

Computer Science Department
University of Crete

*Improving the Performance of Network Intrusion Detection
Systems Using Graphics Processors*

Master's Thesis

Giorgos Vasiliadis

November 2008
Heraklion, Greece

University of Crete
Computer Science Department

**Improving the Performance of Network Intrusion Detection Systems
Using Graphics Processors**

Thesis submitted by

Giorgos Vasiliadis

in partial fulfillment of the requirements for the
Master of Science degree in Computer Science

THESIS APPROVAL

Author: _____

Giorgos Vasiliadis

Committee approvals: _____

Evangelos P. Markatos

Professor, Thesis Supervisor

Sotiris Ioannidis

Visiting Professor

Angelos Bilas

Associate Professor

Departmental approval: _____

Panos Trahanias

Professor, Chairman of Graduate Studies

Heraklion, November 2008

Abstract

The constant increase in link speeds and number of threats poses challenges to network intrusion detection systems (NIDSes), which must cope with higher traffic throughput and perform even more complex per-packet processing. Pattern matching is the most expensive operation of a signature-based NIDS, both in terms of executed CPU instructions and memory consumption. Results have shown that pattern matching consumes up to 75% of a NIDS' CPU time.

In this thesis, we present an intrusion detection system based on the Snort open-source NIDS that exploits the underutilized computational power of modern graphics cards to offload the costly pattern matching operations from the CPU, and thus increase the overall processing throughput. Graphic cards processing units (GPUs) provide high parallelism and offer significant computational power. Their SPMD (Single Process, Multiple Data) operation is ideal for pattern matching purposes since we can search in parallel for multiple patterns over multiple packets. We ported single- and multi-pattern matching algorithms to the GPU and compared their performance with the corresponding CPU implementations. Our prototype system, called *Gnort*, achieved a maximum traffic processing throughput of 2.3 Gbit/s using synthetic network traces, while when monitoring real traffic using a commodity Ethernet interface, it outperformed unmodified Snort by a factor of two.

The results suggest that modern graphics cards can be used effectively to speed up intrusion detection systems, as well as other systems that involve pattern matching operations up to 8x times faster.

Supervisor: Evangelos Markatos, Professor

Περίληψη

Η σταθερή αύξηση στις ταχύτητες δικτύων σε συνδυασμό με την αύξηση των απειλών θέτει νέες προκλήσεις στα Συστήματα Ανίχνευσης Επιθέσεων Δικτύων (Network Intrusion Detection Systems, NIDSes), τα οποία πρέπει να αντιμετωπίσουν την ολοένα και αυξανόμενη κυκλοφορία στα δίκτυα υπολογιστών και να εκτελέσουν ακόμα πιο σύνθετες διαδικασίες ανά πακέτο δικτύου. Το ταίριασμα προτύπων (pattern matching) είναι η πιο ακριβή λειτουργία ενός συστήματος ανίχνευσης επιθέσεων, τόσο από επεξεργαστική άποψη όσο και από άποψη κατανάλωσης μνήμης. Πρόσφατες μετρήσεις έχουν δείξει ότι το ταίριασμα προτύπων καταναλώνει μέχρι και 75% από το συνολικό χρόνο εκτέλεσης ενός συστήματος ανίχνευσης επιθέσεων.

Σε αυτήν την εργασία, παρουσιάζουμε ένα σύστημα ανίχνευσης επιθέσεων βασισμένο στο λογισμικό ανοικτού-κώδικα Snort NIDS, το οποίο εκμεταλλεύεται την υπολογιστική δύναμη των σύγχρονων καρτών γραφικών για να αποφορτώσει την 'δαπανηρή' διαδικασία ταίριασματος προτύπων από τον επεξεργαστή (CPU) και να αυξήσει έτσι τη συνολική απόδοση του συστήματος. Οι επεξεργαστές γραφικών (GPUs) παρέχουν υψηλό παραλληλισμό και προσφέρουν σημαντικές υπολογιστικές δυνατότητες. Το SPMD (Single Process, Multiple Data) μοντέλο εκτέλεσης που παρέχουν είναι ιδανικό για ταίριασμα προτύπων δεδομένου ότι μπορούμε να ψάξουμε παράλληλα για πολλαπλά πρότυπα μέσα σε πολλαπλά πακέτα δικτύου. Υλοποιήσαμε single- και multi-pattern matching αλγόριθμους για εκτέλεση στην GPU και συγκρίναμε την απόδοσή τους με τις αντίστοιχες υλοποιήσεις στην CPU. Το πρωτότυπο σύστημα μας, αποκαλούμενο Gsnort, πέτυχε μια μέγιστη απόδοση στην επεξεργασία δικτυακής κυκλοφορίας ίση με 2.3 Gbit/s χρησιμοποιώντας τεχνητή κίνηση δικτύου, ενώ όταν το χρησιμοποιήσαμε σε πραγματική κίνηση χρησιμοποιώντας μια απλή διεπαφή δικτύου Ethernet, είχε δύο φορές μεγαλύτερη απόδοση από το Snort.

Τα αποτελέσματα μας προτείνουν ότι οι σύγχρονες κάρτες γραφικών μπο-

ρούν να χρησιμοποιηθούν αποτελεσματικά για να επιταχύνουν τόσο τα Συστήματα Ανίχνευσης Επιθέσεων, καθώς επίσης και άλλα συστήματα που περιλαμβάνουν διαδικασίες ταιριάσματος προτύπων έως και 8x φορές πιο γρήγορα.

Επόπτης: Ευάγγελος Μαρκάτος, Καθηγητής

Acknowledgments

I would like to thank my Supervisor, Professor Evangelos P. Markatos, for his valuable guideline in my academic steps in the field of Computer Science, and for a real commitment to my technical and professional growth. I also feel grateful to Dr. Sotiris Ioannidis, who gave me the opportunity to work on this subject and whose contribution was a key for writing this thesis. I am really pleased to cooperate with people of their mental and ethical values.

My best regards to Michalis Polychronakis and Spiros Antonatos for their joint work on Gnort, and to the rest of the Distributed Computing Systems group, Alexandros Kapravelos, Nikos Nikiforakis, Vasilis Pappas, Andreas Makridakis, Antonis Papadogiannakis, Manos Athanatos, Elias Athanasopoulos, Iasonas Polakis, Vasilis Lekakis, Giannis Velegrakis, Giorgos Kondaxis, Michalis Foukarakis, Demetres Antoniadis, Christos Pappachristos, and Manolis Stamatogiannakis that contributed for a pleasant and productive environment all these years in the lab.

Finally, I want to thank anyone else who I did not explicitly mention for all your support.

Στη μητέρα μου.

To my mother.

An early report on this work appeared in the proceedings of the 11th International Symposium on Recent Advances in Intrusion Detection (RAID'2008), September 2008 [35].

Contents

1 Introduction	1
1.1 Thesis organisation	3
2 Background	5
2.1 The GeForce 8 Series Architecture	5
2.2 Pattern Matching Algorithms	7
2.2.1 Single-Pattern Matching Algorithms	7
2.2.2 Multi-Pattern Matching Algorithms	8
2.3 Network Intrusion Detection Systems and Pattern Matching .	10
3 Architecture	13
3.1 Transferring Packets to the GPU	13
3.2 Pattern Matching on the GPU	15
3.3 Transferring the Results to the CPU	17
3.4 Execution flow overview	18
4 Implementation	23
4.1 Assigning a Single Packet to each Multiprocessor	26
4.2 Assigning a Single Packet to each Stream Processor	28
5 Evaluation	31
5.1 Performance of single- and multi-pattern algorithms	32

5.2 Experiments with Real Input and Rules	35
5.2.1 Microbenchmarks	35
5.2.2 Macrobenchmarks	39
6 Related Work	41
7 Conclusions	45

List of Figures

2.1	Aho-Corasick pattern matching machine example (adapted from [1]).	9
3.1	Overall architecture of Gnort.	14
3.2	Execution flow of Gnort. GPU communication and computation may hidden by overlapping it with CPU execution.	19
4.1	Different pattern matching parallelization approaches. In (a), a different packet is processed by each multiprocessor. All stream processors in the multiprocessor search the packet payload concurrently. In (b), a different packet is processed by each stream processor independently of the others.	27
5.1	Throughput sustained for single-pattern matching algorithms.	33
5.2	Throughput sustained for multiple-pattern matching algorithms.	33
5.3	Throughput sustained for single-pattern matching algorithms for various packet sizes.	34
5.4	Throughput sustained for multiple-pattern matching algorithms for various packet sizes.	34
5.5	Average execution times per inspected packet spent on each task.	36

5.6	Packet matching computational throughput for CPU and GPU. In (a) the state machine tables are stored in Global device mem- ory, while in (b) in Texture memory.	37
5.7	Overall performance as a function of the traffic speed.	39

List of Tables

3.1 Snort Data Structures.	15
------------------------------------	----

1

Introduction

Network security architectures such as firewalls and Network Intrusion Detection Systems (NIDSes) attempt to detect break-in attempts by monitoring the incoming and outgoing traffic for suspicious payloads. Most modern network intrusion detection and prevention systems rely on a set of rules that are compared against network packets. Usually, a rule consists of a filter specification based on packet header fields, a string that must be contained in the packet payload, the approximate or absolute location where that string should be present, and an associated action to take if all the conditions of the rule are met.

