GPU Accelerated NIDS Search

Kristian Nordhaug

Master's Thesis
Master of Science in Information Security
30 ECTS
Department of Computer Science and Media Technology
Gjøvik University College, 2012

Avdeling for
informatikk og medieteknikk
Høgskolen i Gjøvik
Postboks 191
2802 Gjøvik

Department of Computer Science
and Media Technology
Gjøvik University College
Box 191
N-2802 Gjøvik
Norway

GPU Accelerated NIDS Search

Kristian Nordhaug

2012/07/01

Abstract

Network Intrusion Detection System (NIDS) analyzes network traffic for malicious activities and report's findings from events that intend to compromise the security of the computers and other equipment. NIDS looks into both headers and payloads of the network packets to identify possible intrusions.

NIDS models that only use Central Processing Units (CPU) such as the IDS Snort, have in the last decade struggled with the CPU as the bottleneck of the system. Network traffic has been increasing more rapidly than the clock-speed of CPUs. The CPUs have gained more cores, but lack implementation for utilizing multi-core CPUs and are unable to cope with the bandwidth throughput we are starting to see in high-tech network infrastructure that they are set to protect. 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, increasing the false-negative rate. 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*. This thesis will focus on accelerating the NIDS by parallelizing this algorithm.

In the recent years modern GPUs have evolved from being a tool that only displays high-end graphics for games, to be used for general-purpose scientific and engineering computing across a range of platforms [35]. GPU computing is the short term used when ordering the GPU to take over and accelerate the computationally-intensive calculations normally done by the CPU, and instead let the CPU take care of the more sequential parts of the application. They then work together solving tasks in a heterogeneous co-processing computing model. Using Graphics Processing Units (GPU) for general-purpose scientific and engineering computing has grown exponentially the last few years. This has happened mostly from the work Nvidia has put into their CUDA platform and programming model. Some of the most common areas for use of GPU is fluid dynamics, seismic processing, molecular dynamics, computational chemistry, finance and supercomputing. Programs need to be specifically designed to run optimized on a GPU, and special programming APIs have been designed explicitly for GPU computing. The most well known ones are CUDA and OpenCL. In the recent year's modern GPUs have evolved from being the tool that displays high-end graphics for games, to be the tool used in general-purpose scientific and engineering computing across a range of platforms.

The goal of this project was to harness the power within GPUs and use it to accelerate NIDS such as Snort, by using CUDA technology. Several papers have been published on the topic of GPU acceleration, however only a handful of them targeted NIDS with varying results. We believe this can be improved dramatically by further research in how different hardware components interact and how to exploit the components and their APIs in new ways for creating high-performance algorithm solutions.

We present our implementations of known string search algorithms programmed in C++ and CUDA, with analysis of these algorithms and conclude with contributions from our experiments

and theoretical analysis.

Sammendrag

NIDS analyserer nettverkstrafikk for ondsinnede aktiviteter og rapporterer funn fra hendelser som har til hensikt å kompromittere sikkerheten til datamaskiner og annet utstyr. En NIDS ser inn både header og payload av nettverkspakker og identifiserer mulige inntrengere i systemer.

NIDS modeller som kun benytter hovedprosessoren (CPU) som produktet Snort, har i det siste tiåret kjempet med CPU som flaskehalsen i systemet. Nettverks trafikken har økt raskere enn klokke-hastigheten til CPU. Selv om CPU har fått flere kjerner, mangler det implementering for å utnytte multi-core prosessorer i NIDS De klarer ikke lenger å takle mengden av trafikk gjennomstrømming vi ser i high-tech nettverksinfrastruktur der de er satt for å beskytte. De massive strømmene av datapakker overbelaster NIDS, som gjør at de krysser forbi uten å bli sjekket for malware og innbruddsforsøk. Den viktigste årsaken til dette er modulen for *inspeksjon av nettverk pakker* i deteksjon motoren på NIDS. Deteksjons motoren består av mange funksjoner og inneholder en algoritme for *streng søk*. Denne oppgaven vil fokusere på å aksellerere NIDS ved å parallelisere denne algoritmen.

GPU databehandling er slang uttrykket for å bruke GPU til å overta og aksellerere intensive beregninger, normalt gjort av CPU. CPU vil istedet få frihet til å ta seg av den mer sekvensielle delen av søknaden. De arbeider deretter sammen for å løse oppgavene i en heterogen co-prosessor databehandlings modell. Bruken av Graphical Processing Units (GPU) for generell vitenskapelig og teknisk databehandling har vokst eksponentielt de siste årene. Dette har skjedd hovedsakelig på grunn av arbeidet Nvidia har lagt inn i sin CUDA plattform og programmerings modell. Noen av de vanligste områdene for bruk av GPU er veskedynamikk, seismisk prosessering, molekylær dynamikk, beregningsorientert kjemi, økonomi og supercomputing. Programmer må være spesielt designet for å kjøre optimalisert på en GPU, og spesielle programmerings APIer har blitt designet eksplisitt for GPU databehandling. De mest kjente er CUDA og OpenCL. I de siste årene har moderne GPUer utviklet seg fra å være verktøyet som viser high-end grafikk for spill, til å være et verktøy som brukes i generell vitenskapelig og teknisk databehandling på en rekke plattformer.

Målet med dette prosjektet er å utnytte kraften i GPU og bruke den til å akselerere NIDS som Snort, ved hjelp av CUDA-teknologi. Flere artikler er publisert om temaet GPU-akselerasjon, men bare en håndfull målrettet mot NIDS, med varierende resultater. Vi tror dette kan forbedres dramatisk ved videre forskning på hvordan ulike maskinvarekomponenter samhandler, og hvordan man best utnytter komponentene og deres APIer på nye måter for å skape nye høyder for ytelse i algoritmiske løsninger.

Vi presenterer våre implementasjoner av kjente streng søk algoritmer, programmert i C++ og CUDA, samt analyse av disse algoritmene og konluderer med bidrag fra våre eksperimenter og teroretiske analyse.

Contents

Abstract . . . . . **iii**
Sammendrag . . . . . **v**
Contents . . . . . **vii**
List of Figures . . . . . **ix**
List of Tables . . . . . **xi**
Preface . . . . . **xiii**
1 Introduction **1**
1.1 Problem Statement 1
1.2 Contributions 3
1.3 Choice of Methods 3
1.4 Outline 4
2 State of the Art **5**
2.1 Related work 5
2.1.1 Summary 7
2.2 Technical Background 7
2.2.1 Graphical Processing Units 7
2.2.2 CUDA 8
2.2.3 NIDS 14
2.2.4 Algorithms 19
3 Implementations & Results **31**
3.1 Scope 31
3.2 Limitations 31
3.3 Lab Environment 32
3.4 Experiment #1 35
3.4.1 Execution 37
3.4.2 Analysis 38
3.5 Experiment #2 39
3.5.1 Knuth-Morris-Pratt (KMP) 39
3.5.2 Aho-Corasick (AC) 43
4 Conclusions **51**
4.1 Research Questions 51
4.1.1 To what extent can NIDS performance be increased by using GPU? 51
4.1.2 What parts of the NIDS can be optimized? 52
4.1.3 Which program specific factors give the increase, or decrease in performance? 52
4.2 Summary of Contributions 52

4.3 Future Research 53
Bibliography **55**
A Appendix **61**
A.1 Experiment #1:Naïve String Search 61
A.1.1 Naïve Search Engine 61
A.1.2 Naïve Search Kernel 64
A.1.3 Naïve Search CPU Implementation 66
A.2 Experiment #2: Knuth-Morris-Phatt (KMP) 68
A.2.1 KMP Engine 68
A.2.2 KMP Preprocessing 71
A.2.3 KMP Kernel 72
A.3 Experiment #2: Aho-Corasick (AC) 75
A.3.1 Aho-Corasick Engine 75
A.3.2 Aho-Corasick Kernel 78
A.3.3 Aho-Corasick Tree Structure 80
A.3.4 Aho-Corasick CPU Implementation 85
A.4 Experiment - Common Functions 89
A.4.1 Dictionary Structure 89
A.4.2 Output Handler 92
A.4.3 Packet Handler 94
A.4.4 Monitoring 95

List of Figures

1 GPU vs CPU - Floating point operations (FLOPs) [13] 2
2 Gainward GeForce GTX 580 1536MB GDDR5 Memory ([18]) 8
3 2D layout of threads and blocks on a GPU [46]. 10
4 Multiple Nvidia GPUs 14
5 Basic data flow of a Snort NIDS 17
6 Runmode for a pcap device in Suricata [43] 18
7 Knuth-Morris-Pratt - Algorithm Walkthrough [7] 21
8 Knuth-Morris-Pratt - Example Execution [7] 22
9 Boyer-Moore - Last Function [6] 23
10 Boyer-Moore - Example Execution [6] 23
11 Aho-Corasick - goto function [44] 25
12 Aho-Corasick - failure function [44] 25
13 Aho-Corasick - Output of Next Move Function δ 27
14 Detection Engine using GPU 33
15 Naïve GPU Execution, 456 patterns checked vs. packets with length of 40 chars. . 38
16 KMP GPU execution, 456 patterns checked vs. packets with length of 40 chars. . . 41
17 KMP CPU Implementation, 456 patterns checked vs. packets with length of 40 chars. 42
18 Aho Corasick Implementation. Red and blue areas are run once to limit the computation needed, green is looping over new packets placed into the malloced memory space on the GPU for each new collection of packets. 44
19 Occupancy at 1024 kernels per block. 45
20 Occupancy at 512 kernels per block. 46
21 Aho-Corasick CPU Execution, 456 patterns checked vs. packet length of 80chars. . 47
22 Aho-Corasick GPU Execution, 35500 patterns checked vs. packet length of 80chars. 48

List of Tables

1 Benchmark results of a Nvidia GF100 chip [16]. 11
2 Bandwidth results between the Main Memory(host), GeForce 580 GTX and GeForce 280GTX, (Bandwidth tester [38]) 11
3 Naïve string search 20
4 Knuth-Morris-Pratt - Failure Function 21
5 Lab Environment Hardware 32
6 Required software for programming 34
7 Hardware Details: GeForce GTX 580 34
8 Hardware Details: GeForce GTX 460 35
9 Average execution runtime of the Naïve string search on GPU 37
10 KMP execution, 456 patterns checked vs. packets with length of 40 chars. 40
11 GPU Aho Corasick. 456 signatures checked vs. packets with length of 80 chars. . . 46
12 GPU Aho Corasick. 35500 signatures checked vs. packets with length of 80 chars. 46
13 GPU Aho Corasick. Multiple runs of 35500 patterns checked vs. packet length of 40chars. 48

Preface

This master's thesis report conclude the end of my Master of Science in Information Security program at Høgskolen i Gjøvik (HIG) 2012. I would first like to thank my supervisor Prof. Slobodan Petrović for all help during the thesis and overall time here at Gjøvik. Petrović's lecture in Intrusion Detection and Prevention was what inspired me to write this thesis. Moreover, i would like to thank the student opponent; David Ormbakken Henriksen for feedback on the report. Finally i would like to thank Svein Engen, Benjamin Adolphi and Andreas Tellefsen for valuable feedback throughout this process.

1 Introduction

Network Intrusion Detection System models that only use Central Processing Units (CPU) such as Snort, have in the last decade struggled with the CPU as the bottleneck of the system. Network traffic has been increasing more rapidly than the clock-speed of CPUs. The CPUs have gained more cores, but lack implementation for utilizing multi-core CPUs and are unable to cope with the bandwidth throughput we are starting to see in high-tech network infrastructure that they are set to protect. 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, increasing the false-negative rate. 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*. This thesis will focus on accelerating the NIDS by parallelizing this algorithm.

Using Graphics Processing Units (GPU) for general-purpose scientific and engineering computing has grown exponentially the last few years. Ordering the GPU to take over and accelerate the computationally-intensive calculations normally done by the CPU, the two work together solving tasks in a heterogeneous co-processing computing model. Some of the most common areas for use of GPU is fluid dynamics, seismic processing, molecular dynamics, computational chemistry, finance and supercomputing. Programs need to be specifically designed to run optimized on a GPU, and special programming APIs have been designed explicitly for GPU computing. The most well known ones are CUDA and OpenCL. In the recent year's modern GPUs have evolved from being the tool that displays high-end graphics for games, to be the tool used in general-purpose scientific and engineering computing across a range of platforms.

1.1 Problem Statement

NIDS models that use Central Processing Units (CPU) for computation have in the last decade struggled more and more with the CPU becoming the bottleneck of the system. Network traffic has been increasing more rapidly than the clock-speed of CPUs making it harder and harder to keep up inspection of all network traffic that they are set to protect. The CPUs have gained more cores, but NIDS such as Snort [52] lack implementation for utilizing multi-core CPUs and are unable to cope with the bandwidth throughput. The massive flow of data packets overloads the NIDS and forces the NIDS to drop packets to keep up the real-time surveillance, letting them pass by unchecked for malware and intrusion attempts further increasing the total false-negative rate. A good GPU implementation can dramatically increase the amount of calculations that can be performed at the same time and boost performance of the NIDS, reducing false-negative rate caused by dropping packets. When gaining more computational power the system can handle more traffic without affecting performance of the NIDS, and the cost of the NIDS can be reduced by many factors as the hardware requirement is less to perform equally to the standard CPU based model. Figure 1 show a graph displaying the evolution of GPU and CPU the last decade in

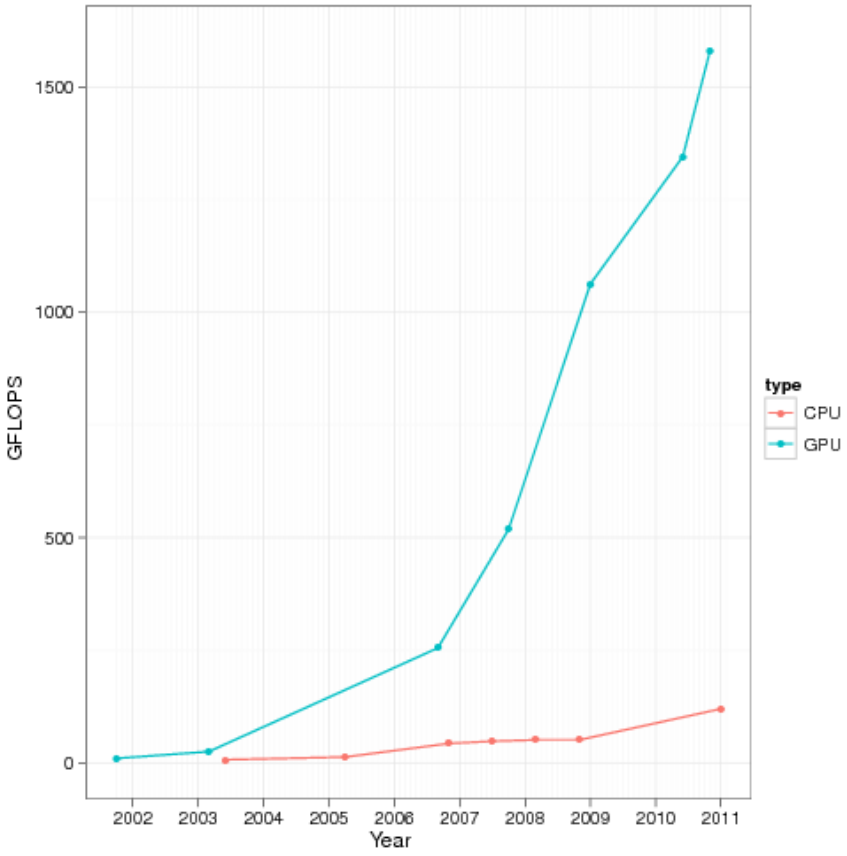


Figure 1: GPU vs CPU - Floating point operations (FLOPs) [13]

the case of floating point operations (FLOPs) per second possible.

There are two main types of NIDS, signature-based, and anomaly-based. This thesis focus on signature-based systems that monitor network traffic using known attack patterns. NIDS analyze network traffic for malicious activities and reports findings from events that intend to compromise the security of the computers and other equipment. A NIDS looks into both headers and payloads of the network packets to identify possible intrusion. This is known as deep packet inspection (DPI). Most Signature based NIDS consist of different modules; from Network Traffic, Packet Capture, Packet Decoding, Preprocessors, Detection Engine, to Output-Logging and Alerting, all highly customizable to the user's needs [34]. In most of these modules there are components consuming huge amounts of computational power, especially Preprocessors and the Detection Engine. Cabrera et al. (2004) [9] reported that signature matching in Snort accounts for over 60% of the total processing time alone. Since then the ammount of signatures active in Snort has quadrupled, making it an even more demanding process.

1.2 Contributions

In this master thesis, we examine the viability of using Graphical Processing Units (GPU) in a Network Intrusion Detection System (NIDS) to accelerate the detection engine process with CUDA parallelization technology from Nvidia. We will show that utilizing GPUs for taking over the signature matching we can theoretically increase the overall performance of NIDS, reducing the chance of dropping packets, increasing false negative rate, from too high packet throughput on the network.

The thesis contribute with answers to the following research questions:

1. To what extent can NIDS performance be increased by using GPU?
2. What parts of the NIDS can be optimized?
3. Which program specific factors give the increase, or decrease in performance?

1.3 Choice of Methods

In addition to theoretical analysis, the experimental part of this thesis is of decisive importance, thus the refinements of the experimental model will show how efficient the GPU implementation of the NIDS search algorithm is compared to its original implementation. For this, a qualitative and experimental methodology [27] will be used to answer the research questions. Learning the tools and programming languages to work with GPUs and CUDA, and technologies that are found in NIDS will take a large portion of time spent on the master thesis. A deep dive into the core of NIDS is needed to find the best abilities and weaknesses.

Addressing the first research question, *Can NIDS performance be increased by using GPU?* a careful review of literature and documentation of current NIDS systems and GPU technology is necessary, to understand exactly which parts of the pipeline is suited for parallelization. We believe that the best way to gain this knowledge is to experiment with the technologies involved and find more optimal solutions to problems encountered in a normal NIDS. Understanding how it everything fit together, the technologies behind and the architectural weaknesses in NIDS will be needed to start the experimental creation of a GPU based implementation to answer the

research questions.

The answer to the second research question *What parts of the NIDS can be optimized?* will be based on the gathered knowledge and be subject of even deeper analysis while conducting experiments to determine to the full extent of which factors that play a part in performance, both positive and negative directions. Experiments will be performed in a lab environment, set up with the latest technology possible to take advantage of all the enhancements made from previous versions of CUDA and compute power to get as accurate results as possible.

Finally, the third research question *Which program specific factors give the increase, or decrease in performance?* will be highlights from our experimentation and programming and be a valuable collection of tips and tricks as to how we found the different systems to work in synergy gaining performance boosts, and performance loss from different methods and techniques used to archive a certain function in our code. These experiences will be part of our contribution to whomever would continue the work for implementing GPU acceleration into an NIDS.

By offloading the work from the CPU to the GPU, the systems can perform massive amounts of parallel calculations and gain high performance boosts to reduce or completely eliminate packet loss. To be able to create algorithms for the GPU, experiments need to be performed to determine what data should be processed on the GPU rather then CPU as the GPU will only give performance boots at tasks that can be done in parallel. Sequential processing would still be done far better by the CPU.

Theoretical problems of the project will be to look at algorithms, how these will need to be rewritten to fit GPU architectures, optimization techniques for the GPU and look for direct access methods towards other hardware, especially the main memory and the network interface itself.

Technical side of the project will be to implement a prototype / proof-of-concept software with the GPU accelerated IDS algorithms utilizing the researched optimization methods.

Finally, methods for benchmarking the performance of NIDS will be applied and a comparison carried out between the results of the project and the state-of-the-art open source systems.

1.4 Outline

In Chapter 2 we give details of related work in regards to signature-based NIDS that utilize content or regular expressions. Next we provide a technical background to the Graphical Processing Units, CUDA technology and NIDS architectures. Chapter 3 shows our design and implementation of string matching algorithms using CUDA and evaluates the results from our prototype implementation compared to related work. Chapter 4 concludes our findings and work.

2 State of the Art

This chapter presents the most related work from the perspective of this thesis, then continues with a background overview of the technologies used; Graphical Processing Units, the CUDA architecture, and Network Intrusion Detection Systems.

2.1 Related work

In normal signature-based NIDS systems the most crucial operation performed is the pattern matching. Pattern matching is the technique of searching through a string to find a specific pattern given as binary data or a set of characters. It is used in all fields of study and technologies, from indexing libraries, search engines on the Internet such as Google, or locating a word in a text file. These search tools can often let the user customize the criteria of the search in different ways, from searching for a particular character, one word or many, titles or content, or based on a given length. This flexibility of search dates back to the 1950's with S.T. Kleene's [45] formalism of regular sets, known in computing as *regular expressions*.

In the art of pattern matching two main classifications exist, single- and multi pattern algorithms. Single pattern matching algorithms will search through a text for each known pattern individually, which means it will loop through the whole text as many times as the total number of patterns. Original examples of single pattern algorithms are the Knuth-Morris-Pratt(MKP) [26] and Boyer-Moore(BM) [8]. Multi-pattern string algorithms operate quite differently, searching through the known text only once, looking for all patterns at the same time. This is done by preprocessing the patterns into a Final State Machine(FSM) or finite automata, that is further used to conduct the actual matching with the text. For multi-pattern algorithms we use two main types of automata, Non-Deterministic Finite Automaton (NFA) and Deterministic Finite Automaton (DFA) (DFA and regular expressions can be found in [50]). An NFA will generate a state for each possible input symbol, leading to many possible next states. The DFA on the other hand will only have one possible next state for each symbol, following the path of transition. It will jump deterministically from one state to another. Downside with DFA is the huge amount of memory needed to generate a next state for all symbols used. Examples of multi-pattern string matching algorithms include Aho-Corasick [1], Wu-Manber [61] and Commentz-Walter [11] (all algorithms can also be found in [33]). Most NIDS that are signature-based use a type of finite automata combined with regular expressions as described for their pattern matching.

