Spring 2017

Network Traffic Anomaly-Detection Framework Using GPUs

Meera Ramesh
San Jose State University

NETWORK TRAFFIC ANOMALY-DETECTION FRAMEWORK USING GPUS

A Thesis

Presented to

The Faculty of the Department of Computer Engineering

San José State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

Meera Ramesh

May 2017

The Designated Thesis Committee Approves the Thesis Titled

NETWORK TRAFFIC ANOMALY-DETECTION FRAMEWORK USING GPUS

by

Meera Ramesh

APPROVED FOR THE DEPARTMENT OF COMPUTER ENGINEERING

SAN JOSÉ STATE UNIVERSITY

May 2017

Dr. Hyeran Jeon	Department of Computer Engineering
Dr. Younghee Park	Department of Computer Engineering
Dr. Xiao Su	Department of Computer Engineering

ABSTRACT

NETWORK TRAFFIC ANOMALY-DETECTION FRAMEWORK USING GPUS

by Meera Ramesh

Network security has been very crucial for the software industry. Deep packet inspection (DPI) is one of the widely used approaches in enforcing network security. Due to the high volume of network traffic, it is challenging to achieve high performance for DPI in real time. In this thesis, a new DPI framework is presented that accelerates packet header checking and payload inspection on graphics processing units (GPUs). Various optimizations were applied to GPU-version packet inspection, such as thread-level and block-level packet assignment, warp divergence elimination, and memory transfer optimization using pinned memory and shared memory. The performance of the pattern-matching algorithms used for DPI was analyzed by using an assorted set of characteristics such as pipeline stalls, shared memory efficiency, warp efficiency, issue slot utilization, and cache hits. The extensive characterization of the algorithms on the GPU architecture and the performance comparison among parallel pattern-matching algorithms on both the GPU and the CPU are the unique contributions of this thesis. Among the GPU-version algorithms, the Aho-Corasick algorithm and the Wu-Manber algorithm outperformed the Rabin-Karp algorithm because the Aho-Corasick and the Wu-Manber algorithms were executed only once for multiple signatures by using the tables generated before the searching phase was begun. According to my evaluation on a NVIDIA K80 GPU, the GPU-accelerated packet processing achieved at least 60 times better performance than CPU-version processing.

ACKNOWLEDGMENTS

It is with immense gratitude that I acknowledge the support of my advisor Dr. Hyeran Jeon throughout the thesis. Prof. Hyeran is one of the best Professors I've met and I've learned many things from her, the most important quality I cultivated because of her is patience. I would also like to thank my co-advisor Dr. Younghee Park for monitoring the progress of the project and for providing me good ideas to solve certain issues I had faced. I am also very thankful to Dr. Xiao Su for being a part of the committee and for providing her advice on the future scope of the thesis.

I am indebted to my parents and my brother for instilling confidence in me. I'm thankful to my in-laws for their constant support and motivation. I'm also thankful to my friend Sindhuja, who gave her ear to my problems.

This thesis would have remained a dream had it not been for the support from my husband Karthik, and I owe my deepest gratitude to him.

TABLE OF CONTENTS

List of Tables ix

List of Figures x

1 Introduction 1

2 Background 3

2.1 Network Intrusion Detection System (NIDS) 3

2.2 GPU Architecture 4

2.2.1 CUDA Programming Environment 5

2.2.2 Pinned Memory 6

2.3 CUDA Thread Execution Model 7

2.3.1 Thread Synchronization 7

2.3.2 Thread Assignment 8

2.3.3 Thread Divergence 9

2.4 Signature Matching 10

2.4.1 Rabin-Karp Algorithm 10

2.4.2 Aho-Corasick Algorithm 13

2.4.3 Wu-Manber Algorithm 15

2.5 Parallelism in CPU Programming with OpenMP 17

3 System Organization 19

3.1 Memory Transfer of Packets to the GPU 19

3.2 Packet Capture Class 20

3.3 Analysis 20

4 Parallel IDS Design on GPUs 22
4.1 Overview of Parallelism Approaches 22
4.1.1 Thread-Level Parallelism 22
4.1.2 Block-Level Parallelism 23
4.2 Header Checking 24
4.3 Parallel Pattern-Matching Algorithms using CUDA 24
4.3.1 Rabin-Karp Algorithm 24
4.3.2 Aho-Corasick Algorithm 25
4.3.3 Wu-Manber Algorithm 26
4.4 Utilization of Pinned Memory 27
5 Implementation 29
5.1 Packet Capture and Transfer to the GPU 29
5.1.1 Capturing the Network Packets 29
5.1.2 Buffer the Network Packets 29
5.2 Dissectors 30
5.2.1 PreAnalyzerDissector 32
6 Evaluation 33
6.1 Block-Level Parallelism vs. Thread-Level Parallelism 33
6.2 Comparison between OpenMP, CPU and GPU solutions 34
6.3 Stall Breakdown 36
6.4 Resource Utilization 39
6.4.1 Memory Utilization 39
6.4.2 Warp Utilization 41

6.4.3 SM Utilization 44
6.5 Cache Hit Rate 45
6.6 Pinned Memory Efficiency 46
7 Related Work 48
8 Conclusion 50
References 52
Appendix
Code snippets 55
A.1 Header Checking in Block-Level Parallelism 55
A.2 Rabin-Karp Algorithm 58
A.2.1 Sequential Pattern-Matching Algorithm using C 58
A.2.2 Parallel Pattern-Matching Algorithm using CUDA 60
A.3 Wu-Manber Algorithm 61
A.3.1 Parallel Pattern-Matching Algorithm using OpenMP 61
A.3.2 Parallel Pattern-Matching Algorithm using CUDA 64
A.4 Aho-Corasick algorithm 65
A.4.1 Sequential Pattern-Matching Algorithm using C 65
A.4.2 Parallel Pattern-Matching Algorithm using OpenMP 71

List of Tables

Table 1 Hash Value Calculation Example (Pattern = 59372) 11

Table 2 Hash Value Calculation Example with Horner's Method 12

Table 3 FSM Transition Example (Pattern = "ushers") 15

Table 4 Shift Table Generated after Preprocessing 16

Table 5 Prefix Table Generated after Preprocessing 16

Table 6 Multi-Pattern Search on "ANDYMEETMESOONATGOOGLE" Using Pre-processed Shift and Prefix Tables 17

Table 7 Execution Times on the GPU 34

Table 8 Performance of Pattern-Matching Algorithms 35

Table 9 Comparison of Execution Times for the Wu-Manber Algorithm . . . 35

Table 10 Pipeline Stall Breakdown 38

List of Figures

Figure 1 2D grid with 6 thread blocks, each having 4 threads 6

Figure 2 Data transfer from the CPU to the GPU using pinned memory 7

Figure 3 A single warp scheduler unit 9

Figure 4 Thread divergence 10

Figure 5 Trie is constructed for the patterns = he, she, hers, his 14

Figure 6 Trie is extended to include the failure transitions 14

Figure 7 Example for an OpenMP program in C 18

Figure 8 System organization 19

Figure 9 Thread-level parallelism 22

Figure 10 Block-level parallelism 23

Figure 11 Parallel implementation of the Rabin-Karp algorithm 25

Figure 12 Parallel implementation of the Aho-Corasick algorithm 26

Figure 13 Calculating the hash value of the suffix 26

Figure 14 Pattern is searched based on the hash value of the prefix 27

Figure 15 Using non-pinned memory for data transfer 28

Figure 16 Using pinned memory for data transfer 28

Figure 17 Pseudo code to store the packet buffer that will be copied to the GPU 29

Figure 18 The array of packets of "Max Buffer" size 30

Figure 19 Dissect methods and targeted OSI layer methods 31

Figure 20 Execution times for packet processing 33

Figure 21 Speedup of the algorithms on the GPU over the CPU 36

Figure 22 Breakdown for the issue stall reasons 37

Figure 23 Shared memory load and store transactions 40
Figure 24 Shared memory replays due to bank conflicts 40
Figure 25 Average number of active threads executed in a warp 41
Figure 26 Branch divergence in the Wu-Manber algorithm 42
Figure 27 Branch divergence in the Wu-Manber algorithm 42
Figure 28 Branch divergence in the Aho-Corasick algorithm 42
Figure 29 Branch divergence in the Aho-Corasick algorithm 42
Figure 30 Branch divergence in the Rabin-Karp algorithm 43
Figure 31 The number of instructions executed per warp 44
Figure 32 SM efficiency 44
Figure 33 L2 hit rate 46
Figure 34 Pinned memory efficiency 47

CHAPTER 1

Introduction

Network intrusion detection systems (IDSs) such as firewalls capture malicious activities and drop malicious packets that intend to attack a network. These IDSs report the malicious activity to an administrator, and the administrator prevents the packets from moving across the network. There are two types of IDSs, host-based IDSs and network-based IDSs. Host-based IDSs monitor the operating system to see if it is being attacked, while network-based IDSs monitor network traffic. Network-based IDSs can detect malicious traffic in two ways by comparing against 1) malicious signatures and 2) reference traffic models. In this thesis, a new network-based IDS design is proposed that uses malicious signature analysis. To detect malicious traffic, individual packet contents were compared against well-known malicious packet signatures.

To identify the existence of malicious signatures in the packet payload, various pattern-matching algorithms were used. Pattern matching is a computationally intensive task that accounts for up to 75% of the execution time of IDSs [6]. There is no dependency between instructions in signature matching. The text can be split into multiple chunks, and each chunk can be processed individually to search for patterns. This can be done in parallel with multiple threads. Thus, the massive parallelism of the GPU can be exploited in signature matching. A few studies have used various accelerators for DPI such as field programmable gate arrays (FPGAs) [13][24][27] and graphics processing units (GPUs) [12][25] for better performance. Hardware-based approaches using FPGAs may perform better than software approaches by optimizing the hardware pipeline dedicated to packet processing. However, they are not flexible enough to apply new signatures. Many researchers have used GPUs in various network security applications, including Gnort [10] (a GPU-version of Snort [11]) and Snap [9] (a GPU version of clickOS [7]).

In this study, I analyzed various approaches used to perform string matching on the GPU. I demonstrated the performance improvement by parallelizing the packet processing with multiple threads. I used various well-known signature-matching algorithms such as naive algorithm, the Rabin-Karp algorithm, the Aho-Corasick algorithm, and the Wu-Manber algorithm. The evaluation results were obtained by running single-pattern matching as well as multi-pattern matching on the GPU.

The proposed IDS consists of two parts, packet header processing and payload signature matching. Various rules were applied to check the integrity of the headers of various protocols such as internet protocol (IPv4) and transmission control protocol (TCP). Various optimizations were applied such as eliminating warp divergence, using pinned memory and using shared memory. To understand the performance difference of these algorithms, I characterized the architectural behavior of the algorithms by using two profilers, nvidia-smi and nvprof.

CHAPTER 2

Background

In this chapter, an introduction to the GPU architecture, CUDA programming environment, and network IDS is presented.

2.1 Network Intrusion Detection System (NIDS)

Network IDSs collect the packets that pass through the network and check if the packets are legitimate. When an IDS identifies any suspicious activity, it either logs the activity or alerts the network administrator about the activity. An example of NIDS is Snort [11]. Snort uses thousands of signatures to detect malicious activities in the packets that are flowing across the network. There are two types of NIDS, signature-based NIDS and machine learning-based NIDS. The drawback of signature-based analysis is the fact that it only can be used for known attacks.

The function of an IDS is partitioned into two parts, header checking and payload checking. Header checking evaluates the validity of the IP addresses and detects if the packet uses any malicious TCP flag bit combination. Payload checking investigates the packet payload for inclusion of any known malicious virus patterns. Pattern-matching algorithms can be classified as single-pattern matching or multi-pattern matching. With single-pattern matching, the entire payload is searched to check if the given pattern exists. If there are multiple patterns, the time complexity is the product of the number of patterns and the length of the longest pattern because the same algorithm should be applied to individual patterns. A finite state machine (FSM, also known as a deterministic finite automaton or DFA) is a way of representing a set of patterns. When a DFA is used for multi-pattern matching, the runtime complexity is independent of the number of patterns and the length of the longest pattern.

2.2 GPU Architecture

NVIDIA Tesla K80 GPU was used for my research. It offers GPU programming capability by providing a compute unified device architecture (CUDA) SDK [5]. The GPU is composed of streaming multiprocessors (SMs), and each SM is composed of many streaming processors (SPs). In Tesla K80, there are 13 SMs and 192 SPs per SM with compute capability 3.7. Each SP has fully pipelined execution units for floating-point and integer arithmetic operations.

The main function of the GPU, namely "kernel function," is launched with the parameters specifying the total number of threads and thread blocks required for the execution. Each SP processes one thread's task, and each SM executes one or more thread blocks. The thread blocks do not migrate to other SMs.

The thread blocks are divided into warps. A warp consists of 32 threads, and all the threads in a warp execute the same instruction. Each thread in a warp executes in single instruction multiple data (SIMD) mode, which means all the threads execute the same instruction but with different data. In Tesla K80, we can assign up to 2048 threads per SM and 1024 threads per block. Since there are 13 SMs, a maximum of 26624 threads can be launched on the GPU.

There are different types of memory in the GPU: global memory, shared memory, constant memory, and registers. Shared memory is an on-chip memory that is faster than off-chip memory [8]. NVIDIA K80 has 11.25 GB global memory, 64 KB constant memory, 48 KB shared memory/thread block and 64K registers/thread block. Global memory can be accessed by the CPU and by all of the threads executing the kernel. Shared memory is shared between all of the threads in the same thread block. Shared memory has 32 banks, and each bank is sectioned into 32-bit words (4 bytes). Every bank can service only one request per cycle. Therefore, if multiple requests are given to the same bank, there will be a bank conflict [2].

In order to avoid bank conflicts, individual threads within a warp should fetch data from different banks. When there is no bank conflict, shared memory is as fast as the register file. Each thread has its own set of registers. The register file is the fastest on-chip memory in the GPU. However, if a kernel's register usage exceeds the register file capacity, performance can be limited. In this case, the data that cannot be stored in the register file are spilled to local memory. Local memory is an off-chip memory that has similar access latency as global memory. The compiler saves the automatic variables in local memory when there is not enough space in the register file. Large structures are typically stored in local memory. Thanks to the shorter access time, a widely used optimization technique in CUDA programming is to store frequently accessed data in shared memory and in the register file (rather than global memory) [14].

2.2.1 CUDA Programming Environment

CUDA C is an extension of standard C [5]. CUDA C provides APIs that support data transfer and kernel invocation between the CPU and the GPU. When kernels are called, they are executed N times in parallel by N CUDA threads, unlike the C functions, which are executed only once. A thread block has multiple dimensions of threads, up to three dimensions. With the multi-dimensional thread block, complex data structures, such as matrices and cubes, can be easily parallelized.

The threads in a thread block run on the same SM by sharing the memory resources provided by the SM. Hence, there is a limit on the number of threads per thread block, which is either 512 or 1024 on the GPUs that have compute capability ≥ 2.0 [5]. The kernel is executed by multiple thread blocks of equal size. Therefore, the total number of threads executing a kernel is equal to the number of threads per block multiplied by the number of blocks. The blocks are grouped together into a one- to three-dimensional grid of thread blocks as shown in Fig. 1.

If the GPU has compute capability ≥ 2.0, then the three-dimensional grid is supported.