Signature matching is a highly computationally intensive process, accounting for about 75% of the total CPU processing time of modern NID-

Ses [2, 7]. This overhead arises from the fact that most of the time, every byte of every packet needs to be processed as part of the string searching algorithm that searches for matches among a large set of strings from all signatures that apply for a particular packet. For example, the rule set of Snort [27], one of the most widely used open-source NIDS, contains about 10000 strings. Searching every packet for all of these strings requires significant resources, both in terms of the computation capacity needed to process a packet, as well as the amount of memory needed to store the rules.

Several research efforts have explored the use of parallelism for improving the packet processing throughput [4, 8, 14, 26, 39]. Specialized hardware devices can be used to inspect many packets concurrently, and such devices include ASICs and Network Processors. Both are very efficient and perform well, however they are complex to modify and program. Moreover, FPGA-based architectures have poor flexibility since most of the approaches are usually tied to a specific implementation.

As Graphics Processing Units (GPUs) are becoming increasingly powerful and ubiquitous, researchers have begun exploring ways to tap their power for non-graphic or general-purpose (GPGPU) applications. The main reason behind this evolution is that GPUs are specialized for computationally-intensive and highly parallel operations—required for graphics rendering—and therefore are designed such that more transistors are devoted to data processing rather than data caching and flow control [24]. The release of software development kits (SDKs) from big vendors, like NVIDIA¹ and ATI,² has started a trend of using GPUs as a computational unit to offload the CPU.

In addition, many attempts have been made to use graphics processors for security purposes, including cryptography [11], data carving [21] and

¹<http://developer.nvidia.com/object/cuda.html>

²<http://ati.amd.com/technology/streamcomputing/index.html>

intrusion detection [18]. Specifically, it has been shown that GPU support can substantially increase the performance of digital forensics software that relies on binary string searches [21]. Jacob and Brodley were the first that tried to use the GPU as a pattern matching engine for NIDS in PixelSnort [18]. They used a simplified version of the Knuth-Morris-Pratt algorithm [19], however, their performance results indicated marginal improvement.

In this thesis, we explore how GPUs can be used to speed up the processing throughput of intrusion detection systems by offloading the string matching operations to the GPU. We show that single pattern matching algorithms, like Knuth-Morris-Pratt and Boyer-Moore [6], do not perform well when executed on the GPU, especially when using an increased number of patterns. However, porting multi-pattern matching algorithms, like the Aho-Corasick algorithm [1] can boost overall performance by a factor of three. Furthermore, we take advantage of DMA and the asynchronous execution of GPUs to impose concurrency between the operations handled by the CPU and the GPU. We have implemented a prototype intrusion detection system that effectively utilizes GPUs for pattern matching operations in real time.

1.1 Thesis organisation

The thesis is organized as follows: Background on pattern matching and an overview of the GPU architecture that we used for this research are presented in Section 2. Section 3 and 4 presents our prototype architecture and the implementation details respectively. In Section 5 we evaluate our implementation and we compare with the previous work. In Section 6 we present a brief survey of related work. Finally, the thesis ends with some conclusions as well as some ideas for future work in Section 7.

2

Background

In this section we briefly describe the architecture of NVIDIA graphics cards, which we have used for this work, as well as the programming capabilities they offer through the Compute Unified Device Architecture (CUDA) SDK. We also discuss some general disciplines about pattern matching algorithms and how they are adopted in network intrusion detection systems.

2.1 The GeForce 8 Series Architecture

The NVIDIA GeForce 8 Series (G80) architecture is based on a set of multiprocessors, each of which contains a set of *stream processors* operating on SPMD (Single Process, Multiple Data) programs. When programmed through CUDA, the GPU can be used as a general purpose processor, capable of executing a very high number of threads in parallel.

A unit of work issued by the host computer to the GPU is called a *kernel*, and is executed on the device as many different *threads* organized in *thread blocks*. Each multiprocessor executes one or more thread blocks, with each group organized into *warps*. A warp is a fraction of an *active group*, which is processed by one multiprocessor in one batch. Each of these warps contains the same number of threads, called the *warp size*, and is executed by the multiprocessor in a SPMD fashion. Active warps are time-sliced: A thread scheduler periodically switches from one warp to another to maximize the use of the multiprocessors' computational resources.

Stream processors within a processor share an instruction unit. Any control flow instruction that causes threads of the same warp to follow different execution paths reduces the instruction throughput, because different execution paths have to be serialized. When all the different execution paths have reached a common end, the threads converge back to the same execution path.

A fast *shared memory* is managed explicitly by the programmer among thread blocks. The *global*, *constant*, and *texture memory spaces* can be read from or written to by the host, are persistent across kernel launches by the same application, and are optimized for different memory usages [24]. The constant and texture memory accesses are cached, so a read from them costs much less compared to device memory reads, which are not being cached. The texture memory space is implemented as a read-only region of device memory.

GPGPU programming on G80 series and later is feasible using the CUDA SDK. CUDA consists of a minimal set of extensions to the C language and a runtime library that provides functions to control the GPU from the host, as well as device-specific functions and data types. CUDA exposes several hardware features that are not available via the graphics API. The most important of these features is the read and write access to the shared memory

shared among the threads, and the ability to access any memory location in the card's DRAM through the general memory addressing mode it provides. Finally, CUDA also offers highly optimized data transfers to and from the GPU.

2.2 Pattern Matching Algorithms

Pattern matching algorithms have been studied for several decades. Over the years, new algorithms have been developed to improve the performance of searching string literals in a stream of data. In general, pattern matching algorithms can be classified into single- and multi-pattern algorithms.

2.2.1 Single-Pattern Matching Algorithms

In single pattern matching algorithms, each pattern is searched in a given text individually. This means that if we have k patterns to be searched, the algorithm must be repeated k times. Knuth-Morris-Pratt [19] and Boyer-Moore [6] are some of the most widely used single pattern matching algorithms.

Knuth-Morris-Pratt [19] exploit the properties of the target pattern to reduce the required number of comparisons. The algorithm is similar to the brute-force algorithm except for the way mismatches are handled. In the brute-force algorithm, the pattern is always shifted right one position after a character mismatch, regardless of the number of character matches that occurred before the mismatch. This can lead to wasted comparisons at positions that cannot possibly result in a complete pattern match. The Knuth-Morris-Pratt algorithm eliminates these unnecessary comparisons by using a partial-match table for each pattern. Each table is built by preprocessing every pattern separately, and indicates how many positions the pattern may be shifted to the right based on the position in the pattern where a mismatch occurs.

Boyer-Moore [6] is a particularly efficient single-pattern algorithm. Its

general idea is to skip as many characters as possible without missing a possible instance of the string being searched for. It starts matching from the last character in the pattern so that, upon a mismatch, it skips an entire span of input. Its execution time can be sublinear if the suffix of the string to be searched for appears infrequently in the input stream. A modified version of the algorithm, called Boyer-Moore-Horspool [16], consumes less memory and suffers no significant speed loss.

2.2.2 Multi-Pattern Matching Algorithms

Multi-pattern string matching algorithms search for a set of patterns in a body of text simultaneously. This is achieved by preprocessing the set of patterns and building an automaton that will be used in the matching phase to scan the text. The automaton can be thought of as a state machine that is represented as a trie, a table or a combination of the two. Each character of the text will be searched only once. Multi-pattern matching scales much better than algorithms that search for each pattern individually. Multi-pattern string matching algorithms include Aho-Corasick [1], Wu-Manber [38] and Commentz-Walter [10].

The Aho-Corasick algorithm has the desirable property that the processing time does not depend on the size or number of patterns in a significant way. Given a set of patterns, the algorithm constructs a pattern matching machine, that is employed to match all patterns at once, one byte at a time. The processing action of the automaton is to start in an initial state, accept an input event, and move the current state to the next correct state based on the input event. There are three functions involved in the process: a *goto* function, a *failure* function, and an *output* function. Figure 2.1 shows an example of these functions for the set of patterns {he, she, his, hers}.

The *goto* function determines if a state transition can be performed based on the current state and the value of the input character. If the input matches one of the characters on an arc leaving the current state,