Many approaches have attempted to take parts of NIDS and split them into elements for basic multi-threading parallelism realized by normal CPU multi-core processors [20, 21, 47, 48, 54]. Attempts at accelerating NIDS through special hardware other then the CPU have also been made for years. Commercial NIDS systems have for example been using technologies like Application-Specific Integrated Circuits (ASIC) or Field-Programmable Gate Arrays(FPGA), chips designed

and programmed solely to run a single algorithm or a small system. The use of Ternary Content-addressable Memory (TCAM) have also been done by Yu et al. [62] and Meiners et al. [30]. Both where quite fast, but in hardware extremely expensive. Chip circuits (FPGAs) also have the downside that when changing a rule or adding a new rule set, one must program a whole new circuit and then recompile the whole automaton, a very time consuming and difficult task. Improved approaches to FPGAs include Sidhu and Prasannas implementation of regular expressions on FPGAs (2001) [49], Baker et al. with a signature based method (2004) [4], and later Dharmapurikar et al. with bloom filters(2004) [14] and Kennedy et al. on a Statix 3 FPGA (2010) [23]. Attig et al. also proposed improvement to FPGAs with payload content scanning and header processing (2005) [2].

Several research efforts have been attempted at developing adoptions of *string matching algorithms* for Graphical Processing Units (GPU):

Huang et al.(2008) [19] attempted intrusion detection on a GPU based on the Wu-Manber algorithm [61]. Smith et al.(2009) [50] created an extended DFA with regular expressions on a GPU for intrusion detection and evaluated GPUs for network packet signature matching. Marziale et al.(2007) [29] use GPU for accelerating in-place file carving tools. Zha et al.(2011) [63] implemented a CUDA algorithm based on Aho-Corasick [1] with a Tesla GT200 card.

Jacob and Broadly attempted to accelerate Snort on a GPU with their *PixelSnort* (2006) [22]. They focused on offloading the string matching algorithm from the CPU to the GPU with a simplified single-pattern KMP algorithm [26]. This was at an early stage, not using new technology such as CUDA or OpenCL. They programmed the basic fragment shader in OpenGL to perform the calculations like normal visual graphics programming. Pixelsnort did not achieve any speed-up under normal conditions.

Related work combining CUDA and NIDS has been done by Vasiliadis et al. in multiple research papers: [55] was an attempt to gain high performance in NIDS (Snort) using Graphics Processors. They observed that single pattern matching algorithms like KMP were not suited for GPU computing, and rather tried using multi-pattern matching algorithms. The authors chose to port the algorithm Aho-Corasick [1] (Snort's standard detection engine algorithm), and concluded that it could boost performance of Snort at the time by a factor of two. The single-threaded architecture of Snort restricted the scalability of their work, not managing to utilize the power of multi-core CPU. The authors tried different approaches to start kernels on the GPU; assigning a single packet to each multiprocessor, assigning a single packet to each stream processor, and splitting up the packet into chunks to utilize the warp with size of their current GPU.

In [56] the team created a regular expression matching engine tailored to IDS use, and combined it with the work from Gnort. This experiment claimed a 60% increase in the overall packet processing throughput in Snort. However, the solution used only one GPU. The team realized the problems within Snort's single-thread architecture and wanted to further research Snort's performance when running multiple executions at the same time, one on each core.

In [58] the authors presented a pattern matching library for GPU. The library supports both

string searching and regular expression matching on the CUDA architecture. In addition the paper discussed the performance impact of different types of memory hierarchies and presented possible solutions for memory congestion problems.

Later Vasiliadis and Ioannidis published [51] modified the open source antivirus ClamAV from Sourcefire with their own GPU accelerated engine. They used the same methods and ideas from prior work [55, 56, 58].

The latest published contribution from the team combining all previous work [57] the authors propose a new framework model for NIDS, combining commodity hardware, general purpose hardware components, in a single-node design for high-performance network traffic analysis. The main idea is that no component needs to be synchronized, or wait for other executions to gain access to resources.

Chen-Hsiung Liu et al. created the PFAC Library [10]; string matching on the GPU using the Aho-Corasick algorithm. The program is written such that for every input byte a thread on the GPU is started, and the instance of the algorithm will run from that location until no match is found against a pattern. The PFAC library does not support multi-GPU but the authors claim it can easily be combined with other threading libraries to perform string matching on multiple GPUs at once.

2.1.1 Summary

The thesis will most importantly focus on [55] by Vasiliadis et al. titled Gnort: High Performance Network Intrusion Detection Using Graphics Processors. Results presented in Chapter 3 will be discussed and analyzed against this paper.

2.2 Technical Background

This section will give overview of technology used throughout the thesis. This includes Graphics Processing Units (GPUs), CUDA programming, Network Intrusion Detection Systems (NIDS) and well known algorithms for use in pattern matching.

2.2.1 Graphical Processing Units

The constant global demand for high-definition graphics in 3D games and realtime visualization for applications have lead to massive investments in developing Graphic Processing Units (GPU) for harnessing computational power in an extremely efficient parallel processing computational model. The GPUs have grown almost exponentially in processing power and memory bandwidth.

In this thesis we have chosen to work with the CUDA architecture. CUDA is Nvidia's standard parallel architecture and is exclusively integrated into nearly every Nvidia graphics card today. API and compiler support is available for C, C++, Fortran, Matlab and more. It gives direct access to the hardware without the need of other APIs [41]. CUDA offers the most optimal platform for GPU computation with fast calls as it is tailored specifically for the hardware, also giving fast Direct Memory Access (DMA) and other functions that would be needed for a real-time NIDS system. We believe CUDA is the most promising framework for further research and development due to the quality of the API, development kit, documentation, free on-line education and training, books and example programs.

The first GPUs were produced by Nvidia in 1999, designed as graphics accelerators to

handle all visuals that are to be seen on the screen of the computer and only supporting some specific functions [35]. Using these GPUs for scientific work was at this time hard, as the only programming language available was OpenGL [25]. Scientists had to take the calculations they wanted to perform and map them to problems with triangles and polygons that could be represented visually on the screen [41]. Newer cards have gained more processors with higher frequency, higher number of CUDA cores, increased memory and API support. Figure: 2, shows an image of a high-end Nvidia GeForce graphics card (GTX 580), also used during the experiments in this thesis.

Figure 2: Gainward GeForce GTX 580 1536MB GDDR5 Memory ([18])

2.2.2 CUDA

Nvidia supplies CUDA developers with an up to date programming guide [37] with the release of each CUDA revision as well as a best practice guide for CUDA C development[36]. We strongly recommend anyone who wish to gain more insight in CUDA to read these documents found in the CUDA-Toolkit documentation: [38]. Books with best practices examples for CUDA has also been released.

Nvidia provides three types of cards that all support CUDA: Tesla, Quadro and GeForce, respectively [39].

Tesla is the scientific GPU card, designed for data centers and workstation computing applications. It is based on the “Fermi” GPU computing architecture, and can deliver massive application performance. Tesla have full double precision on floating numbers, faster PCIe communication and large data set support. It also has Nvidia GPUDirect with InfiniBand, used by the major “supercomputers” around the world. It also has two DMA engines for dual access to main memory. Special drivers for Windows also exist for Tesla, reducing CUDA kernel overhead. Tesla also supports Error Correcting Code (ECC) Memory, for critical applications with uncompromised computing accuracy and reliability, where it protects register files, L1/L2 caches, shared memory and DRAM.

Quadro is the professional *graphics* solution and is created to process the max possible triangles per second possible. Quadro is also based on the Fermi architecture and has two DMA

engines, but not the other functions that are unique to Tesla. Quadro is often used by digital content creators such as in the movie industry, design engineers, geo-scientists and more.

GeForce is the normal consumer version, much cheaper than the other two, though almost the same graphics power as the Quadro and Tesla, but only one built in DMA engine and none of the other features. This is the type that is referred to as a "gamer" card, found in laptops and normal desktop workstations.

Interface

CUDA consists of a set of extensions to the normal C language and a runtime library. Source files containing CUDA-code must therefore be compiled with Nvidia's own compiler: **nvcc**. The runtime is built from the CUDA driver API which is also accessible to developers. A program can use the runtime library or driver API, or both at once without problems. *nvcc* is accessable from command line and is supported in all newer versions of the biggest operating systems, namely Windows(7,Vista and XP), Mac OS X, and Linux. Linux has supported Toolkit versions for Fedora, Ubuntu, RedHat, OpenSuse and SUSE Enterprise Server, but other distributions such as Arch Linux work as well.

New revisions of CUDA have a *compute capability* flag within a given GPU, and is 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 Fermi architecture and has compute capability 2.x. Current features are detailed in [38].

Compute capability is set as an option flag. Example: arch=compute_20, -code=sm_20 will set the compiler to compute capability 2.0. Nvidia has also added macros to be used within code to switch between different code-paths based on compute capabilities by calling for example:_ _CUDA _ ARCH _ _.

Parallelization

The GPU core is organized in a "Single Instruction Multiple Data-architecture(SIMD). The idea is to execute multiple copies of a program in parallel with only one single call to the GPU. In CUDA, the executed code sent to the GPU is called a *kernel*. An example is provided where we add two vectors A and B and store the result in C: Listing 2.1. The code looks and feels like normal C-code. However, there are identifiers that separate host-code from device-code, here represented with __global __. The second important part of this example is how the function VecAdd is called from within main(). The difference from a normal C function call is "«<B, T»>" which is an execution configuration syntax that tells the GPU to launch B-number of *blocks*, containing T-number of *threads*. In the documentation of CUDA the word *warp* can often be seen. This describes the minimum group size of threads that can be processed in SIMD fashion for maximum performance. However, we do not work with warps directly, this is rather the purpose of blocks. A block is a collection of these threads and can share memory and work together. Blocks are again clustered in a *grid* pattern, making it relatively easy to access any specific thread within the program.

```
1 // Kernel definition
2 \_\_global\_\_ void VecAdd(float* A, float* B, float* C)
```

```
3  {
4      int i = threadIdx.x;
5      C[i] = A[i] + B[i];
6  }
7
8  int main()
9  {
10     ...
11     // Kernel invocation of BB blocks with TT threads.
12     VecAdd<<<BB, TT>>>(A, B, C);
13 }
```

Listing 2.1: Example of launching a kernel in CUDA with C code [38]

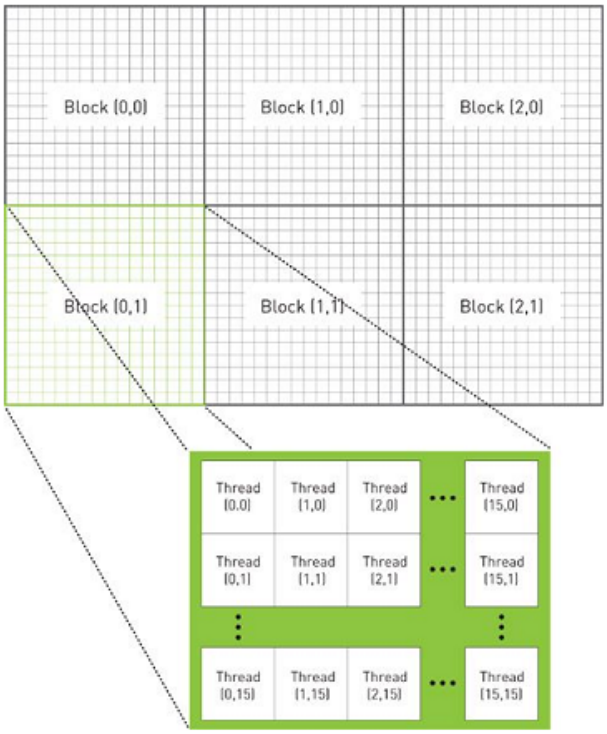


Figure 3: 2D layout of threads and blocks on a GPU [46].

Memory Access

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. Note some General-Purpose computing on Graphical Processing Units (GPGPU devices) such as laptops have an integrated graphics chip on the motherboard, in addition to a second more powerful GPU chip. These share the main memory of the laptop seamlessly, but this does not change the programming model.

On CUDA GPUs contain two main types of memory storage, on-chip and on-board. The on-chip memory is by far the fastest, however limited in size, merely some kilobytes(KB) storage. These memory spaces are known as the register, shared memory and texture memory. They are used effectively when the programmer wants to reuse data multiple times very quickly. The on-board memory is the largest memory storage on the GPU card, measured in gigabytes(GB), known as global memory. Compared to on-chip, global memory is rather slow, and is used for storing data over time while executing programs. Table 1 contains benchmark results of Nvidia's GF100 chip memory access speed, found in cards such as GTX 470 and 480.

Memory	Speed	Location	Cached	Access	Scope
Register memory	$\approx 8000GB/s$	on-chip	No	Read/Write	One thread
Shared memory	$\approx 1600GB/s$	on-chip	N/A	Read/Write	All threads in a block
Global memory	$\approx 177GB/s$	on-board	Yes	Read/Write	All threads + host
Mapped memory	$\approx 8GB/s$	host	Yes	Read/Write	All thread + host

Table 1: Benchmark results of a Nvidia GF100 chip [16].

Device	Memory transfer	Speed
580 GTX	$Host \rightarrow Device$	3889.6 MB/s
580 GTX	$Device \rightarrow Host$	4323.7 MB/s
580 GTX to 280GTX	$Device \rightarrow Device$	163636.7 MB/s

Table 2: Bandwidth results between the Main Memory(host), GeForce 580 GTX and GeForce 280GTX, (Bandwidth tester [38])

Memory Space

Experienced programmers like to call these memory spaces for variable type qualifiers. These qualifiers specify in which device memory a variable will be stored. 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: _ _shared _ _, _ _constant _ _or _ _global _ _.

Device

qualifiers will declare that the variable will be sent to the GPU device. _ _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. Each of the qualifiers have performance and usability consequences that will greatly affect the running program if not used correctly:

Global

memory (_ _global _ _qualifier) is located on-board also known as DRAM and is the biggest memory storage on the GPU card. 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. (Details regarding performance of global memory see [36]).

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 makes it ideal for programs that need cooperation between some threads, without the need of synchronizing every thread on the GPU. This memory is much faster then global memory, and should by all means be used whenever possible where many threads need access to the same memory.

Constant

memory (_ _constant _ _qualifier) has a size of 64KB that also holds an 8KB cache. 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, before kernel is launched. All threads on the GPU have access to constant memory.

Registers

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

Texture

is a wrapper around the global memory already mentioned. Texture memory is read only on the device and can only be set from the host. It is however relatively small, only an 8KB cache. 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.

Page-Locked Host

memory, also called *pinned* memory is located on the host memory and not on the device. The allocated memory will not be swapped to disk by the OS, but remains at the same location, with the same address space for the whole duration of the program.

Performance Metrics

Measuring performance accurately is important when we want to optimize CUDA. [36] lists the most optimal methods:

The *runtime of kernels* can be measured using timers from either CPU or the GPU. There are however many pitfalls and advantages to using either one. The GPU operates asynchronously when calling kernels and memory operations also work asynchronously when set. The CPU has to be synchronized through the CUDA API to be able to measure correctly. This blocks the calling CPU thread until all kernels are completed. This approach is therefore only usable when measuring the overall performance, and not on a particular kernel or stream [36].

Using GPU timers is carried out by recording timestamps of events on the GPU itself, and is therefore OS independent. This is fully supported by the CUDA API and is quite intuitive to use. All parts of the code can be measured by calling *cudaEventRecord()* and give a *start* or *stop* parameter. The *cudaEventElapsedTime()* will then give the duration of the event. The GPU timer is measured in milliseconds, and has around half a millisecond resolution [36].

The most important part to measure is the bandwidth of data transfers. This can be done either as theoretical or as effective bandwidth calculation. The structure of the data itself and GPU memory used will have the largest impact on performance. Theoretical max bandwidth is hardware dependent and found in the product specifications. The GPU used for this thesis was a GeForce 580 GTX, with DDR5 RAM, with a clock rate of 2.01GHz and 384-bit bus width. Theoretical max performance is therefore 192.096 GB/sec: $(2001 * 10^6 * (384/8) * 2)/10^9 = 192.096 GB/sec$

Effective bandwidth is found after timing the events and activities and calculating the data accesses of the program. This can be done for each array, matrix or whatever is needed. The math behind effective bandwidth is straightforward: basically, the *total amount of data bits read per time unit*, here in seconds.

Bandwidth can also be analyzed in the CUDA Visual Profiler *cudaprof*. The tool can display useful statistics such as how well the code is performing compared to the hardware limit of the system.

Multi-GPU programming

Using multiple GPUs in programming does not automatically require much extra effort from the programmer. CUDA itself is built to support it. The part left for the programmer is to select the correct GPU for the upcoming work which can become quite messy if not done correctly. When a CPU program will send work to the GPU, it establishes a context(view of a GPU) between the calling CPU thread and the GPU. The context contains all states for that GPU, from the virtual address space, streams, events, allocated blocks of memory, and more. The important part is that only one context can be active at a GPU at any given time. Even though a GPU can only be working with one context at a time, it can be used by many contexts. The CPU is in charge of swapping between the contexts, and uses the CUDA driver to do so. This can be done in such a way that each CPU thread switches contexts itself, or one thread handle context switching for all, when the GPU goes idle. Note that any memory allocated by one context *cannot* be accessed by other contexts, as the allocations are in different virtual address spaces [36]. The problem left for gaining high performance is building the functions to handle the switching and delegating workloads to the different GPUs, including in the calculation estimates of how long the work will take for each GPU, and taking into account their different specifications if they are not of the same type. Figure 4 displays an installation using two Nvidia graphics cards.

Calculation Correctness

There are standards set for calculations such as binary floating-point representations in the *IEEE 754*. CUDA devices follow these with only some exceptions. They come from the same problem as any other calculation performed in parallel, where doing two operations simultaneously can change the answer if either one is done before the other. In CUDA the programmer must be

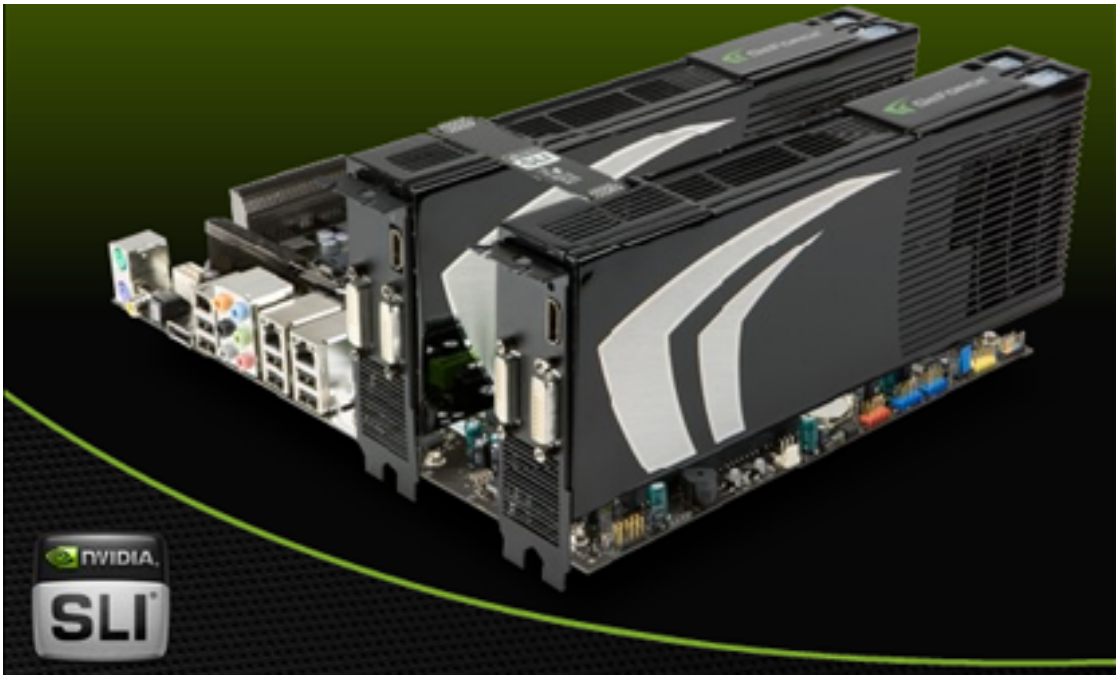


Figure 4: Multiple Nvidia GPUs

extremely careful to make sure threads reading the same memory do so in an atomic fashion to prevent faulty answers.

Nvidia provides a debugger for all systems, such as CUDA-GDB for Linux, and Parallel Nsight for Windows. The debugging tools are created for extensive control over everything that happens on and with the GPU, from what threads, blocks and warps that are active, to memory consumption and variables, and also how much of the total bandwidth of the card is being used by the program or kernel-wise for further optimization of the code. More details of the debuggers can be found at [40].

2.2.3 NIDS

Snort and Suricata methodology is based on *misuse detection*. These are signature-based systems where it looks for difference between abnormal and normal traffic that comes through the network using known attack patterns. Abnormal traffic detection bases itself on previous encounters, turned into attack signatures from observations and traces of malware. Both Snort and Suricata support a variety of accelerators; this includes *pfring*, and spesific network hardware such as *endace capture-cards*, *napatech capture-cards*, *Intel X10 capture-cards*, and *myricom capture-cards*.

Rules and Signatures

Rules and Signatures are created to counter vulnerabilities. Microsoft's definition of a vulnerability, by Scott Culp [52]: "A vulnerability is any flaw that makes it infeasible, even when implemented or used properly, to prevent an attacker from; usurping privileges, regulating

internal protected operations, compromising data, or assuming trust that was not explicitly granted." Exploits are the practical term for the techniques or methodologies that take advantage of the actual vulnerabilities.

Signatures as used in Snort and Suricata are defined as any detection method that use a piece of data, content, marks or characteristics to uniquely identify a known exploits entering the network. The signatures have limited protection capabilities, since the exploit first has to be detected before a signature can be written to detect it. Any zero-day exploit would pass through signature detection easily.

Rules are based on detecting the process of how a vulnerability would run on the system. This detection method is designed to work against zero-day exploits, where the signatures may have been changed to be undetectable, but where the overall process of the vulnerability remain the same and still can be detectable. A rule often uses a signature that it will look for; in Snort Rule Language (SRL) this is named *content* and can consist of plain text or be combined with *regular expressions* 2.1.