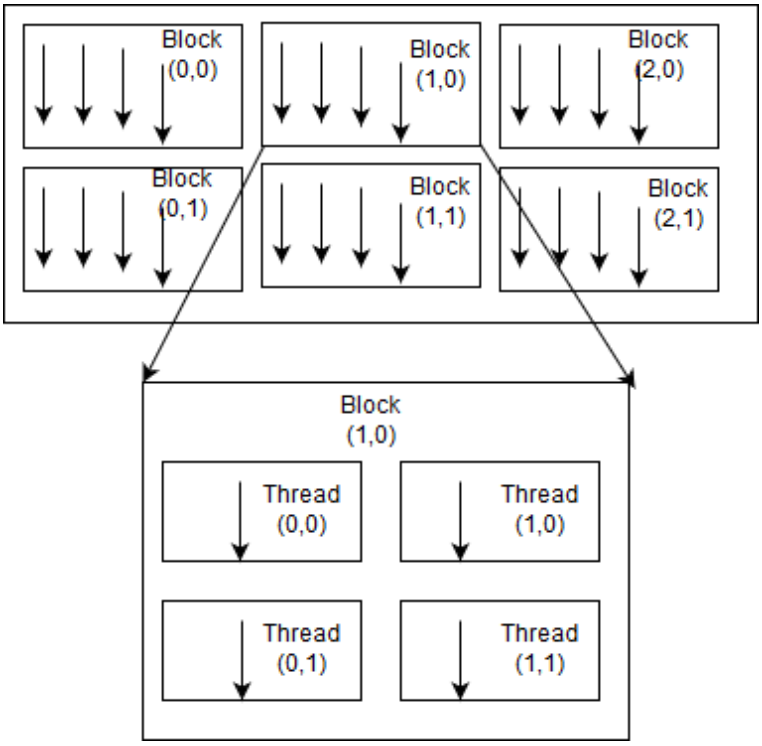

Figure 1: 2D grid with 6 thread blocks, each having 4 threads

2.2.2 Pinned Memory

Pinned memory can be accessed by the GPU and it can be read or written with a higher bandwidth when compared to pageable host memory. The memory allocation function malloc() allocates host memory on the heap, which is pageable by default. The GPU cannot access data directly from pageable host memory. Therefore, when a data is transferred from pageable host memory to device memory, the CUDA driver must first allocate a temporary page-locked, or pinned, host array, copy the host data to the pinned array, and then transfer the data from the pinned array to device memory. As shown in Fig. 2, pinned memory is used as a temporary buffer when transferring data between the CPU and the GPU. Data transfer to the temporary buffer can be avoided by allocating data on pinned memory.

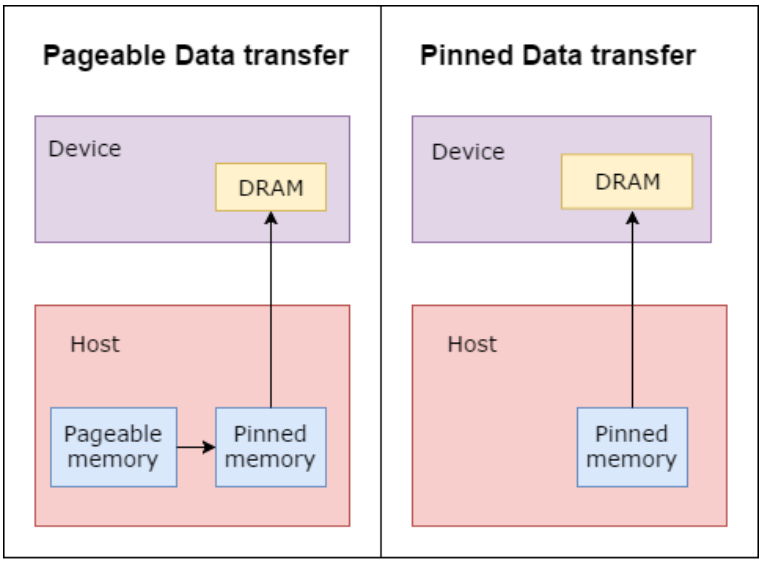

Figure 2: Data transfer from the CPU to the GPU using pinned memory

2.3 CUDA Thread Execution Model

The built-in variables blockDim and gridDim are used for specifying the number of threads in a block and the number of blocks in a grid [5]. A thread block is identified by blockIdx and a thread in a thread block is identified by threadIdx. The variables blockIdx and threadIdx have member variables that represent the x, y and z components. For a one-dimensional kernel, threadIdx.x uniquely identifies a thread within a thread block and blockIdx.x uniquely identifies a block within a grid.

2.3.1 Thread Synchronization

Threads can be synchronized only across threads in a thread block by executing $syncthreads()$ function, but not across threads in a grid [5]. There is no synchronization between thread blocks because thread blocks should execute independently on any SM without having to wait for other thread blocks. This allows CUDA applications to scale well with more SMs as thread blocks can be executed concurrently on multiple SMs.

2.3.2 Thread Assignment

During a kernel invocation, CUDA runtime assigns thread blocks to the SMs in the device [5]. The programmer should ensure there are enough registers, shared memory and threads to execute the blocks. If resources are insufficient, the runtime will assign fewer thread blocks to the SM and the occupancy would decrease.

The total number of blocks that can be concurrently executed on an SM depends on the GPU model. Kepler architecture has a total of 13 SMs where each SM can execute up to 16 thread blocks [5]. Thus, a Kepler GPU can run a total of 208 thread blocks concurrently. As each thread block in Kepler architecture can use up to 1024 threads, a total of 212,992 threads can concurrently run on one Kepler GPU.

An SM schedules threads in a group of 32 threads, which is referred to as a warp. In Kepler architecture, each SM has a quad-warp scheduler that selects four warps and dispatches two instructions from each warp every cycle. As shown in Fig. 3, Kepler's warp scheduler selects four warps, and two independent instructions per warp can be dispatched each cycle [30].

In the Kepler architecture, up to 512 warps can be assigned to each SM. However, in each clock cycle, only four warps out of the 512 warps are scheduled on an SM. Global memory accesses take over 400 cycles and can cause pipeline stalls. To hide the memory-access latency, instructions from other warps are issued to an SM while the previous warp is waiting for its data.

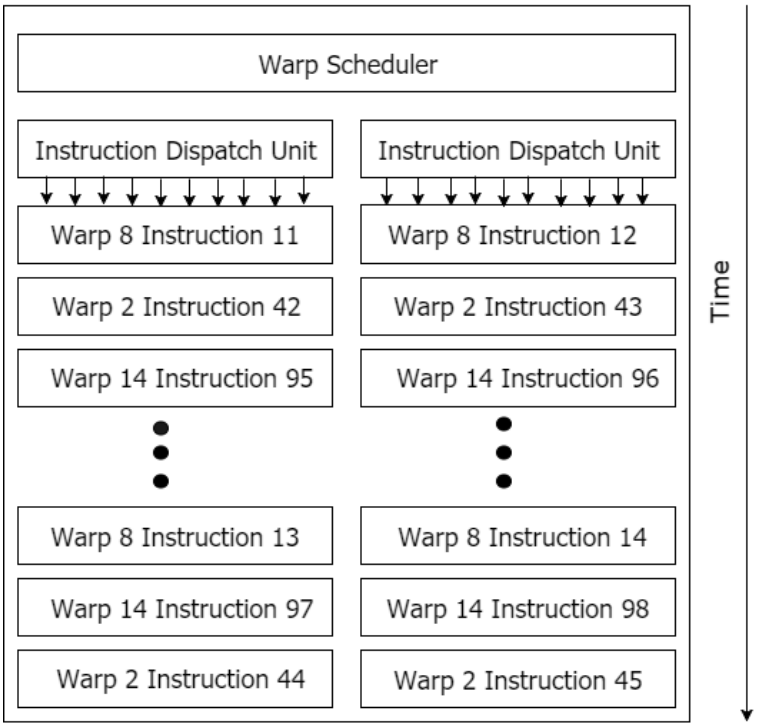

Figure 3: A single warp scheduler unit

2.3.3 Thread Divergence

Thread divergence is caused due to branch statements such as if-then-else, switch and for [5]. Divergence can only happen within a warp. When the branch condition is satisfied for some threads of a warp, the other threads of a warp that do not satisfy the condition are deactivated. For example, if eight threads of a warp evaluate an if condition to be true, then those eight threads will execute the conditional block and the remaining 24 threads will become idle. When the diverged flows merge, all 32 threads of a warp become activated.

The total execution time of a diverged warp is the accumulated execution time of all diverged flows. For example, PathA and PathB in Fig. 4 are sequentially executed by a few threads within the same warp and hence, the total execution time is the accumulated execution time of PathA and PathB. One way to eliminate warp divergence is to have all threads in a block follow the same execution path.

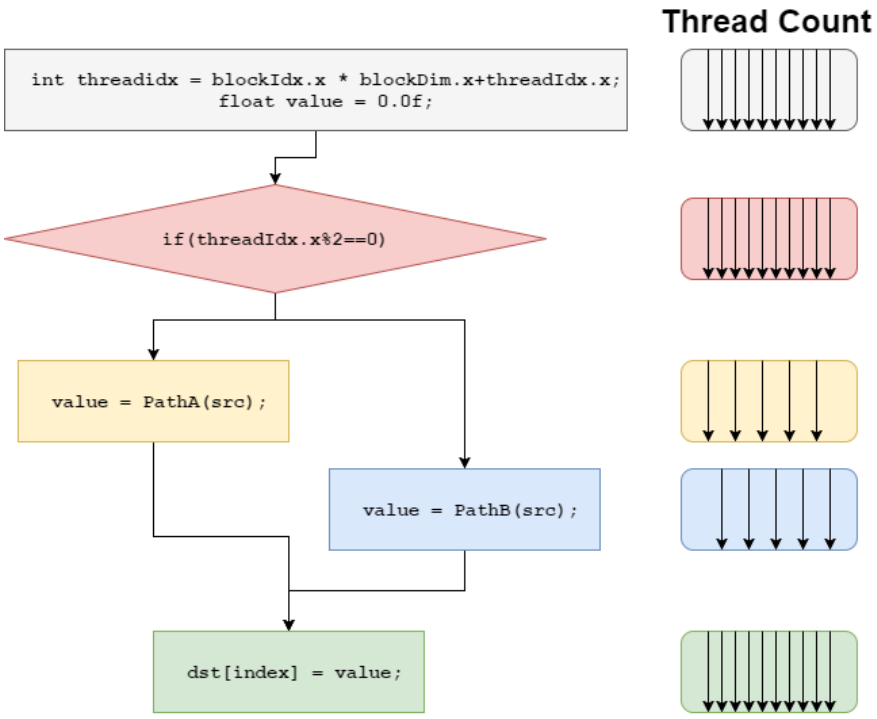

Figure 4: Thread divergence

2.4 Signature Matching

Signature matching inspects the packets payload to detect the presence of malicious network traffic by using various algorithms. The Rabin-Karp, the Wu-Manber and the Aho-Corasick algorithms were used to implement pattern matching.

2.4.1 Rabin-Karp Algorithm

In the Rabin-Karp algorithm [19], each thread computes hash code for the payload it operates on and compares it with hash code of well-known signature patterns. In the CPU-version Rabin-Karp algorithm, hash codes are computed for the neighboring values by using previously computed hash code and a new character. Thus, the run-time complexity of the algorithm is linear. However, the CPU-version algorithm cannot be used on the GPU because all threads execute in a parallel fashion. Hence, all threads compute hash code up to the pattern length and compares it with hash code of the malicious patterns.

Hash code computation and header inspection can be handled in parallel by using threads in different warps. In the Rabin-Karp algorithm, global memory accesses are highly coalesced. Therefore, the Rabin-Karp algorithm outperforms the naive algorithm.

The Rabin-Karp algorithm uses hashing. Considering an example in which the hash table size is 97 and the search pattern is 59372. Hash value is computed as 59372 % 97, which is 8. Hash values of a set of numbers are shown in Table 1.

Table 1: Hash Value Calculation Example (Pattern = 59372)

idx 0	idx 1	idx 2	idx 3	idx 4	idx 5	idx 6	idx 7	idx 8	Hash Value
3	1	5	9	3	7	2	6	3	
3	1	5	9	3					31593%97=68
	1	5	9	3	7				15937%97=29
		5	9	3	7	2			59372%97=8
			9	3	7	2	6		93726%97=24
				3	7	2	6	3	37263%97=20

The modulo hash function can be calculated in linear time using Horner's method [29] and is represented as,

$X_i mod Q = (T_i * R^{M-1} + T_{i+1} * R^{M-2} + ... + T_{i+M-1} * R^0) mod Q$, where R is the range, which equals 10 for decimal numbers, and 256 for ASCII numbers. For this example, assume that R, Q, and M as 10, 97 and 5, respectively. X_i is the number at index i. Then, $X_i mod Q$ is calculated as $(3 * 10000 + 1 * 1000 + 5 * 100 + 9 * 10 + 3 * 1)\%97$, which is 68. The calculations are shown in Table 2.

Table 2: Hash Value Calculation Example with Horner's Method

I	0	1	2	3	4	
0	3	1	5	9	3	
1	(3)%97=3					
2	3	1	(3*10 + 1)%97 = 31			
3	3	1	5	(31*10 + 5)%97= 24		
4	3	1	5	9	(24*10 + 9)%97 = 55	
5	3	1	5	9	3	(55*10 + 3)%97 = 68

The resultant value after applying the hashing algorithm can match for two numbers. For example, 59372 % 97 = 95 and 59469 % 97 = 95. Thus, the potentially matching patterns are compared against the text to see if there is a match. The time complexity of the algorithm is $O(NM)$, where N is the length of the text and M is the length of the pattern.

This algorithm can be optimized by calculating the next hash value by using the previous hash value. The next hash value $X_{i+1} mod Q$ can be computed efficiently by using the previous hash value $X_i mod Q$.

We have, $X_i = T_i * R^{M-1} + T_{i+1} * R^{M-2} + ... + T_{i+M-1} * R^0$ and $X_{i+1} = T_{i+1} * R^{M-1} + T_{i+2} * R^{M-2} + ... + T_{i+1+M-1} * R^0$. Thus, $X_{i+1} = (X_i - T_i * R^{M-1}) * R + T_{i+M}$.

Suppose that, X_i (current value) = 31593, X_{i+1} (next value) = 15937, i = 0, T_i = 3, R^{M-1} = 10000, R = 10, M = 5, and T_{i+M} = 7. When the above formula is applied, $X_{i+1} = (31593 - 3 * 10000) * 10 + 7 = 15937$. After obtaining the next hash value, $X_{i+1} mod Q$, it should be compared with the hash value of the pattern.

Thus, the time complexity of single pattern matching is $O(N)$. When a single pattern-matching algorithm is used for multi-signature matching, the same algorithm is applied to every signature. Hence, the time complexity for multi-pattern matching is $O(N * MaxM)$ where $MaxM$ is the maximum pattern length.

2.4.2 Aho-Corasick Algorithm

The Aho-Corasick algorithm [18] is one of the fastest algorithms for multi-pattern matching. The complexity of this algorithm is linear to the sum of the number of patterns, the length of the input text, and the total number of matches in the text. The algorithm has two phases, preprocessing and search. In the preprocessing phase, the algorithm builds a FSM that looks like a trie using a finite set of patterns. In the search phase, the algorithm traverses the input text along this state machine to find the locations of the patterns in the text.

In the preprocessing phase, three arrays, goto, failure, and output, are constructed.