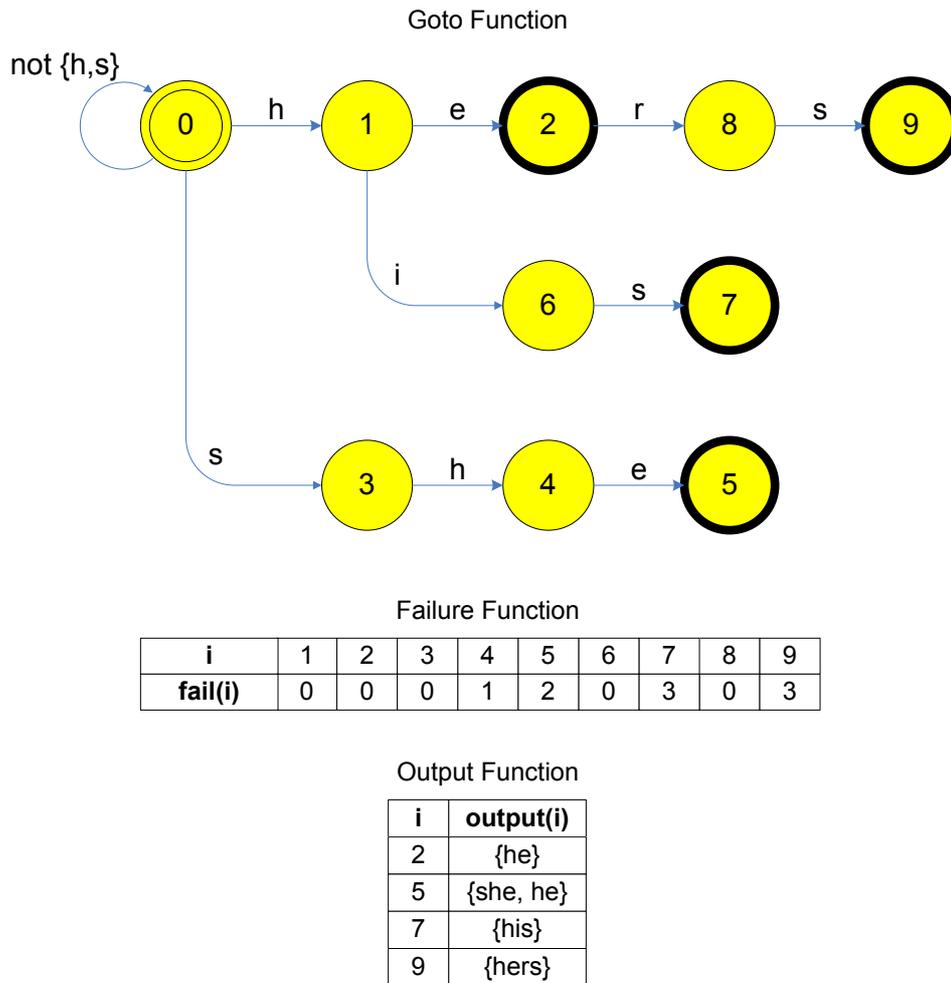


FIGURE 2.1: Aho-Corasick pattern matching machine example (adapted from [1]).

then the state pointed to by the arc becomes the next state. Otherwise, the next state is determined by the *failure* function. The failure function causes a transition to one or more intermediate states, or to the *initial* (or *start*) state that is represented with 0. The *output* function is consulted after each state transition to determine if a pattern has been matched. The process continues until the end of the input text is reached.

Although Aho-Corasick is deterministic by nature, its automaton is called *non-deterministic (NFA)* due to the fact that failure transitions do not consume any input (traditionally called ϵ -transitions). Moreover, the failure function may result to multiple state transitions for a single input symbol. Thus, the searching operation may need to explore multiple paths in the automaton to determine whether the input matches any signatures. Due to failure transitions, the performance of the algorithm can decrease quickly when the size of the patterns grows [37].

An improved version of the Aho-Corasick algorithm replaces all failure transitions with regular ones [1]. The resulting state machine, called *Deterministic Finite Automaton (DFA)*, will have exactly one transition per each state and input symbol, at the cost of extra memory. The matching procedure, is very efficient in terms of speed, since a sequence of n bytes can be matched with $O(n)$ operations.

2.3 Network Intrusion Detection Systems and Pattern Matching

Pattern matching is the most crucial component of a network intrusion detection system [2, 7]. Therefore, the algorithm used for pattern matching has a significant effect on the overall performance of the system.

Most Network Intrusion Detection Systems (NIDS) use finite automata to match patterns [17, 25, 27]. Coit *et al.* [9] improved the performance of Snort by combining the Aho-Corasick keyword trie with the skipping

feature of the Boyer-Moore algorithm. Fisk and Vaghese enhance the Boyer-Moore-Horspool algorithm to simultaneously match a set of rules. The new algorithm, called Set-wise Boyer-Moore-Horspool [15], was shown to be faster than both Aho-Corasick and Boyer-Moore for sets with less than 100 patterns. Tuck *et al.* [32] optimized the Aho-Corasick algorithm by applying bitmap node and path compression.

Snort from version 2.6 and onwards uses only flavors of the Aho-Corasick for exact-match pattern detection. Specifically, it contains a variety of implementations that are differentiated by the type of the finite automaton they use (NFA or DFA), and the storage format they use to keep it in memory (full, sparse, banded, trie, *etc.*). It should be mentioned, however, that the best performance is achieved with the full version that uses a deterministic finite automaton (DFA) at the cost of high memory utilization [31].

3

Architecture

The overall architecture of Gnort, which is based on the Snort NIDS, is shown in Figure 3.1. We can separate the architecture of our system in three different tasks: the transfer of the packets to the GPU, the pattern matching processing on the GPU, and finally the transfer of the results back to the CPU.

3.1 Transferring Packets to the GPU

The first thing to consider is how the packets will be transferred from the network interface to the memory space of the GPU. The simplest approach would be to transfer each packet directly to the GPU for processing. However, due to the overhead associated with a data transfer operation to the GPU, batching many small transfers into a larger one performs much better

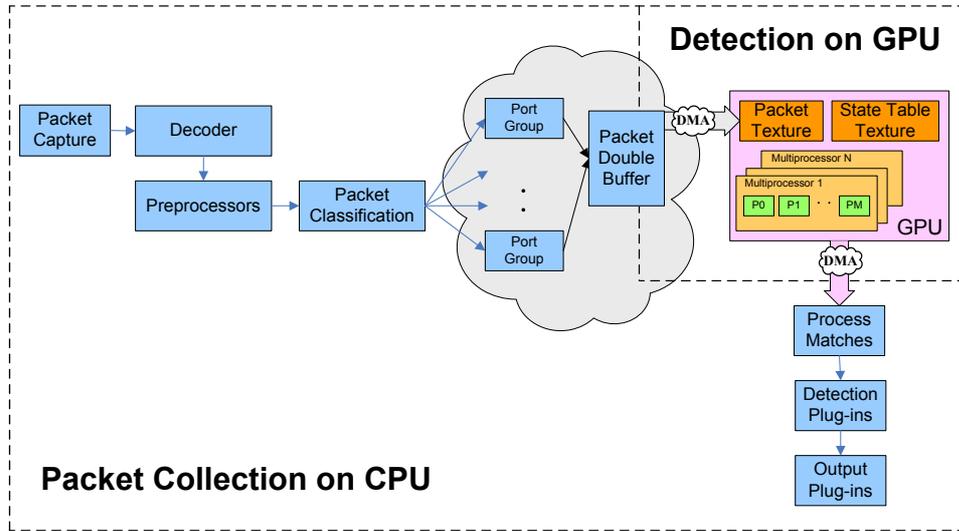


FIGURE 3.1: Overall architecture of Gsnort.

than making each transfer separately [24]. Thus, we have chosen to copy the packets to the GPU in batches.

Snort organizes the content signatures in groups, based on the source and destination port numbers of each rule. A separate detection engine instance is used to search for the patterns of a particular rule group. Table 3.1 shows the number of rules that come with the latest versions of Snort and are enabled by default, as well as the number of groups in which they are organized.

Therefore, we use a buffer for temporarily storing the packets. After a packet has been classified to a specific group, it is copied to the buffer. Since each packet may belong to a different group, we further “mark” each packet so it will be processed against the corresponding detection engine at the searching phase. Therefore, each row of the buffer contains an extra field that is used to store a pointer to the detection engine that the specified packet should be matched for. Whenever the buffer gets full, all packets are transferred to the GPU in one operation. In case the buffer is not yet

Snort version	# Groups	# Rules
2.6	249	7179
2.7	495	8719
2.8	495	8722

TABLE 3.1: Snort Data Structures.

full after 100ms, its packets are explicitly transferred to the GPU.

The buffer that is used to collect the network packets is allocated as a special type of memory, called page-locked or “pinned down” memory. Page-locked memory is a physical memory area that does not map to the virtual address space, and thus cannot be swapped out to secondary storage. The use of this memory results to higher data transfer throughput between the host and the device [24]. Furthermore, the copy from page-locked memory to the GPU can be performed using DMA, without occupying the CPU. Thus, the CPU can continue working and collecting the next batch of packets at the same time the GPU is processing the packets of the previous batch.

To further improve parallelism, we use a double buffering scheme. When the first buffer becomes full, it is copied to a texture bounded array that can be read later by the GPU through the kernel invocation. While the GPU is performing pattern matching on the packets of the first buffer, the CPU will copy newly arrived packets in the second buffer.

3.2 Pattern Matching on the GPU

Once the packets have been transferred to the GPU, the next step is to perform the pattern matching operation. We have ported the implementation of the Aho-Corasick algorithm that is based on a deterministic finite automaton (DFA) [1] to run on the graphics card. The DFA-based implementation seems to be a perfect candidate for SPMD processors like a GPU.

The algorithm iterates through all the bytes of the input stream and moves the current state to the next correct state using a state machine that has been previously constructed during initialization phase. The loop lacks any control flow instructions that would probably lead to thread divergence.

In our GPU implementation, the DFA of the state machine is stored as a two-dimensional array. The dimensions of the array are equal to the number of states and the size of the alphabet (256 in our case), respectively. Each cell contain the next state to move, as well as an indication whether the state is a final state or not. In case the state is final, the corresponding cell should also contain the unique identification number (*ID*) of the matching pattern, in order to report it. Although we can allocate more space per cell to store the *IDs* of the matching patterns, a more optimized solution, in terms of memory, is to store each final state as a negative equivalent value. Since transition numbers can be positive integers only, whenever the state machine reaches into a state that is represented by a negative number, it consider it as a final state and reports a match at the current input offset. Moreover, since each pattern that is recognized by a deterministic state machine will usually have a unique final state, we can use the value of the final state as the *ID* of the matching pattern. In case two or more patterns share the same final state (that is, they have the same suffix), an extra verification step has to be performed, that we will describe in Section 3.3.

A drawback of this structure is that state machine tables will be sparsely populated, containing a significant number of zero elements and only a few non-zero elements. However, the use of more efficient storage structures, like those proposed in [23], are much more complex to map in the memory space of a GPU.