All rules used in Snort and Suricata are based on the SRL. The *Sourcefire VRT Certified Rules* are the official rules explicitly for Snort and are created, tested and approved by the Sourcefire Vulnerability Research Team (VRT). The Snort community also has their own internally shared rules; *Community Rules* referring to all rules that are created and tested by the community and are freely available to everyone using Snort.

Another type, Shared Object (SO) rules, are loadable modules that can extend detection capabilities of the NIDS. The rules were first made with Snort version 2.6.0 with its new API. The rules written in C code, allow for detection of vulnerabilities that would be impossible to detect by a mere text based rule from the normal Snort rule language. This functionality is also used by the VRT to release binary modules without publishing the actual vulnerability. This is used in cooperation with vendors such as Microsoft, who through joint programs like MAPP [31] can share vulnerability information with NIDS and antivirus companies who create patches and detection methods before they are known to the public [24].

The largest open source community *Emerging Threats* specialize in creating rule-sets for both Snort and Suricata and other firewalls. The community has been at work since early 2003, though under different names, originally as *Bleeding Snort*. Emerging Threats is also funded by the US Army Research Office and the National Science Foundation. The rulesets are updated daily and are free to use for any organizations, commercial and private.

A typical SRL rule usually consist of an action, header, rule-options (content signatures), ID, and revision number. An example of a rule from Emerging Threats for both Snort and Suricata is provided in Listing 2.2:

```
1
2  alert tcp HOME_NET any -> EXTERNAL_NET HTTP_PORTS
3  (msg:"ET TROJAN Dapato/Cleaman Checkin";
4  flow:established,to_server;
5  content:".php?rnd="; http_uri;
6  fast_pattern; content:"GET";
7  http_method; pcre:"/\?rnd=\d{5,7}\x20HTTP1\/1\.[01]\x0d\x0aHost\
       x3a\x20/";
8  content:!"User-Agent|3a|"; http_header;
9  content:!"Accept|3a|"; http_header;
10 reference:md5,1d26f4c1cfedd3d34b5067726a0460b0d;
11 reference:md5,45b3b6fcb666c93e305dba35832e1d42;
12 reference:url,www.microsoft.com/security/portal/Threat/
       Encyclopedia/Entry.aspx?Name=Trojan%3AWin32%2FCleaman.G;
       classtype:trojan-activity;
13 sid:2014200;
14 rev:3;)
```

Listing 2.2: An example of a rule [15]

Snort

Snort (developed and maintained by Sourcefire) has for over a decade been de facto standard when it comes to open source NIDS with millions of downloads and over 400,000 registered users world wide. It has become the most known and used *open source* IDS/IPS in the world, capable of performing real-time traffic analysis and packet logging, protocol analysis, content string matching, and attack vectors such as buffer overflows, port scans, and much more. The downside to Snort is that there is no standard support for multi-core CPUs or GPU acceleration. Additionally, the official development of the new version Snort 3.0 has been put on hold since around 2008. However, the community is still developing plugins and minor upgrades. The possibilities an administrator has in regard to configuration in Snort are quite massive. The system can be put into three main modes; run as a mere IP logger like tcpdump on specific IPs, a packet-logger of network traffic, or as full NIDS (default). Together with a configured firewall, Snort can also act as an Intrusion Prevention System (IPS) (snort in-line mode). This however is not much used and is not encouraged, as it requires huge amount of testing and configuration of rules for allowing/blocking the correct traffic. As mentioned, Snort is not multi-threaded, which is its most significant weakness. It is however possible, with great effort to set up snort to run with multi-processes, assigning one snort execution to each CPU core and have them cooperate. It requires lots of configuration but is the way most large companies and organizations have managed to keep up in speed and is well tested.

Snort as a whole has grown fairly large, and consist of many major modules, each addressing its own important part of the NIDS. As it is open source it takes advantage of other projects, and uses the *libpcap* library [53] for *packet capturing*. The packet is then handled by a *packet decoder* and delivered to the module with *pre-processors*. Next in the pipeline is the *detection engine* where *string matching* is done. If the module detects abnormal traffic, the event is reported to the *alert-module*, which will generate a packet alert and send it to screen or a custom database. Figure 5

show the basic data flow of Snort NIDS, which is based on the pcap runmode.

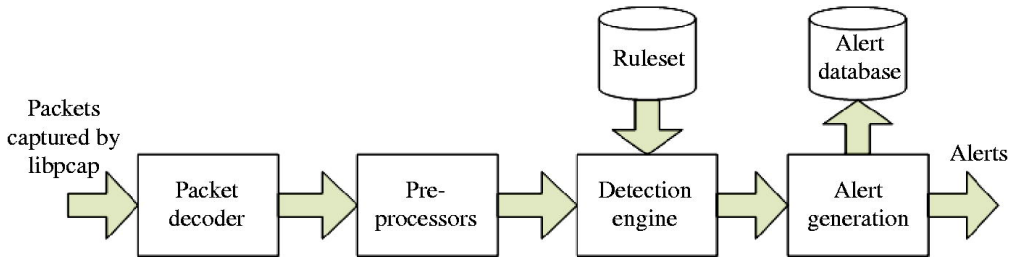


Figure 5: Basic data flow of a Snort NIDS

The problem with Snort lies in the fact that the code base has started to become quite old and outdated for today's hardware. It does not support multi-core processors, and is only run as a single thread, as mentioned before. Snort does not have any actual support for GPU acceleration. It has such a large code base that it has proven a gigantic task to rewrite the system for multi-threading. The Snort 3.0 project, which was supposed to address this issue has been put on ice and has not been in development for the last years. NIDS administrators have however found a workaround for managing the problem. They build the system and configure snort in such a way that they can run one instance of Snort on each core, combining the power and splitting the workload between multiple processes. This has been done for quite some time, but it is hard to set up, and maintain.

A good thing about Snort is that it has been well documented over the decade it has been in development and there are countless sub-projects that have been built to support and enhance the power of snort, releasing the main thread of snort of some of the workload. Most of the projects develop for the alerting module, making better ways of controlling the visual display of alerts and events for the administrators. Some also make it possible to control many Snort sensors at once, such as those used in massive organizations and business infrastructures, over many locations around the world. This makes it possible to have one control center for traffic analysis. Barnyard2 [17] enables parsing of the open source log file standard unified2, and lets Snort save computational power by saving data efficiently by only saving to the disk and taking no further actions. Oinkmaster is another project that handles updating and managing the rule sets in Snort. It can handle all the major types of rules, such as VRT rules, community rules, Emerging Threat rules and other 3rd party such as those released for Suricata. The full list of additional improvements to Snort can be found in [12].

Suricata

Suricata (developed and maintained by the non-profit organization Open Information Security Foundation (OSIF)) is basically a "clone" or "fork" of Snort in the way that it uses the same solution for creating rules and signatures. In this way, the community behind Snort can continue using their expertise without much effort for developing code, rules and signatures for Suricata instead. Suricata's main mission is to continue open IDS development where the Snort team has failed, without being blocked by the licenses governing Snort. The solution is to have a scaled, enterprise-ready IDS for government as well as private sector. The program code is written in C,

and is mainly designed for scalability. It utilizes multi-threading on CPU which allows the IDS to easily take advantage of all hardware power given. Suricata is also the only open source system that has started implementing GPU acceleration support [42], though still at an experimental stage. Worth mentioning is that Nvidia is one of the technology partners of OSIF to help develop CUDA GPU acceleration. Currently, Suricata uses a Wu-Manber algorithm implementation in their GPU detection engine, but the future plan is to implement Aho-Corasick to get rid of extra post processing and pattern chopping, as well as the issue with using byte blocks (see section 2.2.4 for more details on Wu-Manber). It neither takes advantage of the texture, shared memory and mapped memory, which in the latest releases of CUDA has been improved in speed and bandwidth by a great amount[36]. As of now, the speed of the GPU implementation is still not faster then an equal CPU version.

Suricata has also managed to get Ivan Ristić to join their development team, one of the lead developers from ModSecurity. Ristic created the library *libhtp* a security-aware parser for the HTTP protocol, which has been included into the Suricata project, providing very advanced processing of network streams within Suricata [42].

A mentionable problem with Suricata, compared to Snort, is the lack of documentation. There are hardly any resources on the net, by the community or the developers, such as guides for installation and maintenance, or details about the implementation itself. The documentation available on the OSIF WIKI is disappointingly short and barely enough to get the system running [43].

Suricata is also based on the pcap run-mode like Snort, module based implementation for; *capturing*, *decoding*, *stream processing*, and *detection module*. Figure 6 show the Suricata run-mode. Most of the same 3rd party tools used for Snort, also work for Suricata. OSIF recommend the use of the open source software tools BASE [5] and Squil [59] for managing events in Suricata.

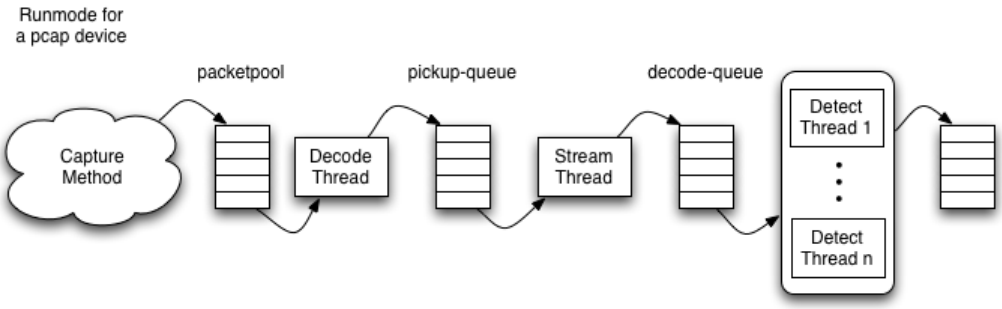


Figure 6: Runmode for a pcap device in Suricata [43]

2.2.4 Algorithms

In a NIDS *detection engine*, the main component is the *string matching algorithm*. The objective of the algorithm is to search for a string (also called a pattern), or multiple strings (multi-pattern). Single pattern matching algorithms will search through a text for each known pattern individually, which means they will loop through the whole text as many times as the total number of patterns. Original examples of single pattern algorithms are the Knuth-Morris-Pratt(KMP) [26] and Boyer-Moore(BM) [8]. Multi-pattern string algorithms operate quite differently, searching through the known text only once, looking for all patterns at the same time. Aho-Corasick [1] and the Wu-Manber [61] are the most original and used multi-string algorithms. String algorithm requires one or many search string inputs, and a text or datafile to compare against. Over the last 50 years there have been many algorithms developed for searching strings and patterns. They are optimized for different types of work, and vary in complexity.

Within string matching there are two different problems for algorithms, *exact* string matching and *approximate* string matching. Exact is the classical approach, and what is mostly used by NIDS, as it sets out to find the perfect match to a given pattern or string. Approximate string matching takes error into account, and can find strings even though some characters are wrong, or missing. An example of approximate search is *Google search*, which give a "Did you mean this word?" if the user spelled wrong compared to what is indexed the most.

String algorithms follow three different approaches in general, *prefix searching*, *suffix searching* and *factor searching*. This is based on which direction the search window is shifted. For prefix search we try to find the longest prefix of the window that is also a *prefix* of the strings or patterns searched after. In suffix search the window shifts backwards, and the algorithm tries to find the longest *suffix* of all strings and patterns. Suffix is best used when the strings are actual words where the endings are the same on many words such as "ion" or "ing". Factor search also work backwards, looking at the suffix of the window, but also takes into account that it is a *factor* of the string or pattern.

Complexity

When we are to analyze an algorithm we concentrate on the order of growth of the algorithm's operation count to perform a given task, meaning the amount of resources such as memory usage and time consumption that are needed for execution. This will give us a indicator of the algorithm's efficiency. Scientists like to talk about this as *algorithm complexity*, the difference in matching time, quantified based on the input to the algorithm. Complexity in this thesis is represented with the O-notation, pronounced "big oh". A formal definition is given in many books, for example in [28]: "A function $f(x)$ is said to be in $O(g(x))$, denoted $f(x) \in O(g(x))$, if $f(x)$ is bounded above by some constant multiple of $g(x)$ for all large x, i.e., if there exist some positive constant c and some nonnegative integer x_0 such that $f(x) \leq cg(x)$ for all $x \geq n_0$".

Basically, time complexity is calculated by counting all the elementary operations the algorithm has to perform on the computer. We estimate that all elementary operations take the same fixed amount of time, leaving us with a constant factor that will provide us with an overall impression of the algorithm. Typically we want to know two complexity scenarios, *average* and

worst-case. For instance, an algorithm could have a complexity of $O(2^n)$ which would be an exponential growth, often seen in worst-case scenarios, or $O(n)$ meaning it would take linear time to the input of the algorithm, often seen in average time. Note, *exponential growth* is really bad and not wanted in any algorithm.

Naïve string search

The naïve string search is the most basic form of string search algorithms. It performs no techniques for optimizing memory or increase of speed. The algorithm will start at the first text symbol in *n*, and try to match it against the first symbol in *s*, the *search string / pattern*. If this is matched as positive, the algorithm increment its counters, and compares the next string symbol *s+1* to the next symbol in the text *n+1*. The process will continue until the whole string is found, or the algorithm reach a mismatch. It will then either report a success, or fail back to the first string symbol *s*, and restart the matching process against the next symbol in the text (doing *n=n+1*). The average complexity of naïve string search is $O(s+n)$, and has a worst-case scenario at $(O(s*n))$. Example of naïve string matching; see Table 3.

Table 3: Naïve string search

String Pattern:	**a**	**b**	**b**	**a**				
Text:	**a**	**b**	**a**	**b**	**a**	**b**	**b**	**a**
Iteration 1:	**a**	**b**	**b**					
Iteration 2:		**a**						
Iteration 3:			**a**	**b**	**a**			
Iteration 4:				**a**				
Iteration 5:					**a**	**b**	**b**	**a**

Knuth-Morris-Pratt (KMP)

There are of course smarter ways of searching for a string in a given text in less worst-case time then $(O(s*n))$, one is the Knuth-Morris-Pratt (KMP) algorithm invented in 1974 by Donald Knuth, Vaughan Pratt and James H. Morris. The optimization performed in this algorithm is that when a mismatch occurs, one can skip the next or several following characters to avoid doing a re-comparison of the same symbol which is a waste of time and computational power. This is because KMP keeps the information unlike the naïve approach that does nothing further with the data found during the scans of the text. To achieve this, the algorithm needs to precompute a failure function(Table 4. The function indicate the largest possible shift for the skipping of characters. In detail, the failure function $\int$ for pattern *s*, store the length of the longest prefix of *s* inside *j* that is a suffix of $s[1,\ldots,j]$. The true objective is to map the repeated substrings inside the pattern itself. When the failure function is complete the main algorithm can start, and will run like the naïve function until it meets a mismatch. When it does it will look up the failure function table, and seek out the location of the closest suffix that matches the prefix of the pattern. This leads to a worst-case of $O(s+n)$, and a required memory usage with size of (s) for the failure table.

The algorithm generates an automaton from the text, using $O(s)$ time (Figure 7).

Table 4: Knuth-Morris-Pratt - Failure Function

x	0	1	2	3	4	5
P[x]	a	b	a	a	b	a
f(x)	0	0	1	1	2	3

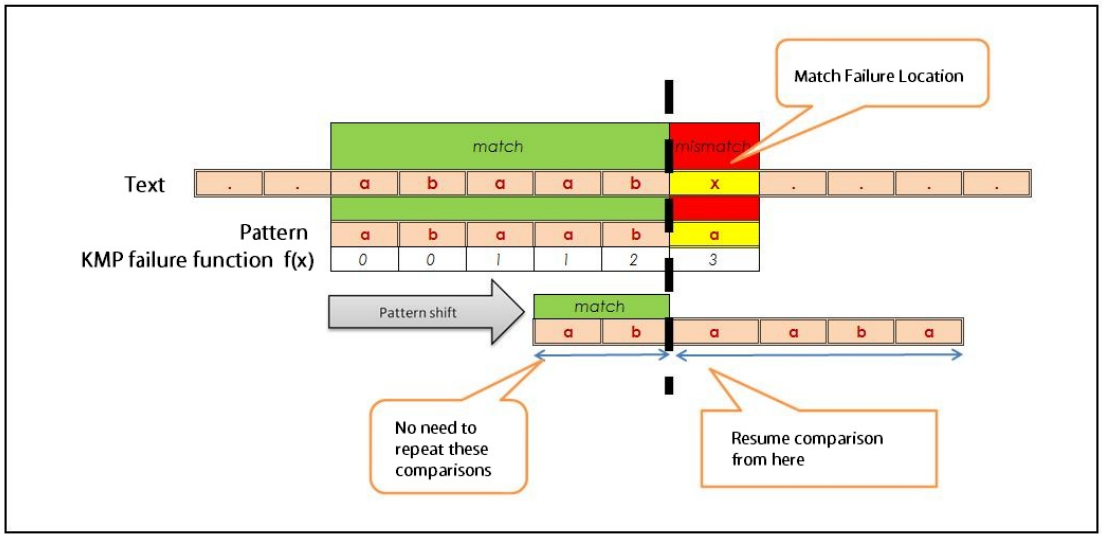
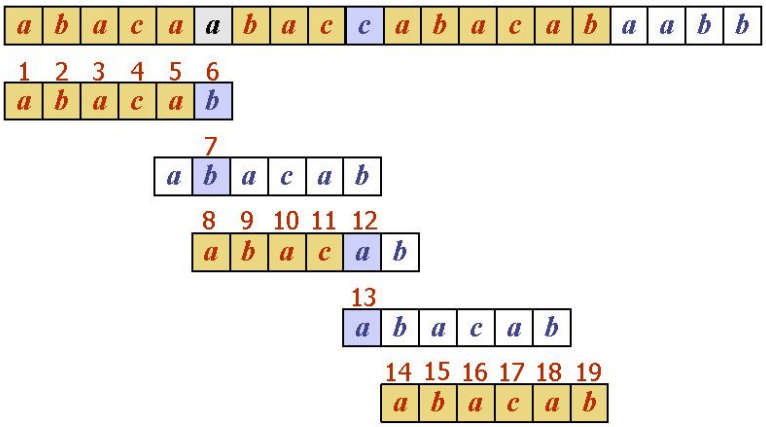


Figure 7: Knuth-Morris-Pratt - Algorithm Walkthrough [7]

When the algorithm then runs the search against this automata, the worst-case scenario matching time has been reduced to $O(s + n)$. This algorithm is still limited by the fact that it is a single pattern matching algorithm, meaning this process has to be repeated for every string or pattern that wants to be searched. The algorithm is great if one only need to look for a low number of strings, but rapidly becomes too demanding when reaching higher numbers such as in NIDS with thousands of rules and signatures. Example execution of KMP see Figure 2.2.4.



x	0	1	2	3	4	5
P[x]	a	b	a	c	a	b
f(x)	0	0	1	0	1	2

Figure 8: Knuth-Morris-Pratt - Example Execution [7]

Boyer-Moore (BM)

The string search algorithm Boyer-Moore was developed in 1977 by Bob Boyer and J Strother Moore. BM is like KMP, a single-pattern search algorithm, the main difference of the two is that BM starts its search with the *last character of the pattern*, rather then the first as most other algorithms - it works backwards. The idea behind it is that if the last character does not match, there is no reason to go through all the characters from the beginning. BM also uses "failure function" like the KMP algorithm although it is called *bad character shift rule* and another *good suffix shift rule*. Improved versions of Boyer-More have been even better, further increasing the logic where where to look for match in fewest iterations possible. The Turbo-Boyer-More requires some extra memory space, storing the factor of the text from the previous iteration where a suffix shift was successful (did not mismatch). Turbo-BM has $O(2n)$ as worst-case scenario, and an average of $O(n)$. Another derivative of BM is Boyer-Moore-Horspool. It uses less space, but require an even better suffix table. The inner loop of the algorithm makes it perform less overhead for each iteration of the loop. While the average complexity of BM-Horspool is $O(n)$, the worst-case scenario $O(s * n)$ is higher then the real BM algorithm, but theoretical worst-case is hard to achieve in practice. All Boyer-Moore algorithms works better with longer pattern strings, longer will make it possible to skip characters faster while shifting the pattern string.

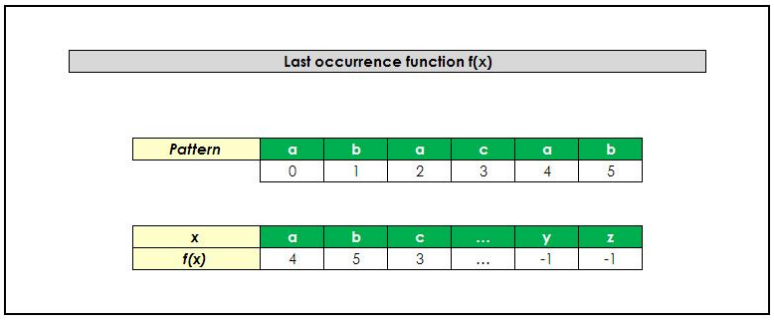


Figure 9: Boyer-Moore - Last Function [6]

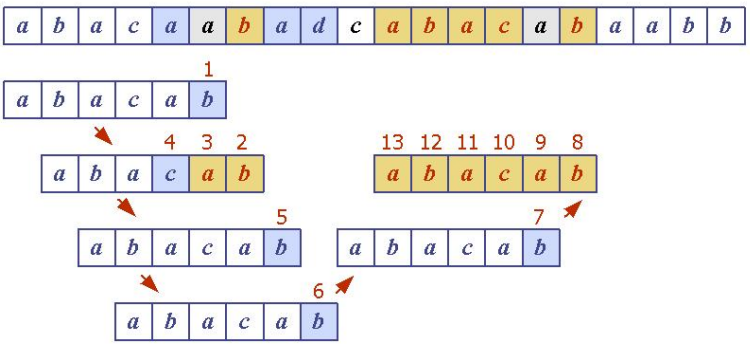


Figure 10: Boyer-Moore - Example Execution [6]

Aho-Corasick (AC)

Aho-Corasick (AC) was published in 1975 by Alfred V. Aho and Margaret J. Corasick [1], and is the most known multi-pattern search algorithm. The main objective is to construct a Final State Machine (FSM) for pattern matching from known keywords (or given as a dictionary), then process input strings against the FSM with *exact-set* matching. When AC was developed the fastest algorithm to perform bibliographic searches was KMP, which was not very effective on large datasets. This leed to the idea of creating a linear-time method by preconstructing a search pattern from all the known keywords, a generalization of KMP for multiple strings. The algorithm itself can be broken into four different parts; construction of a *goto function* (Algorithm 2.2.1 and Figure 11), construction of a *failure function*(Algorithm 2.2.2 and Figure 12) establishing a *pattern matching machine*(Algorithm: 2.2.3), and finally construction of the *next move function*(Algorithm 2.2.4). The last part is only used in the Deterministic Finite Automaton (DFA) version of the algorithm.