1. Goto array is a two-dimensional array and stores the next state for the current state and the currently processed character.
2. Failure array holds all the states that should be reached for those characters, that do not contain the next state for the current state. For example, in Fig. 6, when a state is 0 and the characters at this state are not h and s (!(h,s)), the arrow points back to state 0 (for state 0, any other characters other than h and s, have value 0). It is represented as a one-dimensional array.
3. Output array stores the indexes of the patterns that end at a particular state. It is represented as a vector of bit-sets. For a designated state, there can be more than 64 patterns that end at that state. Hence, the data type of the output array cannot be chosen as an int or long.

The algorithm constructs a trie as shown in Fig. 5 and fills goto and output arrays. The algorithm extends the trie into a finite state automaton in order to achieve linear time complexity.

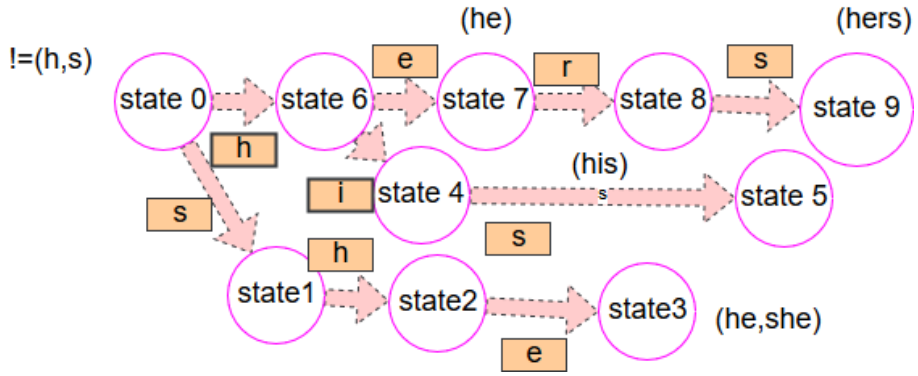

Figure 5: Trie is constructed for the patterns = he, she, hers, his

The value for a state in a failure array can be computed by finding the longest suffix of a pattern which is the prefix of another pattern. For example, consider the pattern “hers” as shown in Fig. 6, the longest suffix is “s,” which is the prefix of “she.” Thus, there is a failure transition from "hers" to “s” of “she.” While constructing the trie, for all characters that do not have the next state at the root, an edge is added back to the root.

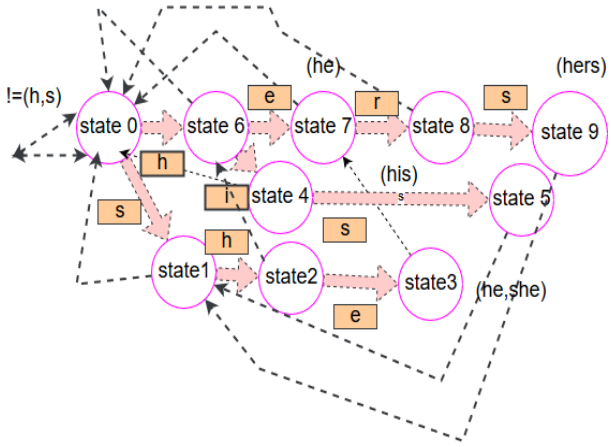

Figure 6: Trie is extended to include the failure transitions

Suppose that the search phase text is "ushers" as shown in Table 3. The algorithm moves from state 0 to state 1 since the first character is a "u." At state 1, the output array is checked. Since the output value at state 1 index is 0, there is no pattern match. Similarly, the FSM is traversed to reach the next state from the present state depending on the input character.

When the output array is checked at state 3 and state 9, it would have the bit corresponding to the pattern set to 1.

Table 3: FSM Transition Example (Pattern = "ushers")

CHARACTER	STATE	MATCHES
u	0	NONE
s	state 1	NONE
h	state 2	NONE
e	state 3	he,she
r	state 8	NONE
s	state 9	hers

2.4.3 Wu-Manber Algorithm

The Wu-Manber algorithm [16] is based on the idea of the Boyer-Moore algorithm [17] and the search starts from right and proceeds to left. According to the Boyer-Moore algorithm, while searching in the text from right to left, if the rightmost character in the text does not match any of the characters in the pattern, a block of characters up to the pattern length can be skipped while searching and the search pointer can be moved to the right by the pattern length. The intuition is that, the right most character is likely to be different when the pattern doesn't match and hence, multiple characters can be skipped while searching and thus, improve the performance.

The algorithm consists of two phases, preprocessing and search. In the preprocessing phase, the shift and prefix tables are constructed. The shift table gives the value that is needed to shift from the current position with respect to the suffix of the patterns. The prefix table is indexed by the hash values of prefixes of the patterns, and contains the corresponding patterns as the value. When multiple patterns have the same hash value, the shift table stores the value which requires the minimum shift and the prefix table stores all the patterns that map to the hash value using a linked list.

When a set of characters at the current position matches a suffix of a pattern, the shift value will be zero. When the value of the shift is zero, the prefix is calculated by considering the characters from 'current position - minimum pattern length' array index up to the current position, which serves as a key to the prefix table.

Consider a text "ANDYMEETMESOONATGOOGLE" and patterns "SOON," "COME" and "GOOGLE." The minimum length of the patterns is four. The shift table and prefix table are constructed using sub-arrays of length four as shown in Table 4.

Table 4: Shift Table Generated after Preprocessing

SO	OO	ON	CO	OM	ME	GO	OO	OG	*
2	1	0	2	1	0	2	1	0	3

Prefix table points to a list of patterns whose first B(2) characters are hashed as the index as shown in Table 5.

Table 5: Prefix Table Generated after Preprocessing

Index: hash of the prefix	Value: list of patterns with common prefix
SO	SOON
CO	COME
GO	GOOGLE

In the search phase, initially the last four characters of the text are examined only because four is the minimum pattern length. Later, the last two characters of the four characters are used for the hash computation. The calculated hash value is used for indexing the shift table. The search pointer is shifted by the value of the shift at every search. When the shift value is 0, the hash value of the first two characters of the four characters is used as the index for the prefix table. The patterns in the prefix table are compared with the text. When the pattern size is longer than four, additional characters from the text are included to continue the search.

Considering the example, as shown in Table 6. In step 1, the index for DY will map to a wild character, *, in the shift table because there is no pattern that has DY as the suffix. Hence, a maximum shift is applied, which is three. In step 3, the index for ME will map to ME in the shift table because the pattern COME has ME as the suffix. Since the shift is 0, the hash value for ET is calculated and the prefix table is checked. As ET does not index to any value in the prefix table, the algorithm continues.

Similarly, in step 5, the prefix table is accessed and the pattern "SOON" is compared against the text, and the result is a pattern match.

Table 6: Multi-Pattern Search on "ANDYMEETMESOONATGOOGLE" Using Pre-processed Shift and Prefix Tables

STEP	INDEX	SHIFT	PREFIX CHECK	RESULT
1	3	3		
2	6	3		
3	9	0	Yes	NoMatch
4	10	3		
5	13	0	Yes	Match(SOON)
6	14	3		
7	17	2		
8	19	1		
9	20	0	Yes	Match(GOOGLE)

2.5 Parallelism in CPU Programming with OpenMP

When a sequential program is executed on the CPU, only one core is used. However, if OpenMP is used, all the cores can be utilized because threads are evenly distributed across the cores. OpenMP follows the Fork-Join model, where the main thread starts and creates a set of worker threads [31]. The number of threads to be created can be decided dynamically or can be specified at compile time. The worker threads are active in a parallel region, and when they exit it, they join with the main thread.

All threads in an OMP region are assigned to a thread block. Each OMP thread maps to one physical core, but more than one thread can be mapped to a core. Omp_get_num_threads() function returns the number of threads, which is greater than one in the parallel region, and is one outside it. Omp_get_max_threads() returns the number of cores in the CPU.

Fig. 7 shows an example for an OpenMP program implemented in C.

The for loop is split into multiple chunks and each thread executes one chunk of the for loop. Even when the compiler does not support OpenMP, the program yields correct output, but without parallelism.

```
1 int numOfProcessors = omp_get_max_threads();
2 omp_set_num_threads(numOfProcessors);
3 #pragma omp parallel for
4 for(int i=0;i<4;i++) {
5 c[i] = a[i] + b[i];
6 }
```

Figure 7: Example for an OpenMP program in C

CHAPTER 3

System Organization

The new IDS architecture consists of three parts: memory transfer of packets from the CPU to the GPU, packet analysis on the GPU, and the transfer of inspection results back to the CPU.

3.1 Memory Transfer of Packets to the GPU

The CPU captures the packets using the APIs provided by the packet capture library LibPCAP [4]. The packet capture library abstracts the packet capture process and provides a mechanism to capture packets from a live stream or to read packets from a saved file. These packets should be transferred from the CPU to the GPU. Due to the overhead associated with transferring data from the CPU to the GPU, all the packets are transferred into a buffer and the buffer is transferred to the GPU.

The system is divided into three components, packet capture, dissector and analysis as shown in Fig. 8.

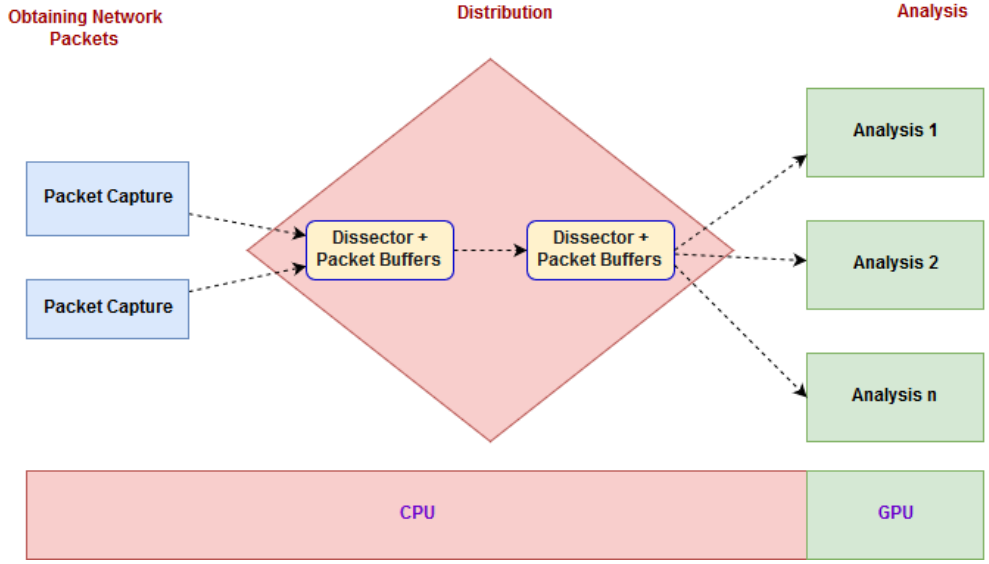

Figure 8: System organization

1. Packet Capture: This component captures the packets and saves them into packet buffers.

2. Dissector: Dissector class consists of virtual functions, which are used to dissect the packet to obtain the attributes of the headers and the start of the payload.
3. Analysis: This component works on the GPU and performs header checking and signature matching on the payload.

3.2 Packet Capture Class

The system defines PacketCapture as a class that is responsible for saving the packets into a PacketBuffer object. The PacketBuffer class has an array of packets that stores the raw data from the network and headers of the packets. The size of the packet buffer array is fixed and the kernel is launched with a specific number of threads that is divisible by the array size. The PacketCapture class obtains the packets in two modes, live capture mode and off-line capture mode.

1. Live capture mode: In this mode, the system can capture the packets in real-time and use them for surveillance and anomaly prevention.
2. Off-line capture mode: In this mode, packets can be obtained from a TCP dump capture file or another source. They can be used to analyze and prevent a malicious attack in the future.

3.3 Analysis

The analysis is performed on the GPU using an array of packets that were transferred from the CPU. The analysis module is divided into three components, auto-mining, operations and hooks.

1. Auto-Mining: The entire packet analysis is distributed among the threads. As already mentioned, shared memory is much faster than global memory. Therefore, the data required for each thread are saved in shared memory in this module.
2. Operations: In this module, the threads operate on the collected data. Header checking and signature matching are implemented. The naive, the Rabin-Karp, the Wu-Manber and the Aho-Corasick algorithms are used for signature matching.

The results are written into an array after the rules are checked.

3. Hooks: This module is written in C++. The results can be logged into a file, or a database, or they can be displayed on the console.

CHAPTER 4

Parallel IDS Design on GPUs

4.1 Overview of Parallelism Approaches

There are two types of parallelism employed to speed up packet processing on the GPU as shown in Fig. 9 and Fig. 10.

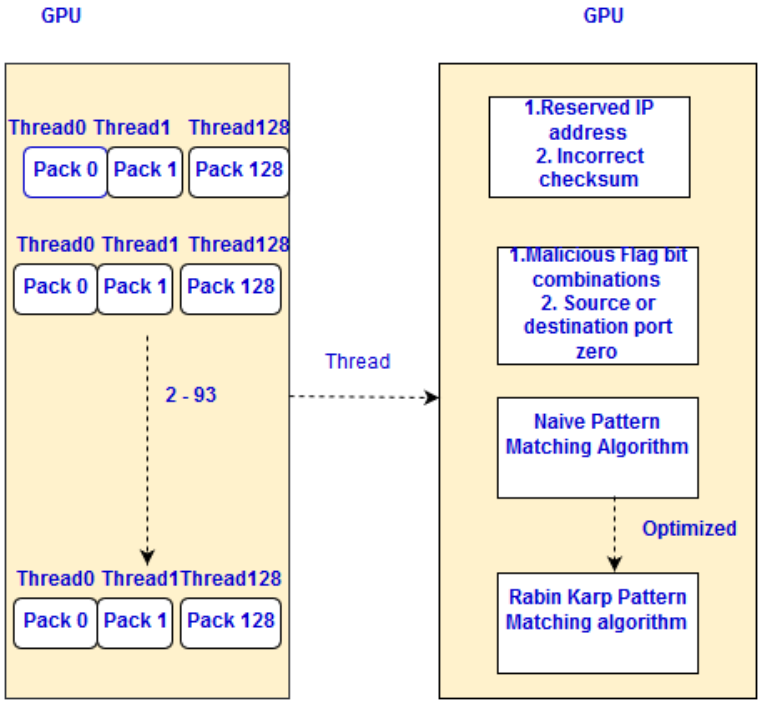

Figure 9: Thread-level parallelism

4.1.1 Thread-Level Parallelism

In thread-level parallelism, each packet was analyzed by one thread. The kernel was launched with 260 blocks of 256 threads each. A thread executed both header checking and pattern matching. The code was optimized to use shared memory. Signatures are stored in global memory. In the naive pattern-matching algorithm, each thread accesses global memory N times, where N is the number of patterns multiplied by the maximum pattern length. The time taken to access data from global memory is approximately 200 to 300 cycles [14].

In order to decrease the amount of global memory accesses, the Rabin-Karp algorithm was developed. In the Rabin-Karp algorithm, each thread accesses global memory only M times, where M is the total number of patterns.

4.1.2 Block-Level Parallelism

In block-level parallelism, a group of 256 threads cooperatively process each packet. The entire block of threads cooperatively execute header checking and pattern matching on a single packet. In this approach, warp divergences was encountered due to if conditions. To eliminate warp divergence, the code was modified such that threads in different warps execute different if conditions. For example, threads in warp 1 execute the IP rules, threads in warp 2 execute the TCP rules and threads in the remaining warp execute pattern matching.