During the initialization phase, the state machine table of each rule group is constructed in host memory by the CPU, and is then copied to texture memory that is accessible directly from the GPU. At the searching

phase, all state machine tables reside only in GPU memory. The use of GPU texture memory instead of generic GPU memory has the benefit that memory fetches are cached. A cache hit consumes only one cycle, instead of several hundreds in case of transfers from generic device memory. The usage of cache can exploit the strong locality of references that Aho-Corasick algorithm exhibits [12]. Furthermore, textures provide efficient random access, in contrast with global memory where memory accesses have to be aligned [24]. Using texture memory, a thread is pretty much free to read any area of one or more textures as many times as it needs to. In Section 5.2.1 we evaluate the performance benefits that texture memory can provide.

We have implemented two different parallelization methods for the Aho-Corasick searching phase. In the first, each packet is splitted into fixed equal parts and each thread searches each portion of the packet in parallel. In the second, each thread is assigned a whole packet to search in parallel. Both techniques have advantages and disadvantages that will be discussed in Section 4.

3.3 Transferring the Results to the CPU

Every time a thread matches a pattern inside a packet, it reports it by appending it in an array that has been previously allocated in the device memory. The reports for each packet will be written in a separate row of the array, following the order they were copied to the texture memory. That means that the array will have the same number of rows as the number of packets contained in the batch. Each report is constituted by the *ID* of the pattern that was matched and the index inside the packet where it was found.

After the pattern matching execution has finished, the array that contains the matching pairs is copied to the host memory. Before raising an

alert for each matching pair, the following extra cases should be examined in case they apply:

- *Case-sensitive patterns.* Since Aho-Corasick cannot distinguish between capital and low letters, an extra, case-sensitive, search should be made at the index where the pattern was found.
- *Offset-oriented rules.* Some patterns must be located in specific locations inside the payload of the packet, in order for the rule to be activated. For example, it is possible to look for a specified pattern within the first 5 bytes of the payload. Such ranges are specified in Snort with special keywords, like `offset`, `depth`, `distance`, etc. The index where the match was found is compared against the offset to argue if the match is valid or not.
- *Patterns with common suffix.* It is possible that if two patterns have the same suffix will also share the same final state in the state machine. Thus, for each pattern, we keep an extra list that contains the “suffix-related” *IDs* in the structure that holds its attributes. If this list is not empty for a matching pattern, the patterns that contained in the list have to be verified to find the actual matching pattern.

3.4 Execution flow overview

Figure 3.2 shows the pipelined execution of the overall system. Each task is represented by a different box that is executed either on the CPU or the GPU. The arrows show how the two devices communicate with each other by exchanging data: the CPU is responsible for providing network packets to the GPU, while the latter returns back the matching results after processing them.

The CPU is gathering packets from the network link, using the `pcap` [22] library. The decode stage is then validate the network packets, detect pro-

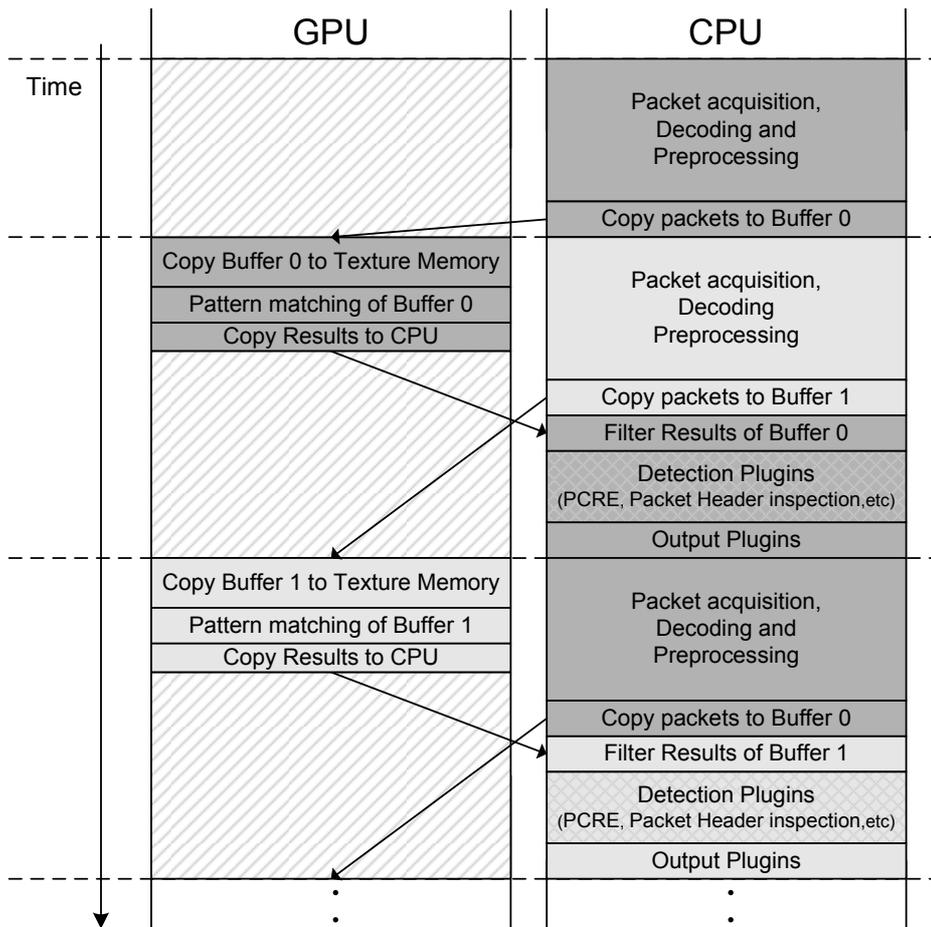


FIGURE 3.2: Execution flow of Gnort. GPU communication and computation may be hidden by overlapping it with CPU execution.

protocol anomalies and provide a referential data structure that contains the tightly-encoded protocol headers and associated information of each packet. Any configured preprocessors may then optionally invoked that use and manipulate packet data in various ways.

The payload of the network packets that need to be checked against a detection engine is then copied to the packet buffer. Everytime the buffer gets full, it is copied to the memory space of the GPU where it will be checked for attack signatures using the corresponding detection engines.

The pattern matching as well as the transfer of the packets to the GPU occur in an asynchronous fashion. This feature allows meaningful work to be done concurrently on the CPU during GPU execution. By using the double-buffering scheme that we described in Section 3.1, the CPU is kept busy while the GPU is matching network packets. When the first buffer becomes full, it is transferred and processed to the GPU. At the meantime, the CPU will collect newly arrived packets in the second buffer, as shown in Figure 3.2.

In case the second buffer is full while the GPU is still processing the first buffer, the CPU is busy-waiting until GPU finishes. As we will see in Section 5.2 though, the computational throughput of the GPU can be increased by providing a sufficient number of network packets for processing each time. Thus, by using a reasonable sized packet buffer, both GPU computation and communication costs can be completely hidden by the overlapped CPU execution.

Network packets may match one or more patterns in the multi-pattern matching stage. All matches are copied back to the CPU, where all the remainder conditions of the rules are checked, including offset-oriented conditions, non-content conditions and regular expressions. Because each match from the multi-pattern algorithm may or may not result in a match for its rule as a whole, this stage may be thought of as a "verification" stage.

Finally, in the last stage, the system owner is notified through alerts that have triggered related to the specific rule matches.

4

Implementation

We have implemented Gnort on the GeForce 8 Series architecture using the CUDA SDK. It is stated by NVIDIA though, that programs developed for the GeForce 8 series will also work without modification on all future NVIDIA video cards.

To facilitate concurrent execution between the host and the device, we associate GPU execution into streams. A *stream* is a sequence of operations that execute in order. It is created by the host and in our case includes the copying of the packets to the device memory, the kernel launch, and the transfer of the results back to the host memory. While the stream is executing, the CPU is able to collect the next batch of packets. The CPU work includes the execution flow of Snort to capture, decode, and classify

the incoming packets, as well as the extra packet copies to the page-locked memory buffers that we have introduced.

The page-locked memory buffers that are used to collect the packets in batches are allocated by the CUDA runtime driver. The driver tracks the relevant virtual memory ranges and automatically accelerates calls to functions that are used to copy data to the device. The copying of the buffers to the device is asynchronous and is associated to the stream. The device memory where the packets are copied is bound to a texture reference of type `unsigned char` and dimensionality 2. Texture fetches are cached using a proprietary 2D caching scheme and cost only one clock cycle when a cache hit occurs; otherwise a fetch can take 400 to 600 clock cycles. Only the packet payloads are copied to the device, and each payload is stored in a separate row of fixed size. The actual length of the payload is stored in the first two bytes of the row.

A special case that need to be taken care, however, is that modern NIDSes usually aggregate distinct packets into TCP streams to prevent an attacker from disguising malicious communications by breaking the data up across several packets. In Snort, the **stream4** preprocessor takes multiple packets from a given direction of a TCP flow and builds a single conceptual packet by concatenating their payloads, allowing rules to match patterns that span packet boundaries. It accomplishes this by keeping a descriptor for each active TCP session and tracking the state of the session. It also keeps copies of the packet data and periodically "flushes" the stream by reassembling the contents and passing a *pseudo*-packet containing the reassembled data to the detection engine.