Algorithm 2.2.1: AHO-CORASICK - GOTO FUNCTION(K)

Input: Set of keywords K = $\{y_1, y_2, \ldots y_k\}$
Output: Goto function *g* and a partially computed *output* function
Comment: We assume output(s) is empty when state s is first created and
g(s,a) = fail if a is undefined or if g(s,a) has not yet been defined.
The procedure enter(*y*) insert into the goto graph a path
that spells out *y*.

$newstate \leftarrow 0$
for $i \leftarrow 1$ **to** k
  do $enter(y_i)$
for each a such that $g(0, a) = fail$
  do $g(0, a) \leftarrow 0$

procedure: $enter(a_1 a_2 \ldots a_m)$:
$state \leftarrow 0$
$j \leftarrow 1$
while $g(state, a_j) \neq fail$
  do $state \leftarrow g(state, a_j)$; $j \leftarrow j + 1$
for $p \leftarrow j$ **to** m
  do $newstate \leftarrow newstate + 1$; $g(state, a_p) \leftarrow newstate$; $state \leftarrow newstate$
$output(state) \leftarrow \{a_1 a_2 \ldots a_m\}$

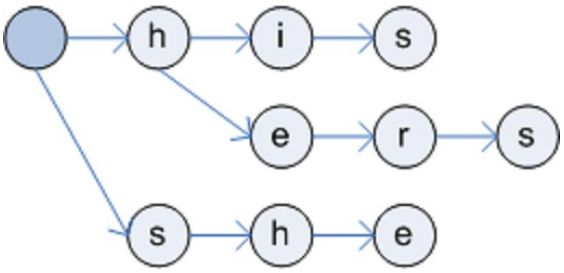


Figure 11: Aho-Corasick - goto function [44]

Algorithm 2.2.2: AHO-CORASICK - FAILURE FUNCTION$(g, output)$

Input: Goto function g, and output function *output*
Output: Failure function f and output function *output*

$queue \leftarrow empty$
for each a such that $g(0, a) = s \neq 0$
  do $\begin{cases} queue \leftarrow queue \bigcup \{s\} \\ f(s) \leftarrow 0 \end{cases}$
while $queue \neq empty$
  do $\begin{cases} \textbf{Comment:} \text{ let } r \text{ be the next state queue} \\ queue \leftarrow queue - \{r\} \\ \textbf{for each } a \text{ such that } g(r, a) = s \neq fail \\ \quad \textbf{do} \begin{cases} queue \leftarrow queue \bigcup \{s\} \\ state \leftarrow f(r) \\ \textbf{while } g(state, a) = fail \\ \quad \textbf{do} \begin{cases} state \leftarrow f(state) \\ f(s) \leftarrow g(state, a) \\ output(s) \leftarrow output(s) \bigcup output(f(s)) \end{cases} \end{cases} \end{cases}$

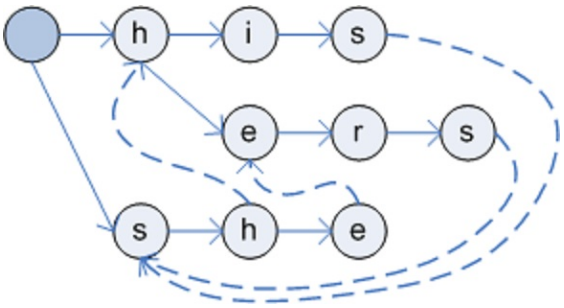


Figure 12: Aho-Corasick - failure function [44]

AC has been used in numerous different projects since its creating. In NIDS the first was written in 2002 by Marc Norton and released with Snort in version 2.0 with support only for Non-Deterministic Finite Automaton (NFA). This early version was not very effective, and in

2003 Daniel Roelker joined the development team. The new version included a massive speed optimization and improved memory usage. The update added sought after state transitions abilities to formats such as full-matrix, sparse-matrix, banded-matrix, and sparse-banded-matrix. The overall biggest update was support for both NFA and DFA. Even today AC is still the fastest algorithm and default in the Snort detection engine. The research team of Zha et al.(2011) [63] attempted to make CUDA implementation of Aho-Corasick and claimed good results, speedups between 8.5-9.5 times more then against a single-thread CPU, and 2.4-3.2 times more then a multi-threaded CPU. Suricata, the NIDS with GPU acceleration has the Aho-Corasick algorithm on their todo-list for future implementation.

Algorithm 2.2.3: AHO-CORASICK - PATTERN MATCHING MACHINE(x)

Input: $x = a_1 a_2 ... a_n$ where each a_i is an input symbol
Output: goto function *g*, failure function *f*, and *output* function

$state \leftarrow 0$
for $i \leftarrow 1$ **to** n
do **while** $g(state, a_i) = fail$
do $state \leftarrow f(state)$
$state \leftarrow g(state, a_i)$
if $output(state) \neq empty$
then **print** i
print $output(state)$

Algorithm 2.2.4: AHO-CORASICK - NEXT MOVE FUNCTION(g, f)

Input: Goto function *g*, falure function *f*
Output: Next move function δ

$queue \leftarrow empty$
for each *symbol* a
do $\delta(0, a) \leftarrow g(0, a)$
if $g(0, a) \neq 0$
then $queue \leftarrow queue \bigcup g(0,a)$
while $queue \neq empty$
do **Comment:** let *r* be the next state in queue
$queue \leftarrow queue - \{r\}$
for each *symbol* a
do **if** $g(r, a) = s \neq fail$
do $\{queue \leftarrow queue \bigcup \{s\} \delta(r, a) \leftarrow s$
else $\delta(r, a) \leftarrow \delta(f(r), a)$

Input symbol		Next state
State 0:	h	1
	s	3
	.	0
state 1:	e	2
	i	6
	h	1
	s	3
	.	0
state 9: state7: state 3	h	4
	s	3
	.	0
state 5: state 2:	r	8
	h	1
	s	3
	.	0
state 6:	s	7
	h	1
	.	0
state 4:	e	5
	i	6
	h	1
	s	3
	.	0
state 8:	s	9
	h	1
	.	0

Figure 13: Aho-Corasick - Output of Next Move Function δ

Wu-Manber (WM)

The Wu-Manber (WM) was published by Sun Wu and Udi Manber in 1992 [61] and was based on the standard Boyer-Moore algorithm and numeric scheme work by Baeza-Yates and Gonnet [3]. Like Aho-Corasick, WM is a multi-pattern string algorithm handling tens of thousands of patterns at once. The algorithm is designed for typical searches, rather than worst-case scenarios. It takes a different path to multi-patterns then AC and is based on the single-pattern Boyer-Moore approach, turned into a multi-pattern algorithm. The major change against normal BM is the preprocessing stage. Instead of looking at characters one by one, WM looks at them in *blocks*. However, short patterns still slow down the algorithm just like with BM, as the max possible shift is the max length of shortest pattern.

The WM algorithm creates four different tables in preprocessing; *shift*, *hash*, *suffix* and *prefix*. The *Shift table* is similar to *regular shift table* in BM and is used to determine how many characters can be skipped, based on the blocks. The blocks are mapped into an *integer* used in the shift table. *Hash table* is used to determine which pattern is the candidate for the match and to verify the match if the shift value is 0, and minimize the total number of patterns needed for comparison. It contains a list of patterns where the last block character hash into *integers* that match the *shift* table index. *Suffix table* give optimization against normal text search, where natural language is not random. Suffixes such as “ing” or “ion” are very often used while writing English text. These endings cause collision in the hash table. At this point the shift will return 0, and the algorithm has to examine all possible patterns with this suffix. This can lead to allot of patterns that need to be checked. The *prefix table* is then introduced to hash prefixes of the patterns as they are much less likely to match and cause collision. The chance that patterns have the same suffix *and* prefix is very small [61]. The average runtime complexity of WM is defined to be $O(BN/m)$, where *B* is the size of *blocks* used, *N* is the size of the text searched and *m* is the minimum length of all patterns.

The original paper of the Wu-Manber algorithm [61] is very fussy. It does not provide anything that permits the reader to calculate the best size of shift and hash tables, and none of the functions are specified, except a partially implemented main method (Algorithm 2.2.5).

Algorithm 2.2.5: WU-MANBER - MULTIPLE STRING MATCHING(B, SHIFT, HASH)

Input: Precomputation of B, SHIFT and HASH tables
$pos \leftarrow \ell min$
while $pos \leq n$
do
- $i \leftarrow h_1(t_{pos-B+1} ... t_{pos})$
- **if** SHIFT[i] = 0
  - **then**
    - list $\leftarrow$ HASH[$h_2(t_{pos-B+1} ... t_{pos})$]
    - Verify all the patterns in list one by one against the text
    - $pos \leftarrow pos + 1$
  - **else** $pos \leftarrow pos +$ SHIFT[i]

Wu-Manber algorithm was used in the NIDS Snort for a period of time, between the first version of Aho-Corasick and the second improved version, however it was found inferior to AC and is no longer the default algorithm, but remains as an optional configuration. Sun Wu and

Udi Manber also implemented their algorithm into a UNIX search tool; *agrep*(1989-1991) [60], that has later been ported to other operating systems such as OS/2, DOS, Windows and Linux.

3 Implementations & Results

This chapter contains the contribution of the thesis. In it, we have created multiple programs in C++ with the CUDA API from Nvidia. The code is based on the algorithms described in section 2.2.4. We have created a CPU version and a GPU version to compare the performances of the algorithms against each other.

3.1 Scope

The experiments we conducted where performed to be comparable with the previous work in the area as described in section 2, most importantly the work of Vasiliadis et al [55]. With [55] the team implemented the Knuth-Morris-Phat and Aho-Corasick algorithm. The main features they found that caused drastic drop of performance was the overhead of the PCI bus while transferring to the GPU. The solution was to create a buffer for packets, then transfer the collected packets in one large combined memory copy. The numbers acquired for the KMP algorithm seemed unrealistically low, under 1Mbit/s at 4000 patterns, which was the reason we chose to perform an experiment on this algorithm as well. The Aho-Corasick implementation results were also vague, as it reached only a 1.4 Gbit/s of speed, limited at 1000 random patterns while a real Snort engine would run on average with 10 000 to 20 000 patterns. They also neglected to mention the size of the network packet. However, the next experiment had a network packet size of 1500 bytes and a speed of 2.6 Gbit/s. The second claim stated that it is unrealistic to process small network packets on the GPU. Together this did not feel like a sound implementation of Aho-Corasick, as this algorithm's speed is mainly bound to the amount of patterns used, not the size of the packets which would only increase processing time, NIDS should theoretically perform slower with larger network packets. Their implementation neither took into account the fact that the signatures may have capital letters which would limit all specific signatures. The kernels implemented also used global variables for all threads to write into. This is strongly discouraged. Even though atomic operations will stop any race conditions, it will force the different threads to use additional resources waiting for access. The PFAC Library [10] mentioned in section 2 uses this type of method and is therefore not chosen for comparison as we do not further investigate this approach. Vasiliadis et al. published more papers since 2008. However, these hold massive implementations, that include packet capturing, preprocessing and multi threading of the CPU to gain even higher effect from asynchronous memory transfer to the GPU. Implementing this would effectively improve our approach as well, however the limited resources forced us to take out these improvements from our scope of work.

3.2 Limitations

To limit the size of the project we focus on the main area of the NIDS, the detection engine itself. While a normal NIDS take input from the network card, we chose to have one static string of text for the experiments where we could place a known number of signatures that we wanted

to detect. This approach was chosen because taking data from the network card and input into the program would be just another module, with basically the same end result, one long string of characters that we use as input into the GPU algorithm. The second limitation introduced is to skip regular expression signatures, as the parser for the input would become significantly more complex, while the end result would still be the same, only a much larger pattern array input for the algorithm. The last limitation applied was to use the same procedure for sending and retrieving data with the GPU for all the algorithms. The approach used for sending and retrieving is most likely not optimal. However, it is consistent throughout the experiments and lets us perform comparison between the algorithms performance in a much more consistent way. We focused on creating a common pool of functions that use the same techniques, such that any optimizations would affect all implementations at the same time, and leave the algorithm specific optimizations for the kernel itself. The overall simplified structure of the NIDS prototype can be seen in figure: 14.

3.3 Lab Environment

For the project we set up a fresh installation with the following hardware components shown in Table 5 and software in Table 6. A clean and minimal system installation was performed to maximize the performance of the hardware we had to our disposal. Note that while the experiments where conducted Nvidia released a new version of their Nsight Visual Studio Edition 2.2, with local single GPU CUDA debugging! This new feature was too good to be left out which gave us a much easier way of debugging our code, and we switched to this software while performing the second experiment and found many bottlenecks for both experiment one and two, which was implemented for both.

CPU	Intel Core i7-920 Processor
Motherboard	Gigabyte GA-EX58-UD5, X58
Hard disk	Corsair CM SSD 128GB
Memory	Corsair XMS3 DDR3 1600MHz 6GB CL7
GPU #1	Nvidia GeForce GTX 580
GPU #2	Nvidia GeForce GTX 460

Table 5: Lab Environment Hardware

Software used for programming CUDA:

CUDA compute versions include support for new functions and hardware calls, and one has to program accordingly towards the target audience of the program. For our system running with two Nvidia cards, the lowest CUDA compute version is 2.0, strangely enough on the newest card Nvidia GTX 580. This has to be specified to the nvcc-compiler, as to what architecture it should compile for. The GPUs does not have the same processing power, and will not finish the task given at the same time. Table 7 shows details from the Nvidia GTX 580 card, and Table 8 for the GTX 460 card. For this reason we chose to use the GTX 580 for the single core tests, as it has the highest performance, even though it has lower compute version. The optimization performed in the higher compute version did not account for the performance gained in general by the improved hardware. From the details it is worth noticing is the difference in maximum global

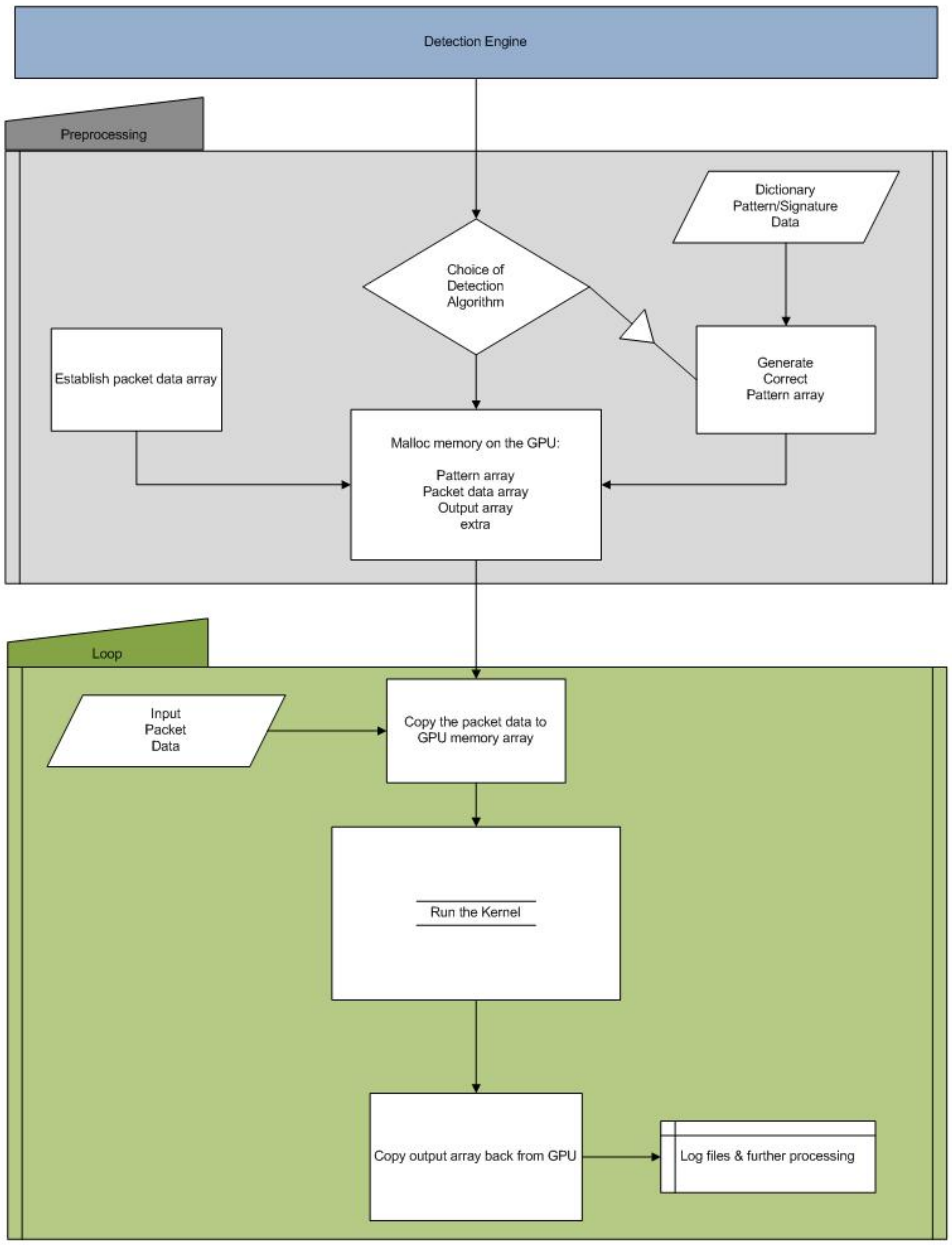


Figure 14: Detection Engine using GPU

Compiler & IDE	Visual Studio 2010 Ultimate Trial
CUDA Toolkit	CUDA Toolkit version 4.1 RC
CUDA SDK	Nvidia GPU Computing SDK 4.1
Debugger Exp 1	Nvidia Parallel Nsight 2.1
Debugger Exp 2	Nvidia Parallel Nsight 2.2
Aditional Libraries	
Libs #1	Boost Library 1.49.0
Libs #2	Thrust Library 1.51
Visuals	Matlab 7.11.0

Table 6: Required software for programming

memory available and also the number of multi-processors. Other variables remain the same, such as the number of warps, threads per block and thread dimensions, which are standard for the compute version 2.x. These numbers must always be carefully taken into account when designing a program. For our project, we chose to lock the code to our specific GPU card, while a real program would implement functions designed for each different compute versions that can be used by the users.

CUDA Information Nvidia GTX 580	
Compute capability: Clock rate: Device copy overlap: Execution timeout:	2.0 1566000 Enabled Disabled
Memory information	
Total global memory: Total constant memory: Max memory pitch: Texture alignment: Multi processor count: Shared mem per mp: Registers per mb: Threads in warp: Max threads per block: Max thread dimensions:	1610612736 bytes 65536 bytes 2147483647 bytes 512 16 49152 bytes 32768 bytes 32 1024 (1024, 1024, 64)

Table 7: Hardware Details: GeForce GTX 580

CUDA Information Nvidia GTX 460	
Compute capability:	2.1
Clock rate:	1630000
Device copy overlap:	Enabled
Execution timeout:	Disabled
Memory information	
Total global memory:	1073741824 bytes
Total constant memory:	65536 bytes
Max memory pitch:	2147483647 bytes
Texture alignment:	512
Multi processor count:	7
Shared mem per mp:	49152 bytes
Registers per mb:	32768 bytes
Threads in warp:	32
Max threads per block:	1024
Max thread dimensions:	(1024, 1024, 64)

Table 8: Hardware Details: GeForce GTX 460

3.4 Experiment #1

The main objective of this experiment was to get CUDA up and running, working together with the different libraries exploring the CUDA environment and in the end create a basic prototype for a naïve string search (see Section: 2.2.4) for more details on this type of algorithm. For input to this prototype we chose to use a small word list of bad words, abbreviations and some “leet speak” alternatives of these bad words from the English language. The “bad word” list contained a total of 458 different words (or patterns) to be processed against our input text. The second larger word list contained over 50000 words. This was downsized to 35500 real and unreal words from A to Z from the British English language [32]. This then created an approximately "“worst case"” having the kernels needed to go through every single option of different starting

characters.

Algorithm 3.4.1: NAÏVE STRING SEARCH, SINGLE THREAD CPU VERSION(x)

Input: Set of patterns (dictionary) and (packet data).
Output: Occurrences of patterns found in the input packet data.
Comment: We assume the patterns has been processed into a list or array and that the input data is an array of characters.

```
index ← 0
for i ← 0 to sizeof(dictionary)
do { for j ← 0 to size of(input text)
     do { for k ← 0 to string length of(dictionary[i])
          do { if dictionary[i][j] ≠ packet[i + index]
               do {Break out of loop
               else Increase index by 1
               if index == string length of (dictionary[i])
               do {Report pattern(dictionary[i]) found at location i
```

Algorithm 3.4.2: NAÏVE STRING SEARCH, ACCELERATED ON GPU(x)

Input: Set of patterns, int array with pattern lengths(offsets) and packet data.
Output: Occurrences of patterns found in the input packet data.
Comment: Each pattern is given its own thread on the GPU
We assume the patterns has been processed into a list or array and that the input data is an array of characters.

```
index ← 0
for i ← 0 to size of(input text)
do { for j ← pattern offset[threadid]
     to pattern offset[threadid +1]
     do { if pattern − list[j] ≠ packet[i + index]
          do {Break out of loop
          else Increase index by 1
          if index == string length of (dictionary[i])
          do { atomically lock the output memory space required
               Report pattern(dictionary[k]) found at location i
```

3.4.1 Execution

For optimizing the code, the best way to transfer data into the GPU is to perform one large memory transfer requiring to only access memory once. This is quite important as for every network packet the information has to be copied over. Saving processing time is our number one objective. CUDA does not have any support for standard library function calls such as STRING, and we must use basic character arrays (char *) to hold our data. Arrays of arrays are also problematic and would make memory operations messier. For performance we rather make a normal one dimensional character array with all the content of the packet and send in other variables containing other metadata needed. The same goes for patterns, we insert the pattern-offset values (the length of the words) in a separate integer array.
The pattern list and offset array has only to be copied to the GPU *once* for the duration of the program. Even though we change input text and rerun the string searching kernel, we only pass pointers to where in the GPU memory the patterns are stored, not the actual information. This is however not true for the actual content of the input text, which is sent to the GPU for each execution of a new kernel. The same optimization technique is used, the packet is stored as a character array, and transfered in one large bulk of memory copy.