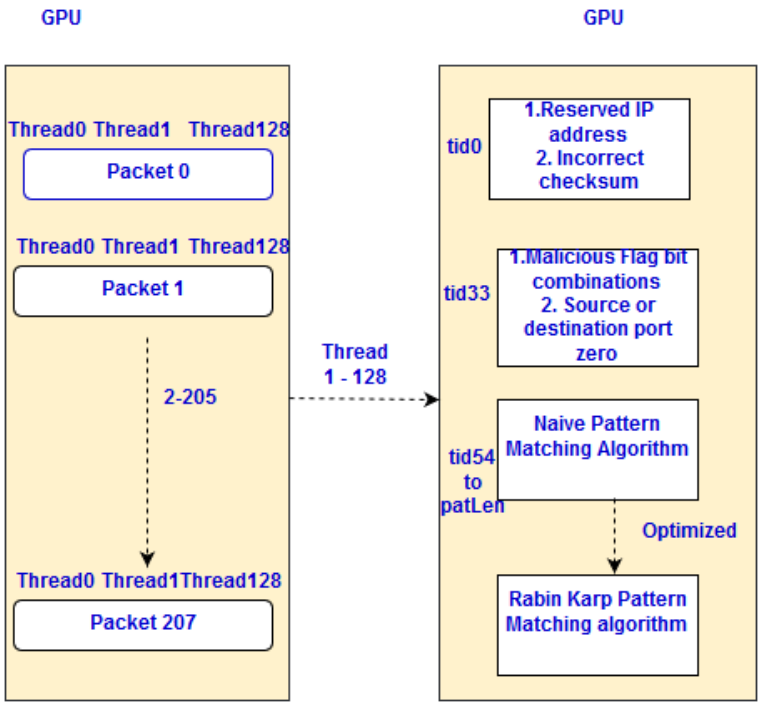

Figure 10: Block-level parallelism

4.2 Header Checking

An attacker may send crafted packets to a computer in order to consume all of its resources, such that it cannot service any other request. To identify crafted packets, header checking evaluates the validity of the header part of a given packet. The headers of the packets were analyzed by applying the following rules on them.

The rules are:

1. Check if the IP address of the packet is in the private address range, because there can be no computer on the Internet whose IP address is in this range.
2. Check if the flag bits in the TCP header are crafted or check if the reserved bits are set. Consider as an example that, it is not possible to have a combination of the SYN and FIN bit set.
3. Check if the acknowledgment bit is set and the acknowledgment number is zero. Check if the source or destination port is zero for a TCP packet.
4. Check if the IPv4 checksum was tampered while in transit. If any of the above rules match, the packet is discarded from further processing.

When a packet passes all the above rules, pattern-matching algorithms are applied on the packet. Otherwise, an administrator is notified. Currently, there are four rules for header checking, but the IDS can be easily scaled to cover more rules.

4.3 Parallel Pattern-Matching Algorithms using CUDA

4.3.1 Rabin-Karp Algorithm

The single-pattern Rabin-Karp algorithm was modified to obtain the multi-pattern-matching algorithm. In the multi-pattern-matching version, the patterns are represented by a two-dimensional array. In order to copy the two-dimensional array to the GPU, the array is flattened to a one-dimensional array, which is then copied to the GPU. Each thread compares the hash code of every pattern against the payload starting from the position that corresponds to its thread index.

Multiple iterations are run by each thread, corresponding to the number of patterns. As shown in Fig. 11, all of the threads with a thread index above 53 perform pattern matching. Each thread iterates over all patterns and applies the Rabin-Karp algorithm to each pattern. Therefore, the execution time is very long. Thus, this algorithm is not suitable for multiple pattern matching.

```
/*Rabin Karp Multi-pattern string matching implementation*/
if(threadIdx.x >= 54) {
GPU_results[0].num_strings = const_num_strings;

for(int i=0;i<const_num_strings;i++) {
int patLen = const_indexes[2*i+1] - const_indexes[2*i];
//This condition checks if the pattern length is < packet length
if(threadIdx.x<=256-patLen) {
int hy,j;
for(hy=j=0;j<patLen;j++) {
if((j+threadIdx.x) >= 256) goto B;
hy = (hy * 256 + elements[j+threadIdx.x].packet) % 997;
}
if(hy == const_patHash[i] &&
      memCmpDev<T>(elements,const_pattern,const_indexes,i,threadIdx.x,patLen) == 0)  {
GPU_results[blockIdx.x].maliciousPayload = 1;
GPU_results[blockIdx.x].signatureNumber = i;
 d_result[i]=1;
}
}
}
```

Figure 11: Parallel implementation of the Rabin-Karp algorithm

4.3.2 Aho-Corasick Algorithm

Each thread searches for multiple patterns starting from its thread index. Each thread uses the goto table present in global memory to advance to the next state based on the character at its position. Then, each thread checks the output array to see if there are any virus patterns at this position and adds the index of the pattern to the result if there was a match. Since each thread starts at a particular byte of the input text, the failure table was not used in the GPU-version algorithm. Thus, the failure-less Aho-Corasick algorithm was implemented [20], as shown in Fig. 12.

```
    // Aho Corasick Algorithm
    if (threadIdx.x>=54)
{
int pos=threadIdx.x;
char ch = elements[pos++].packet;
int chint = ch & 0x000000FF;
int nextState = stateszero [chint];
if (nextState !=0) {
if (d_output[nextState] > 0) result [blockIdx.x] = d_output[nextState];
while(nextState !=0 && pos<256) {
ch = elements[pos++].packet;
chint = ch & 0x000000FF;
nextState = gotofn[nextState*256 + chint];
if (d_output[nextState] > 0) result [blockIdx.x] = d_output[nextState];
}
}
}
```

Figure 12: Parallel implementation of the Aho-Corasick algorithm

4.3.3 Wu-Manber Algorithm

Each thread searches for multiple patterns starting from its thread index. The threads compute the hash value of the suffix from its (position + m-1) array index, where m is the minimum pattern length up to three characters backward, as shown in Fig. 13, and check if the shift value from the shift table is zero.

If the value of shift is zero, then the hash value of the prefix starting from its thread index up to two characters is calculated and searched in the prefix table to check if it exists as shown in Fig. 14.

```
    //Each thread starts searching from its thread Id. elements array contains the 256 byte
        packet //and is stored in shared memory.
    unsigned int hash1, hash2;
    if (threadIdx.x >= 54 + m-1) {
    hash1 = elements[threadIdx.x - 2].packet & 0x000000FF; //bitwise & used because to avoid two
        complement negative numbers
    hash1 <<= 2;
    hash1 += elements[threadIdx.x - 1].packet & 0x000000FF;
    hash1 <<= 2;
    hash1 += elements[threadIdx.x].packet & 0x000000FF;
    int shift = d_SHIFT[hash1];
```

Figure 13: Calculating the hash value of the suffix

The patterns located at a particular prefix are compared with the text by a memory compare operation. If there is a match, the pattern index is added to the resulting structure. Since each thread starts from a particular byte and all indexes of the text are handled by a thread, the threads need not shift forward and continue the search again. Therefore, each thread executes the algorithm only once.

```
if ( shift == 0) {
hash2 = elements[ threadIdx.x - m + 1].packet & 0x000000FF;
hash2 <<= 2;
hash2 += elements[ threadIdx.x - m + 2].packet & 0x000000FF;

// For every pattern with the same suffix as the text
for ( int i = 0; i < d_PREFIX_size[hash1]; i++) {
// If the prefix of the pattern matches that of the text
if (hash2 == d_PREFIX_value[hash1 * prefixPitch + i]) {
int patIndex = d_PREFIX_index[hash1* prefixPitch + i];
int starttxt = threadIdx.x - m + 1 + 2;
int startpat = d_stridx[2*patIndex] + 2;
int endpat = d_stridx[2*patIndex+1];

// memcmp implementation
while(elements[ starttxt ].packet!='\0' && startpat < endpat) {
if(elements[ starttxt ++].packet!=d_pattern[ startpat ++]) return;
}
if( startpat >= endpat) {
printf("The pattern exists %d\n", patIndex);
GPU_results[blockIdx.x].maliciousPayload = 1;
result[blockIdx.x] = patIndex;
}
}
```

Figure 14: Pattern is searched based on the hash value of the prefix

4.4 Utilization of Pinned Memory

Two different versions of the three algorithms were developed. One version used pinned memory to transfer data from the CPU to the GPU and another used non-pinned memory. Fig. 15 illustrates a case in which data are transferred without using pinned memory. In this case, memory should be allocated using the cudaMalloc() function and then the contents should be copied from host to device memory using the cudaMemcpy() function.

```
// Allocate host side memory
int * result = (int*)malloc(N*sizeof(int));

// allocate device side memory
cudaAssert(cudaMallocPitch(&d_gotofn,&pitch,chars * sizeof(int), states));

// copy from host memory to device memory
cudaAssert(cudaMemcpy2D(d_gotofn,pitch,gotofn,chars * sizeof(int),chars *
    sizeof(int), states ,cudaMemcpyHostToDevice));

// copy result from device memory to host memory
cudaAssert(cudaMemcpy(result,d_result ,N*sizeof (int),cudaMemcpyDeviceToHost));
```

Figure 15: Using non-pinned memory for data transfer

Pinned memory should be allocated using the cudaHostAlloc() function and a pointer to the allocated memory should be passed to the GPU as shown in Fig. 16. The host and the device can access pinned memory. Hence, a memory copy is not required to transfer data back and forth between the device and the host.

```
// Allocate Pinned Memory
cudaAssert(cudaHostAlloc((void**) &array, states * 256 * sizeof(int), cudaHostAllocMapped));
cudaAssert(cudaHostAlloc((void**) &result, N * sizeof(int), cudaHostAllocMapped));

// Getting the device pointer for the pinned memory
cudaAssert(cudaHostGetDevicePointer(&d_gotofn, array, 0));
cudaAssert(cudaHostGetDevicePointer(&d_result, result, 0));
```

Figure 16: Using pinned memory for data transfer

CHAPTER 5

Implementation

The netGPU [3] framework was used to capture packets from the network interface card or file and to transfer the packets to the GPU.

5.1 Packet Capture and Transfer to the GPU

5.1.1 Capturing the Network Packets

Packet feeders obtain network packet data using the LibPCAP library and save the data into the packet buffer object, as shown in Fig. 17. LibPCAP [4] is a C/C++ library used to capture network traffic data. If the buffer is full, the thread waits until the processor thread copies the buffer from the CPU to the GPU.

```
1  loadPacketBuffer ()
2  {
3  lockTheMutex();
4   validate  =  ifBufferFull  () ;
5  if ( validate ==1)
6  // WaitForBuffer  waits  until  the  entire  buffer  is  full
7  waitForBufferToBeCopied();
8  unlockTheMutex();
9  lockTheMutex();
10  saveIntoPacketBuffer () ;
11 unlockTheMutex();
12 }
```

Figure 17: Pseudo code to store the packet buffer that will be copied to the GPU

5.1.2 Buffer the Network Packets

getSniffedPacketBuffer() is the getter method of the packet feeder class to obtain the packet buffer object. The derived classes of the packet feeder class should implement this method. The classes which derive from the abstract packet feeder class will obtain the packets from the network card or a file. The packet buffer object defines an array of "Max Buffer" size packets and the maximum size of each packet is "Max Packet" bytes. Each packet has a header and a payload. Fig. 18 shows the data structure of the packet buffer object and packets with the header and body.

The packet buffer object is copied to pinned memory so that the GPU can directly access it. Fig. 18 contains the data structure representing the header of the packet. The header consists of two fields, the proto and offset. The proto and offset fields are one-dimensional arrays of size seven, and they represent the seven layers of the open source interconnect (OSI) model.

```
typedef struct {
int proto [7];
int offset [7];
} headers_t ;

typedef struct {
timeval timestamp;
headers_t headers ;
uint8_t packet [MAX_BUFFER_PACKET_SIZE];
} packet_t ;

packet_t * buffer ;
```

Figure 18: The array of packets of "Max Buffer" size

5.2 Dissectors

Dissector class was used to get the size of the Ethernet, IPv4, TCP, and UDP headers and was used to fill the proto and offset fields. A method from the PacketBuffer class calls the dissect method of this class before pushing the packet into the packet buffer. The dissect method calls a data link layer method depending on the data link layer protocol. The data link layer method calls the network layer method (IPv4 or IPv6) depending on the network layer protocol. The network layer method calls the transport layer protocol method depending on the transport layer protocol (TCP or UDP). DissectEthernet, dissectIpv4, and dissectTcp are data link layer, network layer, and transport layer methods, respectively, as shown in Fig. 19. The dissect methods call the virtual action methods.

```
class Dissector {
public:
unsigned int dissect(const uint8_t* packetPointer,const struct pcap_pkthdr* hdr,const int
    deviceDataLinkInfo,void* user);
private:
void dissectEthernet(const uint8_t* packetPointer,unsigned int * totalHeaderLength,const
    struct pcap_pkthdr* hdr,void* user);
void dissectIp4(const uint8_t* packetPointer,unsigned int * totalHeaderLength,const struct
    pcap_pkthdr* hdr,void* user);
void dissectTcp(const uint8_t* packetPointer,unsigned int * totalHeaderLength,const struct
    pcap_pkthdr* hdr,void* user);
void dissectUdp(const uint8_t* packetPointer,unsigned int * totalHeaderLength,const struct
    pcap_pkthdr* hdr,void* user);
void dissectIcmp(const uint8_t* packetPointer,unsigned int * totalHeaderLength,const struct
    pcap_pkthdr* hdr,void* user);

// Virtual Actions:

virtual void EthernetVirtualAction(const uint8_t* packetPointer,unsigned int*
    totalHeaderLength,const struct pcap_pkthdr* hdr,Ethernet2Header* header,void* user)=0;
virtual void Ip4VirtualAction(const uint8_t* packetPointer,unsigned int*
    totalHeaderLength,const struct pcap_pkthdr* hdr,Ip4Header* header,void* user)=0;
virtual void TcpVirtualAction(const uint8_t* packetPointer,unsigned int*
    totalHeaderLength,const struct pcap_pkthdr* hdr,TcpHeader* header,void* user)=0;
virtual void UdpVirtualAction(const uint8_t* packetPointer,unsigned int*
    totalHeaderLength,const struct pcap_pkthdr* hdr,UdpHeader* header,void* user)=0;
virtual void IcmpVirtualAction(const uint8_t* packetPointer,unsigned int*
    totalHeaderLength,const struct pcap_pkthdr* hdr,IcmpHeader* header,void* user)=0;

virtual void EndOfDissectionVirtualAction(unsigned int* totalHeaderLength,const struct
    pcap_pkthdr* hdr,void* user)=0;

};
```

Figure 19: Dissect methods and targeted OSI layer methods

The virtual functions are defined in the pre-analyzer dissector class and the size dissector class. The pre-analyzer dissector class methods are used for performing header checking in the CPU using C and OpenMP. The size dissector class methods are used to extract the packet to get the protocol and header offset fields. The offsets will be used in the GPU for DPI.

5.2.1 PreAnalyzerDissector

This component was used while developing the CPU-version of DPI. The methods of this class are used to decode and analyze the headers in the packet. The IPv4 virtual action method checks the integrity of the IPv4 layer header. The TCP virtual action method checks the integrity of the TCP header. In the payload module, three string matching algorithms were developed using C and OpenMP.

