	using twok	both twok and shared
128	5.676	150.069	146.24	142.771
256	10.2	157.651	151.031	147.469
512	23.976	173.548	162.8010	154.521
1024	98.985	188.352	179.091	167.708
2048	408.095	359.508	351.155	340.05
4096	981.691	711.802	701.256	693.533
8192	2315.523	1596.056	1559.675	1535.002
16384	4658.869	3028.864	3021.356	3002.905
32768	9631.546	6240.893	6212.098	6193.708

Table 7.21: Week5 Wednesday inside input data test result in ms for preprocess function.

No. of packets	CPU	GPU	using twok	both pinned and shared
128	8.893	221.798	214.237	211.75
256	16.954	225.285	218.4	214.24
512	26.934	234.408	229.529	225.313
1024	97.941	258.24	252.504	248.777
2048	473.678	462.261	447.78	436.464
4096	983.745	793.394	785.503	777.664
8192	2020.485	1371.849	1355.788	1344.345
16384	4138.912	2549.893	2540.882	2527.066
32768	8147.158	4986.907	4974.043	4968.488

Table 7.22: Week5 Thursday test result in ms for preprocess function.

No. of packets	CPU	GPU	using twok	both twok and shared
for outside traffic	-	-	-	
128	5.97	255.364	250.313	245.556
256	12.984	262.44	257.617	250.224
512	17.179	269.298	263.512	257.159
1024	90.379	278.888	274.358	267.348
2048	369.658	418.507	405.502	392.223
4096	1011.176	809.831	798.539	771.108
8192	2151.55	1562.928	1538.963	1527.422
16384	4532.585	2994.708	2977.76	2962.778
32768	9116.538	6018.676	5784.069	5766.022
for inside traffic	-	-	-	
128	8.571	348.42	338.596	334.439
256	16.3729	350.82	342.729	339.282
512	23.072	357.578	346.1677	344.229
1024	95.355	373.588	369.182	364.606
2048	388.81	530.19	522.962	518.783
4096	985.526	906.321	894.35	883.816
8192	2054.164	1526.627	1510.299	1500.658
16384	4144.755	2712.545	2705.573	2698.726
32768	8374.186	5164.759	5151.631	5116.46

Table 7.23: Week5 Friday inside input data test result in ms for preprocess function.

No. of packets	CPU	GPU	using twok	both twok and shared
128	7.899	402.098	397.287	394.379
256	12.878	406.945	402.127	398.433
512	17.851	414.67	406.082	403.146
1024	33.514	419.778	415.824	411.20
2048	330.32	516.806	510.369	493.629
4096	1491.919	1461.512	1446.67	1436.468
8192	2577.481	2135.224	2127.173	2106.144
16384	4897.818	3543.53	3530.562	3498.902
32768	9788.188	6688.755	6634.571	6590.212

Table 7.24: Week4 CPU overall test result in ms.

Speed	Monday	Tuesday	Wednesday	Thursday	Friday
default	1245.38	655.96	663.52	1125.42	667.58
10Mbps	287.23 sec	213.59	227.45	450.55	183.50
1000pps	2559.16	1309.64	1315.65	2308.89	1320.38
Top	35.29	25.15	27.14	53.25	24.86
Inside traffic	-	-	-	-	
Default	542.96	–	683.47	1263.16	646.35
10Mbps	196.64	–	282.29	590.28	256.28
1000pps	1218.95	–	1946.23	2975.56	2138.15
Top	0.75	–	0.83	1.52	0.94

Table 7.25: Week5 CPU overall test result in ms.

Speed	Monday	Tuesday	Wednesday	Thursday	Friday
default	612.3	694.84	625.67	985.34	–
10Mbps	245.03	286.93	247.80	449.91	–
1000pps	1845.68	2158.74	1893.25	2318.56	–
Top	0.65	1.27	0.67	1.14	–
Inside traffic	-	-	-	-	
Default	876.85	914.25	892.72	1289.47	1453.89
10Mbps	334.90	356.45	348.29	590.23	791.29
1000pps	2612.58	2842.74	2789.56	3201.61	3394.30
Top	1.09	1.61	1.02	1.56	1.66

Table 7.26: Week4 GPU overall test result in ms.

Speed	Monday	Tuesday	Wednesday	Thursday	Friday
outside traffic	-	-	-	-	
default	1238.27	652.58	658.32	1118.62	664.45
10Mbps	284.01	210.14	224.13	446.89	181.58
1000pps	2544.98	1299.59	1308.99	2297.08	1314.14
Top	32.16	22.58	24.08	48.41	22.90
Inside traffic	-	-	-	-	
Default	540.86	–	681.02	1260.23	644.31
10Mbps	193.86	–	279.07	586.68	253.18
1000pps	1216.89	–	1943.94	2966.37	2131.28
Top	0.68	–	0.74	1.36	0.83

Table 7.27: Week5 GPU overall test result in ms.

Speed	Monday	Tuesday	Wednesday	Thursday	Friday
outside traffic	-	-	-	-	
default	606.67	691.58	622.64	980.21	–
10Mbps	241.28	281.47	244.05	441.56	–
1000pps	1840.27	2154.92	1888.26	2314.58	–
Top	0.58	1.1	0.60	1.02	–
Inside traffic	-	-	-	-	
Default	873.84	911.26	889.92	1286.78	1450.08
10Mbps	330.35	352.08	344.58	585.987	785.42
1000pps	2608.12	2836.78	2778.43	3192.07	3382.54
Top	0.98	1.44	0.90	1.38	1.48