The last step that needs to be done before launching the GPU kernel is to specify memory space where the kernel is to store the output of the algorithm it will be running. This memory space is reserved(malloc'ed) and GPU memory address pointer stored on the CPU side, and then sent to the GPU as a kernel parameter. This has to be done because the kernel does not have an actual output back to the calling function. We have to manually retrieve the data that now has been placed in the reserved memory space on the GPU from our CPU function that started the kernel in the first place. To save memory transfer bandwidth, the output of the algorithm is purely the ID of the pattern that was found and the location of the input text where it was located.
The final operations on the output are carried out by the CPU. In a normal setting we would process this data into a logging file or a database, however for our prototype implementation, displaying the results directly to the screen is more then sufficient. A commented version of the engine and kernel (also CPU single thread version) implemented to handle naïve string search is included in the Appendix A.1.

Packets	Runtime	Speed
100	5.8ms	11.02Mbit/s
800	45.2ms	11.31Mbit/s
3000	139.5ms	11.32Mbit/s
6000	339.3ms	11.31Mbit/s
9000	508.53ms	11.32Mbit/s
20000	1142.2ms	11.21Mbit/s

Table 9: Average execution runtime of the Naïve string search on GPU

The two implementations of naïve string search were made so they used the same method for processing the word list and input text. Results show a 5x improvement in speed for the GPU version. Compared to the CPU version of the algorithm, the GPU had an average 0.3ms runtime,

and the CPU 2ms runtime. The method used to measure this was to start timing the program at the point where it in an NIDS would run in loop against incoming network packets, and not adding in the time needed to transfer the dictionary to the GPU memory which is only done once for the whole execution of the program.

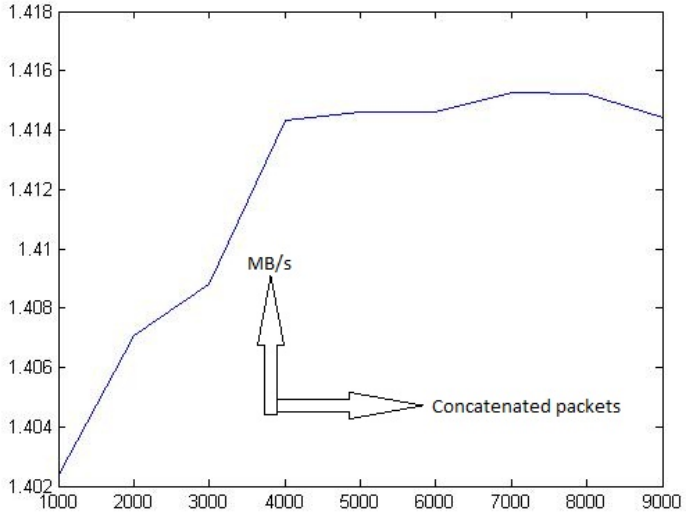


Figure 15: Naïve GPU Execution, 456 patterns checked vs. packets with length of 40 chars.

3.4.2 Analysis

This experiment gave us much insight into how CUDA works in general, parallel bottlenecks, upsides and downsides. Starting up the GPU to do computational work is quite expensive in form of time, meaning starting a new kernel on the GPU for each network packet that enter the system is impossible. What we found out, was that taking a collection of packets that has entered the system, preferably by the use of preprocessors extracts only the main content we wish to search as we simulated in this experiment, then concatenates the data into one long array to be processed by the GPU. Memory transfers themselves is what takes time, not the amount they copy over. The overhead caused by the PCI bus is the main reason why this approach is faster. Doing collections of packets also makes it possible to overlap the different network packets with ease. If the preprocessor that arranges the network packets to the GPU can do so in sequential order, it would mean "“free"” overlap of the packets in a session processed. The GPU handles thousands of threads at once, but there is a limit to how many that are actually processed at the same time. The Nvidia 580GTX as used in this experiment has 16 multiprocessors. Each of them can handle 1024 threads, giving a total of 16384 kernels at once. This however does not stop us from starting even more kernels. The additional kernels are placed in a queue, and will be run by the multiprocessors when they have completed their previous task.

Kernels, or threads as described in Section 2.2.2, are put together in blocks, and it is these blocks that are loaded onto the multiprocessors. The size of these blocks is always set by the

programmer, and is very program dependent. We experimented with many different sizes, where the documentation advises the size should be a multiplication of the size of the warp, in our case 32, where we deemed a block size of 512 yielded the best results in terms of speed.

We experimented on the two different approaches of starting these kernels (single-pattern and multi-pattern algorithms described in section 2.1). The first use the patterns themselves as the kernel base. What this means is that for each pattern, we start a kernel and only process that pattern on this specific kernel. The kernel will loop through the packet or collection of packets, looking for a match. The second approach starts a kernel for each packet and runs every pattern listed against this single packet.

We also attempted a third approach, (which actually was our initial idea from state of the art research for how to process packets). In this method we started a kernel for each single character in the network packet, processing all listed patterns, starting from this character until match or no match was found. This is also the method used in PFAC Library [10] mentioned in Section 2.1. The method proved highly ineffective due to the memory transfers and overall cost of creating variables on the GPU for every single character, and was after testing dismissed as flawed as we could not reach any useful speeds while testing.

3.5 Experiment #2

For this experiment we wanted to accelerate well known algorithms for string search as described in Section 2.2.4. These algorithms are specifically designed for performing string search and should theoretically perform better then a naïve brute force approach. The algorithms we chose to focus on were based on how they would react against being run on a GPU rather then CPU and the overhead calculations needed for the algorithms to function. In the end we chose the Knuth-Morris-Phatt, and Aho-Corasick, as they are two different algorithms from the two main factions, single-pattern and multi-pattern string search algorithms. We did not implement every algorithm listed in Section 2.2.4, as they are merely provided to give the reader an overall picture of the different types of algorithms available, their weaknesses and strengths, where they use similar techniques to achieve their best performance.

3.5.1 Knuth-Morris-Pratt (KMP)

The first algorithm we wanted to implement was the Knuth-Morris-Pratt(KMP) (as described in Section 2.2.4). KMP is a single pattern algorithm, similar to the Naïve approach. The implementation was done in relatively similar way, such that each pattern received a thread on the GPU. The KMP method requires preprocessing of the patterns in order to find prefixes and suffixes, to be able to skip one or more characters in the actual search process. This preprocessing time would in an actual NIDS only be required to run *once* at startup, and is therefore not included in the presented results. The KMP is quite similar to the naïve algorithm. However, we observed that the extra preprocessed failure table gave a marginal speed increase at around 16.5%.

Preprocessing

The preprocessing is done on the GPU where it is based on incrementing two variables, prefix and suffix, where prefix starts at 0 and suffix starts at 1. These variables are indexes of the character

array that makes up the pattern. We do a while loop over the suffix position we count and check that it has not reached the size of the char array. If a match between the prefix position and the suffix position is found, the location is stored in an array on the GPU. The suffix and prefix indexes are incremented and the loop checks the next round of characters. If it did not find a match, only the suffix index is incremented, the loop restarts and the past prefix is again used to match the new suffix. The preprocessed failure table stays on the GPU for the duration of the program. The location of the GPU memory has already been stored in host memory when the pre-kernel was started and will be used as input for the actual KMP kernel later.

Packets	Runtime	Speed
100	5.22ms	12.2 Mbit/s
800	40.5ms	12.6 Mbit/s
3000	152.7ms	12.56 Mbit/s
6000	306.2ms	12.53 Mbit/s
9000	459.3ms	12.54 Mbit/s
20000	1021.3ms	12.53 Mbit/s

Table 10: KMP execution, 456 patterns checked vs. packets with length of 40 chars.

Algorithm 3.5.1: KNUTH-MORRIS-PHATT(KMP), PREPROCESSING(x)

Input: Set of patterns (dictionary), pattern offsets
Output: Table containing failure function
Comment: We assume the patterns has been processed into a list or array

```
index ← 0
for i ← 0 to sizeof(dictionary)
    do { for j ← 0 to size of(input text)
         do { for k ← 0 to string length of(dictionary[i])
              do { if dictionary[i][j] ≠ packet[i + index]
                     do {Break out of loop
                     else Increase index by 1
                   if index == string length of (dictionary[i])
                     do {Report pattern(dictionary[i]) found at location i
```

Execution

As KMP is a single pattern algorithm, we chose to place one pattern on each kernel thread on the GPU. The kernel then starts a while-loop over an index for the packet input, we increment until we reach the size of the packet.We check if the current character in the pattern array matches the character in the packet at our given index. If true, we increment both the index of the packet and the index of the pattern array. When the loop then restarts we check the next character in line from both pattern and packet, which is repeated until we either locate the whole pattern

and store results to the output file, or we find a mismatch. At a mismatch we first check if the index of the pattern has been incremented, which means there might be a suffix that matches this prefix that was found doing the preprocessing of failure states in the patterns. We then set this index to the content of the preprocessed failure table of this given index. For example, if the character behind the index 3 was an *a*, we check the failure table at the index 3 which represent this *a* to see if there are more *a's* in the current pattern that we can jump at, to continue the matching process. This effectively let us skip any extra characters between the first and the last *a*. If there was no increased pattern index to begin with, the algorithm only increments the index for the packet and restarts the loop process.

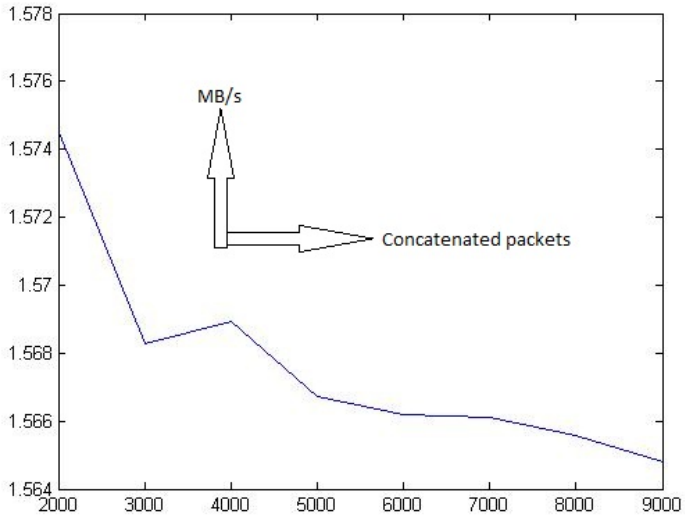


Figure 16: KMP GPU execution, 456 patterns checked vs. packets with length of 40 chars.

Algorithm 3.5.2: KNUTH-MORRIS-PHATT(KMP), GPU EXECUTION(x)

Input: Set of patterns (dictionary), pattern offsets (p_offsets),
table of failure states (f_table)
Output: Occurrences of patterns found in the input packet data.
Comment: We assume the patterns has been processed into a list or array and
that the input data is an array of characters.

```
tid ← ThreadID
pattern_index ← p_offsets[tid]
packet_index ← 0
while pattern_index < packet_size
  do { if dictionary[patternindex] ≠ packet[packet_index]
       do { if pattern_index == pattern_size − 1
            do { while current_output_location ≠ free
                 do {Find new location
                 Report dictionary[tid], found at location i
                 pattern_index + 1
                 packet_index + 1
            else if pattern_index > 0
            do {pattern_index = f_table[p_offset[tid + pattern_index − 1]
            else {packet_index + 1
```

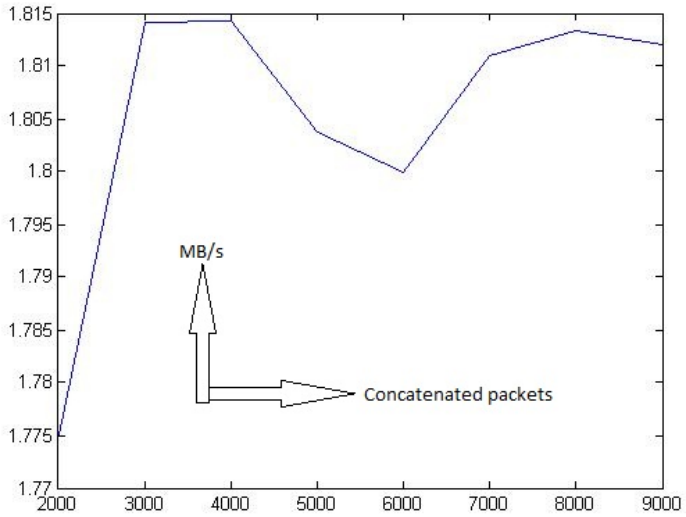


Figure 17: KMP CPU Implementation, 456 patterns checked vs. packets with length of 40 chars.

Analysis

The improvement in speed from Naïve to KMP was less then we had expected, but at the same time confirmed Vasiliadis et al. [55] claim that KMP does not work well on GPU. Overall the

KMP implementation proved that our initial hypothesis of how to parallelize algorithms on the GPU was flawed, and reaching the desired speed could not be done with a single-pattern methodology. However when analyzing our results we saw that our method on how to start kernels in the KMP was poorly chosen as we had no easy way to optimize the number of threads to run simultaneously. If we were to have more patterns then the maximum parallel thread amount, the additional kernels would stay in queue until the multiprocessor had completed the previous work. This does not by itself make it very bad, however the performance drops quite many percent as can be seen in Figure 16. However, this would benefit more from using multiple GPUs instead of one, and split the amount of patterns that has to be started as kernels over all accessible GPUs if the total number of possible threads when combining the GPUs exceeded the total number of patterns. At maximum, we achieved results showing 12.56 Mbit/s processing power with our GPU implementation at 456 patterns. The CPU version was more impressive, yielding at best 14.5 Mbit/s with the same number of patterns. We believe the solution for the GPU would be to have the kernels based on packets for starting a thread. This would solve the issue with unused or queued kernels that we had in our prototype as it bases the kernels on the patterns.

3.5.2 Aho-Corasick (AC)

The Aho Corasick algorithm (see section: 2.2.4) is very different from both the naïve and the KMP algorithms. It is based on performing multiple pattern searches at once, meaning each thread started on the GPU will compare the input towards all given patterns before it stops. This process would be quite troublesome if we did not have preprocessing of the patterns and would take far too much computational power. The Aho Corasick method does this by first creating a tree structure as shown in section: 2.2.4 at Figure 11.

Preprocessing

The preprocessing in Aho-Corasick is more advanced then the KMP algorithm, requires more work, but the increase in speed is worth the trouble. As stated in the technical background of the algorithm in section 2.2.4 it consist of four different parts; construction of a *goto function* (Algorithm 2.2.1 and Figure 11), construction of a *failure function*(Algorithm 2.2.2 and Figure 12) establishing a *pattern matching machine*(Algorithm 2.2.3), and finally construction of the *next move function*(Algorithm 2.2.4). In our implementation for the GPU we early saw the problems with the pattern matching machine part of the algorithm. Aho-Corasick uses a tree structure to handle all patterns, however tree structures proved to be extremely hard to construct on the GPU. Creating it on the CPU and transferring to the GPU was also unsuccessful, as all nodes in the tree contain pointers to memory locations in main memory, and not to memory located on the GPU. To solve this problem, the tree was created in CPU, traversed with a *depth first search* (DFS) approach, then the data was stored as a character array, similar to the naïve and KMP approaches. This gave us a single array containing all the patterns in one single line and the same effectiveness in memory access as the previous algorithms. The failure function and the output function were solved by creating two additional arrays, the same size of the character array, and using its index to map both failure states and output states on the respective index locations as the character array with the DFS traversal results. The output and failure states

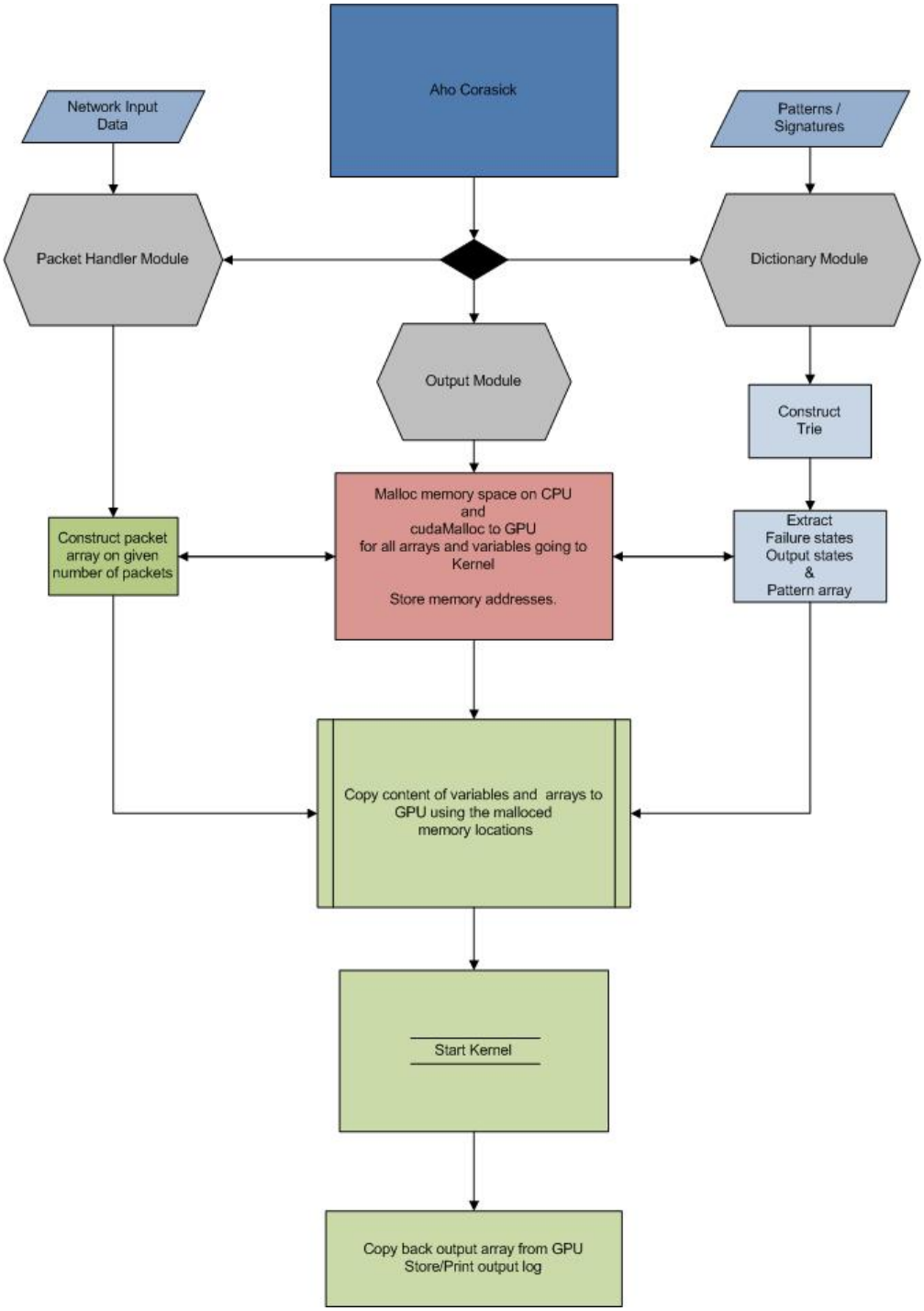


Figure 18: Aho Corasick Implementation. Red and blue areas are run once to limit the computation needed, green is looping over new packets placed into the malloced memory space on the GPU for each new collection of packets.

were mapped inside the tree class for the patterns as they were enrolled in the tree structure. The failure array contained indexes for the pattern character array, as to which new index to jump if the algorithm should not find a positive match. The output array contained the index of the pattern as it was stored in the dictionary of patterns in the host memory. This approach gave us the ability to only send the content of the output array back to the CPU after executing the kernels, if and only if this pattern was found, limiting additional memory transfers from each kernel launch.

Execution

Executing our Aho-Corasick kernel was done by giving each thread one packet. To gain performance we perform one large memory copy of all the packets concatenated into one single transfer. These are again "“split back"” by a second array of indexes containing the length of each packet. Each kernel has a thread index (TID), and by using this TID as index in length array, the kernels know the location in memory of their packet at TID to TID+1, the start of the next packet. Next we created a while loop to run from the start to the end of this packet, where we check if the current character in the packet does not match the current character in the pattern. If so we set the pattern index to the correct position given in the failure table that was preprocessed and restart the loop. However if they did match, we then check if this character was the last character in a word that matched the pattern. This is given in the output array, at the same index as the character being checked. If it was the last character, we store the result in the output array. The loop restarts to continue checking as the current results may be its own pattern, but also a prefix of a longer pattern, not yet found. The kernel ends as it reaches the point checking the last character in the packet input.