CHAPTER 6

Evaluation

For the experiments, Tesla K80 GPU was used, which has 13 SMs and each SM contains 192 SPs. The GPU operates at 562 GHZ with 11.25 GB global memory, 48 KB shared memory/block, 64K registers/block, and 64 KB constant memory. An Intel Xeon Processor E5 family operating at 2.30 GHz was used as a host CPU. The packets were captured using a packet capture file, and there were 100 packets per file. The GPU main function was launched with 260 blocks and 256 threads per block.

6.1 Block-Level Parallelism vs. Thread-Level Parallelism

The total packet processing time was measured by varying the size of the packet. The Rabin-Karp algorithm was used for signature matching. The execution times for thread-level and block-level parallelism are plotted on a log scale shown in Fig. 20.

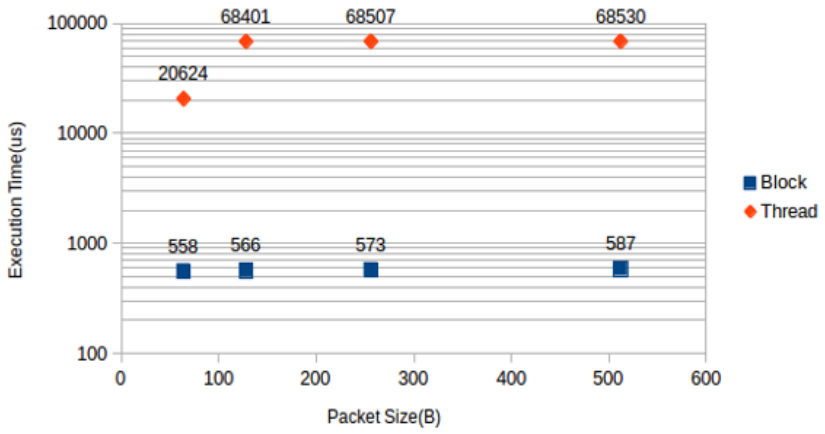

Figure 20: Execution times for packet processing

As can be seen, the execution time gradually increased with the packet size. Block-level packet processing significantly improved the throughput over thread-level packet processing. This is because of the high level of parallelism that block-level packet processing provides.

Note that, block-level processing inspects individual packets with multiple threads. On the other hand, thread-level processing handles one packet per thread. Hence, the individual packet processing time was longer than block-level processing.

6.2 Comparison between OpenMP, CPU and GPU solutions

As shown in Table 7, the execution times of the Aho-Corasick and the Wu-Manber algorithms were almost stable irrespective of the signature count, while that of the Rabin-Karp algorithm was proportional to the signature count. This is because the Aho-Corasick and the Wu-Manber algorithms are multiple signature-matching algorithms in which the text is searched only once for all the patterns. However, in the Rabin-Karp algorithm, the patterns are searched by iterating over the total number of patterns.

Table 7: Execution Times on the GPU

Number Of Patterns	Aho-Corasick	Wu-Manber	Rabin Karp
10	0.222589	0.220862	26.297111
50	0.213149	0.212749	47.350966
100	0.216893	0.211166	45.577234
200	0.217278	0.20659	59.012978
500	0.20355	0.206813	107.944879
1000	0.204189	0.240702	201.787673
1500	0.204285	0.225118	253.685357
1800	0.208605	0.204189	302.226957

The execution times are shown for the optimized version of the algorithms, which were developed using pinned memory. To understand the performance improvement of the GPU-version IDS, I compared the execution times of sequential CPU-based IDS and parallel CPU-based IDS with OpenMP, and GPU-based IDS with and without pinned-memory optimization. The execution times are listed in Table 8 and Table 9.

Table 8: Performance of Pattern-Matching Algorithms

Number Of Patterns	CPU	OpenMP	Not Pinned	Pinned
10	3005	7985	0.105183	0.222589
50	2973	7787	0.103743	0.213149
100	2781	8132	0.104735	0.216893
200	4420	12878	0.101759	0.217278
500	5520	23179	0.101663	0.20355
1000	10185	39891	0.101151	0.204189
1500	7081	51755	0.103423	0.204285
1800	8227	68304	0.100447	0.208605

Table 9: Comparison of Execution Times for the Wu-Manber Algorithm

Number Of Patterns	CPU	OpenMP	Not Pinned	Pinned
10	213.698	1452.645	0.447387	0.220862
50	210.193	1181.612	0.458107	0.212749
100	218.394	2033.47	0.452315	0.211166
200	3263.717	1231.181	0.468123	0.20659
500	7188.387	1296.764	0.454875	0.206813
1000	12594.664	2411.357	0.465852	0.240702
1500	18745.097	2738.667	0.464059	0.225118
1800	18564.539	1307.96	0.452891	0.204189

As can be seen in Fig. 21, the GPU-version IDS had 60 times greater speedup than the CPU-version in all three algorithms. Fig. 21 shows that the speedup increases as the number of patterns increases.

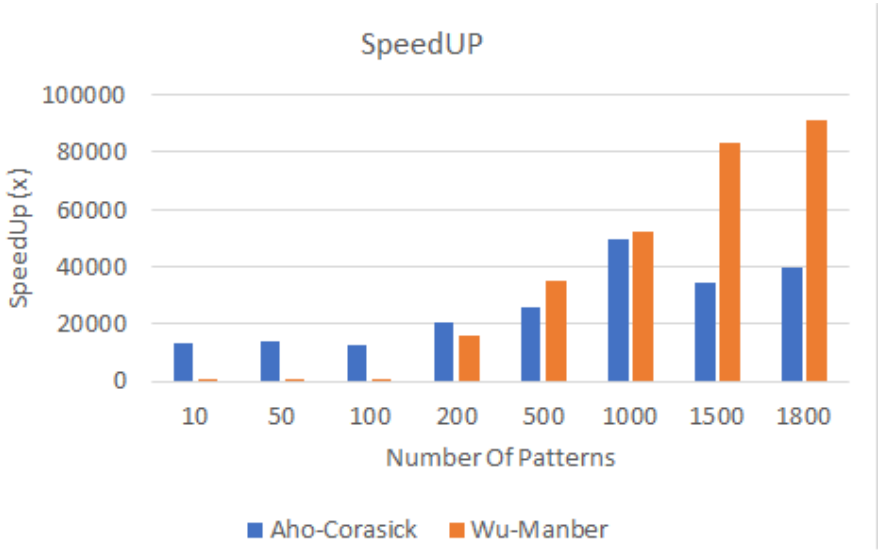

Figure 21: Speedup of the algorithms on the GPU over the CPU

While using the Aho-Corasick algorithm, the OpenMP-based parallel CPU code derived worse performance than sequential CPU code, since the total amount of work done by all threads is larger than the amount of work done by a single thread as explained in the section 5.

In the Wu-Manber algorithm, when the number of patterns is fewer than 200, the sequential CPU code beats the OpenMP version because of the overhead of OpenMP threads. However, when the number of signatures increases, the execution time using OpenMP is significantly shorter than the execution time of sequential version code.

6.3 Stall Breakdown

The evaluation was conducted with 10 packets of 256 bytes and 1800 patterns. Synchronization, memory throttle and execution dependencies had significant differences for pipeline stalls as observed in Fig. 22.

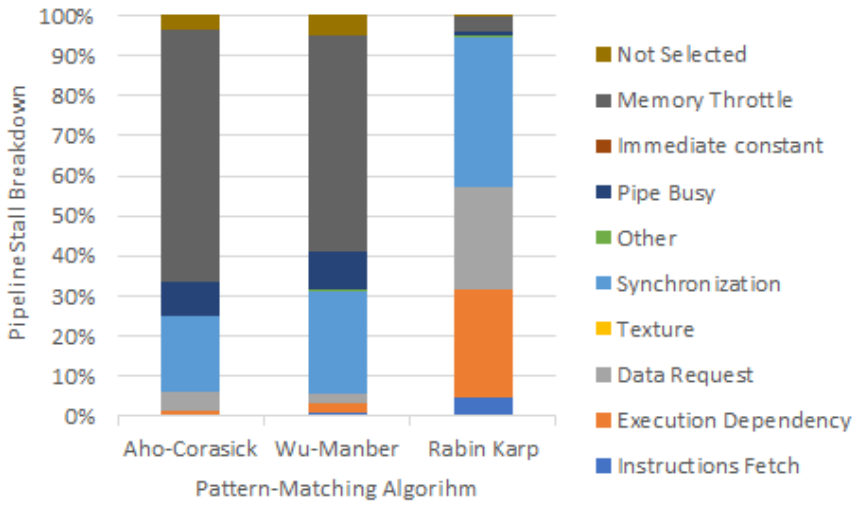

Figure 22: Breakdown for the issue stall reasons

Memory throttle is caused due to huge number of outstanding memory requests to global memory [5]. The global memory load efficiency was < 25%, and store efficiency was < 15% due to un-coalesced memory requests to global memory. Un-coalesced memory requests imply the memory requests are outside the 128 or 64 or 32-byte memory gap. The driver coalesces global memory stores and loads of threads of a warp, when all memory accesses are within a 128 or 32 or 64-byte memory segment [15].

Table 10 shows the distribution of pipeline stalls for the three algorithms. In the Aho-Corasick algorithm, the goto function table and the output table are accessed from global memory. Goto table is a two-dimensional array, where the number of rows is same as the number of states and the number of columns is 256. The number of states in the FSM is 447129. Thus, the size of the goto function table is 447129 x 256 x 4 bytes, which equals 447129KB. Due to the huge size of the goto function table, the data could not be saved in shared memory because shared memory size is 16KB. Because the individual threads access the goto table at random positions it leads to un-coalesced memory accesses.

Table 10: Pipeline Stall Breakdown

Issue Stall Reasons	Aho-Corasick	Wu-Manber	Rabin Karp
Instructions Fetch	0.16%	0.84%	4.74%
Execution Dependency	1.27%	2.35%	26.72%
Data Request	4.75%	2.68%	25.68%
Texture	0.00%	0.00%	0.00%
Synchronization	18.73%	25.44%	37.40%
Other	0.20%	0.21%	0.67%
Pipe Busy	8.20%	9.45%	0.67%
Immediate constant	0.34%	0.33%	0.00%
Memory Throttle	62.76%	53.82%	3.80%
Not Selected	3.60%	4.90%	0.31%

In the Wu-Manber algorithm, the arrays such as shift, prefix size, prefix value, prefix index and pattern are accessed from global memory. The tables are indexed by the hash values, and the hash values calculated by different threads are different. Hence, there would be multiple un-coalesced memory requests to global memory. In the Rabin-Karp algorithm, the memory throttle was much lesser compared to other algorithms. All the threads access the same address of global memory to fetch the pattern hash value. When all the threads in a warp access the same address, the data from global memory are broad-casted to threads in a single memory transaction.

Synchronization occurs because the threads in a warp are blocked due to the call to the syncthreads function. Before the call to a syncthreads function, a store causes shared store latency. The number of calls to syncthreads are the same across three algorithms.

Thread execution time is the sum of the time a thread spends executing instructions plus the idle time a thread resides waiting for the result from previous instructions. This is due to execution dependency.

Multiple threads would share an execution unit of a processor whereby each thread is given a period of time (time slice) to execute before it would be preempted and the execution unit given to another thread. If the threads have un-finished work, it waits for another time slice. The waiting time depends on the number of parallel threads and the availability of resources. In the Aho-Corasick algorithm, the threads perform fewer arithmetic operations on the device (calculations are completed in the pre-processing stage in the CPU), hence the utilization of execution units was low. In the Wu-Manber algorithm, the hash values are computed for suffixes and prefixes of the text. The hash values are computed using three instructions whereby each instruction is dependent on the result of previous instruction. Hence, the execution dependency was slightly higher than the Aho-Corasick algorithm. In the Rabin-Karp algorithm, each thread computes the hash value for 1076 patterns, and the instruction dependency led to a higher execution dependency when compared to other algorithms.

6.4 Resource Utilization

6.4.1 Memory Utilization

Packets are stored in shared memory for the three algorithms. Shared memory load transactions are the number of load requests to shared memory, while shared memory store transactions are the number of store requests to shared memory. The shared memory load transaction count was very high for Rabin-Karp compared to the other two algorithms as shown in Fig. 23. The reasoning is supported by the shared memory replay overhead shown in Fig. 24.

In the Rabin-Karp algorithm, each thread starts signature matching from its thread index, such that tid 0 accesses the byte in shared memory from index 0, tid 1 from index 1, tid 2 from index 2 and so on. The default 32-bit addressing mode is used in shared memory. The bandwidth of each memory transaction is 32-bits. Even though each thread fetches only 1 byte of data, 4 bytes of data are fetched from shared memory.

Since the threads in a warp access the consecutive bytes of the same bank in the shared memory, it led to shared memory bank conflicts.

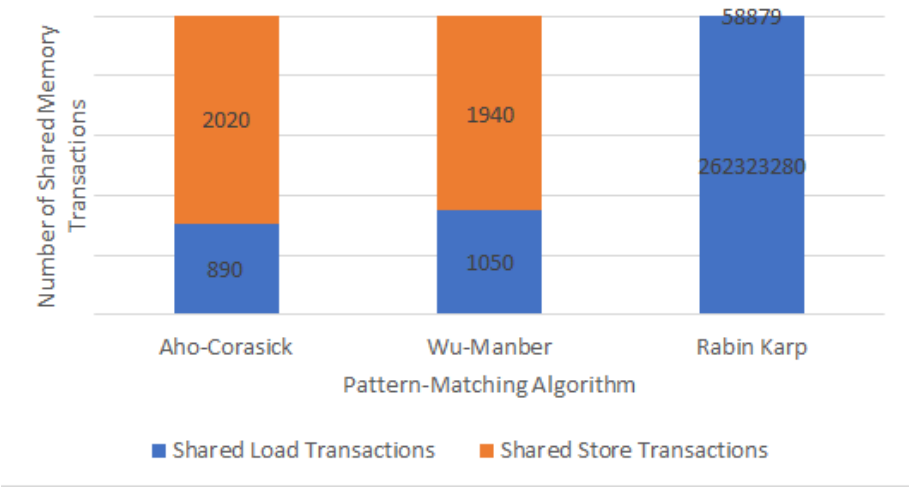

Figure 23: Shared memory load and store transactions

In the Aho-Corasick and the Wu-Manber algorithms, shared memory replay overhead was approximately equal and very low compared to the Rabin-Karp algorithm. The replay overhead was mainly caused by the DPI on the header in both of these algorithms. Shared memory store was slightly higher in Aho-Corasick algorithm because the values of the next state for the first character of every pattern are stored in shared memory.

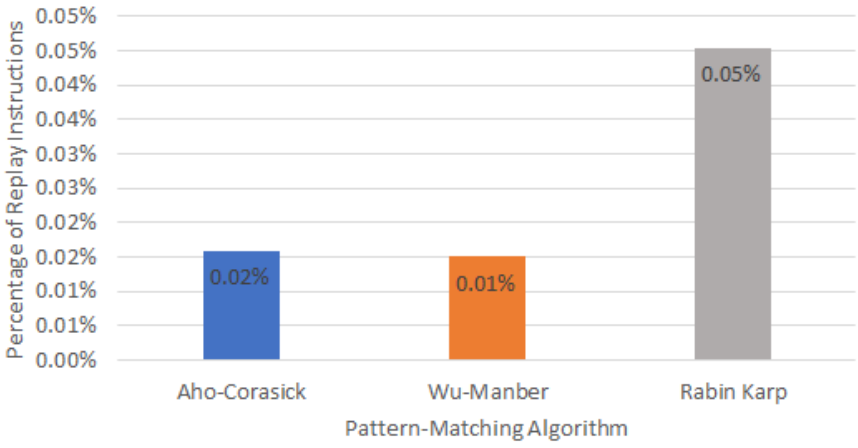

Figure 24: Shared memory replays due to bank conflicts

Since the number of load or stores are high, shared memory accesses could be replaced by shuffle instructions. Using shuffle instructions, threads in a warp exchange data among themselves without the need of shared memory or global memory. Shuffle instructions have lower latency when compared to shared memory instructions, and do not consume any space.