Consequently, the size of the pseudo-packet that is created by the stream4 plugin may be up to 65,535 bytes, which is the maximum IP packet length. However, by assigning the maximum IP packet length as the size of each row of the buffer, will result in a huge array that will be sparsely

populated, since reassembled packets may have much less size. Moreover, the copy of the whole array to the device will result in high communication costs that will limit down the overall performance.

A different approach to store reassembled packets that exceed the Maximum Transmission Unit (MTU) size, is to split them down into several smaller. The size of each portion of the splitted packet will be less or equal to the MTU size and thus can be copied at consecutive rows in the array.

The portions of the splitted packet will be processed by different threads. Nevertheless, in order to avoid missing matches that may span between the portions of a splitted packet during search, whenever a thread searches a splitted portion of a packet, it continues the searching to the following row (that will contain the consecutive bytes of the packet), up to X bytes, where X is the maximum pattern length in the state table.

The state machine tables that are used for each group of rules are stored in a texture reference of type `short`. However, CUDA does not support dynamic allocation of textures yet. To overcome this limitation, we pre-allocate a large amount of linear memory that is statically bound to a texture reference. All state table arrays are stored sequentially in this texture memory segment.

Once the packets have been copied to the texture bound array, the kernel is initiated by the host to perform the pattern matching. The 8-Series (G8X)—as well as the 9-Series (G9X) which was recently released—contain many independent multiprocessors, each comprising eight stream processors that run on a SPMD fashion. However, every multiprocessor has an independent instruction decoder, so they can run different instructions.

The Aho-Corasick algorithm performs multi-pattern search, which means that all patterns of a group are searched simultaneously. We have explored two different approaches for parallelizing the searching phase by splitting the computation in two ways: assigning a single packet to each multipro-

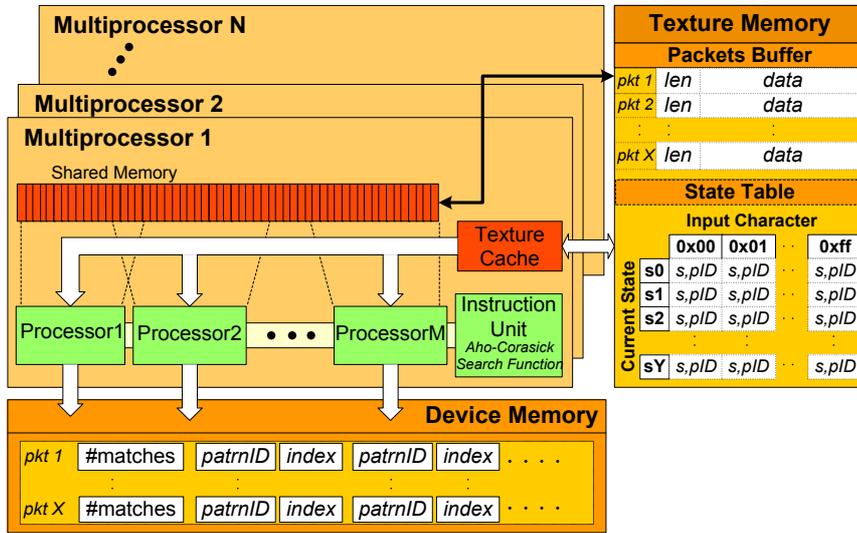
cessor at a time, and assigning a single packet to each stream processor at a time. The two approaches are illustrated in Figure 4.1.

4.1 Assigning a Single Packet to each Multiprocessor

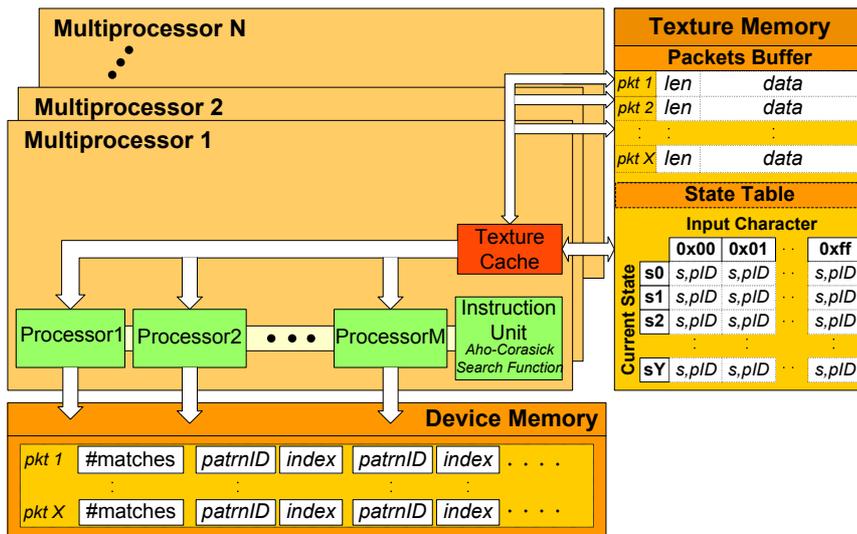
In this approach, each packet is processed by a specific thread block, executed by one multiprocessor. The number of threads in the thread block that search the packet payload is fixed and equal to the warp size (currently 32). Through experimentation we found that 32 threads provide the maximum performance. Even though each multiprocessor consists of eight stream processors, threads within a block are executed in groups called warps. A warp is a maximal subset of threads that execute the same instruction at the same time.

Each thread searches a different part of the packet, and thus the packet is divided in 32 equal chunks. The 32 chunks of the packet are processed by the 32 threads of the warp in parallel. To correctly handle matching patterns that span consecutive chunks of the packet, each thread searches in addition X bytes after the chunk it was assigned to search, where X is the maximum pattern length in the state table. To reduce further communication costs due to the overlapping computations, each packet is also copied to the shared memory of the multiprocessor (besides the texture memory)—all threads copy a different chunk in parallel, so this additional copy does not add significant overhead.

An advantage of this method is that all threads may assigned the same amount of work, so execution does not diverge, which would hinder the SPMD execution. A drawback of this approach is that extra processing is needed for the chunk overlaps.



(a) Packet per multiprocessor.



(b) Packet per stream processor.

FIGURE 4.1: Different pattern matching parallelization approaches. In (a), a different packet is processed by each multiprocessor. All stream processors in the multiprocessor search the packet payload concurrently. In (b), a different packet is processed by each stream processor independently of the others.

4.2 Assigning a Single Packet to each Stream Processor

In this approach, each packet is processed by a different thread. The number of thread blocks that are created is equal to the number of multiprocessors the GPU has, so all are working. Each thread block processes X/N packets using an equal number of threads, where X is the number of packets in the batch sent to the GPU, and N is the number of multiprocessors. However, the maximum number of threads that can be created per block is currently 512. So if the number of threads per thread block is greater, more thread blocks are created to keep the number of threads under this limit.

The disadvantage of this method is that the amount of work per thread will not be the same since packet sizes will vary. This means that threads of a warp will have to wait until all have finished searching the packet that was assigned to them, in order to be synchronized. However, by creating a large number of threads that execute concurrently by the same multiprocessor, the thread scheduler can pipeline threads execution to effectively utilize available resources, as we will see in Section 5.2. Furthermore, no additional computation will occur since every packet will be processed in isolation.

Whenever a match occurs, regardless of the implementation used, the corresponding *ID* of the pattern and the index where the match was found are stored in an array allocated in device memory. Each row of the array contain the matches that were found per packet. We use the first position of each row as a counter to know where to put the next match. Every time a match occurs, the corresponding thread increments the counter and writes the report where the counter points to. The increment is performed using an atomic function supplied by CUDA, to overcome possible race conditions

4.2. ASSIGNING A SINGLE PACKET TO EACH STREAM PROCESSOR 29

for the first parallelization method.

5

Evaluation

In this section, we explore the performance of our implementation. First, we measure the scalability of single- and multi-pattern algorithms for different number of patterns and packet sizes, and how they affect overall performance. We then examine how our prototype implementation performs in a realistic scenario as a function of the traffic load.

In our experiments we used a NVIDIA GeForce 8600GT card, which contains 32 stream processors organized in 4 multiprocessors, operating at 1.2GHz with 512 MB of memory. The CPU in our system was a 3.40 GHz Intel Pentium 4 processor with 2 GB of memory.

For all experiments conducted, we disregard the time spent in the initialization phase of Snort as well as the logging of the alerts to the disk.

Even though it takes less than just a few seconds to load the patterns and build its internal structures in all cases, there is no practical need to include this time in our graphs. For all experiments we measure the performance of the default Snort using the full DFA implementation of Aho-Corasick. We conducted experiments with other implementations as well, however they performed worse in every case. Some information on the different implementations of Aho-Corasick that Snort uses can be found in [31].

5.1 Performance of single- and multi-pattern algorithms

In order to directly compare with prior work, we re-implemented the Knuth-Morris-Pratt (KMP) algorithm on the NVIDIA G80 GPU architecture using the CUDA SDK. In our implementation, the patterns to be searched, and the partial-match tables that KMP uses, are stored in two 2D texture arrays. Each packet is assigned to a different thread block, executed by a different multiprocessor, while each thread in a block is responsible for searching a specific pattern in the entire packet. This way, each warp of threads performs pattern matching against each packet in parallel, as long as the number of patterns is equal with the number of processors. If the number of patterns is greater than 512, the pattern matching is bundled in groups of 512 patterns each time, due to the limitation of the 512 threads that can be created per block.

We also did a GPU implementation of the Boyer-Moore (BM) algorithm, which performs better than KMP. The patterns to be searched, as well as the bad-character shift tables, are stored in two 2D texture arrays similarly to the KMP implementation. Each packet is assigned to a different thread block, while each thread in a block is responsible for searching a specific pattern in the whole packet.