	Function Name	Module ID	Function ID	Start Time (µs)	Duration (µs)	Occupancy
1	ahoKernel<int>	11	1	8 742 988,455	67 647,072	66,67 %
2	ahoKernel<int>	11	1	17 359 005,415	67 995,232	66,67 %
3	ahoKernel<int>	11	1	26 164 939,399	69 658,336	66,67 %
4	ahoKernel<int>	11	1	35 655 961,287	75 808,672	66,67 %
5	ahoKernel<int>	11	1	46 016 288,103	83 363,904	66,67 %
6	ahoKernel<int>	11	1	57 340 428,551	92 025,024	66,67 %
7	ahoKernel<int>	11	1	69 764 191,911	101 797,984	66,67 %
8	ahoKernel<int>	11	1	83 336 867,911	111 718,848	66,67 %
9	ahoKernel<int>	11	1	97 990 470,535	121 343,456	66,67 %
10	ahoKernel<int>	11	1	113 758 399,655	131 315,744	66,67 %
11	ahoKernel<int>	11	1	130 619 938,439	140 979,168	66,67 %
12	ahoKernel<int>	11	1	152 328 398,855	184 459,616	66,67 %
13	ahoKernel<int>	11	1	177 092 449,671	211 607,904	66,67 %
14	ahoKernel<int>	11	1	202 027 565,767	212 718,656	66,67 %

Figure 19: Occupancy at 1024 kernels per block.

CPU Implementation

To have comparable numbers between CPU and GPU we created another implementation purely located on the host memory and using a single thread on CPU, like a Snort system would run. The prototype uses the same dictionary system as the GPU version, but has it’s own functions for handling packets and outputs. The tree structure on the other hand is the same function, though using a parameter to cancel the request for GPU memory assertion built into

	Function Name	Module ID	Function ID	Start Time (µs)	Duration (µs)	Occupancy
1	ahoKernel<int>	7	1	5 178 752,043	36 328,607	100,00 %
2	ahoKernel<int>	7	1	10 913 626,826	42 453,376	100,00 %
3	ahoKernel<int>	7	1	17 603 998,698	50 772,737	100,00 %
4	ahoKernel<int>	7	1	28 717 881,547	90 579,903	100,00 %
5	ahoKernel<int>	7	1	40 129 477,514	92 171,936	100,00 %
6	ahoKernel<int>	7	1	51 561 318,186	92 768,033	100,00 %
7	ahoKernel<int>	7	1	63 129 824,843	93 924,064	100,00 %
8	ahoKernel<int>	7	1	75 253 463,531	98 905,024	100,00 %
9	ahoKernel<int>	7	1	88 560 115,114	109 898,081	100,00 %
10	ahoKernel<int>	7	1	103 060 372,490	119 197,793	100,00 %
11	ahoKernel<int>	7	1	119 316 903,723	134 510,879	100,00 %
12	ahoKernel<int>	7	1	139 940 265,643	175 783,071	100,00 %
13	ahoKernel<int>	7	1	160 912 450,186	177 945,376	100,00 %

Figure 20: Occupancy at 512 kernels per block.

Packets	Runtime	Speed MB/s	Speed Mbit/s
7000	9,03ms	61,99MB/s	496 Mbit/s
8000	9,62ms	**66,48MB/s**	**532 Mbit/s**
9000	15,23ms	47,25MB/s	378 Mbit/s
10000	15,49ms	51,62MB/s	413 Mbit/s
11000	16,22ms	54,23MB/s	434 Mbit/s
12000	16,72ms	57,38MB/s	460 Mbit/s
13000	17,42ms	59,68MB/s	478 Mbit/s

Table 11: GPU Aho Corasick. 456 signatures checked vs. packets with length of 80 chars.

Packets	Runtime	Speed	Speed Mbit/s
7000	6,68ms	83,79MB/s	670Mbit/s
8000	6,99ms	**91,45MB**/s	**732 Mbit/s**
9000	11,17ms	64,40MB/s	515 Mbit/s
10000	11,51ms	69,49MB/s	556 Mbit/s
11000	12,07ms	72,87MB/s	583 Mbit/s
12000	12,34ms	77,78MB/s	622 Mbit/s
13000	12,87ms	80,75MB/s	646 Mbit/s

Table 12: GPU Aho Corasick. 35500 signatures checked vs. packets with length of 80 chars.

that piece of code. For measuring the speed of this function we use the built in windows API call *QueryPerformanceFrequency* and *LARGE_INTEGER* to get as high resolution measurements as possible. The code works at the exact same way as the GPU version with one difference. Instead of analyzing the packets in parallel, every packet is placed in a stack and searched one by one. However the search is non stop, which avoids the need to take overlapping into account.

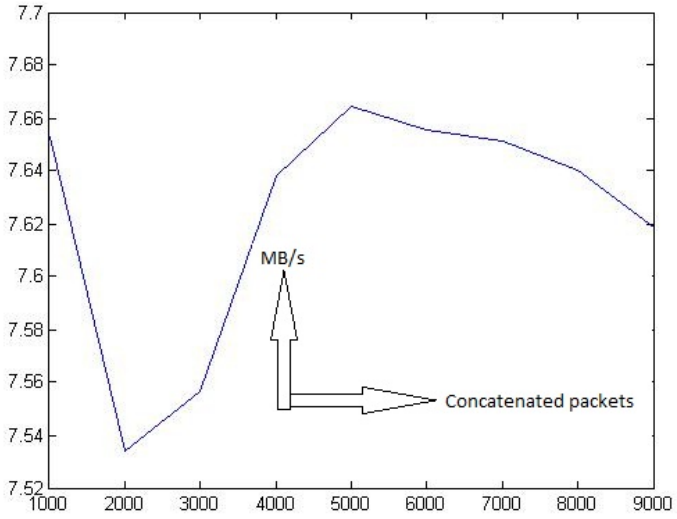


Figure 21: Aho-Corasick CPU Execution, 456 patterns checked vs. packet length of 80chars.

Analysis

The experimentation gave us insight into parallelization of such an advanced algorithm as Aho-Corasick really is. We had to radically change our approach to gain enough processing power if it ever were to support a multi-gigabyte NIDS solution. We did many different attempts to find the most optimal approach. The results yielded many interesting numbers, where few were not expected and gave us reason to overhaul the system again and again. Before the experiment we had a numerous amount of probable bottlenecks that we had to counter. However, it showed many of these bottlenecks were solved by the parallelization process itself. A result that proved quite illogical at first was the speed variation from the different number of signatures loaded into the algorithm. The GPU managed to work even faster with higher number of patterns. Going from the small pattern set of 456 to the large set of 35500 patterns, the prototype execution increased in speed a staggering 37% on the same amount of packets. The results show we gain increase in speed as there are less race conditions from the different threads against the memory of the GPU. (Tables 11 and 12). The size of the input packets however change the speed dramatically for the GPU. This proves the importance of using preprocessors to limit the amount of data checked by the detection engine of the NIDS. We tested a range of sizes of the packets, from 40 to 1500 bytes, where the size of 40 with 35500 patterns yielded results over 1.1 Gbit/s speed (see Table 13). The Aho-Corasick CPU implementation performed better then both naïve search and KMP, both

CPU and GPU versions. It achieved a top peak in performance with packets of size 40 chars, and 35500 patterns a 7.65 MB/s, or 61.5Mbit/s (see Figure 21). However, lowering the number of patterns from the same set down to 1000 gave a dramatic increase in performance reaching a staggering 127 MB/s, or 1016 Mbit/s. The last number is close to the GPUs performance in Mbit/s. However, it should not be miss-interpreted that they perform equally as the GPU handle 76 times more patterns at the same time. At the same time it shows the algorithms do not have linear performance, closest at 35500 patterns the GPU only reaches a 17.8 times increase in performance over the CPU implementation.

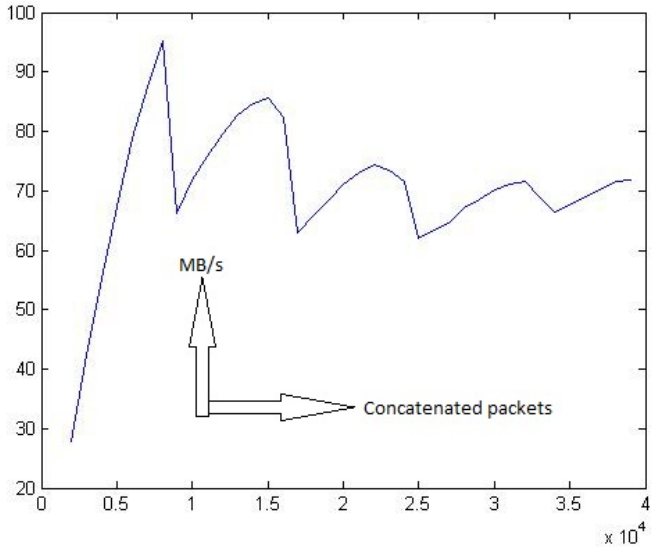


Figure 22: Aho-Corasick GPU Execution, 35500 patterns checked vs. packet length of 80chars.

Packets	Runtime	Speed	Speed Mbit/s
8000	2,72ms	132,2 MB/s	1058,2 Mbit/s
8000	2,63ms	136,8 MB/s	1094,2 Mbit/s
8000	2,61ms	**137,6 MB/s**	**1100,2 Mbit/s**
8000	3,20ms	112,3 MB/s	898,9 Mbit/s
8000	2,64ms	136,1 MB/s	1089,1 Mbit/s
8000	2,63ms	136,3 MB/s	1090,9 Mbit/s

Table 13: GPU Aho Corasick. Multiple runs of 35500 patterns checked vs. packet length of 40chars.

Finding the optimal way to set flags and boundaries in a CUDA program to run proved quite difficult with Aho-Corasick, as it depends much on what is actually being calculated on the GPU, the amount of memory that has to be read and also *IF, FOR and WHILE* loops in the kernels. The goal of the kernels executed is to have them all complete at the same time, avoiding worst case scenarios, making kernels in the active block sit and wait for the last kernel to complete. To do this we used the NSight CUDA Analyser software to spot the bottlenecks in the system.

One large improvement found for the Aho-Corasick specifically was the use of a lower block size then what the graphics card should theoretically use. Our Nvidia 580 GTX card has a max of 1024 kernels per block (see section 3.3 for details of the card), an execution with 1024 kernels on each block showed the program only could reach an occupancy level of 66.7% (see Figure 19), using only 2/3rd of the total power available. This was solved by lowering the amount of kernels per block to 512, wich is also a multiplication of the warp size 32 giving us a total amount of threads multiplied by 16 which then fits the number of multiprocessor cores on the Nvidia 580 GTX perfectly, reaching an occupancy level of 100%, (see Figure 20).

4 Conclusions

In this thesis, we have researched Network Intrusion Detection Systems (NIDS) and experimented with prototype implementations of known algorithms; naïve string search, Knuth-Morris-Pratt and Aho-Corasick. The prototypes were limited to the detection engine of the NIDS that holds string search algorithms that perform matching of one array of characters against another. NIDS systems normally hold tens of thousands patterns or signatures of viruses and known attacks and use these to check incoming network traffic for malicious activity and code. The thesis focused on improving the performance of NIDS systems by accelerating the detection engine algorithms with parallelization by the use of Graphical Processing Units (GPUs), which by hardware are developed for doing thousands upon thousands of similar tasks at the same time. We chose to work with Nvidia GPUs and the CUDA API in C++ programming language to develop our implementations, found to be the leading choice in parallel technology. The main goal of the thesis was to find out if the performance of NIDS algorithms could be enhanced by the use of GPUs, and what factors in the algorithms that created the highest increase in performance and also the largest bottlenecks in the prototype implementation.

First we performed theoretical analysis of graphic cards, CUDA, NIDS solutions and a wide variety of known high-performance string search algorithms. This also included research of tools and technologies to enhance the quality of our work to locate weaknesses and bottlenecks in the systems. Next we performed a series of experiments to dig deeper into CUDA and parallelization techniques starting with the simplest form of algorithmic approach, naïve string search, also known as brute-force method. This provided us with insight into CUDA and also shifts our mindset into how sequential tasks could be performed in parallel which proved to be rather difficult at times. The experiment yielded the expected result that a naïve approach is not sufficient to handle network speeds as we see in a NIDS today. Next, we implemented the KMP algorithm that is an advanced version of naïve string search requiring preprocessing of the patterns. This however did not improve the speed with sufficient amounts such that it could be viable in a NIDS setting. Finally, we worked with Aho-Corasick which is the state of the art multi-pattern algorithm used in the open source NIDS as a CPU implementation today such as Snort and Suricata. This algorithm reached staggering 1100 Mbit/s with 33500 patterns. Previous work shows similar speeds but with only 1000 patterns [55], which is quite a feat in the amount of pattern data processed, 77 times more.

4.1 Research Questions

This subsection will highlight how we answered our research questions in this thesis.

4.1.1 To what extent can NIDS performance be increased by using GPU?

This was the main objective of the thesis and is best presented in the discussion of our implementations of Aho-Corasick at Section 3.5.2, were results show the GPU to execute a

hundred times faster then the CPU version of the algorithm, with over seventy times more patterns.

4.1.2 What parts of the NIDS can be optimized?

This question was discussed and analyzed in the Technical background of NIDS at Section 2.2.3 where we also present the basic data flow charts of the NIDS Snort and Suricata, and highlight that the *Detection Engine* module is best suited for GPU acceleration. We also acknowledge that the preprocessing module of these NIDS also has potential for GPU acceleration. However, this was not further researched in this thesis.

4.1.3 Which program specific factors give the increase, or decrease in performance?

The specific factors presented that increased and decreased the performance for GPU algorithms are based on the implemented algorithms. We discuss and analyze these factors in each subsection of the experiments. For experiment one see Section 3.4.2. For experiment two see Section 3.5.1 and Section 3.5.2. The most important factors are also presented in the following section.

4.2 Summary of Contributions

In the thesis we have contributions as results of our experiments. We present our GPU and CPU implementations of the algorithms naïve string search(Appendix A.1), KMP(Appendix A.2) and Aho-Corasick(Appendix A.3). In addition, we here reveal design specific recommendations of factors we found to have the highest impact on our prototypes. These are design specific recommendations on how to gain performance on the GPU for further implementation of the algorithms into existing NIDS such as Snort and Suricata.

- Results show that our GPU accelerated multi-pattern algorithm perform over a hundred times better, with over 70 times more patterns then any of our single pattern algorithms executed on both GPU and CPU, including the CPU version of our multi-pattern algorithm.
- Network packets should be preprocessed by the use of a buffer into a concatenated packet based on session order and stored into an array. This will provide an easy way to overlap packets under processing, if the pattern to be found is split directly at the packets. In addition, and even more importantly this will provide a way to do a single malloc and transfer of the data into the GPU memory.
- Kernel and block size require runtime testing as it should not be static unless the program is to do the exact same work on the exact same amount of data each time. A better implementation would contain a dynamic function to measure performance and handle the adjustments of the block and kernel sizes(Appendix A.4.3).
- Creation of kernels must be based on packets in one way or another, to achieve performance and be able to work with the recommended solution stated in the previous point(Appendix A.4.3).

4.3 Future Research

The first future research proposal is to use NVIDIA GPUDirect technology found in Tesla CUDA cards. These cards allows sharing of pinned host memory with other devices, and allows for accelerated transfers of GPU data to other devices such as network cards that have support for Infiniband. This would make it possible to make an implementation that directly copied the network packets that enter the computer directly to the GPU memory, without the extra overhead of passing through the host memory. This in theory is perfect for a NIDS solution and would increase performance dramatically. We did get the chance as we hoped in this thesis to work on these types of cards. As of the implementations presented in this paper, there are still areas of improvement. As we executed the algorithms we observed memory leakage on the GPU. However, they proved quite hard to locate with the debugging tools at our disposal. Next is the engine that starts up the kernels on the CPU that is working at a single core level, there are potential speed increases by using multi-threading for this solution. This is also necessary for using more then one GPU at the same time, which would require one thread to manage patterns, packets and output, and one thread for each GPU unit. Next, the implementations do not fully utilize the fastest memory available on the GPU, constant memory, but instead use global memory and the automatic caching within CUDA 4.x. We did not attempt to place functionality within the tree structure of Aho-Corasick for supporting regular expressions. However, we believe this could be altered quite easily and that all the structures including the kernel would work just as well. We also propose more effort in research into GPU acceleration of *preprocessors* of the NIDS, which could be the topic for a master thesis in the future. Finally, the Aho-Corasick algorithm should be attempted to be incorporated into an actual NIDS, Snort or Suricata. The base work and structure should now be known, adjusting the modules to fit within the actual NIDS implementation should not be too difficult. We strongly believe that the proposed recommendations for potential performance increase should justify the efforts taken.

Bibliography

[1] Alfred V. Aho and Margaret J. Corasick. Efficient string matching: an aid to bibliographic search. *Commun. ACM*, 18:333–340, June 1975.

[2] M. Attig and J. Lockwood. A framework for rule processing in reconfigurable network systems. In *Field-Programmable Custom Computing Machines, 2005. FCCM 2005. 13th Annual IEEE Symposium on*, pages 225 – 234, april 2005.

[3] Ricardo Baeza-Yates and Gaston H. Gonnet. A new approach to text searching. *Commun. ACM*, 35:74–82, October 1992.

[4] Zachary K. Baker and Viktor K. Prasanna. Time and area efficient pattern matching on fpgas. In *In The Twelfth Annual ACM International Symposium on Field-Programmable Gate Arrays (FPGA 04)*, pages 223–232. ACM Press, 2004.

[5] Project BASE. Base. `http://base.secureideas.net/`, 2009.

[6] Blogspot. Boyer moore algorithm. `http://tinyurl.com/BMExample`, 2010. Examples and C++ Implementation.

[7] Dev FAQ Blogspot. Knuth morris pratt algorithm. `http://tinyurl.com/KMPExample`, 2010. Examples and C++ Implementation.

[8] Robert S. Boyer and J. Strother Moore. A fast string searching algorithm. *Commun. ACM*, 20:762–772, October 1977.

[9] J.B.D. Cabrera, J. Gosar, W. Lee, and R.K. Mehra. On the statistical distribution of processing times in network intrusion detection. In *Decision and Control, 2004. CDC. 43rd IEEE Conference on*, volume 1, pages 75 – 80 Vol.1, dec. 2004.

[10] Ling-Sheng Chien Shih-Chieh Chang Cheng-Hung Lin, Chen-Hsiung Liu. Accelerating pattern matching using a novel parallel algorithm on gpu. `http://code.google.com/p/pfac/`, 2011. Documentation of PFAC Library.

[11] Beate Commentz-Walter. A string matching algorithm fast on the average. In *Proceedings of the 6th Colloquium, on Automata, Languages and Programming*, pages 118–132, London, UK, 1979. Springer-Verlag.

[12] Open Source Community. Snort 3rd party software. `http://tinyurl.com/snort3rdparty`, 2012. List of all projects that enhance the Snort NIDS.

[13] csgillespie. Cpu and gpu trends over time. `http://www.r-bloggers.com/cpu-and-gpu-trends-over-time/`, Jan 2011.

[14] Sarang Dharmapurikar, Praveen Krishnamurthy, T.S. Sproull, and J.W. Lockwood. Deep packet inspection using parallel bloom filters. *Micro, IEEE*, 24(1):52 – 61, jan.-feb. 2004.

[15] Emerging-Threats. Rule 2014200. `http://doc.emergingthreats.net/bin/view/Main/2014200`, 2012. Depato/Cleaman Trojan Checkin rule, rev. 3.

[16] R. Farber. *CUDA Application Design and Development*. Morgan Kaufmann. Elsevier Science, 2011.

[17] Ian @ SecurixLive Firns. Barndyard2. `https://github.com/firnsy/barnyard2`, 2010. Barnyard2 version 2-1.10.

[18] Gainward. Gainward geforce gtx 580. `http://www.gainward.com/main/vgapro.php?id=452`, 2011. Graphics card used for the experiments.

[19] Nen-Fu Huang, Hsien-Wei Hung, Sheng-Hung Lai, Yen-Ming Chu, and Wen-Yen Tsai. A gpu-based multiple-pattern matching algorithm for network intrusion detection systems. In *Proc. 22nd Int. Conf. Advanced Information Networking and Applications - Workshops AINAW 2008*, pages 62–67, 2008.

[20] Intel. Supra-linear packet processing, performance with intel multi-core processors. `http://download.intel.com/technology/advanced_comm/31156601.pdf`, 2006. Whitepaper.

[21] Intel. Removing system bottlenecks in multi-threaded applications. `ftp://download.intel.com/design/intarch/PAPERS/320631.pdf`, 2008. Whitepaper.

[22] Nigel Jacob. Offloading ids computation to the gpu. In *Proceedings of the 22nd Annual Computer Security Applications Conference (ACSAC2006)*, 2006.

[23] A. Kennedy, Xiaojun Wang, Zhen Liu, and Bin Liu. Ultra-high throughput string matching for deep packet inspection. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2010*, pages 399 –404, march 2010.

[24] Kevin Ross - Emerging Threats. [emerging-sigs] can someone explain so rules to me? `http://lists.emergingthreats.net/pipermail/emerging-sigs/2009-July/003142.html`, 2009.

[25] Khronos-Group. Opengl. `http://www.opengl.org/`, 2012.

[26] Donald E. Knuth, James H. Morris, Jr. :l, and Vaughan R. Pratt. Fast pattern matching in strings*, 1974.

[27] Paul D. Leedy. *Practical research: planning and design [by] Paul D. Leedy*. Macmillan New York,, 1974.

[28] Anany Levetin. *The Design and Aanalysis of Algorithms*. Pearson International Edition, 2007.

[29] Lodovico Marziale, Golden G. Richard III, and Vassil Roussev. Massive threading: Using gpus to increase the performance of digital forensics tools. *Digital Investigation*, 4, Supplement(0):73 – 81, 2007.

[30] Chad R. Meiners, Jignesh Patel, Eric Norige, Eric Torng, and Alex X. Liu. Fast regular expression matching using small tcams for network intrusion detection and prevention systems, 2010.

[31] Microsoft. Mapp faq. http://www.microsoft.com/security/msrc/collaboration/mapp/faq.aspx, 2011.

[32] mieliestronk. Corncob wordlist. http://www.mieliestronk.com/wordlist.html, 2011.

[33] Gonzalo Navarro and Mathieu Raffinot. *Flexible pattern matching in strings: practical online search algorithms for texts and biological sequences*. Cambridge University Press, New York, NY, USA, 2002.

[34] Kristian Nordhaug. The aho-corasick algorithm and its implementation in snort. http://tinyurl.com/74ykdjx, 2011. Unpublished.