6.4.2 Warp Utilization

Compute resources are efficiently utilized when all threads in a warp execute the same branch. When this does not happen, the warp execution efficiency is reduced because of the under utilization of the compute resources. During the header check phase of deep packet inspection, only a few threads were active per warp, which led to a decrease in the execution efficiency. The header check phase is common to the three algorithms, and the differences in the graph in Fig. 25 were due to the pattern-matching algorithms.

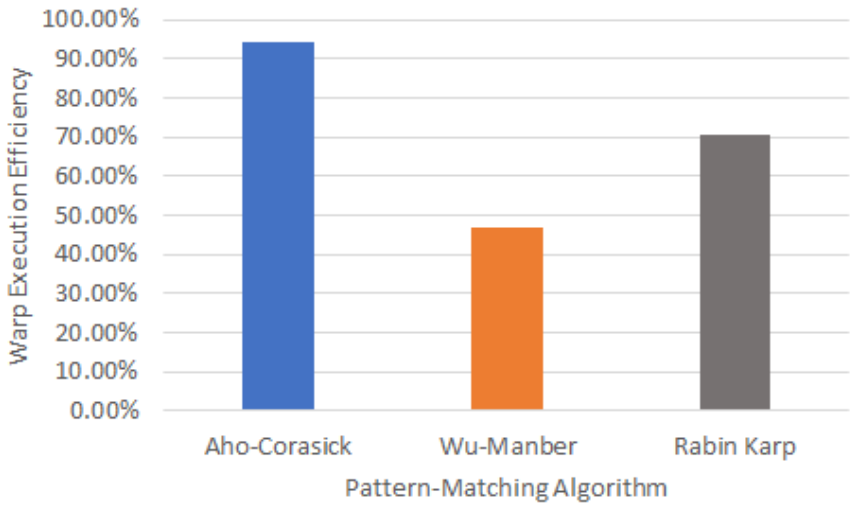

Figure 25: Average number of active threads executed in a warp

While performing the evaluations, the packets were crafted such that the payload of each packet had the pattern “FICDFICD” repeated multiple times. In the Wu-Manber algorithm, threads which have a shift value as 0 are active in the parallel region as shown in Fig. 26.

The threads calculate the hash value of the prefixes and iterative over the patterns with the same prefix. When the hash value of the prefix calculated matches the pattern prefix, then the thread becomes active in the parallel region as shown in Fig. 27. The number of threads, which evaluate the above if conditions to be true is low, and it reduces the warp execution efficiency. The pattern “FICDFICD” is repeated throughout the payload of the packet. As two threads out of eight threads have the first condition true, the warp execution throughput for the signature matching is 25% (8/32).

```
1   if ( shift == 0) {
```

Figure 26: Branch divergence in the Wu-Manber algorithm

```
1   // For every pattern with the same suffix as the text
2   for (int i = 0; i < d_PREFIX_size[hash1]; i++) {
3
4   // If the prefix of the pattern matches that of the text
5   if (hash2 == d_PREFIX_value[hash1 * prefixPitch + i]) {
```

Figure 27: Branch divergence in the Wu-Manber algorithm

In the Aho-Corasick algorithm, there are two conditions which cause branch divergence as shown in Fig. 28 and Fig. 29. If the first character of any pattern exists in the text, the first condition becomes true as shown in Fig. 28. If two characters of the pattern exist in the text, the while loop condition becomes true as shown in Fig. 29. Since the pattern "FICDFICD" appears in the text, two out of eight threads will have both the conditions true and they execute in the parallel region.

```
1   if ( nextState !=0) {
```

Figure 28: Branch divergence in the Aho-Corasick algorithm

```
1   while( nextState !=0 && pos<256) {
```

Figure 29: Branch divergence in the Aho-Corasick algorithm

Even though the number of threads active per warp for both the Wu-Manber and the Aho-Corasick algorithms are the same, the number of threads that use the execution units is lower for the Wu-Manber algorithm. Since the memory accesses are not coalesced, in the Wu-Manber algorithm, the memory transactions are serialized and the number of threads active per warp are reduced. In the Rabin-Karp algorithm, the if condition as shown in Fig. 30 is the cause for branch divergence. There are 749 patterns with length $\leq$ 64, 1441 patterns with $\leq$ 128 and 1743 patterns with length $\leq$ 256. Thus, all the threads in warps 3 to 8 will be active. Thread index 54 is the starting thread index for signature matching because the size of the header is 54. Warp 0 is used for checking the validation of the header. The warp efficiency in the Rabin-Karp algorithm is less than the Aho-Corasick algorithm because the percentage of pipeline stalls due to execution dependency is high.

```
1    if ( threadIdx .x <= 256 - patlen ) {
```

Figure 30: Branch divergence in the Rabin-Karp algorithm

The number of instructions per warp executed by the Rabin-Karp algorithm is very high, because each thread executes the algorithm for 1786 patterns as shown in Fig. 31. There are 20 instructions in the algorithm. Thus, total number of instructions executed by the warp is 1,143,040. In the Aho-Corasick and the Wu-Manber algorithms, the algorithm is executed only once for multiple patters and hence, the number of instructions executed per warp are lower.

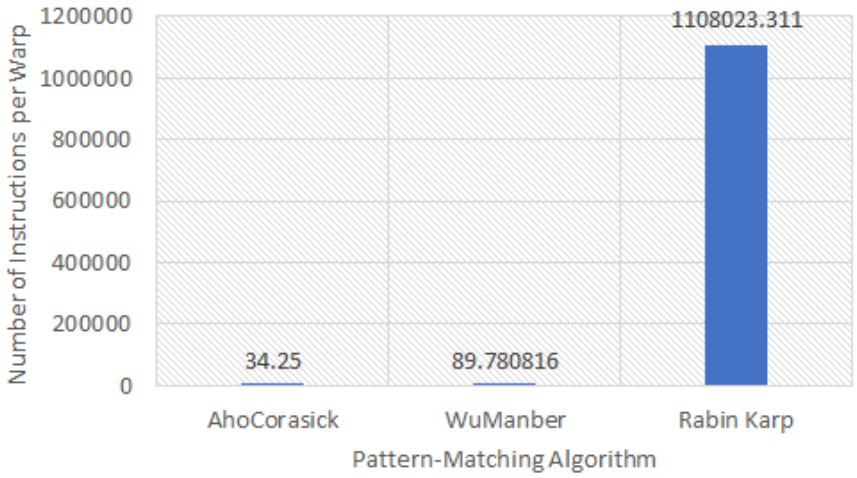

Figure 31: The number of instructions executed per warp

6.4.3 SM Utilization

Fig. 32 shows the average percentage of time each multiprocessor was utilized during the execution of the kernel. An SM is active when there is at least one warp currently executing on the SM. SM utilization is "The percentage of time at least one warp is active on a multiprocessor averaged over all multiprocessors on the GPU."

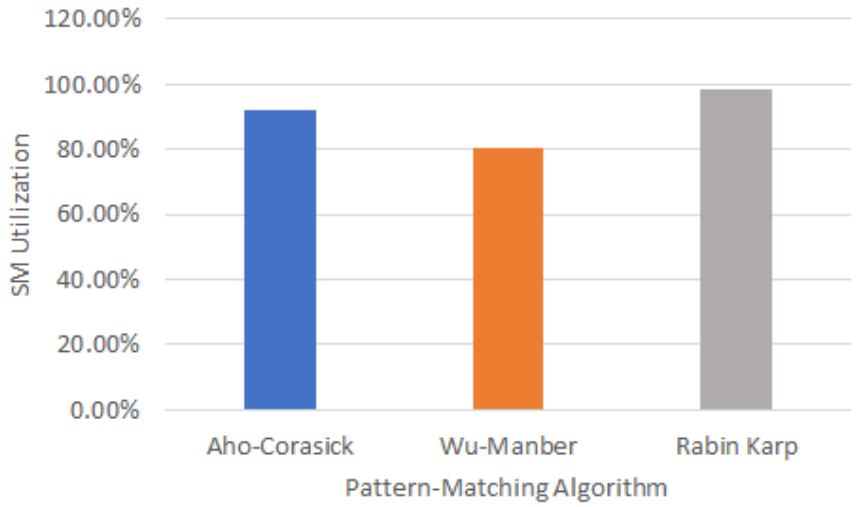

Figure 32: SM efficiency

An SM can become inactive even before the completion of the kernel execution due to three reasons, 1) different execution times of thread blocks, 2) different number of thread blocks scheduled per SM, and 3) combination of the first two reasons.

The phenomenon, where some thread blocks are idle and others are active, is known as tail effect. The idle processors can be utilized efficiently by concurrently launching multiple kernels. In the Wu-Manber and the Aho-Corasick algorithms, the threads will exit early if there is no trace of the pattern in the text. Hence, the thread blocks would have different execution times, and the SM efficiency would be reduced. However, in the evaluation, all the packets are similar and hence, the thread blocks do not have a significant difference in execution times. In the Rabin-Karp algorithm, all threads in warp 3 to 8 are active as explained in the previous section. Thus, the SM efficiency is very close to 100%. In the Aho-Corasick algorithm and the Wu-Manber algorithm, the warps are active during the search phase. In the Wu-Manber algorithm, different warps get scheduled on the SM to hide the memory latency. Hence, the percentage of time at least one warp is active is reduced.

6.5 Cache Hit Rate

L2 cache was introduced to reduce global memory access bottleneck. The policy used for caching is least recently used (LRU). As L1 cache is disabled by default, it needs to be enabled. In this thesis, only the L2 cache is used. The size of the L2 cache line is fixed at 32 bytes. In Kepler GPU, global memory loads are accessed through the L2 cache (or read-only data cache) [5]. In the Rabin-Karp algorithm, cache hit rate is very low because there is no data re-use. However, in the Aho-Corasick and the Wu-Manber algorithms, the tables generated during pre-processing are used often. Hence, the cache hit rate is high as shown in Fig. 33.

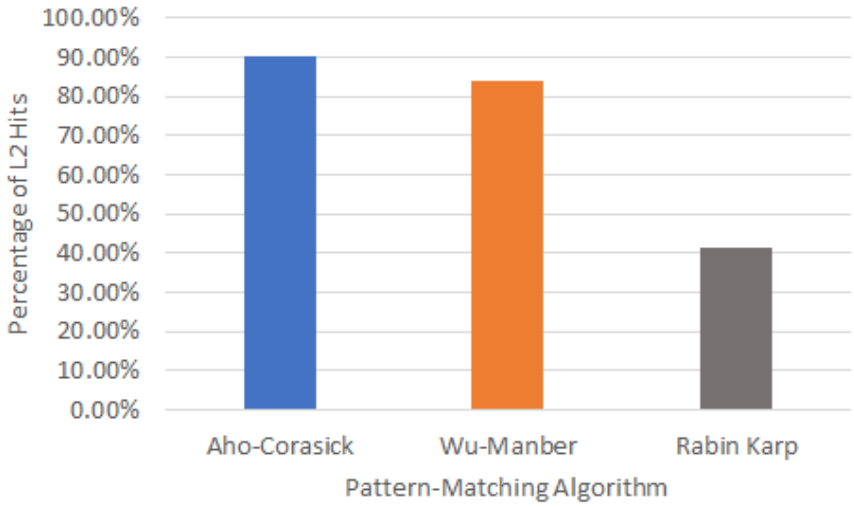

Figure 33: L2 hit rate

6.6 Pinned Memory Efficiency

As shown in Fig. 34, when pinned memory was used, the memory transactions between the CPU and the GPU reduced in the case of the Aho-Corasick and the Wu-Manber algorithms, and the time spent in executing the kernel increased. In the case of the Rabin-Karp algorithm, 98% of the time was spent in executing the kernel. Thus, there is no significant difference between the graphs.

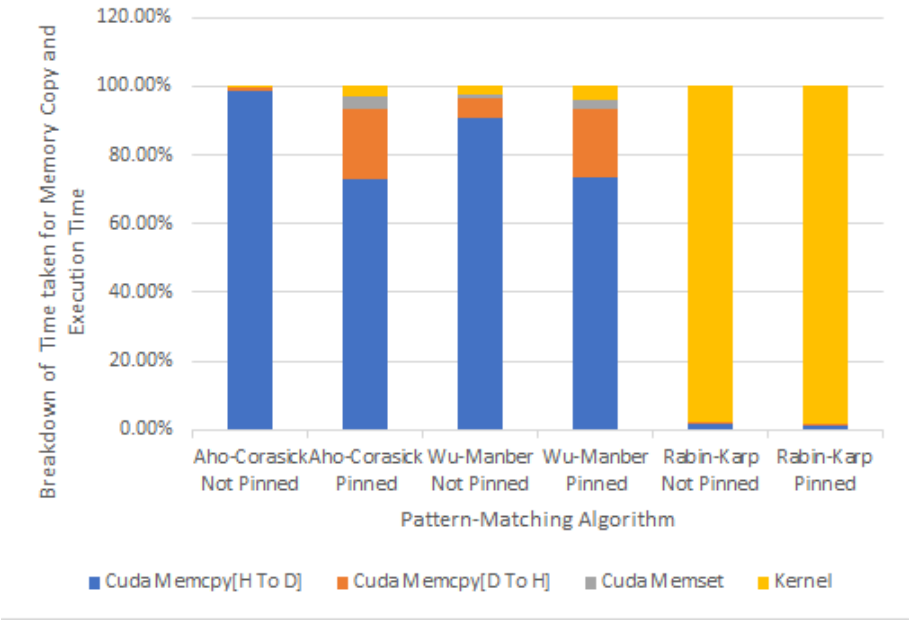

Figure 34: Pinned memory efficiency

CHAPTER 7

Related Work

The use of GPUs for packet processing is not a new invention. Gnort [10] an IDS, uses the parallel version of the Aho-Corasick algorithm [6] for signature matching. G. Vasiliadis et al. [10] demonstrated the packet processing with the GPU based Aho-Corasick algorithm, which outperformed Snort by a factor of two. Two pattern-matching strategies were tested. In the first method, a packet is processed by a stream processor. In the second method, a packet is processed by multiple stream processors cooperatively. They used pinned memory for asynchronous transfer from the CPU to the GPU. Unlike Gnort, my thesis provides wider and deeper analysis of GPU-version IDS design by implementing three signature-matching algorithms and comparing the performance with sequential and parallel CPU-version IDSs.