In the next experiment we evaluated how each detection algorithm scales

5.1. PERFORMANCE OF SINGLE- AND MULTI-PATTERN ALGORITHMS³³

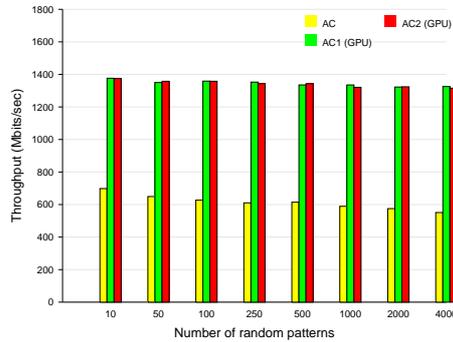
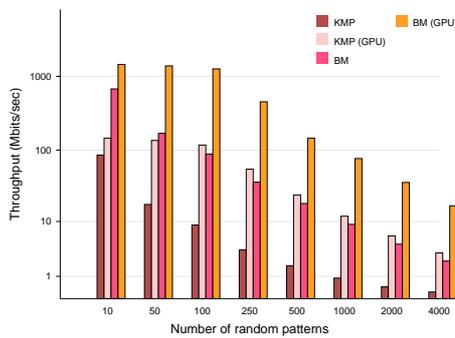


FIGURE 5.1: Throughput sustained for single-pattern matching algorithms. FIGURE 5.2: Throughput sustained for multiple-pattern matching algorithms.

with the number of patterns. We created Snort rules of randomly generated patterns which size varied between 5 and 25 bytes and gave as input to Snort a payload trace that contains UDP packets with random payload, each of 800 bytes in length. All rules are matched against every packet. This is the worst case scenario for a pattern matching engine, as in most cases each packet has to be checked only against a few hundred rules.

Figures 5.1 and 5.2 show the maximum throughput achieved for single- and multi-pattern matching algorithms respectively, to perform string searches through rule-sets of sizes 10 up to 4000 rules. As shown in Figure 5.1, single pattern algorithms do not scale as the rule-set size increases. Performance of the CPU implementations of both KMP and BM decreases linearly with the number of patterns. KMP achieves nearly 100 Mbit/s for 10 patterns but its performance for 4000 patterns drops under 1 Mbit/s. BM presents better results but still for a large number of patterns it can only achieve up to 5 Mbit/s. The GPU implementation of these algorithms boosts their performance by up to an order of magnitude. For the case of 50, 100 and 250 patterns we can see that GPU versions of algorithms are an order of magnitude faster than the CPU ones, while for the case of 4000 patterns

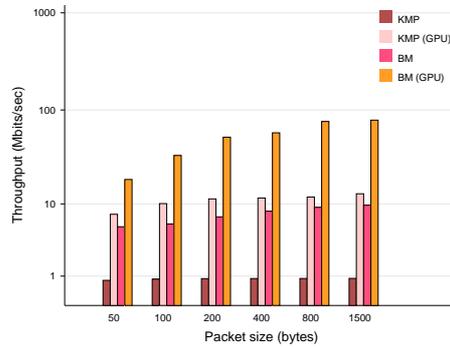


FIGURE 5.3: Throughput sustained for single-pattern matching algorithms for various packet sizes.

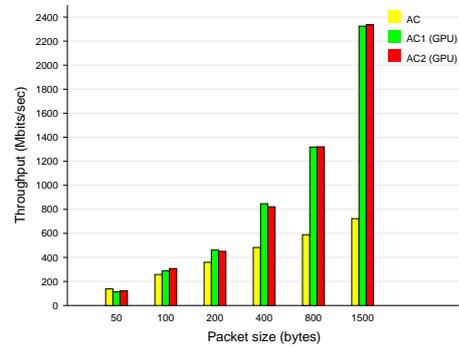


FIGURE 5.4: Throughput sustained for multiple-pattern matching algorithms for various packet sizes.

the improvements reaches a factor of 3.

An interesting observation is that the throughput of the GPU implementations for both KMP and BM remained constant for up to 100 patterns. Even though there are 32 processors available, the thread scheduler can pipeline threads execution to effectively utilizes available resources. To verify it, we changed the kernels to return immediately performing a null computation and we observed the same behavior. Performance of the system remained constant for up to 100 patterns and then began decreasing linearly.

In the case of Aho-Corasick algorithm, the throughput remains constant independently of the number of patterns, a behavior expected for a multi-pattern approach. The results are shown in Figure 5.2. For the CPU implementation, Aho-Corasick achieves nearly 600 Mbit/s throughput, while the GPU implementation reaches up to 1.4 Gbit/s, yielding a 2.4 times improvement. Our two different approaches for implementing Aho-Corasick (displayed as AC1 and AC2 in the graph) do not present significant differences in performance. This can be explained by the fact that in both implementations, GPU computation and communication costs are

completely hidden by the overlapped CPU computation, which is constant for both cases, as we will show in the next section.

Figures 5.3 and 5.4 show the throughput achieved for various UDP packet sizes. Snort was loaded with 1000 random patterns which size varied between 5 and 25 bytes. Each packet contains random data, a property that favors the BM algorithm as it will skip most of the payload. CPU implementations of KMP and BM presented a stable performance of around 1 and 10 Mbit/s respectively, independently of the packet size. Their GPU implementations yield a speedup from 2 up to 10 times. The throughput of Aho-Corasick reached over 2.3 Gbit/s for 1500-byte packets, giving a total speedup of 3.2 compared to the respective CPU implementation. It is important to notice that it is worthless to process small packets on GPU. As it can be seen in Figure 5.4, for small packet sizes (under 100), the CPU implementation performs better than the GPU. However, for sizes larger than 100 bytes, the GPU implementation outperforms the CPU one in all cases.

5.2 Experiments with Real Input and Rules

In this section we present the evaluation of our prototype implementation using real rules from the current Snort rule set on real network traffic.

For all experiments conducted, we used a full payload trace captured at the access link that connects an educational network with thousands of hosts to the Internet. Snort was run with a custom configuration in which preprocessors and regular expression matching were disabled, as both processes are executed only on the CPU. Snort loaded 5467 rules that contain about 7878 content patterns.

5.2.1 Microbenchmarks

In the next experiment, we examine the effect that the size of the buffer has on the overall application performance. We vary the size of the buffer from

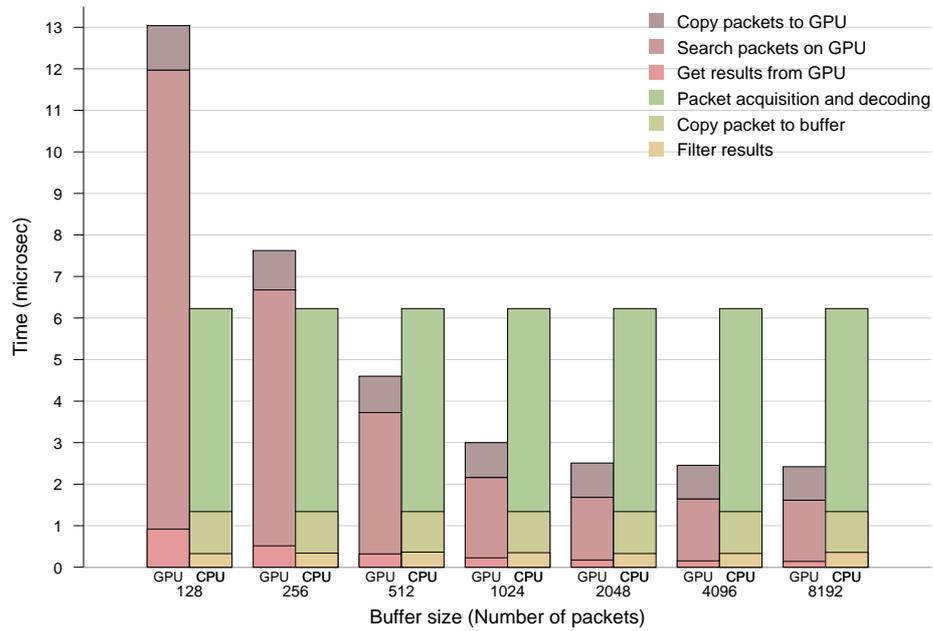


FIGURE 5.5: Average execution times per inspected packet spent on each task.

128 to 8192 packets, while Snort was reading the trace file from the local machine. The whole network trace was saved in cache memory after the first reading, thus no blocks were read from disk when gathering packets. We have verified the absence of I/O latencies using the `iostat(1)` tool.

Figure 5.5 shows the individual execution times per packet for each of the tasks that are executed on the CPU and the GPU. We display the times of each device in different bars, since execution is performed in parallel. Data are transferred to and from the GPU with DMA commands, hence the CPU can continue unaffected while DMA transfers are in progress. We can notice that transferring data to and from the GPU performs better when batching many data together. Increasing the buffer size from 128 to 1024 packets, gave a 28% improvement in the transfer throughput of the network packets to the GPU. We also observe that when the buffer size is larger than

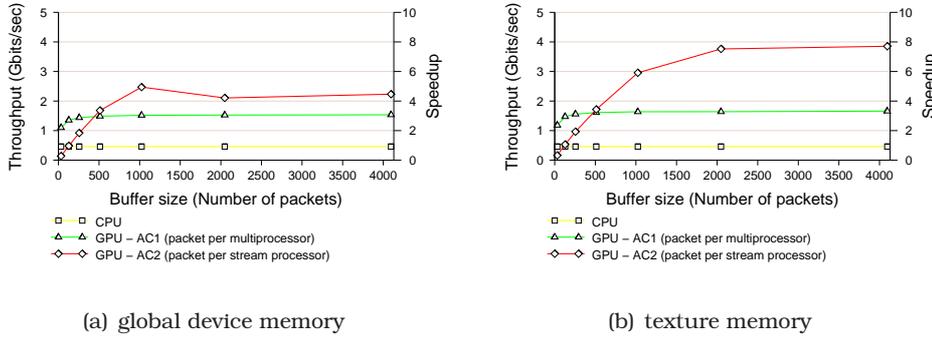


FIGURE 5.6: Packet matching computational throughput for CPU and GPU. In (a) the state machine tables are stored in Global device memory, while in (b) in Texture memory.