[35] Nvidia. World's first gpu. http://www.nvidia.com/page/geforce256.html, 1999.

[36] Nvidia. Cuda c best practices guide 4.0. http://tinyurl.com/NvidiaBestPractice40, March 2011. Version 4.0.

[37] Nvidia. Cuda c programming guide. http://tinyurl.com/cudaprogrammingguide, 2011.

[38] Nvidia. Cuda toolkit. http://developer.nvidia.com/cuda-toolkit, 2011. Latest Toolkit Release.

[39] Nvidia. Getting started - parallel computing. http://tinyurl.com/nvidiagettingstarted, 2011.

[40] Nvidia. Nsight. http://developer.nvidia.com/nvidia-parallel-nsight, 2011. CUDA debugging tool for Windows Vista / 7.

[41] Nvidia. What is cuda? http://tinyurl.com/whatiscuda, 2011.

[42] OSIF. Open information security foundation. http://www.openinfosecfoundation.org/, 2011. Main Developer of Suricata.

[43] OSIF. Suricata developers guide. http://tinyurl.com/suricatadevelopersguide, 2011. OSIF Wiki.

[44] Tomas Petricek. Aho-corasick string matching in c. http://tinyurl.com/ahocorasicktrees, 2005. Examples and Implementation of AC algorithm.

[45] The National Academies Press. Stephen cole kleene. http://www.nap.edu/html/biomems/skleene.html, 1994. Lived from January 5, 1909 to January 25, 1994.

[46] Jason Sanders and Edward Kandrot. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley Professional, 1st edition, 2010.

[47] Daniele Paolo Scarpazza, Oreste Villa, and Fabrizio Petrini. Exact multi-pattern string matching on the cell/b.e. processor. In *Proceedings of the 5th conference on Computing frontiers*, CF '08, pages 33–42, New York, NY, USA, 2008. ACM.

[48] Derek L. Schuff, Yung Ryn Choe, and Vijay S. Pai. Conservative vs. optimistic parallelization of stateful network intrusion detection. In *Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '07, pages 138–139, New York, NY, USA, 2007. ACM.

[49] R. Sidhu and V.K. Prasanna. Fast regular expression matching using fpgas. In *Field-Programmable Custom Computing Machines, 2001. FCCM '01. The 9th Annual IEEE Symposium on*, pages 227 –238, 29 2001-april 2 2001.

[50] R. Smith, N. Goyal, J. Ormont, K. Sankaralingam, and C. Estan. Evaluating gpus for network packet signature matching. In *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*, pages 175 –184, april 2009.

[51] Sourcefire. Clamav. `http://www.clamav.net/lang/en/`, 2011. ClamAV antivirus homepage.

[52] Sourcefire. Snort faq. `http://www.snort.org/snort/faq/`, 2011.

[53] tcpdump. Documentation. `http://tinyurl.com/tcpdumpdoc`, 2012. open source project.

[54] Matthias Vallentin, Robin Sommer, Jason Lee, Craig Leres, Vern Paxson, and Brian Tierney. The nids cluster: Scalable, stateful network intrusion detection on commodity hardware. In *In RAID 2007*.

[55] 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.

[56] Giorgos Vasiliadis, Michalis Polychronakis, Spiros Antonatos, Evangelos P. Markatos, and Sotiris Ioannidis. Regular expression matching on graphics hardware for intrusion detection. In *Proceedings of the 12th International Symposium on Recent Advances in Intrusion Detection*, RAID '09, pages 265–283, Berlin, Heidelberg, 2009. Springer-Verlag.

[57] Giorgos Vasiliadis, Michalis Polychronakis, and Sotiris Ioannidis. Midea: a multi-parallel intrusion detection architecture. In *Proceedings of the 18th ACM conference on Computer and communications security*, CCS '11, pages 297–308, New York, NY, USA, 2011. ACM.

[58] 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.

[59] Bamm Visscher. Squil. `http://sguil.sourceforge.net/`, 2011.

[60] Sun Wu and Udi Manber. agrep. `http://www.tgries.de/agrep/`, 1991.

[61] Sun Wu and Udi Manber. Fast text searching: allowing errors. *Commun. ACM*, 35:83–91, October 1992.

[62] Fang Yu, Randy H. Katz, and T. V. Lakshman. Gigabit rate packet pattern-matching using tcam. In *Proceedings of the 12th IEEE International Conference on Network Protocols*, pages 174–183, Washington, DC, USA, 2004. IEEE Computer Society.

[63] Xinyan Zha and S. Sahni. Multipattern string matching on a gpu. In *Computers and Communications (ISCC), 2011 IEEE Symposium on*, pages 277 –282, 28 2011-july 1 2011.

A Appendix

A.1 Experiment #1:Naïve String Search

A.1.1 Naïve Search Engine

```
#include "cuda_runtime.h"
#include "dictionary.cuh"
#include "monitor.cu"
#include "packetInput.cu"


#include "naiveKernel.cu"

#include "book.cu"

#include <ostream>
#include "outputHandler.cu"

template<class T>
void naiveEngine(int device){

  ofstream myfile ("naiveGPU.data", ios::trunc);
  //myfile << "[GPU]" << endl;

  if(!myfile.is_open()) return;
  else{

  HandleError<T>(cudaSetDevice(device), "ahoe", 15);

  int *current = new int;
  HandleError<T>(cudaGetDevice(current), "ahoe", 18);
  std::cout << "Using GPU: " << *current << endl;

  cudaDeviceReset();


  //Information and monitoring
  PM<T> monitor;
  PacketData<T> *packet;

  //Dictionary<T> dictionary("corncob.txt");
  Dictionary<T> dictionary("badwords.txt");

```

```
39   OutputHandler<T> *output;
40   int repeats = 0;
41
42   //load dict to GPU Memory
43   dictionary.upload();
44
45   int blocks;
46   int threads;
47
48
49   while(true){
50
51     repeats += 1000;
52
53     if (repeats >= 10000){
54       myfile.close();
55       return;
56     }
57
58     cout << "num packets at once: " << repeats << endl;
59
60     //if(repeats == 0){
61     //  myfile.close();
62     //  return;
63     //}
64     //create Packet
65     packet = new PacketData<T>(repeats);
66
67     output = new OutputHandler<T>(repeats, 4);
68
69     //Loop of the NIDS starts here
70     monitor.startPM(); //start the monitor for watching GPU
71
72     //load data to GPU
73     packet->upload();
74     output->upload();
75
76     //easy fix use all multicores
77     blocks = 32;
78     threads = (int)(dictionary.num_patterns/blocks)+1;
79
80
81     naiveKernel<T><<<blocks, threads>>>(dictionary.d_list,
          dictionary.d_offsets, dictionary.num_patterns, packet->
          d_packet, packet->h_packet_size, output->d_output, output->
          repeats, output->deltaout);
82
83     output->download();
84
```

```
85      monitor.stopPM(); //Stop the monitor
86
87      //for(int j=0; j< output->repeats*output->deltaout; j=j+2)
88      //if(output->log_output[j] != -1){
89      //  cout << dictionary.h_list[output->log_output[j]] << "
            found in packet: " << output->log_output[j+1] << endl;
90      //}
91
92      cout << "time: " << monitor.elapsedTime << "ms     "  <<
            (((1000/monitor.elapsedTime) * packet->packet_string_size *
             repeats)/1000000)<<" MB/s     " <<  (((1000/monitor.
            elapsedTime) * packet->packet_string_size * repeats)
            /1000000)*8 << " Mbit/s" << endl;
93
94      myfile.precision(5);
95      myfile.setf(ios::fixed, ios::floatfield);
96      myfile << repeats << "  " << packet->packet_string_size << "
            " <<  monitor.elapsedTime << "  " << (((1000/monitor.
            elapsedTime) * packet->packet_string_size * repeats)
            /1000000) <<endl;
97
98    }
99    }
100 }
```

Listing A.1: Naïve Engine

A.1.2 Naïve Search Kernel

```
#ifndef _KERNEL_CU_
#define _KERNEL_CU_

#include "cuda_runtime.h"
#include <stdio.h>

template<class T>
__global__ void naiveKernel(char* pattern_list,
               int *pattern_offset,
               int num_patterns,
               char* packet_input,
               int packet_size,
               int *output,
               int repeats,
               int deltaout)
{

  //The threadID we use is found by combining the treadId, blockId
     , and blockDim.
  int tid = (threadIdx.x + blockIdx.x * blockDim.x);
  //Counters used for processing
  int index=0;
  int findings = 0;

  //printf("%c, %i, %i%, %c, %i, %i, %i, %i\n", pattern_list[tid],
      pattern_offset[tid+1], num_patterns, packet_input[tid],
     packet_size, output[tid], repeats, deltaout);

  //For every character in the input text we run the kernel
  for(int i=0; i<packet_size; i++){
    //The kernel is limited to one pattern each, based on the
       thread id the current kernel has the kernel will loop
    //over the length of the pattern it was given.
    for(int j=pattern_offset[tid]; j<= pattern_offset[tid+1]; j++)
       {
      //Break out of the loop if the characters in the pattern and
          input text is not the same
      if(pattern_list[j] != packet_input[i+index]) break;
      //The character in the pattern and input text matched, we
         increment the index and
      //check the next characters unless the whole patter is now
         matched
      else index++;

      //We found the whole pattern if the index and the length of
         the pattern is the same
```

```
38        if(index == pattern_offset[tid+1] - pattern_offset[tid]){
39
40          ////atomically try to lock the location in the output
               array for this thread only, if not, try next possible
               location
41          //while(atomicCAS(&output[row], -1, tid) != -1){
42          //  //If the location is taken, jump two elements, as
               another thread has also reserved the second array
               element
43          //  row = row+2;
44          //}
45          ////we insert the position of the pattern in the second
               reserved element
46          //atomicExch(&output[row+1], i);
47          //}
48          output[deltaout * tid + findings] = tid;
49          output[deltaout * tid + findings +1] = i;
50
51          findings = findings + 2;
52
53        }
54        }
55      //The current characters of the pattern and input text did not
           match, we reset the index counter.
56      index = 0;
57    }
58    //The whole input text has been processed, the kernel will end
59  }
60
61  #endif
```

Listing A.2: Naïve Kernel

A.1.3 Naïve Search CPU Implementation

```
#ifndef _CPUENGINE_
#define _CPUENGINE_

//This file will hold the Naive string matching algorithm that
    will only run on the CPU
#include "dict.h"
#include <string>

#include <stdio.h>
#include <windows.h>

struct Container{
  string pattern;
  int pos;

  Container(string pattern, int pos){
    this->pattern = pattern;
    this->pos = pos;
  }
};

template<class T>
void runCPUengine(){

  //Create the Dictionary
  Dict dictionary;
  dictionary.genDictionary("badwords.txt");

  char * packet = "This is a fake network packet containing some
     bad words such as shit and asshole (assh0le), all these
     bitchy words will be detected, and there should  be 5 bad
     words, however it will find prefixes of larger words that are
      also other words in the list, such as ass.";
  int packet_size = strlen(packet);
  int index = 0;
  vector<Container> out_data;

  //To time the exection
  clock_t start = clock();

  for(int j=0; j< dictionary.c_list.size(); j++){
    for(int i = 0; i< packet_size; i++){
      for(int k =0; k < strlen(dictionary.c_list[j]); k++){

        if(dictionary.c_list[j][index] != packet[i+index]) break;
        else index++;
```

```
42
43
44        if(index == strlen(dictionary.c_list[j]))
45          out_data.push_back(Container(string(dictionary.c_list[j]),
                i));
46      }
47        index=0;
48      }
49
50    }
51    //Print the time for the algorithm execution
52    printf("Time elapsed for CPU engine: %3.1f ms\n", ((double)clock
        () - start));
53
54    //Print the findings
55    for(int i = 0; i < out_data.size(); i++){
56      cout << out_data[i].pattern << " at pos: " << out_data[i].pos
          << endl;
57    }
58  }
59
60  #endif
```

Listing A.3: Naïve Kernel

A.2 Experiment #2: Knuth-Morris-Phatt (KMP)

A.2.1 KMP Engine

```
#include "cuda_runtime.h"
#include "dictionary.cuh"
#include "trie.cu"
#include "monitor.cu"

#include "packetInput.cu"
#include "outputHandler.cu"

#include "kmpKernel.cu"

#include "kmpPreProc.cu"

template<class T>
void kmpEngine(int device){


  ofstream myfile ("kmpGPU.data", ios::trunc);
  //myfile << "[KMP]" << endl;

  if(!myfile.is_open()) return;
  else{

  HandleError<T>(cudaSetDevice(device), "ahoe", 15);

  int *current = new int;
  HandleError<T>(cudaGetDevice(current), "ahoe", 18);
  std::cout << "Using GPU: " << *current << endl;

  cudaDeviceReset();


  //Information and monitoring
  PM<T> monitor;
  PacketData<T> *packet;

  //Dictionary<T> dictionary("corncob.txt");
  Dictionary<T> dictionary("badwords.txt");

  //load dict to GPU Memory
  dictionary.upload();

  //easy fix use all multicores
  int blocks;
  int threads;
```

```
45
46    blocks = 32;
47    threads = (int)(dictionary.num_patterns/blocks)+1;
48
49    cout << blocks << " & " << threads << endl;
50
51    //Preprocessing for KMP and upload
52    KmpPreProc<T> preproc(dictionary);
53    preproc.upload(dictionary);
54
55    //Run preproc kernel to generate the table for the patterns
56    prekmpKernel<T><<<blocks,threads>>>(dictionary.d_list,
          dictionary.d_offsets, preproc.d_pre_table);
57
58    OutputHandler<T> *output;
59    int repeats = 1000;
60
61    while(true){
62
63      repeats += 1000;
64
65      if (repeats >= 10000) {
66        myfile.close();
67        return;
68      }
69
70      cout << "num packets at once: " << repeats << endl;
71
72
73      //create Packet
74      packet = new PacketData<T>(repeats);
75
76      output = new OutputHandler<T>(repeats, 3);
77
78      //Loop of the NIDS starts here
79      monitor.startPM(); //start the monitor for watching GPU
80
81      //load data to GPU
82      packet->upload();
83      output->upload();
84
85
86      //kmpKernel<T><<<1, 1>>>(dictionary.d_list, dictionary.
            d_offsets, preproc.d_pre_table, packet->d_packet, packet->
            h_packet_size, output->d_output, output->deltaout);
87      kmpKernel<T><<<blocks, threads>>>(dictionary.d_list,
            dictionary.d_offsets, preproc.d_pre_table, packet->d_packet
            , packet->h_packet_size, output->d_output, output->deltaout
            );
```

```
88
89      output->download();
90
91      monitor.stopPM(); //Stop the monitor
92
93      //for(int j=0; j< output->repeats*output->deltaout; j=j+2)
94      //if(output->log_output[j] != -1){
95      //  cout << dictionary.h_list[output->log_output[j]] << "
            found in packet: " << output->log_output[j+1] << endl;
96      //}
97
98      cout << "time: " << monitor.elapsedTime << "ms    "  <<
            (((1000/monitor.elapsedTime) * packet->packet_string_size *
             repeats)/1000000)<<" MB/s    " <<  (((1000/monitor.
            elapsedTime) * packet->packet_string_size * repeats)
            /1000000)*8 << " Mbit/s" << endl;
99
100     myfile.precision(5);
101     myfile.setf(ios::fixed, ios::floatfield);
102     myfile << repeats << "  " << packet->packet_string_size << "
            " <<  monitor.elapsedTime << "  " << (((1000/monitor.
            elapsedTime) * packet->packet_string_size * repeats)
            /1000000) <<endl;
103   }
104   }
105 }
```

Listing A.4: KMP Engine

A.2.2 KMP Preprocessing

```
#ifndef _PREPROC_
#define _PREPROC_

#include "cuda_runtime.h"
#include "dictionary.cuh"
#include "book.cu"

template<class T>
struct KmpPreProc{
  //Space for the preprocessing table
  int * h_pre_table;
  int * d_pre_table;

  KmpPreProc(Dictionary<T> &dict){
    h_pre_table = new int[dict.num_elements];
    for(int i=0; i< dict.num_elements; i++){
      h_pre_table[i] = 0;
    }
  }


  void upload(Dictionary<T> &dict){
    HandleError<T>(cudaMalloc((void**)&d_pre_table, sizeof(int)*
        dict.num_elements), "preproc", 23);
    HandleError<T>(cudaMemcpy(d_pre_table, h_pre_table, sizeof(int
        )*dict.num_elements, cudaMemcpyHostToDevice), "preproc",
        24);
    }

  ~KmpPreProc(){
    HandleError<T>(cudaFree(d_pre_table), "preproc", 28);
    //delete [] h_pre_table;
  }
};

#endif
```

Listing A.5: KMP PreProcessing

A.2.3 KMP Kernel

```
1  #ifndef _KERNEL_KMP_
2  #define _KERNEL_KMP_
3
4  #include "cuda_runtime.h"
5  #include <stdio.h>
6
7  template<class T>
8
9  __global__ void prekmpKernel(char* g_pattern_list,
10                 int *g_pattern_offset,
11                 int * g_pre_table)
12 {
13
14   //The threadID we use is found by combining the treadId, blockId
       , and blockDim.
15   int tid = (threadIdx.x + blockIdx.x * blockDim.x);
16
17   //First create the failure table
18   int suffix_pos = 1;
19   int prefix_pos = 0;
20   //printf("alloc %i\n", tid);
21
22   //Making the code a bit easyer to read
23   int pattern_size = g_pattern_offset[tid+1] - g_pattern_offset[
       tid];
24   int pattern_start = g_pattern_offset[tid];
25
26   //Continue while the suffix position is less then the size of
       the pattern given to the thread
27   while(suffix_pos < pattern_size){
28
29     //If a match is found between the prefix position and the
         suffix position we store the location in the table
30     if(g_pattern_list[pattern_start + prefix_pos] ==
         g_pattern_list[pattern_start + suffix_pos]){
31   //    printf("in if!!!\n");
32       g_pre_table[g_pattern_offset[tid] + suffix_pos] = prefix_pos
           +1;
33       prefix_pos++;
34       suffix_pos++;
35     }
36     //If they do not match, but prefix is over 0, a match was
         found earlier
37     else if(prefix_pos > 0) {
38     prefix_pos = g_pre_table[g_pattern_offset[tid] + prefix_pos
         -1];
```

```
39
40      }
41      //There was no match, so we continue with next char as suffix
42      else{
43        g_pre_table[g_pattern_offset[tid]+suffix_pos] = 0;
44        suffix_pos++;
45      }
46    }
47  }
48
49
50  template<class T>
51      __global__ void kmpKernel(char* g_pattern_list,
52                int *g_pattern_offset,
53                int * g_pre_table,
54                char* g_packet_input,
55                int g_packet_size,
56                int *g_output,
57                int deltaout)
58  {
59    //The threadID we use is found by combining the treadId, blockId
        , and blockDim.
60    int tid = (threadIdx.x + blockIdx.x * blockDim.x);
61
62    int pattern_size = g_pattern_offset[tid+1] - g_pattern_offset[
        tid];
63    int pattern_start = g_pattern_offset[tid];
64
65    int ip = 0; //index pattern
66    int it = 0; //Index text
67    int findings = 0;
68
69
70    while( it <  g_packet_size) {
71
72    if(g_pattern_list[pattern_start+ip] == g_packet_input[it]){
73      //If we find a match we store the findings in the output file
74      if(ip == pattern_size -1){
75          g_output[deltaout * tid + findings] = tid;
76          g_output[deltaout * tid + findings +1] = (it -(
              pattern_size)-1);
77          findings = findings + 2;
78      }
79        ip++;
80        it++;
81      }
82      //If ip is more then 0, a match was found earlier and we go
          back to this location
83      else if(ip > 0){
```

```
84          ip = g_pre_table[g_pattern_offset[tid]+ip-1];
85        }
86        //No new match was found, and we increase the index of the
              input file
87        else{
88          it++;
89        }
90      }
91    }
92
93    #endif
```

Listing A.6: KMP Kernel

A.3 Experiment #2: Aho-Corasick (AC)

A.3.1 Aho-Corasick Engine

```
#include "cuda_runtime.h"
#include "dictionary.cuh"
#include "trie.cu"
#include "monitor.cu"
#include "packetInput.cu"
#include "outputHandler.cu"

#include "ahoKernel.cu"

#include "book.cu"

#include <ostream>


template<class T>
void ahoEngine(int device){

  ofstream myfile ("ahoGPU.data", ios::trunc);
  //myfile << "[GPU]" << endl;

  if(!myfile.is_open()) return;
  else{

  HandleError<T>(cudaSetDevice(device), "ahoe", 15);

  int *current = new int;
  HandleError<T>(cudaGetDevice(current), "ahoe", 18);
  std::cout << "Using GPU: " << *current << endl;

  cudaDeviceReset();


  //Information and monitoring
  PM<T> monitor;
  PacketData<T> *packet;

  Dictionary<T> dictionary("corncob.txt");
  //Dictionary<T> dictionary("badwords.txt");

  //Aho Corasick
  mTrie<T> trie;

  //dictionary.num_patterns
  for(int i=0; i<1000;i++){
```

```
45      trie.addWord(dictionary.h_list[i], i);
46    }
47      //Parse the trie generate Metadata //1 because GPU
48      trie.parseTrie(1);
49
50      OutputHandler<T> *output;
51      int repeats = 8000;
52
53      int loops = 0;
54        while(loops < 10){
55      //  while(true){
56      //    cout << endl;
57      //    cout << endl;
58
59          //repeats += 1000;
60
61          //if (repeats >= 10000) {
62          //  myfile.close();
63          //  return;
64          //}
65
66          cout << "num packets at once: " << repeats << endl;
67
68          //if(repeats == 0){
69          //  myfile.close();
70          //  return;
71          //}
72          //create Packet
73          packet = new PacketData<T>(repeats);
74
75          output = new OutputHandler<T>(repeats, 1);
76
77          //Loop of the NIDS starts here
78          monitor.startPM(); //start the monitor for watching GPU
79
80          //load data to GPU
81          packet->upload();
82          output->upload();
83
84          ahoKernel<T><<<packet->blocksPerGrid, packet->
               threadsPerBlock>>>(trie.d_pattern, trie.d_failure, trie
               .d_output_check, packet->d_packet, packet->
               d_packet_offsets, output->deltaout, packet->repeats,
               output->d_output);
85
86          output->download();
87
88          monitor.stopPM(); //Stop the monitor
89
```

```
90        //  int j=0;
91          //for(int j=0; j< output->size*output->deltaout; j=j+2)
92          //if(output->log_output[j] != -1){
93          //  cout << dictionary.h_list[output->log_output[j+1]] <<
                " found in packet: " << output->log_output[j] << endl;
94          //}
95
96          cout << "time: " << monitor.elapsedTime << "ms      " <<
               (((1000/monitor.elapsedTime) * packet->
               packet_string_size * repeats)/1000000)<<" MB/s" <<
               (((1000/monitor.elapsedTime) * packet->
               packet_string_size * repeats)/1000000)*8 << " Mbit/s"
               << endl;
97
98          myfile.precision(5);
99          myfile.setf(ios::fixed, ios::floatfield);
100         myfile << repeats << "  " << packet->packet_string_size <<
                "  " <<  monitor.elapsedTime << "  " << (((1000/
               monitor.elapsedTime) * packet->packet_string_size *
               repeats)/1000000) <<endl;
101
102       loops++;
103       }
104     myfile.close();
105     return;
106   }
107 }
```