Many researchers have tried to improve the performance of string matching algorithms with the help of the GPU. For example, X. Zha and S. Sahni [12], developed Aho-Corasick and multi-pattern Boyer-Moore string matching algorithms to address the deficiencies in both the GPU to GPU case and CPU to CPU case. In the GPU to GPU case, the input and output are left in GPU memory, whereas in the CPU to CPU case, the input and output are in host memory. In the GPU to GPU case, the two deficiencies addressed are as follows:

1. Reading from global memory: The Tesla GPU reads data from global memory for 16 threads at a time and the total bandwidth for such a transaction is 128 bytes. As, each thread reads one byte at a time, a total of 16 bytes is read in a single transaction. Thus, only 1/8th of the total bandwidth for a transaction between the GPU and the SM will be utilized.
2. Coalescing multiple read transactions: The Tesla GPU coalesces all the transactions of a half-warp to a single transaction, if the data accesses are in the same 128-byte segment.

But, if the pattern length is greater than 128, the consecutive threads in a half-warp access the data, which are 128 bytes apart, with no coalescing.

X. Zha and S. Sahni [12] proposed a solution to the above problem by having multiple threads in a half-warp collectively access the global memory. In the CPU to CPU case, they proposed two approaches to overlap the I/O between CPU and GPU, and GPU computation using pinned memory.

The parallel failure-less Aho-Corasick algorithm (PFAC) is a variation of the Aho-Corasick algorithm implemented on a GPU [20]. It is remarkably distinct from the other implementations of this algorithm. Each thread is mapped to a character from the input text. Since each thread should search for a pattern from a particular position, there is no need for back tracking. Thus, the failure table was not used. The go-to array was stored in the shared memory, the patterns were grouped based on their prefixes and the patterns were saved in different multi-processors.

Snort [11] uses the Aho-Corasick [18] multi-pattern-matching algorithm for deep packet inspection, whereas another study [21] utilized the Boyer-Moore single pattern-matching algorithm [17]. The Boyer-Moore algorithm is a single pattern-matching algorithm. When the Boyer-Moore algorithm is used for multi-pattern matching, the time complexity is the product of the number of patterns and the size of the input text, whereas Snort uses a multi-pattern-matching algorithm and the time complexity is independent of the number of patterns and is linear to the size of the input text.

PixelSnort [22] is a modified version of the Knuth-Morris-Pratt algorithm [23] that offloads the computationally intensive packet processing to the GPU by grouping signatures and packets into textures. The throughput of PixelSnort outperformed Snort by 40% with respect to the rate of packet processing.

CHAPTER 8

Conclusion

In this thesis, I have presented a framework that utilizes the GPU for header checking and pattern matching. The classic Rabin-Karp algorithm, the Aho-Corasick algorithm, and the Wu-Manber algorithm were parallelized using CUDA to run on the GPU to perform pattern matching. Parallelism approaches explored were 1) thread-level parallelism, in which each thread analyzes a single packet and 2) block-level parallelism, in which a block of threads analyze a single packet. In my evaluation, block-level parallelism outperformed thread-level parallelism by a factor of 116, and the Aho-Corasick and the Wu-Manber algorithms outperformed the Rabin-Karp algorithm for multi-pattern matching. Furthermore, the GPU-version IDS outperformed at least 60 times over the CPU-version. Aho-Corasick algorithm performed better than the Wu-Manber algorithm in terms of shared memory efficiency, warp efficiency, SM utilization and pipeline stalls due to memory throttle. The Wu-Manber algorithm performed better than the Aho-Corasick algorithm in terms of global memory usage. The Rabin-Karp algorithm performed better than the other two algorithms with respect to SM utilization and memory throttle.

Global memory access efficiency was higher in the Rabin-Karp algorithm because the memory accesses are highly coalesced. The global memory usage for the Aho-Corasick algorithm was 10x and 1000x greater than the Wu-Manber algorithm and the Rabin-Karp algorithm, respectively. Because the Aho-Corasick and the Wu-Manber algorithms store large tables generated during the pre-processing stage in global memory. Based on the evaluation results, I conclude that the Aho-Corasick algorithm is ideal for DPI engines with a large number of patterns, the Wu-Manber algorithm works well for engines which require low memory consumption and the Rabin-Karp algorithm is suitable for a DPI engine which requires highly coalesced memory accesses.

In the future, I plan on applying these algorithms to FPGAs and compare the performance between the GPU and the FPGA. In the Aho-Corasick algorithm, there is an overhead associated with the amount of global memory used. To solve this, I plan to use multiple GPUs and break the FSM into multiple chunks and allocate each chunk to a GPU. Thus, I would be able to reduce the total global memory usage while decreasing the number of pipeline stalls that occurred due to memory latency. This would then lead to multiple gigabit per second throughput.

References

[1] J. van. (2011, Nov 10). *CUDA Thread Execution Model 3D Game* [Online]. Available: https://www.3dgep.com/cuda-thread-execution-model/.

[2] D. Cyca. (2014, Feb 10). *Maximizing Shared Memory Bandwidth on NVIDIA Kepler GPUs* [Online]. Available: http://acceleware.com/blog/maximizing-shared-memory-bandwidth-nvidia-kepler-gpus.

[3] M. Sune, "A framework for network traffic analysis using GPUs," M.S. thesis, Dept. Comp. Arch., Polytechnic Univ., Barcelona, CA, 2010.

[4] V. Jacobson et al., "The tcpdump manual page," Lawrence Berkeley Lab., Berkeley, CA, 1989, pp. 143.

[5] Nvidia Co. (2017) *Nvidia CUDA C Programming Guide* [Online]. Available: http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[6] N.P. Tran et al., "Performance Optimization of Aho-Corasick Algorithm on a GPU," in *12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications*, 2013, pp. 1143-1152.

[7] J. Martins et al. "ClickOS and the art of network function virtualization," in *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, 2014, pp. 459-473.

[8] J. McKennon. (2013, August 6). *GPU Memory Types – Performance Comparison* [Online]. Available: https://www.microway.com/hpc-tech-tips/gpu-memory-types-performance-comparison/.

[9] W. Sun and R. Ricci, "Fast and flexible: parallel packet processing with GPUs and click," in *Proceedings of the Ninth ACM/IEEE Symposium on Architectures for Networking and Communications Systems.* IEEE Press, 2013, pp. 25–36.

[10] G. Vasiliadis et al., "Gnort: High performance network intrusion detection using graphics processors," in *International Workshop on Recent Advances in Intrusion Detection,* 2008, pp. 116-134.

[11] M. Roesch, "Snort: Lightweight intrusion detection for networks," *Lisa*, vol. 99, no. 1, pp. 229-238, Nov. 1999.

[12] X. Zha and S. Sahni, "GPU-to-GPU and Host-to-Host Multipattern String Matching on a GPU," *IEEE Transactions on Computers*, vol. 62, no. 6, pp. 1156-1169, 2013.

[13] TN Thinh et al., "A FPGA-based deep packet inspection engine for Network Intrusion Detection System," in *International Conference on Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology (ECTI-CON)*, 2012, pp. 1-4.

[14] M. Harris, "High performance computing with CUDA," Tutorial in IEEE SuperComputing, 2007.

[15] S. Ryoo et al., "Optimization principles and application performance evaluation of a multithreaded GPU using CUDA," In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming,* 2008, pp. 73-82.

[16] S. Wu and U. Manber, "A fast algorithm for multi-pattern searching," Dept. of Comp. Sci., Univ.of Arizona, Arizona, Technical report TR-94-17, May 1994.

[17] Boyer R. S., and J. S. Moore, "A fast string searching algorithm," *Communications of the ACM 20*, pp. 762-772, October 1977.

[18] A. V. Aho and M.J. Corasick, "Efficient string matching: an aid to bibliographic search," *Communications of the ACM*, vol. 18, no. 6, pp. 333-340, 1975.

[19] R.M. Karp and M.O. Rabin, "Efficient randomized pattern-matching algorithms," *IBM Journal of Research and Development*, vol. 31, no. 2, pp. 249-260, 1987.

[20] C. Lui. (2014). *PFAC Library GPU-based string matching algorithm* [Online]. Available: http://code.google.com/p/pfac/.

[21] M. Jaiswal, "Accelerating Enhanced Boyer-Moore String Matching Algorithm on Multicore GPU for Network Security," In *International Journal of Computer Applications,* vol. 97, pp. 30-35, Jul 2015.

[22] N. Jacob and C. Brodley, "Offloading IDS Computation to the GPU," *Computer Security Applications Conference*, pp. 371-380, 2006.

[23] Mandumula and K. Kumar, "Knuth-Morris-Pratt Algorithm," *Poslední zmena 18*, 2011.

[24] Groleat et al., "Hardware acceleration of SVM-based traffic classification on FPGA," In *Wireless Communications and Mobile Computing Conference (IWCMC), 2012 8th International*, pp. 443-449, 2012.

[25] S. Feitoza et al., "Multi-gigabit traffic identification on GPU," In *Proceedings of the first edition workshop on High performance and programmable networking*, pp. 39-44, 2013.

[26] M. Harris. (2012, Dec 4). *How to Optimize Data Transfers in CUDA C/C++* [Online]. Available: https://devblogs.nvidia.com/parallelforall/how-optimize-data-transfers-cuda-cc/.

[27] J. Mu et al., "Accelerating pattern matching for DPI," In *SOC Conference, 2007 IEEE International*, pp. 83-86, 2007.

[28] (2017). *NVIDIA Tesla K80*, [ONLINE]. Available: http://www.nvidia.com/object/tesla-k80.html.

[29] W. S. Forn, "Generalizations of Horner's rule for polynomial evaluation," In *IBM Journal of Research and Development 6,* vol. 2, pp. 239-245, 1962.

[30] (2013). *NVIDIA Kepler GK110 Architecture Whitepaper*, [ONLINE]. Available: http://www.nvidia.com/content/PDF/kepler/NVIDIAKepler-GK110-Architecture-Whitepaper.pdf.

[31] L. Dagum and R. Menon, "OpenMP: an industry standard API for shared-memory programming," In *IEEE Computational Science and Engineering*, pp. 46-55, 1998.

APPENDIX

Code snippets

A.1 Header Checking in Block-Level Parallelism

```
__shared__ T elements[256];
__shared__ clock_t starttime;
__shared__ clock_t stoptime;
__shared__ uchar4 packet[64];

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

//Mine to Shared to save memory accesses
if(threadIdx.x ==0)
starttime = clock();

elements[threadIdx.x] = cudaSafeGet(&GPU_data[threadIndex]);
//Copying the state 0 information to shared memory
// printf("Copying to shared memory");
__syncthreads();

//  printf("In header checking");
if(threadIdx.x == 14) // Representative thread for warp 0 , Check incorrect version and private
    address range
{
if(IS_IP4()) //Checking if it is a IPv4 Packet
{
GPU_results[blockIdx.x].ipPacket = 1;
int verBlock = (elements[14].packet & 0x000000F0) >> 4;
if(verBlock !=4 && verBlock !=6) GPU_results[blockIdx.x].maliciousVer=1;

int octet1 = elements[26].packet & 0x000000FF;
```

```
27 int octet2 = elements[27].packet & 0x000000FF;
28 int octet1Dst = elements[30].packet & 0x000000FF;
29 int octet2Dst = elements[31].packet & 0x000000FF;
30
31 if (octet1 == 10 || octet1Dst == 10)
32 {
33 GPU_results[blockIdx.x].maliciousIP=1;
34 }
35 // 172.16.0.0 – 172.31.255.255
36 if ((octet1 == 172 || octet1Dst == 172) && (octet2 >= 16 || octet2Dst >=16) && (octet2 <= 31
       || octet2Dst<=31))
37 {
38 GPU_results[blockIdx.x].maliciousIP=1;
39 }
40 // 192.168.0.0 – 192.168.255.255
41 if ((octet1 == 192 || octet1Dst == 192) && (octet2 == 168 || octet2Dst == 168))
42 {
43 GPU_results[blockIdx.x].maliciousIP=1;
44 }
45 }
46 }
47 if(threadIdx.x == 33) //Checksum verification
48 {
49 if(IS_IP4()) //check if a IPv4 Packet
50 {
51 int checksum = ((elements[14].packet & 0x000000FF)<<8 + (elements[15].packet & 0x000000FF)) +
52 ((elements[16].packet & 0x000000FF)<<8 + (elements[17].packet & 0x000000FF)) +
53 ((elements[18].packet & 0x000000FF)<<8 + (elements[19].packet & 0x000000FF )) +
54 ((elements[20].packet & 0x000000FF)<<8 + (elements[21].packet & 0x000000FF)) +
55 ((elements[22].packet & 0x000000FF)<<8 + (elements[23].packet & 0x000000FF)) +
56 ((elements[24].packet & 0x000000FF)<<8 + (elements[25].packet & 0x000000FF)) +
```

```
(lastTwoBytes(elements[26].packet)<<8 + lastTwoBytes(elements[27].packet)) +
(lastTwoBytes(elements[28].packet)<<8 + lastTwoBytes(elements[29].packet)) +
(lastTwoBytes(elements[30].packet)<<8 + lastTwoBytes(elements[31].packet)) +
(lastTwoBytes(elements[32].packet)<<8 + lastTwoBytes(elements[33].packet));

unsigned int sum = ~(checksum>>16 + (checksum & 0xFFFF));
if(sum!=-1) GPU_results[blockIdx.x].maliciousCheckSum = 1;
}
}

if(threadIdx.x == 64) // check sport or dport is 0; check ackNo is o, with ack bit set; check
    malicious flag bit combinations
{
if(IS_TCP())
{
GPU_results[blockIdx.x].flags = lastTwoBytes(elements[47].packet);
GPU_results[blockIdx.x].sport = lastTwoBytes(elements[34].packet) << 8 +
    lastTwoBytes(elements[35].packet);
if(lastTwoBytes(elements[33].packet) == 255)
GPU_results[blockIdx.x].maliciousDst = 1;

if((lastTwoBytes(elements[34].packet) == 0 && lastTwoBytes(elements[35].packet) == 0) ||
(lastTwoBytes(elements[36].packet) == 0 && lastTwoBytes(elements[37].packet) == 0))
GPU_results[blockIdx.x].maliciousPort = 1;
int ackNo = lastTwoBytes(elements[42].packet)<<24 + lastTwoBytes(elements[43].packet)<<16 +
    lastTwoBytes(elements[44].packet)<<8 + lastTwoBytes(elements[45].packet);
if(lastTwoBytes(elements[47].packet & 16) ==1 && (ackNo == 0))
GPU_results[blockIdx.x].maliciousAck = 1;

int reservedVal = lastTwoBytes(elements[46].packet) >> 4;
if(reservedVal != 0) GPU_results[blockIdx.x].maliciousReserved = 1;
```

```
85
86  int flagVal = lastTwoBytes(elements[47].packet);
87
88  if(flagVal == 3 || flagVal == 11 || flagVal == 7 || flagVal == 15 || flagVal == 1 || flagVal ==
        0)
89  GPU_results[blockIdx.x].maliciousFlags = 1;
90
91  }
92  }
93
94  if(IS_IP4() && threadIdx.x == 33) // save source and destn IP addres to output
95  {
96  GPU_results[blockIdx.x].ipPacket = 1;
97  uint32_t ipSrc = ((elements[26].packet & 0x000000FF) << 24) + ((elements[27].packet &
        0x000000FF) << 16)
98  + ((elements[28].packet & 0x000000FF )<< 8) + (elements[29].packet & 0x000000FF);
99  uint32_t ipDst = ((elements[30].packet & 0x000000FF) << 24) + ((elements[31].packet &
        0x000000FF) << 16)
100 + ((elements[32].packet & 0x000000FF )<< 8) + (elements[33].packet & 0x000000FF);
101
102 GPU_results[blockIdx.x].ipSrc = ipSrc;
103 GPU_results[blockIdx.x].ipDst = ipDst;
104 }
```