512 packets, the costs for the data transfers and the pattern matching on the GPU are completely hidden by the overlapped CPU workload.

The CPU workload includes the copy of the packets from the kernel and the decoding into its internal data structures, as well as the extra copy we introduced in order to transfer the packets to the GPU in batches.

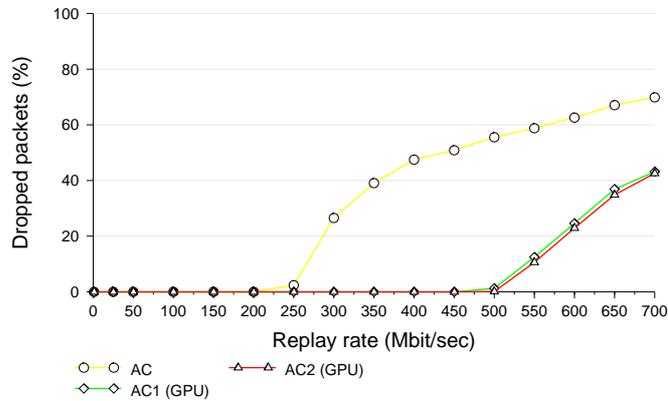
Regarding the pattern matching execution on the GPU, we have used the implementation that assigns each packet to a different stream processor (AC2). We also repeated the experiment using the implementation that assigns each packet to a multiprocessor (AC1). The AC1 was able to hide both computation and communication when buffer size was down to 128 packets, however performed worst than AC2 when packet buffer was larger than 1024.

In Figure 5.6 we plot the raw computational capacity, measured as the mean size of data processed per second, for both AC1 and AC2 implementations as well as for the CPU. We also explore the performance that different types of memory can provide, by using global and texture memory equivalently to store the state machine tables. On the x-axis we show the number of packets that are processed at once by the GPU. The cost to deliver the

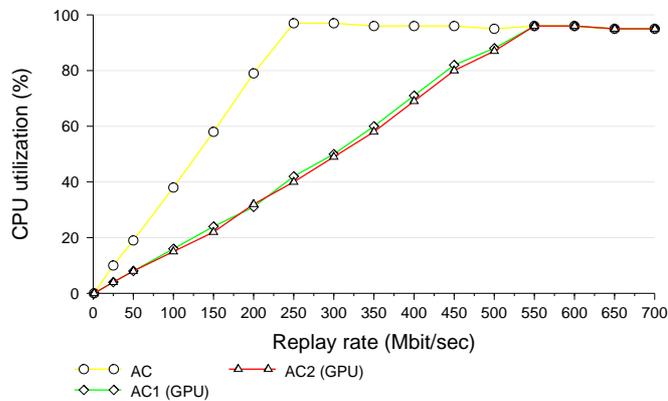
packets from the network trace to the memory space of each device is not included.

When using global device memory, both implementations exposed about a 3x-4x speed increase over the CPU, with AC2 be better than AC1 about 25% for large buffer sizes. The usage of texture memory, however, seems to improve only the execution of AC2. The AC2 achieved an improvement of 8.39 times against the CPU implementation, when using a 2048 packet buffer and the state machines were stored in texture memory. However, increasing the size from 2048 to 4096 packets gave only a slight improvement. AC1 had a constant performance, up to 3.6 times faster than the CPU that was slightly affected by the buffer size. Texture memory gave AC1 a marginal improvement, about 8%, in the kernel execution.

Comparing our two different implementations, AC1 performed better than AC2 for buffer sizes up to 512 packets. When the number of packets is small, it appears to be better to process each packet to a different multiprocessor. When the number of packets increases though, more threads are created per block for the AC2 approach. It is likely that threads of the same block, executed by the same multiprocessor, may be pipelined to effectively utilize available resources and hide memory latency. The usage of texture memory appears to maximize significantly the memory usage of AC2 through the texture cache. On the contrary, AC1 appears to fail to scale as the number of data increases, due to the small number of threads per block that provides, rather than the memory bandwidth. To verify this, we created a simple state machine table with only one state that remained permanently in the texture cache; the performance measured was affected marginally by the absence of memory transfers. It seems that a multiprocessor cannot execute concurrently the many thread blocks that AC1 creates, in order to hide latencies. Thus, it is better to increase the number of threads per block, rather than creating many partially-populated thread blocks.



(a) packet loss ratio



(b) cpu utilization

FIGURE 5.7: Overall performance as a function of the traffic speed.

5.2.2 Macrobenchmarks

In this section we present the evaluation of our prototype implementation on a real network environment that consists of two PCs connected via a 1 Gbit/s Ethernet switch. The first PC is equipped with the NVIDIA GeForce 8600GT card and runs our modified version of Snort, while the second is used for replaying real network traffic traces using tcpreplay [33]. By rewriting the source and destination MAC addresses in all packets of the network trace, the generated traffic can be sent to the first PC.

Figure 5.7(a) shows the packet loss ratio while replaying the trace at

different speeds. We can see that conventional Snort cannot process all packets in rates higher than 250 Mbit/s, so a significant percentage of packets is being lost. On the other hand, our GPU-assisted Snort is twice as fast as the original one. Packet loss for our approach starts at 500 Mbit/s, a 200% improvement to the processing capacity of Snort. The two different GPU implementations of the Aho-Corasick algorithm achieved almost the same performance as was expected, since the CPU was stalling the overall execution in both cases. For completeness, in Figure 5.7(b), we plot the corresponding CPU utilization. Packet loss starts when CPU reaches 100% utilization.

6

Related Work

To speedup the inspection process, many network intrusion detection implementations are based on specialized hardware.

By using content addressable memory (CAM), which is suitable to perform parallel comparison for its contents against the input value, they are very well suited for use in intrusion detection systems [39, 40]. However they have a high cost per bit.

Many reconfigurable architectures have been implemented for intrusion detection. Most approaches involve building an automaton for a string to be searched, generating a specialized hardware circuit using gates and flip-flops for the automaton, and then instantiating multiple such automata in the reconfigurable chip to search the streaming data in parallel. However,

the circuit implemented on the FPGA to perform the string matching is designed based on the underlying hardware architecture to adjust to a given specific rule set. To adjust to a new rule set, one must program a new circuit (usually in a hardware description language), which is then compiled down through the use of CAD tools. Any changes in the rule set requires the re-compilation, regeneration of the automaton, resynthesis, replacement and routing of the circuits which is a time consuming and difficult procedure.

Sidhu and Prasanna implemented a regular expression matching architecture for FPGAs [29]. Baker *et al.* also investigated efficient pattern matching as a signature based method [4]. In [13], the authors used hardware bloom filters to match multiple patterns against network packets at constant time. Attig *et al.* proposed a framework for packet header processing in combination with payload content scanning on FPGAs [3].

Several approaches attempt to reduce the amount of memory required to economically fit it in on-chip memory [4, 14, 32]. However, the on-chip hardware resource consumption grows linearly with the number of characters to be searched. In [30], the authors convert a string set into many tiny state machines, each of which searches for a portion of the strings and a portion of the bits of each string.

Other approaches involve the cooperation with network processors in order to pipeline the processing stages assigned to each hardware resource [8], as well as the entire implementation of an IDS on a network processor [5, 12]. Computer clusters have also been proposed to offload the workload of a single computer [20, 28, 34, 36]. The cost however remains high, since it requires multiple processors, a distribution network, and a clustered management system.

On the contrary, modern GPUs have low design cost while their increased programmability makes them more flexible than ASICs. Most graphic cards manufacturers provide high-level APIs that offer high programming

capabilities and are further ensure forward compatibility for future releases, in contrast with most FPGA implementations that are based on the underlying hardware architecture and need to be reconfigured whenever a change occurs in the rule set. Furthermore, their low design cost, the highly parallel computation and the potential that are usually underutilized, especially in hosts used for intrusion detection purposes, makes them suitable for use as an extra low-cost coprocessor for time-consuming problems, like pattern matching.

The work most related to ours is PixelSnort [18]. It is a port of the Snort IDS that offloads packet matching to an NVIDIA 6800GT. The GPU programming was complicated, since the 6800GT did not support a general purpose programming model for GPUs (as the G80 used in our work). The system encodes Snort rules and packets to textures and performs the string searching using the Knuth-Morris-Pratt algorithm on the 16 fragment shaders in parallel. However, PixelSnort *does not* achieve *any* speedup under normal-load conditions. Furthermore, PixelSnort did not have any multi-pattern matching algorithms ported to GPU. This is a serious limitation since multi-pattern matching algorithms are the default for Snort. In a more recent work, Marziale *et al.* [21] evaluated the effectiveness of offloading the processing of a file carving tool to the GPU. The system was implemented on the G80 architecture and the results show that GPU support can substantially increase the performance of digital forensics software that relies on binary string search.