Listing A.7: The Main function

A.3.2 Aho-Corasick Kernel

```
1
2
3
4  template<class T>
5  __global__  void ahoKernel(char *pattern, int* failure, int2 *
       output_check, char *packet, int *packet_offsets, int deltaout,
       int repeats, int * output){
6
7    //printf("pattern: %c, failure: %i, output1: %i, output2: %i,
         packet: %c, outarr: %i \n", pattern[1], failure[1],
         output_check[1].x, output_check[1].y, packet[0], output[1]);
8    int tid = (threadIdx.x + blockIdx.x * blockDim.x);
9
10   int findings = 0;
11   int index = 1; //offset from start in pattern
12   int i = 0;
13
14
15
16   //packet to work on start at packet_offset given by threadID or
         break
17   int start = packet_offsets[tid];
18   int end = packet_offsets[tid+1];
19
20   //Overlapping the packets.
21   //int overlap = 5;
22   //if(start > overlap) start - overlap;
23   //end = end + overlap;
24
25   if(tid+1 > repeats){
26     return;
27   }
28
29   while(start+i <= end){
30
31     if(packet[start+i] != pattern[index]){
32
33       index = failure[index];
34     }
35     else {
36
37       if(output_check[index].x == true) {
38
39         output[deltaout * tid + findings] = tid;
40         output[deltaout * tid + findings +1] = output_check[index
             ].y;
```

```
41
42          findings = findings + 2;
43
44        }
45          i++;
46          index++;
47      }
48
49      if(index == 0) {
50        //__syncthreads();
51        start++;
52        index = 1;
53        i=0;
54      }
55
56    }
57  }
```

Listing A.8: The Kernel

A.3.3 Aho-Corasick Tree Structure

```
#ifndef _CTRIE_
#define _CTRIE_

#include "cuda_runtime.h"
#include <iostream>
#include <vector>
#include <string>

using namespace std;

template<class T>
class mNode {
public:
    mNode() {
    data = ' ';
    leaf = false;
    id=0; }
    ~mNode() {}
    char content() { return data; }
    void setContent(char c) { data = c; }

  bool getLeaf() { return leaf; }
    void setLeaf() { leaf = true; }

  void setId(int x){id = x;}
  int getId(){return id;}

  void setPid(int x){pid = x;}
  int getPid(){return pid;}

    mNode* findChild(char c);
    void appendChild(mNode* child) { mChildren.push_back(child); }
    vector<mNode*> Children() { return mChildren; }

private:
    char data;
    bool leaf;
  int id;
  int pid;
    vector<mNode<T>*> mChildren;
};

template<class T>
class mTrie {
public:
    mTrie();
```

```
47      ~mTrie();
48      void addWord(std::string s, int pid);
49    void parseTrie(bool GPU);
50    void DFS(mNode<T>* child,int fail);
51    int getElements(){return elements;}
52    void setIndex(mNode<T>* child);
53    int * getAlpha();
54
55
56    //GPU
57    char* d_pattern;
58    int *d_failure;
59    int2 *d_output_check;
60
61    //CPU
62    char * cPattern;
63    int * failure;
64    int2 * output;
65
66  private:
67      mNode<T>* root;
68    int elements;
69    std::string pattern;
70    int index;
71    int* root_data;
72
73  };
74
75  template<class T>
76  mNode<T>* mNode<T>::findChild(char c)
77  {
78      for ( int i = 0; i < mChildren.size(); i++ )
79      {
80          mNode* tmp = mChildren.at(i);
81          if ( tmp->content() == c )
82          {
83              return tmp;
84          }
85      }
86
87      return NULL;
88  }
89  template<class T>
90  mTrie<T>::mTrie()
91  {
92      root = new mNode<T>();
93    root->setPid(-1);
94    elements = 0;
95    index = 0;
```

```
96
97
98  }
99  template<class T>
100 mTrie<T>::~mTrie()
101 {
102   cudaFree(d_pattern);
103   cudaFree(d_failure);
104   cudaFree(d_output_check);
105 }
106 template<class T>
107 void mTrie<T>::addWord(std::string s, int pid)
108 {
109
110     mNode<T>* current = root;
111
112     if ( s.length() == 0 )
113     {
114         current->setLeaf(); // an empty word
115         return;
116     }
117
118
119
120
121     for ( int i = 0; i < s.length(); i++ )
122     {
123         mNode<T>* child = current->findChild(s[i]);
124         if ( child != NULL )
125         {
126             current = child;
127         }
128         else
129         {
130             mNode<T>* tmp = new mNode<T>();
131       tmp->setPid(-1);
132             tmp->setContent(s[i]);
133             current->appendChild(tmp);
134             current = tmp;
135       elements++;
136         }
137         if ( i == s.length() - 1 ){
138             current->setLeaf();
139       current->setPid(pid);
140     }
141     }
142 }
143
144 template<class T>
```

```
145 void mTrie<T>::parseTrie(bool GPU){
146
147   cPattern = new char[elements];
148   failure = new int[elements];
149   output = new int2[elements];
150
151   index =0;
152   setIndex(root);
153   index =0;
154   DFS(root, root->getId());
155   strcpy(cPattern, pattern.c_str());
156
157   if(GPU){
158   //Copy the pattern
159   cudaMalloc((void**)&d_pattern, sizeof(char)*elements);
160   cudaMemcpy( d_pattern, cPattern,sizeof(char)*elements,
        cudaMemcpyHostToDevice);
161
162   //Copy the failure
163   cudaMalloc((void**)&d_failure, sizeof(int)*elements);
164   cudaMemcpy(d_failure, failure, sizeof(int)*elements,
        cudaMemcpyHostToDevice);
165
166   //Copy the output
167   cudaMalloc((void**)&d_output_check, sizeof(int2)*elements);
168   cudaMemcpy(d_output_check, output, sizeof(int2)*elements,
        cudaMemcpyHostToDevice);
169   }
170 }
171
172 template<class T>
173 void mTrie<T>::DFS(mNode<T>* child, int fail){
174   pattern += child->content();
175   failure[index] = fail;
176   output[index].x = child->getLeaf();
177   output[index].y = child->getPid();
178   index++;
179   if(child->Children().size() == 0) {//this is the leaf
180   }
181   else if(child->Children().size() == 1) DFS(child->Children().at
        (0), fail);
182   else{
183     for(int i=0; i< child->Children().size(); i++){
184       if(i <  child->Children().size()-1)
185         DFS(child->Children().at(i), child->Children().at(i+1)->
              getId());
186       else
187         DFS(child->Children().at(i), child->getId());
188     }
```

```
189   }
190 }
191
192 template<class T>
193 int * mTrie<T>::getAlpha(){
194
195   root_data = new int[root->Children().size()];
196
197   for(int i=0; i < root->Children().size(); i++){
198     root_data[i] = root->Children().at(i)->getId();
199   }
200
201   return root_data;
202 }
203
204 template<class T>
205 void mTrie<T>::setIndex(mNode<T>* child){
206
207   child->setId(index);
208   index++;
209
210   for(int i=0; i< child->Children().size(); i++){
211   setIndex(child->Children().at(i));
212   }
213 }
214
215
216
217 #endif
```

Listing A.9: The Tree Structure

A.3.4 Aho-Corasick CPU Implementation

```
#include "cuda_runtime.h"
#include "dictionary.cuh"
#include <string>
#include "trie.cu"
#include <ostream>

#include <Windows.h>
using namespace std;

template<class T>
void ahoCPUEngine(){

  ////Windows get processor count
  SYSTEM_INFO sysinfo;
  GetSystemInfo(&sysinfo);
  int numCPU = sysinfo.dwNumberOfProcessors;

  //Timing high resolution
  LARGE_INTEGER t1, t2, frequency;
  double elapsedTime;

  QueryPerformanceFrequency(&frequency);

  int delta = 1000;

  //Open an output file
  ofstream myfile ("ahoCPU.data", ios::trunc);

  //myfile << "packets  strlen  timeused  MB/s" << endl;
  //myfile << "[CPU]" << endl;


  //If we can open the file
  if(!myfile.is_open()) return;
  else{

    //Create the dictionary of words for the trie
    Dictionary<T> dictionary("badwords.txt");
    //Dictionary<T> dictionary("corncob.txt");

    //Generate trie
    mTrie<T> trie;

    //dictionary.num_patterns
    //Addwords
    for(int i=0; i<1000;i++){
```

```
47      trie.addWord(dictionary.h_list[i], i);
48      }
49
50      //Create Metadata //0 because CPU
51      trie.parseTrie(0);
52
53
54      //const string packet_string = "This is a fake network packet
            containing some bad words such as shit and asshole or (
            assh0le), all these bitchy words will be detected, and
            there should  be 7 results, it will find parts of larger
            words that are prefix of other words in the list aswell,
            such as ass.";
55      const string packet_string = "This is a fake network packet
            containing some bad words such as shit";
56
57
58      string mstring;
59      int * output;
60      int repeats = 0;
61
62      while(true){
63
64        repeats += 1000;
65
66        if (repeats >= 10000) {
67          myfile.close();
68          return;
69        }
70
71        cout << "num packets at once: " << repeats << endl;
72
73        mstring = " ";
74        for(int i=0; i< repeats; i++){
75        mstring += packet_string;
76        }
77
78        output = NULL;
79        output = new int[repeats * delta];
80
81        for(int i=0; i< repeats*delta; i++){
82          output[i] = -1;
83        }
84
85        //Start the counter to make it similar to GPU version
86        QueryPerformanceCounter(&t1);
87
88        int index = 0;
89        int start = 1;
```

```
90        int i=0;
91        int findings = 0;
92
93        while((start+i) < mstring.size()){
94
95          if(mstring[start + i] != trie.cPattern[index]){
96
97            index = trie.failure[index];
98          }
99          else {
100           if(trie.output[index].x == true) {
101
102
103             output[findings] = (start+i); //which packet
104             output[findings +1] = trie.output[index].y; //which
                  signature
105             findings = findings + 2;
106
107           }
108             i++;
109             index++;
110           }
111
112           if(index == 0) {
113             start++;
114             index = 1;
115             i=0;
116           }
117
118         }
119
120       QueryPerformanceCounter(&t2);
121
122     //int j=0;
123     //for(int j=0; j< repeats*delta; j=j+2){
124     //  if(output[j] != -1){
125     //    cout << dictionary.h_list[output[j+1]] << " found in
            packet: " << output[j]/ (int)packet_string.size() +1 << "
            at location: " << output[j]% (int)packet_string.size() <<
            endl;
126     //    }
127     //  }
128     elapsedTime = (t2.QuadPart - t1.QuadPart) * 1000.0 / frequency
            .QuadPart;
129     cout << "Elapsed Time: " << elapsedTime << " ms." << "\
            nProcessors on system: " << numCPU << ".\nTheoretical
            optimal threaded performance: " << elapsedTime/numCPU << "
            ms." << endl;
130
```

```
131     myfile.precision(5);
132     myfile<< repeats << " " << packet_string.size() << "   " <<
            elapsedTime << "   " << (((1000/elapsedTime) * packet_string
            .size() * repeats)/1000000)<<endl;
133     }
134   }
135 }
```

Listing A.10: CPU Implementation of Aho-Corasick

A.4 Experiment - Common Functions

A.4.1 Dictionary Structure

```
#ifndef _DICT_
#define _DICT_

//#include "cuda_runtime.h"
#include <vector>
#include <iostream>
#include <fstream>
#include <string>
#include <list>

#include "book.cu"

using namespace std;

template<class T>
struct Dictionary{
  int dicsize;
  char *list;
  char * d_list;
  int * offsets;
  int * d_offsets;
  int max_len;
  int num_patterns;
  vector<string> h_list;
  int num_elements;

  Dictionary(const char * filename){

    d_list = NULL;
    d_offsets = NULL;

    max_len =0;
    num_patterns =0;
    string temp, line;
    int i =0;
    int curr_offset =0;

    ifstream myfile (filename);

      if (myfile.is_open())
        {
          string size;
          getline (myfile, size);
          dicsize = atoi(size.c_str());
```

```
45            offsets = new int[dicsize];
46            offsets[0]=0; //set first line
47
48          while ( myfile.good() )
49          {
50            getline (myfile, line);
51            h_list.push_back(line);
52            temp +=  line.c_str();
53            curr_offset +=line.size();
54            offsets[i+1] = curr_offset;
55            if (max_len < line.size()) max_len = line.size();
56              i++;
57            }
58          myfile.close();
59          }
60        else{
61          cout << "cant open file:" << filename << endl;
62          system("pause");
63          return;
64        }
65
66        num_elements = temp.size()+1;
67        list = new char[num_elements];
68        strcpy(list, temp.c_str());
69        num_patterns=i;
70    }
71
72    void upload(){
73        HandleError<T>(cudaMalloc((void**)&d_list, strlen(list)*
              sizeof(char)), "dict", 29);
74        HandleError<T>(cudaMemcpyAsync(d_list, list, strlen(list)*
              sizeof(char), cudaMemcpyHostToDevice), "dict", 52);
75
76        HandleError<T>(cudaMalloc((void**)&d_offsets, sizeof(int)*
              dicsize), "dict", 74);
77        HandleError<T>(cudaMemcpyAsync(d_offsets, offsets, sizeof(
              int)*dicsize, cudaMemcpyHostToDevice), "dict", 75);
78    }
79
80    ~Dictionary(){
81      if(d_offsets != NULL) HandleError<T>(cudaFree(d_offsets), "
           dict", 81);
82      if(d_list != NULL)HandleError<T>(cudaFree(d_list), "dict", 81)
           ;
83      //delete [] offsets;
84      //delete [] list;
85    }
86  };
87  #endif
```

Listing A.11: The Dictionary Structure

A.4.2 Output Handler

```
#ifndef _OUTPUTHANDLER_
#define _OUTPUTHANDLER_

#include "cuda_runtime.h"

#include "book.cu"

template<class T>
struct OutputHandler{

  int repeats;
  int *h_output; //CPU
  int *d_output; //Device
  int *log_output;
  int deltaout;

  OutputHandler(int repeats, int delta){


    deltaout = delta;
    this->repeats = repeats;

    //h_output = realloc(h_output, repeats*deltaout);
    h_output = new int[repeats*deltaout];
    //log_output = realloc(log_output, repeats*deltaout);
    log_output = new int[repeats *deltaout];

    //initialize output set every char to -1.
    for(int i=0; i< repeats*deltaout; i++){
      h_output[i] = -1;
    }
    HandleError<T>(cudaMalloc((void**)&d_output, sizeof(int)*
       repeats*deltaout), "out", 29);
  }

  void upload(){
  //Malloc and Copy the output-list to the GPU-
  HandleError<T>(cudaMemcpyAsync(d_output, h_output, sizeof(int)*
     repeats*deltaout, cudaMemcpyHostToDevice), "out", 34);
  }


  void download(){
    HandleError<T>(cudaMemcpyAsync(log_output, d_output, sizeof(
       int)*repeats*deltaout, cudaMemcpyDeviceToHost), "out", 39);
  }
```

```
44
45    ~OutputHandler(){
46      //delete h_output;
47      //delete log_output;
48      HandleError<T>(cudaFree(d_output), "out", 44);
49
50    }
51
52  };
53
54  #endif
```

Listing A.12: The Output Handler

A.4.3 Packet Handler

```
#ifndef _PACKETDATA_
#define _PACKETDATA_

#include "cuda_runtime.h"
#include <algorithm>
#include <math.h>
#include <xutility>
#include <string>
#include "book.cu"

//const string packet_string = "This is a fake network packet containing some bad words such as shit and asshole or (assh0le), all these bitchy words will be detected, and there should be 7 results, it will find parts of larger words that are prefix of other words in the list aswell, such as ass.";
//const string packet_string = "This is a fake network packet containing some bad words such as shit and asshole";
const string packet_string = "This is a fake network packet containing bad words such as shit";
template <class T>
struct PacketData{

  int packet_string_size;



  char * h_packet;
  char * d_packet;
  int  h_packet_size;
  int num_threads;
  int  * packet_offsets;
  int * d_packet_offsets;
  int repeats;
  int threadsPerBlock;
  int blocksPerGrid;

  PacketData(int repeats){


    packet_string_size = packet_string.size();

    this->repeats = repeats;
    packet_offsets = new int[repeats+1];

    string mstring;
    packet_offsets[0] = 0;
```

```
41      for(int i=0; i< repeats; i++){
42      mstring += packet_string;
43      packet_offsets[i+1] = mstring.size();
44
45      }
46
47      h_packet_size = mstring.size();
48      h_packet = new char[h_packet_size];
49      strcpy(h_packet ,mstring.c_str());
50
51      num_threads = repeats;
52
53
54      threadsPerBlock = min(512, num_threads);
55      blocksPerGrid = (num_threads + threadsPerBlock-1) /
            threadsPerBlock;
56
57
58      HandleError<T>(cudaMalloc((void**) &d_packet, h_packet_size*
            sizeof(char)), "packet", 46);
59      HandleError<T>(cudaMalloc((void**) &d_packet_offsets, sizeof(
            int)*(repeats+1)), "packet", 47);
60    }
61
62    void upload(){
63    //Copy the packet
64      HandleError<T>(cudaMemcpyAsync(d_packet, h_packet,
            h_packet_size*sizeof(char), cudaMemcpyHostToDevice), "
            packet", 52);
65      HandleError<T>(cudaMemcpyAsync(d_packet_offsets,
            packet_offsets, sizeof(int)*(repeats+1),
            cudaMemcpyHostToDevice), "packet", 53);
66    }
67
68
69    ~PacketData(){
70      HandleError<T>(cudaFree(d_packet), "packet", 57);
71      HandleError<T>(cudaFree(d_packet_offsets), "packet", 58);
72      delete h_packet;
73      delete packet_offsets;
74    }
75  };
76
77  #endif
```

Listing A.13: The Packet Handler

A.4.4 Monitoring

```
#ifndef _MONITOR_CUH_
#define _MONITOR_CUH_
#include "cuda_runtime.h"

#include <stdio.h>
#include<iostream>
#include<string>

using namespace std;

template<class T>
struct PM{

  cudaEvent_t start, stop;
  float elapsedTime;

  void startPM(){

    //Capture start and end time to see performance.

    cudaEventCreate(&start);
    cudaEventCreate(&stop);
    cudaEventRecord(start, 0);
  }

  void stopPM() {

    cudaEventRecord(stop, 0);
    cudaEventSynchronize(stop);

    cudaEventElapsedTime(&elapsedTime, start, stop);

    //printf("Time used: %3.1f ms\n", elapsedTime);

    cudaEventDestroy(start);
    cudaEventDestroy(stop);
  }


  void devInfo(){


  cudaDeviceProp prop;

  int count;

  cudaGetDeviceCount(&count);
  printf("Number of CUDA devices on this system: %d\n", count);
```

```
49
50
51    for(int i= 0; i<count; i++){
52      cudaGetDeviceProperties(&prop, i);
53      printf("CUDA Information for device %d \n", i+1);
54      printf("Name: %s\n", prop.name);
55      printf("Compute capability: %d.%d\n", prop.major, prop.minor);
56      printf("Clock rate: %d\n", prop.clockRate);
57      printf("Device copy overlap: ");
58
59      if(prop.deviceOverlap)
60        printf("Enabled\n");
61      else
62        printf("Disabled\n");
63
64    printf("Execution timeout: ");
65    if(prop.kernelExecTimeoutEnabled)
66      printf("Enabled\n");
67    else
68        printf("Disabled\n");
69
70      printf("Memory information for device %d \n", i+1);
71      printf("Total global memory: %ld\n", prop.totalGlobalMem);
72      printf("Total constant memory: %ld\n", prop.totalConstMem);
73      printf("Max memory pitch: %ld\n", prop.memPitch);
74      printf("Texture alignment: %ld\n", prop.textureAlignment);
75      printf("Multi processor count: %d\n", prop.multiProcessorCount
          );
76      printf("Shared mem per mp: %ld\n", prop.sharedMemPerBlock);
77      printf("Registers per mb: %d\n", prop.regsPerBlock);
78      printf("Threads in warp: %d\n", prop.warpSize);
79      printf("Max threads per block: %d\n", prop.maxThreadsPerBlock)
          ;
80      printf("Max thread dimensions: (%d, %d, %d)\n", prop.
          maxThreadsDim[0], prop.maxThreadsDim[1], prop.maxThreadsDim
          [2]);
81      printf("<<<<<<<<<<End of device %d>>>>>>>>>>\n", i+1);
82      cout << "." << endl;
83      cout << ".." << endl;
84      cout << "..." << endl;
85      }
86
87
88      printf("<<<<<<<<<<STARTING PROGRAM>>>>>>>>>>\n");
89      cout << "." << endl;
90      cout << ".." << endl;
91      cout << "..." << endl;
92    }
93  };
```

```
94
95 #endif
```

Listing A.14: Monitoring