A.2 Rabin-Karp Algorithm

A.2.1 Sequential Pattern-Matching Algorithm using C

```
1   long hashCal(const char* pattern, int m, int offset) {
2   long h = 0;
3   for (int j = 0; j < m; j++) {
4   h = (256 * h + pattern[offset + j]) % 997;
5   }
```

```
6   return h;
7   }
```

```
1   // Starting from 0, move for every pattern length, computing the hash values
2   // Time complexity O(N* number of pattern lengths)
3   // Tmp is the vector of patterns
4   for(int i=0;i<tmp.size();i++)
5   setlen.insert(tmp[i].length());
6
7   // Fill the map with pattern hashes
8   for(int i=0;i<tmp.size();i++)
9   {
10  long patHash = hashCal(tmp[i].c_str(), tmp[i].size(),0);
11  mapHash[patHash] = i;
12  }
13
14  // Choosing a large prime
15  int q = 997;
16  int R = 256;
17
18  for(auto it = setlen.begin(); it != setlen.end(); it++)
19  {
20  int m = *it;
21  int RM = 1;
22  for (int i = 1; i <= m-1; i++)
23  RM = (256 * RM) % 997;
24
25  if (m > payLoadLength) break;
26  int txtHash = hashCal((char*) packetPointer, m,0);
27
28  // check for match at offset 0
```

```
29    if ((mapHash[txtHash]>0) && memcmp((char*)packetPointer,
30    tmp[mapHash[txtHash]].c_str(),m)==0)
31    { cout<<"Virus Pattern " << tmp[mapHash[txtHash]] <<" exists "<<endl; break;}
32
33    // check for hash match; if hash match, check for exact match
34    for (int j = m; j < payLoadLength; j++) {
35    // Remove leading digit, add trailing digit, check for match.
36    txtHash = (txtHash + q - RM*packetPointer[j-m] % q) % q;
37    txtHash = (txtHash*R + packetPointer[j]) % q;
38
39    // match
40    int offset = j - m + 1;
41    if ((mapHash[txtHash]>0) &&
42    memcmp((char*) (packetPointer + offset), tmp[mapHash[txtHash]].c_str(),m)==0)
43    { cout<<"Virus Pattern " << tmp[mapHash[txtHash]] <<" exists "<<endl; break;}
44    }
45    }
```

A.2.2 Parallel Pattern-Matching Algorithm using CUDA

```
1  /*Rabin Karp Muli-pattern string matching implementation*/
2  if(threadIdx.x >= 54) {
3  GPU_results[0].num_strings = const_num_strings;
4  for(int i=0;i<const_num_strings;i++) {
5  int patLen = const_indexes[2*i+1] - const_indexes[2*i];
6  //This condition checks if the pattern length is < packet length
7  if(threadIdx.x<=256-patLen) {
8  int hy,j;
9  for(hy=j=0;j<patLen;j++) {
10 if((j+threadIdx.x) >= 256) goto B;
11 hy = (hy * 256 + elements[j+threadIdx.x].packet) % 997;
12 }
```

```
13 if (hy == const_patHash[i] &&
       memCmpDev<T>(elements,const_pattern,const_indexes,i,threadIdx.x,patLen) == 0) // Complete
       this
14 {
15 GPU_results[blockIdx.x].maliciousPayload = 1;
16 GPU_results[blockIdx.x].signatureNumber = i;
17 d_result[i]=1;
18 }
19 }
20 B:
21 }
22 }
```

A.3 Wu-Manber Algorithm

A.3.1 Parallel Pattern-Matching Algorithm using OpenMP

A.3.1.1 Pre-Processing Phase

```
1  void preproc_wuManber(vector<string> pattern, int m, int B,
2  int *SHIFT, int *PREFIX_value, int *PREFIX_index, int *PREFIX_size) {
3
4  int p_size = pattern.size();
5  DEBUG2("p_size= %d",p_size);
6  #pragma omp parallel for
7  for (int j = 0; j < p_size; ++j) {
8  int threadNum = omp_get_thread_num();
9  DEBUG2("ThreadNum= %d",threadNum);
10 /* Don't want to add #pragma for the inner loop because
11 * you may need to use the previous value of SHIFT[hash]
12 * in the future loops, reduction is used if the data needs to be
13 * gathered together at the end.
14 */
15 //add each 3-character subpattern (similar to q-grams)
```

```
16
17 }
18 }
```

A.3.1.2 Search Phase

```
1  unsigned int search_wuManber(vector<string> pattern , int m,
2  string text , int n, int *SHIFT, int *PREFIX_value,
3  int *PREFIX_index, int *PREFIX_size) {
4
5  int column = m - 1;
6
7  unsigned int hash1, hash2;
8
9  unsigned int matches = 0;
10
11 size_t shift ;
12 int p_size = pattern . size () ;
13
14 const char* textC = text . c_str () ;
15
16 for (column = m - 1;column < n;) {
17
18 hash1 = text [column - 2];
19 hash1 <<= m_nBitsInShift;
20 hash1 += text [column - 1];
21 hash1 <<= m_nBitsInShift;
22 hash1 += text [column];
23
24 shift = SHIFT[hash1];
25
26 // printf ("column %i hash1 = %i shift = %i\n", column, hash1, shift ) ;
```

```
27
28 if ( shift == 0) {
29
30 hash2 = text[column - m + 1];
31 hash2 <<= m_nBitsInShift;
32 hash2 += text[column - m + 2];
33 volatile bool flag=false;
34 // printf ("hash2 = %i PREFIX[hash1].size = %i\n", hash2, PREFIX[hash1].size);
35
36 //For every pattern with the same suffix as the text
37 #pragma omp parallel for
38 for (int i = 0; i < PREFIX_size[hash1]; i++) {
39 // if (flag) continue;
40 // If the prefix of the pattern matches that of the text
41 if (hash2 == PREFIX_value[hash1 * p_size + i]) {
42
43 //Compare directly the pattern with the text
44 if (memcmp(pattern[PREFIX_index[hash1 * p_size + i]].c_str(),
45 textC + column - m + 1, m) == 0) {
46
47 #pragma omp critical
48 {
49 matches++;
50 }
51 // flag = true;
52 printf ("Match of pattern index %i at %i\n", PREFIX_index[hash1 * p_size + i], column-m+1);
53 }
54
55 }
56 }
57
```

```
58 column++;
59 } else
60 column += shift ;
61 }
62
63 return matches;
64 }
```

A.3.2 Parallel Pattern-Matching Algorithm using CUDA

Each thread starts searching from its thread Id. elements array contains the 256 byte packet and is stored in shared memory.

```
1  unsigned int hash1, hash2;
2  if (threadIdx.x >= 54 + m-1)
3  {
4  hash1 = elements[threadIdx.x - 2].packet & 0x000000FF; //bitwise & used because to avoid two
       complement negative numbers
5  hash1 <<= 2;
6  hash1 += elements[threadIdx.x - 1].packet & 0x000000FF;
7  hash1 <<= 2;
8  hash1 += elements[threadIdx.x].packet & 0x000000FF;
9
10 int shift = d_SHIFT[hash1];
11
12 if (shift == 0) {
13
14 hash2 = elements[threadIdx.x - m + 1].packet & 0x000000FF;
15 hash2 <<= 2;
16 hash2 += elements[threadIdx.x - m + 2].packet & 0x000000FF;
17
18 //For every pattern with the same suffix as the text
19 for (int i = 0; i < d_PREFIX_size[hash1]; i++) {
```

```
20
21 // If the prefix of the pattern matches that of the text
22 if (hash2 == d_PREFIX_value[hash1 * prefixPitch + i]) {
23
24
25 int patIndex = d_PREFIX_index[hash1* prefixPitch + i];
26
27 int starttxt = threadIdx.x - m + 1 + 2;
28 int startpat = d_stridx[2*patIndex] + 2;
29 int endpat = d_stridx[2*patIndex+1];
30 // memcmp implementation
31 while(elements[ starttxt ].packet!='\0' && startpat < endpat) {
32 if(elements[ starttxt ++].packet!=d_pattern[ startpat ++]) return;
33 }
34 if( startpat >= endpat) {
35 printf("The pattern exists %d\n", patIndex);
36 GPU_results[blockIdx.x].maliciousPayload = 1;
37 result[blockIdx.x] = patIndex;
38 }
39 }
40 }
41 }
42 }
```

A.4 Aho-Corasick algorithm

A.4.1 Sequential Pattern-Matching Algorithm using C

A.4.1.1 Pre-Processing Phase

```
1 // Returns the number of states that the built machine has.
2 // States are numbered 0 up to the return value - 1, inclusive.
3 int buildMatchingMachine(vector<string> arr, int k)
```

```
4  {
5  //  Initialize  all  values  in  output  function  as  0.
6
7
8  //  Initialize  all  values  in  goto  function  as  -1.
9  memset(g, -1, sizeof  g);
10
11 //  Initially , we just have the 0 state
12 int  states  = 1;
13
14 // Construct values for goto function , i.e .,  fill  g[][]
15 // This is same as building a Trie for arr[]
16
17 omp_set_dynamic(0);
18 omp_set_num_threads(4);
19
20 //#pragma omp parallel  for
21 for  (int  i = 0; i < k; ++i)
22 {
23 const  string  &word = arr[i];
24 int  currentState  = 0;
25  printf ("tid = %d",omp_get_thread_num());
26 //  Insert  all  characters  of  current  word in  arr[]
27 for  (int  j = 0; j < word.size(); ++j)
28 {
29 int  ch = word[j];
30
31 //  Allocate  a  new node (create  a  new  state)  if  a
32 //  node for  ch doesn't  exist .
33 if  (g[currentState][ch] == -1)
34 g[currentState][ch] = states++;
```

```
35
36  currentState  = g[ currentState ][ch];
37  }
38
39  // Add current word in output function
40
41  out[ currentState ] |= ( patterns << i);
42
43  }
44
45  // For all characters which don't have an edge from
46  // root (or state 0) in Trie, add a goto edge to state
47  // 0 itself
48  for (int ch = 0; ch < MAXC; ++ch)
49  if (g[0][ch] == -1)
50  g[0][ch] = 0;
51
52  // Now, let's build the failure function
53
54  // Initialize values in fail function
55  memset(f, -1, sizeof f);
56
57  // Failure function is computed in breadth first order
58  // using a queue
59  queue<int> q;
60
61  // Iterate over every possible input
62  for (int ch = 0; ch < MAXC; ++ch)
63  {
64  // All nodes of depth 1 have failure function value
65  // as 0. For example, in above diagram we move to 0
```

```
66 // from states 1 and 3.
67 if (g[0][ch] != 0)
68 {
69 f[g[0][ch]] = 0;
70 q.push(g[0][ch]);
71 }
72 }
73
74 // Now queue has states 1 and 3
75 while (q.size())
76 {
77 // Remove the front state from queue
78 int state = q.front();
79 q.pop();
80
81 // For the removed state, find failure function for
82 // all those characters for which goto function is
83 // not defined.
84 for (int ch = 0; ch < MAXC; ++ch)
85 {
86 // If goto function is defined for character 'ch'
87 // and 'state'
88 if (g[state][ch] != -1)
89 {
90 // Find failure state of removed state
91 int failure = f[state];
92
93 // Find the deepest node labeled by proper
94 // suffix of string from root to current
95 // state.
96 while (g[failure][ch] == -1)
```

```
97  failure = f[failure];
98
99  failure = g[failure][ch];
100 f[g[state][ch]] = failure;
101
102 // Merge output values
103 out[g[state][ch]] |= out[failure];
104
105 // Insert the next level node (of Trie) in Queue
106 q.push(g[state][ch]);
107 }
108 }
109 }
110
111 return states;
112 }
113
114 // Returns the next state the machine will transition to using goto
115 // and failure functions.
116 // currentState - The current state of the machine. Must be between
117 //                0 and the number of states - 1, inclusive.
118 // nextInput - The next character that enters into the machine.
119 int findNextState(int currentState, char nextInput)
120 {
121 int answer = currentState;
122 int ch = nextInput;
123
124 // If goto is not defined, use failure function
125 while (g[answer][ch] == -1)
126 answer = f[answer];
127
```

```
return g[answer][ch];
}
```

A.4.1.2 Search Phase

```
void Dissector::searchWords(vector<string> arr, int k, string text)
{
// Preprocess patterns.
// Build machine with goto, failure and output functions
buildMatchingMachine(arr, k);
cout<<"Completed building machine"<<endl;
// Initialize current state
// Traverse the text through the nuilt machine to find
// all occurrences of words in arr[]
int currentState = 0;
for (int i = 0; i < text.size(); ++i)
{
int tid = omp_get_thread_num();
int nthreads = omp_get_num_threads();

// cout<<"threadID = "<<tid<<"loop index= "<<i<<endl;
int pos = i;
currentState = findNextState(currentState, text[pos]);
// cout<<tid<<" "<<pos<<" "<<currentState<<" "<<endl;
// If match not found, move to next state
if (out[currentState] == checkPattern)
{
pos = pos + 1;
continue;
}
// Match found, print all matching words of arr[]
// using output function.
for (int j = 0; j < k; ++j)
```

```
{
if ((out[currentState] & (patterns << j)) != checkPattern)
{
cout << "Word " << arr[j] << " appears from "
<< pos - arr[j].size() + 1 << " to " << pos << endl;
}
}
}
}
```

A.4.2 Parallel Pattern-Matching Algorithm using OpenMP

OpenMP directive “pragma omp parallel for” is added to the search phase of the algorithm. The pre-processing phase has to be done sequentially by a single thread, because the states are constructed as a result of the previous states and the new character. In the search phase, the algorithm was evaluated by 256 threads and each thread starts searching for the patterns from its thread Id. The for loop is split into 256 chunks and each chunk is assigned to a thread.

```
\caption{Search Phase}
// This function finds all occurrences of all array words
// in text.
void Dissector::searchWords(vector<string> arr, int k, string text)
{
// Preprocess patterns.
// Build machine with goto, failure and output functions
buildMatchingMachine(arr, k);
cout<<"Completed building machine"<<endl;
// Initialize current state
// Traverse the text through the nuilt machine to find
// all occurrences of words in arr[]

```

```
14 int nProcessors = omp_get_max_threads();
15 // cout<<"max threads = "<<nProcessors<<endl;
16 omp_set_dynamic(0);
17 omp_set_num_threads(256);
18 omp_set_dynamic(1);
19 #pragma omp parallel for
20 for (int i = 0; i < text.size(); ++i)
21 {
22 int tid = omp_get_thread_num();
23 int nthreads = omp_get_num_threads();
24
25 // cout<<"threadID = "<<tid<<"loop index= "<<i<<endl;
26 int pos = i;
27 int currentState = 0;
28 while(pos<text.size()) {
29 currentState = findNextState(currentState, text[pos]);
30 // If match not found, move to next state
31 if (out[currentState] == checkPattern)
32 {
33 pos = pos + 1;
34 continue;
35 }
36 // Match found, print all matching words of arr[]
37 // using output function.
38 for (int j = 0; j < k; ++j)
39 {
40 if ((out[currentState] & (patterns << j))!=checkPattern)
41 {
42 cout << "Word " << arr[j] << " appears from "
43 << pos - arr[j].size() + 1 << " to " << pos << endl;
44 }
```

```
45 }
46 pos = pos + 1;
47 }
48 }
49 }
```