7

Conclusions

In this thesis, we presented Gnort, a network intrusion detection system that utilizes the GPU to offload pattern matching computation in real time. We ported the classic Aho-Corasick algorithm to run on the GPU exploiting the SPMD parallel model. Our prototype system was able to achieve a maximum throughput of 2.3 Gbit/s, while in a real world scenario outperformed conventional Snort by a factor of two.

As future work we plan on eliminating the extra copy we introduced in order to transfer the packets to the GPU in batches. One way to accomplish this, is to transfer the packets directly from the kernel buffer. This would require that the buffer will be allocated from the application and will be shared between the user and kernel spaces. We believe that by modifying

the pcap-mmap, that already implements this shared buffer capability, we can benefit from the lack of copies of both from kernel to user space as well as the one to our defined buffer. An even more efficient way would be to DMA directly the packets from the NIC to the GPU, without occupying the CPU at all. Currently, this is not supported but it may be in the future.

Finally, we plan on utilizing multiple GPUs instead of a single one. Modern motherboards support dual GPUs, and there are PCI Express backplanes that support multiple slots. We believe that building such “clusters” of GPUs will be able to support multiple Gigabit per second Intrusion Detection Systems.

Bibliography

- [1] A. V. Aho and M. J. Corasick. Efficient string matching: an aid to bibliographic search. *Communications of the ACM*, 18(6):333–340, June 1975.
- [2] S. Antonatos, K. Anagnostakis, and E. Markatos. Generating realistic workloads for network intrusion detection systems. In *Proceedings of the 4th ACM Workshop on Software and Performance*, January 2004.
- [3] M. Attig and J. Lockwood. A framework for rule processing in reconfigurable network systems. In *Proceedings of the 13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '05)*, pages 225–234, Washington, DC, USA, 2005. IEEE Computer Society.
- [4] Z. K. Baker and V. K. Prasanna. Time and area efficient pattern matching on FPGAs. In *Proceedings of the 2004 ACM/SIGDA 12th International Symposium on Field Programmable Gate Arrays (FPGA '04)*, pages 223–232, New York, NY, USA, 2004. ACM.
- [5] H. Bos and K. Huang. Towards software-based signature detection for intrusion prevention on the network card. In *Proceedings of 8th International Symposium on Recent Advances in Intrusion Detection (RAID)*, Seattle, WA, September 2005.
- [6] R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Communications of the Association for Computing Machinery*, 20(10):762–772, October 1977.
- [7] J. B. D. Cabrera, J. Gosar, W. Lee, and R. K. Mehra. On the statistical distribution of processing times in network intrusion detection. In *43rd IEEE Conference on Decision and Control*, pages 75–80, December 2004.

- [8] C. Clark, W. Lee, D. Schimmel, D. Contis, M. Kone, and A. Thomas. A hardware platform for network intrusion detection and prevention. In *Proceedings of the 3rd Workshop on Network Processors and Applications (NP3)*, 2004.
- [9] C. Coit, S. Staniford, and J. McAlerney. Towards faster string matching for intrusion detection or exceeding the speed of Snort. In *Proceedings of DARPA Information Survivability Conference & Exposition II (DISCEX '01)*, June 2001.
- [10] B. Commentz-Walter. A string matching algorithm fast on the average. In *Proceedings of the 6th International Colloquium on Automata, Languages and Programming*, pages 118–131.
- [11] D. L. Cook, J. Ioannidis, A. D. Keromytis, and J. Luck. Cryptographics: Secret key cryptography using graphics cards. In *Proceedings of RSA Conference, Cryptographer's Track (CT-RSA)*, pages 334–350, 2005.
- [12] W. de Bruijn, A. Slowinska, K. van Reeuwijk, T. Hruby, L. Xu, and H. Bos. SafeCard: a Gigabit IPS on the network card. In *Proceedings of 9th International Symposium on Recent Advances in Intrusion Detection (RAID)*, Hamburg, Germany, September 2006.
- [13] S. Dharmapurikar, P. Krishnamurthy, T. S. Sproull, and J. W. Lockwood. Deep packet inspection using parallel bloom filters. *IEEE Micro*, 24(1):52–61, 2004.
- [14] S. Dharmapurikar and J. Lockwood. Fast and scalable pattern matching for content filtering. In *Proceedings of the 2005 ACM symposium on Architecture for networking and communications systems (ANCS '05)*, pages 183–192, New York, NY, USA, 2005. ACM.
- [15] M. Fisk and G. Varghese. Applying fast string matching to intrusion detection. Technical Report In preparation, successor to UCSD TR CS2001-0670, University of California, San Diego, 2002.
- [16] R. Horspool. Practical fast searching in strings. *Software - Practice and Experience*, pages 501–506, 1980.
- [17] C. IOS. IPS deployment guide. <http://www.cisco.com>.

- [18] N. Jacob and C. Brodley. Offloading IDS computation to the GPU. In *Proceedings of the 22nd Annual Computer Security Applications Conference on Annual Computer Security Applications Conference (ACSAC '06)*, pages 371–380, Washington, DC, USA, 2006. IEEE Computer Society.
- [19] D. E. Knuth, J. Morris, and V. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2):127–146, 1977.
- [20] C. Kruegel, F. Valeur, G. Vigna, and R. Kemmerer. Stateful intrusion detection for high-speed networks. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 285–294, May 2002.
- [21] G. G. R. I. Lodovico Marziale and V. Rousev. Massive threading: Using GPUs to increase the performance of digital forensics tools. *Digital Investigation*, 1:73–81, September 2007.
- [22] S. McCanne, C. Leres, and V. Jacobson. libpcap. Lawrence Berkeley Laboratory, Berkeley, CA. (software available from <http://www.tcpdump.org/>).
- [23] M. Norton. Optimizing pattern matching for intrusion detection. <http://docs.idsresearch.org/OptimizingPatternMatchingForIDS.pdf>, July 2004.
- [24] NVIDIA. NVIDIA CUDA Compute Unified Device Architecture Programming Guide, version 1.1. http://developer.download.nvidia.com/compute/cuda/1_1/NVIDIA_CUDA_Programming_Guide_1.1.pdf.
- [25] V. Paxson. Bro: A system for detecting network intruders in real-time. In *Proceedings of the 7th conference on USENIX Security Symposium (SSYM '98)*, pages 3–3, Berkeley, CA, USA, 1998. USENIX Association.
- [26] V. Paxson, R. Sommer, and N. Weaver. An architecture for exploiting multi-core processors to parallelize network intrusion prevention. In *Proceedings of the IEEE Sarnoff Symposium*, May 2007.
- [27] M. Roesch. Snort: Lightweight intrusion detection for networks. In *Proceedings of the 1999 USENIX LISA Systems Administration Conference*, November 1999.

- [28] L. Schaelicke, K. Wheeler, and C. Freeland. SPANIDS: a scalable network intrusion detection loadbalancer. In *CF '05: Proceedings of the 2nd conference on Computing frontiers*, pages 315–322, New York, NY, USA, 2005. ACM.
- [29] R. Sidhu and V. Prasanna. Fast regular expression matching using FPGAs. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM01)*, 2001.
- [30] L. Tan, B. Brotherton, and T. Sherwood. Bit-split string-matching engines for intrusion detection and prevention. *ACM Transactions on Architecture and Code Optimization*, 3(1):3–34, 2006.
- [31] The Snort Project. Snort users manual 2.8.0. http://www.snort.org/docs/snort_manual/2.8.0/snort_manual.pdf.
- [32] N. Tuck, T. Sherwood, B. Calder, and G. Varghese. Deterministic memory-efficient string matching algorithms for intrusion detection. In *Proceedings of the IEEE Infocom Conference*, pages 333–340, 2004.
- [33] A. Turner. Tcpreplay. <http://tcpreplay.synfin.net/trac/>.
- [34] M. Vallentin, R. Sommer, J. Lee, C. Leres, V. Paxson, and B. Tierney. The NIDS cluster: Scalable, stateful network intrusion detection on commodity hardware. In *Proceedings of the 10th International Symposium on Recent Advances in Intrusion Detection (RAID)*, pages 107–126, 2007.
- [35] G. Vasiliadis, S. Antonatos, M. Polychronakis, E. P. Markatos, and S. Ioannidis. Gnort: High performance network intrusion detection using graphics processors. In *Proceedings of 11th International Symposium on Recent Advances in Intrusion Detection (RAID)*, pages 116–134, 2008.
- [36] K. Watanabe, N. Tsuruoka, and R. Himeno. Performance of network intrusion detection cluster system. In *Proceedings of The 5th International Symposium on High Performance Computing (ISHPC-V)*, 2003.
- [37] B. W. Watson. The performance of single-keyword and multiple-keyword pattern matching algorithms. Technical report, 1994.
- [38] S. Wu and U. Manber. A fast algorithm for multi-pattern searching. Technical Report TR-94-17, 1994.

- [39] F. Yu, R. H. Katz, and T. V. Lakshman. Gigabit Rate Packet Pattern-Matching Using TCAM. In *Proceedings of the 12th IEEE International Conference on Network Protocols (ICNP '04)*, pages 174–183, Washington, DC, USA, October 2004. IEEE Computer Society.
- [40] S. Yusuf and W. Luk. Bitwise optimised CAM for network intrusion detection systems. In *Proceedings of International Conference on Field Programmable Logic and Applications*, pages 444–449, 2005.